華中科技大学

课 程 设 计 报 告

题目：密码学原理课程设计

课程名称：密码学原理课程设计

专业班级：信息安全 1501 班

学　　号：U201514476

姓　　名：余雨田

指导教师：汤学明

报告日期：2017.10.1

计算机科学与技术学院

目 录

1 绪言

1.1 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）结合 RSA 和增强后的 SPN 实现文件（或通信）的加解密。

（8）构造彩虹表破解 hash 函数。

1.2 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。

（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。

2　原理、实现思路和算法实现

2.1　SPN 算法的实现、破解与增强

迭代密码是一种乘积密码体系。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……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 wr-1 进行变换，输出新的状态值 w^r，$g(w^{r-1},k^i)=w^r$；轮函数是单射函数，存在一个逆变换 g^{-1}，有 $g^{-1}(w^r,k^i)=w^{r-1}$。

SPN 密码体制

设 l,m,Nr 是正整数,$P=C=\{0,1\}^{lm}$

$K\subseteq(\{0,1\}^{lm})^{Nr+1}$ 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$(k^1,k^2,\ldots,k^{Nr+1})$

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

$$w=w_{<1>},w_{<2>},\ldots,w_{<m>},\ \text{其中}$$

$W_{<i>}=W_{(i-1)l+1},W_{(i-1)l+2},\ldots,W_{(i-1)l+1}$

2.1.1　SPN 加密算法

原始 SPN 使用 EBC 工作模式，每次从文件读取 2 字节共 16bits 使用算法 2.1 加密，加密结果为 2 字节共 16bits，写入到以”ab”二进制追加写入方式打开的文件，直到遇到明文文件尾为止。加密算法函数伪代码如下，主函数主要完成从文件中读取数据、加密、写入到文件中的操作，其算法流程图如图 2.1 所示。

算法 2.1 $SPN(x,\pi_s,\pi_p,(k^1,k^2,\ldots,k^{Nr+1}))$

$w^0=x$

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>= π_s(u^Nr<i>)          // 代替
}
y=v^Nr ⊕k^Nr+1          // 白化
return  y
```

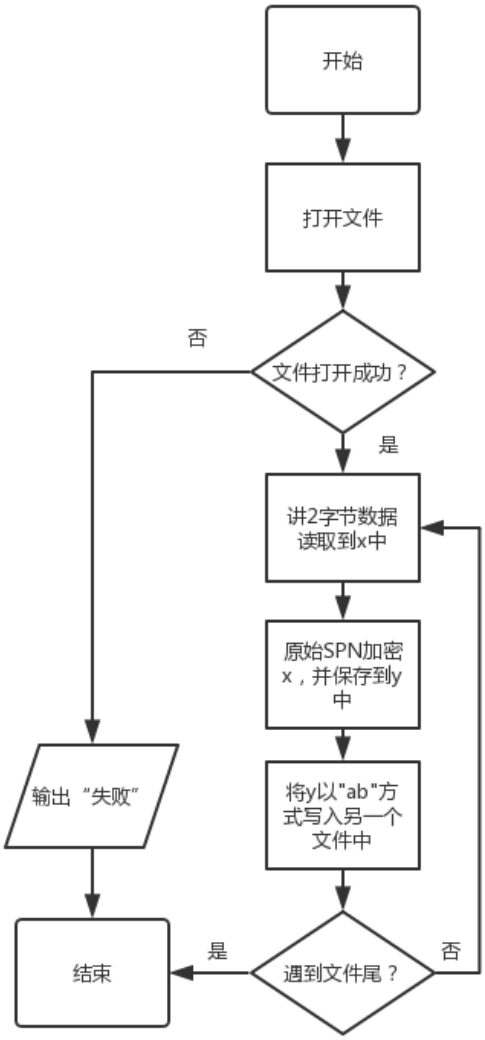

图 2.1　原始 SPN 主函数

原始 SPN 算法共计五轮加密操作，每一轮中包含异或、置换和代替三种加密方法。经过封装后形成函数，参数为明文和密钥，返回值为加密后的密文。

异或操作的 C 预言实现较为简单，只需要使用位运算符^即可。

密钥编排方案是每轮将密钥循环左移 4 位，得到该轮的轮密钥，这一方案使用一个宏定

义实现。使用 unsigned long 型变量存储得到的密钥，再将其利用宏定义作循环右移运算，将得到的 NUM_OF_ROLLKEY（宏定义）个轮密钥存储到 unsigned short 类型的数组当中，之后定义结构体数组 w、u、v，分别用于存储加密过程中的中间结果，根据 SPN 加密算法逐步实现加密过程即可。

置换和代替操作需要使用字段结构。分别定义两个字段结构，其单位大小均为 2 字节。分别拥有 16 个和 4 个成员变量，通过成员变量之间的交换和代替即可实现 S 盒和 P 盒的功能。例如下面的 pi_s 和 pi_p 函数，将 2 字节的字段结构体作为参数输入，在函数体内对字段结构的成员变量作交换操作和选择条件后的赋值操作，返回值同样为 2 字节的字段结构体。即可实现置换和代替两种加密方法。数据结构的具体定义如下所示。

```
struct WORD1
{
            unsigned int s1:4;
            unsigned int s2:4;
            unsigned int s3:4;
            unsigned int s4:4;
};

struct WORD2
{
            unsigned int p1:1;
            unsigned int p2:1;
            unsigned int p3:1;
            unsigned int p4:1;
            unsigned int p5:1;
            unsigned int p6:1;
            unsigned int p7:1;
            unsigned int p8:1;
            unsigned int p9:1;
            unsigned int p10:1;
            unsigned int p11:1;
            unsigned int p12:1;
            unsigned int p13:1;
            unsigned int p14:1;
            unsigned int p15:1;
```

```
            unsigned int p16:1;
};

struct WORD2 pi_s(struct WORD1);/*代替密码*/
struct WORD2 pi_p(struct WORD2);/*置换密码*/
```

2.1.2 SPN 密码的线性分析

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。

如果 S 盒设计失当，可以得出密文比特和明文比特的线性关系式：

$$y_j = x_i \oplus k^1_{m_1} \oplus k^2_{m_2} \oplus k^3_{m_3} \oplus k^4_{m_4} \oplus k^5_{m_5} \text{ 或}$$

$$x_i \oplus y_j = k^1_{m_1} \oplus k^2_{m_2} \oplus k^3_{m_3} \oplus k^4_{m_4} \oplus k^5_{m_5}$$

这样一来，无需破解密钥 k，只需要一对明-密文对，即可知道所有 xi 和 yj 的线性关系。

输入比特和输出比特或他们的部分位不存在线性关系，否则可能存在下式的运算结果始终为 0 或 1：

$$x_{i_1} \oplus x_{i_2} \oplus \ldots \oplus x_{i_k} \oplus y_{j_1} \oplus y_{j_2} \oplus \ldots \oplus y_{j_1}$$

如果 S 盒的非线性特性不够理想，上式的值可能的概率为 0 或 1。

总结以上思路，可以设计以下算法：

收集尽可能多的在同一未知密钥 k 加密的 T 对明-密文对，用 T 表示明-密文对的集合(|T|=T)，目标是获得候选子密钥(k5<2>,k5<4>)。

每个候选子密钥分配一个计数器，初始值为 0。

对每对明-密文，尝试所有可能的候选子密钥，计算出随机变量 $x_5 \oplus x_7 \oplus x_8 \oplus u^4_6 \oplus u^4_8 \oplus u^4_{14} \oplus u^4_{16}$ 的结果，如果结果为 0，则相应的计数器加 1。

T 对明-密文尝试完毕后，真子密钥对应的计数值最接近 T/2±T/32，其他则接近 T/2。

T 越大，结果越准确。

算法的描述如下：

```
算法 2.2 SPN 线性分析
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,y) ∈ T  {
    for (L1,L2)=(0,0) to (F,F)  {
        v^4_<2> = L1 ⊕ y<2>
        v^4_<4> = L2 ⊕ y<4>
        u^4_<2> = π_s^-1(v^4_<2>)
        u^4_<4> = π_s^-1(v^4_<4>)
        z = x5 ⊕ x7 ⊕ x8 ⊕ u^4_6 ⊕ u^4_8 ⊕ u^4_14 ⊕ u^4_16   // 计算随机变量值
        if  z=0  {
            Count[L1,L2] ++;
        }
    }
}
max = -1
for (L1,L2)=(0,0) to (F,F)  {
Count[L1,L2] = | Count[L1,L2] - T/2 |
if  Count[L1,L2] > max  {
    max = Count[L1,L2]
    maxkey = (L1,L2)
}
// maxkey 即为所求子密钥
```

通过线性分析，应当可以分析出 32 位密钥中的 8 位，具体到本 SPN，即为第 0-3 位和第 8-11 位（从低位计起）。对于剩余的 24bits 密钥，使用暴力破解法求解。理论上说，通过线性分析，密钥破解的时间缩短了 2^8 倍。

线性分析需要 8000 对左右的明密文对。为了生成这些明密文对，首先循环调用 8000 次伪随机数生成函数，生成 8000 个长度为 2 字节的数，之后用原始 SPN 加密算法对这些数加密，得到密文，将这些明密文以指定格式存放到文件当中，在进行线性分析时，再将这些明密文对从文件中读取出来使用。

2.1.3 SPN 密码的差分分析

差分分析是一种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。是一种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。

其算法的思路是：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

收集尽可能多的在同一未知密钥 k 加密的 T 个四重组(x,x*,y,y*) ，用 T 表示四重组的集合(|T|=T)，目标是获得候选子密钥(k5<2>,k5<4>)。

每个候选子密钥分配一个计数器，初始值为 0。

对每对明-密文，尝试所有可能的候选子密钥，计算出 u4’，如果 u4’=0000011000000110 则相应的计数器加 1。

T 对明-密文尝试完毕后，真子密钥对应的计数值最大

T 越大，结果越准确。

算法 2.3 差分攻击(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 $k^5_{\langle 2\rangle}$和 $k^5_{\langle 4\rangle}$

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 $(x,x*,y,y*) \in T$ {

if $(y_{\langle 1\rangle}=y*_{\langle 1\rangle}$ and $y_{\langle 3\rangle}=y*_{\langle 3\rangle})$ { // 只考虑 $y'_{\langle 1\rangle}$和 $y'_{\langle 3\rangle}=0$

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

$v^4_{\langle 2\rangle} = L_1 \oplus y_{\langle 2\rangle}$

$v^4_{\langle 4\rangle} = L_2 \oplus y_{\langle 4\rangle}$

$u^4_{\langle 2\rangle} = \pi_s^{-1}(v^4_{\langle 2\rangle})$

$u^4_{\langle 4\rangle} = \pi_s^{-1}(v^4_{\langle 4\rangle})$

$(v^4_{\langle 2\rangle})* = L_1 \oplus (y_{\langle 2\rangle})*$

$(v^4_{\langle 4\rangle})* = L_2 \oplus (y_{\langle 4\rangle})*$

$(u^4_{\langle 2\rangle})* = \pi_s^{-1}((v^4_{\langle 2\rangle})*)$

$(u^4_{\langle 4\rangle})* = \pi_s^{-1}((v^4_{\langle 4\rangle})*)$

$(u^4_{\langle 2\rangle})' = u^4_{\langle 2\rangle} \oplus (u^4_{\langle 2\rangle})*$

$(u^4_{\langle 4\rangle})' = u^4_{\langle 4\rangle} \oplus (u^4_{\langle 4\rangle})*$

if $(u^4_{\langle 2\rangle})'$=0110 and $(u^4_{\langle 4\rangle})'$ = 0110 {

Count[L_1,L_2] ++;

}

}

}

}

max = -1

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

```
if  Count[L1,L2] > max  {
        max = Count[L1,L2]
        maxkey = (L1,L2)  }
}
// maxkey 即为所求子密钥
```

本差分分析需要 80 对左右的明密文对（每对包含一对 x 和一对 y），为了生成这些明密文对，首先循环调用 80 次伪随机数生成函数，生成 80 个长度为 2 字节的数，再用输入异或值 x'=1011 对它们作异或运算得到另外的 80 个明文。之后用原始 SPN 加密算法对这些数加密，得到密文，将这些明密文以指定格式存放到文件当中，在进行差分分析时，再将这些明密文对从文件中读取出来使用。

2.1.4 基于密码分析的破解

2.1.2 和 2.1.3 中介绍的线性分析和差分分析算法均只能分析出 32 位密钥中的 8 位，具体到本 SPN，即为第 0-3 位和第 8-11 位（从低位计起）。由于密钥的其他各位不再具有显著的偏差值，因此这两种分析算法不再适用。对于剩余的 24 位密钥，采用暴力破解的办法，即在确定 8bits 值的情况下，对其他所有密钥作遍历，从 0x0000060f 至 0xfffff6ff（跳过所有子密钥位 6f 的密钥值），尝试使用它们对明文作加密运算，将所得的密文值与正确的密文值作比较，如果相同即可判断此密钥可能正确密钥。

在实际操作中可以发现，如果只使用一对明密文作检验，得到的很多密钥对其他的明密文对作检验是无效的。因此可以再添加若干对明密文对，只有当被检测的密钥对多个不同明文均可做出正确的加密结果时，方判定通过检测，由于伪密钥几乎没有概率通过多次不同明文的检测，所以可以判定该结果为正确的密钥。

2.1.5 SPN 算法安全性的增强

下面介绍的增强 SPN 算法主要从以下几个方面增加了算法的安全性。

1、 分组宽度。宽度由原先的 16 位增加到 32 位，同时，S 盒也从 4 位扩充到 8 位，P 盒由 16 位扩充到 32 位。
2、 密钥长度。随着分组宽度的增加，密钥长度也同样增加，由原先的 32 位增加到 64 位。

3、 加密轮数。由 5 轮更为 9 轮。

4、 S 盒与 P 盒。使用 DES 加密算法的置换以及 AES 加密算法代替盒。

5、 工作模式。由 ECB 变更为更加安全的 CBC 模式。

6、 短块的填充。使用 PKCS7 填充方法对结尾不足 4 字节的数据作填充处理。

为了实现分组长度和 S、P 盒子的扩展，需要改进原先的数据结构，S 盒使用一个数组存储，而字段结构则扩充一倍。需要注意的是，这里的结构体中成员变量虽然仍然为 4bits，但在函数体内运算时，会将相邻的两个成员变量通过位运算组合到一个 char 型变量中再作为参数带入到 Sbox 求出代替加密后的加密值。

```
char Sbox[] =
{
            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
};

struct WORD1
{
              unsigned int s1:4;
              unsigned int s2:4;
              unsigned int s3:4;
              unsigned int s4:4;
              unsigned int s5:4;
              unsigned int s6:4;
              unsigned int s7:4;
              unsigned int s8:4;
};

struct WORD2
{
              unsigned int p1:1;
              unsigned int p2:1;
              unsigned int p3:1;
              unsigned int p4:1;
              unsigned int p5:1;
              unsigned int p6:1;
              unsigned int p7:1;
              unsigned int p8:1;
              unsigned int p9:1;
              unsigned int p10:1;
```

```
            unsigned int p11:1;
            unsigned int p12:1;
            unsigned int p13:1;
            unsigned int p14:1;
            unsigned int p15:1;
            unsigned int p16:1;
            unsigned int p17:1;
            unsigned int p18:1;
            unsigned int p19:1;
            unsigned int p20:1;
            unsigned int p21:1;
            unsigned int p22:1;
            unsigned int p23:1;
            unsigned int p24:1;
            unsigned int p25:1;
            unsigned int p26:1;
            unsigned int p27:1;
            unsigned int p28:1;
            unsigned int p29:1;
            unsigned int p30:1;
            unsigned int p31:1;
            unsigned int p32:1;
};

struct WORD2 pi_s(struct WORD1);/*代替密码*/
struct WORD2 pi_p(struct WORD2);/*置换密码*/
```

密钥长度的增加意味着需要使用新的宏定义来定义循环移位：

#define ROTATE_LEFT(x, s, n) ((x) >> (n)) | ((x) << ((s) - (n)))/*循环左移*/

#define ROTATE_RIGHT(x, s, n) ((x) >> (n)) | ((x) << ((s) - (n)))/*循环右移*/

加密轮数已经实现使用宏定义定义好，只需要将宏定义改为 9 即可。

工作模式调整需要定义一个临时变量 iv。iv 设定了初始异或值 0x0，每次加密之前，明文先与上一轮的 y 值异或，再行加密算法，结果用 y 写入。

```
    unsigned long x=0;
    unsigned long y=0;
    int filesize=file_size(FILENAME);//求文件长度
    int size=0;
    unsigned long iv=0x0;
    while(size<filesize-4)
    {
        fread(&x, sizeof(unsigned long), WIDTH, fp);
        if(size==0)
            x = x ^ iv;
        else
            x = x ^ y;
        y = encode(x,key);//SPN 加密算法
        fwrite(&y, sizeof(unsigned long), WIDTH, fp1);
        size=size+4;
    }
```

填充算法使用 PKCS7，在求出文件长度后，先判断其是否为 4 的倍数，如果是，则直接在文件最后追加 0x00000000 的加密结果。

如果不是，则要进一步判断余数。对于余数 1，最后 3 个字节左移 8 位后和 0x01 相或，再送入 encode 函数加密得到结果；对于余数 2，最后 2 个字节左移 16 位后和 0x0202 相或，再送入 encode 函数加密得到结果；对于余数 3，最后 1 个字节左移 24 位后和 0x030303 相或，再送入 encode 函数加密得到结果。最终将结果写入到文件结尾，加密过程结束。代码具体实现如下所示。

```
    switch(filesize%4)//PKCS7
    {
    case 0: x=0x00000000;x = x ^ y;y = encode(x,key);fwrite(&y, 4, 1, fp1); break;
    case 1: x=x&0xff000000|0x00030303;x = x ^ y;y = encode(x,key);fwrite(&y, 4, 1,
fp1); break;
    case 2: x=x&0xffff0000|0x00000202;x = x ^ y;y = encode(x,key);fwrite(&y, 4, 1,
fp1); break;
    case 3: x=x&0xffffff00|0x00000001;x = x ^ y;y = encode(x,key);fwrite(&y, 4, 1,
fp1); break;
```

```
}
```

解密过程即加密过程的逆向实现。需要修改的是：将 P 盒和 S 盒逆序存放得到逆 P 盒与逆 S 盒；短块填充改为取最后四个字节，识别末尾字节的值，据值判断加密时填充了几位。

2.2 公钥加密体系 RSA 的实现

RSA 是目前最有影响力和最常用的公钥加密算法，它能够抵抗到目前为止已知的绝大多数密码攻击，已被 ISO 推荐为公钥数据加密标准。

今天只有短的 RSA 钥匙才可能被强力方式解破。到 2008 年为止，世界上还没有任何可靠的攻击 RSA 算法的方式。只要其钥匙的长度足够长，用 RSA 加密的信息实际上是不能被解破的。

C 语言标准不支持长达 1024bits 的变量数据及其运算，为此需要引入外部库。这次课程设计使用的是 gmp 大数库，这一大数库支持任意长度的整数的加减乘除模幂等运算，利用该库可以实现 RSA 算法。

2.2.1 RSA 算法参数的生成

RSA 生成算法如下所示。

算法 2.4 RSA 参数生成
选取两个大素数 p 和 q，计算 n=pq。
随机选取加密密钥 e，使 e 和(p-1)(q-1)互素。
计算解密密钥 d，使 ed≡1 mod(p-1)(q-1)
　　即 ed=k×(p-1)(q-1)＋1
返回 e、n、d。

e 和 n 是公开密钥，放在一个公共目录供大家访问，d 是私人密钥，p 和 q 不再需要，密钥生成后可抹去，但决不能泄漏。加密过程需要的是 e、n，解密过程需要的是 d、n，但为了之后使用中国剩余定理、蒙哥马利算法等加速解密过程，需要保留 p、q 两个参数。

首先生成两个大素数。使用 gmp 函数 mpz_urandomb 生成两个长度为 1024bits 的整数，将其分别赋给变量 key_p、key_q，利用 mpz_even_p 函数判断其是否为奇数，如果是偶数，自增 1，使之成为奇数。接下来使用 mpz_probab_prime_p 检测其是否为素数。这个函数使

用了 miller-rabin 素性检测算法检测素数，每个数检测 25 遍，使得误报率降低到一个不可能出现的极小概率。如果检测出非素数，则自增 2 后再次检测，重复这一步骤直到得到两个素数为止。这样，两个 1024bits 大素数得以生成。

初始化并设置 key_e 为 65537，由于 e 为素数，所以 e 和(p-1)(q-1)互素恒成立。

利用 mpz_invert 函数，求出 key_d，使得 ed≡1 mod(p-1)(q-1)。

2.2.2 快速实现 RSA 加解密

给定明文 m、公钥 e，可作以下计算，以实现加密。

将它分成比 n 小的数据分组，对每个分组 $m_i<n$ 加密，得到密文 c_i：

$$c_i=m_i^e \bmod n$$

给定密文 c、私钥 d，可作以下计算，以实现解密。

对每个密文分组 c_i 计算，得到明文 m_i：

$$m_i=c_i^d \bmod n=m_i^{ed} \bmod n=m_i^{k(p-1)(q-1)+1} \bmod n$$

$$=m_i*m_i^{k(p-1)(q-1)} \bmod n=m_i$$

这个过程中的模幂运算和模乘运算较为费时，可以使用若干算法加速这些运算过程。模重复平方算法可以加速运模幂运算，蒙哥马利算法可以加速模乘运算，中国剩余定理可以在特定条件下加速模幂运算。

在模算术计算中，常常要对大整数 m 和 n，计算 b^n (mod m). 如果用递归，则可以得到：

b^n (mod m) = b^n % m

= b*b^(n-1) % m .

根据模的运算规则，可以进一步得到：

b*b^(n-1) % m = b*(b^(n-1) % m) % m

= b* (b^(n-1) (mod m)) (mod m).

所以有：

b^n (mod m) = b* (b^(n-1) (mod m)) (mod m).

得到 b^n (mod m)的递归公式为：

func(b, n, m) = b* func(b, n-1, m) % m

然而，模重复平方算法是一个递归算法，在实际应用中可能会存在递归层次过深的问题，同时需要执行 n-1 次乘法，所以要改进为非递归实现。模重复平方算法的一个变种算法叫做平方模算法，时间复杂度 O(log n)。这一算法的伪代码表述为：

算法 2.5 平方-模算法 Square-Multiply(a, b, n)

```
z=1
for i=l-1 downto 0 {
        z = z2 mod n
        if   bi=1 {
        z = (z×a) mod n
        }
}
return z
```

在 RSA 加密算法中需要计算 $c_i=m_i^e \bmod n$，这一模幂运算即可使用平方-模算法加速。

对于乘模运算 A*B%N，如果 A、B 都是 1024 位的大数，先计算 A*B，再% N，就会产生 2048 位的中间结果，如果不采用动态内存分配技术就必须将大数定义中的数组空间增加一倍，这样会造成大量的浪费，因为在绝大多数情况下不会用到那额外的一倍空间，而采用动态内存分配技术会使大数存储失去连续性而使运算过程中的循环操作变得非常繁琐。所以模乘运算的首要原则就是要避免直接计算 A*B。

蒙哥马利模乘实际上是解决了这样一个问题，即不使用除法（用移位操作）而求得模乘运算的结果。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。不难发现，模乘过程中复杂度最高的环节是求模运算，因为一次除法实际上包含了多次加法、减法和乘法，如果在算法中能够尽量减少除法甚至避免除法，则算法的效率会大大提高。

算法 2.6 求 C’=A*B*2**(-k)%N 即找到一个与 A*B%N 同余的一个数 H*2**K，以计算 H%N

```
C'=0
FOR i FROM 0 TO k
    C'=C'+A*B
    C'=C'+C'[0]*N
    C'=C'/2
IF C'>=N
    C'=C'-N
RETURN C'
```

既然 C'总是小于 2N，所以求 C' %N 就可以很简单地在结束循环后用一次减法来完成，即在求 A*B*2**(-k) %N 的过程中不用反复求模，达到了我们避免做除法的目的。当然，这一算法求得的是 A*B*2**(-k) %N，而不是我们最初需要的 A*B %N。但是利用 A*B*2**(-k)我们同样可以求得 A**E %N。

```
算法 2.7 蒙哥马利算法
    m=r-N[0]'
    C'=0
    FOR i FROM 0 TO k
        q=(C'[0]+A*B[0])*m %r
        C'=(C'+A*B+q*N)/r
    IF C'>=N
       C'=C'-N
     RETURN C'
```

RSA 解密者负责生成参数 p、q 以及公私钥，因为 n=pq，因此解密时的模幂运算可以转换成两个同余方程求解。即

$$c_i=m_i^e \bmod n$$

等价于

$$x=m_i^e \bmod p$$

$$x=m_i^e \bmod q$$

利用中国剩余定理可以解以上方程组，加速模幂运算。

2.3 结合 RSA 和增强 SPN 实现通信加密

通过之前的工作，已经实现了增强 SPN 的加解密以及 RSA 公钥加密体系，将这两者简单集合可以实现简单的加密 Socket 通信程序。

该 Socket 通信程序的主体部分为 C/S 结构，即程序分为客户机和服务器两个程序。启动服务器程序后，双方会进行一个简单的秘钥交换握手协议通信。程序将生成 RSA 密钥，并将公钥保存到公共目录当中。之后启动客户机程序，程序将生成增强 SPN 的 64 位密钥，并从公共目录中取出 RSA 公钥，用 RSA 密钥加密增强 SPN 密钥并发送给服务器程序。服务器程序使用公钥解密，得到 SPN 的密钥。至此，通信加密过程的密钥交换工作完成。之后即可传送使用 SPN 加密的文件了。整个过程如图所示。

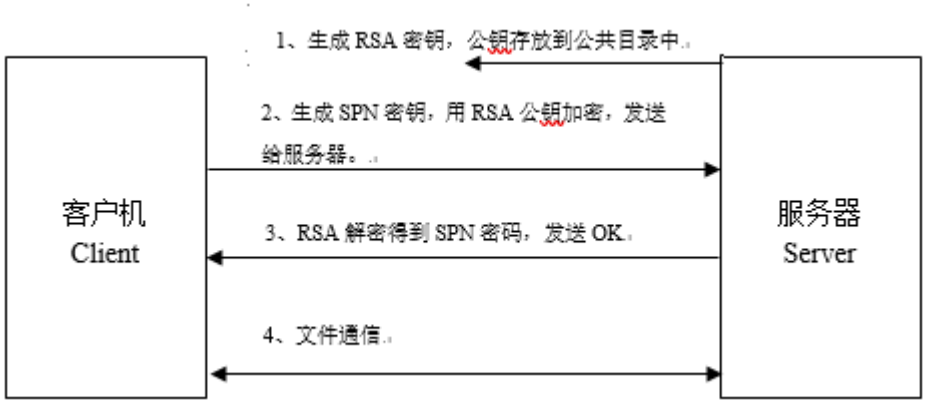

图 2.2　加密通信模型

通信严格遵守 Socket 通信模型，即步骤分为：创建 ServerSocket 和 Socket、打开连接到的 Socket 的输入/输出流、按照协议对 Socket 进行读/写操作、关闭输入输出流以及 Socket。

课程设计中为了方便测试起见，服务器端亦架设在本机（127.0.0.1）上，端口号为 4000，即客户机将尝试连接 127.0.0.1:4000。

2.4　利用彩虹表破解哈希函数

2.4.1 原理

本次课程涉及计划实现对 MD5 哈希函数和 LM-hash 函数的破解。

MD5 算法是一种广泛使用的散列函数产生一个 128 位的散列值。

LM-hash 是 Windows 系统下的 hash 密码格式。

假设我们有一个密码散列函数 H 和一组有限密码 P.目的是预先计算一个数据结构，给定任何哈希函数的输出 h 可以在 P 中定位一个元素 p，使得 H（p）= h，或确定在 P 中没有这样的 p。最简单的方法是对 P 中的所有 p 计算 H（p），然后存储表需要 Θ（| P | n）个空间，其中 n 是 H 的输出的大小，对于大| P |是禁止的。

哈希链是减少这种空间需求的技术。这个想法是定义将散列值映射到 P 值的缩减函数 R. 然而，减小函数实际上并不是散列函数的倒数。通过将哈希函数与缩减函数交替，形成交替密码和哈希值的链。

减少功能的唯一要求是能够返回特定大小的“纯文本”值。

为了生成表，我们从 P 中选择一组随机的初始密码，每一个固定长度为 k 的计算链，并且只存储每个链中的第一个和最后一个密码。第一个密码叫做起点，最后一个叫做端点。在上面的示例链中，“aaaaaa”将是起点，“kiebgt”将是端点，并且不存储其他任何密码（或哈希值）。

现在，给定一个哈希值 h，我们要反转（找到相应的密码），通过应用 R，然后 H，然

后 R 等计算从 h 开始的链。如果在任何时候我们观察到与表中的一个端点匹配的值，我们得到相应的起始点并使用它重新创建链。这个链很有可能包含值 h，如果是，链中的前一个值就是我们寻求的密码 p。

表内容不依赖于要颠倒的哈希值。它创建一次，然后重复用于未修改的查找。增加链的长度减小了表的大小。它还增加了执行查找所需的时间，这是彩虹表的时间记忆权衡。在一个单项链的简单情况下，查找非常快，但是表格非常大。一旦链条变长，查找速度就会减慢，但是桌面大小就会下降。

简单的哈希链有几个缺陷。最严重的是，如果在任何时候两个链子碰撞（产生相同的值），它们将合并，因此尽管已经付出了相同的计算成本来生成表，但是表不会覆盖尽可能多的密码。因为以前的链没有完整地存储，所以这是不可能有效地检测的。例如，如果链 3 中的第三个值与链 7 中的第二个值相匹配，则两个链将覆盖几乎相同的值序列，但是它们的最终值将不相同。哈希函数 H 不太可能产生冲突，因为它通常被认为是不这样做的重要的安全特征，但由于需要正确地覆盖可能的明文，减少函数 R 不能抵抗碰撞。

其他困难是因为选择 R 的正确功能的重要性。选择 R 作为身份比一种强力方法好一些。只有当攻击者了解可能的明文是什么，他或她可以选择一个功能 R，确保时间和空间只能用于可能的明文，而不是可能的密码的整个空间。实际上，R 将先前哈希计算的结果传递给可能的明文，但是这个好处带来了这样的缺点：R 可能不会在类中产生每一个可能的明文，攻击者希望检查否认攻击者没有密码来自他的选课 此外，设计函数 R 可能难以匹配明文的预期分布。

彩虹表通过用相关的缩减函数 R 1 至 R k 的序列替换单个缩减函数 R，有效地解决了与普通散列链冲突的问题。这样，对于两条链来进行碰撞和合并，它们必须在相同的迭代中达到相同的值。因此，每个链中的最终值将是相同的。最终的后处理通行证可以对表中的链进行排序，并删除与其他链具有相同最终值的任何“重复”链。然后生成新的链以填写表。这些链不是无碰撞的（它们可能会短暂地重叠），但是它们不会合并，从而大大减少了整体的碰撞次数。

2.4.2 具体实现

这一部分的程序分成两个部分，一个部分用于生成彩虹表，另一个部分用于利用先前生成的彩虹表逆推明文。

首先介绍生成彩虹表的程序。该程序无输入值，输出的是一个存放在文件中的彩虹表。在宏定义中，首先定义彩虹表的基本属性。指定彩虹表链数为 1000，字符库中总计有 36 个字符，破解密码长度为 6，粗略计算大约有 21 亿种密码组合，在最理想的情况（不存在任何碰撞的链）下，彩虹表的链数应有 220 万条左右，为保证碰撞的成功率，计划生成总计约大小的彩虹表。

```
#define LEN 6//密码最大长度
#define SRC "0123456789abcdefghijklmnopqrstuvwxyz"//密码字符库
#define NUM_OF_R 1000//链的长度
```

1、生成明文可以采用顺序生成和随机生成的办法，这里采用 rand()伪随机数生成的办法。利用 rand()生成 6 个 0-35 间的伪随机整数，将他们作为 SRC 的索引，即可得到 6 个字符库中的字符，组成一个初始的 6 位明文。

2、对这 6 位密码进行 MD5 哈希操作，对返回值动用 R 函数操作。重复这一操作 1000 次，得到经过 1000 轮 H-R 变换后的 6 位字符串。

3、将这初始的明文和生成的 6 位字符串以”%s %s\n”的格式存放到指定文件中。

重复以上 3 步若干次，直到生成的彩虹表大小满足要求。即可得到 MD5 的彩虹表。

将 MD5 哈希函数换为 LM-hash 函数，重复以上所有步骤，即可得到 LM-hash 函数的彩虹表。

再来介绍利用先前生成的彩虹表逆推明文的程序。这一程序首先接受用户输入的一个 hash 值，利用先前生成的彩虹表反查明文，如果找到，则返回该明文，否则返回失败信息。

收到用户的 hash 值后，读取某条链的信息，假设用户的 hash 值在这条链第 n 个位置，则对用户 hash 值作一次 R 处理，再作 n-1 次 H-R 处理后，应当得到与链尾相同的字符串。

利用这个思路，依次读取所有链的信息，每次读取后将 n 从 1 遍历到 1000（链长），在每次遍历中，对用户 hash 值作一次 R 处理，再作 n-1 次 H-R 处理，得到的字符串与彩虹表链尾字符串比较。如果相同，则返回该明文，说明查找成功；如果在比较了所有的链之后仍未返回成功，则判定失败。

R 函数使用了移位的思想，目标与哈希函数基本相似，即尽量保证高的碰撞稳定性。

```
void reductionfunction(char *encrypt,    int *decrypt,    int k,    int length)
{
    int i;
    char x, y;
    k = k%SIZE;
    int q = strlen(SRC);
    for(i = 0; i<length; i++){
        x = decrypt[i] ^ decrypt[(i+1)&0xf] ^ decrypt[(i+2)&0xf] ^ decrypt[(i + 3) & 0xf];
        y = decrypt[k&0xf] + decrypt[(k-1)&0xf] +decrypt[(k-2)&0xf] + decrypt[(k - 3) & 0xf];
        encrypt[i] = SRC[(x+y)%q];
        k = k + k + k + 1;
```

```
    }
    encrypt[length] = '\0';
}
```

2.4.3 改进

上述彩虹表模型在实际使用时发现存在问题。由于彩虹表的链数十分庞大，多达数百万条，而每次查询都要从头逐个遍历，多次从磁盘读取数据将耗费大量时间，时间成本上无法承受，必须寻找能够快速查找的数据结构。

经过筛选尝试，确定使用哈希表的数据结构来存储彩虹表。即每次生成一条链后，根据链尾字符串的特征，将其存放到指定的文件当中，并且建立文件的索引。在查询时，每次生成一个查询目标，即根据索引找寻对应的文件查询。

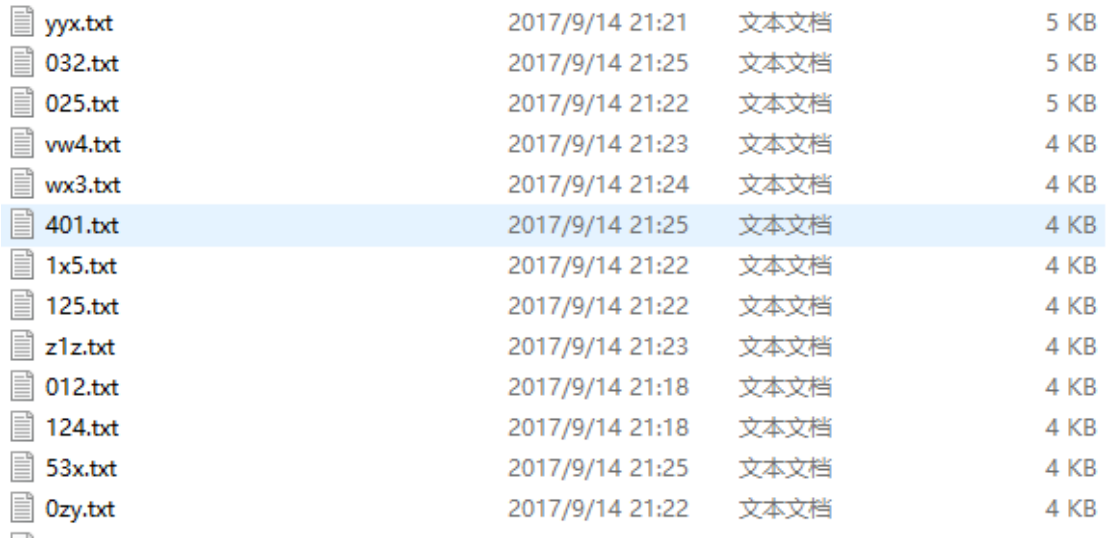

图 2.3　哈希表存储的彩虹表（局部）

以链尾字符串的前三个字符为索引建立哈希表，并且直接以这三个字符作为文件名。如图 2.3 所示。这样做将建立 36**3=46656 个索引文件，查询效率提高约 46656 倍。

图 2.4 显示的是彩虹表的部分文件，文件名即为链尾字符的前三个字符。图 2.5 展示了名为 yyx.txt（即索引为 yyx）的文件打开后内部存储情况。

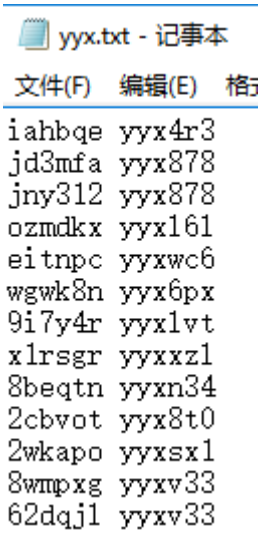

图 2.4　打开一个索引后的文件内容（局部）

3 测试

3.1 程序功能介绍

1.cpp 原始 SPN 加密演示程序：从 x.dat 读入，写到 y.dat。
1.3.cpp 原始 SPN 解密演示程序：从 y.dat 读入，写到 xp.dat。
3.cpp 原始 SPN 加密（文件）。注意：要在删除 y.dat 文件后执行此操作。
3.1.cpp 原始 SPN 解密（文件）。
1.1.cpp 线性分析准备程序：生成大量明文，并且存放到 1.1.dat 中，明文间间隔一个换行。
1.2.cpp 线性分析准备程序：根据 1.1.dat 生成密文 1.2.dat。
2.1.cpp 线性分析：根据 1.2.dat 线性分析得到密钥。
2.2.cpp 差分分析准备程序：根据 1.1.dat 生成明密文对 2.1.dat。
2.3.cpp 差分分析：根据 2.1.dat 差分分析得到密钥。
2.4.cpp 暴力破解密钥。
2.5.cpp 用子密钥 6f 暴力破解密钥。
3.6.cpp 增强 SPN 加密：密钥长度、分组、S 盒、P 盒、轮数等。
3.7.cpp 增强 SPN 解密。

以下文件保存在../MinGW/bin 文件夹路径中。
5.cpp 生成 RSA 的参数，保存到 private.dat 中，公钥保存到 public.dat 中。
6.cpp 快速实现 RSA 的加密：模重复平方与 gmp 模幂结合。加密明文 6.1.dat（十六进制），密文保存到 6.2.dat 中。
6_.cpp 快速实现 RSA 的加密：蒙哥马利模幂。
6.1.cpp 快速实现 RSA 的解密：中国剩余定理+模重复平方+蒙哥马利。解密密文 6.2.dat（十六进制），明文保存到 6.3.dat 中。
myclient.cpp 客户机：结合 RSA 和增强后的 SPN 实现 socket 通信的加解密。
myserver.cpp 服务器：结合 RSA 和增强后的 SPN 实现 socket 通信的加解密。

以下文件保存在../rainbow_newR_1000 文件夹路径中。
rainbow_produce_newR_1000.cpp MD5 彩虹表生成。
rainbow_attack_newR_1000.cpp MD5 彩虹表破解。

3.2 随机性测试

随机性测试结果

C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	P-VALUE	PROPORTION	STATISTICAL TEST
4	2	1	5	8	7	3	4	4	4	0.350485	1.0000	Frequency
4	4	2	3	4	4	5	7	4	5	0.911413	0.9762	Block-Frequency
3	3	4	1	7	4	6	5	2	7	0.392456	1.0000	Cusum
3	2	6	4	10	2	3	4	5	3	0.162606	1.0000	Cusum
2	3	5	5	5	4	5	6	4	3	0.941144	1.0000	Runs
6	4	3	5	1	4	5	4	5	5	0.875539	0.9762	Long-Run
6	5	4	6	4	1	4	4	3	5	0.834308	0.9762	Rank
3	5	6	8	1	4	1	5	2	7	0.141256	1.0000	FFT
5	2	9	5	2	5	1	0	5	8	0.021262	1.0000	Aperiodic-Template
2	3	5	3	4	6	8	3	5	3	0.585209	1.0000	Aperiodic-Template
3	3	5	3	2	4	9	3	4	6	0.392456	0.9762	Aperiodic-Template
2	4	1	8	6	3	4	6	4	4	0.392456	1.0000	Aperiodic-Template

图 3.1　随机性测试（局部）

4 总结与展望

课程设计主要完成了如下工作：

（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。

（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。

参考文献

[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