Московский авиационный институт
(государственный технический университет)
Факультет прикладной математики и физики
Кафедра вычислительной математики и программирования

Лабораторная работа №2

по спецкурсу «Криптография»: Дискретное логарифмирование

Выполнил: Карпова В.А.
Группа: 08-306
№ по списку: 9
Преподаватель: Рисенберг Д.В.
Оценка:
Дата:

Москва
2012 г.

Задание

Необходимо написать программу на языке C++, C# или Python, реализующую алгоритм дискретного логарифмирования. Должны поддерживаться числа длиннее 64 бит.

Вариант №2: Алгоритм Полига-Хеллмана.

Задача дискретного логарифмирования.

Алгоритм Полига-Хеллмана относится к алгоритму дискретного логарифмирования. Основная задача дискретного логарифмирования состоит в том, чтобы решить уравнение $a^x = b$, где a (порядок группы) и b -- элементы мультипликативной абелевой группы G (Группа — непустое множество с определённой на нём ассоциативной бинарной операцией с нейтральным элементом, при этом для каждого элемента множества должен существовать обратный. Группа, в которой любые два элемента коммутируют, называется коммутативной или абелевой.)

Для криптографии особо важен случай, когда в качестве группы рассматривается группа обратимых по умножению элементов кольца (Z/pZ)*, где p - простое число. В соответствии с этим и рассматривается уравнение:

$a^x \equiv b(mod\ p)$.

При предположении, что порядок группы равен p-1, уравнение разрешимо и x является элементом поля из p-1 элементов Z/pZ.

Алгоритм Полига-Хеллмана.

Предположим, что известно разложение p-1 на простые множители: $\prod_{i=1}^{s} q_i^{\alpha_i}$

1. На 1 шаге мы для каждого простого числа q_i составляем таблицу чисел

$$r_{q,j} \equiv a^{\frac{j(p-1)}{q}}(mod\ p), j = 0, \dots, q-1.$$

Заметим, что таблица расширяется в зависимости от числа q_i.

2. Для каждого простого q_i находим логарифм $\log_a b(mod\ q^\alpha)$.

Те логарифмы, которые мы ищем для q_i, могут быть записаны в виде:

$$x \equiv \log_a b(mod\ q^\alpha) \equiv x_0 + x_1 q + \cdots + x_{\alpha-1} q^{\alpha-1}(mod\ q^\alpha), 0 \le x_i \le q-1.$$

Но, для того, чтобы найти x_0, нам потребуется таблица, которую мы строили на предыдущем шаге.

Т.к. $b \equiv a^{-x}(mod\ p)$, то возведя в степень (p-1)/q, получаем:

$$b^{(p-1)/q} \equiv a^{\frac{x_0(p-1)}{q}}(mod\ p)$$

$x_0 = j$ из таблицы 1, построенной на предыдущем шаге, где j -- позиция соответсвующего элемента в списке $r_{q,j}$.

Из предыдущего соотношения получаем сравнение:

$$(ba^{-x_0})^{\frac{p-1}{q^2}} \equiv a^{\frac{x_0(p-1)}{q}}(mod\ p)$$

Заметим, что в данном случае мы левую часть помножили на обратный элемент по отношению к рассматриваемой группе. Помним, что для нахождения обратного элемента в группе, применяем обобщенный алгоритм Евклида.

Из таблицы 1 можем найти $x_1 = j$.

Тогда будет верно построение по индукции, и для значения x_i будет выполняться сравнение:

$$\left(ba^{-x_0 - x_1 q - \cdots - x_{i-1} q^{i-1}}\right)^{\frac{p-1}{q^{i+1}}} \equiv a^{\frac{x_i(p-1)}{q}}(mod\ p)$$

Итог: нашли все необходимые коэффициенты для получения $\log_a b\left(mod\ q_i^{\alpha_i}\right), i = 1, \dots, s$.

3. Найдя логарифмы для каждогоq_i, ищем решение поставленной задачи, а именно: находим

$$\log_a b(mod\ p-1)$$

Для этого используем китайскую теорему об остатках.

Формулировка:

Рассматривается система сравнений:

$$\begin{cases} x \equiv a_1 (mod\ m_1), \\ \dots \\ x \equiv a_k (mod\ m_k) \end{cases}$$

Переходя на обозначения, в соответствии с предыдущими шагами, получаем расшифровку для каждой переменной:

x - искомое решение.

m_i- делители числа p-1 (т.е. $m_i = q_i^{\alpha_i}$)

M - порядок рассматриваемой группы. M = p-1

$M_i = \frac{M}{m_i}$

a_i- найденные на 2м шаге логарифмыx_i.

b_i- решения соотношения $M_i b_i \equiv 1 (mod\ m_i), i = 1, \dots, k.$ где b_i - обратный элемент M_i в группе порядкаm_i.

Для вычисления обратного элемента во 2м и 3ем шаге использовался обобщенный алгоритм Евклида.

Оценка сложности.

Алгоритм Полига-Хеллмана работает за O($\sum_{i=1}^{s} \alpha_i (logp + q_i)$). Докажем это.

1) Набор элементов $r_{q,j}$ для таблицы из 1го шага вычисляется за $\sum_{i=1}^{s} O(log\ p)$.

2) Все такие наборы для каждого q_iвычисляются за $\sum_{i=1}^{s} O(q_i)$.

3) Затраты на нахождение x_iдля каждогоx_i:

- возведение в степень O(logp)
- нахождение обратного элемента (Обобщенный алгоритм Евклида) - O(logp)

Все вместе для разных q_i и дает оценку O($\sum_{i=1}^{s} \alpha_i (logp + q_i)$).

Реализация.

Из функции main() считываются необходимые данные a, b, p и разложение q_i, i=1,...

Вся работа по нахождению дискретного логарифма происходит в функции dlog.

```
def dlog(a,b,p,q):
    q_al = get_alpha(q)
#создаем словарь q_al, в котором ключ -- q_i, значение alpha_i
    r = get_table(a,p,q)
#создаем словарь r, в котором ключ -- q_i, значение -- список из r_ij в
количестве 0..q_i-1 (1 шаг алгоритма)
    coeff = get_china_coeff(a,b,p,r,q,q_al)
# создаем словарь, в котором ключ q_i^alpha_i, значение -- соответсвующий
логарифм x_i найденный во2 мшаге алгоритма.
    x = china(coeff,p)
# решение системы сравнений -- решение китайской теоремы об остатках

    return x

# Выполняем 2ой шаг алгоритма
def get_china_coeff(a,b,p,r,q,q_al):
    x_f = {}
    for qi in q:
        x = [] # список коэффициентов для нахождения логарифма log_a b
(mod qi^alpha_i)
        c = pow(b,(p-1)//qi,p)
        x.append(r[qi].index(c)) # рассчитали x0 и добавили в список x
        pow_sum = 0
        for i in range(1,q_al[qi]): # проходим по все alpha_i у данного qi
            pow_sum += x[i-1]*(qi**(i-1)) # подсчитываем степень
```

```
                  c = pow(b*get_inv((a**pow_sum),p),(p-1)//(qi**(i+1)), p)
#считаем элемент, значение которого будем сравнивать с таблицей, построенной
на 1 шаге.
                  x.append(r[qi].index(c)) #находим индекс j==xi
            print ("x = ",x)
            x_f[qi**q_al[qi]] = sum([xi*qi**i for i,xi in enumerate(x) ])
#находим логарифм для соответсвующего qi^alphai

      return x_f

 #Находим обратный элемент в группе через обобщенный алгоритм Евклида.
 def get_inv(x,y):
      d, ax, ay = eGCD(x,y)
      if d<=1: return ax
      print("Error")
      exit(-1)

 #Считаем НОД и возвращаем коэффициенты, т.ч. x*a+y*b = nod(a,b)
 def eGCD(a, b):
      if b == 0:
            return a , 1, 0
      d1, x1, y1 = eGCD(b, a%b )
      d, x, y = d1, y1, x1 - (a//b)*y1
      return d, x, y
```

Выводы.

Задача дискретного логарифмирования является одной из основных задач, на которых базируется криптография с открытым ключом. Их криптостойкость основывается на предположительно высокой вычислительной сложности обращения показательной функции. Показательная функция вычисляется достаточно эффективно, в то время как даже самые современные алгоритмы вычисления дискретного логарифма имеют очень высокую сложность, которая сравнима со сложностью наиболее быстрых алгоритмов разложения чисел на множители.

Другая возможность эффективного решения задачи вычисления дискретного логарифма связана с квантовыми вычислениями.