“密码学课程设计”实验报告

班　　级：信息安全 1701 .

姓　　名：罗南清 .

学　　号：U201714868 .

项目	平时成绩	实验过程	实验报告	总分	教师签字
分值	10	60	30	100	
评分					

目 录

任务书

课题内容：

（1）原始 SPN（教材上）算法的实现。

（2）对上述算法进行线性密码分析及差分密码分析（求出所有 32 比特密钥）。

（3）增强以上 SPN 的安全性（如增加分组的长度、密钥的长度、S 盒、轮数等）。

（4）对原始及增强的 SPN 进行随机性检测，对检测结果进行说明。

（5）生成 RSA 算法的参数（如 p、q、N、私钥、公钥等）。

（6）快速实现 RSA（对比模重复平方、蒙哥马利算法和中国剩余定理）。

（7）利用椭圆曲线密码算法、HASH 函数、压缩函数、对称加密算法实现一个类似 PGP 的文件加解密及完整性校验功能。

（8）构造彩虹表破解 hash 函数。

课题任务要求：

（1）掌握线性、差分分析的基本原理与方法。

（2）体会位运算、预计算在算法快速实现中的作用。

（3）可借助 OpenSSL、GMP、BIGINT 等大数运算库的低层基本函数，实现过程中必须体现模重复平方、中国剩余定理和蒙哥马利算法的过程。内容(7)的算法可以直接调用 OpenSSL 或者其它密码库。

（4）了解和掌握彩虹表构造的基本原理和方法，能够设计和实现约化函数（reduction function），针对特定的 hash 函数构造彩虹表，进行口令破解。

（5）独立完成课程设计内容，现场演示并讲解。

（6）课程设计完成后一周内，提交课程设计报告。

主要参考文献（由指导教师选定）

（1）密码学原理与实践（第三版）. DouglasR.Stinson 著，冯登国译，电子工业出版社，2009

（2）应用密码学：协议算法与 C 源程序（第二版）. Bruce Schneier 著，吴世忠等译，机械工业出版社，2014

同组设计者 无

1 SPN 实验

1.1 实验内容

1）加密：实现《密码学原理与实践（第三版）》 Page.59 例 3.1 的加密过程。
2）线性分析：通过对明密文对的线性分析，破解出初始加密密钥；
3）差分分析：通过对明密文对的差分分析，破解出初始加密密钥；
4）密钥攻击：求出例 3.1 所有 32 比特密钥；
5）增强以上 SPN 的安全性（如增加分组的长度、密钥的长度、S 盒、轮数等）；
6）运用工具对 SPN 加强生成的密文进行随机性检测，来测试 SPN 的密码强度

1.2 实验原理

1.2.1 迭代密码

迭代密码的核心是一个密钥编排方案和一个轮函数。

密钥编排方案对密钥 K 进行变换，生成Nr个子密钥(也叫轮密钥)，记为$k^1, k^2, \ldots k^{Nr}$

轮函数 g 是一个状态加密函数，以k^i为密钥对当前状态w^{r-1}进行变换，输出新的状态值w^r，即$g(w^{r-1}, K^i) = w^r$; 轮函数是单射函数，存在一个逆变换 g^{-1}，有$g^{-1}(w^r, K^i) = w^{r-1}$

迭代密码的加密为将密钥 k 编排成Nr个轮密钥$k^1, k^2, \ldots k^{Nr}$

将明文 x 定义为初始状态 w^0，经过 Nr 轮变换得到w^{Nr}为密文y，即

$$w^0 = x$$
$$w^1 = g(w^0, K^1)$$
$$w^2 = g(w^1, K^2)$$
$$\ldots\ldots$$
$$w^{Nr-} = g(w^{Nr-}, K^{Nr-})$$
$$w^{Nr} = g(w^{Nr-}, K^{Nr})$$
$$y = w^{Nr}$$

迭代密码的解密为将密文 y 定义为初始状态w^{Nr}，经过 Nr 轮逆变换得到 w^0 为明文 x，即

$$y = w^{Nr}$$
$$w^{Nr-} = g^{-1}(w^{Nr}, K^{Nr})$$
$$w^{Nr-} = g^{-1}(w^{Nr-1}, K^{Nr-})$$
$$\ldots\ldots\ldots$$
$$w^1 = g^{-1}(w^2, K^2)$$
$$w^0 = g^{-1}(w^1, K^1)$$
$$x = w^0$$

1.2.2 代替置换网络

代替-置换网络(Substitution-Permutation Network)是一种简单的迭代密码。处理的明文单元和状态值长度为$L \times m$，轮函数 g 包括两个核心变换——代替和置换，分别记为 π_s 和 π_p，有

$$\pi_s: \{0,1\}^L \rightarrow \{0,1\}^L$$

$$\pi_p: \{1,2,\ldots.,L_m\} \rightarrow \{1,2,\ldots.,L_m\}$$

在进行轮函数变换前，先用轮密钥和状态值进行异或(称为白化)。

1.2.3 SPN 密码体制设计

设 L, m, Nr 是正整数, $P = C = \{0,1\}L_m, K \in \{0,1\}L_m$, Nr+1 是由初始密钥 K 用密钥编排算法生成的所有可能的密钥编排方案集合，一个密钥编排方案记为$k^1, k^2,\ldots\ k^{Nr}$

状态值 w 长度为 L×m，记为$w^1, w^2,\ldots\ w^{Lm}$, w 可以看成 m 个长度为 L 的子串连接而成，记为 $w = w_{<1>}, w_{<2>}, \ldots\ldots w_{<m>}$

加密过程使用如下算法描述: $SPN(x, \pi_s, \pi_p, (k^1, k^2,\ldots\ k^{Nr}))$

$w^0 = x$

for r=1 to Nr-1 {

$u^r = w^{r-1} \oplus k^r$ // 白化

for i=1 to m

$v^r_{<i>} = \pi_s(u^r_{<i>})$ // 代替

$w^r = (v^r_{\pi p(1)}, v^r_{\pi p(2)}, \ldots, v^r_{\pi p(lm)})$ // 置换

}

$u^{Nr} = w^{Nr-1} \oplus k^{Nr}$

for i = 1 to m

$v^{Nr}_{<i>} = \pi_s(u^{Nr}_{<i>})$ // 代替

$y = v^{Nr} \oplus k^{Nr+1}$ // 白化

return y

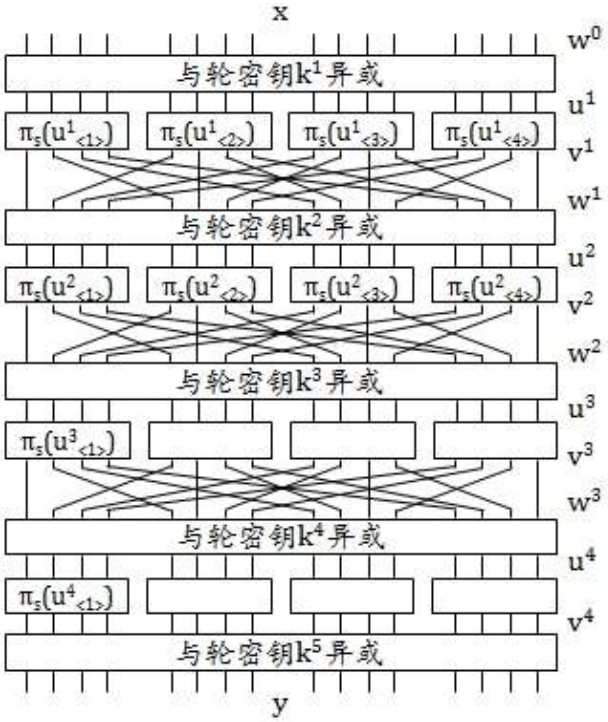

图 1.1 SPN 加密体制示意图

1.2.4 线性密码分析

线性密码分析，是通过分析 S 盒的线性特性，从而发现明文比特、密文比特和密钥比特之间可能存在的线性关系，如果 S 盒设计不当，这种线性关系会以较大的概率存在，称为概率线性关系。

线性密码分析一种已知明文攻击方法，已知 x 和 y，确定 k 或 k 的部分比特。

其中，S 盒的选择对 SPN 的安全性影响巨大，假设一个 S 盒按如下规则设计。

z	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
$\pi_s(z)$	0	2	4	6	8	10	C	E	1	3	5	7	9	B	D	F

图 1.2　S 盒示例。

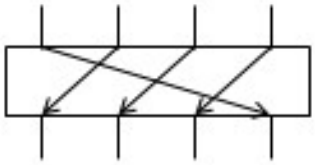

图 1.3　输入输出线性关系示例

这种输入输出关系是一种线性关系。

采用已知明文攻击方法，如果掌握了足够多的明-密文对，即可求出轮密钥 ki，进而根据轮密钥编排方案反向推导出加密密钥 k。线性密码分析思路为找到足够多的明文-密文对，对可能的密钥进行穷举，计算相关随机变量的偏差，正确的密钥作用下，偏差的绝对值最大

不需要对全部密钥空间进行穷举，只需要对随机变量有影响的密钥比特进行穷举，这些密钥比特称为候选子密钥

具体算法如下：线性攻击(T, T, π_s^{-1})

for $(L_1, L_2)=(0, 0)$ *to* $(F, F)\{$ // L_1, L_2 表示候选子密钥 $k^5_{<2>}$ 和 $k^5_{<4>}$

　　$Count[L_1, L_2]=0$ // 每个候选子密钥分配一个计数器并初始化为 0

}

for each $(x, y) \in T\{$

　　for $(L_1, L_2) = (0, 0)$ *to* $(F, F)\{$

　　$v^4_{<2>} = L_1 \oplus y_{<2>}$

　　$v^4_{<4>} = L_2 \oplus y_{<4>}$

　　$u^4_{<2>} = \pi_s^{-1}(v^4_{<2>})$

　　$u^4_{<4>} = \pi_s^{-1}(v^4_{<4>})$

　　$z = x_5 \oplus x_7 \oplus x_8 \oplus u^4_6 \oplus u^4_8 \oplus u^4_{14} \oplus u^4_{16}$ //计算随机变量值

　　if $z=0$

　　$Count[L_1, L_2]$ ++;

}

$max = -1$

for $(L_1, L_2) = (0, 0)$ *to* $(F, F)\{$

$Count[L_1, L_2] = | Count[L_1, L_2] - T/2 |$

if $Count[L_1, L_2] > max\{$

　　$max = Count[L_1, L_2]$

　　$maxkey = (L_1, L_2)$ // maxkey 即为所求子密钥

}

此算法理论上可行，但实际上只是缩小了穷举密钥的范围，算法本身也需要穷举候选密钥，因

此和直接穷举密钥相比，总体上提高的效率有限，除非 S 盒有较大的线性缺陷。

1.2.5 差分密码分析

通过分析明文对的差值对密文对差值的影响来恢复某些密钥比特的分析方法。分析两个输入的异或和两个输出的异或之间的线性关系。构造若干个明文串对，每对明文的异或结果相同，观察相应的密文异或结果。

差分分析是一种选择明文攻击方法，比线性分析更早提出，分析效果略差于线性分析。

仍以“循环左移”S 盒为例：

假设两个输入分别是 x=1010 和 x*=1101

则相应的输出是 y=0101 和 y*=1011

输入的异或为 x'= x⊕x*=0111

输出的异或为 y'= y⊕y*=1110

可以发现不论 x 和 x*如何变化，只要它们的异或是 0111，相应输出的异或都是 1110，(x', y')被称为一个差分。

如果 S 盒是线性的，整个 SPN 也会是线性的，明文和密文的差分也会是线性的。

差分分析的优势在于，分析过程基本可以忽略密钥的干扰作用。

差分密码分析思路：

找到足够多的四元组(x, x*, y, y*)，其中 x'=x⊕x*固定不变。对可能的密钥进行穷举，计算相关差分的扩散率，正确的密钥作用下，扩散率应最大。

和线性分析一样，不需要对全部密钥空间进行穷举，只需要对候选子密钥进行穷举即可。

具体算法如：差分攻击(T, T, π_{s-1})

```
for (L1,L2)=(0,0) to (F,F)          // L1,L2 表示候选子密钥 k5<2>和 k5<4>
    Count[L1,L2]=0      // 每个候选子密钥分配一个计数器并初始化为 0
for  each (x, x*, y, y*)  ∈  T {
    if (y<1>=y*<1>  and  y<3>=y*<3>)  {  // 只考虑 y'<1>和 y'<3>=0
            for (L1,L2)=(0,0) to (F,F)  {
                v^4<2> = L1 ⊕y<2>
                v^4<4> = L2 ⊕y<4>
                u^4<2> = πs^-1(v^4<2>)
                u^4<4> = πs^-1(v^4<4>)
                (v^4<2>)*= L1 ⊕(y<2>)*
                (v^4<4>)* = L2 ⊕(y<4>)*
                (u^4<2>)* = πs^-1((v^4<2>) *)
                (u^4<4>)* = πs^-1((v^4<4>)*)
                (u^4<2>)'=u^4<2> ⊕(u^4<2>)*
                (u^4<4>)'=u^4<4> ⊕(u^4<4>)*
             if  (u^4<2>)'=0110  and  (u^4<4>)' = 0110
                 Count[L1,L2] ++;
            }
     }
}
max = -1
for (L1,L2)=(0,0) to (F,F)  {
    if  Count[L1,L2] > max  {
```

max = Count[L_1,L_2]
maxkey = (L_1,L_2) // maxkey 即为所求子密钥
}
}

1.2.6 密钥穷举攻击

已知明文 Plain，通过选择明文攻击(Chosen-plaintext Attack)，得出通过原密钥 K 加密后的密文 Cipher1。然后通过穷举的密钥 K_test 加密后，得到密文 Cipher2。

如果 Cipher1 与 Cipher2 相符合，则穷举的密钥 K_test 与原密钥 K 相同，破解完成。否则继续穷举，直到破解出密钥

1.2.7 SPN 加强

安全性增强分为：增加分组的长度，密钥的长度，S 盒，轮数四个方面。

1) 分组长度从 16 位增加到 64 位
2) 密钥长度从 64 位增加到 128 位
3) S 盒采用 AES 加密的 S 盒，安全性大大增强
4) 轮数从 4 轮增加到 8 轮

1.3 实验过程

1.3.1 SPN 加密

1）初始定义 32 位密钥 K，16 位 S 盒，16 位 P 盒：

```
unsigned int HexKey = 0x3a94d63f;

unsigned short Sub_map_value[] = {0xE,0x4,0xD,0x1,0x2,0xF,0xB,0x8,0x3,0xA,0x6,0xC,0x5,0x9,0x0,0x7};

unsigned short Per_map_value[] = {1,5,9,13,2,6,10,14,3,7,11,15,4,8,12,16};
```

图 1.4 初始定义

2）密钥编排算法

每一次加密密钥为 16 位，密钥编排算法从 32 位密钥中从左往右每隔 4 比特移动一次，抽离出八位作为该轮的轮密钥。

```
void Key_arrange(int i,unsigned short &Kr,unsigned int Key){
    unsigned int mask = 0xffff0000;
    i -= 1;
    mask = mask>>(4*i);
    Kr = (Key & mask)>>(4-i)*4;
}
```

图 1.5 密钥编排

3）S 盒替换

S 盒的输入每一个值经过 S 盒都对应一个输出，输出即为初始定义中的 Sub_map_value 数组。

```
void Substi(unsigned short Ur, unsigned short &Vr){
    unsigned short mask[] = {0xf000,0x0f00,0x00f0,0x000f};
    unsigned short res;
    unsigned short temp = 0x0000;
    for (int i = 0; i < 4; ++i) {
        res = (Ur & mask[i])>>(4*(3-i));
        temp |= (Sub_map_value[res]<<(4*(3-i)));
    }
    Vr = temp;
}
```

图 1.6 S 盒替换

4）P 盒置换

P 盒的每一比特输入都对应输出的某一位，该 P 盒的输出范围是 0~16 位，输出结果即为 Per_map_value 数组。

```
void Permutation(unsigned short Vr, unsigned short &w){
    unsigned int mask = 0x10000;
    unsigned short temp;
    unsigned short res = 0x0000;
    for (int i = 0; i < 16; ++i) {
        mask = mask>>1;
        temp = (Vr & mask)>>(15-i);
        res = res | (temp<<(16-Per_map_value[i]));
    }
    w = res;
}
```

图 1.7 P 盒置换

5）SPN 加密

根据 1.2.3 节中 SPN 密码体制设计的原理，下图是 SPN 的代码实现：

```
void SPN(unsigned short hex_x, unsigned short &res, unsigned int Key){
    unsigned short w = hex_x;
    unsigned short Kr,Ur,Vr = 0x0000;
    int Nr = 4;
    for (int r = 1; r <= Nr-1; ++r) {
        Key_arrange(r, Kr, Key);
        //cout<<"Kr:"<<hex<<Kr<<endl;
        Ur = Kr ^ w;
        //cout<<"Ur:"<<Ur<<endl;
        Substi(Ur, Vr);
        //cout<<"Vr:"<<Vr<<endl;
        Permutation(Vr,w);
        //cout<<"w:"<<w<<endl;
        //cout<<"-----"<<endl;
    }
    Key_arrange(Nr, Kr, Key);
    Ur = Kr ^ w;
    Substi(Ur,Vr);
    Key_arrange(Nr+1, Kr, Key);
    res = Vr ^ Kr;
}
```

图 1.8 SPN 加密

1.3.2 线性分析

依据线性分析的原理，先分析出书上已有方法的 8 位密钥，再分析出自定义的一条差分链的 8 位密码，则依据线性分析可分析出 16 位密码。最后结合分析的 16 位密钥再穷举剩下的 16 位密钥。

1）生成明密文对

首先利用随机数随机生成明文，但是明文的范围必须在(0, 0xffff)之内，所以利用 rand() % 0xffff；再利用 SPN 加密机制生成对应的密文 Output_y。输入 Input_x 放入 Input 数组中存储，输出 Output_y 放入 Output 数组中存储。

```
void generator(unsigned short* Input,unsigned short* Output){
    unsigned short Input_x;
    unsigned short Output_y;
    srand((unsigned)time(nullptr));
    for (int i = 0; i < pairs; ++i) {
        Input_x = rand() % 0xffff;
        SPN(Input_x,Output_y,HexKey);
        Input[i] = Input_x;
        Output[i] = Output_y;
        //cout<<hex<<Input_x<<" "<<Output_y<<endl;
    }
}
```

图 1.9 生成明密文对

2）线性分析

图 1.10 中的 for 循环穷举最后一轮的密钥，根据线性分析的原理，最后一轮中符合条件（即对应比特位异或值为 0，其中 Z_24, Z_13 = 0）的密钥就使用计数器 count_24 或者 count_13 数组进行记录。

```
void LinearAttack(unsigned short* Input, unsigned short* Output,unsigned short &key_r16){
    short count_24[16][16] = {0};
    short count_13[16][16] = {0};
    unsigned short Y1,Y2,Y3,Y4,L1,L2,L3,L4,V_41,V_42,V_43,V_44,U_41,U_42,U_43,U_44 = 0x0;
    for (int i = 0; i < pairs; ++i) {
        Y1 = (Output[i] & 0xf000)>>12;
        Y2 = (Output[i] & 0x0f00)>>8;
        Y3 = (Output[i] & 0x00f0)>>4;
        Y4 = (Output[i] & 0x000f);
        for (short j = 0; j < 16; ++j) {
            L4 = j;
            L1 = L4;
            for (short k = 0; k < 16; ++k) {
                L2 = k;
                L3 = L2;
                V_41 = Y1 ^ L1;
                V_42 = Y2 ^ L4;
                V_43 = Y3 ^ L3;
                V_44 = Y4 ^ L2;
                U_41 = Reverse_SubMap[V_41];
                U_42 = Reverse_SubMap[V_42];
                U_43 = Reverse_SubMap[V_43];
                U_44 = Reverse_SubMap[V_44];
                short Z_24 = ((Input[i] & 0x0800)>>11) ^ ((Input[i] & 0x0200)>>9) ^ ((Input[i] & 0x0100)>>8)
                        ^ ((U_42 & 0x4)>>2) ^ (U_42 & 0x1) ^ ((U_44 & 0x4)>>2) ^ (U_44 & 0x1);
                if (Z_24 == 0)
                    count_24[j][k]++;
                short Z_13 = ((Input[i] & 0x0400)>>10) ^ ((Input[i] & 0x0200)>>9) ^ ((Input[i] & 0x0100)>>8)
                        ^ ((U_43 & 0x4)>>2) ^ (U_43 & 0x1) ^ ((U_41 & 0x4)>>2) ^ (U_41 & 0x1);
                if(Z_13 == 0)
                    count_13[j][k]++;
            }
        }
    }
```

图 1.10 线性分析-1

图 1.11 中的 for 循环是将计数器所记录的密钥符合数目的最大值挑选出来，并将这个值作为最终破解的密钥。Key_r16 则为最终的 16 位密钥值。

```
    short max_24 = -1;
    short max_13 = -1;
    unsigned short Max_1,Max_2,Max_3,Max_4 = 0x0;
    for (short i = 0; i < 16; ++i) {
        for (short j = 0; j < 16; ++j) {
            count_24[i][j] = abs(count_24[i][j] - pairs/2);
            count_13[i][j] = abs(count_13[i][j] - pairs/2);
            if(count_24[i][j] > max_24){
                max_24 = count_24[i][j];
                Max_2 = i;
                Max_4 = j;
            }
            if(count_13[i][j] > max_13){
                max_13 = count_13[i][j];
                Max_1 = i;
                Max_3 = j;
            }
        }
    }
    key_r16 = (Max_1<<12) | (Max_2<<8) | (Max_3<<4) | Max_4;
    //cout<<hex<<Max_1<<" "<<Max_2<<" "<<Max_3<<" "<<Max_4<<endl;
    cout<<hex<<key_r16<<endl;
}
```

图 1.11 线性分析-2

1.3.3 差分分析

1）生成明密文对

图 1.11 中先随机生成x，利用 SPN 加密生成密文，将二者对应的存入 Input 和 Output 数组中。再利用差分的特点，定义一个差分特征值，再生成一个x'，利用x'和 SPN 加密生成对应的密文将二者对应的存入_Input_和_Output_数组中。

```
void gene_Di(unsigned short* Input, unsigned short* Output,unsigned short* _Input_,unsigned short* _Output_){
    unsigned short x,y,_x_,_y_;
    srand((unsigned)time(nullptr));
    for (int i = 0; i < Di_pairs; ++i) {
        x = rand() % 0xffff;
        SPN(x,y,HexKey);
        _x_ = x ^ Ruler;
        SPN(_x_,_y_,HexKey);
        Input[i] = x;
        Output[i] = y;
        _Input_[i] = _x_;
        _Output_[i] = _y_;
        //cout<<hex<<"x: "<<x<<" y: "<<y<<" x*: "<<_x_<<" y*: "<<_y_<<endl;
    }
}
```

图 1.12 生成明密文对

2）差分分析

图 1.13 中的 for 循环穷举最后一轮的密钥，最后一轮中符合差分条件（即对应比特位异或值为

0，其中 Z_24, Z_13 = 0）的密钥就使用计数器 count_24 或者 count_13 数组进行记录。

```
void Dif_Attack(unsigned short* Output,unsigned short* _Output_, unsigned short &key_r16){
    short count_24[16][16] = {0};
    short count_13[16][16] = {0};
    unsigned short y1,_y1_,y2,_y2_,y3,_y3_,y4,_y4_,L1,L2,L3,L4,
    V41,V42,V43,V44,U41,U42,U43,U44,_V41_,_V42_,_V43_,_V44_,
    _U41_,_U42_,_U43_,_U44_,uu41,uu42,uu43,uu44;
    for (short i = 0; i < Di_pairs; ++i) {
        y1 = (Output[i] & 0xf000)>>12;
        _y1_ = (_Output_[i] & 0xf000)>>12;
        y2 = (Output[i] & 0x0f00)>>8;
        _y2_ = (_Output_[i] & 0x0f00)>>8;
        y3 = (Output[i] & 0x00f0)>>4;
        _y3_ = (_Output_[i] & 0x00f0)>>4;
        y4 = (Output[i] & 0x000f);
        _y4_ = (_Output_[i] & 0x000f);
        //2,4
        if (y1 == _y1_ && y3 == _y3_) {
            for (short j = 0; j < 16; ++j) {
                L2 = j;
                for (short k = 0; k < 16; ++k) {
                    L4 = k;
                    V42 = L2 ^ y2;
                    V44 = L4 ^ y4;
                    U42 = Reverse_SubMap[V42];
                    U44 = Reverse_SubMap[V44];
                    _V42_ = L2 ^ _y2_;
                    _V44_ = L4 ^ _y4_;
                    _U42_ = Reverse_SubMap[_V42_];
                    _U44_ = Reverse_SubMap[_V44_];
                    uu42 = U42 ^ _U42_;
                    uu44 = U44 ^ _U44_;
                    if (uu42 == Ruler_24 && uu44 == Ruler_24)
                        count_24[j][k]++;
                }
            }
        }
        //1,3
        if (y2 == _y2_ && y4 == _y4_){
            for (short i = 0; i < 16; ++i) {
                L1 = i;
                for (short j = 0; j < 16; ++j) {
                    L3 = j;
                    V41 = y1 ^ L1;
                    V43 = y3 ^ L3;
                    U41 = Reverse_SubMap[V41];
                    U43 = Reverse_SubMap[V43];
                    _V41_ = _y1_ ^ L1;
                    _V43_ = _y3_ ^ L3;
                    _U41_ = Reverse_SubMap[_V41_];
                    _U43_ = Reverse_SubMap[_V43_];
                    uu41 = U41 ^ _U41_;
                    uu43 = U43 ^ _U43_;
                    if (uu41 == Ruler_13 && uu43 == Ruler_13)
                        count_13[i][j]++;
                }
            }
        }
    }
```

图 1.13 差分分析-1

图 1.14 中 for 循环是将计数器所记录的密钥符合数目的最大值挑选出来，并将这个值作为最终的密钥。Key_r16 则为最终的 16 位密钥值。

```
short max_24,max_13 = -1;
unsigned short max_1,max_2,max_3,max_4;
for (short i = 0; i < 16; ++i) {
    for (short j = 0; j < 16; ++j) {
        if (count_24[i][j] > max_24){
            max_24 = count_24[i][j];
            max_2 = i;
            max_4 = j;
        }
        if (count_13[i][j] > max_13){
            max_13 = count_13[i][j];
            max_1 = i;
            max_3 = j;
        }
    }
}
key_r16 = (max_1<<12) | (max_2<<8) | (max_3<<4) | (max_4);
cout<<hex<<key_r16<<endl;
//cout << hex <<max_1<<" "<< max_2 <<" "<<max_3<<" "<< max_4 <<endl;
```

图 1.14 差分分析-2

1.3.4 穷举攻击

线性分析或者差分分析已经求得 16 位密钥，剩下的 16 位密钥需要穷举。利用一对已知的明密文对，将尝试的 16 位密钥与已知的 16 位密钥结合，加密检验是否与已知的密文相等。若相等，则设置成为密钥。有时可能存在假密钥，则再设置一对明密文对进行筛选检验即可得出正确结果。

```
void Crack(unsigned short key_r16){
    unsigned short hex_x = 0x26b7;
    unsigned short res,temp;
    SPN(hex_x,res,HexKey);//result of hex_x
    unsigned short key_f16 = 0x0000;
    unsigned int Key_new = 0x00000000;
    bool flag = true;
    while(flag){
        Key_new = (key_f16<<16) | key_r16;
        SPN(hex_x,temp,Key_new);
        if(temp == res)
            flag = false;
        else
            key_f16 += 1;
    }
    cout<<hex<<Key_new<<endl;
}
```

图 1.15 穷举攻击

1.3.5 SPN 加强

1）密钥编排

每一次加密密钥为 64 位，密钥编排算法从 128 位密钥中从左往右每隔 8 比特移动一次，抽离出 64 位作为该轮的轮密钥。

```
void Key_arrange_Plus(int index, unsigned long long &Kr, unsigned long long* key){
    index -= 1;
    unsigned long long temp = 0x0000000000000000;//64 bits
    for (int j = 1; j <= 8; j++,index++)
        temp |= (key[index] << (8 * (8 - j)));
    Kr = temp;
}
```

图 1.16 密钥编排

2）初始定义 128 位密钥 K，16 位 S 盒，64 位 P 盒

```
unsigned long long Substi_box8[16][16] = {{0x63,0x7c,0x77,0x7b,0xf2,0x6b,0x6f,0xc5,0x30,0x01,0x67,0x2b,0xfe,0xd7,0xab,0x76},
                                          {0xca,0x82,0xc9,0x7d,0xfa,0x59,0x47,0xf0,0xad,0xd4,0xa2,0xaf,0x9c,0xa4,0x72,0xc0},
                                          {0xb7,0xfd,0x93,0x26,0x36,0x3f,0xf7,0xcc,0x34,0xa5,0xe5,0xf1,0x71,0xd8,0x31,0x15},
                                          {0x04,0xc7,0x23,0xc3,0x18,0x96,0x05,0x9a,0x07,0x12,0x80,0xe2,0xeb,0x27,0xb2,0x75},
                                          {0x09,0x83,0x2c,0x1a,0x1b,0x6e,0x5a,0xa0,0x52,0x3b,0xd6,0xb3,0x29,0xe3,0x2f,0x84},
                                          {0x53,0xd1,0x00,0xed,0x20,0xfc,0xb1,0x5b,0x6a,0xcb,0xbe,0x39,0x4a,0x4c,0x58,0xcf},
                                          {0xd0,0xef,0xaa,0xfb,0x43,0x4d,0x33,0x85,0x45,0xf9,0x02,0x7f,0x50,0x3c,0x9f,0xa8},
                                          {0x51,0xa3,0x40,0x8f,0x92,0x9d,0x38,0xf5,0xbc,0xb6,0xda,0x21,0x10,0xff,0xf3,0xd2},
                                          {0xcd,0x0c,0x13,0xec,0x5f,0x97,0x44,0x17,0xc4,0xa7,0x7e,0x3d,0x64,0x5d,0x19,0x73},
                                          {0x60,0x81,0x4f,0xdc,0x22,0x2a,0x90,0x88,0x46,0xee,0xb8,0x14,0xde,0x5e,0x0b,0xdb},
                                          {0xe0,0x32,0x3a,0x0a,0x49,0x06,0x24,0x5c,0xc2,0xd3,0xac,0x62,0x91,0x95,0xe4,0x79},
                                          {0xe7,0xc8,0x37,0x6d,0x8d,0xd5,0x4e,0xa9,0x6c,0x56,0xf4,0xea,0x65,0x7a,0xae,0x08},
                                          {0xba,0x78,0x25,0x2e,0x1c,0xa6,0xb4,0xc6,0xe8,0xdd,0x74,0x1f,0x4b,0xbd,0x8b,0x8a},
                                          {0x70,0x3e,0xb5,0x66,0x48,0x03,0xf6,0x0e,0x61,0x35,0x57,0xb9,0x86,0xc1,0x1d,0x9e},
                                          {0xe1,0xf8,0x98,0x11,0x69,0xd9,0x8e,0x94,0x9b,0x1e,0x87,0xe9,0xce,0x55,0x28,0xdf},
                                          {0x8c,0xa1,0x89,0x0d,0xbf,0xe6,0x42,0x68,0x41,0x99,0x2d,0x0f,0xb0,0x54,0xbb,0x16}
};

int PerMap_plus[64] = {1,9,17,25,33,41,49,57,
                       2,10,18,26,34,42,50,58,
                       3,11,19,27,35,43,51,59,
                       4,12,20,28,36,44,52,60,
                       5,13,21,29,37,45,53,61,
                       6,14,22,30,38,46,54,62,
                       7,15,23,31,39,47,55,63,
                       8,16,24,32,40,48,56,64};

unsigned long long Key_Plus[] = {0x92,0xd7,0x8b,0x0c,0x3f,0x46,0x5e,0x9a,0x67,0xcd,0x26,0xb7,0x3a,0x94,0xd6,0x3f};//128 bits
```

图 1.17 初始定义

3）S 盒

S 盒的输入每一个值经过 S 盒都对应一个输出，输出即为初始定义中的 Substi_box8 数组。

```
void Substi_Plus(unsigned long long Ur, unsigned long long &Vr){
    unsigned long long mask_x = 0xf000000000000000;//64 bits
    unsigned long long mask_y = 0x0f00000000000000;
    unsigned long long temp = 0x0000000000000000;//64 bits
    unsigned short x,y = 0x0;
    for (int i = 0; i < 8; i++) {
        x = (mask_x & Ur)>>(64-4*(2*i+1));//4 bits
        y = (mask_y & Ur)>>(64-4*(2*i+1+1));
        temp |= (Substi_box8[x][y] << (64 - 8 * (i + 1)));
        mask_x = mask_x>>8;
        mask_y = mask_y>>8;
    }
    Vr = temp;
}
```

图 1.18 S 盒

4）P 盒

S 盒的输入每一个值经过 S 盒都对应一个输出，输出即为初始定义中的 PerMap_plus 数组。

```
void Permutation_Plus(unsigned long long Vr, unsigned long long &w){
    unsigned long long mask = 0x8000000000000000;//64 bits
    unsigned long long temp,res = 0x0000000000000000;
    for (int k = 0; k < 64; ++k) {
        temp = (Vr & mask)>>(63-k);
        res |= (temp<<(64-PerMap_plus[k]));
        mask = mask>>1;
    }
    w = res;
}
```

图 1.19 P 盒

5）加强 SPN 加密

根据 1.2.3 节中 SPN 密码体制设计的原理，下图是 SPN 加强版的代码实现：

```
void SPN_Plus(unsigned long long x, unsigned long long &res){
    unsigned long long w = x;
    unsigned long long Kr,Ur,Vr = 0x0000000000000000;//64 bits
    int Nr = 8;
    for (int r = 1; r <= Nr-1; ++r) {
        Key_arrange_Plus(r,Kr,Key_Plus);
        //cout<<"Kr: "<<hex<<Kr<<endl;
        Ur = Kr ^ w;
        //cout<<"Ur:"<<hex<<Ur<<endl;
        Substi_Plus(Ur,Vr);
        //cout<<"Vr: "<<hex<<Vr<<endl;
        Permutation_Plus(Vr,w);
        //cout<<"w:"<<hex<<w<<endl;
    }
    Key_arrange_Plus(Nr,Kr,Key_Plus);
    Ur = Kr ^ w;
    Substi_Plus(Ur,Vr);
    Key_arrange_Plus(Nr+1,Kr,Key_Plus);
    res = Vr ^ Kr;
}
```

图 1.20 SPN 加密

1.3.6 随机性检测

1）写入文件

先将 16 进制的 0 写入到文件中，文件的大小为 10M，即为 10*1024*1024。采用 c++文件流 write 函数写入文件。

```
void Write(){
    fstream fin("C:\\Users\\PC\\Desktop\\crypto\\test.txt",ios::binary|ios::out);
    //write into file with hex
    char *temp=new char[10*1024*1024];
    memset(temp,0,10*1024*1024);
    fin.write(temp,10*1024*1024);
    fin.close();
}
```

图 1.21 写入文件

2）加密文件

加密文件采用加强版 SPN 进行加密，加密模式采用 CBC 模式，初始向量设置为全 0.。将加密后的内容写入到文件中，文件大小为 10M。

```
void Rand(){
    fstream fin("C:\\Users\\PC\\Desktop\\crypto\\test.txt",ios::binary|ios::in);
    ofstream fout("C:\\Users\\PC\\Desktop\\crypto\\Output.txt",ios::binary|ios::out);

    unsigned long long Iv = 0x0000000000000000;
    char *buffer = new char[10*1024*1024];
    fin.read(buffer,10*1024*1024);
    unsigned long long temp_x,temp_res = 0x0000000000000000;
    char t[8];

    for (int i = 0,count = 0; i <10*1024*1024; ++i) {
        temp_x |= (unsigned long long) ((unsigned char)buffer[i]) << (64 - 8 * (count + 1));//8 bits per buffer[i]
        count++;
        if(count == 7){
            temp_x = temp_x ^ Iv;
            SPN_Plus(temp_x,temp_res);
            Iv = temp_res;
            //cout<<hex<<temp_res<<endl;
            //output
            unsigned long long mask = (unsigned long long)(0xff)<<56;
            for (int j = 0; j < 8; ++j) {
                t[j] = (mask&temp_res)>>((7-j)*8);
                mask = mask>>8;
            }
            fout.write(t,8);

            count = 0;
        }
    }

    fin.close();
    fout.close();
}
```

图 1.22 加密文件

1.4 实验结果

1.4.1 SPN 加密

明文是 0x26b7，结果如下：

```
The result is: bcd6
```

图 1.23 SPN 加密

1.4.2 线性分析

线性攻击中，其中八位采用 8000 对明密文，其中八位采用 32000 对明密文，结果如下：

```
d63f
3a94d63f
226.655ms
```

图 1.24 线性分析

1.4.3 差分分析

差分分析中，其中八位采用 500 对明密文对，其中八位采用 7000 对明密文对，结果如下：

```
d63f
3a94d63f
13.636ms
```

图 1.25 差分分析

1.4.4 随机性检测

利用工具对加密的文件进行随机性检测，结果如下：

图 1.26 随机性检测

由上可知，加密后的文件通过了随机性检测。

2 RSA 实验

2.1 实验内容

1）生成 RSA 算法的参数：（如 p、q、N、私钥、公钥等）；
2）编写 RSA 的加/解密过程；
3）快速实现 RSA（对比模重复平方、蒙哥马利算法和中国剩余定理）；

2.2 实验原理

2.2.1 公钥密码算法

公钥密码算法的特点：
加密: $C = E_{K\text{-}pub}(P)$
解密: $P = D_{K\text{-}prv}(C) = D_{K\text{-}prv}(E_{K\text{-}pub}(P))$
两个密钥不可相互推导（或推导的难度不亚于密码分析），其中一个密钥公开 K_{pub}(公钥)，另一个保密 K_{prv}(私钥)，每个用户掌握一个私钥，并把相应的公钥放在公共目录中

2.2.2 RSA 算法

RSA 密码体制是目前为止最成功的非对称密码算法，它是在 1977 年由 Rivest、Shamir 和 Adleman 提出的第一个比较完善的非对称密码算法。它的安全性是建立在“大数分解和素性检测”这个数论难题的基础上，即将两个大素数相乘在计算上容易实现，而将该乘积分解为两个大素数因子的计算量相当大。虽然它的安全性还未能得到理论证明，但经过 20 多年的密码分析和攻击，迄今仍然被实践证明是安全的。
RSA 算法描述如下：
（1）公钥
选择两个互异的大素数 p 和 q，n 是二者的乘积，即 n=p*q，使 Φ(n)=(p-1)(q-1)为欧拉函数。随机选取正整数 e，使其满足 gcd (e, Φ(n)) =1，即 e 和 Φ(n) 互质，则将(n, e)作为公钥。
（2）私钥
求出正数 d，使其满足 $e\times d=1 (\mathrm{mod}\ \Phi(n))$，则将(n, d)作为私钥。
（3）加密算法
对于明文 M，由 $C=M^e (\mathrm{mod}\ n)$，得到密文 C。
（4）解密算法
对于密文 C，由 $M=C^d (\mathrm{mod}\ n)$，得到明文 M。
如果窃密者获得了 n、e 和密文 C，为了破解密文必须计算出私钥 d，为此需要先分解 n。为了提高破解难度，达到更高的安全性，一般商业应用要求 n 的长度不小于 1024 位，更重要的场合不小于 2048 位。

2.2.3 快速实现 RSA 算法

1）模重复平方
在模算术计算中，我们常常要对大整数模 m 和大整数 n，计算 $b^n (\mathrm{mod}\ m)$ 。如果用递归地计算 $b^n \equiv (b^{n-1} (\mathrm{mod}\ m)) \cdot b \ (\mathrm{mod}\ m)$ ，这种计算较为费时，须作 n-1 次乘法.。
现在，将 n 写成二进制：$n = n_0 + n_1 2 + \cdots + n_{k-1} 2^{k-1}$，其中，$n_i \in \{0, 1\}$, $i = 0, 1, \cdots, k-1$, 则

b^n (mod m)的计算可归纳为：

$$b^n \equiv b^{n_0}(b^2)^{n_1} \ldots \left(b^{2^{k-1}}\right)^{n_{k-1}} \ (mod\ m)$$

我们最多作 2[$\log_2 n$]次乘法.这个计算方法叫做“模重复平方计算法”。

具体计算如下:

(a) 令 a = 1，并将 n 写成二进制：$n = n_0 + n_1 2 + \cdots + n_{k-1}2^{k-1}$，其中 $n_i \in \{0, 1\}$，i = 0,1,…,k-1 .

(b) 如果 $n_0 = 1$,则计算 $a_0 \equiv a*b(\text{mod m})$,否则取 $a_0 = a$,即计算$a_0 \equiv a * b^{n_0}(\text{mod m})$，再计算$b_1 \equiv b_0^2(\text{mod m})$.

(c) 如果 $n_1 = 1$,则计算 $a_1 \equiv a_0*b_1 (\text{mod m})$,否则取 $a_1 = a_0$,即计算$a_1 \equiv a_0 * b_1^{n_1}(\text{mod m})$，再计算$b_2 \equiv b_1^2(\text{mod m})$.

(d) 如果 $n_{k-2} = 1$,则计算 $a_{k-2} \equiv a_{k-3}*b_{k-2} (\text{mod m})$,否则取 $a_{k-2} = a_{k-3}$,即计算$a_{k-2} \equiv a_{k-3} * b_{k-2}^{n_{k-2}}(\text{mod m})$，再计算$b_{k-1} \equiv b_{k-2}^2(\text{mod m})$.

(k)如果 $n_{k-1} = 1$,则计算 $a_{k-1} \equiv a_{k-2} \cdot b_{k-1} (\text{mod m})$,否则取 $a_{k-1} = a_{k-2}$，即计算$a_{k-1} \equiv a_{k-2} \cdot b_{k-1}^{n_{k-1}}(\text{mod m})$.即$a_{k-1} = b^n(\text{mod m})$.

2）中国剩余定理（CRT）

中国剩余定理（CRT）：已知$n_1, n_2 \ldots\ldots n_k$为两两互素的正整数，则同余方程组为

$$x \equiv b_i \ mod\ n_i$$

X 在 0 到 N 内有唯一解，其中 i=1, 2, ⋯, k，b_i为正整数。根据高斯算法，中国剩余定理的解为 $x = b_1M_1y_1 + \cdots + b_kM_ky_k \ (modN)$，其中$N = n_1 * n_2 \ldots\ldots * n_k$，$M_i = \frac{N}{n_i} = n_1 * n_2 \ldots\ldots * n_k, y_i = M_{i-1} \ (mod\ n_i)$。

由此可见，中国剩余定理为对高位宽(如 1024bit)大数的模幂运算转化成对低位宽(如 512bit)相对较小的数进行模幂运算提供了可能

3）蒙哥马利模乘算法

由于 RSA 的核心算法是模幂运算，模幂运算又相当于模乘运算的循环，要提高 RSA 算法的效率，首要问题在于提高模乘运算的效率。

模乘过程中复杂度最高的环节是求模运算，因为一次除法实际包含了多次加法、减法和乘法，如果在算法中能尽量减少甚至避免除法，则算法的效率会大大提高。

蒙哥马利算法的核心思想在于将求 A*B %N 转化为不需要反复取模的 A*B*R' %N（移位即可，因为 R 是 2^K,总之 R 是与进制相关的数），即不使用除法（用移位操作）而求得模乘运算的结果

对于任意指数 E，都可采用以下算法计算 D=C**E % N：

```
D=1
WHILE E>0{
  IF（E%2=0）{
    C=C*C % N
```

```
        E=E/2
      }
      ELSE{
        D=D*C % N
        E=E-1
      }
    }
    RETURN D
```

继续分析会发现，要知道 E 何时能整除 2，并不需要反复进行减一或除二的操作，只需验证 E 的二进制各位是 0 还是 1 就可以了，从左至右或从右至左验证都可以，从左至右会更简洁，

设 E=Sum[i=0 to n](E*2**i)，0<=E<=1，则：

```
D=1
FOR i=n to 0{
  D=D*D % N
  IF（E=1）
    D=D*C % N
}
RETURN D
```

2.3 实验过程

2.3.1 生成 RSA 参数

1）生成大素数

利用 OpenSSL 库里的函数 BIGNUM *BN_generate_prime，可以直接取得所需位数的大素数。

```
//generate p,q
do{
    BN_generate_prime(p, 512, NULL, NULL, NULL, NULL, NULL);
}while (!BN_is_prime(p,NULL,NULL,NULL,NULL));
do{
    BN_generate_prime(q, 512, NULL, NULL, NULL, NULL, NULL);
}while (!BN_is_prime(q,NULL,NULL,NULL,NULL));
```

图 2.1 生成大素数

2）生成参数

大数运算直接利用 OpenSSL 既有函数进行运算。其中取 e 用到欧几里得辗转相除法。

```
BN_mul(n,p,q,ctx);//n=p*q
BN_sub(p_Cutone,p,BN_value_one()); //p_Cutone = p - 1
BN_sub(q_Cutone,q,BN_value_one());//q_Cutone = q - 1
BN_mul(exp,p_Cutone,q_Cutone,ctx);//exp = p_Cutone * q_Cutone

do{
    BN_rand_range(e,exp);
    BN_gcd(GCD,e,exp,ctx);
}while (BN_cmp(GCD,BN_value_one()));// e
while (!BN_mod_inverse(d,e,exp,ctx));//d
```

图 2.2 生成 RSA 参数

2.3.2 模重复平方方法

根据前面的原理，将算法的核心部分展示如下：

```
//(b^pow)mod(mod)=result
//将Pow转换为二进制的算法：位上为1是an=an-1*bn,bn=bn-1*bn-1;位上为0时是an=an-1,bn=bn-1*bn-1
do {
    BN_div(DivRes, Rem, Cup_Pow, Two, ctx);//计算Cup_Pow与Two的商，值储存在DivRes中，余数储存在rem中。
    if (BN_is_one(Rem))
        BN_mod_mul(a, a_new, Cup_b, Mod, ctx);//计算a_new与Cup_b的积，再模mod，值储存在a中。
    else BN_copy(a, a_new);

    BN_mod_sqr(b_sqr, Cup_b, Mod, ctx);//计算Cup_b^2，再模mod，值储存在b_sqr中;


    BN_copy(a_new, a);
    BN_copy(Cup_Pow, DivRes);
    BN_copy(Cup_b, b_sqr);

} while (!BN_is_zero(DivRes));//余数为0跳出循环
```

图 2.3 模重复平方方法

2.3.3 中国剩余定理

根据前面的原理，将算法的核心部分展示如下：

```
BN_mul(Mod, p, q, ctx);//mod=n=p*q

BN_sub(p_CutOne, p, BN_value_one());//p-1
BN_sub(q_CutOne, q, BN_value_one());//q-1

BN_nnmod(Pow1, Pow, p_CutOne, ctx);//计算Pow与p-1的模，小于0就加上p-1，存在Pow1中;
BN_nnmod(Pow2, Pow, q_CutOne, ctx);

ModRepeatSquare(Cp, y, p, Pow1);//Cp=y^Pow1(mod p)
ModRepeatSquare(Cq, y, q, Pow2);//Cq=y^Pow2(mod q)

BN_mod_inverse(c1,q,p,ctx);//模逆，((c*q)%p==1)
BN_mod_inverse(c2,p,q,ctx);

BN_mul(x11, Cp, c1, ctx);//x11=Cp*c1
BN_mul(x12, x11, q, ctx);//x12=x11*q
BN_mul(x21, Cq, c2, ctx);//x21=Cq*c2
BN_mul(x22, x21, p, ctx);//x22=x21*p

BN_mod_add(Result, x12, x22, Mod, ctx);//x=(x12+x22)mod Mod
```

图 2.4 中国剩余定理

2.3.4 蒙哥马利算法

根据前面的原理，将算法的核心部分展示如下：

```
while (BN_cmp(Pow_backup, Zero) == 1)//大于0循环
{
    BN_mod(Cup, Pow_backup, Two, ctx);//计算Pow_backup与2的模，存入Cup(取余)

    if (BN_is_zero(Cup)) {
        BN_mod_sqr(Res, Res, Mod, ctx);//Res=Res^2(mod Mod)

        BN_div(Pow_backup, NULL, Pow_backup, Two, ctx);//Pow_backup=Pow_backup/2

    }
    else {
        BN_mod_mul(Result, Result, Res, Mod, ctx);//Result=Result*Res(mod Mod)

        BN_sub(Pow_backup, Pow_backup, BN_value_one());//Pow_backup=Pow_backup-1
    }
}
```

图 2.5 蒙哥马利算法

2.4 实验结果

2.4.1 生成 RSA 参数

RSA 生成的参数包括公钥(N, e)，私钥(N, d)

```
--------------------------------RSA 参数生成 （16进制） -------------------------------
public key:
 n = d1ed6132980dd1ba4a67e9159efde97f95d59f172aedb1b8a7ec30fa0e1bf1f4748e512abadb0dcf17
066d87f474a4f7415217dd0ccb4437613ab02c89e31ab7173cbe0a0a387cdd7d8266ebc34f750fb88916494
7464651b38e99e7353c55b959c459b4619ff2cdc52ae6825c536583189a69457c9f7e5484a5360c7c04346b
 e = 10001
pravate key:
 n = d1ed6132980dd1ba4a67e9159efde97f95d59f172aedb1b8a7ec30fa0e1bf1f4748e512abadb0dcf17
066d87f474a4f7415217dd0ccb4437613ab02c89e31ab7173cbe0a0a387cdd7d8266ebc34f750fb88916494
7464651b38e99e7353c55b959c459b4619ff2cdc52ae6825c536583189a69457c9f7e5484a5360c7c04346b
 d = 6202067ffd776e34202ad52a4ff9517b4c73d87b93823797f36615139964b8989d961814eff9f73f79
0b7a35a00e993a0a980e2618a409ac254ba61695259f4184e74b9840e4a588e8422eb0e82083acabfa9ddee
96231bc8f20906b6ee78a264489d95d3cdd6e1b197469cd5ab6cc6ac788d1add5585be8226792f4e9b56ab9
--------------------------------RSA 参数生成完成 ---------------------------------
```

图 2.6 生成 RSA 参数

2.4.2 模重复平方根

```
--------------------------------RSA 加解密测试---------------------------------
明文:
2460fad5dcf3af243a0388b7ffb5bcbbb958aafb000
密文:
56f231f86b60147cc43ae8b1fea763c278b04ce9edeb9e08927ee03bc52468658008d7b99fa3804028f5f5ea06b5fb44a853fe
    0ca06d3433f5ac338632bd5fa43718ce9ba73571dc11bb602c86213e65bf99785c40d984d9cf61c3c594acc17cacf3e6b9
    5a55fa4c3c92f83fd2c4d023c8125fb0e3cc2b0fcd15ff7c98ff19e9
CRT加速解密耗时 :0.002483s
解密信息:
2460fad5dcf3af243a0388b7ffb5bcbbb958aafb000
模重复平方法耗时 : 0.00254 s
```

图 2.7 模重复平方加解密

2.4.3 中国剩余定理

```
----------------------------------RSA 加解密测试-----------------------------------
明文:
2460fad5dcf3af243a0388b7ffb5bcbbb958aafb000
密文:
30f5499d9967b2adbef11e775f9f11f04b9373387087b78fc5ba92a0358c6a246d1798c528718d44c66102b769e9e59e2e80ab
    22e0d60975722601feef2f8c63c4d52b81e1626858d36d8e60a6dc4e7cc85f8d646738c839c8f4f4a2858f08c3153e7e7a
    ca18514522e7451797c32f650ce79ef2a33577db2c0c17bb0697d6d3
CRT加速解密耗时 :0.016327s
解密信息:
2460fad5dcf3af243a0388b7ffb5bcbbb958aafb000
```

图 2.8 中国剩余定理加解密

2.4.4 蒙哥马利算法

```
----------------------------------RSA 加解密测试-----------------------------------
明文:
2460fad5dcf3af243a0388b7ffb5bcbbb958aafb000
密文:
30f5499d9967b2adbef11e775f9f11f04b9373387087b78fc5ba92a0358c6a246d1798c528718d44c66102b769e9e59e2e80ab
    22e0d60975722601feef2f8c63c4d52b81e1626858d36d8e60a6dc4e7cc85f8d646738c839c8f4f4a2858f08c3153e7e7a
    ca18514522e7451797c32f650ce79ef2a33577db2c0c17bb0697d6d3
CRT加速解密耗时 :0.016327s
解密信息:
2460fad5dcf3af243a0388b7ffb5bcbbb958aafb000
蒙哥马利算法耗时 : 0.01658 s
```

图 2.9 蒙哥马利加解密

3 ECC 实验

3.1 实验内容

利用椭圆曲线密码算法、HASH 函数、压缩函数、对称加密算法实现一个类似 PGP 的文件加解密及完整性校验功能。

3.2 实验原理

3.2.1 PGP 方案加解密的过程

假令发送方为 Alice，接收方 Bob

发送方：发送方将数据先取摘要值，再将摘要值用 ECC 签名，签名使用 Alice 的私钥。将签名后的数据盒原始数据结合后用 DES 加密，将加密用的 DES 密钥用 RSA 进行非对称加密，加密时使用 Bob 的公钥。将加密后的数据和 RSA 加密的密钥结合，通过压缩算法将数据打包。

接收方：接收方先将数据解压，提取出其中 RSA 加密的密钥和 DES 加密的数据。Bob 利用自己的私钥将数据解密得到 DES 密钥，用解密后的 DES 密钥解密数据。将解密后的数据提取出原始数据和数据的摘要，将原始数据进行哈希运算后与已有的哈希值进行比对校验。

具体流程如下图所示：

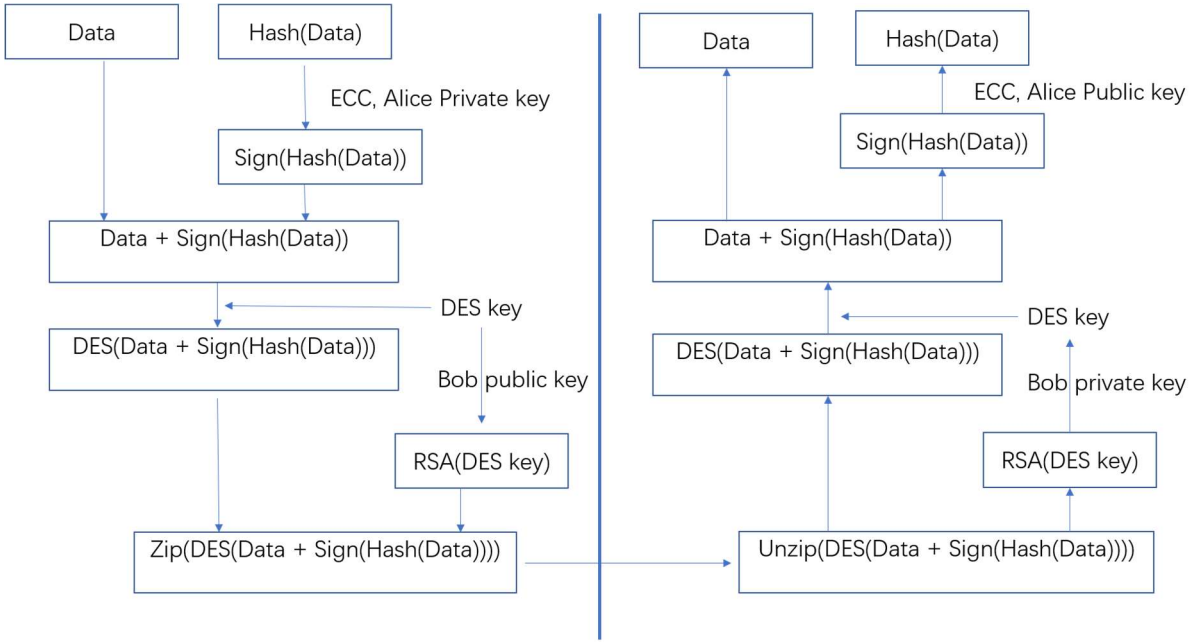

图 3.1 加解密流程

3.3 实验过程

3.3.1 生成密钥

利用 Python 的 crypto 库中的函数，生成 RSA 加密与 ECC 加密所需的公钥与私钥，并且随机生成一个 8 bytes 的 DES 密钥。

```
# return: (private_key, public_key)
def generate_ECC_key():
    private_key = ECC.generate(curve='P-256')
    return private_key, private_key.public_key()


# return: (private_key, public_key)
def generate_RSA_key():
    private_key = RSA.generate(1024)
    return private_key, private_key.publickey()


def generate_DES_key():
    key = Crypto.Random.get_random_bytes(8)
    return key
```

图 3.2 生成密钥

3.3.2 DES 加解密

```
def DES_encrypt(data, key):
    cipher = DES.new(key, DES.MODE_ECB)
    cipher_text = cipher.encrypt(pad(data, 32))
    return cipher_text


def DES_decrypt(data, key):
    cipher = DES.new(key, DES.MODE_ECB)
    data_dec = cipher.decrypt(data)
    plain_text = unpad(data_dec, 32)
    return plain_text
```

图 3.3 DES 加解密

DES 加密中，采用 ECB 模式进行加密；分组长度设置为 32 位，对于数据达不到 32 位的，在末尾进行全 0 补充。DES 解密就是 DES 加密的逆过程。

3.3.3 RSA 加解密

```
def RSA_encrypt(data, key):
    cipher = PKCS1_OAEP.new(key)
    cipher_text = cipher.encrypt(data)
    return cipher_text


def RSA_decrypt(data, key):
    cipher = PKCS1_OAEP.new(key)
    plain_text = cipher.decrypt(data)
    return plain_text
```

图 3.4 RSA 加解密

在 RSA 加密中，先声明一个加密的对象，再利用 crypto 的库函数对其进行加密。

3.3.4 签名和验证

签名函数，先对数据进行取摘要值，再对数据进行签名。

验证函数，所得的 hash 值与将所得的数据进行 hash 得到的值进行比对，若相等则验证成功，若不同则验证失败。

```
def sign(data, key):
    h = SHA256.new(data)
    signer = DSS.new(key, 'fips-186-3')
    signature = signer.sign(h)
    return signature


# return True if data is authentic
def verify(data, signature, key):
    h = SHA256.new(data)
    verifier = DSS.new(key, 'fips-186-3')
    try:
        verifier.verify(h, signature)
        return True
    except ValueError:
        return False
```

图 3.5 签名和验证

3.3.5 发送与接收

根据以上的原理，发送与接收方代码如下：

```
def PGP_send(data, alice_pri_key, bob_pub_key):
    data += sign(data, alice_pri_key)
    des_key = generate_DES_key()
    data = DES_encrypt(data, des_key)
    des_key_enc = RSA_encrypt(des_key, bob_pub_key)  # 加密后的DES密钥
    data += des_key_enc
    data = Zip(data)
    return data


def PGP_recieve(data, alice_pub_key, bob_pri_key):
    data = UnZip(data)
    data, des_key_enc = data[:-128], data[-128:]
    des_key = RSA_decrypt(des_key_enc, bob_pri_key)
    data = DES_decrypt(data, des_key)
    data, signature = data[:-64], data[-64:]
    assert verify(data, signature, alice_pub_key)
    return data
```

图 3.6 发送与接收

其中 sign()函数表示对摘要值进行签名，用生成的 DES 密钥利用 DES 加密算法对数据进行加密。按照如上的流程，Zip()函数是定义的压缩函数，在传送之前将数据压缩，减小网络数据的传输量，降低网络负载。

3.4 实验结果

以下图显示接收方成功将接收到的数据进行解密且得到正确结果：

图 3.7 PGP 示意图

4 彩虹表实验

4.1 实验内容

了解和掌握彩虹表构造的基本原理和方法，能够设计和实现约化函数，针对特定的 hash 函数构造彩虹表，进行口令破解。

4.2 实验原理

假设我们有一个哈希方程 H 和一个有限的密码集合 P。定义一个归约函数 R 来影射散列值 h 在集合 P 中对应的密码 p。然后形成交替的密码和哈希值。

例如，如果 P 是 6 个字符的密码集合，而哈希值有 100 位长，那么他们形成的长链如下：

bq2zcg->2357038fb111e5f1a61961071b5451d20a127af0->68vyx0->b701c0c157beb999014cb0bb7c832d01e16a3918-> qicqj2->20dac81ae519d2755df7051c4f7626bcc43e0d1f -> qicqj2

为了生成查找表，从初始密码集合 P 中随机选择一个子集，对子集中的每个元素计算长度为 k 的哈希链，然后只保存每条哈希链的初始和末尾密码。把初始密码称为起点，把末尾密码称为终点。

在上述哈希链的例子中，‘bq2zcg’和‘qicqj2’即为起点和终点，而哈希链中的其他密码或者哈希值均不会被保存。

对于给定的哈希值 h 我们来计算它的逆（即找到对应的密码），从哈希值 h 开始，我们通过依次施加函数 R 和 H 生成一个哈希链。如果哈希链的任何节点与查找表的某个终点发生了碰撞，我们就可以找到与之对应的起点，然后用它重建对应的哈希链。而这个哈希链很可能包含了 h，如果这样的话，那么它的前驱节点即为我们要寻找的密码 p。

4.3 实验过程

彩虹表是一个用于破解加密过的密码散列的表，它通过用时间换空间的方法，折衷了暴力法和查找法破解 HASH 函数在空间和时间上的代价，使得破解 HASH 函数的时间和空间成本控制在可接受的范围内。彩虹表破解 HASH 函数的原理如下：

构造形式如下的 HASH 链：

$$aaaaaa \xrightarrow{H} 281DAF40 \xrightarrow{R} sgfnyd \xrightarrow{H} 920ECF10 \xrightarrow{R} kiebgt$$

其中 H 为待破解的 HASH 函数，例如本次实验中选择了 MD5；R 为自行设计的约减函数，它将字符串经过 HASH 函数得到的摘要值映射到事先约定的字符集中，得到特定字符的纯文本值。通过一个给定链头文本“aaaaaa”经过多次 R、H 操作得到链尾文本“kiebgt”，最终只需要保存这条链的链头和链尾文本值。若在破解时给定一 HASH 值，则依次假设其在链中的位置，并依次施加相应次数的 H、R 操作，直到计算的链尾值与该链的链尾值匹配，则说明在这条链中有相应的目标密码。例如给定 HASH 值 920ECF10，则依次假设其在链中的位置，当假设其为链中最后一个 HASH 值时，对其施加一次 R 函数计算得到“kiebgt”，与链尾值匹配，所以我们从对应的链头值“aaaaaa”施加相应次数 H、R 函数，直到得到 920ECF10，则说明目标密码为“sgfnyd”，当假设完所有位置后都不能与链尾值匹配，则说明目标密码不包含在这条链中，应选取其他链进行计算。

应该注意到，当对不同位置的摘要值都施加同一个 R 函数时，其碰撞概率较大，容易产生大量的重复链，导致 HASH 链的生成效率不高。所以应该对一条链中不位置的摘要值使用不同的 R 函

数，以减小碰撞的概率。此外可以避免添加链尾值相同的链，这样也可以大幅减小链中重复节点出现的概率，增加信息存储的密度。

在生成彩虹表时，选取了 7 位数字+小写字母的字符空间。链头是从“0000000”开始遍历至‘zzzzzzz’，但具体生成时并不需要完全遍历。HASH 函数选取了 MD5，R 函数是选取 MD5 值中确定的若干位经过简单计算映射到约定的字符空间中。实现时选取的链长为 1000，链长太短会导致占用过多存储空间，链长太长会增加计算量，降低生成效率。为了为了确保每一条链的链尾不与已生成的链的链尾重复，每一条链计算完成后将链头和链尾添加到一个动态维护的平衡二叉树中。此外为了进一步优化运行效率，HASH 链计算完成后并不直接写入文件，而是待生成一定数量的链后再集中写文件，这样可一定程度上降低频繁写文件的开销。

在进行解密操作时，先读取彩虹表文件，将每一条链按照链尾值进行排序，为了方便起见，仍采用了平衡二叉树完成排序，但显然这样的做法并不是最优的。对于每一条链，依次假设该 HASH 值在链中的位置，并施加相应次数 R、H 函数计算到链尾，并与该链尾比较，若相等则从链头倒推计算至假设位置，若非误报，则可确定该值前一字符值即为目标密码，若为误报，即倒推计算的 HASH 值与给定的 HASH 值不相等，则略过这次操作，若假设完所有位置后仍未匹配到链尾值，则说明这条链中没有目标密码，应选取另外的链进行操作，直到对文件中所有的链都进行完这样的操作。

4.4 实验结果

待程序运行约 25h 后生成了约 1.42GB 文件，共生成 88453000 条 HASH 链。一个成功破解的样例如下图所示

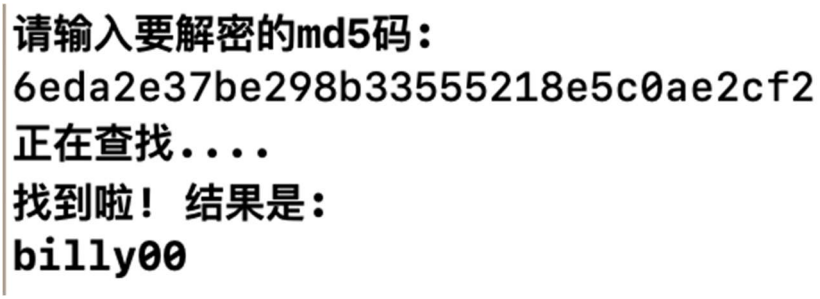

图 4.1 彩虹表实验

随机生成 1000 个 7 位待破解的样例字符串，对其求 MD5 摘要值，将其作为输入供彩虹表程序测试，共有 450 个样例正确被破解，破解成功的概率为 45%，在可接受的范围内。

尽管程序在功能上已经基本达到了要求，但是还是有很多问题需要解决。其中最明显的问题是求解时彩虹表的读取问题：时间和能力所限，在破解 MD5 函数之前将彩虹表中所有链的信息全部读到内存中进行查询，这样的做法虽然简单易行而且查询效率客观，但是显然这样做的开销是很大的——将生成的 1.42G 文件读取到内存中并构建一颗平衡二叉树要消耗近 6.5GB 内存，若要读取更大的彩虹表，这样的做法显然是不可取的。目前能想到的优化办法是对彩虹表文件建立索引，减小每次查询时对内存的占用。

5 实验总结

此次实验因为任务较多，所以耗时较长。考察了分组密码体制中的 SPN 密码，公钥密码体制中的 RSA，随机性检测，C 标准文件操作等等。

刚拿到任务时，就觉得比较繁重，因为考察的方面太多了。但是再细细捋一遍，发现按照任务顺序做，其实并不是很难。很多算法书上都有伪代码，直接翻译成 C 语言版本即可。

此次任务，难点还是有很多的，主要有以下几点：

1）SPN 的位运算

SPN 的位运算包括白化（按位异或）、分组 S 盒代替、P 盒置换三个方面，当时做的时候比较急躁，没有细致的按部就班。之后检查的时候只能每次转换都输出，然后对照书本看错在了哪步。

2）密码分析问题

SPN 的密码问题包括线性分析与差分分析，当时准备将书本上的内容看完看懂之后再实现，耗费了大量时间。但是后来直接按照书本算法的伪代码翻译即可，不用费太多的时间。最后依据原理，自己寻找了一条新的差分链分析出了 16 位的密码，取得的结果效率的确是很高，给我带来了很大的成就感。

3）安装 OpenSSL 库

在 windows 上安装 OpenSSL 很耗费时间，最后不得不转向 Linux 开发平台。

4）大数库的操作

OpenSSL 里的 BIGNUM 大数操作并不像想象中那么简单，在不能使用 RSA 方面的函数之后，就只能使用 BN 方面的基本运算函数，将各个算法（模重复平方、中国剩余定理、蒙哥马利算法）实现。

5）C 标准文件操作

因为要将明文分组加密，所以在读取文件时要一段一段读取并加密。在加密之后的解密也需要一段段的读取解密。这对指针的要求较高，而且每次加\解密文件耗费时间比较长，所以整体调试时间费时比较长。

6）彩虹表的构建

彩虹表由于数据量太大，检查错误起来十分耗费时间，大多时候只能静态分析代码逻辑。而且我生成彩虹表时，由于占用内存过大，使得系统将该进程杀死，最后不得不重新想办法进行优化。

总的来说，这次课设完成情况还是比较好的，大部分问题都是自己自行搜索解决的，部分实在不懂的细节在问了同学之后也迎刃而解。对密码学的认识，对 C 语言编程的认识又有了新的提高。对密码学的研究不再停止于理论上，而是真正投入实践了。