

实验六 DLP问题和DH密钥交换协议

17341125 蒙亚愿 信息安全

一、实验目的

了解DLP问题的挑战性和DH密钥交换协议，计算位数较小的离散对数，解决DLP问题，并获取DH密钥交换协议中的共同信息。

二、实验要求

设 p 是一个素数， g 是模 p 剩余类群中的非零元素。已知模数 p ， g 和 y ，求解 x 满足同余方程 $y \equiv g^x \pmod{p}$ 的计算数论问题，称为模 p 剩余类域中离散对数求解问题。DH（Diffie-Hellman）密钥交换协议是 20 世纪 70 年代由 Whitfield Diffie 和 Martin Hellman 共同提出的，在网络安全中有着广泛的应用，其目的是让参与双方能够共享一个秘密信息（密钥）。DH 密钥交换协议的过程如下图：

A 发起方	B 应答方
产生 x_a 并计算 $y_a \equiv g^{x_a} \pmod{p}$	产生 x_b 并计算 $y_b \equiv g^{x_b} \pmod{p}$
A 使用公共信道把 y_a 发送给 B	B 使用公共信道把 y_b 发送给 A
接收到 y_b 并计算 $y_b^{x_a} \pmod{p}$	接收到 y_a 并计算 $y_a^{x_b} \pmod{p}$
此时，A 有信息 $y_b^{x_a} \equiv g^{x_b x_a} \pmod{p}$ ，B 有信息 $y_a^{x_b} \equiv g^{x_a x_b} \pmod{p}$ ，而 $g^{x_a x_b} \equiv g^{x_b x_a} \pmod{p}。$ 这样，A 和 B 就得到了共同的信息。	

DH 协议的安全性基于素域（模素数 p 剩余类域）上离散对数求解的困难性。如果第三方可以在公共信道截获 y_a 和 y_b ，通过求取离散对数 x_a ，就可求出 A 和 B 的共同信息

$$y_b^{x_a} \equiv g^{x_b x_a} \pmod{p}$$

三、实验介绍

3.1 实验综述

已知一个乘法群 (Z_p^*, \cdot) ， $g \in Z_p^*$ 是一个 ord_g 阶的元素，在信道中截取到A发给B的 y_a 和B发给A的 y_b ，通过Shanks算法计算出唯一的指数 x_a 和 x_b ， $0 \leq x_a \leq \text{ord}_g$ ，

$0 \leq x_b \leq \text{ord}_g$ ，使得

$$\begin{aligned} g^{x_a} &= y_a \pmod{p} \\ g^{x_b} &= y_b \pmod{p} \end{aligned}$$

A、B双方得到对方的 y 之后，就用自己的信息和对方的信息结合起来，计算共同信息，而不需要知道对方的离散对数。

$$A: \text{ same} = y_b^{x_a} \bmod p = g^{x_b * x_a} \bmod p$$

$$B: \text{ same} = y_a^{x_b} \bmod p = g^{x_a * x_b} \bmod p$$

3.2 开发环境

运行环境：Windows

使用语言：C++

编译器：CodeBlocks

四、算法原理以及代码实现

题中已经给出了 g 和 p ，首先需要计算元素 g 的阶。通过网站计算得到

$$\text{ord}_g = 4309874666$$

接着就可以根据书中的伪代码实现计算出离散对数：

算法 6.1 SHANKS (G, n, α, β)

1. $m \leftarrow \lceil \sqrt{n} \rceil$

2. **for** $j \leftarrow 0$ **to** $m-1$

do 计算 α^{mj}

3. 对 m 个有序对 (j, α^{mj}) 关于第二个坐标排序，得到一个列表 L_1

4. **for** $i \leftarrow 0$ **to** $m-1$

do 计算 $\beta\alpha^{-i}$

5. 对 m 个有序对 $(i, \beta\alpha^{-i})$ 关于第二个坐标排序，得到一个列表 L_2

6. 找到对 $(j, y) \in L_1$ 和 $(i, y) \in L_2$ （即找到两个具有相同第二坐标的对）

7. $\log_\alpha \beta \leftarrow (mj + i) \bmod n$

其中, $n = \text{ord}_g$, $\alpha = g$, $\beta = y$ 。

4.1 算法原理

按照我的理解，主要是以下的式子：

$$i, j = 0 \dots m-1$$
$$\text{若 } \alpha^{mj} = \beta\alpha^{-i} \bmod p$$

$$\text{等式两边乘 } \alpha^{-i} \text{ 的逆 } \alpha^i$$
$$\alpha^{mj+i} = \beta \bmod p$$

$$\text{则 } \log_\alpha \beta = (mj + i) \bmod p$$

很好理解，就是计算 α^{mj} 和 $\beta\alpha^{-i}$ ，找到一样的就计算 $mj + i$ 即可。

4.2 算法优化

针对其题目的特殊性，考虑到存储空间和运行时间可以更进一步的减少，可以对算法进行稍微的优化：

1. 只需建立一个列表 L_1 存储有序对。

当计算第二组有序对时，可以“边算边查找检验”。也就是在第二个循环当中，如果是在第 i 个有序对发现第二坐标相同，而在第 i 个有序对和离散对数的计算过程中，前 $i - 1$ 个有序对实际上是用不到的，所以没有必要存储。

也就是只需要一个变量 ans 来存储当前第 i 个循环计算出的 $\beta\alpha^{-i}$ ，并且直接去查找列表 L_1 当中有没有有序对的第二坐标与之相同，如果没有，就继续循环，用下一个循环的值更新 ans 。

2. 使用map容器来查找。

在计算出列表 L_1 之后，不需要进行坐标排序。个人认为坐标排序只是为了更快的查找到第二坐标相同的有序对。但是map容器在查找上也能提供一对一的映射，并且能够自己排序，不需要我们亲自排序，会更加的便捷。

特别的，map中存储的有序对的第一个变量（关键字）不是伪代码中的 i ，而是伪代码中的第二坐标，也就是(value, number)，这对于之后的查找会比较方便，而且对于排序什么的也不会有影响，甚至实际上排序并没有什么作用。

3. 只需要建立 L_1 一次。

考虑到题目的特殊性。计算 x_a 之后已经建立好了 L_1 ，再计算 x_b 实际上也是在相同的 L_1 表上查找，所以就不需要再次建立了，毕竟要迭代 $ord_g + 1$ 次，这样可以减少运行时间。

使用布尔变量 $ready$ 标记是否建立了 L_1 ，如果为真，就省去了这一个迭代，否则，就建立 L_1 。

4.3 代码实现

1. Shanks函数

传参：mpz_class y (模幂结果) mpz_class &x (离散对数)

建立 L_1 列表：couple(g^{mj} , j)

```
if(ready==false){
    for(mpz_class j=0; j<m-1; j++){
        ans = j*m;
        mpz_powm(ans.get_mpz_t(), g.get_mpz_t(), ans.get_mpz_t(),
p.get_mpz_t());
        couple[ans] = j ;
    }
    ready = 1 ;
}
```

计算 $\beta\alpha^{-i}$ ，并查找 i ，计算离散对数。

```

for(mpz_class i=0; i<m-1; i++){
    mpz_powm(ans.get_mpz_t(), g.get_mpz_t(), i.get_mpz_t(), p.get_mpz_t()) ;
    mpz_invert(ans.get_mpz_t(), ans.get_mpz_t(), p.get_mpz_t()) ;
    ans = y*ans ;
    mpz_mod(ans.get_mpz_t(), ans.get_mpz_t(), p.get_mpz_t()) ;

    if(couple[ans]){
        mpz_class j = couple[ans] ;
        x = m*j+i ;
        mpz_mod(x.get_mpz_t(), x.get_mpz_t(), ord_g.get_mpz_t()) ;
        return ;
    }
}

```

2. main函数

验证计算出的 x_a 和 x_b 是否是真解。

```

mpz_class ta, tb ;
mpz_powm(ta.get_mpz_t(), g.get_mpz_t(), xa.get_mpz_t(), p.get_mpz_t()) ;
mpz_powm(tb.get_mpz_t(), g.get_mpz_t(), xb.get_mpz_t(), p.get_mpz_t()) ;
if(ta!=ya || tb!=yb){
    cout<<" 实验失败！ ";
    if(ta!=ya)    cout<<xa<<"不是正确解！"<<endl;
    else cout<<xb<<"不是正确解！"<<endl;
    return 1 ;
}

```

计算共同信息并验证：

$$g^{x_a x_b} = g^{x_b x_a} \bmod p$$

```

mpz_class same_a, same_b;
mpz_powm(same_a.get_mpz_t(), yb.get_mpz_t(), xa.get_mpz_t(), p.get_mpz_t()) ;
mpz_powm(same_b.get_mpz_t(), ya.get_mpz_t(), xb.get_mpz_t(), p.get_mpz_t()) ;

if(same_a==same_b)    cout<<"\n 实验成功！\n 共同信息为 " <<same_a<<endl;
else { cout<<"\n 实验失败,共同信息错误！"<<endl; return 1; }

```

五、实验效果及分析

经过计算和验证，结果如下：

xa = 3958420340

xb = 1004913511

共同信息：

**1040257607091539426816569019565155047118333857441840239876578973939486070545132
48743759414513814738665197005346311279644611122147838150193586022949825713770**

只算一次 L_1 列表：

```
"C:\Users\DELL\Desktop\密码学\实验6 LDP\code\for.exe"
p = 5682549022748424631339131913370125786212509227588493537874673173634936008725904358935442101466555561124455782847468
955028529037660533553941399408331331403379
g = 2410497055970432881345493397846112198995088771364307195189734031205605186951241875096796459061741781667380437076874
705300974836586165283741668656930807264789
ya = 973768284341326272301553751114322685324340805902069613339667142187801529585352406975030927008752571917079716318221
077758236884342662829402529734009607531649
yb = 414982276598503114655429877712273205187086843138732391374778579168531050883683628370292644681700011486800755571754
6362425841865173929670156568682745060708314

经过shanks算法得到
xa is 3958420340
xb is 1004913511

经验证，计算正确！

验证通过！实验成功！
共同信息为 104025760709153942681656901956515504711833385744184023987657897393948607054513248743759414513814738665197005
346311279644611122147838150193586022949825713770

Process returned 0 (0x0)   execution time : 5.780 s
Press any key to continue.
```

算两次 L_1 列表：

```
"C:\Users\DELL\Desktop\密码学\实验6 LDP\code\for.exe"
p = 5682549022748424631339131913370125786212509227588493537874673173634936008725904358935442101466555561124455782847468
955028529037660533553941399408331331403379
g = 2410497055970432881345493397846112198995088771364307195189734031205605186951241875096796459061741781667380437076874
705300974836586165283741668656930807264789
ya = 973768284341326272301553751114322685324340805902069613339667142187801529585352406975030927008752571917079716318221
077758236884342662829402529734009607531649
yb = 414982276598503114655429877712273205187086843138732391374778579168531050883683628370292644681700011486800755571754
6362425841865173929670156568682745060708314

经过shanks算法得到
xa is 3958420340
xb is 1004913511

经验证，计算正确！

验证通过！实验成功！
共同信息为 104025760709153942681656901956515504711833385744184023987657897393948607054513248743759414513814738665197005
346311279644611122147838150193586022949825713770

Process returned 0 (0x0)   execution time : 8.764 s
Press any key to continue.
```

显然，运行时间短了。而关于排序时间、变量存储的优化效果，显然是成功的。

在计算version 1和version 2的过程中，发现如果使用 $Shanks$ 算法，内存是不够的，version 2需要循环34728980674次，在第14283662次内存就不够了，并且跑了快3h，所以， $Shanks$ 算法适用于数比较小的DLP问题，如果要计算出更大的数，我想似乎会有更好的方法。

```
j is 14283659
j is 14283660
j is 14283661
j is 14283662
GNU MP: Cannot reallocate memory (old_size=4 new_size=12)

Process returned 3 (0x3)   execution time : 9705.255 s
Press any key to continue.
```

六、总结与思考

1. 这次实验我几乎尝试了所有方法。首先是 $Pollard\ \rho$ 离散对数算法，因为比较看得懂，但是实现之后发现数组不能创建足够大的，而且我也没有很好的解决方法。然后尝试 $Pohlig - Hellman$ 算

法，这个算法真的太费劲了，看了半天没看懂原理，看例子加上伪代码才实现出来的，不过只能通过小一点的数的测试，这次实验的数据要跑好久，两天都不出来。最后看了 $Shanks$ 算法，原以为第一个算法都是最不推荐的，最后没想到是最好的，不到一小时就写完了，跑得很快，很好理解，优化一下算得很快，不需要一板一眼地按照伪代码实现，后来想想似乎 $BSGS$ 算法其实跟这个差不多哎，不知道为什么推荐emmm。害，打了这么多算法还是有点收获的，最起码知道在小一点的数上，这些算法是咋算的，也顺便复习了MR素性检验，不亏！

2. 数据给了这么多，其实只有最后一个快，其他的真的算的好慢好慢，最开始还没有阶，想到要自己算真的都不敢写了，耽误了好多天不知道阶怎么办，下次希望TA能够一次性把数据都说清楚。
3. 不过也通过这么多数据，感受到了DLP问题的挑战性，对于很大的数据真的跑了好久，电脑放着跑两天都不出来。这次试验也比较简单，不需要很绕的算法，就是一直算一直找碰撞，最后的DH密钥交换就是模幂运算一下就可以了。收获很大，辛苦老师和TA了。