Министерство образования и науки Российской Федерации

Московский государственный институт электроники и математики
(технический университет) Кафедра «Компьютерная безопасность»

# ОТЧЕТ К РАБОТЕ
# «**ро-методом Полларда**» по дисциплине
# «Теоретико-числовые методы в криптографии»

Работу выполнили
Студенты группы СКБ 181
Файнберг Т.
Кекеев А.

Работу проверил
Нестеренко А. Ю.

Москва, 2022

**Постановка задачи:**

Разработать алгоритм, выполняющий получение дискретного алгоритма числа b по основанию a и по модулю m на основе ρо-метода Полларда. Результат оформить в виде отчета.

**Использованные инструменты:**
В работе были использованы SageMath, JupyterNotebook

**Теоретическая база:**

Ро-Метод Полладна для нахождения дискретного алгоритма числа b по базе a и модулю p основывается на построениии множества случайных вычетов z, построеннныз по формуле $z_{n+1} = z_n * a^{\alpha[i]} b^{\beta[i]}$, где α и β — произвольные множества из случайных неповторяющихся значений в кольце m. Каждый индекс рассчитывается по формуле $z_n \% s$. Значение *s* является константой на время поиска логарифма конкретного числа.

По мере генерации чисел *z* в некоторый момент происходит зацикливание последовательности, т. к., находясь в кольце вычетов по модулю *p*, можно создать не менее *m = ord(p)* уникальных элементов последовательности, а, так как элементы последовательности (по построению) представимы в виде степеней числа *a*, то получить можно не более, чем порядок мультипликативной группы, порождаемой элементом *a*, уникальных элементов.
Следовательно, после генерации следующего по номеру за порядком такой группы элемента, образуется цикл. Имея два некоторых числа z и y, таких, что z = y, но ind(z) != ind(y) (здесь ind(z) означает порядковый номер числа z в нашей последовательности), получаем следующее:

$a^{A_z} b^{B_z} = a^{A_y} b^{B_y}$

где $A_z, B_z$ (и аналогично $A_y, B_y$) определяются следующим образом:

$A_z = \sum (alpha[z_i \% s]) + k_0$, $B_z = \sum (beta[z_i \% s])$, i $\in$ [1, ind(z)].

Так как b = $a^x$,

$$a^{A_z + x * B_z} = a^{A_y + B_y * x}$$

$$Az + Bz * x = Ay + By * x$$

$$x = (Az — Ay)/(By — Bz) \pmod m.$$

Для того, чтобы найти *x*, разность *(By — Bz)* должна быть обратима в кольце вычетов по модулю *m*. Для этого, должно выполняться *НОД(*$B_y - B_z$*, m) = 1*. В случае, если это условие не выполнено, так как в рамках решения задачи было решено пренебречь поиском мультипликативного порядка *a* и использовать *m*, выполняется понижение модуля до *m / НОД(*$B_y - B_z$*, m)* при условии *НОД(*$B_y - B_z$*, m) /¿)*. Если последнее условие не выполнено, алгоритм перезапускается на других значениях *alpha* и *beta*. После успешного понижения модуля, осуществляется поиск значения *x* по новому модулю и возврат к модулю *m* путем последовательного перебора значений $a^x$ и сравнения их с *b*. Найденное таким образом значение и является ответом.

Для поиска цикла в целях уменьшения объема используемой памяти, но ценой некоторой дополнительной операционной сложности, используется критерий Флойда — проверка условия $z_n = z_{2n}$

**Результаты выполнения работы:**

```
def filledMassive(count, modulo,):
    arr = []
    l=0
    while(l<count):
        arr.append(randint(0, modulo))
        l=l+1
    return arr

def Poland(a: Mod, b: Mod, p: Integer) -> Integer:
    s = 500

    m = p - 1
    k0 = Mod(ZZ.random_element(m, distribution='uniform'), m)
```

```
    y = z = (a ** k0) % p
    Ay = Az = Mod(k0, m)
    By = Bz = Mod(0, m)
    x = Mod(0, m)
    i = j = 0
    alphas = filledMassive(s, m)
    betas = filledMassive(s, m)

    isStart = True
    while isStart or z != y:
        isStart = False
        z = f(z, a, b, s, alphas, betas)
        i = lift(z) % s
        Az += alphas[i]
        Bz += betas[i]

        y = f(y, a, b, s, alphas, betas)
        i = lift(y) % s
        y = f(y, a, b, s, alphas, betas)
        j = lift(y) % s
        Ay += alphas[i] + alphas[j]
        By += betas[i] + betas[j]

    Adif = lift(Az - Ay)
    Bdif = lift(By - Bz)

    GCD = gcd(Bdif, m)
    if GCD > 1:
        if Adif % GCD != 0:
            x = Poland(a, b, p)
            return x
        else:
            Adif = Adif // GCD
            Bdif = Bdif // GCD
            m = m // GCD

    Bdif = inverse_mod(Bdif, m)
    x = Mod(Adif * Bdif, m * GCD)
    x -= m
    for i in range(GCD):
        x += m
        if a ** x == b:
            return x, opCount
    x = Poland(a, b, p)
    return x

sptime = []
import time
xArr = []
lArr = []
for jj in range(30):
    print(jj)
    p = next_prime(2 ** jj)
    a = Mod(ZZ.random_element(p), p)
    b = a ** ZZ.random_element(p - 1)

    t1 = time.time()
```

```
    x = Poland(a, b, p)

    t2 = time.time()
    sptime.append((t2 - t1))
    xArr.append(x)
    if b==1:
        lArr.append(-1)
    else:
        lArr.append(log(lift(b), lift(a)))
```

Ниже приведен график, сравнения теоретического и экспериментального логарифма. Синий – Теоретический, Оранжевый экспериментальный. Для проверки использовалась функция math.log(b,a)

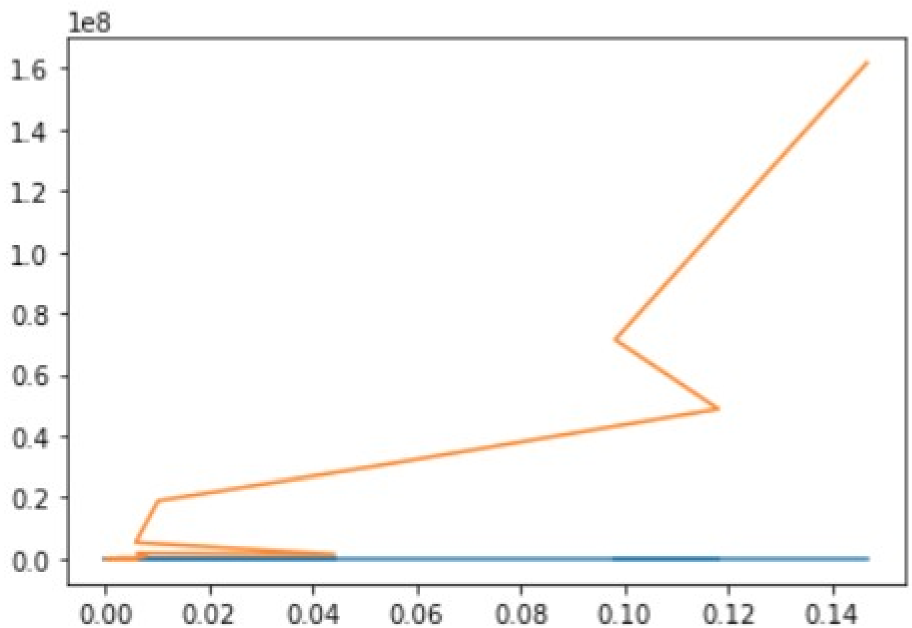


*Рисунок 1Сравнение Экспериментального и Теоретического Логарифма.*

:

## Список литературы


Shi Bai, R. P. On the Efficiency of Pollard's Rho Method for Discrete Logarithms.
Teske, E. SPEEDING UP POLLARD'S RHO METHOD FOR COMPUTING.
А.Ю.Нестеренко. Теоретико-числовые методы в крипографии.