Министерство науки и высшего образования Российской Федерации

Санкт-Петербургский Политехнический Университет Петра Великого

—

Институт Кибербезопасности и Защиты Информации

ЛАБОРАТОРНАЯ РАБОТА № 8

«Разложение числа на множители на эллиптической кривой»

по дисциплине «Криптографические методы защиты информации»

Выполнил
студент гр. 4851003/80802 Сошнев М.Д.

<подпись>

Проверил
преподаватель Ярмак А.В.

<подпись>

Санкт-Петербург

2022

Цель работы

Реализация метода разложения числа на множители с использованием эллиптических кривых.

Задача

Получить у преподавателя вариант задания и разработать программу **П-1**, которая находит разложение составного числа n на эллиптической кривой. В качестве входных данных программа **П-1** должна принимать число n, размер m базы D; на выходе – возвращать нетривиальный делитель d числа n.

Ход работы

В результате выполнения работы была разработана программа, которая с помощью алгоритма эллиптических кривых выполняет факторизацию числа. При разработке использовался язык программирования python. Для начала был разработан класс эллиптической кривой и точки н аэллиптической кривой — реализована процедура сложения точек на эллиптической кривой. С помощью быстрого алгоритма возведения в степень была реализована процедура умножения точки эллиптической кривой на число. Далее, с использованием данных классов был реализован сам алгоритм факторизации числа. Его входными данными являются составное число и размер базы. Алгоритм, генерируя различные эллиптические кривые пытается решить задачу факторизации на них (в данной программе различные эллиптические кривые проверяются последовательно, но в перспективе можно разделять данный алгоритм на несколько потоков). Продемонстрируем работу программы на 64-битном числе n=9679022848099028737:

```
┌──(maxim㉿maxim)-[~/…/4year/КМЗИ/Work/EllipticAlgs]
└─$ python main.py --number 9679022848099028737 --base 200
n = 9679022848099028737
m = 200
Кривая: y^2 = x^3 + 1253264795999317445x + 5952976954208151294 (mod 9679022848099028737)
Точка: Q = (7987152860504744532 ; 1345308780473837582)

Кривая: y^2 = x^3 + 7167724264800316431x + 3715069910705310054 (mod 9679022848099028737)
Точка: Q = (5073445374074458493 ; 9527203863914904264)

Кривая: y^2 = x^3 + 3703762667300122921x + 897087821299785183 (mod 9679022848099028737)
Точка: Q = (7973785993345603552 ; 8437621415243853167)

Кривая: y^2 = x^3 + 8677404615316368369x + 2910269157801347542 (mod 9679022848099028737)
Точка: Q = (429015186263905116 ; 9246348792034595350)

Кривая: y^2 = x^3 + 68095866492035854 42x + 7501356899673277084 (mod 9679022848099028737)
Точка: Q = (4511452321821463 56 ; 80729359838025733 90)

Успешно!
p = 3111111191
q = 3111114407
```

Рисунок 1 — успешный результат работы программы

Алгоритм успешно нашел множитель p = 3111111191

Контрольные вопросы

1. Как зависит сложность разложения составного числа заданной длины методом эллиптических кривых от числа различных простых делителей числа n?

Алгоритм Ленстры зависит от длины минимального простого делителя — следовательно при увеличении количества делителей их длина будет уменьшаться, а вместе с ней и скорость работы алгоритма.

2. Сравните сложность разложения на эллиптической кривой составного числа вида $n = p^2 q$, где p, q − различные простые числа, и составного числа такой же длины, состоящего из двух различных простых делителей.

Пусть есть два числа следующего вида и одинаковой длины:

$$n_1 = p_1^2 q_1, n_2 = p_2 q_2.$$

Пусть длины множителей q_1 и q_2 равны, тогда длины множителей p_1^2 и p_2 будут также равными.

Следовательно, будет справедливо следующее неравенство относительно длин множителей: $|p_1| < |p_2|$. Так как при использовании алгоритма Ленстры быстрее раскладываются те числа, которые имеют простой делитель меньшего размера, то в данном случае сложность разложения n_1 будет меньше сложности разложения n_2.

Вывод

В результате выполнения работы была изучена математическая модель эллиптических кривых и применена на практике — реализован аналог p-1 метода полларда, который с использованием эллиптических кривых находит нетривиальный делитель составного числа.

Приложение

```
import random
import math
import rsa

import matplotlib.pyplot as plt
import numpy as np



def bits(n):
    """
    Генерирует двоичные разряды n, начиная
    с наименее значимого бита.

    bits(151) -> 1, 1, 1, 0, 1, 0, 0, 1
    """
    while n:
        yield n & 1
        n >>= 1



class EllipticCurvePoint:
    def __init__(self, x, y, curve):
        self.x = x
        self.y = y
        self.__curve = curve

    def __str__(self):
        return '({} ; {})'.format(self.x, self.y)

    def clone(self):
        return EllipticCurvePoint(self.x, self.y, self.__curve)

    def __add__(self, other):
        if self.x == other.x:
            m = (3 * self.x ** 2 + self.__curve.a) * rsa.common.inverse(2 * self.y, self.__curve.n)
            m %= self.__curve.n
        else:
            dx = (self.x - other.x) % self.__curve.n
            dy = (self.y - other.y) % self.__curve.n
            m = dy * rsa.common.inverse(dx, self.__curve.n)
            m %= self.__curve.n

        x = (m ** 2 - self.x - other.x) % self.__curve.n
        y = (self.y + m * (x - self.x)) % self.__curve.n
        y = (-y) % self.__curve.n

        return EllipticCurvePoint(x, y, self.__curve).check_curve_exist()
```

```
    def __mul__(self, n: int):
        """
        Возвращает результат n * self, вычисленный
        алгоритмом удвоения-сложения.
        """
        result = None
        addend = self

        for bit in bits(n):
            if bit == 1:
                if result is None:
                    result = addend.clone()
                else:
                    result += addend
            addend = addend + addend

        return result.check_curve_exist()

    def check_curve_exist(self):
        # if pow(self.y, 2, self.__curve.n) != \
            # (pow(self.x, 3, self.__curve.n) + self.__curve.a * self.x + self.__curve.b) %
self.__curve.n:
            # print('[WARNING] {} не лежит на кривой {}'.format(self, self.__curve))
            # pass
        return self

    def show(self):
        self.__curve.show(self)



class EllipticCurve:
    def __init__(self, a, b, n):
        self.a = a
        self.b = b
        self.n = n

    def __str__(self):
        return 'y^2 = x^3 {} {}x {} {} (mod {})'.format(
            '+' if self.a >= 0 else '-',
            self.a,
            '+' if self.b >= 0 else '-',
            self.b,
            self.n)

    def __call__(self, x):
        y_2 = (x ** 3 + self.a * x + self.b) % self.n
        Y = []
        for y in range(self.n):
            if pow(y, 2, self.n) == y_2:
                Y.append(y)
```

```
        return Y

    @staticmethod
    def generate_curve_and_point(n):
        while True:
            a = random.randint(0, n - 1)
            x = random.randint(0, n - 1)
            y = random.randint(0, n - 1)
            b = (y ** 2 - x ** 3 - a * x) % n

            if (4 * a ** 3 + 27 * b ** 2) % n != 0:
                break

        ec = EllipticCurve(a, b, n)
        point = EllipticCurvePoint(x, y, ec)

        return ec, point

    def show(self, point=None):
        x_space = np.linspace(0, self.n, self.n+1)
        x_show, y_show = [], []
        for x in x_space:
            Y = self(x)
            for y in Y:
                x_show.append(x)
                y_show.append(y)

        plt.scatter(x_show, y_show)
        if point is not None:
            plt.scatter(point.x, point.y, c='black')

        plt.show()


import argparse
import rsa.prime
import threading
from elliptic_cryptography import *



def prime_generator(n):
    yield 2
    yield 3

    p, i = 5, n - 2
    while True:
        if rsa.prime.is_prime(p):
            i -= 1
            yield p
```

```
        p += 2
        if i == 0:
            break



class Attack(threading.Thread):
    def __init__(self, n, m, on_done, on_iter):
        super().__init__()
        self.__n = n
        self.__m = m
        self.__on_done = on_done
        self.__on_iter = on_iter

    def run(self):
        if rsa.prime.is_prime(self.__n):
            return self.__on_done(self.__n, None)

        print('n = {}'.format(self.__n))
        print('m = {}'.format(self.__m))

        while True:
            # Генерируем эллиптическую кривую и точку на ней

            ec, Q = EllipticCurve.generate_curve_and_point(self.__n)
            # ec = EllipticCurve(-1, 3231, n)
            # Q = EllipticCurvePoint(87, 2, ec)

            self.__on_iter(ec, Q)

            for i, p in enumerate(prime_generator(self.__m)):
                a = int(math.log2(self.__n) / math.log2(p) / 2)
                try:
                    for j in range(a):
                        Q *= p
                except rsa.common.NotRelativePrimeError as ex:
                    self.__on_done(self.__n, ex.d)
                    return



def report(n, p):
    if (p is not None) and (n % p == 0) and p != n:
        q = n // p

        print('Успешно!')
        print('p = {}'.format(p))
        print('q = {}'.format(q))
    else:
        print('Безуспешно')
```

```
def iter(ec, Q):
    print('Кривая: {}'.format(ec))
    print('Точка: Q = {}\n'.format(Q))



def main(context):
    n = int(context.number)
    m = int(context.base)

    attack = Attack(n, m, on_done=report, on_iter=iter)
    attack.start()



if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument('--number', required=True)
    parser.add_argument('--base', required=True)

    args = parser.parse_args()

    main(args)
```