

Министерство образования и науки Российской Федерации

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

ОТЧЕТ К РАБОТЕ  
«ро-метод 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 алгоритмы, описанные в статье [2], выполняющие получение дискретного алгоритма числа  $b$  по основанию  $a$  и по модулю  $m$  на основе ро-метода Полларда. Результат оформить в виде отчета.

### Использованные инструменты:

В работе были использованы SageMath, JupyterNotebook

### Теоретическая база:

В статье [2] рассматриваются четыре вариации алгоритма Полларда:

1. Оригинальный ро-Алгоритм
2. Модифицированный ро-Алгоритм
3. Линейный ро-Алгоритм с 20 множителями
4. Объединенный алгоритм с 16 множителями и 4 квадратами

#### 1. Оригинальный ро-Алгоритм

Рассматриваются последовательность пар  $\{u_i, v_i\}$  целых чисел по модулю  $p - 1$  и последовательность  $\{z_i\}$  целых чисел по модулю  $p$ , определенные следующим образом:

$$\{u_i\}, \{v_i\}, \{z_i\}, \quad i \in N,$$

$$u_0 = v_0 = 0, \quad z_0 = 1;$$

$$u_{i+1} = \begin{cases} u_i + 1 \bmod (p - 1), & 0 < z_i < \frac{p}{3}; \\ 2u_i \bmod (p - 1), & \frac{p}{3} < z_i < \frac{2}{3}p; \\ u_i \bmod (p - 1), & \frac{2}{3}p < z_i < p; \end{cases}$$

$$v_{i+1} = \begin{cases} v_i \bmod (p - 1), & 0 < z_i < \frac{p}{3}; \\ 2v_i \bmod (p - 1), & \frac{p}{3} < z_i < \frac{2}{3}p; \\ v_i + 1 \bmod (p - 1), & \frac{2}{3}p < z_i < p; \end{cases}$$

$$z_{i+1} \equiv b^{u_{i+1}} a^{v_{i+1}} \pmod{p} = \begin{cases} bz_i \bmod p, & 0 < z_i < \frac{p}{3}; \\ z_i^2 \bmod p, & \frac{p}{3} < z_i < \frac{2}{3}p; \\ az_i \bmod p, & \frac{2}{3}p < z_i < p; \end{cases}$$

Поскольку каждая треть отрезка, которой принадлежит элемент, вероятно, никак не связана с элементами последовательностей  $\{u_i, v_i\}$ , полученная последовательность — псевдослучайная. Поэтому могут существовать такие числа  $j$  и  $k$ , что  $z_k = z_j$ . Если удастся найти такую пару чисел, то получится:

$$b^{u_j} a^{v_j} \equiv b^{u_k} a^{v_k} \pmod{p}.$$

Если число  $u_j - u_k$  взаимно простое с числом  $p - 1$ , то это сравнение можно решить и найти дискретный логарифм:

$$b^{u_j - u_k} \equiv a^{v_k - v_j} \pmod{p}.$$

$$x \equiv \log_a b \equiv (u_j - u_k)^{-1}(v_k - v_j) \pmod{p - 1}.$$

Если же наибольший общий делитель чисел  $u_j - u_k$  и  $p - 1$  равен числу  $d > 1$ , то существует решение этого сравнения для  $x$  по модулю  $(p - 1)/d$ . Пусть  $x = x_0 \pmod{(p - 1)/d}$ , тогда искомое число  $x = x_0 + m(p - 1)/d$ , где  $m$  может принимать значения  $0, 1, \dots, d - 1$ . Поэтому если  $d$  — достаточно небольшое число, то задача решается перебором всех возможных значений для  $m$ . В худшем случае — когда  $d = p - 1$  — метод оказывается ничем не лучше полного перебора всех возможных значений для дискретного логарифма.

Для поиска индексов  $j$  и  $k$  используется алгоритм поиска циклов Флойда. При использовании данного алгоритма на  $i$ -м шаге имеются значения  $(z_i, u_i, v_i, z_{2i}, u_{2i}, v_{2i})$  и ищется номер  $i$ , для которого  $z_i = z_{2i}$ . Если при этом  $(u_{2i} - u_i, p - 1) = 1$ , то

$$x \equiv \log_a b \equiv (u_{2i} - u_i)^{-1}(v_i - v_{2i}) \pmod{p - 1}.$$

## 2. Модифицированный ро-Алгоритм

Как и в исходной прогулке Полларда, мы используем разбиение  $G$  на 3 части. Но теперь  $m, n \in_R \llbracket 1, |G| \rrbracket$

и пусть  $M = g^m, N = h^n$ . Теперь определим  $f_{Pm} : G \rightarrow G$ ,

$$f_{Pm}(y) = \begin{cases} M * y, & y \in T_1, \\ y^2, & y \in T_2, \\ N * y, & y \in T_3. \end{cases}$$

для  $\alpha_i u \beta_i$  это значит

$$\begin{aligned} \alpha_{i+1} &\equiv \alpha_i + m, & \alpha_{i+1} &\equiv 2\alpha_i, & \text{or } \alpha_{i+1} &= \alpha_i \pmod{|G|}, \\ \beta_{i+1} &\equiv \beta_i, & \beta_{i+1} &\equiv 2\beta_i, & \text{or } \beta_{i+1} &\equiv \beta_i + n \pmod{|G|}, \end{aligned}$$

### 3. Линейный ро-Алгоритм с 20 множителями

Мы используем разбиение  $G$  на 20 частей. Пусть  $m_1, n_1, \dots, m_{20}, n_{20} \in_R \llbracket 1, |G| \rrbracket$

$$M_s = g^{m_s} * h^{n_s}, \quad s = 1, \dots, 20$$

Определим  $f_T = G \rightarrow G$

$$f_T(y) = M_s * y, \quad \text{with } s = s(y) \text{ such that } y \in T_s$$

$$\alpha_{i+1} \equiv \alpha_i + m_s \quad \text{and} \quad \beta_{i+1} \equiv \beta_i + n_s \pmod{|G|}$$

### 4. Объединенный алгоритм с 16 множителями и 4 квадратами

Мы используем разбиение  $G$  на 20 частей. Выбираем 4 попарно различных значения между 1 и 20.

$$m_1, n_1, \dots, m_{20}, n_{20} \in_R \llbracket 1, |G| \rrbracket$$

$$M_s = g^{m_s} * h^{n_s}, \quad s \in \{1, \dots, 20\} \setminus \{u_1, u_2, u_3, u_4\}.$$

Определим  $f_C = G \rightarrow G$

$$f_C(y) = \begin{cases} M_s * y, & \text{Если } s \neq u_1, u_2, u_3, u_4 \\ y^2, & \text{В остальных случаях} \end{cases}$$

$$\begin{aligned} \alpha_{i+1} &\equiv \alpha_i + m_s \quad \text{or} \quad \alpha_{i+1} \equiv 2 * \alpha_i \pmod{|G|}, \\ \beta_{i+1} &\equiv \beta_i + n_s \quad \text{or} \quad \beta_{i+1} \equiv 2 * \beta_i \pmod{|G|}, \end{aligned}$$

### Результаты выполнения работы:

```
def ext_euclid_algor(a, b):
    # алгоритм Евклида используется для обратного вычисления
    if b == 0:
        return a, 1, 0
    else:
        d, xx, yy = ext_euclid_algor(b, a % b)
        x = yy
        y = xx - (a // b) * yy
        return d, x, y

def inverse(a, n):
    #инверсия a по mod n
    return int(ext_euclid_algor(a, n)[1])

def step1(x, a, b, tuple):
    #Шаг Ро-алгоритма Полланда
```

```

G, H, P, Q = tuple[0], tuple[1], tuple[2], tuple[3]

sub = lift(x) % 3 # Subsets

if sub == 0:
    x = x*G % P
    a = (a+1) % Q

if sub == 1:
    x = x * H % P
    b = (b + 1) % Q

if sub == 2:
    x = x*x % P
    a = a*2 % Q
    b = b*2 % Q
return x, a, b

def step2(x, a, b, tuple):
    #Шаг модифицированного Ро-алгоритма Полланда
    G, H, P, Q = tuple[0], tuple[1], tuple[2], tuple[3]

    l = np.random.randint(1, G)
    k = np.random.randint(1, G)
    L = pow(G, l)
    K = pow(H, k)

    sub = lift(x) % 3 # Subsets

    if sub == 0:
        x = x*L % P
        a = (a+l) % Q
        b=b

    if sub == 1:
        x = x * K % P
        b = (b + l) % Q
        a = a % Q

    if sub == 2:
        x = x*x % P
        a = a*2 % Q
        b = b*2 % Q
    return x, a, b

def step3(x, a, b, tuple):
    #Линейный шаг
    G, H, P, Q = tuple[0], tuple[1], tuple[2], tuple[3]
    l = []
    k = []
    i = 0
    while i<20:
        l.append(np.random.randint(1, G))
        k.append(np.random.randint(1, G))
        i+=1

    i = 0

```

```

sub = lift(x) % 20 # Subsets

L = pow(G, l[sub]) * pow(H, k[sub])
x = x*L % P
a = (a + l[sub]) % Q
b = (b + k[sub]) % Q

return x, a, b

def step4(x, a, b, tuple):
    #Комбинаторный шаг
    G, H, P, Q = tuple[0], tuple[1], tuple[2], tuple[3]
    l = []
    k = []
    i = 0
    while i<20:
        l.append(np.random.randint(1, G))
        k.append(np.random.randint(1, G))
        i+=1
    u1 = -1
    u2 = -1
    u3 = -1
    u4 = -1
    while u1 == u2 or u1 == u3 or u1 == u4 or u2 == u3 or u2 == u4 or u3 ==
u4:
        u1 = np.random.randint(1, G)
        u2 = np.random.randint(1, G)
        u3 = np.random.randint(1, G)
        u4 = np.random.randint(1, G)

    i = 0
    sub = lift(x) % 20 # Subsets
    if sub != u1 or sub != u2 or sub != u3 or sub != u4:
        L = pow(G, l[sub]) * pow(H, k[sub])
        x = x*L % P
        a = (a + l[sub]) % Q
        b = (b + k[sub]) % Q
    else:
        x = x * x
        a = (2 * a) % Q
        b = (2 * b) % Q

    return x, a, b

def pollard1(G, H, P):
    #Оригинальный алгоритм
    opCount = 0

    Q = int((P - 1) // 2) # подгруппа
    x = G*H
    a = 1
    b = 1

    X = x
    A = a
    B = b

```

```

for i in range(1, P):
    #Ёж
    x, a, b = step1(x, a, b, (G, H, P, Q))
    opCount += 1
    #Заяц
    X, A, B = step1(X, A, B, (G, H, P, Q))
    opCount += 1
    X, A, B = step1(X, A, B, (G, H, P, Q))
    opCount += 1

    if x == X:
        break

nom = a-A
denom = B-b

res = (inverse(denom, Q) * nom) % Q

if verify(G, H, P, res):
    return res, opCount

return res + Q, opCount

def pollard2(G, H, P):
    #Модифицированный алгоритм
    opCount = 0

    Q = int((P - 1) // 2)
    x = G*H
    a = 1
    b = 1

    X = x
    A = a
    B = b

    for i in range(1, P):
        #Ёж
        x, a, b = step2(x, a, b, (G, H, P, Q))
        opCount += 1
        #Заяц
        X, A, B = step2(X, A, B, (G, H, P, Q))
        opCount += 1
        X, A, B = step2(X, A, B, (G, H, P, Q))
        opCount += 1

        if x == X:
            break

    nom = a-A
    denom = B-b

    res = (inverse(denom, Q) * nom) % Q

    if verify(G, H, P, res):
        return res, opCount

    return res + Q, opCount

```

```

def pollard3(G, H, P):
    #Линейный алгоритм
    opCount = 0

    Q = int((P - 1) // 2) # подгруппа
    x = G*H
    a = 1
    b = 1

    X = x
    A = a
    B = b

    for i in range(1, P):
        #Ёж
        x, a, b = step3(x, a, b, (G, H, P, Q))
        opCount += 1
        #Заяц
        X, A, B = step3(X, A, B, (G, H, P, Q))
        opCount += 1
        X, A, B = step3(X, A, B, (G, H, P, Q))
        opCount += 1

        if x == X:
            break

    nom = a-A
    denom = B-b

    res = (inverse(denom, Q) * nom) % Q

    if verify(G, H, P, res):
        return res, opCount

    return res + Q, opCount

def pollard4(G, H, P):
    #Объединенный алгоритм с 16 множителями и 4 квадратами
    opCount = 0

    Q = int((P - 1) // 2) # подгруппа
    x = G*H
    a = 1
    b = 1

    X = x
    A = a
    B = b

    for i in range(1, P):
        #Ёж
        x, a, b = step4(x, a, b, (G, H, P, Q))
        opCount += 1
        #Заяц
        X, A, B = step4(X, A, B, (G, H, P, Q))
        opCount += 1
        X, A, B = step4(X, A, B, (G, H, P, Q))
        opCount += 1

```



```

        if x == X:
            break

    nom = a-A
    denom = B-b

    res = (inverse(denom, Q) * nom) % Q

    if verify(G, H, P, res):
        return res, opCount

    return res + Q, opCount

def verify(g, h, p, x):
    #Проверяет заданный набор g, h, p и x
    return pow(g, int(x), p) == h

import time
jj=5
sptime = [[],[],[],[]]
spoperation = [[],[],[],[]]

while jj < 20:
    print(jj)
    M = next_prime(2 ** jj)
    A = Mod(ZZ.random_element(M), M)
    B = A ** ZZ.random_element(M - 1)

    t1 = time.time()

    res, opCount = pollard1(A, B, M)

    t2 = time.time()
    sptime[0].append((t2 - t1))
    spoperation[0].append(opCount)
    t1 = time.time()

    res, opCount = pollard2(A, B, M)

    t2 = time.time()
    sptime[1].append((t2 - t1))
    spoperation[1].append(opCount)
    t1 = time.time()

    res, opCount = pollard3(A, B, M)

    t2 = time.time()
    sptime[2].append((t2 - t1))
    spoperation[2].append(opCount)
    t1 = time.time()

    res, opCount = pollard4(A, B, M)

    t2 = time.time()
    sptime[3].append((t2 - t1))
    spoperation[3].append(opCount)
    jj+=1
    # print(x)

```

```
# print('Time elapsed for 2 in',jj,'=', (t2 - t1))
# print('Operations invoked:', operationCount)
print(sptime)
print('Operations invoked:', opCount)
```

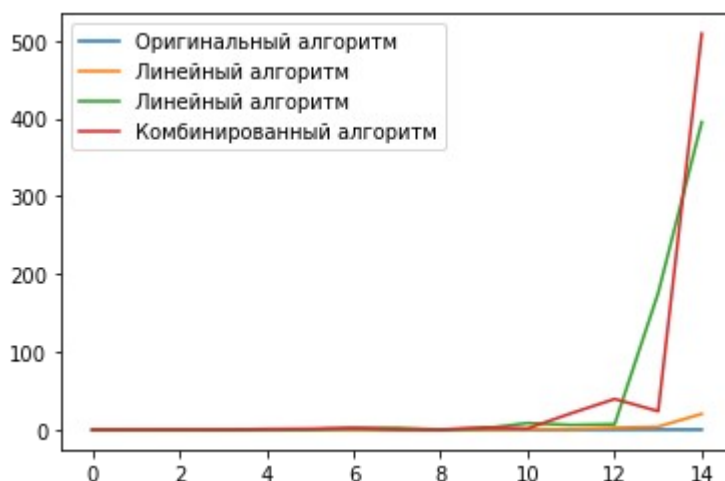
## Время выполнения

```
1 print(sptime)
[[0.00034689903259277344, 9.822845458984375e-05, 0.00042176246643066406, 0.00011801719665527344, 0.000460863113403
3203, 0.0006537437438964844, 0.0010986328125, 0.0004684925079345703, 0.0005865097045898438, 0.00199127197265625,
0.0045621395111083984, 0.006134986877441406, 0.0026373863220214844, 0.003795146942138672, 0.014660358428955078],
[0.007293224334716797, 0.0007696151733398438, 0.010759592056274414, 0.03950023651123047, 0.04629397392272949, 0.08
526086807250977, 0.01108551025390625, 0.012731313705444336, 0.029937744140625, 0.021947860717773438, 0.51121997833
25195, 0.19187498092651367, 2.5103070735931396, 3.786053419113159, 20.3322012424469], [0.04786944389343262, 0.0082
58819580078125, 0.04816174507141113, 0.0020949840545654297, 0.5705082416534424, 0.323472261428833, 1.9187703132629
395, 2.3329644203186035, 0.06703305244445801, 1.5301806926727295, 8.436009407043457, 6.051910400390625, 6.96676611
9003296, 175.67547011375427, 395.18510007858276], [0.036438941955566406, 0.02779555320739746, 0.14207172393798828,
0.018241167068481445, 0.6303024291992188, 1.1482038497924805, 2.1006581783294678, 0.18091320991516113, 0.039686918
25866699, 2.90712833404541, 1.5606694221496582, 21.000721216201782, 39.42337894439697, 23.718202829360962, 508.948
4143257141]]
```

## Количество операций

```
1 print(soperation)
[[24, 15, 84, 12, 42, 123, 207, 87, 108, 384, 882, 1152, 504, 738, 3492], [105, 27, 390, 768, 876, 3090, 168, 459,
1104, 804, 17238, 6432, 82647, 131940, 890046], [108, 24, 132, 3, 1560, 906, 4830, 5643, 192, 4302, 23238, 15606,
19185, 519153, 1348395], [90, 69, 333, 24, 1560, 2877, 4986, 405, 102, 7464, 3816, 49290, 102765, 71676, 1572924]]
```

## График времени работы



## Список литературы

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