Правительство Российской Федерации
Федеральное государственное автономное образовательное
учреждение высшего образования
"Национальный исследовательский университет
"Высшая школа экономики"
Московский институт электроники и математики им. А.Н.Тихонова
Кафедра «Компьютерная безопасность»

ОТЧЕТ

по дисциплине «Теоретико-числовые методы в криптографии»

Программная реализация алгоритма Полига – Хеллмана

Выполнили студенты гр. СКБ181:
Петров Артём
Рымкулова Диана

Проверил доцент
Нестеренко А.Ю.

Москва, 2022 г.

Введение

Алгоритма Полига – Хеллмана – это детерминированный алгоритм дискретного логарифмирования в кольце вычетов по модулю простого числа.

Данный алгоритм был впервые опубликован американскими математиками Стефаном Полигом и Мартином Хеллманом в 1978 году [1]. Важной особенностью этого метода является то, что для простых чисел специального вида, можно находить дискретный логарифм за полиномиальное время.

Теоретическая часть

Решаемая задача

Входные данные:	a, p, y, причем мы знаем, что p – простое число и $$p-1=\prod_{i=1}^{k} q_i^{\alpha_i}$$ На практике всегда рассматривается случай, когда a – примитивный элемент $GF(p)$
Выходные данные:	x, $0 \le x \le p-2$, которое удовлетворяет сравнению $a^x \equiv y \ (mod\ p)$

Идея алгоритма

Суть алгоритма в том, что достаточно найти x по модулям $q_i^{\alpha_i}$ для всех i, а затем решение исходного сравнения можно найти с помощью китайской теоремы об остатках. Чтобы найти x по каждому из таких модулей, нужно решить сравнение:

$$(a^{\frac{p-1}{\left(q_i^{\alpha_i}\right)}})^x \equiv \ y^{\frac{p-1}{\left(q_i^{\alpha_i}\right)}} \ (mod\ p)$$

Особый случай

Как предлагается в исходной статье, рассмотрим особый случай, когда $p = 2^n + 1$. В этом случае, мы можем представить искомый x следующим образом:

$$x = \sum_{i=0}^{n-1} b_i 2^i$$

Задача поиска x сводится к поиску b_i, где $i \in \{0, n-1\}$

Для нахождения младшего бита b_0 вычислим $y^{\frac{p-1}{2}} \equiv y^{2^{n-1}} \ (mod\ p)$

Как мы знаем из условия,

$$y^{\frac{p-1}{2}} \equiv (a^x)^{\frac{p-1}{2}} \equiv \left(a^{\frac{p-1}{2}}\right)^x \ (mod\ p) \qquad (1)$$

По малой теореме Ферма получим:

$$a^{p-1} \equiv 1 \equiv a^0 \ (mod\ p)$$

$$a^{p-1} - 1 \ \equiv 0 \ (mod\ p)$$

$$\left(a^{\frac{p-1}{2}}-1\right)\left(a^{\frac{p-1}{2}}+1\right) \equiv 0\ (mod\ p)$$

Но так как $a^{\frac{p-1}{2}} \not\equiv 1(mod\ p)$

$$a^{\frac{p-1}{2}} \equiv -1\ (mod\ p)$$

Тогда выражение (1) будет выглядеть:

$$y^{\frac{p-1}{2}} \equiv (a^x)^{\frac{p-1}{2}} \equiv (-1)^x\ (mod\ p)$$

$$y^{\frac{p-1}{2}} \equiv y^{2^{n-1}} \equiv \begin{cases} +1 & b_0 = 0 \\ -1 & b_0 = 1 \end{cases}$$

Следующий бит разложения определяется следующим образом

$$z \equiv ya^{-b_0} \equiv a^{x_1}\ (mod\ p), \text{где}$$

$$x_1 = \sum_{i=1}^{n-1} b_i 2^i$$

Заметим, что x_1 делится на 4 только, если $b_1 = 0$. Но если $b_1 = 1$, то x_1 делится на 2, но не на 4.

Рассуждая как раньше, получим

$$z_1^{\frac{p-1}{2}}(mod\ p) \equiv \begin{cases} +1 & b_1 = 0 \\ -1 & b_1 = 1 \end{cases}$$

По аналогии мы можем получить оставшиеся биты.

Обобщим полученный результат и напишем общую формулу вычисления b_i.

Введем переменные:

$$m_i = \frac{p-1}{2^{i+1}}$$

$$z_i \equiv ya^{-b_0-b_1 2^1-\cdots-b_{i-1}2^{i-1}} \equiv a^{x_i}\ (mod\ p), \text{где } x_i = \sum_{k=i}^{n-1} b_k 2^k$$

Тогда b_i можно получить, возведя z_i в степень m_i:

$$z_i^{m_i} \equiv a^{(x_i * m_i)} \equiv \left(a^{\frac{p-1}{2}}\right)^{\frac{x_i}{2^i}} \equiv (-1)^{\frac{x_i}{2^i}} \equiv (-1)^{b_i}\ (mod\ p)$$

Следовательно

$$z_i^{m_i}\ (mod\ p) \equiv \begin{cases} +1 & b_i = 0 \\ -1 & b_i = 1 \end{cases}$$

Найдя все биты, получаем ответ:

$$x = \sum_{i=0}^{n-1} b_i 2^i$$

Общий случай

Шаг 1. Подсчет значений $r_{i,j}$

$$r_{i,j} = a^{j\frac{p-1}{q_i}}, i \in \{1, \dots, k\}, j \in \{0, \dots, q_i - 1\}$$

Шаг 2. Вычисление $\log_{\mathrm{a}} y \ \left(mod\ q_i^{\alpha_i}\right)$

Определим $x \equiv \log_{\mathrm{a}} y \equiv x_0 + x_1 q_i + \cdots + x_{\alpha_i - 1} q_i^{\alpha_i - 1} \left(mod\ q_i^{\alpha_i}\right)$,

где $0 \leq x_i \leq q_i - 1$

Тогда верно сравнение:

$$a^{x_j\frac{p-1}{q_i}} \equiv \left(y a^{-x_0 - x_1 q_i - \cdots - x_{\alpha_i - 1} q_i^{j-1}}\right)^{\frac{p-1}{q_i^{j+1}}} (mod\ p)$$

С помощью найденных значений j и i в первом шаге, производим вычисления и находим x из сравнения (в зависимости от шага 2 определяются какие именно x (пр. x_0, x_1, x_2 и тд) необходимо найти). Исходя из полученного с помощью вычисления сравнения также находим ответ с помощью найденных в шаге 1 значений.

Шаг 3. Нахождение ответа

Найдя $\log_{\mathrm{a}} y \ \left(mod\ q_i^{\alpha_i}\right)$ для всех i, находим $\log_{\mathrm{a}} y \ \ (mod\ p - 1) \equiv \log_{\mathrm{a}} y \ \ (mod\ p)$ по китайской теореме об остатках.

Программная реализация

Реализуем алгоритм на Python на системе Sage [2] с использованием следующих модулей.

```
from sage.all import * # для использования возможностей Sage
import random # для генерации тестовых значений
import time # для замера времени
```

Особый случай

Как и предлагается в статье, рассмотрим особый случай, когда $p = 2^n + 1$

```
def simple_case_alg(a, y, p, factor_list):
    if len(factor_list) != 1 or factor_list[0][0] !=2:
        raise Exception('Wrong value of argument factor_list')
    log_p = factor_list[0][1]
    z_i = y
    B_i = Mod(a ^ (-1), p)
    m_i = (p-1)/2
    base = 1
    x = 0
    for i in range(0, log_p):
        q = Mod(z_i ^ m_i, p)
        if q == -1:
            x += base
            z_i = Mod(z_i * B_i, p)
        elif q != 1:
            raise Exception('Wrong value of q (not 1 and not -1). Found: %s', str(q))
        base *= 2
        m_i /= 2
        B_i = Mod(B_i ^ 2, p)
    return x
```

Входные аргументы:

- `a, y, p` – входные параметры для решаемой задачи
- `factor_list` – список (list) кортежей, содержащий значения (q_i, α_i), то есть основание и показатель степени из разложения p. Данный формат задан возвращаемым значением функции factor из библиотеки sage

Проверка входных значений:

Реализована проверка корректности заполнения `factor_list`: количество множителей в разложении должно быть равно 1 и первый элемент должен представлять степень числа 2.

Вводимые переменные:

`x` – переменная для хранения результата алгоритма

`log_p` – это n из $p = 2^n + 1$

`z_i` – представляет значение выражения

$$z_i \equiv y a^{-b_0 - b_1 2^1 - \dots - b_{i-1} 2^{i-1}} \ (mod\ p)$$

`B_i` – имеет значение a^{2^i} и необходим, чтобы осуществлять переход

$$z_{i+1} \equiv z\_i * B\_i \ \ (mod\ p)$$

`m_i` – переменная со значением $\frac{p-1}{2^i}$

`base` – переменная со значением 2^i, использующаяся для изменения значения `x`

Также в основном теле цикла добавлена проверка на допустимые значения $z_i^{m_i}$: если оно не равно 1 или -1, то будет вызвано исключение с соответствующим значением.

После цикла возвращается содержимое переменной x, которое является ответом

Общий случай:

```
def get_r_ij(a, j, q, p):
    return Mod(a ^ (((p-1)/q)*j), p)

def get_r_ij_dict(a, p, factor_list):
    r = dict()
    for i in range(len(factor_list)):
        factor = factor_list[i]
        q = factor[0]
        r[q] = []
        for j in range(0, q):
            r_ij = get_r_ij(a, j, q, p)
            r[factor[0]].append(r_ij)
    return r

def common_case_alg(a, b, p, factor_list):
    r = get_r_ij_dict(a, p, factor_list)

    x = []
    q_x = []
    for i in range(len(factor_list)):
        factor = factor_list[i]
        q = factor[0]
        q_a = q
        base_additional = 1
        x_i = 0
        for j in range(0, factor[1]):
            result = Mod(Mod(b * base_additional, p) ^ ((p-1)/q_a), p)
            for k in range(0, len(r[q])):
                if r[q][k] == result:
                    pre_calc = Mod(a^((-1) * k * q^j), p)
                    base_additional = Mod(base_additional * pre_calc, p)
                    x_i = Mod(x_i + k * q^j, p)
                    break
            q_a *= q
        x.append(sage.rings.integer.Integer(x_i))
        q_x.append(q^factor[1])
    return CRT_list(x, q_x)
```

Весь набор входных аргументов аналогичен simple_case_alg

Функция get_r_ij_dict выполняет шаг 1 из теоретического описания алгоритма.

x и q_x представляют собой описание системы для решения с помощью Китайской теоремы об остатках (CRT_list). Иными словами, список значений $x = [x_i], q_x = [q_{x_i}]$ таких, что

$$x \equiv x_i \left(mod\ q_{x_i}\right)$$

Тестирование

Продемонстрируем то, что алгоритмы работают:

Вызов функции	`p = 5` `print(simple_case_alg(2, 3, p, list(factor(p-1))))`
Результат	3
Проверка	$2^3 = 8 \equiv 3\ (mod\ 5)$

Вызов функции	`p = 7` `print(common_case_alg(3, 5, p, list(factor(p-1))))`
Результат	5
Проверка	$3^5 = 243 \equiv 5\ (mod\ 7)$

Теперь попробуем сравнить оба алгоритма, проверив, есть ли смысл использовать simple_case_alg при $p = 2^n + 1$ в итоговой реализации.

План: для начала необходимо сгенерировать тестовые данные, потом прогнать на них оба наших алгоритма с одинаковыми данными и сравнить итоговое время.

Для упрощения процесса сравнения алгоритмов, создадим метод, который будет последовательно запускать оба алгоритма на одном и том же наборе тестовых данных, высчитывать общее и среднее время работы на всех тестовых случаях.

```
def run_timetest(samples, first_alg, second_alg):
    f_count = 0
    f_sum = 0
    s_count = 0
    s_sum = 0
    for sample in samples:
        factorization=list(factor(sample[3]-1))
        # first algorithm
        start_time_first = time.time()
        result = check_alg(first_alg, sample, factorization)
        end_time_first = time.time()
        # check answer
        if not check_answer(sample, result):
            print("first have ERROR", sample, result)
        else:
            f_count +=1
            f_sum += (end_time_first - start_time_first)

        # second algorithm
        start_time_second = time.time()
        result = check_alg(second_alg, sample, factorization)
        end_time_second = time.time()
        # check answer
        if not check_answer(sample, result):
            print("second have ERROR", sample, result)
        else:
            s_count +=1
            s_sum += (end_time_second - start_time_second)
    # report
    report_template = "Algorithm: %s.\n Count: %d. All time: %f. Avg time: %f"
    print(report_template % (first_alg.__name__ , f_count, f_sum, (f_sum/f_count)))
    print(report_template % (second_alg.__name__ , s_count, s_sum, (s_sum/s_count)))
```

Определим, как будем генерировать тестовые данные. Так как у нас есть требование: a должно быть примитивным элементом – найдем способ его поиска.
Самый простой и явный, который удалось найти в sage это через построение конечного поля

Вызов функции	`GF(7, modulus="primitive").gen()`
Результат	3
Проверка	$3^1 \equiv 3\ (mod\ 7);\ 3^2 \equiv 2\ (mod\ 7);\ 3^3 \equiv 6\ (mod\ 7);$ $3^4 \equiv 4\ (mod\ 7);\ 3^5 \equiv 5\ (mod\ 7);\ 3^6 \equiv 1\ (mod\ 7);$

Так как для «простого случая» число p должно выглядеть как $p = 2^n + 1$, но при этом быть простым, то найдем такие числа:

Вызов функции:

```
list_p = []
base = 2
for i in range (1, 100):
    if is_prime(int(base+1)):
        list_p.append(base+1)
    base *= 2
print(list_p)
```

Результат: [3, 5, 17, 257, 65537]

Для наших тестов подходит 65537: оно не очень большое, поэтому ожидание подсчета логарифма будет не долгим, но в то же время не маленькое, в разрезе чего мы сможем увидеть разницу между двумя алгоритмами.

С учетом метода поиска примитивного элемента, напишем функцию для генерация тестовых случаев:

```
def generate_samples(list_p, count):
    samples = []
    for p in list_p:
        a = GF(p, modulus="primitive").gen()
        for i in range(count):
            x = randint(2, p-2)
            b = int(Mod(a^x, p))
            samples.append((a, x, b, p))
    return samples
```

Вызов функции	`samples = generate_samples([65537],10000)`
Результат	*Список samples, состоящий из 10000 кортежей вида (a, x, b, p), причем* $a^x \equiv b\ (mod\ p), p = 65537$

Теперь на наборе таких тестовых данных мы можем запустить каждый из алгоритмов.

Вызов функции	`samples = generate_samples([65537],10000)` `run_timetest(samples, simple_case_alg, common_case_alg)`
Результат	Algorithm: simple_case_alg. Count: 10000. All time: 19.459785. Avg time: 0.001946 Algorithm: common_case_alg. Count: 10000. All time: 6.714492. Avg time: 0.000671

Вызов функции	`samples = generate_samples([65537],100000)` `run_timetest(samples, simple_case_alg, common_case_alg)`
Результат	Algorithm: simple_case_alg. Count: 100000. All time: 204.730563. Avg time: 0.002047 Algorithm: common_case_alg. Count: 100000. All time: 73.273343. Avg time: 0.000733

По результатам мы видим, что алгоритм для общего случая быстрее упрощенного, поэтому в итоговой реализации нет смысла в разделении на «общий» и «особый» случай.

Проведем также тест в сравнении с перебором всех значений. Для этого потребуется функция для генерации p.

```
def generate_list_of_p(p_min, p_max, count):
    p = []
    i = 0
    fail = 0
    while i < count:
        random_p = randint(p_min, p_max)
        next_p = next_prime(random_p)
        if next_p <= p_max:
            p.append(next_p)
            i += 1
        else:
            fail += 1
        if fail > count and len(p) == 0:
            raise Exception("Bad attempt to generate p")
    return p
```

Ну и напишем алгоритм полного перебора

```
def dummy_alg(a, b, p, factor_list):
    result = 1
    degree = 0
    while degree < p:
        if result == b:
            return degree
        result = int(Mod(result*a, p))
        degree += 1
    raise Exception("Not found result for %d^x = %d (mod %d)" % (a,
b, p))
```

Проведем сравнение двух алгоритмов:

Вызов функции	`list_p = generate_list_of_p(10^2, 10^4, 10)`
	`samples = generate_samples(list_p, 100)`
	`run_timetest(samples, common_case_alg, dummy_alg)`

Результат	Algorithm: common_case_alg. Count: 1000. All time: 3.508328. Avg time: 0.003508 Algorithm: dummy_alg. Count: 1000. All time: 16.090028. Avg time: 0.016090

Вызов функции	`list_p = generate_list_of_p(10^4, 10^6, 10)`
	`samples = generate_samples(list_p,100)`
	`run_timetest(samples, common_case_alg, dummy_alg)`

Результат	Algorithm: common_case_alg. Count: 1000. All time: 130.164235. Avg time: 0.130164 Algorithm: dummy_alg. Count: 1000. All time: 974.589804. Avg time: 0.974590

Выводы

Алгоритм Полига—Хеллмана крайне эффективен, если $p-1$ раскладывается на небольшие простые множители. Игнорирование этого факта при выборе параметров криптографических схем влияет на их стойкость. Тут стоит заметить тот факт, что при разложении $p-1$ на небольшие множители приводит к тому, что задача факторизации не является существенной и может решиться даже методом последовательного деления.

Сложность алгоритма является полиномиальной $O((\log p)^{c_1})$ в случае, когда все простые множители не превосходят $(\log p)^{c_2}$. c_1, c_2 – положительные константы [3]. В общем случае сложность алгоритма экспоненциальная.

Источники

1. S. C. Pohlig and M. E. Hellman. An Improved Algorithm for Computing Logarithms Over GF(p) and its Cryptographic Significance // IEEE Transactions on Information Theory. — 1978.
2. SageMath - open-source mathematics software system [Электронный ресурс]. URL: https://www.sagemath.org/
3. О. Н. Василенко. Теоретико-числовые алгоритмы в криптографии. — 2003.