Instytut Podstawowych Problemów Techniki
Polska Akademia Nauk

Rozprawa doktorska

Wydajne metody generowania bezpiecznych parametrów algorytmów klucza publicznego

Andrzej Chmielowiec

Rozprawa doktorska wykonana pod kierunkiem
dr hab. Janusza Szczepańskiego prof. IPPT PAN
Instytut Podstawowych Problemów Techniki
Polskiej Akademii Nauk

Warszawa, 2012

Streszczenie

Przedmiotem niniejszej pracy jest opracowanie i analiza nowego, efektywnego algorytmu mnożenia wielomianów i szeregów potęgowych o współczynnikach całkowitych oraz jego zastosowanie w procesie generowania krzywych eliptycznych na potrzeby kryptografii. Operacje tego typu odgrywają fundamentalną rolę podczas generowania parametrów dla systemów klucza publicznego, a ich efektywna implementacja ma bezpośrednie przełożenie na szybkość działania tych algorytmów w praktycznych zastosowaniach. Algorytm został specjalnie opracowany w celu przyspieszenia procesu generowania wielomianów modularnych, ale jego właściwości numeryczne okazały się na tyle dobre, że z całą pewnością znajdzie on zastosowanie również w przypadku innych problemów obliczeniowych. Istotą konstrukcji nowej metody było przede wszystkim dostosowanie jej do równoległego wykonywania obliczeń. W dzisiejszych czasach jest to bardzo ważna cecha, gdyż pozwala na pełne wykorzystanie mocy obliczeniowej oferowanej przez współczesne procesory.

Połączenie twierdzenia chińskiego o resztach i szybkiej transformaty Fouriera umożliwiło opracowanie bardzo efektywnej metody mnożenia. Pod pewnymi warunkami jest ona asymptotycznie szybsza niż algorytm wykorzystujący szybką transformatę Fouriera zarówno do mnożenia liczb, jak i wielomianów. Wynik ten jest niewątpliwie najważniejszym wnioskiem teoretycznym płynącym z tej pracy.

Praca zawiera dokładny opis konstrukcji algorytmu wraz z lematami i twierdzeniami uzasadniającymi poprawność jego działania. Przedstawione zostały wyniki testów numerycznych oraz implementacja wzorcowa proponowanego algorytmu. Otrzymane wyniki praktyczne pokazują, że nową metodę można skutecznie stosować już w przypadku wielomianów mających kilkaset współczynników.

Abstract

This paper aims to develop and analyze a new, effective algorithm for multiplying polynomials and power series with integer coefficients, and to use this algorithm in the process of generating elliptic curves for cryptographic applications. Such operations are of fundamental importance when generating parameters for public key cryptosystems, whereas their effective implementation translates directly into the speed of such algorithms in practical applications. The algorithm has been designed specifically to accelerate the process of generating modular polynomials, but due to its good numerical properties it may surely be used to solve other computational problems as well. The basic idea behind this new method was to adapt it to parallel computing. Nowadays, it is a very important property, as it allows to fully exploit the computing power offered by modern processors.

The combination of the Chinese Remainder Theorem and the Fast Fourier Transform made it possible to develop a highly effective multiplication method. Under certain conditions, it is asymptotically faster than the algorithm based on Fast Fourier Transform when applied to multiply both: numbers and polynomials. Undoubtedly, this result is the major theoretical conclusion of this paper.

This paper describes in detail the construction of the algorithm as well as theorems and lemmas justifying its correct operation. Moreover, it presents the results of numerical tests and a model implementation of the proposed algorithm. The achieved practical results show that the new method may be successfully applied to polynomials even with several hundred coefficients.

Spis treści

Spis tablic

Spis rysunków

Lista Algorytmów

Rozdział 1

Wstęp

Rozwój współczesnej gospodarki jest ściśle związany z rozwojem infrastruktury teleinformatycznej i Internetu. Nowoczesne metody przetwarzania i przesyłania informacji sprawiły, że dziś trudno wyobrazić sobie naszą egzystencję bez takich wynalazków jak: telefony komórkowe, poczta elektroniczna, sklepy internetowe, sieci społecznościowe, itp. Z roku na rok przesyłamy coraz więcej cyfrowych informacji. Zwiększa się też dość znacznie poziom wrażliwości danych, które przesyłane są w sieciach teleinformatycznych (numery kart kredytowych, dane osobowe, dokumenty medyczne, dokumenty finansowe). Stanowi to ogromne wyzwanie dla kryptografii. Nieustannie rosnąca liczba użytkowników sieci sprawia, że konieczne staje się poszukiwanie takich rozwiązań, które z jednej strony zapewnią odpowiedni poziom bezpieczeństwa, z drugiej zaś będą wydajne obliczeniowo.

Zastosowanie systemów klucza publicznego stanowi jeden z podstawowych i nierozerwalnych elementów zapewniających bezpieczeństwo protokołów kryptograficznych stosowanych do transmisji danych w sieciach teleinformatycznych. Algorytmy klucza publicznego pozwalają na zestawianie bezpiecznego kanału transmisyjnego, uwierzytelnianie użytkowników sieci i szyfrowanie przesyłanych informacji. Korzystamy z nich między innymi podczas logowania się na stronę internetową banku, podczas transakcji kartą kredytową w Internecie i podczas wymiany zaszyfrowanych wiadomości e-mailowych (protokoły SSL, PGP, S/MIME).

1.1 Motywacja i zakres pracy

W ostatnich latach szczególne zainteresowanie wzbudzają systemy oparte na krzywych eliptycznych. Ich wykorzystanie w kryptografii zostało zaproponowane w latach 80-tych ubiegłego wieku przez Millera [41] i Koblitza [30]. Przez długie lata kryptosystemy te cieszyły się jedynie zainteresowaniem środowisk naukowych. W zastosowaniach praktycznych prym wiodły techniki wykorzystu-

jące algorytm stworzony przez Diffiego i Hellmana [14] oraz Rivesta, Shamira i Adlemana [47]. Pojawienie się algorytmów DH (Diffiego-Hellmana), RSA (Rivesta, Shamira i Adlemana) oraz algorytmów opartych na krzywych eliptycznych spowodowało dynamiczny rozwój kryptoanalizy mechanizmów klucza publicznego. Lata badań utwierdzają nas w przekonaniu, że u podstaw bezpieczeństwa wspomnianych systemów leżą dwa następujące problemy matematyczne:

> *Problem logarytmu dyskretnego (DLP)* – Jest to problem znalezienia takiej liczby naturalnej x, że
>
> $$g^x = h,$$
>
> dla pewnych $g, h \in G$ i pewnej grupy skończonej G o zapisie multiplikatywnym.
>
> *Problem faktoryzacji* – Jest to problem znalezienia czynników pewnej złożonej liczby całkowitej.

Choć oba problemy mają bardzo prosty opis, to do tej pory nie znaleziono efektywnych algorytmów ich rozwiązywania. Wszystkie znane metody rozwiązywania tych problemów działają w czasie wykładniczym lub podwykładniczym względem rozmiaru danych wejściowych. Trudność obliczeniowa problemów nie przekłada się jednak bezpośrednio na bezpieczeństwo algorytmów kryptograficznych. Często bowiem trzeba mieć na uwadze szczególne przypadki, które mogą znacznie ułatwiać atak. Niezwykle wrażliwy pod tym względem okazał się algorytm RSA [8]. Jego implementacja musi brać pod uwagę wiele scenariuszy, które znacznie obniżają poziom bezpieczeństwa.

Jeśli chodzi o wzrost popularności krzywych eliptycznych, to można wyróżnić trzy najważniejsze powody takiego stanu rzeczy:

1. Zmiana wymagań dotyczących minimalnej długości kluczy szyfrujących opublikowana przez NIST (National Institute for Standards and Technology).
2. Potrzeba implementacji mechanizmów kryptograficznych w urządzeniach posiadających niewielkie możliwości pamięciowe i obliczeniowe.
3. Rosnąca popularność różnego rodzaju protokołów radiowych i konieczność ich kryptograficznej ochrony.

Zmiana wymagań dotyczących długości kluczy kryptograficznych związana jest z nieustającym wzrostem dostępnej mocy obliczeniowej oraz coraz lepszymi metodami ataków na konkretne algorytmy. Niewątpliwie największe postępy zostały poczynione w przypadku problemu faktoryzacji i łamania algorytmu RSA. Na przełomie ostatnich kilkunastu lat padały kolejne rekordy faktoryzacji:

– 1994 – faktoryzacja liczby 429 bitowej (sito kwadratowe),

– 1999 – faktoryzacja liczby 515 bitowej (sito ciał liczbowych),

– 2003 – faktoryzacja liczby 576 bitowej (sito ciał liczbowych),

– 2005 – faktoryzacja liczby 665 bitowej (sito ciał liczbowych),

– 2009 – faktoryzacja liczby 768 bitowej (sito ciał liczbowych).

Tak duże postępy mogły zostać osiągnięte dzięki rozwojowi metod sita, a zwłaszcza sita ciał liczbowych. Aktualnie złożoność obliczeniową najlepszego algorytmu faktoryzacji liczby całkowitej N (metoda sita ciał liczbowych) możemy zapisać wzorem [32]

$$C_{RSA}(N) = \exp\left(c_0 \log(N)^{1/3} (\log\log N)^{2/3}\right).$$

Z punktu widzenia teorii złożoności obliczeniowej jest to złożoność podwykładnicza: istotnie mniejsza od wykładniczej, ale równocześnie znacznie większa od wielomianowej.

Faktoryzacja modułu RSA o długości 768 bitów była tak naprawdę sygnałem ostrzegawczym dla wielu dostawców rozwiązań kryptograficznych. Jej efektem było masowe przechodzenie ze standardowo wykorzystywanych kluczy 1024 bitowych na klucze 2048 bitowe. Niestety taki wzrost długości klucza powoduje wiele problemów w przypadku urządzeń o ograniczonych zasobach. Wiąże się on bowiem z:

1. dwa razy większym zapotrzebowaniem na zasoby pamięciowe,
2. osiem razy większym zapotrzebowaniem na moc obliczeniową potrzebną do wykonania operacji prywatnej lub publicznej na kluczu,
3. szesnaście razy większym zapotrzebowaniem na moc obliczeniową podczas procesu generowania klucza.

Te wszystkie problemy są mało istotne w aplikacjach programowych, które działają na serwerach, komputerach PC, a nawet współczesnych telefonach. Natomiast w przypadku urządzeń, których pamięć RAM nie przekracza 1kB, a częstotliwość taktowania procesora oscyluje w przedziale od 8 do 32 MHz jest to nie lada wyzwanie. Bardzo ciekawe wyniki praktyczne uzyskali badacze z firmy Sun Microsystems, którzy porównywali praktyczną implementację krzywych eliptycznych z RSA na 8-bitowym procesorze ATmega128 taktowanym zegarem 8 MHz [23]. Uzyskane przez nich wyniki potwierdzają jednoznacznie opłacalność wykorzystania krzywych eliptycznych w małych układach mikroprocesorowych. Z przytoczonych wyników można wyciągnąć wniosek, że praktyczna implementacja operacji prywatnej RSA (podpisywania, deszyfrowania) jest zupełnie nieopłacalna, jeżeli mamy możliwość zastosowania krzywych eliptycznych.

Aby dokładnie zrozumieć gdzie leży przewaga krzywych eliptycznych nad takimi mechanizmami jak RSA i DH, musimy porównać ich poziomy bezpieczeństwa. Wiersze Tabeli 1.2 zawierają zestawienie równoważnych długości

Algorytm	Czas działania	RAM [bajty]	Kod [bajty]
160-bitowa krzywa eliptyczna	0.81 s	282	3682
192-bitowa krzywa eliptyczna	1.24 s	336	3979
224-bitowa krzywa eliptyczna	2.19 s	422	4812
1024-bitowa operacja publiczna RSA	0.43 s	542	1073
1024-bitowa operacja prywatna RSA	10.99 s	930	6292
2048-bitowa operacja publiczna RSA	1.94 s	1332	2854
2048-bitowa operacja prywatna RSA	83.26 s	1853	7736

Tablica 1.1: Porównanie czasu działania i zajętości zasobów dla algorytmu RSA i mechanizmów opartych na krzywych eliptycznych w 8-bitowym mikrokontrolerze ATmega128

Liczba bitów klucza szyfru symetrycznego	Liczba bitów klucza RSA/DSA/DH	Liczba bitów klucza krzywych eliptycznych
80	1024	160
112	2048	224
128	3072	256
192	7680	384
256	15360	512

Tablica 1.2: Równoważne długości kluczy kryptograficznych dla różnych algorytmów

kluczy dla poszczególnych mechanizmów kryptograficznych. Tak duża różnica w długościach kluczy pomiędzy RSA, a algorytmami opartymi na krzywych eliptycznych wynika z faktu, że nie istnieje podwykładniczy algorytm rozwiązywania problemu logarytmu dyskretnego w grupie punktów krzywej eliptycznej. W ogólnym przypadku złożoność obliczeniowa tego algorytmu dana jest wzorem [46]

$$C_{EC}(N) = N^{1/2},$$

gdzie N jest liczbą punktów krzywej eliptycznej. Warto w tym miejscu zaznaczyć, że w przypadku pewnych klas krzywych istnieją szybsze metody rozwiązania problemu logarytmu dyskretnego. Zagadnienie to będziemy poruszali w części poświęconej generowaniu krzywej eliptycznej do zastosowań kryptograficznych.

Popularność systemów wykorzystujących krzywe eliptyczne wzrosła również w bardzo konserwatywnym kręgu, jaki stanowią instytucje i agencje rządowe. Jednostki te przywiązują bardzo dużą wagę do bezpieczeństwa przechowywanych i przesyłanych danych. Ważnym atutem jest to, że koszt obliczeniowy

ewentualnego ataku na kryptosystem oparty na krzywej eliptycznej jest znacznie większy niż koszt ataku na RSA [5]. Dodatkowo istnieje możliwość wygenerowania i utajnienia własnej krzywej, co niewątpliwie stanowi dodatkowe zabezpieczenie. Niestety proces generowania krzywej do zastosowań kryptograficznych jest niezwykle wymagający obliczeniowo. Może być on podzielony na dwa etapy:

1. fazę obliczeń wstępnych dla wybranego ciała (generowanie wielomianów modularnych),
2. fazę znajdowania krzywej eliptycznej (zliczanie punktów dla losowo wybranych krzywych i weryfikacja ich własności kryptograficznych).

Przy klasycznym podejściu faza obliczeń wstępnych dla dużych krzywych (zadanych nad 512 bitowym ciałem) może trwać nawet miesiąc. Tak długi czas działania stanowi bardzo dobrą motywację do poszukiwania nowych, szybszych algorytmów, które pozwoliłyby na skrócenie całego procesu.

W literaturze można znaleźć wiele sposobów na wyznaczanie wielomianów modularnych. Najnowsze metody generowania klasycznych wielomianów modularnych zostały zaproponowane przez Charlesa i Lautera [3] oraz Engego [16]. Poza tym Müller zaproponował [43] inną rodzinę wielomianów modularnych, które również mogą być wykorzystywane w procesie zliczania punktów krzywej eliptycznej. Wielomiany Müllera charakteryzują się mniejszą liczbą niezerowych współczynników i mniejszymi wartościami bezwzględnymi współczynników niż klasyczne wielomiany modularne. Wszyscy wspomniani autorzy podają złożoność obliczeniową algorytmów służących do wyznaczania wielomianów modularnych w oparciu o założenie, że mnożenie zarówno wielomianów, jak i ich współczynników realizowane jest przy użyciu szybkiej transformaty Fouriera. Taki algorytm mnożenia ma złożoność

$$O((d \log d)(n \log n)),$$

gdzie d jest stopniem wielomianu, a n jest liczbą bitów największego czynnika. Jednak zastosowanie asymptotycznie szybkiego algorytmu mnożenia liczb staje się opłacalne dopiero, gdy liczby mają znaczną długość. Według raportu Garcia [19], szybka implementacja mnożenia w GMP (GNU Multiple Precision Arithmetic Library) jest w stanie dorównać klasycznym algorytmom mnożenia dopiero dla liczb mających co najmniej $2^{17} = 131072$ bitów. Dlatego też warto byłoby opracować taki algorytm mnożenia wielomianów, który działałby szybko dla wielomianów mających relatywnie małe współczynniki.

1.2 Cel i struktura pracy

Praca koncentrować się będzie na opracowaniu i analizie nowego, szybkiego algorytmu mnożenia wielomianów o współczynnikach całkowitych, opartego na

twierdzeniu chińskim o resztach (CRT - Chinese Remainder Theorem) i szybkiej transformacie Fouriera (FFT - Fast Fourier Transform). Algorytm ten ma fundamentalne znaczenie w procesie efektywnego generowania parametrów dla systemów ECDSA i ECDH działających na krzywych eliptycznych. Uzyskane wyniki pozwalają na zwiększenie wydajności procesu generowania krzywych poprzez zrównoleglenie obliczeń. Główna idea proponowanej metodyki polega na wykorzystaniu twierdzenia CRT do reprezentacji liczb całkowitych jako skończonych ciągów liczb o mniejszych rozmiarach, a następnie zrównolegleniu obliczeń na takich reprezentacjach. Z kolei, w celu przyśpieszenia obliczeń na wielomianach zastosowana jest szybka transformata Fouriera wykorzystująca w istocie przekształcenia wielomianów z reprezentacji przez współczynniki do reprezentacji przez wartości w punktach. Proponowane metody są na tyle ogólne, że mogą być wykorzystane do przyspieszenia dowolnych obliczeń w pierścieniach wielomianów i szeregów potęgowych o współczynnikach całkowitych.

Celem pracy jest zaprojektowanie efektywnego, równoległego algorytmu mnożenia wielomianów i szeregów potęgowych z użyciem jednocześnie CRT i FFT. Ma to duże znaczenie praktyczne nie tylko dla procesu generowania krzywych eliptycznych, ale może również być wykorzystane do przeprowadzania innego rodzaju obliczeń (takich, które wykorzystują operacje na długich wielomianach o współczynnikach rzędu od kilkuset do kilku tysięcy bitów). Trzeba również zaznaczyć, że proponowana metoda pozwala na bardzo łatwe zrównoleglenie obliczeń. Jest to jej istotna zaleta, gdyż uzyskujemy w ten sposób możliwość pełnego wykorzystania mocy obliczeniowej oferowanej przez współczesne wielordzeniowe procesory.

Rozpoczęcie prac badawczych motywowane było zagadnieniami związanymi z generowaniem krzywych eliptycznych na potrzeby kryptografii. Dlatego też całość pracy została utrzymana w ścisłym związku z tematem krzywych eliptycznych i ich praktycznego wykorzystania do ochrony informacji.

W rozdziale 2 zdefiniowano pojęcie krzywej eliptycznej, wprowadzono działanie grupowe na jej punktach oraz przedstawiono interpretację krzywej eliptycznej nad ciałem liczb zespolonych i w rzeczywistej przestrzeni rzutowej.

Rozdział 3 pokazuje w jaki sposób krzywe eliptyczne wykorzystywane są we współczesnej kryptografii. Zawarto w nim przykładowe mechanizmy służące między innymi do: bezpiecznej wymiany kluczy, szyfrowania asymetrycznego i podpisu cyfrowego. Wszystkie zaprezentowane algorytmy wykorzystują w swojej implementacji grupę punktów krzywej eliptycznej jako dziedzinę, w której wykonywane są operacje.

W rozdziale 4 zaprezentowano warunki, jakie musi spełniać krzywa przeznaczona do zastosowań kryptograficznych oraz przedstawiono ideę algorytmu zliczania punktów krzywej. Algorytm ten jest podstawowym narzędziem, które daje możliwość sprawdzenia, czy dana krzywa może być wykorzystana w kryp-

tografii, czy też nie. W rozdziale tym wyjaśniono również dlaczego wielomiany modularne są tak istotne z punktu widzenia algorytmów służących do zliczania punktów krzywej.

Rozdział 5 niemal w całości stanowi wkład własny. Zostały tam zebrane lematy i twierdzenia uzasadniające poprawność działania opracowanego algorytmu oraz oszacowania jego złożoności obliczeniowej. Wnioski zawarte w tym rozdziale są kluczowe dla określenia w jakich przypadkach proponowany algorytm powinien być stosowany, a kiedy jego wykorzystanie nie ma najmniejszego sensu.

W rozdziale 6 przedstawione zostały algorytmy wchodzące w skład proponowanej metody mnożenia. Udowodniono też twierdzenie o transformacie Fouriera i transformacie do niej odwrotnej. Twierdzenia te zostały podane w formie, która została bezpośrednio przełożona na efektywną implementację algorytmu. Stanowią więc teoretyczne uzasadnienie implementacji zawartej w załączniku B. Ostatnia część tego rozdziału opisuje wykonane eksperymenty numeryczne oraz ich wyniki.

Załącznik B zawiera zoptymalizowaną, równoległą implementację proponowanego algorytmu. Kod został przygotowany w oparciu o popularny i powszechnie używany standard programowania równoległego – OpenMP. W załączniku A umieszczono informacje niezbędne podczas kompilacji biblioteki oraz przykładowy program wykorzystujący opracowany algorytm.

Rozdział 2

Krzywe eliptyczne

Można zaryzykować stwierdzenie, że pierwsze rezultaty dotyczące krzywych eliptycznych pochodzą z prac Diofantosa, żyjącego w III wieku naszej ery [31]. Niewiele wiadomo o samym Diofantosie, a jedyne wskazówki co do jego pochodzenia daje styl, którym się posługiwał w swoich pracach. Styl ten można określić mianem *babilońskiego*, gdyż są to zadania z rozwiązaniami. Nie ma tam natomiast śladu żadnych twierdzeń, czy aksjomatów, z których wyprowadzane są określone własności. Głównym polem zainteresowań Diofantosa było poszukiwanie liczb naturalnych spełniających równania wielomianowe (stąd też dzisiejsza nazwa – równania diofantyczne) stopnia mniejszego lub równego 6. Jego ogromnym wkładem było również wprowadzenie elementów notacji symbolicznej. Nie była to jeszcze pełna notacja symboliczna, ale koncepcje Diofantosa stworzyły solidne podstawy do jej rozwoju.

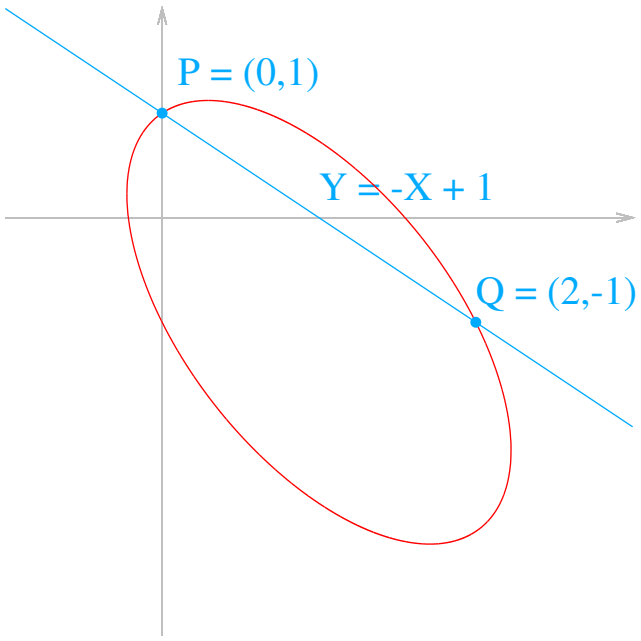
Już od czasów Pitagorasa wiadomo było, że niektóre równania kwadratowe (na przykład $Y^2 + X^2 = 3$) nie mają rozwiązań w liczbach wymiernych. Diofantos poszedł jednak dalej i stwierdził, że jeżeli równanie kwadratowe ma przynajmniej jedno rozwiązanie wymierne, to ma ich nieskończenie wiele. Jego rozumowanie prześledzimy na następującym przykładzie.

Przykład 1 Rozważmy równanie kwadratowe (Rysunek 2.1):

$$3X^2 + 2XY + Y^2 - 4X - 1 = 0.$$

Jednym z rozwiązań tego równania jest punkt $(0, 1)$. Przez ten punkt możemy przeprowadzić rodzinę prostych $Y = mX + 1$. Taka prosta albo jest styczna do krzywej wyznaczonej przez nasze równanie kwadratowe, albo przecina ją jeszcze w innym punkcie. Aby wyznaczyć ten punkt musimy rozwiązać układ równań:

$$\begin{cases} 3X^2 + 2XY + Y^2 - 4X - 1 = 0, \\ Y = mX + 1. \end{cases}$$



Rysunek 2.1: Równanie $3X^2+2XY+Y^2-4X-1=0$ wraz z dwoma punktami wymiernymi

Eliminując zmienną Y otrzymujemy

$$\begin{aligned} & 3X^2+2X(mX+1)+(mX+1)^2-4X-1 \\ = \; & 3X^2+2mX^2+2X+m^2X^2+2mX+1-4X-1 \\ = \; & (m^2+2m+3)X^2+(2m-2)X \\ = \; & X\left((m^2+2m+3)X+(2m-2)\right) \end{aligned}$$

Jak widać odcięta drugiego punktu przecięcia prostej z krzywą wynosi $x = \frac{-2m+2}{m^2+2m+3}$. W związku z tym proste $Y = mX+1$ przechodzące przez punkt $(0,1)$ wyznaczają nam całą rodzinę innych punktów wymiernych:

$$x = \frac{-2m+2}{m^2+2m+3}, \quad y = mx+1 = \frac{-m^2+4m+3}{m^2+2m+3}.$$

□

Metodę podobną do tej opisanej w powyższym przykładzie Diofantos próbował również zastosować dla krzywych stopnia trzeciego. Nie udało mu się

jednak sparametryzować punktów wymiernych tej krzywej. Mimo to odkrył, że jeśli prosta przecina krzywą w dwóch punktach wymiernych, to trzeci punkt przecięcia (o ile istnieje), też jest punktem wymiernym. Obserwacja ta daje możliwość wprowadzenia na krzywej stopnia trzy działania grupowego. Szczegóły można znaleźć w książce Silvermana i Tatea [56].

Dlaczego jednak krzywą stopnia trzy nazywa się *krzywą eliptyczną*, jeżeli nie ma ona z elipsą nic wspólnego? Odpowiedź na to pytanie wiąże się z problemem wyznaczenia długości łuku elipsy. Dla pierwszej ćwiartki kartezjańskiego układu współrzędnych problem ten sprowadza się do wyznaczenia całki

$$\ell = \int_0^a \sqrt{1 + y'(x)^2} dx,$$

gdzie

$$y(x) = b\sqrt{1 - (x/a)^2}, \quad y'(x) = -\frac{xb}{a^2} \frac{1}{\sqrt{1 - (x/a)^2}}.$$

Całkę tą można po podstawieniu $z = x/a$ przekształcić do postaci

$$\begin{aligned} \ell &= a \int_0^1 \sqrt{\frac{1 - ez^2}{(1 - z^2)}} dz \\ &= a \int_0^1 \frac{1 - ez^2}{\sqrt{(1 - ez^2)(1 - z^2)}} dz \end{aligned}$$

gdzie $e = 1 - (b/a)^2$. Jest to tak zwana *całka eliptyczna drugiego rodzaju*. Okazuje się, że wiele problemów praktycznych prowadzi do całek postaci

$$u(y) = \int_C^y R\left(t, \sqrt{P(t)}\right) dt,$$

dla których R jest funkcją wymierną dwóch zmiennych, a P jest wielomianem stopnia 3 lub 4. Całki te noszą nazwę *całek eliptycznych* i nie dają się wyrazić za pomocą funkcji elementarnych. Funkcje odwrotne do całek eliptycznych noszą nazwę *funkcji eliptycznych*.

Przykład 2 Jedną z całek eliptycznych jest funkcja

$$u = \int_y^\infty \frac{1}{\sqrt{4t^3 - g_2 t - g_3}} dt.$$

Funkcją do niej odwrotną jest eliptyczna funkcja $\wp$ Weierstrassa

$$y = \wp(u),$$

która spełnia zależność

$$\wp'(u)^2 = 4\wp(u)^3 - g_2\wp(u) - g_3.$$

Jak widać funkcja eliptyczna spełnia równanie pewnej krzywej. Krzywą tą właśnie z tego powodu nazywamy *krzywą eliptyczną*. □

2.1 Równanie krzywej eliptycznej

Nasze rozważania rozpoczniemy od podania definicji krzywej eliptycznej.

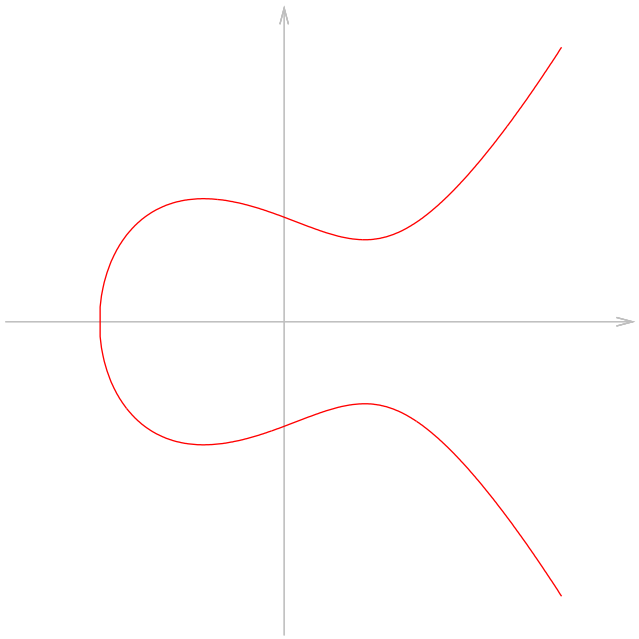
Definicja 1 Krzywą eliptyczną C nad ciałem K w przestrzeni afinicznej $K \times K$ nazywamy nieosobliwą krzywą zadaną równaniem Weierstrassa:

$$C : Y^2 + a_1XY + a_3Y = X^3 + a_2X^2 + a_4X + a_6 \tag{2.1}$$

wraz z punktem *w nieskończoności* oznaczanym przez $\mathcal{O}$. Nieosobliwość krzywej oznacza, że jeśli

$$F(X, Y) = Y^2 + a_1XY + a_3Y - X^3 - a_2X^2 - a_4X - a_6,$$

to pochodne cząstkowe $\partial F/\partial X$ i $\partial F/\partial Y$ nie są jednocześnie równe zero w żadnym punkcie krzywej. □



Rysunek 2.2: Nieosobliwa krzywa eliptyczna zadana równaniem $Y^2 = X^3 - X + 1$

Rysunek 2.2 przedstawia krzywą eliptyczną nad ciałem liczb rzeczywistych, dla której odpowiednie współczynniki równania Weierstrassa mają wartość:

$$\begin{aligned} a_1 &= 0, \\ a_3 &= 0, \\ a_2 &= 0, \\ a_4 &= -1, \\ a_6 &= 1. \end{aligned}$$

Na Rysunkach 2.3 i 2.4 przedstawiono natomiast osobliwe krzywe zadane równaniem Weierstrassa. Rysunek 2.3 ilustruje krzywą o równaniu $Y^2 = X^3 + X^2$, która posiada samoprzecięcie w punkcie $(0, 0)$. Punkt ten jest punktem osobliwym, gdyż:

$$\begin{aligned} \frac{Y^2 - X^3 - X^2}{\partial X}(0,0) &= (-3X^2 - 2X)(0,0) = 0, \\ \frac{Y^2 - X^3 - X^2}{\partial Y}(0,0) &= (2Y)(0,0) = 0. \end{aligned}$$

Na Rysunku 2.4 została natomiast przedstawiona krzywa $Y^2 = X^3$. Jej osobliwość znajduje się również w punkcie $(0, 0)$:

$$\begin{aligned} \frac{Y^2 - X^3}{\partial X}(0,0) &= (-3X^2)(0,0) = 0, \\ \frac{Y^2 - X^3}{\partial Y}(0,0) &= (2Y)(0,0) = 0. \end{aligned}$$

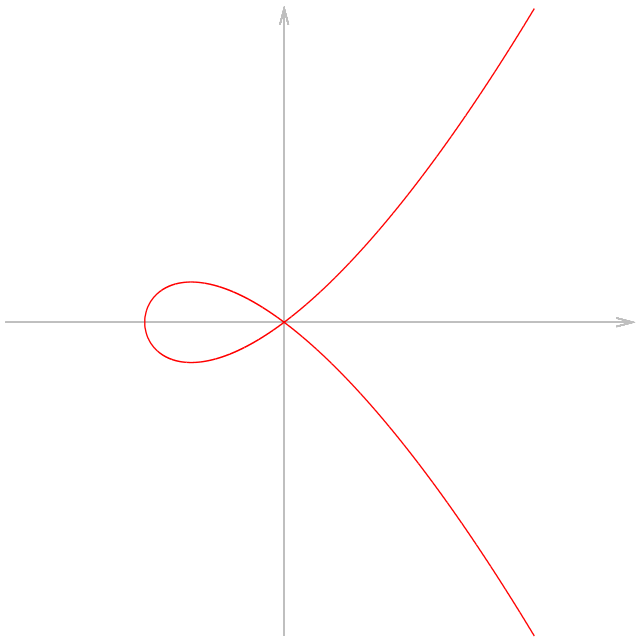
Jeżeli krzywa eliptyczna określona jest za pomocą równania 2.1, to wygodnie jest wprowadzić następujące oznaczenia:

$$\begin{aligned} b_2 &= a_1^2 + 4a_2, \\ b_4 &= a_1 a_3 + 2a_4, \\ b_6 &= a_3^2 + 4a_6, \\ b_8 &= a_1^2 a_6 + 4a_2 a_6 - a_1 a_3 a_4 + a_2 a_3^2 - a_4^2, \\ c_4 &= b_2^2 - 24 b_4, \\ c_6 &= -b_2^3 + 36 b_2 b_4 - 216 b_6. \end{aligned}$$

Dla tak określonych stałych definiujemy *wyróżnik krzywej eliptycznej* jako

$$\Delta = -b_2^2 b_8 - 8b_4^3 - 27b_6^2 + 9 b_2 b_4 b_6.$$

Dla tak zdefiniowanej wielkości Δ można udowodnić twierdzenie, że krzywa zadana równaniem Weierstrassa 2.1 jest nieosobliwa wtedy i tylko wtedy, gdy



Rysunek 2.3: Osobliwa krzywa eliptyczna zadana równaniem $Y^2 = X^3 + X^2$

$\Delta \neq 0$. Inną wartością charakterystyczną dla krzywej eliptycznej jest jej *j-niezmiennik*

$$j(C) = \frac{c_4^3}{\Delta}.$$

Funkcja ta jest ściśle związana z pojęciem izomorfizmu krzywych eliptycznych:

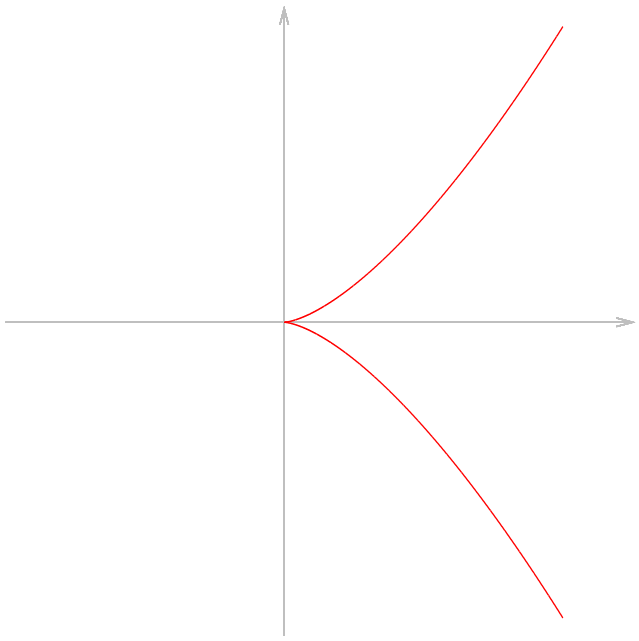
Definicja 2 Dwie krzywe eliptyczne $C(X, Y)$ i $C'(X', Y')$ określone równaniami Weierstrassa są izomorficzne nad ciałem K wtedy i tylko wtedy, gdy istnieją stałe $r, s, t \in K$ oraz $u \in K^\times$ takie, że

$$C(u^2X' + r, u^3Y' + su^2X' + t) = C'(X', Y').$$

□

Określone w ten sposób pojęcie izomorfizmu krzywych eliptycznych wiąże się z wartością j-niezmiennika w sposób następujący [55]:

Twierdzenie 1 Jeżeli krzywe C i C' są izomorficzne nad ciałem K, to ich j-niezmienniki są sobie równe. Jeżeli j-niezmienniki krzywych C i C' określonych nad ciałem K są sobie równe, to krzywe są izomorficzne nad domknięciem algebraicznym ciała K. ■



Rysunek 2.4: Osobliwa krzywa eliptyczna zadana równaniem $Y^2 = X^3$

Równanie Weierstrassa definiuje krzywą eliptyczną w postaci ogólnej. Jeżeli założymy, że ciało K ma charakterystykę różną od 2 i 3, to stosując następującą zamianę zmiennych

$$\begin{aligned} X &\mapsto X - \frac{b_2}{12}, \\ Y &\mapsto Y - \frac{a_1}{2}\left(X - \frac{b_2}{12}\right) - \frac{a_3}{2}, \end{aligned}$$

możemy uprościć równanie krzywej do postaci:

$$E : Y^2 = X^3 + aX + b.$$

Dla takiej krzywej wyróżnik jest zadany wzorem:

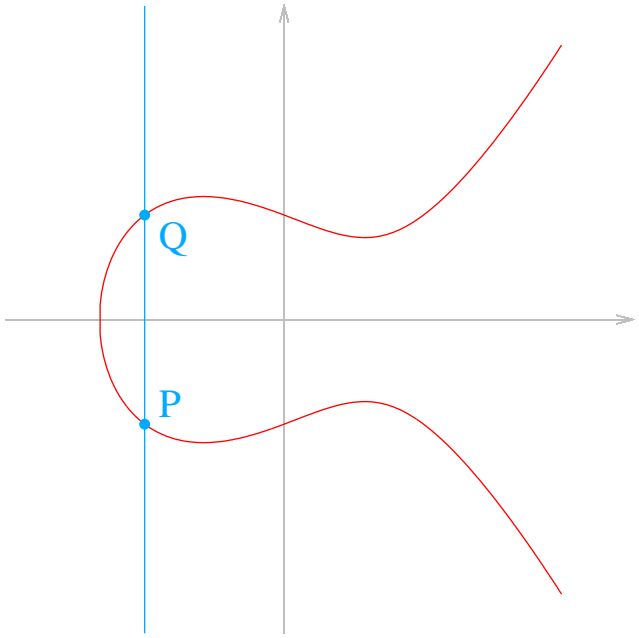
$$\Delta(E) = -16(4a^3 + 27b^2),$$

a j-niezmiennik ma postać:

$$j(E) = 1728\frac{4a^3}{4a^3 + 27b^2}.$$

2.2 Działanie grupowe na punktach krzywej

Mając do dyspozycji równanie Weierstrassa 2.1 możemy wprowadzić działanie grupowe na punktach krzywej eliptycznej. Wykorzystamy w tym miejscu takie samo podejście, jakie stosował Diofantos. Mianowicie będziemy przez dwa dane punkty krzywej prowadzili prostą i szukali trzeciego punktu przecięcia. Aby jednak nasze rozważania miały sens musimy mieć pewność, że każda prosta mająca dwa punkty wspólne z krzywą C będzie też miała trzeci punkt wspólny. Osiągniemy to dzięki punktowi, który nazwaliśmy punktem w nieskończoności. Na płaszczyźnie afinicznej możemy sobie wyobrażać, że każda prosta pionowa przecina punkt w nieskończoności (gdy zinterpretujemy krzywą w terminach przestrzeni rzutowej stanie się jasne dlaczego tak się dzieje).



Rysunek 2.5: Znajdowanie punktu przeciwnego: $P + Q + \mathcal{O} = \mathcal{O}$

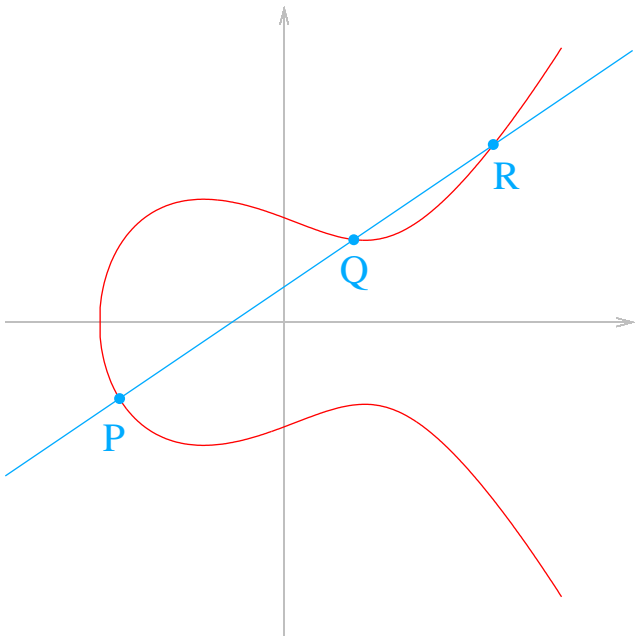
1. Elementem neutralnym naszego działania grupowego jest punkt w nieskończoności – $\mathcal{O}$.

2. Jeśli dany jest punkt $P = (x_1, y_1) \in C$, to punkt przeciwny definiujemy jako
$$Q = -P = (x_1, -y_1 - a_1 x_1 - a_3).$$

Rysunek 2.5 przedstawia geometryczną interpretację punktu przeciwnego na krzywej eliptycznej.

3. Jeśli dane są punkty $P = (x_1, y_1) \in C$, $Q = (x_2, y_2) \in C$ i spełniony jest warunek $Q \neq -P$, to wyróżniamy dwa przypadki:

   (a) Dla $x_1 \neq x_2$ (dodawanie dwóch różnych punktów) przyjmujemy



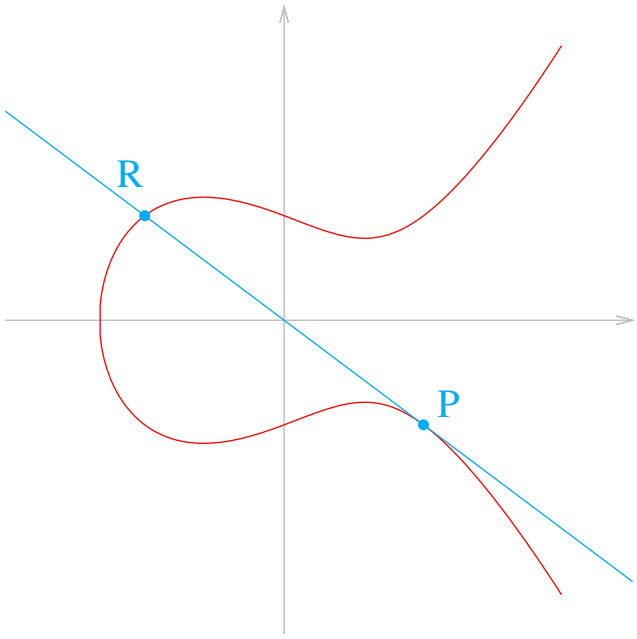
Rysunek 2.6: Dodawanie dwóch różnych punktów: $P + Q + R = \mathcal{O}$

$$\begin{aligned} \lambda &= \frac{y_2 - y_1}{x_2 - x_1}, \\ \nu &= \frac{y_1 x_2 - y_2 x_1}{x_2 - x_1}. \end{aligned}$$

Rysunek 2.6 przedstawia geometryczną interpretację dodawania dwóch różnych punktów krzywej eliptycznej. W tym przypadku $P + Q = -R$.

   (b) Dla $x_1 = x_2$ (podwajanie punktu) przyjmujemy

$$\begin{aligned} \lambda &= \frac{3x_1^2 + 2a_2 x_1 + a_4 - a_1 y_1}{2y_1 + a_1 x_1 + a_3}, \\ \nu &= \frac{-x_1^3 + a_4 x_1 + 2a_6 - a_3 y_1}{2y_1 + a_1 x_1 + a_3}. \end{aligned}$$



Rysunek 2.7: Podwajanie punktu: $[2]P + R = \mathcal{O}$

Rysunek 2.7 przedstawia geometryczną interpretację podwajania punktu krzywej eliptycznej. W tym przypadku $[2]P = -R$.

Zwróćmy uwagę na fakt, że prosta $Y = \lambda X + \nu$ przecina krzywą w punktach P i Q, gdy $P \neq Q$ i jest styczna do krzywej w punkcie P jeżeli $P = Q$. W obu przypadkach wynik dodawania punktów określa formuła

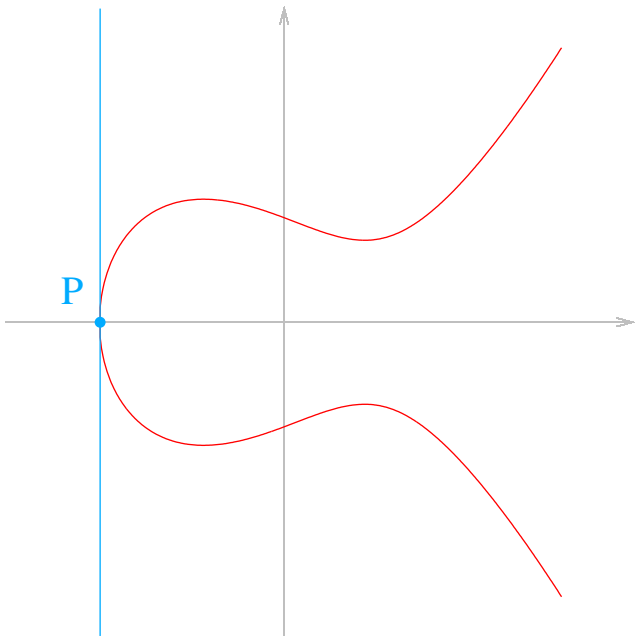
$$P + Q = (\lambda^2 + a_1\lambda - a_2 - x_1 - x_2, -(\lambda + a_1)x_3 - \nu - a_3).$$

Wyprowadzone wzory pozwalają na wykonywanie działania grupowego na punktach krzywej eliptycznej opisanej równaniem 2.1. W przypadku gdy krzywa eliptyczna zadana jest nad ciałem charakterystyki różnej od 2 i 3, wzory te można znacznie uprościć. Przyjmijmy zatem, że E jest krzywą postaci

$$E : Y^2 = X^3 + aX + b.$$

Działanie grupowe dla tej krzywej upraszcza się w następujący sposób:

1. Elementem neutralnym naszego działania grupowego jest punkt w nieskończoności – $\mathcal{O}$.



Rysunek 2.8: Podwajanie punktu rzędu 2: $[2]P + \mathcal{O} = \mathcal{O}$

2. Jeśli dany jest punkt $P = (x_1, y_1) \in E$, to punkt przeciwny definiujemy jako
$$Q = -P = (x_1, -y_1).$$

3. Jeśli dane są punkty $P = (x_1, y_1) \in E$, $Q = (x_2, y_2) \in E$ i spełniony jest warunek $Q \neq -P$, to wyróżniamy dwa przypadki:

   (a) Dla $x_1 \neq x_2$ (dodawanie dwóch różnych punktów) przyjmujemy
   $$\lambda = \frac{y_2 - y_1}{x_2 - x_1}.$$

   (b) Dla $x_1 = x_2$ (podwajanie punktu) przyjmujemy
   $$\lambda = \frac{3x_1^2 + a}{2y_1}.$$

   Wynik dodawania punktów określa formuła
   $$P + Q = (\lambda^2 - x_1 - x_2, (x_1 - x_3)\lambda - y_1).$$

2.3 Krzywa eliptyczna nad ciałem liczb zespolonych

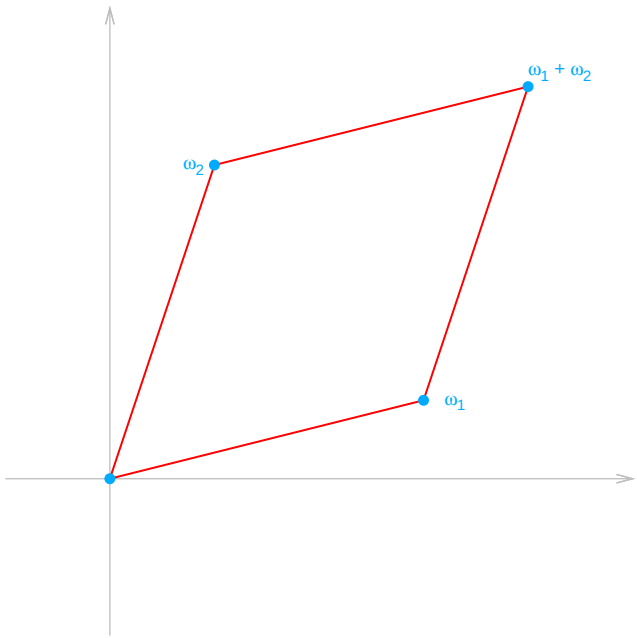
Jeśli krzywa eliptyczna dana jest równaniem $Y^2 = X^3 + aX + b$, to wykonując podstawienie

$$\begin{aligned} X &\mapsto 4X, \\ Y &\mapsto 4Y \end{aligned}$$

otrzymujemy równanie $Y^2 = 4X^3 + \frac{a}{4}X + \frac{b}{16}$. Przyjmując, że $g_2 = -\frac{a}{4}$ i $g_3 = -\frac{b}{16}$ mamy

$$Y^2 = 4X^3 - g_2X - g_3, \tag{2.2}$$

które pojawiło się już w Przykładzie 2. Do końca tej części będziemy zakładali, że krzywa eliptyczna zadana jest równaniem 2.2.



Rysunek 2.9: Krata na płaszczyźnie zespolonej powiązana z krzywą eliptyczną

Weierstrass wykazał, że jeśli stałe zespolone g_2, g_3 są takie, że wielomian $4X^3 - g_2X - g_3$ nie ma pierwiastków wielokrotnych ($g_2^3 - 27g_3^2 \neq 0$), to można

znaleźć takie liczby zespolone ω_1, ω_2 definiujące kratę $\Lambda = \omega_1\mathbb{Z} + \omega_2\mathbb{Z}$, że:

$$\begin{aligned} g_2 &= 60 \sum_{\omega\in\Lambda\setminus\{0\}} \frac{1}{\omega^4}, \\ g_3 &= 140 \sum_{\omega\in\Lambda\setminus\{0\}} \frac{1}{\omega^6}, \\ \wp(u) &= \frac{1}{u^2} + \sum_{\omega\in\Lambda\setminus\{0\}} \left(\frac{1}{(u-\omega)^2} - \frac{1}{\omega^2} \right) \end{aligned}$$

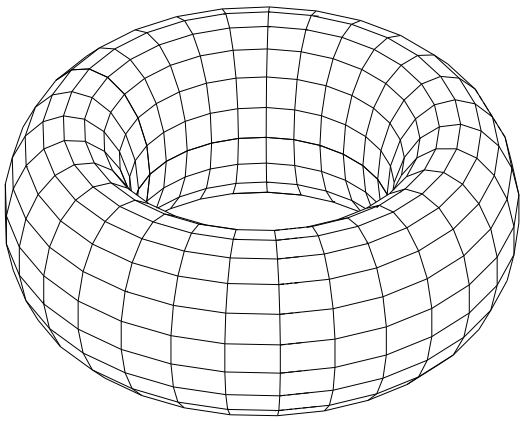
i zachodzi równość

$$\wp'(u)^2 = 4\wp(u)^3 - g_2\wp(u) - g_3.$$

Widzimy zatem, że każdej liczbie zespolonej $u \in \mathbb{C} \setminus \Lambda$ odpowiada punkt na krzywej

$$P(u) = (\wp(u), \wp'(u)).$$

Dla liczb zespolonych $u \in \Lambda$ przyjmujemy $P(u) = \mathcal{O}$. Niestety postać funkcji $\wp$



Rysunek 2.10: Krzywa eliptyczna nad $\mathbb{C}$ jest izomorficzna jako grupa z torusem

Weierstrassa sprawia, że jeśli tylko dwie liczby zespolone u, v spełniają warunek

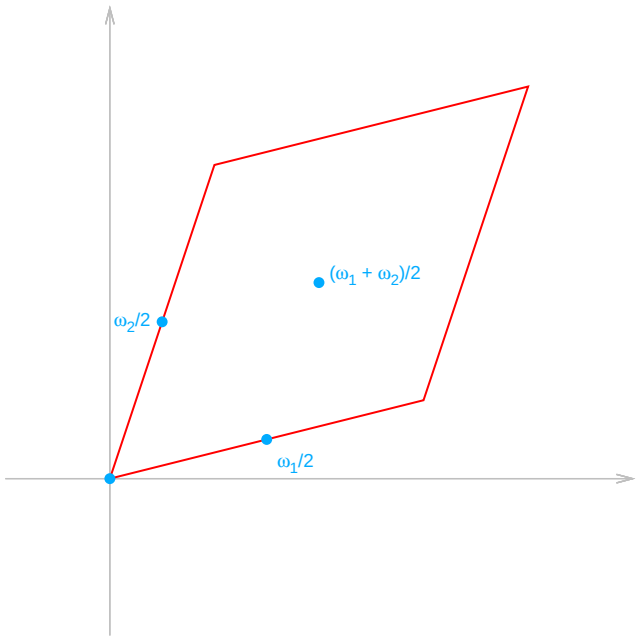
$u - v \in \Lambda$, to $P(u) = P(v)$. Zatem przekształcenie P nie jest różnowartościowe. Jeśli jednak przyjmiemy, że dziedziną P nie jest cała grupa $\mathbb{C}$, a jedynie jej grupa ilorazowa $\mathbb{C}/\Lambda$, to otrzymamy bijekcję.

Można wykazać, że określone w ten sposób przekształcenie P jest tak naprawdę izomorfizmem grupy $\mathbb{C}/\Lambda$ i grupy punktów krzywej $Y^2 = 4X^3 - g_2X - g_3$

$$P(u + v) = P(u) + P(v).$$

W powyższej formule $u+v$ oznacza zwykłe dodawanie liczb zespolonych, a $P(u)+P(v)$ oznacza dodawanie punktów na krzywej eliptycznej.

Wprowadzony izomorfizm pokazuje, że krzywa eliptyczna nad $\mathbb{C}$ może być utożsamiana z pewnym równoległobokiem, którego wierzchołki znajdują się w punktach kraty Λ, a przeciwległe krawędzie są ze sobą utożsamiane. Można więc grupę $\mathbb{C}/\Lambda$ utożsamiać z powierzchnią torusa tak, jak to zostało zobrazowane na Rysunku 2.10.



Rysunek 2.11: Punkty rzędu 2 na krzywej eliptycznej zadanej nad $\mathbb{C}$

Taka interpretacja punktów krzywej eliptycznej pozwala też bardzo dużo powiedzieć o elementach torsyjnych (punktach skończonego rzędu). Przypuśćmy,

że chcemy znaleźć punkt rzędu 2. Oznacza to, że musimy znaleźć taką liczbę zespoloną $u \notin \Lambda$, że $2u \in \Lambda$. W grupie $\mathbb{C}/\Lambda$ istnieją dokładnie trzy takie niezerowe elementy: $\frac{\omega_1}{2}$, $\frac{\omega_2}{2}$ i $\frac{\omega_1+\omega_2}{2}$ tak, jak to zostało przedstawione na Rysunku 2.11.

Uogólniając to rozumowanie na punkty, których rząd dzieli dowolną liczbę m, możemy zapisać, że są to wszystkie liczby u, dla których zachodzi $mu \in \Lambda$. Bez trudu można zauważyć, że każda podgrupa punktów o rzędzie dzielącym liczbę m jest generowana przez dwa liniowo niezależne elementy: $\frac{\omega_1}{m}$ i $\frac{\omega_2}{m}$. Jeśli zatem przez $E[m]$ oznaczymy zbiór tych punktów krzywej zespolonej, których rząd dzieli liczbę m, to mamy

$$E[m] \simeq \mathbb{Z}/m\mathbb{Z} \times \mathbb{Z}/m\mathbb{Z}.$$

Warto zapamiętać tą zależność, gdyż jest ona przy dodatkowych założeniach prawdziwa również dla ciał charakterystyki różnej od zera.

Pojęcie izomorfizmu krzywych eliptycznych ma też dużo prostszą interpretację nad ciałem liczb zespolonych niż w przypadku ogólnym. Okazuje się, że okresy ω_1, ω_2 można dobrać w taki sposób, aby iloraz ω_2/ω_1 leżał w górnej półpłaszczyźnie płaszczyzny zespolonej

$$\tau = \frac{\omega_2}{\omega_1} \in \mathcal{H} = \{z \in \mathbb{C} : \Im(z) > 0\}.$$

Dla tak zdefiniowanej liczby τ można wykazać, że dwie krzywe zadane za pomocą krat

$$\begin{aligned} \Lambda &= \omega_1\mathbb{Z} + \omega_2\mathbb{Z}, \\ \Lambda' &= \omega_1'\mathbb{Z} + \omega_2'\mathbb{Z} \end{aligned}$$

są izomorficzne wtedy i tylko wtedy, gdy $\tau = \omega_2/\omega_1 = \omega_2'/\omega_1'$. Krzywą zdefiniowaną przez kratę $\mathbb{Z}+\tau\mathbb{Z}$ oznaczamy przez E_τ. Wprowadzamy również specjalne oznaczenie na j-niezmiennik takiej krzywej

$$j(\tau) = j(E_\tau).$$

Dla tak określonego j-niezmiennika prawdziwy jest następujący lemat.

Lemat 1 Dla każdej macierzy

$$\gamma = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \in SL_2(\mathbb{Z})$$

prawdziwa jest zależność

$$j(\tau) = j\left(\frac{a\tau + b}{c\tau + d}\right),$$

co oznacza, że j-niezmiennik jest funkcją okresową o okresie równym 1, a jego rozwinięcie w szereg Fouriera ma postać

$$j(\tau) = q^{-1} + 744 + \sum_{n\geq 1} c_n q^n,$$

gdzie współczynniki c_n są dodatnimi liczbami całkowitymi, a $q = e^{2\pi i\tau}$. ■

2.4 Krzywe eliptyczne w rzeczywistej przestrzeni rzutowej

Podczas wyprowadzania wzorów na sumę punktów krzywej eliptycznej zadanej na płaszczyźnie afinicznej konieczne było sprawdzanie, czy nie próbujemy dodawać do siebie punktu P i $-P$. Wynikało to z faktu, że nie zawsze prosta mająca dwa punkty wspólne z krzywą zadaną równaniem 2.1 miała z nią wspólny również trzeci punkt. Problem ten rozwiązuje się automatycznie, gdy przeniesiemy krzywą do przestrzeni rzutowej. Zanim jednak szczegółowo wyjaśnimy ten pomysł, podamy definicję krzywej eliptycznej w przestrzeni rzutowej.

Definicja 3 Krzywą eliptyczną C' nad ciałem K w przestrzeni rzutowej $P^2(K)$ nazywamy nieosobliwą krzywą zadaną jednorodnym równaniem Weierstrassa:

$$C' : Y^2Z + a_1XYZ + a_3YZ^2 = X^3 + a_2X^2Z + a_4XZ^2 + a_6Z^3 \tag{2.3}$$

Nieosobliwość krzywej oznacza, że jeśli

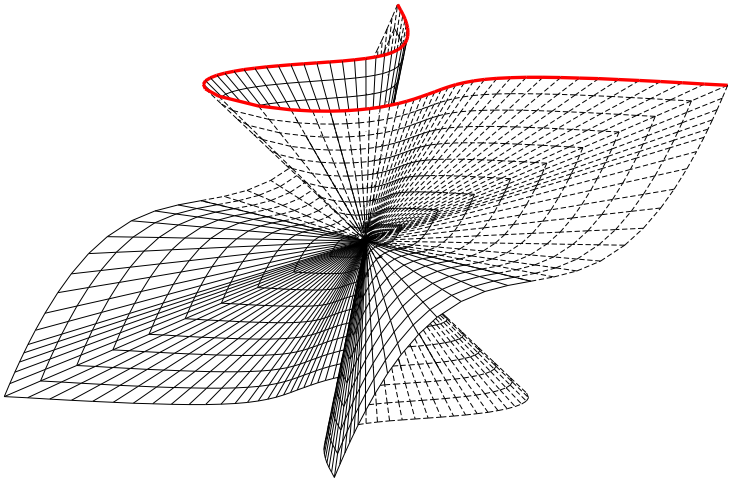
$$F(X, Y, Z) = Y^2Z + a_1XYZ + a_3YZ^2 - X^3 - a_2X^2Z - a_4XZ^2 - a_6Z^3,$$

to pochodne cząstkowe $\partial F/\partial X$, $\partial F/\partial Y$ i $\partial F/\partial Z$ nie są jednocześnie równe zero w żadnym punkcie krzywej. □

Rysunek 2.12 ilustruje krzywą $Y^2Z = X^3 - XZ^2 + Z^3$ w rzeczywistej przestrzeni rzutowej. Na rysunku kolorem czerwonym został zaznaczony jej rzut na płaszczyznę afiniczną. Należy w tym miejscu zaznaczyć, że niezależnie od tego w jaki sposób przeprowadzimy rzutowanie, nigdy afiniczna krzywa eliptyczna nie będzie odwzorowywała wszystkich punktów rzutowej krzywej. Z sytuacją zgoła odmienną mamy do czynienia w przypadku stożkowych, dla których okrąg i elipsa są przykładami krzywych odwzorowujących wszystkie punkty stożka rzutowego.

Gdy opisywaliśmy krzywą eliptyczną w postaci afinicznej musieliśmy wprowadzić dodatkowy punkt, który nazwany został punktem w nieskończoności. Przejście do postaci rzutowej krzywej daje nam możliwość określenia, gdzie dokładnie ów punkt się znajduje. Na Rysunku 2.13 został on oznaczony kolorem czerwonym. Zwróćmy uwagę, że jest to jedyny punkt krzywej rzutowej, który nie pojawia się na poziomej płaszczyźnie przecinającej krzywą $Y^2Z = X^3 - XZ^2 + Z^3$ – ma on współrzędne $[0, 1, 0]$.

Skoro jesteśmy w stanie określić współrzędne wszystkich punktów rzutowych krzywej eliptycznej, to wykorzystamy twierdzenie Bézout, aby pokazać, że każda prosta przechodząca przez dwa punkty krzywej, przecina krzywą również w trzecim punkcie. W tym celu przyjmijmy, że krzywa określona jest nad pewnym ciałem K, dla którego $\overline{K}$ jest domknięciem algebraicznym. Na mocy

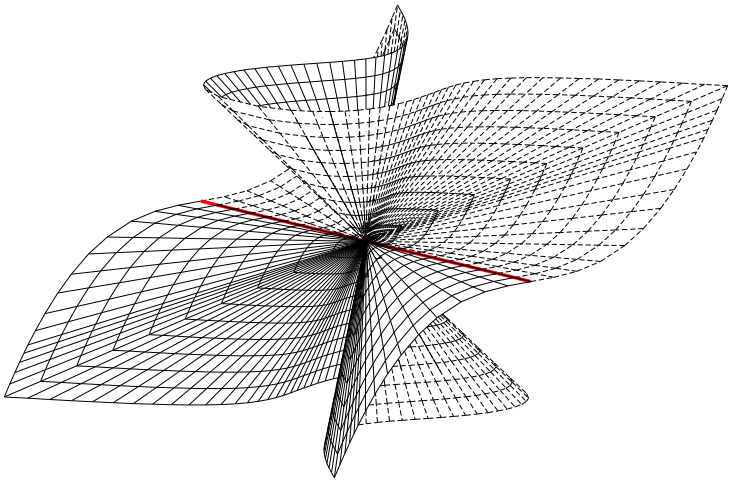


Rysunek 2.12: Krzywa eliptyczna $Y^2Z = X^3 - XZ^2 + Z^3$ wraz z oznaczoną projekcją na krzywą afiniczną $Y^2 = X^3 - X + 1$

twierdzenia Bézout wiemy, że dowolna prosta $aX + bY + cZ \in P^2(K)$ przecina krzywą eliptyczną (krzywą stopnia trzy) w przestrzeni rzutowej dokładnie w trzech punktach, których współrzędne należą do $\overline{K}$. Dwa z tych punktów są K-wymierne, gdyż są to punkty należące do C'. Niech $P_1 = [x_1, y_1, z_1]$ i $P_2 = [x_2, y_2, z_2]$ będą tymi punktami. Korzystając z równania prostej eliminujemy X lub Y z równania krzywej C'

$$Y^2Z + a_1XYZ + a_3YZ^2 = X^3 + a_2X^2Z + a_4XZ^2 + a_6Z^3.$$

W ten sposób otrzymujemy jednorodny wielomian stopnia 3 nad K, który ma dwa pierwiastki K-wymierne. W związku z tym rozkłada się na czynniki liniowe, a każdy czynnik zawiera jedynie elementy K-wymierne. Stąd wniosek, że współrzędne trzeciego punktu również należą do ciała K.



Rysunek 2.13: Krzywa eliptyczna $Y^2Z = X^3 - XZ^2 + Z^3$ wraz z oznaczonym punktem w nieskończoności

Rozdział 3

Kryptografia oparta na krzywych eliptycznych

Jak pokazaliśmy w poprzednim rozdziale, punkty krzywej eliptycznej tworzą grupę. W związku z tym można sobie zadać pytanie, czy trudno jest dla pewnych punktów $G, H \in E(K)$ znaleźć taką liczbę całkowitą x, że

$$[x]G = H.$$

Jest to problem logarytmu dyskretnego w grupie punktów krzywej eliptycznej. Okazuje się, że zadanie to jest znacznie trudniejsze niż analogiczny problem w grupie multiplikatywnej ciała skończonego. Poniższy przykład pokazuje, że trudność rozwiązania takiego równania w przypadku różnych grup może być diametralnie różna.

Przykład 3 Rozważmy trzy grupy $\mathbb{Z}/p\mathbb{Z}$, $\mathbb{F}_p^\times$ i $E(\mathbb{F}_p)$.

1. W przypadku grupy $\mathbb{Z}/p\mathbb{Z}$ problem ma postać

   $$xG = H \bmod p,$$

   a jego rozwiązanie jest trywialne, gdyż wyznaczenie rozwiązania postaci $x = HG^{-1} \bmod p$ nie nastręcza dużych trudności obliczeniowych. Złożoność obliczeniowa tego zadania w przypadku zastosowania rozszerzonego algorytmu Euklidesa wynosi:

   $$O(\log^2 p).$$

2. W przypadku grupy $\mathbb{F}_p^\times$ problem ma postać

   $$G^x = H \bmod p.$$

   To zadanie jest już znacznie trudniejsze od poprzedniego, a najlepszy znany algorytm znajdowania wartości x nosi nazwę metody Indeksu i ma złożoność

   $$O\left(\exp\left(c_0(\log p)^{1/3}(\log\log p)^{2/3}\right)\right).$$

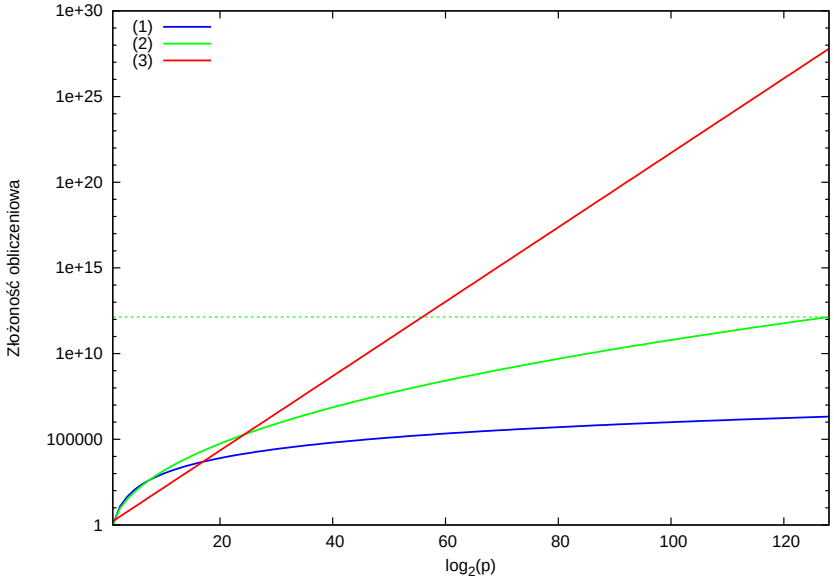
3. W przypadku grupy punktów na krzywej $E(\mathbb{F}_p)$ problem ma postać

$$[x]G = H.$$

Najlepszy znany algorytm znajdowania wartości x dla dowolnej krzywej opiera się na metodzie ρ-Pollarda lub metodzie *małych i dużych kroków*. Złożoność obliczeniowa obu algorytmów jest taka sama i wynosi

$$O\left(\exp\left(\frac{1}{2}\log p\right)\right).$$

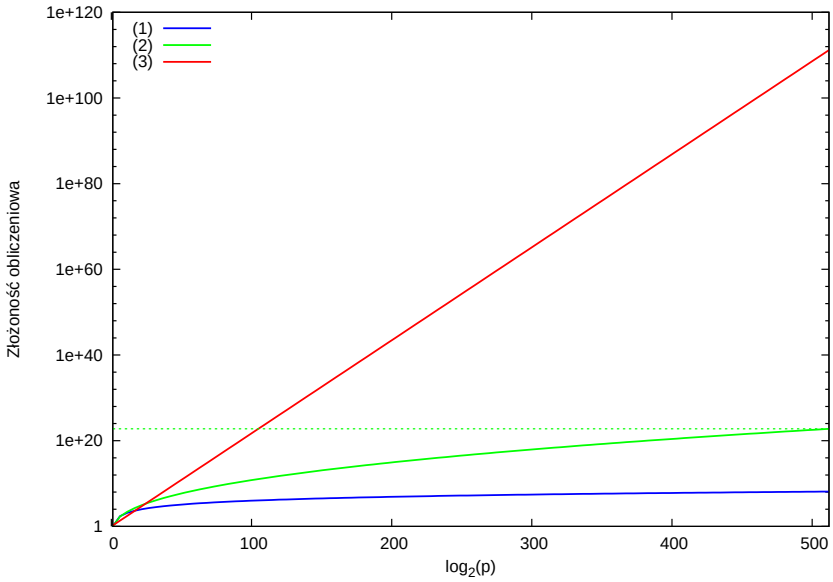
W przypadku niektórych krzywych problem ten można efektywnie zredukować do problemu logarytmu dyskretnego w grupie multiplikatywnej ciała skończonego. Dlatego też do zastosowań kryptograficznych wybierane są jedynie te krzywe, które spełniają określone warunki dotyczące bezpieczeństwa.



Rysunek 3.1: Graficzna interpretacja złożoności trzech przykładowych problemów logarytmu dyskretnego, $\log_2 p$ zmienia się od 1 do 128

Warto przy tej okazji zaznaczyć, że metoda ρ-Pollarda oraz metoda małych i dużych kroków są algorytmami generycznymi. Oznacza to, że można je zastosować w przypadku dowolnej grupy. Ich złożoność dla grupy n-elementowej wyraża się wzorem $O\left(\exp\left(\frac{1}{2}\log n\right)\right)$. W praktyce jest to równoważne stwierdzeniu, że nie istnieje grupa skończona, dla której problem logarytmu dyskretnego byłby trudniejszy. □

Na Rysunkach 3.1 i 3.2 przedstawiono wykresy pokazujące jak zmienia się złożoność problemu logarytmu dyskretnego dla grup przedstawionych w przykładzie. Jak widać złożoność tego problemu na krzywej eliptycznej rośnie nieporównywalnie szybciej niż w przypadku grupy multiplikatywnej ciała skończonego. Ma to swoje bezpośrednie odbicie w długościach kluczy kryptograficznych wykorzystywanych przez poszczególne systemy – klucze dla krzywych eliptycznych mogą być znacznie krótsze. Większość algorytmów kryptograficznych, które bazują



Rysunek 3.2: Graficzna interpretacja złożoności trzech przykładowych problemów logarytmu dyskretnego, $\log_2 p$ zmienia się od 1 do 512

na krzywych eliptycznych została stworzona z myślą o możliwości zastosowania dowolnej grupy, dla której problem logarytmu dyskretnego jest obliczeniowo trudny. Oczywiście są pewne różnice w implementacji tych ogólnych algorytmów dla poszczególnych grup, ale wynikają one raczej z potrzeby odpowiedniej konwersji elementów danej grupy.

3.1 Schemat wymiany kluczy Diffiego-Hellmana

Jest to niewątpliwie pierwszy publicznie opisany algorytm kryptografii asymetrycznej, to jest takiej, która rozróżnia pojęcia klucza prywatnego i publicznego. W swojej pierwotnej formie został on zaproponowany dla grupy multiplikatywnej ciała skończonego i opublikowany przez Whitfielda Diffiego i Martina Hellmana [14]. Algorytm ten jest bardzo ważny z punktu widzenia rozwoju kryptografii. Można powiedzieć, że zrewolucjonizował podejście do bezpiecznej

komunikacji elektronicznej, gdyż umożliwia bezpieczną wymianę kluczy przy użyciu kanału narażonego na podsłuchanie.

Protokół wymiany klucza metodą Diffiego-Hellmana zakłada, że strony A i B mają uzgodnioną pewną publicznie znaną grupę cykliczną G, która spełnia następujące warunki:

1. $\langle g \rangle = G$,
2. $|G| = n$,
3. rozwiązanie równania $[x]g = h$ jest obliczeniowo trudne,
4. istnieje funkcja $KDF_G : G \to \{0,1\}^*$, która pozwala przekształcić dowolny element grupy w ciąg bitów o zadanej długości.

Aby uzgodnić klucz sesyjny strony A i B realizują następujący protokół.

1. Strona A wybiera losową liczbę całkowitą $x_A \in \{0, \ldots, n-1\}$ i przesyła stronie B element
$$h_A = [x_A]g.$$
2. Strona B wybiera losową liczbę całkowitą $x_B \in \{0, \ldots, n-1\}$ i przesyła stronie A element
$$h_B = [x_B]g.$$
3. Strona A po otrzymaniu elementu h_B wyznacza element
$$h = [x_A]h_B = [x_A x_B]g$$
i oblicza wspólny klucz $k = KDF_G(h)$.
4. Strona B po otrzymaniu elementu h_A wyznacza element
$$h = [x_B]h_A = [x_A x_B]g$$
i oblicza wspólny klucz $k = KDF_G(h)$.

Podczas realizacji protokołu Diffiego-Hellmana w kanale publicznym pojawiają się jedynie takie wartości, jak $G, g, h_A = [x_A]g$ i $h_B = [x_B]g$. Aby złamać protokół należałoby wyznaczyć wartość
$$h = [x_A x_B]g$$
na podstawie danych przesłanych kanałem publicznym. Oczywiście, jeżeli problem logarytmu dyskretnego w grupie G jest łatwy do rozwiązania, to atakujący może wyznaczyć wartości x_A, x_B i na ich podstawie obliczyć h. Powyższe rozumowanie możemy zapisać w postaci następującego twierdzenia.

Twierdzenie 2 Jeżeli problem logarytmu dyskretnego w cyklicznej grupie G o n elementach ma złożoność $O(f(n))$, to algorytm łamiący schemat Diffiego-Hellmana w tej grupie ma złożoność co najwyżej $O(f(n))$. ■

Problem złamania schematu Diffiego-Hellmana w grupie G jest bardzo powszechny w literaturze kryptograficznej i formułowany następująco:

> *Problem Diffiego-Hellmana (DHP)* – Jest to problem znalezienia takiego elementu $h \in G$, że
> $$h = g^{ab},$$
> dla pewnych $g, g^a, g^b \in G$ i pewnej grupy skończonej G o zapisie multiplikatywnym.

Z punktu widzenia bezpieczeństwa warto byłoby dysponować twierdzeniem pokazującym równoważność obu problemów. Niestety w przypadku ogólnym dysponujemy jedynie twierdzeniem, że problem logarytmu dyskretnego (DLP) jest wielomianowo redukowalny do problemu Diffiego-Hellmana (DHP). Redukowalność odwrotna została jak do tej pory udowodniona jedynie w pewnych określonych przypadkach. Najistotniejszy jest tutaj wynik Maurera i Wolfa [34, 35, 36, 37, 38]. Pokazali oni, że dla dowolnej grupy cyklicznej G rzędu p problem DLP jest równoważny problemowi DHP, jeżeli istnieje *pomocnicza krzywa eliptyczna* nad ciałem $\mathbb{F}_p$, której rząd jest liczbą gładką (dzieli się tylko przez małe liczby pierwsze).

Wyniki Maurera i Wolfa zostały w praktyce wykorzystane przez Muzereau, Smarta i Vrecauterena [45] dla grupy punktów krzywej eliptycznej. Muzereau i pozostali wykazali, że takie pomocnicze krzywe eliptyczne można z dużym prawdopodobieństwem znaleźć dla niemal wszystkich grup opartych na krzywej eliptycznej. Ich publikacja podkreśla jednak, że staje się to niezmiernie trudne wraz ze wzrostem liczby punktów na krzywej eliptycznej. Niemniej jednak Muzereau, Smart i Vrecauteren zdołali skonstruować pomocnicze krzywe eliptyczne dla niemal wszystkich dziedzin określonych w standardzie SECG [64]. W praktyce oznacza to tyle, że dla większości krzywych eliptycznych określonych aktualnie w normach istnieje dowód równoważności problemu logarytmu dyskretnego i problemu Diffiego-Hellmana.

3.2 Schemat szyfrowania ECIES

Schemat szyfrowania ECIES (Elliptic Curve Integrated Encryption Scheme) [63] jest tak naprawdę statyczną wersją protokołu Diffiego-Hellmana. Oznacza to, że uzgadnianie klucza nie odbywa się przy aktywnym udziale obu stron protokołu. W praktyce sprowadza się to do tego, że jedna ze stron (na przykład strona A) udostępnia swój klucz publiczny wszystkim, którzy chcieliby wymienić z nią informacje w sposób bezpieczny.

Jeżeli grupa G spełnia wymagania nałożone na nią przy okazji protokołu Diffiego-Hellmana, a strona A wygenerowała klucz prywatny $x_A \in \{0, \ldots, n-1\}$ i udostępniła swój klucz publiczny

$$h_A = [x_A]g,$$

to przebieg protokołu ECIES jest następujący:

1. Strona B wybiera losową liczbę $x_B \in \{0, \ldots, n-1\}$, tworzy na jej podstawie klucz publiczny $h_B = [x_B]g$ i wyznacza klucz sesyjny

$$k = KDF_G([x_B]h_A) = KDF_G([x_A x_B]g).$$

2. Strona B wykorzystuje klucz k do zaszyfrowania wiadomości m przy użyciu algorytmu symetrycznego E i wysyła stronie A szyfrogram wraz ze swoim kluczem publicznym h_B:

$$C = (h_B, c = E_k(m)).$$

3. Strona A odbiera szyfrogram i odtwarza klucz symetryczny k:

$$k = KDF_G([x_A]h_B) = KDF_G([x_A x_B]g).$$

4. Strona A odtwarza zaszyfrowaną wiadomość $m = E_k^{-1}(c)$.

Algorytm ECIES cieszy się sporą popularnością przede wszystkim ze względu na bardzo dużą powszechność wykorzystania protokołu Diffiego-Hellmana. Z przebiegu protokołu wynika bowiem jasno, że wszystkie systemy implementujące protokół ECDH (Elliptic Curve Diffie-Hellman) można w bardzo prosty sposób dostosować do obsługi szyfrowania ECIES. Jest to szczególnie ważne w systemach dysponujących niewielkimi zasobami pamięciowymi.

3.3 Podpis cyfrowy ElGamala i standard DSS

W oparciu o schematy szyfrowania i wymiany kluczy nie da się zbudować w pełni funkcjonalnego systemu kryptografii asymetrycznej. Do tego potrzebny jest jeszcze jeden element – podpis cyfrowy, czyli mechanizm gwarantujący autentyczność przesłanych danych. W roku 1985 taki mechanizm został zaproponowany przez ElGamala [15] dla dowolnej grupy, w której problem logarytmu dyskretnego jest obliczeniowo trudny.

Schemat podpisu ElGamala zakłada, że znana jest publicznie pewna cykliczna grupa $G = \langle g \rangle$ rzędu n, dla której określono efektywną obliczeniowo bijekcję $f : G \to \mathbb{Z}/n\mathbb{Z}$. Dla tak przyjętych oznaczeń schemat podpisu ElGamala realizowany jest w następujący sposób:

1. Strona A wybiera losowo liczbę $x \in \{0, \ldots, n-1\}$, która będzie kluczem prywatnym, a następnie udostępnia element

$$h = [x]g,$$

który jest kluczem publicznym i będzie służył innym stronom do weryfikacji podpisu złożonego przez A.

2. Aby podpisać wiadomość $m \in \mathbb{Z}/n\mathbb{Z}$ strona A wybiera losowo liczbę $k \in \{1, \ldots, n-1\}$ i wyznacza
$$a = [k]g.$$

3. W kolejnym kroku A znajduje taką liczbę $b \in \mathbb{Z}/n\mathbb{Z}$, że
$$b = (m - xf(a))k^{-1} \bmod n.$$

4. Strona A publikuje podpis $(a, b) \in G \times (\mathbb{Z}/n\mathbb{Z})$ pod wiadomością m.

5. Weryfikacja podpisu wykonywana jest poprzez sprawdzenie prawdziwości następującej równości
$$[f(a)]h + [b]a = [m]g.$$
Należy zauważyć, że w przypadku, gdy podpis jest poprawny, to lewa strona równości może być przedstawiona jako $[f(a)]h + [b]a = [xf(a)]g + [kb]g = [xf(a) + kb]g = [m]g$.

Pewną odmianą schematu podpisu ElGamala jest algorytm DSA (Digital Signature Algorithm), który stanowi podstawę standardu DSS [62]. Został on stworzony przez NIST (National Institute of Standards and Technology) w 1991 roku, jako alternatywa dla objętego patentami algorytmu RSA (patent wygasł w 2000 roku). Trzeba w tym miejscu podkreślić, że algorytm DSA jest również opatentowany, ale może być wykorzystywany w implementacjach komercyjnych bez konieczności wnoszenia jakichkolwiek opłat. W terminologii kryptograficznej nazwy DSA używa się w odniesieniu do algorytmu działającego w oparciu o grupę multiplikatywną ciała skończonego, natomiast jego odmiana dla grupy punktów krzywej eliptycznej określana jest mianem ECDSA (Elliptic Curve Digital Signature Algorithm). Poniżej znajduje się opis działania protokołu DSA i ECDSA.

1. Generowanie podpisu.
   (a) Strona A wybiera losowo liczbę $x \in \{0, \ldots, n-1\}$, która będzie kluczem prywatnym, a następnie udostępnia element
   $$h = [x]g,$$
   który jest kluczem publicznym i będzie służył innym stronom do weryfikacji podpisu złożonego przez A.
   (b) Aby podpisać wiadomość $m \in \mathbb{Z}/n\mathbb{Z}$ strona A wybiera losowo liczbę $k \in \{1, \ldots, n-1\}$ i wyznacza
   $$a = [k]g.$$
   (c) W kolejnym kroku A znajduje taką liczbę $b \in \mathbb{Z}/n\mathbb{Z}$, że
   $$b = (m + xf(a))k^{-1} \bmod n.$$

(d) Strona A publikuje podpis $(a, b) \in G \times (\mathbb{Z}/n\mathbb{Z})$ pod wiadomością m.

2. Weryfikacja podpisu.

   (a) Strona B w celu weryfikacji podpisu (a, b) pod wiadomością m wyznacza najpierw liczby $u, v \in \mathbb{Z}/n\mathbb{Z}$ takie, że

$$\begin{aligned} u &= mb^{-1} \bmod n, \\ v &= f(a)b^{-1} \bmod n. \end{aligned}$$

   (b) Strona B sprawdza poprawność podpisu poprzez weryfikację warunku

$$[u]g + [v]h = a.$$

   Jeśli podpis jest poprawny, to powyższa równość zachodzi, gdyż

$$\begin{aligned} [u]g + [v]h &= [mb^{-1}]g + [vx]g = [mb^{-1} + xf(a)b^{-1}]g \\ &= [(m + xf(a))b^{-1}]g = [kbb^{-1}]g = [k]g \\ &= a. \end{aligned}$$

Można odnieść wrażenie, że algorytm DSA jest bardziej skomplikowany niż algorytm zaproponowany przez ElGamala. Jeśli jednak przyjrzymy się obu schematom dokładniej, to zobaczymy zasadniczą różnicę w sposobie weryfikacji podpisu. Algorytm DSA wymaga wyznaczenia tylko dwóch krotności elementów w grupie G, podczas gdy ElGamal trzech. Ponieważ operacje te stanowią główny ciężar obliczeniowy procesu weryfikacji, to można stwierdzić, że weryfikacja podpisu DSA jest o około 33% szybsza od weryfikacji ElGamala.

Rozdział 4

Krzywe eliptyczne silne kryptograficznie i ich generowanie

W poprzednim rozdziale pokazaliśmy, że problem logarytmu dyskretnego może mieć różny stopień złożoności obliczeniowej (w niektórych grupach jest łatwiejszy, w innych trudniejszy). Podczas prac nad algorytmami wykorzystującymi krzywe eliptyczne okazało się, że również nie każda krzywa gwarantuje odpowiedni poziom bezpieczeństwa. Dlatego też w tym rozdziale skoncentrujemy swoją uwagę na analizie warunków jakie muszą spełniać krzywe stosowane w kryptografii oraz na sposobach generowania takich krzywych.

W zastosowaniach kryptograficznych wykorzystuje się krzywe zdefiniowane nad ciałem skończonym charakterystyki 2 lub nad ciałem prostym charakterystyki $p > 3$. W przypadku ciał charakterystyki 2 istnieją bardzo szybkie metody zliczania punktów krzywej [49, 57, 17, 60, 25] i stwierdzania, czy nadaje się do zastosowań kryptograficznych. Dlatego też w dalszych rozważaniach ograniczymy się jedynie do krzywych zdefiniowanych nad prostymi ciałami skończonymi charakterystyki $p > 3$. Dla tej klasy krzywych czas potrzebny na znalezienie odpowiedniej krzywej jest dość długi, a proponowany w pracy algorytm znacznie przyspiesza obliczenia w fazie wstępnej algorytmu zliczania punktów.

4.1 Krzywe eliptyczne nad ciałami skończonymi charakterystyki $p > 3$

Jak już pisaliśmy w części 2.1 równanie Weierstrassa 2.1 można w ciałach charakterystyki różnej od 2 i 3 uprościć do postaci

$$E: \qquad Y^2 = X^3 + aX + b. \tag{4.1}$$

Dla takiej krzywej wyróżnik i j-niezmiennik upraszczają się do postaci

$$\Delta(E) = -16(4a^3 + 27b^2), \tag{4.2}$$

$$j(E) = 1728\frac{4a^3}{4a^3 + 27b^2}. \tag{4.3}$$

W przypadku krzywej o równaniu 4.1 uproszczeniu ulega również definicja izomorfizmu dwóch krzywych eliptycznych.

Definicja 4 Dwie krzywe eliptyczne

$$E_1(\mathbb{F}_p): \quad Y^2 = X^3 + a_1X + b_1,$$
$$E_2(\mathbb{F}_p): \quad Y^2 = X^3 + a_2X + b_2,$$

są izomorficzne wtedy i tylko wtedy, gdy istnieje taki element $u \in \mathbb{F}_p^\times$, dla którego zachodzi

$$a_2 = u^4a_1,$$
$$b_2 = u^6b_1.$$

□

Oczywiście w ciele skończonym $\mathbb{F}_p$ istnieje tylko skończona liczba rozwiązań równania E. Oznacza to, że każda krzywa posiada w tym ciele tylko skończoną liczbę punktów wymiernych. Liczbę tą oznaczamy przez $\#E(\mathbb{F}_p)$ i zapisujemy w postaci

$$\#E(\mathbb{F}_p) = p + 1 - t,$$

gdzie t jest nazywane *śladem Frobeniusa* dla p.

Ślad Frobeniusa ma ścisły związek z *odwzorowaniem Frobeniusa* stopnia p, które przekształca krzywą $E(\overline{\mathbb{F}}_p)$ w siebie. Przekształcenie to definiujemy jako

$$\varphi(X, Y) = (X^p, Y^p),$$

dla $(X, Y) \neq \mathcal{O}$ i $\varphi(\mathcal{O}) = \mathcal{O}$. Ponieważ podnoszenie do p-tej potęgi jest endomorfizmem ciała $\overline{\mathbb{F}}_p$, to zachowuje wszystkie działania wymierne. Skoro więc działania na punktach krzywej zdefiniowane są za pomocą funkcji wymiernych, to odwzorowanie Frobeniusa jest w rzeczywistości endomorfizmem krzywej $E(\overline{\mathbb{F}}_p)$ i z tego tytułu nazywane jest również *endomorfizmem Frobeniusa*.

Endomorfizm Frobeniusa i jego ślad t są bardzo istotne z punktu widzenia krzywych eliptycznych. Okazuje się bowiem, że łączy je następujący związek

$$\varphi^2 - [t]\varphi + [p] = [0].$$

Równanie to oznacza tak naprawdę, że dla każdego punktu krzywej eliptycznej $(X, Y) \in E(\overline{\mathbb{F}}_p)$ zachodzi równość

$$(X^{p^2}, Y^{p^2}) - [t](X^p, Y^p) + [q](X, Y) = \mathcal{O}.$$

Dodawanie i odejmowanie oznaczają w powyższym wzorze działanie grupowe na punktach krzywej eliptycznej.

Z łatwością można zauważyć, że liczba punktów na krzywej $E(\mathbb{F}_p)$ nie może być zbyt duża. Jeżeli bowiem będziemy podstawiali kolejne elementy ciała $\mathbb{F}_p$ pod zmienną X, to za każdym razem możemy uzyskać co najwyżej dwa rozwiązania równania ze względu na Y. W związku z tym liczba punktów krzywej nie może być większa niż $2p + 1$ (doliczamy punkt w nieskończoności). Z drugiej strony możemy się spodziewać, że rozwiązanie równania ze względu na Y będzie istniało *średnio* w co drugim przypadku. Zatem oczekiwana liczba punktów powinna być w przybliżeniu równa p. Ta intuicja statystyczna okazuje się być bardzo trafna. Potwierdza ją udowodnione przez Hassego następujące twierdzenie, którego dowód można znaleźć w książce Silvermana [55].

Twierdzenie 3 Ślad Frobeniusa spełnia nierówność

$$|t| \leq 2\sqrt{p}.$$

■

Twierdzenie Hassego mówi nam, że liczba punktów każdej krzywej $E(\mathbb{F}_p)$ zawiera się w wąskim przedziale $(p + 1 - 2\sqrt{p}, p + 1 + 2\sqrt{p})$. Można udowodnić, że jeśli tylko $p > 2$, to dla każdej liczby naturalnej n z tego przedziału istnieje krzywa eliptyczna mająca dokładnie n punktów. Ponadto prawdopodobieństwo tego, że losowo wybrana krzywa będzie miała rząd równy n jest w przybliżeniu równe $\frac{1}{4\sqrt{p}}$. Obserwacja ta jest podstawą algorytmu Lenstry [33], który służy do faktoryzacji liczb całkowitych.

Skoro liczbę punktów krzywej wyrażamy poprzez zależność $p + 1 - t$, to powstaje naturalne pytanie, czy można jakoś scharakteryzować krzywe, dla których ślad Frobeniusa ma wartość $-t$. Okazuje się, że jest to możliwe przy pomocy następującego pojęcia.

Definicja 5 *Skręceniem* krzywej $E(\mathbb{F}_p)$ nazywamy taką krzywą

$$E'(\mathbb{F}_p): \qquad Y^2 = X^3 + a'X + b',$$

że

$$\begin{aligned} a' &= v^2 a, \\ b' &= v^3 b, \end{aligned}$$

dla pewnego v, które nie jest resztą kwadratową modulo p. □

Związek między krzywą, a jej skręceniem określa następujące twierdzenie

Twierdzenie 4 Jeżeli $E'(\mathbb{F}_p)$ jest skręceniem krzywej $E(\mathbb{F}_p)$, to

$$\#E(\mathbb{F}_p) + \#E'(\mathbb{F}_p) = 2p + 2.$$

■

4.2 Krzywe eliptyczne stosowane w kryptografii

Gdy chcemy wykorzystać krzywą do celów kryptograficznych pojawia się pytanie, czy problem logarytmu dyskretnego jest dla niej odpowiednio trudny? Bowiem z faktu, że dana klasa problemów jest trudna nie wynika, że konkretna instancja również będzie trudna. Za przykład niech nam tutaj posłuży problem faktoryzacji liczb całkowitych. Wiemy, że w ogólności jest to zadanie złożone, ale jeśli ktoś każe nam rozłożyć liczbę 1 000 000 000 000 000 na czynniki pierwsze to bez trudu wykonamy to zadanie w pamięci:

$$1\ 000\ 000\ 000\ 000\ 000 = 2^{15} \cdot 5^{15}.$$

Podobnie sytuacja wygląda z problemem logarytmu dyskretnego na krzywej eliptycznej. W ogólnym przypadku problem ten jest obliczeniowo trudny, ale istnieją takie klasy krzywych, dla których jest on łatwiejszy. Aktualnie za krzywe kryptograficznie słabe uważa się krzywe *anomalne* i *superosobliwe*. Wynika to z faktu, że na te rodzaje krzywych istnieją ataki o niewielkiej złożoności obliczeniowej.

Definicja 6 Mówimy, że krzywa $E(\mathbb{F}_p)$ jest *krzywą anomalną*, jeżeli jej ślad Frobeniusa jest równy 1. W praktyce oznacza to, że rząd krzywej jest równy p. □

Definicja 7 Mówimy, że krzywa $E(\mathbb{F}_p)$ jest *krzywą superosobliwą*, jeżeli charakterystyka ciała dzieli ślad Frobeniusa. □

Krzywe superosobliwe charakteryzuje następujące twierdzenie

Twierdzenie 5 Krzywa eliptyczna $E(\mathbb{F}_{p^k})$ jest superosobliwa wtedy i tylko wtedy, gdy:

1. $p = 2$ i $j(E) = 0$,
2. $p = 3$ i $j(E) = 0$,
3. $p \geq 5$ i $t = 0$,

gdzie t jest śladem Frobeniusa. ■

Ataki na krzywe eliptyczne

Ataki na kryptosystemy oparte na krzywych eliptycznych to tak naprawdę algorytmy rozwiązujące problem logarytmu dyskretnego w grupie punktów krzywej. Przypomnijmy, że problem ten polega na znalezieniu takiej liczby całkowitej x, dla której prawdziwa jest równość

$$Q = [x]P,$$

dla pewnych $P, Q \in E(\mathbb{F}_p)$ takich, że $Q \in \langle P \rangle$. Okazuje się, że rozwiązanie problemu logarytmu dyskretnego (ECDLP) może być łatwe w przypadku niektórych klas krzywych. W związku z tym wiedza na temat pewnych własności krzywej jest niezmiernie istotna z punktu widzenia bezpieczeństwa systemu kryptograficznego. Z tego powodu opracowane zostały specjalne procedury, które pozwalają zweryfikować własności krzywej i ocenić jej przydatność. Procedury te biorą pod uwagę między innymi aktualny stan wiedzy na temat trudności problemu logarytmu dyskretnego w grupie punktów krzywej eliptycznej.

Aktualnie możemy wyróżnić cztery główne rodzaje algorytmów służących do rozwiązywania problemu ECDLP. Dwa z nich są podstawą do praktycznych ataków na konkretne klasy krzywych i prowadzą do poznania kluczy kryptograficznych. Kolejne dwa dają możliwość przeprowadzenia ataku generycznego o złożoności wykładniczej. W praktyce oznacza to tyle, że odpowiedni dobór krzywej eliptycznej zabezpiecza nas przed skutecznymi atakami na kryptosystem.

1. Atak Menezesa, Okamoto i Vanstone'a (określany w literaturze mianem ataku MOV) został opublikowany w roku 1993 [39], a następnie uogólniony Freya i Rücka [18]. Jego idea polega na przeniesieniu problemu logarytmu dyskretnego z grupy punktów krzywej eliptycznej $E(\mathbb{F}_p)$ do grupy multiplikatywnej $\mathbb{F}_{p^k}^{\times}$, gdzie możliwe jest zastosowanie metody indeksu. Podejście to jest bardzo skuteczne o ile tylko liczba k nie jest zbyt duża. Dla odpowiednio dużego k okazuje się, że złożoność ataku MOV przekracza złożoność ataku generycznego na grupę $E(\mathbb{F}_p)$. Z tego powodu podczas określania przydatności krzywej do celów kryptograficznych sprawdza się tak zwany *warunek MOV*. Spełnienie tego warunku daje gwarancję, że atak MOV będzie kosztowniejszy niż atak generyczny.

   Jeżeli liczba punktów krzywej wynosi $h \cdot n = \#E(\mathbb{F}_p)$, gdzie n jest rzędem generatora, to atak MOV przeprowadza się dla najmniejszej liczby k, dla której
   $$p^k = 1 \bmod n.$$
   Przy obecnie stosowanych wielkościach kluczy (do 512 bitów) wystarczające jest sprawdzenie, czy powyższa tożsamość nie jest spełniona dla $k \leq 20$.

2. Rodziną krzywych, dla których problem logarytmu dyskretnego jest bardzo prosty są krzywe anomalne. Są to krzywe, których liczba punktów jest równa p (ślad endomorfizmu Frobeniusa jest równy 1). Atak ten jest bardzo skuteczny, co w praktyce eliminuje możliwość użycia anomalnych krzywych eliptycznych. Dokładny opis ataku można znaleźć w artykułach [58, 48, 54]. Ciekawostką jest natomiast fakt, że ataki te zostały opracowane po tym, jak Miyaji zaproponował wykorzystanie krzywych anomalnych do celów kryptograficznych. Związane było to z faktem, że

krzywe te są całkowicie odporne na atak MOV. W przypadku tych krzywych $n = p$ i dla każdej wartości k zachodzi nierówność $p^k \neq 1 \bmod n$. Niestety okazało się, że atak na krzywą anomalną ma znacznie mniejszą złożoność niż atak MOV.

3. Atak oparty o metodę małych i dużych kroków Shanksa jest atakiem generycznym, co oznacza, że może być stosowany w przypadku dowolnej grupy. Jego złożoność pamięciowa i obliczeniowa jest wykładnicza wynosi $\sqrt{n}$, gdzie n jest rzędem generatora.

4. Kolejnym rodzajem ataku generycznego jest metoda błądzenia przypadkowego. Ma ona taką samą złożoność obliczeniową jak metoda Shanksa, ale zapotrzebowanie pamięciowe jest stałe. Dodatkowym atutem tej metody jest możliwość jej efektywnego zrównoleglenia.

Procedura weryfikacji krzywej eliptycznej

Parametry krzywych eliptycznych stosowanych w kryptografii zostały tak wybrane, aby możliwa była ich prosta weryfikacja. O ile wygenerowanie krzywej jest procesem bardzo czasochłonnym i skomplikowanym, to już weryfikacja poprawności parametrów jest bardzo szybka. Poniżej przedstawiamy procedurę weryfikacji krzywej eliptycznej, która została zdefiniowana w standardach FIPS 186-3 [62] i ANSI X9.62 [61]. Ogromną zaletą tej procedury jest możliwość stwierdzenia, czy krzywa została wygenerowana w sposób losowy. Taka funkcjonalność ma ogromne znaczenie dla całej rzeszy użytkowników, którzy powinni mieć pewność, że krzywa dostarczona przez instytucję zewnętrzną nie została w jakiś sposób spreparowana.

Procedura weryfikująca możliwość wykorzystania krzywej eliptycznej do celów kryptograficznych korzysta z następujących oznaczeń:

$p > 3$ – liczba pierwsza określająca ciało, nad którym zdefiniowano krzywą,

E – krzywa eliptyczna zdefiniowana nad ciałem $\mathbb{F}_p$, która zadana jest równaniem $Y^2 = X^3 + aX + b$,

$G = (x_G, y_G)$ – generator grupy punktów na krzywej E,

n – rząd punktu G,

h – liczba warstw podgrupy $\langle G \rangle \subset E$, czyli $h \cdot n = \#E$,

s – ciąg bitów reprezentujący ziarno, na podstawie którego została wygenerowana krzywa E (jeżeli ciąg ten traktowany jest jak liczba, to bity leżące z lewej strony są najbardziej znaczącymi bitami reprezentacji binarnej).

Przy takich oznaczeniach procedura określona w normie FIPS 186-3 ma następujący przebieg:

1. Zweryfikuj, czy p jest liczbą pierwszą.

   Weryfikacja tego warunku jest bardzo istotna z punktu widzenia bezpieczeństwa. Można sobie bowiem bez trudu wyobrazić sytuację, kiedy ktoś podstawia nam krzywą dla której zachodzi $p = q \cdot r$, $(q, r) = 1$. Wtedy zachodzą również następujące relacje $\mathbb{Z}/p\mathbb{Z} \simeq (\mathbb{Z}/q\mathbb{Z}) \times (\mathbb{Z}/r\mathbb{Z})$ oraz $E(\mathbb{Z}/p\mathbb{Z}) \simeq E(\mathbb{Z}/q\mathbb{Z}) \times E(\mathbb{Z}/r\mathbb{Z})$. Izomorfizmy te pozwalają znacznie zredukować złożoność problemu logarytmu dyskretnego – atakujący będzie musiał rozwiązać dwa problemy w znacznie mniejszych strukturach algebraicznych.

2. Zweryfikuj, czy $a, b, x_G, y_G \in [0, p-1]$.

   Weryfikacja tego warunku nie jest konieczna, gdyż wszystkie operacje kryptograficzne prowadzone są modulo p i nawet, gdyby liczby były spoza wskazanego zakresu, to zostaną one zredukowane. Jest to raczej wymóg formalny, którego zadaniem jest ujednolicenie reprezentacji parametrów krzywej eliptycznej.

3. Zweryfikuj, czy współczynniki krzywej zostały wygenerowane na bazie podanego ziarna. Procedura weryfikacji losowości krzywej ma następujący przebieg:

   (a) Niech l będzie liczbą bitów liczby p, czyli $l = \lfloor \log_2 p \rfloor + 1$. Dla tak zdefiniowanego l wyznaczamy

   $$\begin{aligned} v &= \lfloor (l-1)/160 \rfloor, \\ w &= l - 160v - 1. \end{aligned}$$

   (b) Niech h_0 oznacza ciąg powstały z SHA-1(s) poprzez wzięcie w bitów z prawej strony.

   (c) Jeżeli przez $|s|$ oznaczymy liczbę bitów ciągu s, to dla $i \in [1, v]$ definiujemy $h_i = \text{SHA-1}((s + i) \bmod 2^{|s|})$.

   (d) Niech teraz r oznacza liczbę naturalną, której reprezentacja binarna ma postać ciągu $h_0||h_1||\ldots||h_v$.

   (e) Jeżeli zachodzi tożsamość

   $$rb^2 = a^3 \bmod p,$$

   to możemy stwierdzić, że krzywa została wygenerowana w sposób losowy.

   Warto w tym miejscu dodać kilka słów komentarza dotyczącego właśnie takiej definicji współczynników krzywej i powiązania ich z parametrem r. Jeśli cofniemy się do definicji izomorfizmu

krzywych zadanych równaniem $Y^2 = X^3 + aX + b$, to stwierdzimy, że stosunek a^3/b^2 dla wszystkich krzywych izomorficznych jest stały. Oznacza to tak naprawdę, że liczba r wskazuje krzywą E w sposób jednoznaczny z dokładnością do izomorfizmu. Widzimy zatem, że dla danego ziarna s wszystkie krzywe są sobie równoważne jako grupy (trudność złamania każdej z nich jest taka sama).

Drugą ważną cechą tej procedury jest zdefiniowanie współczynnika izomorficzności r nie bezpośrednio na podstawie s, a na podstawie skrótu z ziarna s. Daje to pewność, że najpierw zostało wygenerowane s, a dopiero później wartość r. Ze względu na zastosowanie funkcji skrótu nie ma bowiem możliwości odwrócenia tego procesu.

4. Zweryfikuj, czy krzywa jest nieosobliwa

$$\Delta(E) = 4a^3 + 27b^2 \neq 0 \bmod p.$$

Ten punkt weryfikuje, czy krzywa o podanych współczynnikach jest rzeczywiście krzywą eliptyczną.

5. Zweryfikuj, czy punkt G leży na krzywej E

$$y_G^2 = x_G^3 + ax_G + b \bmod p.$$

Sprawdzenie, czy punkt leży na krzywej jest bardzo ważne, gdyż daje nam pewność, że operacje wykonywane są rzeczywiście w grupie punktów krzywej.

6. Zweryfikuj, czy n jest liczbą pierwszą spełniającą warunki $n > 2^{160}$ i $n > 4\sqrt{p}$.

Pierwszość liczby n jest istotna, gdyż daje nam gwarancję co do wartości rzędu elementu G. Gdyby liczba n była złożona, to w celu weryfikacji rzędu G musielibyśmy znać jej dokładny rozkład na czynniki pierwsze. W przeciwnym razie wystarczy sprawdzić, czy $[n]G = \mathcal{O}$.

Warunek na to, by rząd generatora był większy od 2^{160} jest wymagany ze względu na bezpieczeństwo systemu kryptograficznego. Złożoność ataku na podgrupę rzędu 2^{160} wynosi 2^{80}, co było do niedawna przyjmowane za minimalny poziom bezpieczeństwa. Aktualnie wymaga się, aby nowe aplikacje gwarantowały co najmniej 128-bitowy poziom bezpieczeństwa. Oznacza to, że parametr n powinien spełniać nierówność $n > 2^{256}$.

Bardzo ciekawe jest natomiast trzecie wymaganie, czyli spełnienie nierówności $n > 4\sqrt{p}$. Zauważmy, że z twierdzenia Hassego wynika, że liczba punktów na krzywej E mieści się w przedziale $(p+1-2\sqrt{p}, p+1+2\sqrt{p})$. Oznacza to, że istnieje tylko jedna liczba całkowita h' taka, że iloczyn $h'n$ należy do tego przedziału. Daje nam to więc możliwość jednoznacznego określenia liczby punktów krzywej bez konieczności ich zliczania (co jest bardzo czasochłonną i skomplikowaną operacją).

7. Zweryfikuj, czy punkt G jest rzędu n

$$[n]G = \mathcal{O}.$$

Warunek ten wraz z pierwszością liczby n gwarantuje, że rząd punktu G będzie równy dokładnie n. Przypomnijmy w tym miejscu, że G musi być punktem skończonym, gdyż jego współrzędne spełniają równanie krzywej E w postaci afinicznej.

8. Wyznacz $h' = \lfloor (\sqrt{p}+1)^2/n \rfloor$ i sprawdź, czy $h = h'$.

Zauważmy, że liczba $h' = \lfloor (\sqrt{p}+1)^2/n \rfloor$ jest największą liczbą całkowitą, dla której zachodzi nierówność $h'n < p+1+2\sqrt{p}$. Ponadto z nierówności $n > 4\sqrt{p}$ wynika zależność $(h'-1)n < p+1-2\sqrt{p}$. Ponieważ n jest dzielnikiem rzędu krzywej (jest rzędem punktu G), to z twierdzenia Hassego wynika, że h' jest równa liczbie warstw grupy $\langle G \rangle$ i powinna być równa liczbie h.

9. Zweryfikuj, czy krzywa nie jest anomalna.

10. Zweryfikuj, czy krzywa jest odporna na atak MOV.

11. Odrzuć możliwość wykorzystania krzywej E jeżeli nie zachodził którykolwiek z warunków opisanych powyżej.

Zweryfikowanie krzywej eliptycznej w oparciu o powyższą procedurę z jednej strony jest bardzo proste do przeprowadzania, a z drugiej gwarantuje losowość krzywej oraz prawdziwość wszystkich jej parametrów. Jest to bardzo ważne dla przeciętnego użytkownika systemu kryptograficznego. Wykonanie weryfikacji krzywej daje mu bowiem całkowitą pewność, że parametry nie zostały spreparowane pod kątem ułatwienia ataku na kryptosystem.

4.3 Wielomiany podziału i algorytm Schoofa

Część tą rozpocznijmy od zapisania równania krzywej w postaci

$$E: \qquad Y^2 = g(X) = X^3 + aX + b.$$

Jeśli przyjrzeć się wielomianowi $g(X)$, to może on mieć w ciele $\mathbb{F}_p$: 0, 1 lub 3 pierwiastki. Każdy z tych pierwiastków definiuje jeden punkt krzywej o rzędnej równej 0. Jeżeli więc $g(X)$ ma

– 0 pierwiastków, to rząd krzywej eliptycznej nie jest podzielny przez 2 (brak punktów rzędu 2),

– 1 pierwiastek, to rząd krzywej jest podzielny przez 2 (istnieje jeden punkt rzędu 2),

– 3 pierwiastki, to rząd krzywej jest podzielny przez 4 (istnieją dwa niezależne punkty rzędu 2, trzeci punkt jest ich sumą).

Jak widzimy analiza pierwiastków wielomianu $g(X)$ daje nam częściową informację o rzędzie krzywej eliptycznej. Rodzi się naturalne pytanie, czy możemy takich informacji zebrać na tyle dużo, aby określić rząd krzywej? Okazuje się, że odpowiedź na to pytanie jest twierdząca, a po raz pierwszy udzielił jej René Schoof [50, 51].

Zanim jednak przejdziemy do opisu pierwszego efektywnego algorytmu zliczania punktów krzywej eliptycznej, przyjrzyjmy się operacji zwielokrotniania punktu

$$(X, Y) \mapsto [m](X, Y).$$

Jak widzieliśmy w części 2.2 operacja dodawania punktów $P + Q$ zdefiniowana jest jako funkcja wymierna ich współrzędnych. Oznacza to, że krotność punktu $[m](X, X)$ możemy przedstawić jako funkcję wymierną X i Y. Poniższy lemat pokazuje jaka jest postać tych funkcji wymiernych.

Lemat 2 Przyjmijmy, że dana jest krzywa E określona nad ciałem $\mathbb{F}_p$, a m jest dodatnią liczbą całkowitą. Jeżeli $P(X, Y) \in E(\overline{\mathbb{F}}_p)$ i $[m]P \neq \mathcal{O}$, to istnieją wielomiany $\psi_m, \theta_m, \omega_m \in \mathbb{F}_p[X, Y]$ takie, że

$$[m]P = \left(\frac{\theta_m(X, Y)}{\psi_m(X, Y)^2}, \frac{\omega_m(X, Y)}{\psi_m(X, Y)^3} \right).$$

■

Definicja 8 Wielomian $\psi_m(X, Y)$ nazywamy *m-tym wielomianem podziału* krzywej E. □

Wielomiany $\psi_m, \theta_m, \omega_m$ są przedstawione jako wielomiany dwóch zmiennych. Musimy jednak pamiętać, że współrzędne X, Y są punktami krzywej, więc muszą spełniać jej równanie $Y^2 = X^3 + aX + b$. W związku z tym termy Y^2 możemy utożsamiać z wyrażeniem $X^3 + aX + b$. Dlatego też każdy z wielomianów $\psi_m, \theta_m, \omega_m$ możemy w sposób jednoznaczny reprezentować za pomocą wyrażenia postaci

$$a_0(X) + a_1(X)Y,$$

czyli tak, jakbyśmy reprezentowali elementy pierścienia ilorazowego $\mathbb{F}_p[X, Y]/(E)$.

Okazuje się, że wielomiany θ_m i ω_m można w ciałach charakterystyki różnej od 2 wyrazić dla $m \geq 1$ jako funkcje ψ_m:

$$\begin{aligned} \theta_m &= X\psi_m^2 - \psi_{m-1}\psi_{m+1}, \\ \omega_m &= \frac{\psi_{2m}}{2\psi_m}. \end{aligned}$$

Ponieważ funkcje dodawania i podwajania punktu mogą być stosowane rekurencyjnie, to naturalnym jest, że wielomiany $\psi_m, \theta_m, \omega_m$ mogą być przedstawione w formie rekurencyjnej. Z racji tego, że θ i ω zależą bezpośrednio od ψ skupimy swoją uwagę jedynie na wielomianach podziału.

$$\begin{aligned} \psi_0 &= 0, \\ \psi_1 &= 1, \\ \psi_2 &= 2Y, \\ \psi_3 &= 3X^4 + 6aX^2 + 12bX - a^2, \\ \psi_4 &= 4Y(X^6 + 5aX^4 + 20bX^3 - 5a^2X^2 - 4abX - a^3 - 8b^2), \end{aligned}$$

$$\begin{aligned} \psi_{2m+1} &= \psi_{m+2}\psi_m^3 - \psi_{m-1}\psi_{m+1}^3, \quad m \geq 2, \\ \psi_{2m} &= \frac{\psi_m(\psi_{m+2}\psi_{m-2}^2 - \psi_{m-2}\psi_{m+1}^2)}{2Y}, \quad m > 2. \end{aligned}$$

Należy w tym miejscu zauważyć, że przedstawione formuły na $\psi_m, \theta_m, \omega_m$ zostały wyprowadzone dla krzywej o równaniu $Y^2 = X^3 + aX + b$ i różnią się od formuł dla równania Weierstrassa w postaci ogólnej.

Wielomian podzielności ψ_m jest ściśle związany z punktami m-torsyjnymi krzywej E

$$E[m] = \{P \in E(\overline{\mathbb{F}}_p) : \ [m]P = \mathcal{O}\}.$$

Związek ten określa następujące twierdzenie

Twierdzenie 6 Jeżeli $P \in E(\overline{\mathbb{F}}_p) \setminus \{\mathcal{O}\}$ i $m \geq 1$, to $P \in E[m]$ wtedy i tylko wtedy, gdy $\psi_m(P) = 0$. ■

Dzięki temu twierdzeniu uzyskujemy bardzo dużą wiedzę na temat punktów m-torsyjnych. Z algebraicznego punktu widzenia będzie jednak istotne dla nas, że każdy punkt $P \in E[m]$ spełnia układ równań

$$\begin{cases} E(P) = 0, \\ \psi_m(P) = 0. \end{cases}$$

Te dwa równania pozwalają nam wykonywać operacje algebraiczne na punktach m-torsyjnych. W tym celu wystarczy przyjąć, że punkt taki ma współrzędne (X, Y) i wykonywać operacje algebraiczne w pierścieniu $\mathbb{F}_p[X, Y]/(E, \psi_m)$. Ten pierścień ilorazowy jest skończony, a ideał (E, ψ_m) gwarantuje nam, że znajdą się w nim wszystkie punkty m-torsyjne krzywej E. To właśnie operacje w tym

pierścieniu stanowią podstawę algorytmu Schoofa zliczania punktów krzywej eliptycznej. Zanim jednak przejdziemy do opisu tego algorytmu zobaczmy na przykładzie jaka jest struktura tego pierścienia.

Przykład 4 Rozważmy krzywą eliptyczną $E(\mathbb{F}_p)$ o równaniu $Y^2 = X^3 + aX + b$ oraz wielomian $\psi_3 = 3X^4 + 6aX^2 + 12bX - a^2$, który określa punkty rzędu 3 na krzywej $E(\overline{\mathbb{F}}_p)$. Krzywa $E(\mathbb{F}_p)$ może nie mieć punktów rzędu 3, ale na pewno istnieje takie rozszerzenie K ciała $\mathbb{F}_p$, które te punkty zawiera. Z naszych rozważań wynika, że ciało K jest zawarte w pierścieniu $\mathbb{F}_p[X,Y]/(E,\psi_3)$. Oznacza to, że współrzędne punktów rzędu 3 możemy reprezentować przez wyrażenia postaci

$$(a_0(X) \bmod \psi_3) + (a_1(X) \bmod \psi_3)Y.$$

Biorąc pod uwagę stopień ψ_3 możemy elementy pierścienia $\mathbb{F}_p[X,Y]/(E,\psi_3)$ traktować jako 8-elementowe wektory nad ciałem $\mathbb{F}_p$. □

René Schoof dokonał przełomu w podejściu do zliczania punktów krzywej eliptycznej nad ciałem skończonym. Jego prace [50, 51] zmniejszyły złożoność algorytmu zliczania punktów z $O(p^{1/4})$ do $O(\log^8 p)$. Był to ogromny postęp, który zapoczątkował intensywne prace nad tym zagadnieniem. Aby przedstawić ideę algorytmu Schoofa zacznijmy od przypomnienia twierdzenia Hassego, które mówi, że liczba punktów krzywej $E(\mathbb{F}_p)$ zadana jest wzorem

$$\#E(\mathbb{F}_p) = p + 1 - t,$$

gdzie $|t| \leq 2\sqrt{p}$. Podstawowy pomysł polega na tym, aby znaleźć wartości t_ℓ takie, że $t_\ell = t \bmod \ell$ dla pewnego zbioru małych liczb pierwszych ℓ. Jeżeli liczb pierwszych ℓ będzie wystarczająco dużo, to na podstawie twierdzenia chińskiego o resztach możliwe będzie jednoznaczne określenie wartości t. Na podstawie twierdzenia o gęstości rozkładu liczb pierwszych możemy stwierdzić, że takich liczb ℓ będzie potrzeba $O(\log p/\log\log p)$, a ich wielkość będzie rzędu $O(\log p)$.

Aby jednak wyznaczać odpowiednie wartości t_ℓ musimy dysponować jakąś tożsamością algebraiczną, która pozwoli nam zweryfikować ich poprawność. W tym momencie z pomocą przychodzi nam endomorfizm Frobeniusa φ spełniający równanie

$$\varphi^2 - [t]\varphi + [p] = [0].$$

Ponieważ równanie to jest spełnione dla wszystkich punktów krzywej $E(\overline{\mathbb{F}}_p)$, to w szczególności spełniają je punkty ℓ-torsyjne. W naszych dalszych rozważaniach będziemy zakładali, że $\ell \neq 2$ i $P(X,Y) \in E[\ell] \setminus \{\mathcal{O}\}$. Równanie charakterystyczne dla endomorfizmu Frobeniusa możemy przekształcić w następujący sposób

$$\begin{aligned} \varphi^2(P) + [p](P) &= [t]\varphi(P) \\ \varphi^2(P) + [p_\ell](P) &= [t_\ell]\varphi(P) \\ (X^{p^2}, Y^{p^2}) + [p_\ell](X,Y) &= [t_\ell](X^p, Y^p), \end{aligned}$$

gdzie $p_\ell = p \bmod \ell$. Wiedząc w jaki sposób wykonuje się działanie grupowe na punktach krzywej E możemy zweryfikować powyższą tożsamość dla wszystkich potencjalnych wartości $t_\ell \in \{1, \ldots, \ell - 1\}$. Jest to dość proste, gdyż operacje wykonujemy w pierścieniu $\mathbb{F}_p[X, Y]/(E, \psi_\ell)$ za każdym razem podstawiając kolejnego kandydata na t_ℓ. W większości przypadków postępowanie to kończy się pozytywną weryfikacją jednej z równości i wyznaczeniem poprawnej wartości t_ℓ. Problem pojawia się w przypadku, gdy

1. $\varphi^2(P) = [p_\ell](P)$,
2. $\varphi^2(P) = -[p_\ell](P)$.

Są to dwa przypadki, dla których nie można zastosować standardowego wzoru na dodawanie punktów krzywej. Załóżmy najpierw, że mamy do czynienia z przypadkiem $\varphi^2(P) = [p_\ell](P)$. Wtedy wielomian charakterystyczny endomorfizmu Frobeniusa przyjmuje postać

$$[2p_\ell](P) = [t_\ell]\varphi(P)$$

$$[2p_\ell](X, Y) = [t_\ell](X^p, Y^p).$$

Teraz możemy zweryfikować powyższą tożsamość dla wszystkich potencjalnych wartości $t_\ell \in \{1, \ldots, \ell - 1\}$. Wszystkie obliczenia, tak jak poprzednio, wykonujemy w pierścieniu $\mathbb{F}_p[X, Y]/(E, \psi_\ell)$. Wartość $t_\ell = 0$ pomijamy, gdyż dla $\ell \neq 2$ wyrażenie $2p_\ell$ nigdy nie jest zerem. Jeżeli i tym razem nie znajdziemy wartości t_ℓ, to stwierdzamy, że mamy do czynienia z przypadkiem $\varphi^2(P) = -[p_\ell](P)$, dla którego $t_\ell = 0$.

Opisana metoda pozwala wyznaczać wartość śladu Frobeniusa t modulo małe liczby pierwsze $\ell_i \neq 2$. Ponieważ $|t| \leq 2\sqrt{p}$, to dokładną wartość śladu będziemy mogli wyznaczyć z twierdzenia chińskiego o resztach jeżeli tylko

$$\prod_{i=1}^{k} \ell_i > 4\sqrt{p}.$$

Na koniec warto jeszcze zaznaczyć, że w literaturze generalnie funkcjonuje inny opis tego algorytmu. Pochodzi on z oryginalnej pracy Schoofa [50] i uwzględnia optymalizacje pozwalające na redukcję jego złożoności obliczeniowej. W naszej prezentacji algorytmu celowo zostały one pominięte, gdyż utrudniają nieco zrozumienie ogólnej idei.

4.4 Wielomiany modularne, ich modyfikacje i algorytm Elkiesa

W algorytmie Schoofa podstawową strukturą algebraiczną, w której wykonywane są obliczenia, jest pierścień $\mathbb{F}_p[X, Y]/(E, \psi_\ell)$. Jak było wspomniane wcześniej, pierścień ten zawiera pewne ciało K o tej własności, że $E[\ell] \subset E(K)$.

Elementy tego pierścienia to wektory o współczynnikach należących do ciała $\mathbb{F}_p$ i długości rzędu ℓ^2. Operacje na elementach takiej wielkości są dość kosztowne. Wystarczy zauważyć, że dla $\ell = 101$ mamy już ponad 10 000 współczynników. W związku z tym rodzi się naturalne pytanie, czy nie istnieje jakiś mniejszy pierścień $R \subset \mathbb{F}_p[X,Y]/(E,\psi_\ell)$, który dałby możliwość ograniczenia złożoności obliczeniowej i prowadził do efektywniejszej metody zliczania punktów krzywej eliptycznej.

Aby zredukować rozmiary pierścienia, w którym prowadzimy obliczenia, przyjrzyjmy się najpierw strukturze grupy punktów m-torsyjnych. W części 2.3 poświęconej krzywym eliptycznym nad ciałem liczb zespolonych została zaprezentowana struktura grupy $E[m]$ dla krzywej określonej nad ciałem liczb zespolonych. Okazuje się, że w przypadku ciał skończonych sytuacja jest bardzo podobna i prawdziwy jest następujący lemat:

Lemat 3 Jeżeli E jest krzywą eliptyczną nad ciałem K charakterystyki $p > 0$, a m jest dodatnią liczbą całkowitą, to

1. $E[m] \simeq (\mathbb{Z}/m\mathbb{Z}) \times (\mathbb{Z}/m\mathbb{Z})$ jeśli $m \neq 0 \bmod p$,
2. $E[p^r] \simeq (\mathbb{Z}/p^r\mathbb{Z})$ lub $E[p^r] \simeq \{0\}$ jeśli $m = 0 \bmod p$.

■

Z naszego punktu widzenia interesujący jest jedynie pierwszy przypadek, gdyż dla algorytmu Schoofa liczby pierwsze ℓ są mniejsze niż p. W związku z tym możemy przyjąć, że struktura punktów ℓ-torsyjnych jest następująca:

$$E[\ell] \simeq (\mathbb{Z}/\ell\mathbb{Z}) \times (\mathbb{Z}/\ell\mathbb{Z}) \simeq \langle P_1 \rangle \times \langle P_2 \rangle,$$

gdzie P_1 i P_2 są punktami krzywej generującymi różne podgrupy rzędu ℓ. Podczas analizy algorytmu Schoofa w poprzedniej części można było zauważyć, że potrzebny jest tylko jeden punkt ℓ-torsyjny. W celu zmniejszenia złożoności obliczeniowej algorytmu powinniśmy więc szukać takiego rozszerzenia $K \supset \mathbb{F}_p$, dla którego zachodzi $E(K) \cap E[\ell] \neq \{\mathcal{O}\}$. Należy przy tym zauważyć, że im mniejszy będzie stopień jego rozszerzenia $[K : \mathbb{F}_p]$ tym mniejsza będzie złożoność algorytmu zliczania punktów krzywej.

Skoro grupa $E[\ell]$ generowana jest przez punkty P_1 i P_2, to ma w sobie $\ell + 1$ podgrup. Oznaczmy je jako $C_1 = \langle P_1 \rangle$, $C_2 = \langle P_2 \rangle$ oraz $C_i = \langle P_1 + [i-2]P_2 \rangle$ dla $i = 2, \ldots, \ell+1$. Każda taka grupa może stanowić jądro pewnego przekształcenia izogenicznego krzywych eliptycznych.

Definicja 9 *Izogenią* krzywych E_1 i E_2 nazywamy regularne przekształcenie wymierne z E_1 do E_2, które przeprowadza punkt w nieskończoności na punkt w nieskończoności. □

Jeśli spojrzymy na izogenię jedynie z punktu widzenia teorii grup, to można wykazać, że jest ona homomorfizmem grup punktów odpowiednich krzywych. Taki homomorfizm może mieć niezerowe jądro, którym jest pewna podgrupa punktów krzywej E_1. Naszą uwagę skupimy teraz na tych izogeniach, których jądrami są podgrupy C_i. Okazuje się, że jeśli jądrem izogenii ma być grupa ℓ-elementowa, to istnieje możliwość powiązania j-niezmiennika krzywej wyjściowej z j-niezmiennikami krzywych znajdujących się w obrazie izogenii. Elementem łączącym oba j-niezmienniki jest wielomian modularny $\Phi_\ell(X, Y)$. Ma on tę własność, że pierwiastki wielomianu $\Phi_\ell(X, j(E_1))$ są równe j-niezmiennikom krzywych powstałych przez izogeniczne przekształcenie krzywej E_1 z jądrem równym odpowiednio $C_1, C_2, \ldots, C_{\ell+1}$. Z tego właśnie powodu stopień ℓ-tego wielomianu modularnego jest zawsze równy $\ell + 1$.

Przykład 5 Przykład klasycznego wielomianu modularnego

$$\begin{aligned}\Phi_3(X,Y) \;=\;& X^4 - X^3Y^3 + Y^4 + \\ & 2232\,(X^3Y^2 + Y^3X^2) - \\ & 1069956\,(X^3Y + Y^3X) + \\ & 36864000\,(X^3 + Y^3) + \\ & 2587918086\,X^2Y^2 + \\ & 8900222976000\,(X^2Y + Y^2X) + \\ & 452984832000000\,(X^2 + Y^2) - \\ & 770845966336000000\,XY + \\ & 1855425871872000000000\,(X + Y).\end{aligned}$$

Wielomiany modularne Φ_ℓ są symetryczne i charakteryzują się bardzo dużymi współczynnikami. Jak wynika z [10] logarytm naturalny największego współczynnika wielomianu Φ_ℓ jest rzędu

$$6\ell \log \ell + O(\ell).$$

Oznacza to tak naprawdę, że współczynniki tych wielomianów rosną wykładniczo ze wzrostem ℓ. Można oszacować, że na przechowanie pojedynczego wielomianu modularnego dla ℓ rzędu 2^{512} potrzeba około 500 MB miejsca. Dlatego też czasami lepiej jest generować wspomniane wielomiany na potrzeby konkretnego zadania i automatycznie redukować ich współczynniki modulo p. To właśnie z myślą o takim podejściu opracowany został algorytm szybkiego mnożenia wielomianów i szeregów potęgowych o współczynnikach całkowitych, który będzie prezentowany w następnym rozdziale. □

Wróćmy w tym miejscu do wielomianu podziału ψ_m. Można wykazać, że dla nieparzystych liczb m wielomian ten zależny jest tylko od zmiennej X, a jego stopień jest równy $(m^2 - 1)/2$. Z taką sytuacją mamy do czynienia w naszym przypadku, gdyż rozważamy wielomiany ψ_ℓ, gdzie ℓ jest nieparzystą liczbą pierwszą.

Mając na uwadze Twierdzenie 6 możemy zapisać wielomian ψ_ℓ dla $\ell \neq 2$ jako

$$\psi_\ell(X) = \prod_{P \in E[\ell] \setminus \{\mathcal{O}\}} (X - P_X)^{1/2},$$

gdzie P_X oznacza współrzędną X punktu P. Zauważmy również, że wykładnik $1/2$ znika, gdyż dla każdego skończonego punktu rzędu ℓ istnieje punkt przeciwny o tej samej współrzędnej X. Korzystając z faktu, że grupa $E[\ell]$ dzieli się na $\ell+1$ podgrup $C_1, C_2, \ldots, C_{\ell+1}$ możemy wielomian podziału zapisać również w postaci

$$\begin{aligned} \psi_\ell(X) &= \prod_{i=1}^{\ell+1} \left(\prod_{P \in C_i \setminus \{\mathcal{O}\}} (X - P_X)^{1/2} \right) \\ &= \prod_{i=1}^{\ell+1} F_{\ell,i}(X). \end{aligned}$$

Jak można zauważyć stopień wielomianu $F_{\ell,i}(X)$ wynosi $(\ell-1)/2$ i jest znacząco niższy od stopnia wielomianu podziału $\psi_\ell(X)$. Gdyby więc udało się znaleźć taki wielomian $F_{\ell,i}(X)$, którego współczynniki leżałyby w ciele $\mathbb{F}_p$, to moglibyśmy prowadzić weryfikację tożsamości Frobeniusa w pierścieniu $\mathbb{F}_p[X,Y]/(E, F_{\ell,i})$. Okazuje się, że średnio dla co drugiej liczby ℓ jest to możliwe. Liczby te noszą nazwę *liczb pierwszych Elkiesa* i można je rozpoznać po sposobie w jaki wielomian $\Phi_\ell(X, j(E))$ rozkłada się na czynniki pierwsze w pierścieniu $\mathbb{F}_p[X]$. René Schoof wykazał [51], że prawdziwe jest następujące stwierdzenie.

Stwierdzenie 1 Załóżmy, że $E(\mathbb{F}_p)$ nie jest superosobliwa i $j(E) \notin \{0, 1728\}$. Jeżeli $\Phi_\ell(X, j(E)) = h_1 h_2 \cdots h_s$ jest rozkładem na czynniki nierozkładalne nad $\mathbb{F}_p$ to stopnie wielomianów $h_1, h_1, \ldots, h_s$ spełniają jeden z poniższych warunków:

1. Jest jeden czynnik stopnia 1 i jeden czynnik stopnia ℓ – w tej sytuacji ślad Frobeniusa spełnia zależność $t^2 = 4p \bmod \ell$ i ℓ jest liczbą pierwszą Elkiesa.

2. Wszystkie czynniki mają stopień 1 – w tej sytuacji ślad Frobeniusa również spełnia zależność $t^2 = 4p \bmod \ell$ i ℓ jest liczbą pierwszą Elkiesa.

3. Są dwa czynniki stopnia 1, a pozostałe mają stopień r – w tej sytuacji $t^2 - 4p$ jest kwadratem w $\mathbb{F}_\ell$, ℓ jest liczbą pierwszą Elkiesa, $r|\ell - 1$, a endomorfizm Frobeniusa φ działa na grupie $E[\ell]$ za pomocą macierzy

   $$\begin{pmatrix} \lambda & 0 \\ 0 & \mu \end{pmatrix},$$

   gdzie $\lambda, \mu \in \mathbb{F}_\ell^\times$.

4. Wszystkie czynniki mają stopień $r \neq 1$ – w tej sytuacji $t^2 - 4p$ nie jest kwadratem w $\mathbb{F}_\ell$, ℓ jest tak zwaną *liczbą pierwszą Atkina*, $r|\ell + 1$, a endomorfizm Frobeniusa φ działa na grupie $E[\ell]$ za pomocą pewnej macierzy 2×2, której wielomian charakterystyczny jest nierozkładalny nad $\mathbb{F}_\ell$.

■

Możemy powiedzieć, że elementem wspólnym wszystkich liczb pierwszych Elkiesa jest fakt, że wielomian modularny $\Phi_\ell(X, j(E))$ ma przynajmniej jeden pierwiastek w ciele $\mathbb{F}_p$. Ten pierwiastek jest j-niezmiennikiem krzywej izogenicznej, a jądrem tej izogenii jest jedna z podgrup C_i. Te dane pozwalają na wyznaczenie wielomianu $F_{\ell,i}|\psi_\ell$ według następującego schematu:

1. Wyznaczamy j-niezmiennik krzywej $E(\mathbb{F}_p)$, który oznaczamy przez j.
2. Znajdujemy pierwiastek $\tilde{j}$ wielomianu $\Phi_\ell(X, j)$, który jest j-niezmiennikiem krzywej izogenicznej.
3. Korzystając z modelu zespolonego krzywej wyznaczamy współczynniki $\tilde{a}$ i $\tilde{b}$ krzywej izogenicznej
$$Y^2 = X^3 + \tilde{a}X + \tilde{b}.$$
4. Znając obie krzywe wyznaczamy jądro izogenii C i wielomian
$$F_\ell(X) = \prod_{P \in C \setminus \{\mathcal{O}\}} (X - P_X)^{1/2}.$$

Szczegółowy opis sposobu wyznaczania wielomianu $F_\ell(X)$ można znaleźć w pracy Schoofa [51].

Jednak aby wyznaczyć wielomian $F_\ell(X)$ konieczne jest wyznaczenie wielomianu modularnego. Może to być klasyczny wielomian modularny [3, 16] lub wielomian Müllera [43]. Oba z nich dają możliwość prawidłowego wyznaczenia $F_\ell(X)$ z tym, że wielomian Müllera ma mniej współczynników i są one mniejsze co do wartości bezwzględnej niż w przypadku klasycznego wielomianu modularnego.

Aby wyznaczyć wielomiany modularne wykorzystuje się teorię krzywych eliptycznych nad ciałem liczb zespolonych. Ponieważ wielomiany mają współczynniki całkowite, to można je później zredukować modulo dowolna liczba pierwsza i przenieść do pierścienia $\mathbb{F}_p[X, Y]$. Przypomnijmy (część 2.3), że krzywa eliptyczna nad ciałem liczb zespolonych może być z dokładnością do izomorfizmu utożsamiana z torusem zespolonym $\mathbb{C}/\Lambda$, gdzie $\Lambda = \mathbb{Z} + \tau\mathbb{Z}$. Zgodnie z Lematem 1 j-niezmiennik takiej krzywej rozwija się w następujący szereg Fouriera
$$j(\tau) = q^{-1} + 744 + \sum_{n \geq 1} c_n q^n,$$
gdzie współczynniki c_n są dodatnimi liczbami całkowitymi, a $q = e^{2\pi i \tau}$. Jeżeli rozważymy teraz wszystkie izogenie, których jądro stanowią kolejne podgrupy

rzędu ℓ, to można udowodnić, że j-niezmienniki odpowiednich krzywych izogenicznych są równe

$$j(\ell\tau), j\left(\frac{\tau}{\ell}\right), j\left(\frac{\tau+1}{\ell}\right), \ldots, j\left(\frac{\tau+\ell-1}{\ell}\right).$$

Dla tych j-niezmienników klasyczny wielomian modularny Φ_ℓ ma postać

$$\Phi_\ell(X, j(\tau)) = (X - j(\ell\tau)) \prod_{k=0}^{\ell-1} \left(X - j\left(\frac{\tau+k}{\ell}\right)\right).$$

Aby wyznaczyć współczynniki należy j-niezmienniki potraktować jako szeregi formalne q i zapisać iloczyn

$$(X - j(\ell\tau)) \prod_{k=0}^{\ell-1} \left(X - j\left(\frac{\tau+k}{\ell}\right)\right) = X^{\ell+1} + \sum_{r=0}^{\ell} c_r X^r$$

jako wielomian $\mathbb{Z}[[q]][X]$. Można udowodnić, że każdy ze współczynników c_r jest elementem pierścienia $\mathbb{Z}[j(\tau)]$. W związku z tym wyznaczenie współczynników wielomianu modularnego sprowadza się do przedstawienia elementów c_r jako wielomianów od $j(\tau)$. Jest to bardzo proste w przypadku, gdy zarówno współczynnik c_r, jak i j-niezmiennik $j(\tau)$ mają odpowiednio długie rozwinięcie.

Jak widać cały proces wyznaczania współczynników wielomianu modularnego sprowadza się do wykonywania operacji arytmetycznych na szeregach potęgowych $\mathbb{Z}[[q]]$. Szeregi te reprezentowane są ze skończoną precyzją rzędu $O(\ell^2)$. Aby cały proces przebiegał sprawnie należy zatroszczyć się o efektywną implementację arytmetyki na szeregach potęgowych o skończonej precyzji. W tym miejscu trzeba zauważyć, że arytmetyka na szeregach potęgowych nie różni się specjalnie od arytmetyki na wielomianach w pierścieniu $K[X]/(X^n)$. Jedyna różnica polega na tym, że w przypadku szeregów dopuszcza się, aby zmienna formalna miała ujemny wykładnik.

W kolejnych dwóch rozdziałach pracy zostanie opisany algorytm do realizacji szybkiej arytmetyki na wielomianach i szeregach potęgowych. Został on skonstruowany z myślą o zastosowaniu w procesie generowania wielomianów modularnych. Niemniej jednak jego efektywność okazała się na tyle duża, że można rozważać jego wykorzystanie również w przypadku innych problemów obliczeniowych.

Rozdział 5

Nowy, efektywny algorytm szybkiego mnożenia wielomianów o współczynnikach całkowitych

W 1971 roku Schönhage i Strassen [52] zaproponowali nowy algorytm mnożenia dużych liczb całkowitych. Szybkie algorytmy mnożenia oparte na przekształceniu FFT (Fast Fourier Transform) są od tamtego czasu nieustannie doskonalone i poprawiane. Dzisiaj dysponujemy wieloma szybkimi algorytmami mnożenia liczb całkowitych ([44], [12]) i wielomianów ([53], [27], [28] [26]). Niektóre z nich są niezależne od architektury procesora, a inne dedykowane są na konkretny układ obliczeniowy. Pomimo że FFT ma ogromną przewagę nad klasycznymi algorytmami mnożenia, to jej wersja przeznaczona na maszyny wielordzeniowe nie jest zbyt łatwa w implementacji. Poza tym mnożenie liczb całkowitych z wykorzystaniem FFT jest opłacalne dopiero, gdy czynniki mają ponad 100 000 bitów [19]. Chcąc zmienić ten stan rzeczy można stworzyć algorytm jednocześnie używający FFT i twierdzenia chińskiego o resztach (CRT – Chinese Remainder Theorem). Twierdzenie chińskie jest często wykorzystywane do przyspieszenia obliczeń w popularnym algorytmie kryptograficznym RSA. Zastosowanie CRT pozwala na rozdzielenie i równoległe wykonanie operacji podpisu lub deszyfrowania. Inspiracja tą ideą i podejściem przedstawionym w pracy [21] była główną motywacją do rozpoczęcia prac badawczych nad nowym, szybkim, równoległym algorytmem mnożenia wielomianów o współczynnikach całkowitych. Wstępna koncepcja algorytmu została zaprezentowana w ramach artykułów autora [7, 6].

5.1 Szybka transformata Fouriera i jej implementacje

Szybka transformata Fouriera (FFT) jest efektywnym algorytmem wyznaczania dyskretnej transformaty Fouriera (DFT). Podstawą DFT jest reprezentowanie wielomianów nie przez współczynniki, a przez wartości w punktach. Wyznaczenie DFT zadanej w n punktach na podstawie definicji pochłania $O(n^2)$ operacji arytmetycznych. Zastosowanie FFT jest znacznie korzystniejsze i pozwala otrzymać DFT po wykonaniu $O(n \log n)$ operacji. Jest to główny powód tego, że FFT jest szeroko rozważana w publikacjach dotyczących efektywności obliczeń [29], [13], [11], [22], [27], [28], [26], [59]. Podczas naszych eksperymentów numerycznych wykorzystywaliśmy klasyczny algorytm FFT, ale nic nie stoi na przeszkodzie, żeby zastosować inną implementację. Szczególnie godna uwagi jest tak zwana *cache-friendly truncated FFT* [26], która została zoptymalizowana pod kątem architektury współczesnych procesorów. Szczegóły dotyczące zaimplementowanego algorytmu można znaleźć w części poświęconej praktycznej implementacji algorytmu mnożenia. Część ta zawiera również twierdzenia uzasadniające poprawność algorytmu zaproponowanego w pracy.

5.2 Reprezentacja elementów ciała i szybkie mnożenie w pierścieniu wielomianów

Zacznijmy od krótkiego przypomnienia dotyczącego metody redukcji Montgomery'ego. Algorytm ten jest szeroko stosowany w implementacjach mnożenia modularnego dla funkcji kryptograficznych [40]. Popularność tej metody jest ściśle związana z jej podstawową własnością: redukt wyznaczany jest bez konieczności wykonywania dzielenia całkowitoliczbowego. Niestety algorytm Montgomery'ego wymaga wcześniejszego wyznaczenia kilku wartości. Z tego powodu stosowany jest jedynie w tych sytuacjach, gdzie określony moduł wykorzystywany jest wielokrotnie. Poniższy lemat stanowi podstawę metody redukcji Montgomery'ego [42]:

Lemat 4 Niech a, b, M, R będą liczbami całkowitymi takimi, że $(M, R) = 1$, $a, b < M < R$, $q = -M^{-1} \bmod R$ i

$$\begin{aligned} t_1 &= a \cdot b \\ t_2 &= t_1 \bmod R \\ t_3 &= t_2 \cdot q \\ t_4 &= t_3 \bmod R \\ t_5 &= t_4 \cdot M \\ t &= (t_1 + t_5)/R. \end{aligned}$$

Wtedy prawdziwa jest jedna z następujących równości:

$$abR^{-1} \bmod M = t \qquad \text{lub} \qquad abR^{-1} \bmod M = t - M.$$

W praktyce liczba M jest zazwyczaj nieparzysta, a R jest potęgą liczby 2. Oznacza to, że redukcja modularna i dzielenie przez R sprowadzają się do obcięcia odpowiedniej liczby starszych lub młodszych (dzielenie) bitów reprezentacji. W naszych rozważaniach pójdziemy jednak dalej i pokażemy, że redukcja Montgomery'ego daje nam możliwość zdefiniowania efektywnej obliczeniowo reprezentacji pierścienia $\mathbb{Z}/M\mathbb{Z}$.

Twierdzenie 7 Niech M i R będą dodatnimi liczbami całkowitymi. Jeśli $(M, R) = 1$, $M < R$ i pierścienie

$$R_1 = \langle \{0, 1, \ldots, M-1\}, 0, 1, +, -, \cdot \rangle,$$

$$R_2 = \langle \{0, 1, \ldots, M-1\}, 0, R \bmod M, +, -, \odot \rangle$$

zdefiniowane są następująco

$$\begin{aligned} a \pm b &= a \pm b \bmod M \\ a \cdot b &= ab \bmod M \\ a \odot b &= abR^{-1} \bmod M, \end{aligned}$$

to R_1 i R_2 są izomorficzne.

Dowód: Definiujemy przekształcenie $h : R_1 \to R_2$ jako

$$h(x) = xR \bmod M.$$

Ponieważ M i R są względnie pierwsze, to R jest odwracalne modulo M. Oznacza to, że h jest w rzeczywistości bijekcją. Do zakończenia dowodu musimy więc pokazać, że przekształcenie h jest homomorfizmem:

1. $h(0) = 0$, $h(1) = R \bmod M$,
2. $h(a \pm b) = (a \pm b)R \bmod M = (aR \pm bR) \bmod M = h(a) + h(b)$,
3. $h(a \cdot b) = abR \bmod M = (aRbR)R^{-1} \bmod M = h(a) \odot h(b)$.

To kończy dowód. ■

Do końca tej części będziemy zakładali, że liczba n jest potęgą dwójki i jest większa od maksymalnego stopnia wielomianu, który chcemy mnożyć. Będziemy ponadto przyjmowali, że wartość bezwzględna współczynników wielomianu jest mniejsza niż B.

Definicja 10 Niech $f(X) = f_{n-1}X^{n-1} + \cdots + f_1X + f_0 \in \mathbb{Z}[X]$ i $M \in \mathbb{Z}$. Definiujemy $f(X) \bmod M$ jako

$$f(X) \bmod M = (f_{n-1} \bmod M)X^{n-1} + \cdots + (f_0 \bmod M),$$

gdzie

$$f_i \bmod M \in \left\{ \left\lfloor \frac{-M+1}{2} \right\rfloor, \ldots, -1, 0, 1, \ldots, \left\lfloor \frac{M-1}{2} \right\rfloor \right\},$$

dla i od 0 do $n-1$. □

Okazuje się, że dla odpowiednio dobranej liczby M mnożenie dwóch wielomianów w pierścieniu $(\mathbb{Z}/M\mathbb{Z})[X]$ daje taki sam wynik, co mnożenie tych samych wielomianów w pierścieniu $\mathbb{Z}[X]$. Poniższy lemat pokazuje w jaki sposób należy dobrać liczbę M, aby ten efekt osiągnąć.

Lemat 5 Niech $f(X) = f_{n-1}X^{n-1} + \cdots + f_1X + f_0$, $g(X) = g_{n-1}X^{n-1} + \cdots + g_1X + g_0$ będą takimi wielomianami o współczynnikach całkowitych, że $|f_i| < B$ i $|g_i| < B$ dla $i = 0, 1, \ldots, n-1$. Jeśli liczba całkowita M spełnia poniższy warunek

$$2nB^2 < M$$

to $f(X)g(X) \bmod M = f(X)g(X)$.

Dowód: Jeśli $f(X)g(X) = h(X) = h_{2n-2}X^{2n-2} + \cdots + h_1X + h_0$ to

$$\begin{aligned} h(X) &= \left(\sum_{i=0}^{n-1} f_iX^i\right)\left(\sum_{j=0}^{n-1} g_jX^j\right) \\ &= \sum_{i=0}^{n-1}\sum_{j=0}^{i} f_jg_{i-j}X^i + \sum_{i=1}^{n-1}\sum_{j=0}^{n-1-i} f_{i+j}g_{n-1-j}X^{n-1+i} \\ &= \sum_{i=0}^{n-1} X^i\sum_{j=0}^{i} f_jg_{i-j} + \sum_{i=1}^{n-1} X^{n-1+i}\sum_{j=0}^{n-1-i} f_{i+j}g_{n-1-j} \end{aligned}$$

Ponieważ $|f_i| < B$ i $|g_i| < B$ to mamy

1. $|h_i| = \left|\sum_{j=0}^{i} f_jg_{i-j}\right| \leq \sum_{j=0}^{i} |f_j||g_{i-j}| < \sum_{j=0}^{i} B^2 = (i+1)B^2$ dla wszystkich i od 0 do $n-1$,
2. $|h_{n-1+i}| = \left|\sum_{j=0}^{n-1-i} f_{i+j}g_{n-1-j}\right| \leq \sum_{j=0}^{n-1-i} |f_{i+j}||g_{n-1-j}| < \sum_{j=0}^{n-1-i} B^2 = (n-i)B^2$ dla wszystkich i od 1 do $n-1$.

To oznacza, że $|h_i| < nB^2$ dla wszystkich i od 0 do $2n-2$. Jeśli $M > 2nB^2$ to wszystkie współczynniki (zapisane jak w Definicji 10) wielomianów $f(X), g(X)$ i $h(X)$ są reprezentowane w pierścieniu reszt modulo M bez redukcji. To prowadzi do równania $f(X)g(X) \bmod M = f(X)g(X)$ i kończy dowód. ■

Twierdzenie 8 Niech $f(X) = f_{n-1}X^{n-1} + \cdots + f_1X + f_0$, $g(X) = g_{n-1}X^{n-1} + \cdots + g_1X + g_0$ będą takimi wielomianami o współczynnikach całkowitych, że $|f_i| < B$ i $|g_i| < B$. Jeśli liczba pierwsza p spełnia następujące warunki:

(1) $2nB^2 < p$,

(2) $p = 2^{m+1}r + 1$ dla pewnego $2^{m+1} \geq 2n$ i $r \in \mathbb{Z}$,

to

$$f(X)g(X) = f(X)g(X) \bmod p = (f(X) \bmod p)(g(X) \bmod p) \bmod p$$

i ciało $\mathbb{F}_p$ może być wykorzystane do mnożenia wielomianów f i g z wykorzystaniem algorytmu FFT.

Dowód: Ponieważ operacja $\bmod\ p$ jest homomorfizmem naturalnym pierścienia $\mathbb{Z}$, to

$$(f(X) \bmod p)(g(X) \bmod p) \bmod p = f(X)g(X) \bmod p$$

Na podstawie Lematu 5 otrzymujemy natomiast równość

$$f(X)g(X) \bmod p = f(X)g(X).$$

W związku z tym mnożenie wielomianów $f(X), g(X) \in \mathbb{Z}[X]$ daje taki sam wynik, co mnożenie wielomianów $f(X) \bmod p, g(X) \bmod p \in \mathbb{F}_p[X]$ pod warunkiem, że elementy ciała $\mathbb{F}_p$ są reprezentowane przez liczby

$$\left\{-\frac{p-1}{2}, \ldots, -1, 0, 1, \ldots, \frac{p-1}{2}\right\}.$$

Forma liczby pierwszej $p = 2^{m+1}r + 1$ została wybrana pod kątem zastosowania algorytmu FFT. Wynika to z faktu, że dla tak określonej liczby p, ciało $\mathbb{F}_p$ zawiera pierwiastek pierwotny z jedności stopnia 2^{m+1}. ■

Zastosowanie szybkiej transformaty Fouriera nad ciałem skończonym zamiast nad ciałem liczb zespolonych pozwala usunąć kosztowne operacje zmiennoprzecinkowe i zastąpić je szybkimi działaniami na liczbach całkowitych. Prowadzi to do szybszego algorytmu, gdyż na procesorach rodziny Intel Core 2 mnożenie całkowitoliczbowe jest około 30 razy szybsze od mnożenia zmiennoprzecinkowego. Możemy powiedzieć, że za zaproponowanym w pracy wykorzystaniem ciała skończonego przemawiają dwa argumenty:

1. maksymalizacja szybkości algorytmu,
2. eliminacja błędu zaokrągleń, którymi cechują się operacje zmiennoprzecinkowe.

Oczywiście powstaje naturalne pytanie o warunki, jakie muszą być spełnione aby implementacja algorytmu mnożenia wielomianów była jak najszybsza. Chcąc odpowiedzieć na to pytanie należy zwrócić uwagę na fakt, że główną operacją wykonywaną podczas mnożenia wielomianów jest mnożenie w ciele skończonym $\mathbb{F}_p$. Składa się ono z mnożenia całkowitoliczbowego i redukcji modularnej. To właśnie optymalizacja redukcji daje najlepsze efekty jeśli chodzi o całkowitą szybkość działania algorytmu mnożącego wielomiany. Zazwyczaj problem redukcji modularnej rozwiązywany jest na jeden z trzech sposobów:

(a) Znalezienie takiej liczby pierwszej p, dla której operacja redukcji modularnej sprowadza się do wykonania kilku dodawań i odejmowań. Istnieje wiele takich liczb pierwszych, na przykład: $2^{224}-2^{96}+1 = 2^{96}(2^{128}-1)+1$ lub $2^{512} - 2^{32} + 1 = 2^{32}(2^{480} - 1) + 1$.

(b) Inną metodą optymalizacji redukcji modularnej jest zastosowanie algorytmu redukcji Montgomery'ego [44]. Metoda ta jest dużo bardziej ogólna i może być zastosowana w przypadku każdej liczby pierwszej różnej od 2.

(c) W ostateczności można również zastosować klasyczny algorytm wyznaczania ilorazu i reszty z dzielenia. Wadą tej metody jest to, że jest trudna w efektywnej implementacji [42].

Metoda zaproponowana w punkcie (a) pozwala wyznaczyć iloczyn elementów ciała skończonego przy użyciu jednego mnożenia całkowitoliczbowego i jest niewątpliwie najszybsza spośród powyżej zaproponowanych. Liczby pierwsze o takiej specjalnej postaci znalazły już swoje stałe miejsce w kryptografii i są powszechnie wykorzystywane w algorytmach opartych na krzywych eliptycznych (FIPS 186-3 [62] i ANSI X509.62 [61]). Więcej informacji dotyczących szybkiej implementacji redukcji modularnej dla liczb szczególnej postaci można znaleźć w artykule [2].

Teraz jesteśmy gotowi, aby oszacować złożoność obliczeniową zaprezentowanej wcześniej metody mnożenia. Mnożenie wielomianów składa się z następujących operacji: wykonanie transformaty Fouriera, wymnożenia otrzymanych wektorów i wykonania odwrotnej transformaty Fouriera. We wszystkich tych operacjach główny koszt stanowią mnożenia w ciele $\mathbb{F}_p$. Dlatego też złożoność algorytmu oszacujemy na podstawie liczby niezbędnych mnożeń modulo p.

Twierdzenie 9 Niech dane będą dwa wielomiany całkowitoliczbowe stopnia mniejszego od n, dla których wartość bezwzględna każdego współczynnika jest mniejsza od B. Jeśli liczba pierwsza p spełnia warunki z Twierdzenia 8, to wymnożenie takich wielomianów przy użyciu algorytmu FFT, wymaga wykonania $n(2 + 3\log(n))$ mnożeń w $\mathbb{F}_p$.

Dowód: Pojedyncze mnożenie FFT złożone jest z trzech następujących kroków:

1. Transformata Fouriera dwóch wielomianów stopnia mniejszego od n wymaga $2n\log(n)$ mnożeń w $\mathbb{F}_p$ (każdy wielomian transformujemy do wektora o $2n$ współczynnikach).

2. Mnożenie po współrzędnych dwóch wektorów o $2n$ współczynnikach wymaga $2n$ mnożeń w $\mathbb{F}_p$.

3. Odwrotna transformata Fouriera wektora o $2n$ współczynnikach wymaga $n\log(n)$ mnożeń w $\mathbb{F}_p$.

Oznacza to, że pojedyncze mnożenie wielomianów z wykorzystaniem algorytmu FFT pochłania $n(2 + 3\log(n))$ mnożeń w ciele $\mathbb{F}_p$, co kończy dowód. ■

Zaproponowany w pracy algorytm mnożenia może być wykorzystany do wyznaczenia odwrotności w pierścieniu formalnych szeregów potęgowych o współczynnikach całkowitych. Taki szereg jest odwracalny wtedy i tylko wtedy, gdy jego współczynnik przy najniższej potędze jest równy 1 lub -1. Zastosowanie metody iteracyjnej Newtona dla pierścieni p-adycznych pozwala nam szybko wyznaczyć odwrotność szeregu. W [13] można znaleźć ogólny zarys tej metody, a artykuły [24] i [1] zawierają jej dokładny opis. Metoda Newtona wyznaczania odwrotności jest szybka i wykorzystuje jedynie mnożenia, dodawania i odejmowania szeregów. Aby odwrócić szereg o n współczynnikach musimy wykonać $\log n$ iteracji. Jeżeli nasz szereg potęgowy jest odwracalny i ma postać:

$$A = X^k \sum_{j=0}^{n-1} a_j X^j,$$

to poniższa procedura pozwala znaleźć jego odwrotność z dokładnością do n współczynników:

Data: Szereg potęgowy $A = X^k \sum_{j=0}^{n-1} a_j X^j$.
Result: Szereg potęgowy I taki, że $(A \cdot I) \bmod X^n = 1$.

begin
  $m \leftarrow 0$;
  $I \leftarrow \frac{1}{a_0}$;
  while $2^m < n$ **do**
    $I \leftarrow (2I - I^2 \sum_{j=0}^{2^m} a_j X^j) \bmod X^n$;
    $m \leftarrow m + 1$;
  end
  $I \leftarrow X^{-k} I$;
  return I;
end

Algorytm 1: Odwracanie szeregu potęgowego za pomocą metody iteracyjnej Newtona

5.3 Wykorzystanie twierdzenia chińskiego o resztach do rozdzielenia obliczeń między wiele procesorów

Aby zmniejszyć złożoność obliczeniową algorytmu przedstawionego w poprzedniej części wykorzystamy twierdzenie chińskie o resztach. Takie podejście pozwoli nam zastąpić pojedyncze mnożenie w ciele $\mathbb{F}_p$ przez wiele mnożeń w znacznie mniejszych ciałach $\mathbb{F}_{p_i}$. Dodatkowym atutem takiego podejścia jest możliwość rozdzielenia obliczeń w ciałach $\mathbb{F}_{p_i}$ pomiędzy wiele procesorów. Aby osiągnąć nasz cel musimy jednak najpierw znaleźć takie liczby p_i, które pozwolą na realizację naszego pomysłu w praktyce.

Twierdzenie 10 Niech $f(X) = f_{n-1}X^{n-1}+\cdots+f_1X+f_0$, $g(X) = g_{n-1}X^{n-1}+\cdots+g_1X+g_0$ będą takimi wielomianami o współczynnikach całkowitych, że $|f_i| < B$ i $|g_i| < B$. Jeśli liczby pierwsze p_i spełniają następujące warunki:

(1) $p_i \neq p_j$,

(2) $M = \prod_{i=1}^{k} p_i$,

(3) $2nB^2 < \prod p_i = M$,

(4) $p_i = 2^{m+1}r_i + 1$ dla pewnego $2^{m+1} \geq 2n$ i $r_i \in \mathbb{Z}$,

to

$$f(X)g(X) = f(X)g(X) \bmod M = (f(X) \bmod M)(g(X) \bmod M) \bmod M$$

i ciała $\mathbb{F}_{p_i}$ mogą być wykorzystane do równoległego mnożenia wielomianów f i g z użyciem szybkiej transformaty Fouriera (FFT).

Dowód: Dowód twierdzenia będzie bardzo zbliżony do dowodu Twierdzenia 8. Ponieważ operacja mod$M$ jest homomorfizmem naturalnym pierścienia $\mathbb{Z}$ to:

$$(f(X) \bmod M)(g(X) \bmod M) \bmod M = f(X)g(X) \bmod M.$$

Korzystając z Lematu 5 otrzymujemy kolejną równość:

$$f(X)g(X) \bmod M = f(X)g(X).$$

Oznacza to, że mnożenie wielomianów $g(X), f(X) \in \mathbb{Z}[X]$ daje taki sam wynik, co mnożenie wielomianów $g(X) \bmod M, f(X) \bmod M \in (\mathbb{Z}/M\mathbb{Z})[X]$ o ile tylko elementy pierścienia $\mathbb{Z}/M\mathbb{Z}$ są reprezentowane przez liczby ze zbioru

$$\left\{-\frac{M-1}{2}, \ldots, -1, 0, 1, \ldots, \frac{M-1}{2}\right\}.$$

Ponadto zakładaliśmy, że liczba M jest iloczynem różnych liczb pierwszych p_i, co pozwala zastosować twierdzenie chińskie o resztach i otrzymać następujący izomorfizm:

$$\mathbb{Z}/M\mathbb{Z} \simeq \mathbb{F}_{p_1} \times \cdots \times \mathbb{F}_{p_k}.$$

Powyższy izomorfizm może być bardzo łatwo rozszerzony na izomorfizm pierścieni wielomianów:

$$(\mathbb{Z}/M\mathbb{Z})[X] \simeq \mathbb{F}_{p_1}[X] \times \cdots \times \mathbb{F}_{p_k}[X].$$

Oznacza to, że mnożenie w pierścieniu $(\mathbb{Z}/M\mathbb{Z})[X]$ może być zastąpione przez k mnożeń w pierścieniach $\mathbb{F}_{p_i}[X]$, a każde z nich może być wykonywane niezależnie. Daje to możliwość rozdzielenia obliczeń pomiędzy k procesorów i wykonywania ich równolegle. Ponadto wszystkie liczby $p_i = 2^{m+1}r_i + 1$ zostały dobrane w taki sposób, aby możliwe było zrealizowanie mnożenia przy użyciu

algorytmu FFT. Wynika to z faktu, że każde ciało $\mathbb{F}_{p_i}$ zawiera pierwiastek pierwotny z jedności stopnia 2^{m+1}. ■

Pokażemy, że podejście zaproponowane w powyższym twierdzeniu jest o wiele bardziej efektywne niż standardowa metoda zaprezentowana w poprzedniej części. Aby zapewnić maksymalną wydajność obliczeń powinniśmy dobierać takie liczby pierwsze p_i, które mieszczą się w pojedynczym rejestrze procesora. Jest to bardzo łatwe do osiągnięcia w przypadku procesorów 32 i 64-bitowych.

Przypuśćmy teraz, że znaleźliśmy k liczb pierwszych p_i, które mają taką samą liczbę bitów i spełniają założenia Twierdzenia 10. Dla takich liczb prawdziwe jest następujące twierdzenie:

Twierdzenie 11 Niech dane będą dwa wielomiany całkowitoliczbowe stopnia mniejszego od n, dla których wartość bezwzględna każdego współczynnika jest mniejsza od B. Jeśli liczby pierwsze p_i spełniają warunki z Twierdzenia 10 oraz $\lfloor \log_2(p_i) \rfloor = \lfloor \log_2(p_j) \rfloor$, to pomnożenie takich wielomianów przy użyciu algorytmu FFT, wymaga wykonania

$$c_1 k^2 n + kn(2 + 3\log(n)) + c_2 k^2 n$$

mnożeń w ciele $\mathbb{F}_{p_i}$. Gdzie c_1, c_2 są pewnymi stałymi.

Dowód: Ponieważ $\lfloor \log_2(p_i) \rfloor = \lfloor \log_2(p_j) \rfloor$ dla wszystkich i, j, to możemy przyjąć, że koszt wykonania mnożenia w każdym z ciał $\mathbb{F}_{p_i}$ jest taki sam. Operacja mnożenia wielomianów z użyciem FFT składa się z trzech etapów:

1. Redukcja modularna współczynników wymaga wykonania $c_1 k^2 n$ mnożeń w ciele $\mathbb{F}_{p_i}$. Każdy ze współczynników wielomianu ma precyzję k względem rozmiaru liczb p_i. Dlatego pojedynczy współczynnik może być zredukowany przy użyciu $\frac{1}{2} c_1 k$ mnożeń w ciele $\mathbb{F}_{p_i}$ (Algorytm 2). Ponieważ mamy dwa takie wielomiany, każdy z nich ma n współczynników, a liczb pierwszych jest k, to całkowita liczba niezbędnych mnożeń jest równa $\frac{1}{2} c_1 k \cdot 2 \cdot n \cdot k = c_1 k^2 n$.

2. Używamy szybkiego algorytmu mnożenia FFT w ciałach $\mathbb{F}_{p_i}$ dla wszystkich $i \in \{1, \ldots, k\}$:

   (a) transformata Fouriera dwóch wielomianów stopnia mniejszego niż n wymaga wykonania $2n\log(n)$ mnożeń w $\mathbb{F}_{p_i}$ (wielomiany są transformowane do wektorów o $2n$ współczynnikach),

   (b) mnożenie dwóch wektorów o $2n$ współczynnikach wymaga wykonania $2n$ mnożeń w $\mathbb{F}_{p_i}$,

   (c) odwrotna transformata Fouriera dla pojedynczego wektora o $2n$ współczynnikach wymaga $n\log(n)$ mnożeń $\mathbb{F}_{p_i}$.

3. Zastosowanie twierdzenia chińskiego o resztach pozwala na odtworzenie współczynników iloczynu za pomocą c_2k^2n mnożeń w ciele $\mathbb{F}_{p_i}$. Wynika to z faktu, że każde rozwiązanie układu równań $x \equiv a_i \bmod p_i$ może być odtworzone przy użyciu $\frac{1}{2}c_2k^2$ mnożeń w ciele $\mathbb{F}_{p_i}$ (Algorytm 5). Ponieważ musimy odtworzyć $2n$ współczynników, to całkowita liczba niezbędnych mnożeń jest równa $\frac{1}{2}c_2k^2 \cdot 2n = c_2k^2n$.

Dlatego algorytm opisany w Twierdzeniu 10 wymaga wykonania

$$c_1k^2n + kn(2 + 3\log(n)) + c_2k^2n$$

mnożeń w ciele $\mathbb{F}_{p_i}$. ■

Teraz porównamy złożoności obliczeniowe algorytmów opisanych w dowodach Twierdzeń 9 i 11. Aby tego dokonać musimy najpierw wprowadzić jednolitą miarę złożoności obliczeniowej obu algorytmów. Naturalną miarą, która się w tym przypadku pojawia jest liczba mnożeń w ciele $\mathbb{F}_p$. W związku z tym mamy następujący wniosek:

Wniosek 1 Niech p będzie najmniejszą liczbą pierwszą spełniającą założenia Twierdzenia 8 oraz taką, że $\prod_{i=1}^{k} p_i < p$. Dla tak wybranej liczby p złożoność obliczeniowa algorytmu określonego w dowodzie Twierdzenia 11 wynosi

$$\frac{n(2 + 3\log(n))}{ck} + n\frac{c_1 + c_2}{c}$$

mnożeń w ciele $\mathbb{F}_p$. Natomiast złożoność algorytmu określonego w dowodzie Twierdzenia 9 jest równa

$$n(2 + 3\log(n))$$

mnożeń w $\mathbb{F}_p$.

Dowód: Założenie dotyczące wielkości liczby p oznacza, że ma ona k razy więcej bitów niż pojedyncza liczba pierwsza p_i. W związku z tym pojedyncze mnożenie w ciele $\mathbb{F}_p$ jest proporcjonalne do k^2 mnożeń w ciele $\mathbb{F}_{p_i}$. Przyjmiemy, że c jest stałą tej proporcjonalności: to znaczy jedno mnożenie w $\mathbb{F}_p$ jest równoważne ck^2 mnożeniom w $\mathbb{F}_{p_i}$. Dlatego złożoność metody opisanej w dowodzie Twierdzenia 11 wynosi

$$\frac{c_1k^2n + kn(2 + 3\log(n)) + c_2k^2n}{ck^2} = \frac{n(2 + 3\log(n))}{ck} + n\frac{c_1 + c_2}{c}$$

mnożeń w ciele $\mathbb{F}_p$. ■

Otrzymany rezultat pokazuje, że zastosowanie chińskiego twierdzenia o resztach pozwala nie tylko na zrównoleglenie obliczeń, ale również około ck razy przyspiesza działanie samego algorytmu mnożenia wielomianów (z eksperymentów numerycznych wynika, że dla współczynników mających kilkaset bitów

$ck > 1$). Warto w tym miejscu zaznaczyć, że jest to dość nietypowa sytuacja – algorytm równoległy ma mniejszą złożoność niż algorytm sekwencyjny.

Na zakończenie sprawdźmy, jak wypada porównanie nowego algorytmu z metodą, która stosuje szybką transformatę Fouriera zarówno do mnożenia wielomianów, jak i do mnożenia liczb. Jeżeli podtrzymać założenie, że liczby p_i mieszczą się w pojedynczym rejestrze procesora, to złożoność algorytmu mnożącego wielomian i jego współczynniki za pomocą FFT jest rzędu

$$O((n \log n)(k \log k)).$$

Nasz algorytm ma natomiast złożoność

$$O(kn \log n + k^2 n).$$

Jeżeli przyjąć, że $k = O(n)$, to widzimy, że algorytm całkowicie oparty o FFT jest znacznie szybszy. Jego złożoność wynosi $O(n^2 \log^2 n)$ podczas, gdy zaproponowany przez nas algorytm działa w czasie $O(n^3)$. Co jednak się stanie, gdy współczynniki wielomianu będą mniejsze? Załóżmy, że $k = O(\log n)$. Wtedy algorytm oparty całkowicie o FFT ma złożoność $O(n \log^2 n \log \log n)$ podczas gdy nasza metoda ma złożoność asymptotyczną $O(n \log^2 n)$. Nie jest to znacząca różnica, ale niewątpliwie nasz cel został osiągnięty – opracowano efektywny algorytm mnożenia wielomianów, których współczynniki są znacznie niższego rzędu niż stopień.

Wniosek 2 Jeśli $k = O(\log n)$, to złożoność zaproponowanego algorytmu jest mniejsza niż złożoność algorytmu mnożenia opartego jedynie na zastosowaniu FFT i wynosi

$$O(n \log^2 n),$$

podczas gdy złożoność algorytmu bazującego na FFT jest rzędu

$$O(n \log^2 n \log \log n).$$

■

W praktyce osiągnięto jednak znacznie więcej. Jak się okazało podczas eksperymentów numerycznych, zaproponowany algorytm daje ewidentne korzyści już w przypadku, gdy współczynniki wielomianu mają po kilkaset bitów. Oznacza to, że opłaca się go stosować już dla niewielkich wartości k i n.

Rozdział 6

Praktyczna, efektywna implementacja zaproponowanego algorytmu

Mimo, że idea zaproponowanego algorytmu jest relatywnie prosta, to już jego efektywna implementacja nie jest taka oczywista. Szczególną uwagę należy zwrócić na stałe c_1 i c_2 gdyż odgrywają one istotną rolę w praktycznym wykorzystaniu algorytmu. Ich wielkość zależy od efektywności przejścia pomiędzy pierścieniem $\mathbb{Z}/M\mathbb{Z}$ i pierścieniem $\mathbb{F}_{p_1} \times \cdots \times \mathbb{F}_{p_k}$. Z tego powodu w pierwszej kolejności skupimy naszą uwagę na implementacji chińskiego twierdzenia o resztach. Następnie zaproponujemy w jaki sposób można efektywnie realizować szybką transformatę Fouriera w niedużych ciałach skończonych. Na zakończenie przedstawione zostaną wyniki praktycznej implementacji algorytmu przygotowanej na procesor Intel Core 2 Quad.

Opracowane oprogramowanie przeznaczone jest na procesory 32-bitowe, ale może być z łatwością zaadoptowane na inne procesory, w szczególności 64-bitowe. W dalszej części naszych rozważań będziemy zakładali, że $2^{32} > p_i > 2^{31}$. Oczywiście nie mogą to być dowolne liczby pierwsze ze wskazanego przedziału, gdyż będą one wykorzystywane do implementacji szybkiej transformaty Fouriera. Dlatego do przykładowej implementacji zostały wybrane liczby pierwsze, które mają postać:

$$p_i = r_i \cdot 2^{22} + 1, \quad \text{gdzie} \quad 512 < r_i < 1024.$$

Wszystkich takich liczb jest 56, a kolejne r_i mają następujące wartości:

513,	517,	544,	553,	559,	565,	573,	589,
592,	604,	610,	627,	628,	637,	639,	645,
648,	649,	655,	663,	669,	670,	684,	688,
694,	714,	715,	733,	742,	757,	765,	768,
772,	790,	814,	819,	823,	832,	837,	847,
853,	859,	862,	865,	874,	879,	894,	915,
928,	939,	940,	957,	972,	979,	1000,	1014.

Wybrany zbiór liczb pierwszych pozwala na mnożenie wielomianów stopnia mniejszego od $2^{21} = 2\ 097\ 152$, których wartość bezwzględna współczynników nie przekracza

$$B = \sqrt{\frac{p_1 \cdot \cdots \cdot p_{56}}{2 \cdot 2^{21}}} \sim 2^{871}.$$

Warto zaznaczyć, że możliwość manewru w przypadku 32-bitowych liczb pierwszych jest dość ograniczona. Dlatego w przypadku zastosowań wymagających mnożenia jeszcze dłuższych wielomianów o większych współczynnikach uzasadnione jest wykorzystanie 64-bitowych liczb pierwszych. Na przykład zbiór liczb pierwszych postaci

$$\mathcal{P}_{64} = \{p_i : p_i \in \mathcal{P}, p_i = r_i \cdot 2^{32} + 2^{63} + 1, 1 < r_i < 2^{31}\}$$

pozwala na mnożenie wielomianów stopnia mniejszego od $2^{31} = 2\ 147\ 483\ 648$, których współczynniki są rzędu

$$B = \sqrt{\frac{\prod_{p \in \mathcal{P}_{64}} p}{2 \cdot 2^{31}}} \sim \sqrt{\frac{2^{63 \cdot |\mathcal{P}_{64}|}}{2 \cdot 2^{31}}}.$$

Ponieważ $\mathcal{P}_{64}$ jest podzbiorem postępu arytmetycznego spełniającego założenia twierdzenia Dirichleta, to liczność zbioru $\mathcal{P}_{64}$ możemy oszacować przez $\frac{1}{\varphi(2^{32})} \frac{2^{64}}{\ln(2^{64})} = 193\ 635\ 250$. Oznacza to, że wybrany zbiór liczb pierwszych pozwoli na mnożenie wielomianów, których współczynniki są rzędu $2^{6099510377}$.

6.1 Chińskie twierdzenie o resztach

Twierdzenie chińskie o resztach określa izomorfizm pomiędzy pierścieniem $\mathbb{Z}/M\mathbb{Z}$, a pierścieniem $\mathbb{Z}/m_1\mathbb{Z} \times \cdots \times \mathbb{Z}/m_k\mathbb{Z}$ przy założeniu, że spełnione są warunki $M = m_1 \cdot \cdots \cdot m_k$ i $(m_i, m_j) = 1$ dla $i \neq j$. W naszym przypadku wszystkie m_i są różnymi liczbami pierwszymi p_i.

Stwierdzenie 2 Niech a_0, a_1, b i M będą nieujemnymi liczbami całkowitymi. Jeśli $a = a_1 b + a_0$ i $a_1' = a_1 \bmod M$, to

$$a_1 b + a_0 \equiv a_1' b + a_0 \bmod M.$$

Dowód: Dowód wynika bezpośrednio z faktu, że operacja mod$M$ jest homomorfizmem naturalnym pierścienia liczb całkowitych. W związku z tym mamy:

$$\begin{aligned}(a_1 b + a_0) \bmod M &= ((a_1 \bmod M)b + a_0) \bmod M \\ &= (a_1' b + a_0) \bmod M.\end{aligned}$$

■

Stwierdzenie 2 daje nam podstawę do implementacji bardzo prostego i efektywnego algorytmu wyznaczania reszty z dzielenia modulo liczba pojedynczej precyzji. Algorytm 2 ilustruje w jaki sposób należy przeprowadzić taką redukcję.

Data: Liczba całkowita $a = a_k b^k + \cdots + a_1 b + a_0$, $a_i < b$, moduł $M < b$.
Result: Liczba całkowita $a' < M$ taka, że $a' \equiv a \bmod M$.

```
1 begin
2 |  a'_k ← a_k mod M;
3 |  for i = k to 1 do
4 |  |  a'_{i-1} ← (a'_i b + a_{i-1}) mod M;
5 |  end
6 |  return a'_0;
7 end
```

Algorytm 2: Redukcja modularna dla modułu pojedynczej precyzji

Na platformach, dla których czas wykonania instrukcji dzielenia z resztą jest dużo dłuższy niż czas mnożenia pojedynczej precyzji warto zastosować algorytm wyznaczania reszty z dzielenia w sposób rekurencyjny. Poniższy schemat ilustruje ideę działania takiego algorytmu:

$$\begin{array}{lll}
 & & a \bmod m_1 \\
 & a \bmod (m_1 m_2) & \\
 & & a \bmod m_2 \\
a \bmod (m_1 m_2 m_3 m_4) & & \\
 & & a \bmod m_3 \\
 & a \bmod (m_3 m_4) & \\
 & & a \bmod m_4 \\
a & & \\
 & & a \bmod m_5 \\
 & a \bmod (m_5 m_6) & \\
 & & a \bmod m_6 \\
a \bmod (m_5 m_6 m_7 m_8) & & \\
 & & a \bmod m_7 \\
 & a \bmod (m_7 m_8) & \\
 & & a \bmod m_8
\end{array}$$

Data: Liczba całkowita $a = a_k b^k + \cdots + a_1 b + a_0$, $a_i < b$, zbiór modułów $\{m_1, \ldots, m_l\}$ takich, że $m_i < b$.
Result: Liczby całkowite $a'_i < m_i$ takie, że $a'_i \equiv a \bmod m_i$.

```
1  begin
2      if l = 1 then
3          return a mod m_1;
4      else
5          a' ← a mod (m_1 · ··· · m_l);
6          s_1 ← wywołaj ten algorytm dla a' i {m_1, ..., m_{l/2}};
7          s_2 ← wywołaj ten algorytm dla a' i {m_{l/2+1}, ..., m_l};
8          return (s_1, s_2);
9      end
10 end
```

Algorytm 3: Rekurencyjna redukcja modularna dla wielu modułów pojedynczej precyzji

Zmiana sposobu wyznaczania reszty z dzielenia nie zmienia złożoności obliczeniowej algorytmu. Pozwala jednak zmniejszyć liczbę wykonywanych dzieleń całkowitoliczbowych, które zostają zastąpione mnożeniami. Opłacalność wykorzystania takiego podejścia jest jednak ściśle uzależniona od możliwości obliczeniowych danej platformy. Dlatego też nie można z góry stwierdzić, która z opisanych tutaj metod redukcji okaże się lepsza. W przypadku implementacji przygotowanej na procesor Intel Core 2 Quad okazało się, że czas działania obu algorytmów różni się o niespełna 2%. Przy tak niewielkiej różnicy warto więc zastosować Algorytm 2, który jest przejrzysty i bardzo prosty w implementacji.

Skoro wiemy jak efektywnie przejść do reprezentacji resztowej, to czas wybrać efektywną metodę odtworzenia liczby zakodowanej za pomocą względnie pierwszych reszt. Biorąc pod uwagę fakt, że zaproponowany w pracy zestaw modułów p_i jest stały stwierdzono, że najlepszy do tego celu będzie algorytm zaproponowany przez Garnera [20]. Jest to algorytm, który działa bardzo szybko, pod warunkiem poprzedzenia go wstępnym obliczeniem stałych charakterystycznych dla odpowiednich modułów.

Algorytm Garnera w swojej podstawowej wersji został zaprezentowany jako Algorytm 4. Zauważmy, że pierwsza pętla tego algorytmu nie zależy zupełnie od argumentów, jakie zostały podane, a jedynie od wartości poszczególnych modułów. Ponadto każda stała C_i zależy jedynie od modułów $p_1, \ldots, p_i$. Jest to bardzo dobra sytuacja z naszego punktu widzenia. Oznacza bowiem, że stałe te mogą zostać wyznaczone dla wybranego zestawu modułów i zapisane w programie. Procedura mnożąca wielomiany będzie po prostu brała niezbędną liczbę modułów $p_1, \ldots, p_l$ oraz odpowiadające im stałe C_i. Drugim elementem, który jest niezależny od argumentów, są iloczyny $\prod_{j=1}^{i-1} p_j$. One również mogą być wyznaczone w fazie preobliczeń i zapisane w programie. Wyeliminowanie obliczeń, które mogą być wykonane w fazie wstępnej daje nam znacznie szybszy

Algorytm 5.

Data: Układ równań $x \equiv a_i \bmod p_i$, $M = p_1 \cdot \dots \cdot p_k$.
Result: Liczba całkowita $a < M$ taka, że $a \equiv a_i \bmod p_i$ dla każdego $i \in \{1, \dots, k\}$.

```
1  begin
2      for i = 2 to k do
3          C_i ← 1;
4          for j = 1 to i − 1 do
5              u ← p_j^{−1} mod p_i;
6              C_i ← u · C_i mod p_i;
7          end
8      end
9      u ← a_1;
10     a ← a_1;
11     for i = 2 to k do
12         u ← (v_i − a)C_i mod p_i;
13         a ← a + u · ∏_{j=1}^{i−1} p_j;
14     end
15     return a;
16 end
```

Algorytm 4: Algorytm Garnera wyznaczania rozwiązania układu kongruencji liniowych

Data: Układ równań $x \equiv a_i \bmod p_i$, $M = p_1 \cdot \dots \cdot p_k$, $P_i = p_1 \cdot \dots \cdot p_{i-1}$, $C_i = P_i^{-1} \bmod p_i$.
Result: Liczba całkowita $a < M$ taka, że $a \equiv a_i \bmod p_i$ dla każdego $i \in \{1, \dots, k\}$.

```
1  begin
2      u ← a_1;
3      a ← a_1;
4      for i = 2 to k do
5          u ← (v_i − a)C_i mod p_i;
6          a ← a + u · P_i;
7      end
8      return a;
9  end
```

Algorytm 5: Algorytm Garnera wyznaczania rozwiązania układu kongruencji liniowych z preobliczeniami

6.2 Szybka transformata Fouriera z wykorzystaniem ciał skończonych

Rozważania dotyczące implementacji szybkiej transformaty Fouriera zacznijmy od prostego przykładu, który ilustruje własność kluczową z punktu widzenia efektywnej implementacji tego przekształcenia.

Przykład 6 Rozważmy wielomian $X^8 - 1 \in \mathbb{C}[X]$. Wielomian ten ma 8 pierwiastków, którymi są kolejne potęgi liczby $\omega_8 = e^{2\pi i\ /8}$. W związku z tym $X^8 - 1$ rozkłada na czynniki liniowe.

$$
\begin{array}{ccccc}
 & & & (X-1) & = (X-\omega_8^0) \\
 & & (X^2-1) & & \\
 & & & (X+1) & = (X-\omega_8^4) \\
 & (X^4-1) & & & \\
 & & & (X-\omega_8^2) & = (X-\omega_8^2) \\
 & & (X^2+1) & & \\
 & & & (X+\omega_8^2) & = (X-\omega_8^6) \\
(X^8-1) & & & & \\
 & & & (X-\omega_8) & = (X-\omega_8^1) \\
 & & (X^2-\omega_8^2) & & \\
 & & & (X+\omega_8) & = (X-\omega_8^5) \\
 & (X^4+1) & & & \\
 & & & (X-\omega_8^3) & = (X-\omega_8^3) \\
 & & (X^2+\omega_8^2) & & \\
 & & & (X+\omega_8^3) & = (X-\omega_8^7)
\end{array}
$$

Kolejność czynników na powyższym diagramie nie została dobrana przypadkowo. Okazuje się, że dla każdego wielomianu postaci $X^{2^m} - 1$ można tak je uporządkować, aby iloczyn kolejnych dwóch był dwumianem. Ta własność, jak się później okaże, jest bardzo istotna z punktu widzenia implementacji szybkiej transformaty Fouriera. □

W dalszej części naszych rozważań będziemy zakładali, że K jest ciałem zawierającym pierwiastek pierwotny ω stopnia 2^m z jedności. Obserwację poczynioną Przykładzie 6 możemy sformalizować w formie następującego lematu.

Lemat 6 Jeżeli $\Phi_{0,k} = X - \omega^{l_k}$, gdzie $l_k = \sum_{j=0}^{m-1} \left(\left\lfloor \frac{k}{2^j} \right\rfloor \mod 2 \right) \cdot 2^{m-1-j}$, to wszystkie wyrażenia

$$\Phi_{j,k} = \Phi_{j-1,2k}\Phi_{j-1,2k+1}$$

są dwumianami o niezerowym wyrazie wolnym i stopniu równym 2^j. Ponadto

istnieje taki element $\alpha \in \langle\omega\rangle$, że

$$\begin{aligned} \Phi_{j,k} &= X^{2^j} - \alpha^{2^j}, \\ \Phi_{j-1,2k} &= X^{2^{j-1}} - \alpha^{2^{j-1}}, \\ \Phi_{j-1,2k+1} &= X^{2^{j-1}} + \alpha^{2^{j-1}}. \end{aligned}$$

Zanim przejdziemy do dowodu tego lematu wyjaśnimy jaki jest faktyczny związek pomiędzy potęgą pierwiastka z jedności, a pozycją na której powinien być ustawiony. Zawarty w treści lematu wzór $l_k = \sum_{j=0}^{m-1} \left(\lfloor \frac{k}{2^j} \rfloor \mod 2\right) \cdot 2^{m-1-j}$, choć mało czytelny, wyraża bardzo prostą zależność. Jeżeli bowiem przedstawimy liczbę k w jej m-bitowej reprezentacji $\sum_{j=0}^{m-1} b_j 2^j$, to liczba l_k jest niczym innym jak $\sum_{j=0}^{m-1} b_{m-1-j} 2^j$. Oznacza to, że liczba l_k powstaje z liczby k poprzez odwrócenie kolejności bitów reprezentacji.

Dowód: Rozwijając wzór rekurencyjny na wyrażenie $\Phi_{j,k}$ otrzymujemy związek

$$\Phi_{j,k} = \prod_{i=2^j k}^{2^j(k+1)-1} \Phi_{0,i} = \prod_{i=2^j k}^{2^j(k+1)-1} \left(X - \omega^{l_i}\right).$$

Na mocy uwagi poczynionej przed dowodem lematu możemy stwierdzić, że skoro i przebiega wszystkie liczby ze zbioru $\{2^j k + r : 0 \leq r < 2^j\}$, to wykładniki l_i przebiegają wszystkie liczby ze zbioru $\{2^{m-j} r + k' : 0 \leq r < 2^j\}$. Liczba k' powstaje z liczby k poprzez zamianę kolejności bitów i co do wartości jest równa $l_{2^j k}$. Możemy zatem zapisać, że

$$\Phi_{j,k} = \prod_{r=0}^{2^j-1} \left(X - \omega^{2^{m-j} r + k'}\right).$$

Przyjmując $\alpha = \omega^{k'}$ i $\beta = \omega^{2^{m-j}}$ upraszczamy powyższe wyrażenie do postaci

$$\Phi_{j,k} = \prod_{r=0}^{2^j-1} (X - \alpha\beta^r) = \alpha^{2^j} \prod_{r=0}^{2^j-1} \left(\frac{X}{\alpha} - \beta^r\right).$$

Ale potęgi elementu β generują wszystkie pierwiastki stopnia 2^j z jedności. Oznacza to, że ostatni iloczyn w powyższej formule reprezentuje wielomian $(X/\alpha)^{2^j} - 1$ i ostatecznie wzór na $\Phi_{j,k}$ upraszcza się do postaci

$$\Phi_{j,k} = X^{2^j} - \alpha^{2^j} = X^{2^j} - \omega^{2^j k'},$$

co kończy dowód pierwszej części lematu oraz równości dla $\Phi_{j,k}$. Dla dowodu drugiej części rozwiniemy wzór rekurencyjny na $\Phi_{j-1,2k}$

$$\Phi_{j-1,2k} = \prod_{i=2^j k}^{2^j k + 2^{j-1} - 1} \Phi_{0,i} = \prod_{i=2^j k}^{2^j k + 2^{j-1} - 1} \left(X - \omega^{l_i}\right).$$

Podobnie, jak w przypadku wielomianu $\Phi_{j,k}$ stwierdzamy, że skoro i przebiega wszystkie liczby ze zbioru $\{2^j k + r : 0 \leq r < 2^{j-1}\}$, to wykładniki l_i przebiegają wszystkie liczby ze zbioru $\{2^{m-j+1} r + k' : 0 \leq r < 2^{j-1}\}$. Możemy więc zapisać

$$\Phi_{j-1,2k} = \prod_{r=0}^{2^{j-1}-1} \left(X - \alpha\beta^{2r}\right) = \alpha^{2^{j-1}} \prod_{r=0}^{2^{j-1}-1} \left(\frac{X}{\alpha} - (\beta^2)^r\right).$$

Ponieważ elementy $\beta^2 = \omega^{2^{m-j+1}}$ generują wszystkie pierwiastki stopnia 2^{j-1}, to ostatecznie otrzymujemy

$$\Phi_{j-1,2k} = X^{2^{j-1}} - \alpha^{2^{j-1}}.$$

Wzór na $\Phi_{j-1,2k+1}$ uzyskujemy jako iloraz $\Phi_{j,k}/\Phi_{j-1,2k}$. ■

Przykład 7 Zobaczmy jak działa wprowadzony lemat w praktyce. Niech naszymi pierwiastkami z jedności będą kolejne potęgi liczby $\omega = e^{2\pi i/8}$.

$$\begin{array}{ll}
k = 0 = (0,0,0)_2 & l_0 = (0,0,0)_2 = 0 \\
k = 1 = (0,0,1)_2 & l_1 = (1,0,0)_2 = 4 \\
k = 2 = (0,1,0)_2 & l_2 = (0,1,0)_2 = 2 \\
k = 3 = (0,1,1)_2 & l_3 = (1,1,0)_2 = 6 \\
k = 4 = (1,0,0)_2 & l_4 = (0,0,1)_2 = 1 \\
k = 5 = (1,0,1)_2 & l_5 = (1,0,1)_2 = 5 \\
k = 6 = (1,1,0)_2 & l_6 = (0,1,1)_2 = 3 \\
k = 7 = (1,1,1)_2 & l_7 = (1,1,1)_2 = 7
\end{array}$$

Jak można było przypuszczać, otrzymana kolejność pierwiastków jest taka sama, jak w poprzednim przykładzie. □

Lemat 7 Niech K będzie ciałem, $x \in K$, a wielomiany $A, B, C, R \in K[X]$ spełniają warunki

$$\begin{aligned}
A \mod B &= R, \\
B \mod C &= 0.
\end{aligned}$$

Wtedy prawdziwe są następujące równości

$$A(x) = A \mod (X - x) \qquad \text{i} \qquad A \mod C = R \mod C.$$

Dowód: Dla dowodu pierwszej tożsamości przyjmijmy, że $A(X) = \sum_{j=0}^{n-1} a_j X^j$. Poniższa równość

$$X^k = (X^{k-1} + xX^{k-2} + \cdots + x^{k-2}X + x^{k-1})(X - x) + x^k,$$

pokazuje, że $X^k \mod (X-x) = x^k$. Wykorzystując teraz fakt, że operacja mod jest homomorfizmem naturalnym pierścienia $K[X]$ otrzymujemy zależność

$$\begin{aligned} A \mod (X-x) &= \left(\sum_{j=0}^{n-1} a_j X^j\right) \mod (X-x) \\ &= \sum_{j=0}^{n-1} a_j \left(X^j \mod (X-x)\right) \\ &= \sum_{j=0}^{n-1} a_j x^j = A(x), \end{aligned}$$

co kończy dowód pierwszej części lematu. Dla dowodu drugiej części zauważmy, że skoro $A \mod B = R$, to istnieje taki wielomian $D \in K[X]$, dla którego $A = D \cdot B + R$. Uwzględniając warunek $B \mod C = 0$ otrzymujemy ostatecznie

$$\begin{aligned} A \mod C &= (D \cdot B + R) \mod C \\ &= (D \mod C)(B \mod C) + (R \mod C) \\ &= R \mod C, \end{aligned}$$

co kończy dowód lematu. ■

Przeprowadzone rozważania dają nam możliwość udowodnienia twierdzenia o szybkiej transformacie Fouriera. Twierdzenie 12 i jego dowód zostały opracowane w ścisłym związku z iteracyjną wersją implementowanego algorytmu FFT. Dlatego stanowią one teoretyczne uzasadnienie poprawności Algorytmu 6.

Twierdzenie 12 Niech $A \in K[X]$ będzie wielomianem stopnia mniejszego od $n = 2^m$, którego wartości w pierwiastkach z jedności mają być obliczone. Przyjmijmy, tak jak w Lemacie 6, że $\Phi_{0,k} = X - \omega^{l_k}$ dla

$$l_k = \sum_{j=0}^{m-1} \left(\left\lfloor \frac{k}{2^j} \right\rfloor \mod 2 \right) \cdot 2^{m-1-j}$$

oraz

$$\Phi_{j,k} = \Phi_{j-1,2k} \Phi_{j-1,2k+1}.$$

Jeżeli ciąg $A_{j,k}$ zdefiniowany jest jako

$$A_{j,k} = A_{j+1,\lfloor k/2 \rfloor} \mod \Phi_{j,k} \qquad \text{i} \qquad A_{m,0} = A,$$

to wszystkie jego wyrazy można wyznaczyć wykonując $\frac{1}{2}mn$ mnożeń w ciele K i $A_{0,k} = A(\omega^{l_k})$.

Dowód: Najpierw wykażemy, że $A_{0,k} = A(\omega^{l_k})$. W tym celu przeanalizujemy ciąg operacji, który prowadzi do wyznaczenia wyrazu $A_{0,k}$.

$$\begin{aligned} A_{0,k} &= \left(A_{1,\lfloor k/2 \rfloor} \mod \Phi_{0,k}\right) \\ &= \left(A_{2,\lfloor k/2^2 \rfloor} \mod \Phi_{1,\lfloor k/2 \rfloor}\right) \mod \Phi_{0,k} \\ &\vdots \\ &= \left(\left(\ldots\left(A_{m,\lfloor k/2^m \rfloor} \mod \Phi_{m,\lfloor k/2^{m-1} \rfloor}\right)\ldots\right) \mod \Phi_{1,\lfloor k/2 \rfloor}\right) \mod \Phi_{0,k} \end{aligned}$$

Biorąc pod uwagę, że $k < n = 2^m$ i $A_{m,0} = A$ otrzymujemy związek

$$A_{0,k} = \left(\left(\ldots\left(A \mod \Phi_{m,\lfloor k/2^{m-1} \rfloor}\right)\ldots\right) \mod \Phi_{1,\lfloor k/2 \rfloor}\right) \mod \Phi_{0,k}.$$

Z rekurencyjnej definicji $\Phi_{j,k}$ wynika zależność $\Phi_{j,k} \mid \Phi_{j+1,\lfloor k/2 \rfloor}$, która prowadzi do ciągu podzielności

$$\Phi_{0,k} \mid \Phi_{1,\lfloor k/2 \rfloor} \mid \cdots \mid \Phi_{m,\lfloor k/2^{m-1} \rfloor}.$$

Stosując Lemat 7 otrzymujemy zatem

$$A_{0,k} = A \mod \Phi_{0,k} = A(\omega^{l_k}).$$

W celu oszacowania liczby niezbędnych mnożeń zauważmy, że wielomiany $A_{j-1,2k}$ i $A_{j-1,2k+1}$ powstają przez redukcję wielomianu $A_{j,k}$ modulo dwumiany

$$\begin{aligned} \Phi_{j-1,2k} &= X^{2^{j-1}} - \alpha^{2^{j-1}}, \\ \Phi_{j-1,2k+1} &= X^{2^{j-1}} + \alpha^{2^{j-1}}. \end{aligned}$$

Jednoczesne wykonanie obu tych redukcji jest bardzo łatwe do przeprowadzenia i wymaga wykonania 2^{j-1} mnożeń

$$\begin{aligned} A_{j,k} \mod \left(X^{2^{j-1}} \mp \alpha^{2^{j-1}}\right) &= \left(\sum_{i=0}^{2^j-1} a_i X^i\right) \mod \left(X^{2^{j-1}} \mp \alpha^{2^{j-1}}\right) \\ &= \sum_{i=0}^{2^{j-1}-1} a_i X^i \pm \alpha^{2^{j-1}} \sum_{i=0}^{2^{j-1}-1} a_{2^{j-1}+i} X^i \\ &= \sum_{i=0}^{2^{j-1}-1} \left(a_i \pm \alpha^{2^{j-1}} \cdot a_{2^{j-1}+i}\right) X^i. \end{aligned}$$

Oznacza to, że na każdym poziomie rekurencji wykonujemy $\frac{1}{2}n$ mnożeń. Ponieważ wszystkich poziomów rekurencji jest m, to całkowitą liczbę mnożeń szacujemy przez $\frac{1}{2}mn$. ■

Przykład 8 Tym razem nasze rozważania będziemy prowadzili w ciele skończonym $\mathbb{F}_{17}$. Wszystkie niezerowe elementy tego ciała są pierwiastkami stopnia

16 z jedności. Do naszego przykładu wykorzystamy jedynie pierwiastki stopnia 4, którymi są $\omega^0 = 1, \omega^1 = 13, \omega^2 = 16, \omega^3 = 4$. Z dotychczasowych rozważań wynika następująca hierarchia wielomianów $\Phi_{j,k}$.

$$\begin{array}{ccc} & & \Phi_{0,0} = X - 1 \\ & \Phi_{1,0} = X^2 - 1 & \\ & & \Phi_{0,1} = X - 16 \\ \Phi_{2,0} = X^4 - 1 & & \\ & & \Phi_{0,2} = X - 13 \\ & \Phi_{1,1} = X^2 - 16 & \\ & & \Phi_{0,3} = X - 4 \end{array}$$

Powiedzmy, że chcemy znaleźć wartości wielomianu $A(X) = X^3 + 2X^2 + 3X + 4$ w punktach $\omega^0, \ldots, \omega^3$. Postępując zgodnie z procedurą opisaną w poprzednim twierdzeniu otrzymujemy następujący ciąg wielomianów $A_{j,k}$.

$$\begin{aligned} A_{2,0} &= X^3 + 2X^2 + 3X + 4 \\ \\ A_{1,0} &= A_{2,0} \mod \Phi_{1,0} = (X^3 + 2X^2 + 3X + 4) \mod (X^2 - 1) \\ &= 4X + 6 \\ A_{1,1} &= A_{2,0} \mod \Phi_{1,1} = (X^3 + 2X^2 + 3X + 4) \mod (X^2 - 16) \\ &= 2X + 2 \\ \\ A_{0,0} &= A_{1,0} \mod \Phi_{0,0} = (4X + 6) \mod (X - 1) \\ &= 10 = A(1) \\ A_{0,1} &= A_{1,0} \mod \Phi_{0,1} = (4X + 6) \mod (X - 16) \\ &= 2 = A(16) \\ A_{0,2} &= A_{1,1} \mod \Phi_{0,2} = (2X + 2) \mod (X - 13) \\ &= 11 = A(13) \\ A_{0,3} &= A_{1,1} \mod \Phi_{0,2} = (2X + 2) \mod (X - 4) \\ &= 10 = A(4) \end{aligned}$$

□

Data: Wielomian $v = \sum_{i=0}^{n-1} v_i X^i$ dla n będącego potęgą liczby 2, pierwiastek pierwotny $\omega \in \mathbb{F}_p$ stopnia n z jedności, potęgi pierwiastka pierwotnego $r_i = \omega^{l_i}$.
Result: Transformata wielomianu v jako wektor $(v_0, \ldots, v_{n-1})$ taki, że $v_i = v(\omega^{l_i})$.

```
1  begin
2      m ← n;
3      while m ≠ 1 do
4          k ← n/m;
5          m ← m/2;
6          for i = 0 to k − 1 do
7              Redukcja modulo dwumian X^m + ω^{l_k} i dwumian X^m − ω^{l_k};
8              for j = 0 to m − 1 do
9                  t ← v_{(2i+1)m+j} · r_i mod p;
10                 v_{(2i+1)m+j} ← v_{2im+j} − t mod p;
11                 v_{2im+j} ← v_{2im+j} + t mod p;
12             end
13         end
14     end
15     return (v_0, ..., v_{n−1});
16 end
```

Algorytm 6: Transformata Fouriera nad ciałem skończonym $\mathbb{F}_p$

Dla zupełności naszych rozważań przedstawię teraz Twierdzenie 13 o odwrotnej transformacie Fouriera. Jest ono teoretycznym uzasadnieniem poprawności Algorytmu 7.

Twierdzenie 13 Przyjmijmy, tak jak w Lemacie 6, że $\Phi_{0,k} = X - \omega^{l_k}$ dla

$$l_k = \sum_{j=0}^{m-1} \left(\left\lfloor \frac{k}{2^j} \right\rfloor \mod 2 \right) \cdot 2^{m-1-j}$$

oraz

$$\Phi_{j,k} = \Phi_{j-1,2k} \Phi_{j-1,2k+1}.$$

Niech $A \in K[X]$ będzie wielomianem stopnia mniejszego od $n = 2^m$, którego wartości w pierwiastkach z jedności $\omega^0, \omega^1, \ldots, \omega^{n-1}$ są znane. Jeżeli ciąg $A_{j,k}$ zdefiniowany jest jako

$$A_{j,k} = A_{j+1,\lfloor k/2 \rfloor} \mod \Phi_{j,k} \qquad \text{i} \qquad A_{0,k} = A(\omega^{l_k}),$$

to wszystkie jego wyrazy można wyznaczyć wykonując $\frac{1}{2}mn$ mnożeń w ciele K i $A_{m,0} = A$.

Dowód: Z Lematu 6 wynika, że dwumiany $\Phi_{j,k}$ mają postać $X^{2^j} - \alpha^{2^j}$, gdzie element α jest zadany dla każdego z tych dwumianów osobno. Ponadto wiemy,

że

$$\Phi_{j-1,2k} = X^{2^{j-1}} - \alpha^{2^{j-1}} \quad \text{i} \quad \Phi_{j-1,2k+1} = X^{2^{j-1}} + \alpha^{2^{j-1}}.$$

Wyznaczenie wyrazów ciągu $A_{j,k}$ przy pomocy formuły podanej w treści twierdzenia jest niewykonalne, ponieważ dysponujemy jedynie wyrazami $A_{0,k}$. Potrzebujemy zatem warunku, który byłby równoważny i pozwalał na odtwarzanie ciągu w kierunku przeciwnym. Sprawdzimy teraz, że takim warunkiem jest

$$\begin{aligned} A_{j,k} &= \frac{1}{2}\left(A_{j-1,2k} + A_{j-1,2k+1}\right) + \frac{X^{2^{j-1}}}{2\alpha^{2^{j-1}}}\left(A_{j-1,2k} - A_{j-1,2k+1}\right) \\ &= \frac{1}{2}\left(\left(A_{j-1,2k} + A_{j-1,2k+1}\right) + \frac{X^{2^{j-1}}}{\alpha^{2^{j-1}}}\left(A_{j-1,2k} - A_{j-1,2k+1}\right)\right). \end{aligned}$$

Dla dowodu słuszności powyższej formuły wystarczy wykazać, że prawdziwe są zależności

$$\begin{aligned} A_{j-1,2k} &= A_{j,k} \bmod \Phi_{j-1,2k}, \\ A_{j-1,2k+1} &= A_{j,k} \bmod \Phi_{j-1,2k+1}. \end{aligned}$$

Biorąc pod uwagę postać dwumianów $\Phi_{j-1,2k}$ i $\Phi_{j-1,2k+1}$ mamy

$$\begin{aligned} A_{j,k} \mod \Phi_{j-1,2k} &= \frac{1}{2}\left(A_{j-1,2k} + A_{j-1,2k+1}\right) \bmod \left(X^{2^{j-1}} - \alpha^{2^{j-1}}\right) + \\ &\quad \frac{X^{2^{j-1}}}{2\alpha^{2^{j-1}}}\left(A_{j-1,2k} - A_{j-1,2k+1}\right) \bmod \left(X^{2^{j-1}} - \alpha^{2^{j-1}}\right) \\ &= \frac{1}{2}\left(A_{j-1,2k} + A_{j-1,2k+1}\right) + \frac{1}{2}\left(A_{j-1,2k} - A_{j-1,2k+1}\right) \\ &= A_{j-1,2k} \end{aligned}$$

$$\begin{aligned} A_{j,k} \mod \Phi_{j-1,2k+1} &= \frac{1}{2}\left(A_{j-1,2k} + A_{j-1,2k+1}\right) \mod \left(X^{2^{j-1}} + \alpha^{2^{j-1}}\right) + \\ &\quad \frac{X^{2^{j-1}}}{2\alpha^{2^{j-1}}}\left(A_{j-1,2k} - A_{j-1,2k+1}\right) \mod \left(X^{2^{j-1}} + \alpha^{2^{j-1}}\right) \\ &= \frac{1}{2}\left(A_{j-1,2k} + A_{j-1,2k+1}\right) - \frac{1}{2}\left(A_{j-1,2k} - A_{j-1,2k+1}\right) \\ &= A_{j-1,2k+1}. \end{aligned}$$

Teraz wystarczy zauważyć, że w celu wyznaczenia każdego z wielomianów $A_{j,k}$ trzeba wykonać 2^{j-1} mnożeń przez element $\alpha^{-2^{j-1}}$. Mnożenia przez $1/2$ pomijamy, gdyż można je wciągnąć poza rekurencję. Dlatego wykonanie transformaty odwrotnej pochłania $\frac{1}{2}mn$ mnożeń w ciele K. ■

Data: Transformata wielomianu v jako wektor $(v_0, \ldots, v_{n-1})$ dla n będącego potęgą liczby 2, pierwiastek pierwotny $\omega \in \mathbb{F}_p$ stopnia n z jedności, potęgi pierwiastka pierwotnego $r_i = \omega^{-l_i}$.
Result: Wielomian $v = \sum_{i=0}^{n-1} v_i X^i$ taki, że $v_i = v(\omega^{l_i})$.

```
1  begin
2      m ← 1;
3      s ← 1;
4      while m < n do
5          k ← n/(2m);
6          s ← s/2 mod p;
7          for i = 0 to k − 1 do
8              for j = 0 to m − 1 do
9                  t ← v_{2im+j} + v_{(2i+1)m+j} mod p;
10                 v_{(2i+1)m+j} ← v_{2im+j} − v_{(2i+1)m+j} mod p;
11                 v_{2im+j} ← t;
12                 v_{(2i+1)m+j} ← v_{(2i+1)m+j} · r_i mod p;
13             end
14         end
15         m ← 2m;
16     end
17     for i = 0 to n − 1 do
18         v_i ← s · v_i mod p;
19     end
20     return Σ_{i=0}^{n−1} v_i X^i;
21 end
```

Algorytm 7: Odwrotna transformata Fouriera nad ciałem skończonym $\mathbb{F}_p$

6.3 Mnożenie wielomianów przy użyciu szybkiej transformaty Fouriera i twierdzenia chińskiego o resztach

Wykorzystując transformatę Fouriera jesteśmy w stanie szybko wyznaczyć wartości wielomianu $f(X) = \sum_{i=0}^{d_f} f_i X^i$ w punktach będących pierwiastkami z jedności. Jeżeli $\omega_0, \ldots, \omega_{n-1}$ są ustalonymi pierwiastkami z jedności stopnia n, to wynikiem działania transformaty Fouriera jest wektor

$$t_f = (f(\omega_0), \ldots, f(\omega_{n-1})).$$

Otrzymujemy w ten sposób alternatywną reprezentację wielomianu f – reprezentację przez wartości w punktach. Podobnie możemy uczynić z wielomianem $g(X) = \sum_{i=0}^{d_g} g_i X^i$, dla którego po wykonaniu transformaty Fouriera otrzymujemy

$$t_g = (g(\omega_0), \ldots, g(\omega_{n-1})).$$

Reprezentacja wielomianu za pomocą jego wartości w punktach jest bardzo efektywna pod względem obliczeniowym. Wynika to z faktu, że iloczyn wielomianów wyznaczamy przez wymnożenie wartości w odpowiadających sobie punktach ω_i. Dlatego też iloczyn wielomianów $h = fg$ reprezentujemy jako

$$t_h = (f(\omega_0)g(\omega_0), \ldots, f(\omega_{n-1})g(\omega_{n-1})).$$

Teraz wystarczy wykonać transformatę odwrotną wektora t_h, aby otrzymać współczynniki iloczynu gh.

Data: Wielomiany $f = \sum_{i=0}^{d_f} f_i X^i$, $g = \sum_{i=0}^{d_g} g_i X^i$ takie, że $f_i, g_i \in \mathbb{Z}$. Zbiór liczb pierwszych $\{p_1, \ldots, p_k\}$ spełniających założenia twierdzenia 10.
Result: Wielomian $h \in \mathbb{Z}[X]$ taki, że $h = fg$.

begin
  $n \leftarrow 2^{\lceil \log_2(d_f + d_g + 1) \rceil}$;
  for $j = 1$ **to** k **do**
    Redukcja współczynników modulo p_j;
    $r_f \leftarrow (f_0 \bmod p_j, \ldots, f_{d_f} \bmod p_j, 0, \ldots, 0)$;
    $r_g \leftarrow (g_0 \bmod p_j, \ldots, g_{d_g} \bmod p_j, 0, \ldots, 0)$;
    Transformata Fouriera;
    $t_f \leftarrow FFT(r_f, n, p_j)$;
    $t_g \leftarrow FFT(r_g, n, p_j)$;
    Mnożenie współczynnik po współczynniku;
    $t_h \leftarrow (t_{f,0}t_{g,0} \bmod p_j, \ldots, t_{f,n-1}t_{g,n-1} \bmod p_j)$;
    Odwrotna transformata Fouriera;
    $h_j \leftarrow FFT^{-1}(t_h, n, p_j)$;
  end
  $h \leftarrow 0$;
  Rekonstrukcja współczynników iloczynu;
  for $j = 0$ **to** $n - 1$ **do**
    $h \leftarrow h + CRT^{-1}(h_{1,j}, \ldots, h_{k,j}, p_1, \ldots, p_k)X^j$;
  end
  return h;
end

Algorytm 8: Mnożenie wielomianów z wykorzystaniem transformaty Fouriera i twierdzenia chińskiego o resztach

Należy przy tym pamiętać, że o ile reprezentacja przez współczynniki niesie ze sobą informację na temat stopnia wielomianu, to reprezentacja przez wartości w punktach gubi tą informację. Dlatego też trzeba zadbać, aby wartości obu wielomianów zostały wyznaczone w takiej liczbie punktów, która zagwarantuje jednoznaczne odtworzenie iloczynu. Ponieważ stopnie wielomianów wynoszą odpowiednio d_f i d_g, to transformatę Fouriera wykonujemy na $n = 2^{\lceil \log_2(d_f + d_g + 1) \rceil}$

pierwiastkach z jedności. Liczba n jest w tym przypadku najmniejszą potęgą dwójki, która spełnia nierówność $\deg(fg) + 1 \leq n$. Spełnienie tej nierówności daje nam pewność, że współczynniki iloczynu zostaną odtworzone w sposób jednoznaczny.

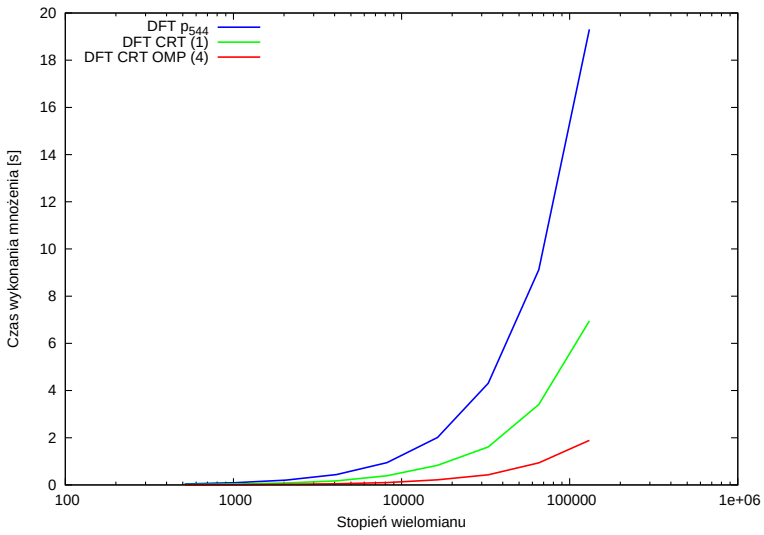
6.4 Wyniki implementacji dla procesorów 32-bitowych

Implementacja szybkiego algorytmu mnożenia wielomianów została przygotowana pod kątem architektury 32-bitowej z wykorzystaniem interfejsu OpenMP. Otrzymane wyniki czasowe okazały się nadzwyczaj dobre. Potwierdzają one w praktyce, że połączenie szybkiej transformaty Fouriera z chińskim twierdzeniem o resztach znacznie przyspiesza wykonywanie obliczeń. W Tabelach 6.1 i 6.2 zestawione są czasy działania algorytmu mnożącego wielomiany tego samego stopnia o współczynnikach z zakresu $[0, 2^{256})$ i $[0, 2^{512})$. Natomiast na Rysunkach 6.1 i 6.2 przedstawiono graficzną interpretację uzyskanych wyników.

Stopień wielomianu	FFT nad $\mathbb{F}_{p_{544}}$ (1 rdzeń)	FFT-CRT nad $\bigotimes_{i=1}^{18} \mathbb{F}_{p_i}$ (1 rdzeń)	FFT-CRT nad $\bigotimes_{i=1}^{18} \mathbb{F}_{p_i}$ (4 rdzenie)		
$n/2-1$	T_1	T_2	T_3	T_1/T_2	T_2/T_3
511	0.0423 s	0.0183 s	0.0052 s	2.3	3.5
1023	0.0930 s	0.0383 s	0.0111 s	2.4	3.4
2047	0.2020 s	0.0803 s	0.0259 s	2.5	3.1
4095	0.4360 s	0.1705 s	0.0481 s	2.6	3.5
8191	0.9370 s	0.3575 s	0.1012 s	2.6	3.5
16383	2.0100 s	0.7444 s	0.2161 s	2.7	3.4
32767	4.2700 s	1.5491 s	0.4283 s	2.8	3.6
65535	9.0700 s	3.2168 s	0.9339 s	2.8	3.4
131071	19.1700 s	6.6716 s	1.8919 s	2.9	3.5

Tablica 6.1: Mnożenie dwóch wielomianów stopnia $n/2-1$ o współczynnikach nie przekraczających 2^{256}

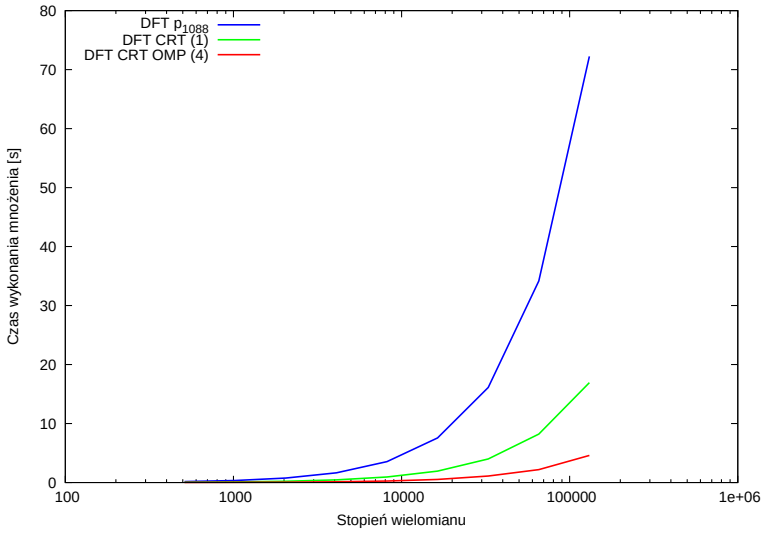
Dla kompletności przedstawionych pomiarów wyznaczony został również czas działania klasycznego algorytmu mnożenia wielomianów. Wyniki porównujące czas działania algorytmu klasycznego z wielordzeniową implementacją opartą na transformacie Fouriera i chińskim twierdzeniu o resztach zostały zebrane w Tabelach 6.3 i 6.4. Na Rysunkach 6.3 i 6.4 przedstawiono graficzną interpretację uzyskanych wyników. Należy zwrócić uwagę, że zarówno współrzędne na osi rzędnych, jak i odciętych są podane w skali logarytmicznej.



Rysunek 6.1: Porównanie szybkości działania algorytmów DFT mnożących wielomiany o współczynnikach mających 256 bitów

Stopień wielomianu	FFT nad $\mathbb{F}_{p_{1088}}$ (1 rdzeń)	FFT-CRT nad $\bigotimes_{i=1}^{36}\mathbb{F}_{p_i}$ (1 rdzeń)	FFT-CRT nad $\bigotimes_{i=1}^{36}\mathbb{F}_{p_i}$ (4 rdzenie)		
$n/2-1$	T_1	T_2	T_3	T_1/T_2	T_2/T_3
511	0.1598 s	0.0511 s	0.0136 s	3.1	3.7
1023	0.3500 s	0.1055 s	0.0280 s	3.3	3.8
2047	0.7600 s	0.2203 s	0.0608 s	3.4	3.6
4095	1.6420 s	0.4562 s	0.1210 s	3.6	3.8
8191	3.5310 s	0.9430 s	0.2527 s	3.7	3.7
16383	7.5500 s	1.9412 s	0.5254 s	3.9	3.7
32767	16.0900 s	3.9944 s	1.0960 s	4.0	3.6
65535	34.1300 s	8.2184 s	2.1926 s	4.1	3.7
131071	72.2100 s	16.9245 s	4.5895 s	4.3	3.7

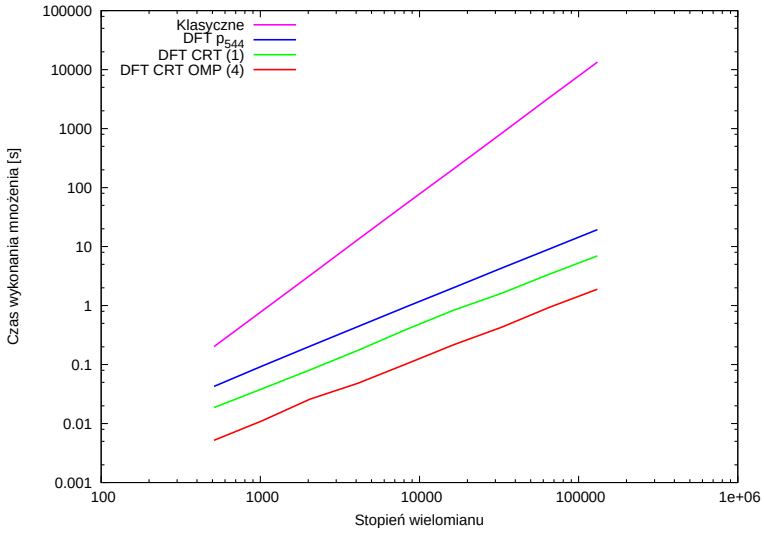
Tablica 6.2: Mnożenie dwóch wielomianów stopnia $n/2-1$ o współczynnikach nie przekraczających 2^{512}



Rysunek 6.2: Porównanie szybkości działania algorytmów DFT mnożących wielomiany o współczynnikach mających 512 bitów

Stopień wielomianu	Klasyczne mnożenie wielomianów (1 rdzeń)	FFT-CRT nad $\bigotimes_{i=1}^{18} \mathbb{F}_{p_i}$ (4 rdzenie)	
$n/2-1$	T_1	T_2	T_1/T_2
511	0.2018 s	0.0052 s	39
1023	0.8074 s	0.0111 s	73
2047	3.2296 s	0.0259 s	125
4095	13.0862 s	0.0481 s	272
8191	52.3449 s	0.1012 s	517
16383	206.6953 s	0.2161 s	956
32767	826.7812 s	0.4283 s	1930
65535	3350.0744 s	0.9339 s	3587
131071	13400.2979 s	1.8919 s	7083

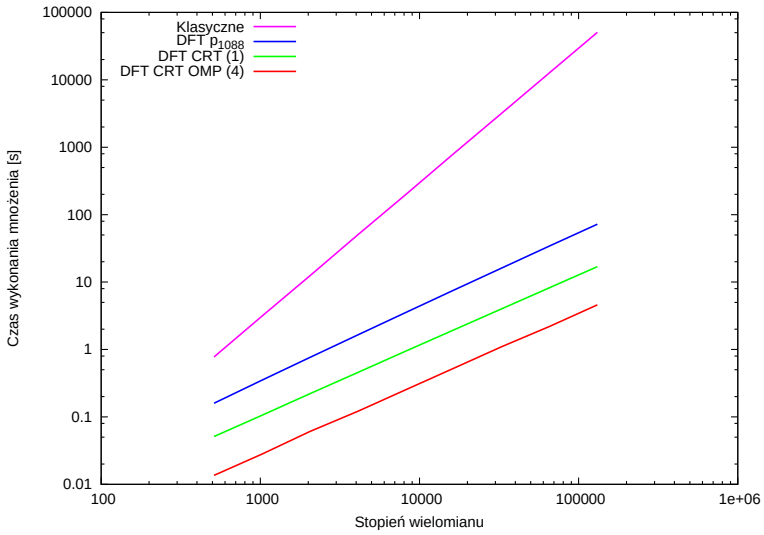
Tablica 6.3: Porównanie klasycznej metody z zaproponowanym algorytmem mnożenia dwóch wielomianów stopnia $n/2-1$ o współczynnikach mających 256 bitów



Rysunek 6.3: Porównanie szybkości działania algorytmów mnożących wielomiany o współczynnikach mających 256 bitów

Stopień wielomianu	Klasyczne mnożenie wielomianów (1 rdzeń)	FFT-CRT nad $\bigotimes_{i=1}^{36} \mathbb{F}_{p_i}$ (4 rdzenie)	
$n/2-1$	T_1	T_2	T_1/T_2
511	0.7759 s	0.0136 s	57
1023	3.1247 s	0.0280 s	112
2047	12.3731 s	0.0608 s	203
4095	50.1638 s	0.1210 s	415
8191	197.9711 s	0.2527 s	783
16383	799.9376 s	0.5254 s	1522
32767	3167.5383 s	1.0960 s	2890
65535	12670.1535 s	2.1926 s	5778
131071	50508.8154 s	4.5895 s	11013

Tablica 6.4: Porównanie klasycznej metody z zaproponowanym algorytmem mnożenia dwóch wielomianów stopnia $n/2-1$ o współczynnikach mających 512 bitów



Rysunek 6.4: Porównanie szybkości działania algorytmów mnożących wielomiany o współczynnikach mających 512 bitów

Rozdział 7

Podsumowanie i wnioski

W rozdziałach 5 i 6 została zawarta kompletna analiza nowego, zaproponowanego przez autora algorytmu mnożenia wielomianów i szeregów potęgowych. Jego konstrukcja została tak pomyślana, aby móc w pełni wykorzystać możliwości obliczeniowe współczesnych procesorów wielordzeniowych. Wykorzystanie twierdzenia chińskiego o resztach pozwala na bardzo łatwe rozdzielenie zadań pomiędzy dostępne wątki. Dodatkowym atutem takiego podejścia jest brak konieczności synchronizowania obliczeń i zapewnienia komunikacji między poszczególnymi wątkami. Z tego powodu algorytm jest bardzo wdzięczny do implementacji wykorzystującej standard programowania równoległego OpenMP. W tabelach 6.1 i 6.2 stosunek T_2/T_3 określa ile procesorów z czterech dostępnych było średnio wykorzystywanych podczas pojedynczego mnożenia. Na podstawie wykonanych pomiarów możemy stwierdzić, że algorytm wykorzystuje od 80% do 90% dostępnej mocy obliczeniowej. Jest to bardzo dobry wynik, jak na algorytm arytmetyczny. Można więc powiedzieć, że cel polegający na zaprojektowaniu równoległego algorytmu mnożenia wielomianów został osiągnięty.

Jeśli chodzi o wyniki teoretyczne płynące z pracy, to kluczowa pod tym względem jest analiza przeprowadzona w rozdziale 5 i będący kwintesencją tych rozważań Wniosek 2. Jeżeli przyjąć, że stopień wielomianu wynosi n, a precyzja współczynników wynosi k, to asymptotyczna złożoność proponowanego algorytmu jest rzędu

$$O(kn \log n + k^2 n).$$

Ze względu na dwa istotne składniki funkcji asymptotycznej nie można w sposób jednoznaczny określić, czy nowe rozwiązanie jest lepsze, czy gorsze od metody wykorzystującej jedynie FFT. Jeżeli bowiem do mnożenia zarówno wielomianu, jak i jego współczynników wykorzystywana jest szybka transformata Fouriera, to złożoność jest rzędu

$$O((n \log n)(k \log k)).$$

Widać więc, że jeśli $k = O(n)$, to proponowany algorytm wypada słabiej niż metoda bazująca tylko na FFT. Jeżeli jednak $k = O(\log n)$ to złożoność no-

wego algorytmu jest mniejsza. Stosunek złożoności obliczeniowych jest wtedy na poziomie $O(\log \log n)$ na korzyść metody zaprezentowanej w pracy. Przeprowadzone rozumowanie pozwala nam wyciągnąć wniosek, że algorytm oparty o CRT i FFT powinien być stosowany w przypadku, gdy liczba współczynników wielomianu znacznie przekracza ich precyzję. Z taką sytuacją mamy często do czynienia w przypadku obliczeń na długich wielomianach lub szeregach potęgowych z redukcją modularną współczynników.

Wyniki testów numerycznych zebrane w części 6.4 pokazują, że praktyczne znaczenie proponowanej metody jest bardzo duże. W tej części celowo algorytm porównywany był z implementacją wykorzystującą klasyczny algorytm mnożenia współczynników w dużych ciałach $\mathbb{F}_p$. Wynika to z faktu, że dla liczb p mających 500 lub 1000 bitów zastosowanie mnożenia opartego o transformatę Fouriera jest kompletnie nieopłacalne. Rezultat pomiarów był bardzo dużym zaskoczeniem. Okazało się bowiem (Tabele 6.1 i 6.2), że proponowany algorytm jest kilkukrotnie szybszy nawet w przypadku gdy nie jest wykonywany równolegle. Jego porównanie z klasycznym algorytmem mnożenia wielomianów o złożoności $O(n^2)$ (Tabele 6.3 i 6.4) wypada również nadzwyczaj korzystnie. Nie ma więc wątpliwości, że przedstawiony algorytm spisuje się nadzwyczaj dobrze w zastosowaniach praktycznych.

Jeśli chodzi o wkład pracy do ogólnego rozwoju wiedzy, to niewątpliwie najważniejszym elementem są rozważania przeprowadzone w rozdziale 5 i płynący z nich Wniosek 2. O ile idea samego algorytmu była klarowna od samego początku, to wniosek wyciągnięty na końcu rozdziału 5 był naprawdę dużym zaskoczeniem. Przewidywano co prawda, że algorytm będzie szybszy od metody wykorzystującej FFT do mnożenia wielomianu i jego współczynników, ale tylko dla współczynników o ograniczonej wielkości. Okazało się jednak, że jeśli precyzja współczynników zależy logarytmicznie od stopnia wielomianu, to uzyskujemy algorytm asymptotycznie szybszy od metody opartej jedynie na szybkiej transformacie Fouriera.

Prace badawcze w swoim pierwotnym założeniu miały obejmować również zagadnienie efektywnego generowania parametrów dla algorytmu RSA. Niestety wynik uzyskany w artykule [9] może być interesujący jedynie z teoretycznego punktu widzenia i nie wnosi specjalnie nic wartościowego do sfery praktycznej. Dlatego też w dalszej części projektu skoncentrowano się na opracowaniu szybkiego algorytmu mnożenia wielomianów i szeregów potęgowych. Okazało się to bardzo dobrym posunięciem, gdyż prace badawcze zaowocowały powstaniem nowej, bardzo efektywnej metody mnożenia wielomianów o dużym znaczeniu praktycznym, szczególnie w kontekście zastosowań kryptograficznych.

Jednym z nowych kierunków badań, które można podjąć na podstawie osiągniętych wyników jest próba adaptacji algorytmu mnożenia na potrzeby dowodzenia pierwszości metodą sum Jacobiego przedstawionych w pracy autora [4]. Test ten wykorzystuje operacje arytmetyczne w pierścieniu $\mathbb{Z}[\zeta_n]$, gdzie

$\zeta_n = e^{2\pi i/n}$. Zwróćmy uwagę, że pierścień $\mathbb{Z}[\zeta_n]$ jest izomorficzny z pierścieniem $\mathbb{Z}[X]/(f_n)$, gdzie f_n jest n-tym wielomianem cyklotomicznym. W związku z tym można przeanalizować opłacalność wykorzystania zaproponowanego algorytmu w metodzie dowodzenia pierwszości opartej na sumach Jacobiego.

Rozdział 8

Elementy wkładu oryginalnego

Oryginalne wyniki teoretyczne zostały ujęte w rozdziałach 5 i 6. Lematy i twierdzenia zawarte w tych rozdziałach pracy zostały pomyślane w ten sposób, aby ich dowody były konstruktywne i dawały możliwość bezpośredniego przełożenia teorii na efektywną implementację. Kod źródłowy opracowanego algorytmu został umieszczony w załączniku B.

Przeprowadzona analiza wykazała, że mnożenie oparte na metodzie Montgomery'ego [42] jest wydajniejsze niż klasyczna redukcja modularna nawet w przypadku liczb pojedynczej precyzji (mieszczących się w pojedynczym rejestrze procesora). W związku z tym konieczne było rozszerzenie Lematu 4 pochodzącego od Montgomery'ego do Twierdzenia 7. Twierdzenie to stanowi teoretyczną podstawę poprawnego działania operacji arytmetycznych w ciałach $\mathbb{F}_{p_i}$.

Zdefiniowane zostały parametry, dla których możliwe jest traktowanie mnożenia wielomianów całkowitoliczbowych jak mnożenia wielomianów nad $\mathbb{Z}/M\mathbb{Z}$. Warunki niezbędne do wykonania takiej operacji zostały ujęte w Lemacie 5. Lemat ten jest bardzo ważnym elementem rozdziału 5, gdyż to właśnie on stanowi podstawę dla twierdzeń uzasadniających możliwość wykorzystania twierdzenia chińskiego o resztach i szybkiej transformaty Fouriera podczas mnożenia wielomianów o współczynnikach całkowitych.

Udowodnione zostało Twierdzenie 8, które określa warunki niezbędne do poprawnego wymnożenia wielomianów z pierścienia $\mathbb{Z}[X]$ w pierścieniu $\mathbb{F}_p[X]$. Następnie w Twierdzeniu 9 sprecyzowano złożoność tej metody mnożenia. Twierdzenie 9 nie stanowi wkładu oryginalnego. Jest to powszechnie znany fakt, który cechuje każdy algorytm mnożenia oparty na FFT. Zostało ono jednak podane dla kompletności wywodu i stanowi swoisty punkt odniesienia dla złożoności nowego algorytmu.

Udowodnione zostało Twierdzenie 10 określające w jaki sposób poprawnie wykonać mnożenie wielomianów całkowitoliczbowych w pierścieniu $\mathbb{F}_{p_1}[X] \times \cdots \times \mathbb{F}_{p_k}[X]$. Twierdzenie to stanowi teoretyczną podstawę proponowanego algorytmu. W Twierdzeniu 11 wyznaczona została złożoność opracowanej metody mnożenia, a Wniosek 1 porównuje szybkość działania nowego algorytmu z tradycyjną implementacją opartą na transformacie Fouriera w pierścieniu $\mathbb{F}_p[X]$. Stałe pojawiające się podczas porównania obu algorytmów są zależne od architektury procesora i jakości implementacji. Dlatego też ich oszacowanie jest bardzo trudne. W związku z tym przeprowadzono eksperymenty numeryczne, które pokazały ogromną przewagę nowej metody nad podejściem klasycznym. Na szczególną uwagę zasługuje fakt, że nowy algorytm działa szybciej nawet w przypadku gdy jest uruchomiony na pojedynczym procesorze (Tabela 6.1 i Tabela 6.2). Dokonano też porównania nowego algorytmu z metodą wykorzystującą FFT zarówno do mnożenia wielomianów, jak i do mnożenia ich współczynników. Okazało się, że jeśli współczynniki wielomianu nie są zbyt duże w relacji do jego stopnia, to nowy algorytm ma bardzo dobrą złożoność. Wynik tej obserwacji zawarto w Wniosku 2.

W części 6.2 zawarty został kompletny opis transformaty Fouriera, transformaty do niej odwrotnej oraz dowód poprawności działania obu algorytmów. Zarówno postać twierdzeń, jak i ich dowody są oryginalne i konstruktywne. Zostały one opracowane w taki sposób, aby miały bezpośrednie przełożenie na efektywną implementację zawartą w załączniku B. W literaturze można znaleźć wiele dowodów poprawności działania algorytmu mnożenia opartego na transformacie Fouriera. W tym przypadku dostępne materiały okazały się jednak mało przydatne ze względu na konieczność zastosowania zrównoleglonej wersji twierdzenia chińskiego o resztach.

Część 6.4 opisuje uzyskane wyniki czasowe dla poszczególnych implementacji algorytmu mnożenia wielomianów o współczynnikach mających odpowiednio 256 i 512 bitów. Na szczególną uwagę zasługuje tutaj fakt, że wraz ze wzrostem stopnia wielomianów i wielkości współczynników rośnie też przewaga algorytmu opartego o CRT nad algorytmem wykorzystującym FFT w pierścieniu $\mathbb{F}_p[X]$. Zależność ta została oznaczona w Tabelach 6.1 i 6.2 jako wielkość T_1/T_2. W części tej zostało również uwzględnione porównanie czasu działania nowego algorytmu z klasycznym algorytmem mnożenia o złożoności kwadratowej. W tym przypadku stosunek czasu działania obu algorytmów jest dramatycznie duży i dochodzi do 11 000 dla wielomianów stopnia 131 071 o współczynnikach rzędu 2^{512}.

W załączniku B znajduje się implementacja algorytmu szybkiego mnożenia wielomianów i szeregów potęgowych o współczynnikach całkowitych nieujemnych. Została ona przygotowana w oparciu o osiągnięte wyniki teoretyczne. Kod został przygotowany zgodnie ze standardem ANSI C, a mechanizmy zrównoleglające obliczenia wykorzystują interfejs OpenMP. Daje to możliwość użycia

kodu na różnych platformach obliczeniowych. Warunkiem poprawnej kompilacji jest posiadanie kompilatora zgodnego z ANSI C. Natomiast mechanizmy zrównoleglające dostępne są w środowiskach wspierających standard OpenMP.

Jeśli chodzi o praktyczne wykorzystanie, to przedstawiony algorytm szybkiego mnożenia wielomianów już znalazł swoje zastosowanie. Został użyty przez firmę Enigma Systemy Ochrony Informacji Sp. z o.o. do przyspieszenia algorytmu generowania krzywych eliptycznych silnych kryptograficznie. Czterordzeniowa implementacja skróciła czas fazy obliczeń wstępnych z około 648 godzin (27 dni) do około 41 godzin. Daje to 16-krotne przyspieszenie i wydatnie skraca czas obliczeń wstępnych.

Bibliografia

[1] E. Bach, "Iterative root approximation in p-adic numerical analysis", *Journal of Complexity*, vol. 25, str. 511–529, 2009.

[2] M. Brown, D. Hankerson, J. López, A. Menezes, "Software Implementation of the NIST Elliptic Curves Over Prime Fields", *Proceedings of the 2001 Conference on Topics in Cryptology: The Cryptographer's Track at RSA*, 2001.

[3] D. Charles, K. Lauter, "Computing modular polynomials", *Journal of Computational Mathematics*, str. 195–204, 2005.

[4] A. Chmielowiec, "Primality proving with Gauss and Jacobi Sums", *Journal of Telecommunications and Information Technology*, vol. 4, str. 69–75, 2004.

[5] A. Chmielowiec, "Estymacja kosztów łamania systemu kryptograficznego", *XI Krajowa Konferencja Kryptografii i Ochrony Informacji*, str. 169–176, Warszawa, 2007.

[6] A. Chmielowiec, "Kryptografia na procesorach wielordzeniowych", *XII Krajowa Konferencja Kryptografii i Ochrony Informacji*, str. 101–104, Warszawa, 2008.

[7] A. Chmielowiec, "Asymptotic arithmetic for polynomials and formal power series", XI Międzynarodowe Warsztaty dla Młodych Matematyków - Teoria Liczb, Kraków, 2008.

[8] A. Chmielowiec, "Ataki na RSA", *XIII Krajowa Konferencja Kryptografii i Ochrony Informacji*, str. 91–97, Warszawa, 2009.

[9] A. Chmielowiec, "Fixed points of the RSA encryption algorithm", *Theoretical Computer Science*, vol. 411-1, str. 288–292, 2010.

[10] P. Cohen, "On the coefficients of the transformation polynomials for the elliptic modular function", *Math. Proc. Camb. Phil. Soc.*, vol. 95, str. 389–402, 1984.

[11] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, "Introduction to Algorithms", MIT Press, 2003.

[12] R. Crandall, B. Fagin, "Discrete weighted transforms and large integer arithmetic", *Mathematics of Computation*, vol. 62, str. 305–324, 1994.

[13] R. Crandall, C. Pomerance, "Prime Numbers – a computational perspective", Springer-Verlag, 2001.

[14] W. Diffie, M. Hellman, "New directions in cryptography", *IEEE Transactions on Information Theory*, vol. 22, str. 644–654, 1976.

[15] T. ElGamal, "A public key cryptosystem and a signature scheme based on discrete logarithms", *IEEE Trans. Info. Theory*, vol. 31, str 469–472, 1985.

[16] A. Enge, "Computing modular polynomials in quasi-linear time", *Math. Comp.*, vol. 78, str. 1809–1824, 2009.

[17] M. Fouquet, P. Gaudry, R. Harley, "On Satoh's algorithm and its implementation", *J. Ramanujan Math. Soc.*, vol. 15, str. 281–318, 2000.

[18] G. Frey, H. Rück, "A remark concerning m-divisibility and the discrete logarithm problem in the divisor class group of curves", *Math. Comp.*, vol. 62, str. 865–874, 1994.

[19] L. Garcia, "Can Schönhage multiplication speed up the RSA encryption or decryption?", *University of Technology, Darmstadt*, 2005.

[20] H. L. Garner, "The Residue Number System", *IRE Trans. Electronic Computers*, vol. EL-8, No. 6, str. 140–147, 1959.

[21] S. Gorlatch, "Programming with divide-and-conquer skeletons: A case study of FFT", *Journal of Supercomputing*, vol. 12, str. 85–97, 1998.

[22] A. Grama, A. Gupta, G. Karypis, V. Kumar, "Introduction to Parallel Computing", Addison Wesley, 2003.

[23] N. Gura, A. Patel, A. Wander, H. Eberle, S. C. Shantz, "Comparing Elliptic Curve Cryptography and RSA on 8-bit CPUs", *Lecture Notes in Computer Science, CHES 2004*, vol. 3156, str. 119–132, Springer-Verlag, 2004.

[24] G. Hanrot, M. Querica, P. Zimmermann, "Speeding up the division and square roots of power series", Research Report 3973, INRIA, retrived from http://hal.inria.fr/docs/00/07/26/75/PS/RR-3973.ps, 2000.

[25] R. Harley, J.F. Mestre, P. Gaudry, "Counting points with the Arithmetic-Geometric Mean", *Advances in Cryptology – Eurocrypt 2001*, Springer-Verlag, 2001.

[26] D. Harvey, "A cache-friendly truncated FFT", *Theoretical Computer Science*, vol. 410, str. 2649–2658, 2009.

[27] J. van der Hoeven, "The truncated Fourier transform and applications", in: ISSAC 2004, ACM, str. 290–296, 2004.

[28] J. van der Hoeven, "Notes on the truncated Fourier transform", unpublished, retrieved from http://www.math.u-psud.fr/~vdhoeven/, 2005.

[29] D. E. Knuth, "Art of Computer Programming", Addison-Wesley Professiona, 1998.

[30] N. Koblitz, "Elliptic curve cryptosystems", *Mathematics of Computation*, vol. 48 (177), str. 203–209, 1987.

[31] M. Kordos, "Wykłady z historii matematyki", Wydawnictwa Szkolne i Pedagogiczne, Warszawa, 1994.

[32] A.K. Lenstra, H.W. Lenstra, "The Development of the Number Field Sieve", *LNM*, vol. 1554, 1993.

[33] H.W. Lenstra, "Factoring integers with elliptic curves", *Ann. Math.*, vol. 126, str. 649–673, 1987.

[34] U. M. Maurer, "Towards the equivalence of breaking the Diffie-Hellman protocol and computing discrete logarithms", *Lecture Notes in Computer Science, CRYPTO '94*, vol. 839, str. 271–281, Springer-Verlag, 1994.

[35] U. M. Maurer, S. Wolf, "On the difficulty of breaking the DH protocol", *Technical Report*, vol. 24, Department of Computer Science, ETH Zurich, 1996.

[36] U. M. Maurer, S. Wolf, "Diffie-Hellman Oracles", *Lecture Notes in Computer Science, CRYPTO '96*, vol. 1109, str. 268–282, Springer-Verlag, 1996.

[37] U. M. Maurer, S. Wolf, "The relationship between breaking the Diffie-Hellman protocol and computing discrete logarithms", *SIAM Journal on Computing*, vol. 28, str. 1689–1721, 1999.

[38] U. M. Maurer, S. Wolf, "The Diffie-Hellman protocol", *Designs, Codes, and Cryptography*, vol. 19, str. 147–171, 2000.

[39] A. Menezes, T. Okamoto, S. Vanstone, "Reducing elliptic curve logarithms to a finite field", *IEEE Trans. Info. Theory*, vol. 39, str. 1639–1646, 1993.

[40] A. J. Menezes, P. C. Van Örschot, S. A. Vanstone, "Handbook of Applied Cryptography", CRC Press, 1997.

[41] V. Miller, "Use of elliptic curves in cryptography", *CRYPTO 85*, str. 417–426, 1985.

[42] P. Montgomery, "Modular Multiplication Without Trial Division", *Mathematics of Computation*, vol. 44, str. 519–521, 1985.

[43] V. Müller, "Ein Algorithmus zur Bestimmung der Punktzahlen elliptischer Kurven über endlichen Körpern der Charakteristik grösser drei", *Ph.D. Thesis, Universität des Saarlandes*, 1995.

[44] H. J. Nussbaumer, "Fast polynomial transform algorithms for digital convolution", *IEEE Trans. Acoust. Speech Signal Process.*, vol. 28 (2), str. 205–215, 1980.

[45] A. Muzereau, N. P. Smart, F. Vrecauteren, "The equivalence between the DHP and DLP for elliptic curves used in practical applications", *LMS J. Comput. Math.*, vol. 7(2004), str. 50–72, 2004.

[46] J.M. Pollard, "Monte Carlo methods for index computation (mod p)", *Math. Comp.*, vol. 32, str. 918–924, 1978.

[47] R. Rivest, A. Shamir, L. Adleman, "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems", *Communications of the ACM*, vol. 21 (2), str. 120–126, 1978.

[48] T. Satoh, K. Araki, "Fermat quotients and the polynomial time discrete log algorithm for anomalous elliptic curves", *Comm. Math. Univ. Sancti Pauli*, vol. 47, str. 81–92, 1998.

[49] T. Satoh, "The canonical lift of an ordinary elliptic curve over a finite field and its point counting". *J. Ramanujan Math. Soc.*, vol. 15, str. 247–270, 2000.

[50] R. Schoof, "Elliptic curves over finite fields and the computation of square roots mod p", *Math. Comp.*, vol. 44, str. 483–494, 1985.

[51] R. Schoof, "Counting points on elliptic curves over finite fields", *J. Théorie des nombres de Bordeaux*, vol. 7, str. 219–254, 1995.

[52] A. Schönhage, V. Strassen, "Schnelle Multiplikation grosser Zahlen", *Computing*, vol. 7, str. 281–292, 1971.

[53] A. Schönchage, "Asymptotically fast algorithms for the numerical multiplication and division of polynomials with complex coefficients", *Lecture Notes in Computer Science*, vol. 144, str. 3–15, 1982.

[54] I. Semaev, "Evaluation of discrete logarithms on some elliptic curves", *Math. Comp.*, vol. 67, str. 353–356, 1998.

[55] J. H. Silverman, "The arithmetic of elliptic curves", *Graduate Texts in Mathematics*, vol 106, Springer-Verlag, New York, 1986.

[56] J. H. Silverman, J. Tate, "Rational points on elliptic curves", *Undergraduate Texts in Mathematics*, Springer-Verlag, New York, 1992.

[57] B. Skjernaa, "Satoh's algorithm in characteristic 2", *Preprint*, 2000.

[58] N. Smart, "The discrete logarithm problem on elliptic curves of trace one", *J. Crypto.*, vol. 12, str. 193–196, 1999.

[59] D. Takahashi, Y. Kanada, "High-performance radix-2, 3 and 5 parallel 1-D complex FFT algorithms for distributed-memory parallel computers", *Journal of Supercomputing*, vol. 15, str. 207–228, 2000.

[60] F. Vercauteren, B. Preneel, J. Vandewalle, "A Memory Efficient Version of Satoh's Algorithm", *Advances in Cryptology – Eurocrypt 2001, LNCS 2045*, str. 1–13, Springer-Verlag, 2001.

[61] ANSI X9.62-1998, "Public Key Cryptography for the Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA)", 2005.

[62] NIST, "Digital Signature Standard", *FIPS Publication 186-3*, 2010.

[63] SECG, "SEC1: Elliptic Curve Cryptography", Dostępne na http://www.secg.org/, 2009.

[64] SECG, "SEC2: Recommended Elliptic Curve Domain Parameters", Dostępne na http://www.secg.org/, 2000.

Dodatek A

Biblioteka do szybkiego mnożenia wielomianów

A.1 Kompilacja biblioteki

W załączniku B zamieszczony został kod źródłowy biblioteki do szybkiego mnożenia wielomianów o współczynnikach, które są nieujemnymi liczbami całkowitymi. Implementacja ta została przygotowana w oparciu o standard języka ANSI C oraz jego rozszerzenie OpenMP. W związku z tym biblioteka może być uruchomiona na dowolnym procesorze, którego kompilator wspiera oba standardy.

Kod biblioteki składa się z sześciu plików (dwóch nagłówkowych i czterech źródłowych), które zawierają niezbędne deklaracje oraz implementację algorytmu mnożenia.

1. `uintpoly.h` – główny plik nagłówkowy zawierający interfejs wysokiego poziomu do funkcji działających na wielomianie.
2. `uintpoly_locl.h` – pomocniczy plik nagłówkowy zawierający deklaracje typów i funkcji wykorzystywanych przez wewnętrzne procedury biblioteki.
3. `uintpoly.c` – plik zawiera implementację podstawowych funkcji operujących na wielomianach (alokowanie i zwalnianie pamięci, porównywanie, wyświetlanie i mnożenie metodą *szkolną*).
4. `arth.c` – plik zawiera implementację funkcji arytmetycznych (dodawanie, mnożenie, redukcja modularna) oraz funkcje inicjujące stałe dla algorytmu Garnera (CRT).
5. `dft_fp32.c` – plik zawiera implementację funkcji generującej dane niezbędne do wykonywania transformaty Fouriera (DFT).

6. `dft_fp32_crt.c` – plik zawiera implementację szybkiego, zrównoleglonego algorytmu mnożenia wielomianów. Przygotowana implementacja wykorzystuje dyrektywy OpenMP, a obliczenia wykonuje na maksymalnej dostępnej liczbie rdzeni.

Aby skompilować bibliotekę pod systemem Linux można wykorzystać narzędzie make. W takim przypadku tworzymy plik `Makefile` o następującej treści, a następnie kompilujemy bibliotekę poleceniem `make`.

```
TARGET=libfastpolymul
CC=gcc
AR=ar
CFLAGS=-O2 -Wall -ansi -fPIC -fopenmp
LIBS=
OBJS=uintpoly.o arth.o dft_fp32.o dft_fp32_crt.o

.c.o:
	$(CC) $(CFLAGS) -c $< -o $@

all: static dynamic

static: $(OBJS)
	$(AR) crv $(TARGET).a $^

dynamic: $(OBJS)
	$(CC) $(CFLAGS) --shared -o $(TARGET).so $^ $(LIBS)

clean: FORCE
	rm -f *.o

distclean: clean
	rm -f $(TARGET).a $(TARGET).so

FORCE:
```

Należy pamiętać, że plik `Makefile` musi znajdować się w tym samym katalogu, co pliki źródłowe biblioteki szybkiego mnożenia wielomianów. Polecenie `make` powinno wygenerować dwa pliki biblioteki:

1. `libfastpolymul.a` – biblioteka statyczna,
2. `libfastpolymul.so` – biblioteka dynamiczna.

Aby skompilować bibliotekę w środowisku Microsoft Visual Studio tworzymy pusty projekt biblioteczny i dodajemy wszystkie pliki nagłówkowe i źródłowe. Następnie konieczne jest ustawienie opcji pozwalającej na użycie OpenMP. Dokonujemy tego poprzez wybranie opcji *OpenMP Support* (opcja wspierana począwszy od MSVC 2005) w ustawieniach projektu:

Configuration Properties $\rightarrow$ *C/C++* $\rightarrow$ *Language.*

Należy zauważyć, że środowisko Microsoft nie zapewnia pełnej zgodności ze standardem ANSI. W związku z tym w wersjach starszych niż MSVC 2010 mogą pojawić się problemy z brakiem nagłówka `stdint.h` definiującego wykorzystywane przez bibliotekę typy. Problem ten można rozwiązać poprzez samodzielną definicję typów `uint32_t` (arytmetyczny typ 32-bitowy) i `uint64_t` (arytmetyczny typ 64-bitowy).

A.2 Przykład użycia biblioteki

Poniższy kod pokazuje w jaki sposób należy korzystać z funkcji bibliotecznych. Najważniejszą operacją przed wykonaniem szybkiego mnożenia jest zainicjowanie wewnętrznych struktur biblioteki. Programista powinien w procesie inicjacji określić maksymalny stopień wielomianów, na których będą wykonywane operacje (chodzi tu o maksymalny stopień zarówno czynników, jak i całego ilorazu). Maksymalnym stopniem akceptowany przez tą wersję biblioteki jest $2^{22} - 1$. Wynika on z postaci wykorzystanych liczb pierwszych.

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

int
main(int argc, char *argv[])
{
    /* Deklaracja wielomianów. */
    UINT_Poly a, b, c, d;

    /*
     * Inicjacja struktur niezbędnych do szybkiego przeprowadzania
     * DFT i CRT.
     *
     * W tym przypadku zakładamy, że nie będziemy operowali na
     * wielomianach stopnia większego niż 127. Dotyczy to również
     * iloczynów.
     */
    UINT_Poly_dft_crt_init(127);

    /* Inicjacja wielomianów. */
    UINT_Poly_init(&a);
    UINT_Poly_init(&b);
    UINT_Poly_init(&c);
    UINT_Poly_init(&d);

    /*
     * Przypisanie na a losowego wielomianu stopnia 2, którego
     * współczynniki są 64-bitowe (2*32).
     */
    UINT_Poly_random(&a, 2, 2);
    /*
     * Przypisanie na b losowego wielomianu stopnia 4, którego
     * współczynniki są 96-bitowe (3*32).
     */
    UINT_Poly_random(&b, 3, 4);

    /*
     * Wykonanie mnożenia metodą klasyczną i przypisanie iloczynu
     * na wielomian c.
     */
    UINT_Poly_mul(&c, &a, &b);
    /*
     * Wykonanie mnożenia metodą FFT + CRT i przypisanie iloczynu
     * na wielomian d.
     */
    UINT_Poly_dft_crt_mul(&d, &a, &b);

    /*
     * Wydruk czynników oraz ilorazów wyznaczonych za pomocą obu
     * algorytmów (wielomiany c i d są identyczne z dokładnością
     * do wiodących zer we współczynnikach.
     */
    printf("a = \n");
    UINT_Poly_print(&a);
```

```
    printf("b = \n");
    UINT_Poly_print(&b);
    printf("c = \n");
    UINT_Poly_print(&c);
    printf("d = \n");
    UINT_Poly_print(&d);

    /* Zwolnienie pamięci zarezerwowanej przez wielomiany. */
    UINT_Poly_clear(&a);
    UINT_Poly_clear(&b);
    UINT_Poly_clear(&c);
    UINT_Poly_clear(&d);

    /* Zwolnienie struktur przeznaczonych do obliczania DFT i CRT. */
    UINT_Poly_dft_crt_finish();

    return 0;
}
```

Dodatek B

Kod źródłowy biblioteki szybkiego mnożenia wielomianów

B.1 Plik uintpoly.h - deklaracja funkcji i struktur niezbędnych do wykonania szybkiego mnożenia wielomianów

```
#ifndef UINT_POLY_H
#define UINT_POLY_H

#ifdef __cplusplus
extern "C"
{
#endif

#include <stdint.h>

/* Struktura reprezentująca wielomian. */
typedef struct UINT_Poly
{
    /* Tablica wskaźników do współczynników wielomianu. */
    uint32_t **v;
    /* Liczba cyfr pojedynczego współczynnika wielomianu. */
    int digits;
    /* Stopień wielomianu. */
    int deg;
} UINT_Poly;

/* Inicjacja struktury wielomianu. */
extern int UINT_Poly_init(UINT_Poly * a);
/* Inicjacja pamięci na wielomian o zadanym stopniu i wielkości współczynników. */
extern int UINT_Poly_init_mem(UINT_Poly * a, int digits, int deg);
/* Inicjacja współczynnika wielomianu dla termu o zadanym stopniu. */
extern int UINT_Poly_init_coef(UINT_Poly * a, const uint32_t * coef, int digits,
    int deg);
/* Wyczyszczenie struktury wielomianu. */
extern int UINT_Poly_clear(UINT_Poly * a);
/* Wygenerowanie wielomianu o zadanych parametrach. */
```

```
extern int UINT_Poly_random(UINT_Poly * a, int digits, int deg);
/* Porównanie wielomianów. */
extern int UINT_Poly_eq(const UINT_Poly * a, const UINT_Poly * b);
/* Wymnożenie wielomianów metodą klasyczną. */
extern int UINT_Poly_mul(UINT_Poly * c, const UINT_Poly * a,
    const UINT_Poly * b);
/* Wydrukowanie wielomianu na ekran (szesnastkowo). */
extern int UINT_Poly_print(const UINT_Poly * a);
/* Inicjacja stałych do szybkiego mnożenia wielomianów. */
extern int UINT_Poly_dft_crt_init(int deg_max);
/* Zwolnienie stałych do szybkiego mnożenia wielomianów. */
extern int UINT_Poly_dft_crt_finish();
/* Szybkie mnożenie wielomianów. */
extern int UINT_Poly_dft_crt_mul(UINT_Poly * c, const UINT_Poly * a,
    const UINT_Poly * b);

#ifdef __cplusplus
};
#endif

#endif /* UINT_POLY_H */
```

B.2 Plik uintpoly_locl.h - lokalne deklaracje dla algorytmu

```
#ifndef UINT_POLY_LOCL_H
#define UINT_POLY_LOCL_H

#ifdef __cplusplus
extern "C"
{
#endif

#include <stdlib.h>
#include <string.h>
#include "uintpoly.h"

/* Struktura przechowująca dane do DFT nad ciałem skończonym. */
typedef struct ARTH_Fp32_dft
{
    /* Liczba pierwsza. */
    uint32_t p;
    /* Współczynnik Montgomeryego q = p^(-1) mod(2^32). */
    uint32_t q;
    /* Współczynnik s = 2^64 mod(p). */
    uint32_t s;
    /* Odwrotność liczby 2, i2 = 2^(-1) mod(p). */
    uint32_t i2;
    /* Tablica pierwiastków z jedności. */
    uint32_t *r;
    /* Odwrotności pierwiastków umieszczonych w tablicy r. */
    uint32_t *ir;
    /* Liczba pierwiastków w tablicach: r, ir. */
    int roots;
    /* Maksymalny rozmiar wektora, który może być poddany transformacie. */
    int n;
} ARTH_Fp32_dft;

/* Inicjacja struktury DFT. */
extern int ARTH_Fp32_dft_init(ARTH_Fp32_dft * dft, int prec, uint32_t p,
    uint32_t r);
/* Zwolnienie struktury DFT. */
extern int ARTH_Fp32_dft_free(ARTH_Fp32_dft * dft);
```

```
/* Struktura przechowująca dane dla algorytmu Ganera. */
typedef struct ARTH_Fp32_crt
{
    /* Tablica liczb pierwszych. */
    uint32_t *p;
    /* Wektor współczynników dla algorytmu Garnera.
     *
     *   c[i] = (p[0]*p[1]*...*p[i-1])^(-1) mod p[i]
     *   pp[i*n] = p[0]*p[1]*...*p[i-1]
     */
    uint32_t *c;
    uint32_t *pp;
    /* Liczba liczb pierwszych umieszczonych w tablicy p. */
    int n;
} ARTH_Fp32_crt;

/* Porównanie dwóch liczb naturalnych. */
extern int ARTH_cmp(const uint32_t * op1, int len1, const uint32_t * op2,
    int len2);
/* Porównanie liczby naturalnej i pojedynczej cyfry. */
extern int ARTH_cmp_digit(const uint32_t * op1, uint32_t op2, int len);
/* Dodanie dwóch liczb naturalnych. */
extern int ARTH_add(uint32_t * dst, const uint32_t * src, int len);
/* Dodanie liczby naturalnej i pojedynczej cyfry. */
extern int ARTH_add_digit(uint32_t * dst, uint32_t src, int len);
/* Pomnożenie liczby naturalnej przez pojedynczą cyfrę. */
extern uint32_t ARTH_mul_digit(uint32_t * a, int na, uint32_t b);
/* Pomnożenie liczby naturalnej przez pojedynczą cyfrę i dodanie do liczby naturalnej. */
extern uint32_t ARTH_mul_digit_add(uint32_t * a, int na, const uint32_t * b,
    int nb, uint32_t d);
/* Pomnożenie dwóch liczb naturalnych. */
extern void ARTH_mul(uint32_t * d, const uint32_t * a, int na,
    const uint32_t * b, int nb);
/* Redukcja liczby naturalnej modulo pojedyncza cyfra. */
extern uint32_t ARTH_mod_digit(const uint32_t * a, int na, uint32_t b);
/* Inicjacja struktury CRT dla algorytmu Garnera. */
extern int ARTH_Fp32_crt_init(ARTH_Fp32_crt * crt, const uint32_t * p, int n);
/* Zwolnienie struktury CRT. */
extern void ARTH_Fp32_crt_free(ARTH_Fp32_crt * crt);


#define ARTH_ADC(arg_r, arg_a, arg_b, arg_c) \
    do { \
        uint64_t __arth_tmp = (uint64_t)(arg_a) + (uint64_t)(arg_b) + (uint64_t)(arg_c); \
        (arg_c) = (uint32_t)(__arth_tmp >> 32); \
        (arg_r) = (uint32_t)(__arth_tmp); \
    } while (0);

#define ARTH_SBB(arg_r, arg_a, arg_b, arg_c) \
    do { \
        uint64_t __arth_tmp = (uint64_t)(arg_a) - (uint64_t)(arg_b) - (uint64_t)(arg_c); \
        (arg_c) = (uint32_t)(__arth_tmp >> 32) & 1; \
        (arg_r) = (uint32_t)(__arth_tmp); \
    } while (0);

#define ARTH_MUL(arg_r, arg_a, arg_b, arg_c) \
    do { \
        uint64_t __arth_tmp = (uint64_t)(arg_a) * (uint64_t)(arg_b) + (uint64_t)(arg_c); \
        (arg_c) = (uint32_t)(__arth_tmp >> 32); \
        (arg_r) = (uint32_t)(__arth_tmp); \
    } while (0);

#define ARTH_MUL2(arg_r, arg_a, arg_b, arg_c, arg_d) \
    do { \
        uint64_t __arth_tmp = (uint64_t)(arg_a) * (uint64_t)(arg_b) + (uint64_t)(arg_c) + (uint64_t)(arg_d); \
        (arg_c) = (uint32_t)(__arth_tmp >> 32); \
        (arg_r) = (uint32_t)(__arth_tmp); \
    } while (0);
```

```
#define ARTH_MUL_ADD(arg_r, arg_a, arg_b, arg_c) \
    do { \
        uint64_t __arth_tmp = (uint64_t)(arg_a) * (uint64_t)(arg_b) + (uint64_t)(arg_c) + (uint64_t)(arg_r); \
        (arg_c) = (uint32_t)(__arth_tmp >> 32); \
        (arg_r) = (uint32_t)(__arth_tmp); \
    } while (0);

#define ARTH_DIV(arg_q, arg_r, arg_h, arg_l, arg_d) \
    do { \
        uint64_t __arth_tmp = ((uint64_t)(arg_h) << 32) | (uint64_t)(arg_l); \
        (arg_q) = (uint32_t)(__arth_tmp / (arg_d)); \
        (arg_r) = (uint32_t)(__arth_tmp - (uint64_t)(arg_q)*(arg_d)); \
    } while (0);

#ifdef __cplusplus
};
#endif

#endif /* UINT_POLY_LOCL_H */
```

B.3 Plik uintpoly.c - podstawowe operacje na wielomianie

```
#include "uintpoly_locl.h"

#include <stdio.h>


int
UINT_Poly_init(UINT_Poly * a)
{
    memset(a, 0, sizeof(UINT_Poly));
    a->deg = -1;
    return 0;
}


int
UINT_Poly_init_mem(UINT_Poly * a, int digits, int deg)
{
    int i;

    a->digits = digits;
    a->deg = deg;

    if ((a->v = calloc(a->deg + 1, sizeof(uint32_t *))) == 0) {
        return -1;
    }
    memset(a->v, 0, (a->deg + 1) * sizeof(uint32_t));

    for (i = 0; i <= deg; i++) {
        if ((a->v[i] = calloc(a->digits, sizeof(uint32_t))) == 0) {
            UINT_Poly_clear(a);
            return -1;
        }
        memset(a->v[i], 0, a->digits * sizeof(uint32_t));
    }

    return 0;
}


int
```

```
UINT_Poly_init_coef(UINT_Poly * a, const uint32_t * coef, int digits, int deg)
{
    if (a->deg < deg) {
        return -1;
    }

    if (a->digits < digits) {
        return -1;
    }

    memset(a->v[deg], 0, a->digits * sizeof(uint32_t));
    memcpy(a->v[deg], coef, digits * sizeof(uint32_t));

    return 0;
}


int
UINT_Poly_clear(UINT_Poly * a)
{
    int i;

    if (a->v) {
        for (i = 0; i <= a->deg; i++) {
            if (a->v[i]) {
                free(a->v[i]);
            }
        }
        free(a->v);
    }

    memset(a, 0, sizeof(UINT_Poly));
    a->deg = -1;

    return 0;
}


int
UINT_Poly_random(UINT_Poly * a, int digits, int deg)
{
    int i, j, k;

    if (UINT_Poly_init_mem(a, digits, deg) < 0) {
        return -1;
    }

    for (i = 0; i <= deg; i++) {
        for (j = 0; j < digits; j++) {
            for (k = 0; k < sizeof(uint32_t); k++) {
                a->v[i][j] += (uint32_t) rand() << 8 * k;
            }
        }
    }

    return 0;
}


int
UINT_Poly_eq(const UINT_Poly * a, const UINT_Poly * b)
{
    int deg = a->deg;
    int i;

    if (b->deg < a->deg) {
        deg = b->deg;
    }
```

```
    for (i = 0; i <= deg; i++) {
        if (ARTH_cmp(a->v[i], a->digits, b->v[i], b->digits) != 0) {
            return 0;
        }
    }

    if (a->deg < b->deg) {
        for (i = a->deg + 1; i <= b->deg; i++) {
            if (ARTH_cmp_digit(b->v[i], 0, b->digits) != 0) {
                return 0;
            }
        }
    } else if (b->deg < a->deg) {
        for (i = b->deg + 1; i <= a->deg; i++) {
            if (ARTH_cmp_digit(a->v[i], 0, a->digits) != 0) {
                return 0;
            }
        }
    }

    return 1;
}


int
UINT_Poly_mul(UINT_Poly * c, const UINT_Poly * a, const UINT_Poly * b)
{
    UINT_Poly t;
    uint32_t *d;
    int i, j;

    /* Alokacja pamięci na wynik. */
    UINT_Poly_init_mem(&t, a->digits + b->digits + 1, a->deg + b->deg);
    d = calloc(t.digits - 1, sizeof(uint32_t));

    /* Mnożenie. */
    for (i = 0; i <= a->deg; i++) {
        for (j = 0; j <= b->deg; j++) {
            ARTH_mul(d, a->v[i], a->digits, b->v[j], b->digits);
            t.v[i + j][t.digits - 1] += ARTH_add(t.v[i + j], d, t.digits - 1);
        }
    }

    /* Zapisanie wyniku. */
    UINT_Poly_clear(c);

    c->v = t.v;
    c->digits = t.digits;
    c->deg = t.deg;

    free(d);

    return 0;
}


int
UINT_Poly_print(const UINT_Poly * a)
{
    int i, j;

    for (i = 0; i <= a->deg; i++) {
        printf("[%d] ", i);
        for (j = a->digits - 1; j > -1; j--) {
            printf(":%08x", a->v[i][j]);
        }
        printf("\n");
```

```
    }

    return 0;
}
```

B.4 Plik arth.c - CRT i funkcje arytmetyczne

```
#include "uintpoly_locl.h"


static uint32_t
mod_mul(uint32_t a, uint32_t b, uint32_t m)
{
    uint32_t c = 0;

    ARTH_MUL(a, a, b, c);
    ARTH_DIV(c, a, c, a, m);

    return a;
}


static uint32_t
mod_exp(uint32_t a, uint32_t e, uint32_t m)
{
    uint32_t t = 1;

    while (e) {
        if (e & 1) {
            t = mod_mul(t, a, m);
        }
        a = mod_mul(a, a, m);
        e >>= 1;
    }

    return t;
}


int
ARTH_cmp(const uint32_t * op1, int len1, const uint32_t * op2, int len2)
{
    int len = len1;
    int i;

    if (len2 < len1) {
        len = len2;
    }

    for (i = len - 1; i > -1; i--) {
        if (op1[i] < op2[i]) {
            return 1;
        } else if (op1[i] > op2[i]) {
            return -1;
        }
    }

    if (len1 < len2) {
        return ARTH_cmp_digit(op2 + len1, 0, len2 - len1);
    } else if (len2 < len1) {
        return ARTH_cmp_digit(op1 + len2, 0, len1 - len2);
    }

    return 0;
}
```

```
int
ARTH_cmp_digit(const uint32_t * op1, uint32_t op2, int len)
{
    int i;

    for (i = len - 1; i > 0; i--) {
        if (op1[i] != 0) {
            return -1;
        }
    }

    if (op1[0] < op2) {
        return 1;
    } else if (op1[0] > op2) {
        return -1;
    }

    return 0;
}


int
ARTH_add(uint32_t * dst, const uint32_t * src, int len)
{
    uint32_t c = 0;
    int i;

    for (i = 0; i < len; i++) {
        ARTH_ADC(dst[i], dst[i], src[i], c);
    }

    return (int) c;
}


int
ARTH_add_digit(uint32_t * dst, uint32_t src, int len)
{
    uint32_t c = 0;
    int i;

    ARTH_ADC(dst[0], dst[0], src, c);

    if (c == 1) {
        for (i = 1; i < len; i++) {
            dst[i] += 1;
            if (dst[i] != 0) {
                return 0;
            }
        }
    }

    return (int) c;
}


uint32_t
ARTH_mul_digit(uint32_t * a, int na, uint32_t b)
{
    uint32_t c = 0;
    int i;

    for (i = 0; i < na; i++) {
        ARTH_MUL(a[i], a[i], b, c);
    }

    return c;
```

```
}


uint32_t
ARTH_mul_digit_add(uint32_t * a, int na, const uint32_t * b, int nb, uint32_t d)
{
    uint32_t c = 0;
    int i;

    for (i = 0; i < nb; i++) {
        ARTH_MUL_ADD(a[i], b[i], d, c);
    }

    for (i = nb; i < na; i++) {
        ARTH_ADC(a[i], a[i], 0, c);
    }

    return c;
}


void
ARTH_mul(uint32_t * d, const uint32_t * a, int na, const uint32_t * b, int nb)
{
    uint32_t c;
    int i;
    int j;

    for (i = 0; i < na + nb; i++) {
        d[i] = 0;
    }

    for (i = 0; i < na; i++) {
        c = 0;
        for (j = 0; j < nb; j++) {
            ARTH_MUL_ADD(d[i + j], a[i], b[j], c);
        }
        d[i + j] = c;
    }
}


uint32_t
ARTH_mod_digit(const uint32_t * a, int na, uint32_t b)
{
    uint32_t q = 0;
    uint32_t r = 0;

    while (na--) {
        ARTH_DIV(q, r, r, a[na], b)
    }

    return r;
}


int
ARTH_Fp32_crt_init(ARTH_Fp32_crt * crt, const uint32_t * p, int n)
{
    int i, j;

    crt->n = n;

    /* Alokacja pamięci. */
    if ((crt->p = calloc(crt->n, sizeof(uint32_t))) == 0) {
        return -1;
    }
```

```
    if ((crt->c = calloc(crt->n, sizeof(uint32_t))) == 0) {
        free(crt->p);
        return -1;
    }

    if ((crt->pp = calloc(crt->n * crt->n, sizeof(uint32_t))) == 0) {
        free(crt->p);
        free(crt->c);
        return -1;
    }

    /* Kopiowanie liczb pierwszych. */
    for (i = 0; i < n; i++) {
        crt->p[i] = p[i];
    }

    /* Wyznaczenie współczynników dla algorytmu Garnera:
     *
     *   q[i] = (p[0]*p[1]*...*p[i-1])^(-1) mod p[i]
     *   pp[i*n] = p[0]*p[1]*...*p[i-1]
     */
    for (i = 0; i < n; i++) {
        crt->c[i] = 1;
        for (j = 0; j < i; j++) {
            crt->c[i] =
                mod_mul(crt->c[i], mod_exp(crt->p[j], crt->p[i] - 2, crt->p[i]),
                crt->p[i]);
        }
    }

    crt->pp[0] = 1;

    for (i = 1; i < n; i++) {
        crt->pp[i * n] = crt->p[0];
    }

    for (i = 1; i < n - 1; i++) {
        for (j = i + 1; j < n; j++) {
            crt->pp[j * n + i] = ARTH_mul_digit(crt->pp + j * n, i, crt->p[i]);
        }
    }

    return 0;
}


void
ARTH_Fp32_crt_free(ARTH_Fp32_crt * crt)
{
    if (crt->p) {
        free(crt->p);
        crt->p = 0;
    }

    if (crt->c) {
        free(crt->c);
        crt->c = 0;
    }

    if (crt->pp) {
        free(crt->pp);
        crt->pp = 0;
    }
}
```

B.5 Plik dft_fp32.c - dane dla DFT

```
#include "uintpoly_locl.h"


static uint32_t
mont_mul(uint32_t a, uint32_t b, uint32_t p, uint32_t q)
{
    uint64_t t1, t2, t3;

    t1 = (uint64_t) a *b;
    t2 = (uint64_t) (uint32_t) t1 *q;
    t2 = (uint64_t) (uint32_t) t2 *p;
    t3 = (t1 >> 32) + (t2 >> 32);

    if ((uint32_t) t1 != 0) {
        t3 += 1;
    }

    if ((uint64_t) p <= t3) {
        t3 -= (uint64_t) p;
    }

    return (uint32_t) t3;
}


static uint32_t
mont_exp(uint32_t a, int e, uint32_t p, uint32_t q)
{
    uint32_t t = (uint32_t) (0x0100000000ULL % p);
    int mask = 1 << 30;

    while (mask != 0) {
        t = mont_mul(t, t, p, q);
        if (e & mask) {
            t = mont_mul(t, a, p, q);
        }
        mask >>= 1;
    }

    return t;
}


static int
dft_index(int i, int prec)
{
    int rprec = 1;
    int di = 0;

    while (1 < prec) {
        prec >>= 1;
        if (i & prec) {
            di |= rprec;
        }
        rprec <<= 1;
    }

    return di;
}


int
ARTH_Fp32_dft_init(ARTH_Fp32_dft * dft, int prec, uint32_t p, uint32_t r)
{
    uint32_t q, s, t1, t2, ir, i2;
    int i, idx;
```

```
    memset(dft, 0, sizeof(ARTH_Fp32_dft));

    if (p < 3) {
        return -1;
    }

    /* Ustalenie maksymalnego rozmiaru wektorów do transformaty. */
    dft->n = 1;
    while (1) {
        /* Zbyt mała liczba pierwiastków. */
        if (((dft->n & (p - 1)) == 1) && (dft->n < prec)) {
            return -1;
        }
        /* Wystarczające liczba pierwiastków. */
        if (prec <= dft->n) {
            break;
        }
        dft->n <<= 1;
    }
    dft->roots = dft->n / 2;

    /* Wyznaczenie współczynnika q dla mnozenia Montgomeryego. */
    q = 1;
    while (p * q != 1) {
        q = q * (2 - p * q);
    }
    q = -q;

    /* t1 = t2 = 2^32 % p. */
    t1 = (uint32_t) (0x0100000000ULL % p);
    t2 = t1;
    /* s = 2^64 % p. */
    s = (uint32_t) (((uint64_t) t1 * t1) % p);
    dft->p = p;
    dft->q = q;
    dft->s = s;

    /* Transformacja r i 1/2 do postaci Montgomeryego. */
    r = mont_mul(r, s, p, q);
    i2 = mont_mul((p + 1) / 2, s, p, q);
    dft->i2 = i2;

    /* Alokacja pamięci. */
    if ((dft->r = calloc(dft->roots, sizeof(uint32_t))) == 0) {
        return -1;
    }

    if ((dft->ir = calloc(dft->roots, sizeof(uint32_t))) == 0) {
        free(dft->r);
        return -1;
    }

    /* Wyznaczenie pierwiastka pierwotnego i jego odwrotności dla zadanej precyzji. */
    r = mont_exp(r, (p - 1) / dft->n, p, q);
    ir = mont_exp(r, dft->n - 1, p, q);

    /* Przygotowanie tablicy pierwiastków i ich opdwrotności. */
    for (i = 0; i < dft->roots; i++) {
        idx = dft_index(i, dft->roots);
        dft->r[idx] = t1;
        dft->ir[idx] = t2;
        t1 = mont_mul(t1, r, p, q);
        t2 = mont_mul(t2, ir, p, q);
    }

    return 0;
}
```

```
int
ARTH_Fp32_dft_free(ARTH_Fp32_dft * dft)
{
    if (dft->r != 0) {
        free(dft->r);
    }

    if (dft->ir != 0) {
        free(dft->ir);
    }

    return 0;
}
```

B.6 Plik dft_fp32_crt.c - algorytm szybkiego mnożenia

```
#include "uintpoly_locl.h"


/* 32-bitowe liczby pierwsze postaci b*2^22 + 1. */
static const uint32_t p32[56] = {
    513 * 0x400000U + 1, 517 * 0x400000U + 1, 544 * 0x400000U + 1,
    553 * 0x400000U + 1, 559 * 0x400000U + 1, 565 * 0x400000U + 1,
    573 * 0x400000U + 1, 589 * 0x400000U + 1, 592 * 0x400000U + 1,
    604 * 0x400000U + 1, 610 * 0x400000U + 1, 627 * 0x400000U + 1,
    628 * 0x400000U + 1, 637 * 0x400000U + 1, 639 * 0x400000U + 1,
    645 * 0x400000U + 1, 648 * 0x400000U + 1, 649 * 0x400000U + 1,
    655 * 0x400000U + 1, 663 * 0x400000U + 1, 669 * 0x400000U + 1,
    670 * 0x400000U + 1, 684 * 0x400000U + 1, 688 * 0x400000U + 1,
    694 * 0x400000U + 1, 714 * 0x400000U + 1, 715 * 0x400000U + 1,
    733 * 0x400000U + 1, 742 * 0x400000U + 1, 757 * 0x400000U + 1,
    765 * 0x400000U + 1, 768 * 0x400000U + 1, 772 * 0x400000U + 1,
    790 * 0x400000U + 1, 814 * 0x400000U + 1, 819 * 0x400000U + 1,
    823 * 0x400000U + 1, 832 * 0x400000U + 1, 837 * 0x400000U + 1,
    847 * 0x400000U + 1, 853 * 0x400000U + 1, 859 * 0x400000U + 1,
    862 * 0x400000U + 1, 865 * 0x400000U + 1, 874 * 0x400000U + 1,
    879 * 0x400000U + 1, 894 * 0x400000U + 1, 915 * 0x400000U + 1,
    928 * 0x400000U + 1, 939 * 0x400000U + 1, 940 * 0x400000U + 1,
    957 * 0x400000U + 1, 972 * 0x400000U + 1, 979 * 0x400000U + 1,
    1000 * 0x400000U + 1, 1014 * 0x400000U + 1
};


/* Pierwiastki pierwotne dla liczb pierwszych z tablicy p32. */
static const uint32_t p32_r[56] = {
    7, 6, 3, 3, 3, 3, 5, 3, 3, 3, 3, 13, 3, 3, 7, 11, 5, 3, 6, 10, 15, 11,
    35, 3, 3, 10, 3, 3, 3, 3, 13, 5, 3, 6, 3, 10, 3, 3, 41, 3, 3, 3, 3, 3,
    3, 5, 5, 31, 3, 10, 3, 19, 7, 3, 3, 5
};


/* Maksymalny rozmiar wektorów do transformaty. */
static int dft_size = 0;


/* Współczynniki do obliczania transformaty dla liczb pierwszych z tablicy p32. */
static ARTH_Fp32_dft dft[56];


/* Współczynniki do CRT dla liczb pierwszych z tablicy p32. */
static ARTH_Fp32_crt crt;
```

```
static uint32_t
mont_add(uint32_t a, uint32_t b, uint32_t p)
{
    uint32_t c = 0;

    ARTH_ADC(a, a, b, c)

        if ((c == 1) || (p <= a)) {
        a -= p;
    }

    return a;
}


static uint32_t
mod_mul(uint32_t a, uint32_t b, uint32_t m)
{
    uint32_t c = 0;

    ARTH_MUL(a, a, b, c);
    ARTH_DIV(c, a, c, a, m);

    return a;
}


static uint32_t
mont_sub(uint32_t a, uint32_t b, uint32_t p)
{
    uint32_t c = 0;

    ARTH_SBB(a, a, b, c)

        if (c == 1) {
        a += p;
    }

    return a;
}


static uint32_t
mont_mul(uint32_t a, uint32_t b, uint32_t p, uint32_t q)
{
    uint64_t t1, t2, t3;

    t1 = (uint64_t) a *b;
    t2 = (uint64_t) (uint32_t) t1 *q;
    t2 = (uint64_t) (uint32_t) t2 *p;
    t3 = (t1 >> 32) + (t2 >> 32);

    if ((uint32_t) t1 != 0) {
        t3 += 1;
    }

    if ((uint64_t) p <= t3) {
        t3 -= (uint64_t) p;
    }

    return (uint32_t) t3;
}


static int
dft_size_for_poly(int deg)
```

```
{
    int size = 2;

    while (size < deg + 1) {
        size *= 2;
    }

    return size;
}


int
UINT_Poly_dft_crt_init(int deg_max)
{
    int i, j;

    dft_size = dft_size_for_poly(deg_max + 1);

    if (ARTH_Fp32_crt_init(&crt, p32, 56) < 0) {
        return -1;
    }

    for (i = 0; i < 56; i++) {
        if (ARTH_Fp32_dft_init(dft + i, dft_size, p32[i], p32_r[i]) < 0) {
            for (j = 0; j < i; j++) {
                ARTH_Fp32_dft_free(dft + j);
            }
            ARTH_Fp32_crt_free(&crt);
            return -1;
        }
    }

    return 0;
}


int
UINT_Poly_dft_crt_finish()
{
    int i;

    dft_size = 0;

    for (i = 0; i < 56; i++) {
        ARTH_Fp32_dft_free(dft + i);
    }

    ARTH_Fp32_crt_free(&crt);

    return 0;
}


int
UINT_Poly_dft_crt_mul(UINT_Poly * c, const UINT_Poly * a, const UINT_Poly * b)
{
    uint32_t *dft_a, *dft_b, *pa, *pb, *ha, *la, *hb, *lb;
    uint32_t p, q, r, ir, s, t, i2e;

    int deg_c;
    int primes;
    int m, n, bs, i, j, k;

    /* Stopień wielomianu wynikowego jest poza obsługiwanym zakresem. */
    deg_c = a->deg + b->deg;
    if (dft_size < deg_c + 1) {
        return -1;
    }
```

```
    /* Współczynniki iloczynu są zbyt duże, aby można je było reprezentować. */
    primes = a->digits + b->digits + 2;
    if (56 < primes) {
        return -1;
    }

    /* Alokacja pamięci. */
    n = dft_size_for_poly(deg_c);
    dft_a = calloc(n * primes, sizeof(uint32_t));
    dft_b = calloc(n * primes, sizeof(uint32_t));

    /*
     * 1) Przygotowanie wektorów do transformaty w formie Montgomeryego.
     * 2) Przeprowadzenie transformaty FFT.
     * 3) Wykonanie mnożenia.
     * 4) Przeprowadzenie transformaty odwrotnej.
     * 5) Konwersja do reprezentacji klasycznej.
     */
#pragma omp parallel for default(none) shared(dft, a, b, c, dft_a, dft_b, deg_c, primes, n) \
private(pa, pb, ha, la, hb, lb, p, q, r, ir, s, t, i2e, m, bs, i, j, k)
    for (i = 0; i < primes; i++) {
        p = dft[i].p;
        q = dft[i].q;
        s = dft[i].s;
        pa = dft_a + i * n;
        pb = dft_b + i * n;

        memset(pa, 0, n * sizeof(uint32_t));
        memset(pb, 0, n * sizeof(uint32_t));

        /* (1) Przejście do reprezentacji Montgomeryego. */
        for (j = 0; j <= a->deg; j++) {
            pa[j] = mont_mul(ARTH_mod_digit(a->v[j], a->digits, p), s, p, q);
        }
        for (j = 0; j <= b->deg; j++) {
            pb[j] = mont_mul(ARTH_mod_digit(b->v[j], b->digits, p), s, p, q);
        }

        /* (2) FFT. */
        m = n;

        while (m != 1) {
            bs = n / m;
            m /= 2;
            for (j = 0; j < bs; j++) {
                r = dft[i].r[j];
                la = pa + 2 * j * m;
                ha = la + m;
                lb = pb + 2 * j * m;
                hb = lb + m;
                for (k = 0; k < m; k++, la++, ha++, lb++, hb++) {
                    t = mont_mul(*ha, r, p, q);
                    *ha = mont_sub(*la, t, p);
                    *la = mont_add(*la, t, p);
                    t = mont_mul(*hb, r, p, q);
                    *hb = mont_sub(*lb, t, p);
                    *lb = mont_add(*lb, t, p);
                }
            }
        }

        /* (3) Mnożenie wektorów wyraz po wyrazie. */
        for (j = 0; j < n; j++) {
            pa[j] = mont_mul(pa[j], pb[j], p, q);
        }

        /* (4) FFT^(-1). */
```

```
        m = 1;
        i2e = dft[i].ir[0];

        while (m < n) {
            bs = n / (2 * m);
            i2e = mont_mul(i2e, dft[i].i2, p, q);
            for (j = 0; j < bs; j++) {
                ir = dft[i].ir[j];
                la = pa + 2 * j * m;
                ha = la + m;
                for (k = 0; k < m; k++, la++, ha++) {
                    t = mont_add(*la, *ha, p);
                    *ha = mont_sub(*la, *ha, p);
                    *la = t;
                    *ha = mont_mul(*ha, ir, p, q);
                }
            }
            m *= 2;
        }

        for (j = 0; j < n; j++) {
            pa[j] = mont_mul(pa[j], i2e, p, q);
        }

        /* (5) Konwersja do reprezentacji klasycznej. */
        for (j = 0; j < n; j++) {
            pa[j] = mont_mul(pa[j], 1, p, q);
        }
    }

    /*
     * 6) Użycie algorytmu Garnera do rekonstrukcji wyniku.
     */
#pragma omp parallel for default(none) shared(dft, crt, a, b, c, dft_a, dft_b, deg_c, primes, n) \
private(pa, pb, ha, la, hb, lb, p, q, r, ir, s, t, i2e, m, bs, i, j, k)
    for (i = 0; i <= deg_c; i++) {
        pa = dft_a + i;
        pb = dft_b + i * primes;
        memset(pb, 0, primes * sizeof(uint32_t));

        pb[0] = pa[0];

        for (j = 1; j < primes; j++) {
            t = ARTH_mod_digit(pb, j, crt.p[j]);
            if (t != 0) {
                t = crt.p[j] - t;
            }
            t += pa[j * n];
            if ((t < pa[j * n]) || (crt.p[j] <= t)) {
                t -= crt.p[j];
            }
            t = mod_mul(t, crt.c[j], crt.p[j]);
            ARTH_mul_digit_add(pb, j + 1, crt.pp + j * crt.n, j, t);
        }
    }

    UINT_Poly_clear(c);

    /* Wyznaczenie maksymalnej liczby cyfr dla współczynników c. */
    c->digits = 1;
    for (i = 0; i <= deg_c; i++) {
        pb = dft_b + i * primes;
        for (j = 0; j < primes; j++) {
            if ((pb[j] != 0) && (c->digits <= j)) {
                c->digits = j + 1;
            }
        }
    }
```

```
    /* Skopiowanie wyniku. */
    c->deg = deg_c;
    c->v = calloc(c->deg + 1, sizeof(uint32_t *));
    for (i = 0; i <= deg_c; i++) {
        c->v[i] = calloc(c->digits, sizeof(uint32_t));
        pb = dft_b + i * primes;
        for (j = 0; j < c->digits; j++) {
            c->v[i][j] = pb[j];
        }
    }

    free(dft_a);
    free(dft_b);

    return 0;
}
```