

密码学lab2

1813540 陈鸿运

2020 年 12 月 1 日

目录

1	DES算法实现	3
1.1	DES加密算法大致流程	3
1.2	初始置换IP	4
1.3	轮结构	5
1.3.1	E 表	6
1.3.2	S 盒	6
1.3.3	P 表	7
1.4	函数 $F(R, K)$	8
1.5	尾置换 IP^{-1}	8
1.6	子密钥的产生	9
2	DES运行结果	10
3	雪崩效应	13

1 DES算法实现

DES算法的加密过程主要由四个部分完成：

- 1、初始置换 IP ；
- 2、获取子密钥 K_i ；
- 3、密码函数 F ；
- 4、尾置换 IP^{-1} ；

下面我们逐一分析其过程：

1.1 DES加密算法大致流程

DES的密钥有64比特，其中以8为倍数的比特位作为校验位，因此其有效长度为56位；DES将56比特的密钥进行置换选择后进行十六轮循环移位，得到十六个子密钥分别用于每一轮的加密过程；DES将明文分为64比特一组，分别对每组进行加密；

图1是DES加密的大致过程*：

- 1、DES首先对64位明文依照初始置换表 IP 进行
- 2、初始置换：置换完毕后，其将64位分为左32位与右32位；
- 3、在随后的每一轮加密中，这一轮的右32位作为下一轮的左32位；
- 4、这一轮的右32位在密码函数 F 的作用下，与该轮对应的子密钥相作用，得到一个32位中间结果，该结果再与这一轮的左32位进行异或操作，得到下一轮的右32位；这一过程持续十六轮；
- 5、十六轮加密结束后，将得到的左32位数据与右32位数据左右交换，得到64位数据；
- 6、这64位数据再进行一次逆初始置换，得到最终的64位密文；

*图片源自网络

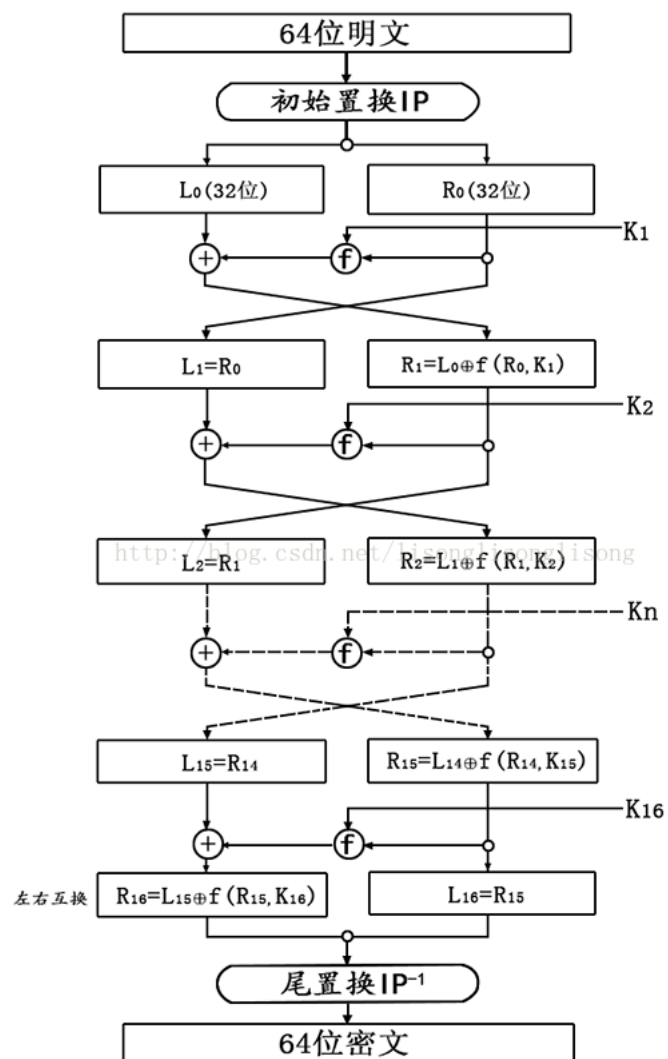


图 1: DES加密算法整体流程图

1.2 初始置换IP

初始置换 IP 表用于对64比特明文的置换，其含义是将对应明文的位上的数据置换到其对应的位置上。譬如， IP 表中第一位为58，表示将明文第58位的数据置换到第一位上；表中第二位为50，表示将明文第50位的数据置换到第二位上，以此类推。

我们在程序中，对其的实现如下：

```
1      int P[] = { 16,  7, 20, 21,  
2                  29, 12, 28, 17,  
3                  1, 15, 23, 26,  
4                  5, 18, 31, 10,  
5                  2,  8, 24, 14,  
6                  32, 27,  3,  9,  
7                  19, 13, 30,  6,  
8                  22, 11,  4, 25};  
9
```

1.3 轮结构

图2是DES加密算法的轮结构[†]，我们主要来看密码函数 F 的实现过程。可以看到，在每一轮加密过程中：

- 1、DES首先将这一轮的右32位作为下一轮的左32位；
- 2、这一轮的右32位经过E表进行扩展/置换，由32比特变为48比特；
- 3、随后这48比特数据与这一轮对应的48位子密钥进行异或操作，得到48比特的数据；
- 4、使用S盒对这48比特数据进行代换/选择，数据由48比特变为32比特；
- 5、利用P表对上一步得到的32比特数据进行置换操作，得到新的32比特数据；
- 6、最后使用这一轮的左32比特与上一步得到的32比特数据进行异或操作，就得到了下一轮的右32比特数据；
- 7、上述步骤重复16次，即16轮加密；

[†]使用教材中对应的图片

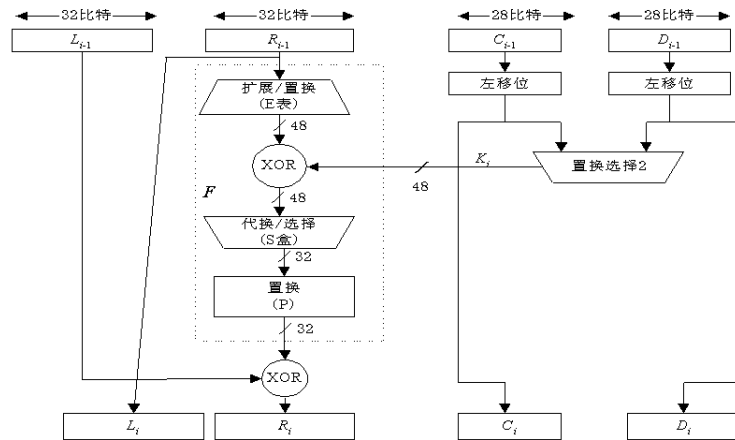


图 2: DES加密算法的轮结构

这些步骤中用到的工具如下：

1.3.1 E表

```

1  int E[] = { 32,  1,  2,  3,  4,  5,
2              4,  5,  6,  7,  8,  9,
3              8,  9, 10, 11, 12, 13,
4              12, 13, 14, 15, 16, 17,
5              16, 17, 18, 19, 20, 21,
6              20, 21, 22, 23, 24, 25,
7              24, 25, 26, 27, 28, 29,
8              28, 29, 30, 31, 32,  1 };
9

```

1.3.2 S盒

```

1  int S_BOX[8][4][16] = {
2      {
3          { 14, 4, 13, 1, 2, 15, 11, 8, 3, 10, 6, 12, 5, 9, 0, 7 },
4          { 0, 15, 7, 4, 14, 2, 13, 1, 10, 6, 12, 11, 9, 5, 3, 8 },
5          { 4, 1, 14, 8, 13, 6, 2, 11, 15, 12, 9, 7, 3, 10, 5, 0 },
6          { 15, 12, 8, 2, 4, 9, 1, 7, 5, 11, 3, 14, 10, 0, 6, 13 }
7      },
8      {
9          { 15, 1, 8, 14, 6, 11, 3, 4, 9, 7, 2, 13, 12, 0, 5, 10 },
10         { 3, 13, 4, 7, 15, 2, 8, 14, 12, 0, 1, 10, 6, 9, 11, 5 },
11         { 0, 14, 7, 11, 10, 4, 13, 1, 5, 8, 12, 6, 9, 3, 2, 15 },
12         { 13, 8, 10, 1, 3, 15, 4, 2, 11, 6, 7, 12, 0, 5, 14, 9 }
13     },
14     {

```

```

15         {10,0,9,14,6,3,15,5,1,13,12,7,11,4,2,8},
16         {13,7,0,9,3,4,6,10,2,8,5,14,12,11,15,1},
17         {13,6,4,9,8,15,3,0,11,1,2,12,5,10,14,7},
18         {1,10,13,0,6,9,8,7,4,15,14,3,11,5,2,12}
19     },
20     {
21         {7,13,14,3,0,6,9,10,1,2,8,5,11,12,4,15},
22         {13,8,11,5,6,15,0,3,4,7,2,12,1,10,14,9},
23         {10,6,9,0,12,11,7,13,15,1,3,14,5,2,8,4},
24         {3,15,0,6,10,1,13,8,9,4,5,11,12,7,2,14}
25     },
26     {
27         {2,12,4,1,7,10,11,6,8,5,3,15,13,0,14,9},
28         {14,11,2,12,4,7,13,1,5,0,15,10,3,9,8,6},
29         {4,2,1,11,10,13,7,8,15,9,12,5,6,3,0,14},
30         {11,8,12,7,1,14,2,13,6,15,0,9,10,4,5,3}
31     },
32     {
33         {12,1,10,15,9,2,6,8,0,13,3,4,14,7,5,11},
34         {10,15,4,2,7,12,9,5,6,1,13,14,0,11,3,8},
35         {9,14,15,5,2,8,12,3,7,0,4,10,1,13,11,6},
36         {4,3,2,12,9,5,15,10,11,14,1,7,6,0,8,13}
37     },
38     {
39         {4,11,2,14,15,0,8,13,3,12,9,7,5,10,6,1},
40         {13,0,11,7,4,9,1,10,14,3,5,12,2,15,8,6},
41         {1,4,11,13,12,3,7,14,10,15,6,8,0,5,9,2},
42         {6,11,13,8,1,4,10,7,9,5,0,15,14,2,3,12}
43     },
44     {
45         {13,2,8,4,6,15,11,1,10,9,3,14,5,0,12,7},
46         {1,15,13,8,10,3,7,4,12,5,6,11,0,14,9,2},
47         {7,11,4,1,9,12,14,2,0,6,10,13,15,3,5,8},
48         {2,1,14,7,4,10,8,13,15,12,9,0,3,5,6,11}
49     }
50 };
51

```

1.3.3 P表

```

1      int P[] = { 16,  7, 20, 21,
2                  29, 12, 28, 17,
3                  1,  15, 23, 26,
4                  5,  18, 31, 10,
5                  2,   8, 24, 14,
6                  32, 27,  3,  9,
7                  19, 13, 30,  6,
8                  22, 11,  4, 25 };
9

```

1.4 函数 $F(R, K)$

下面我们仔细考虑函数 F 的执行过程(图3)[‡]:

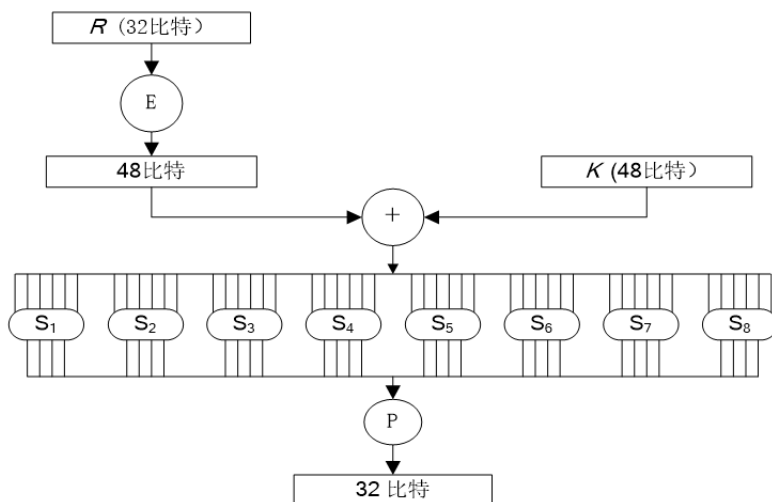


图 3: 函数 $F(R, K)$ 的计算过程

可以看到, 函数的输入分别为 R (32位数据)、 K (48位子密钥)。

我们主要考虑 S 盒的代换/选择过程, 在48比特数据与48比特子密钥进行异或操作后, 对得到的 48比特结果进行分组。总共分为8组, 每组有6比特数据。对于这8组数据, 我们分别使用8个 S 盒进行代换/选择。

譬如, 第一组数据, 我们选择第一个 S 盒。我们取这一组数据的第一个比特和第六个比特作为行号, 选第二至五个比特作为列号, 找到对应的表项, 并将其转化为对应的二进制数即为这个 S 盒的输出。

最后将每个 S 盒的输出组合, 得到一个32比特的数据, 作为 F 函数的输出。

1.5 尾置换 IP^{-1}

我们将最后一轮得到的数据进行左右交换, 然后利用 IP^{-1} 表进行置换操作, 最终就得到了我们想要的密文。

IP^{-1} 在我们的代码实现中定义如下:

[‡]使用教材中对应的图片


```

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

```

1.6 子密钥的产生

下面我们来考虑子密钥的生成过程，依然参照教材中的图：

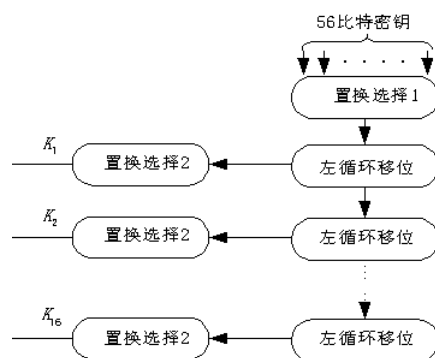


图 4: 子密钥产生过程

可以看到，我们首先对去除校验位的56比特密钥进行置换选择，该操作使用 $PC-1$ 表进行， $PC-1$ 表在我们的代码实现中定义如下：

```

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

```

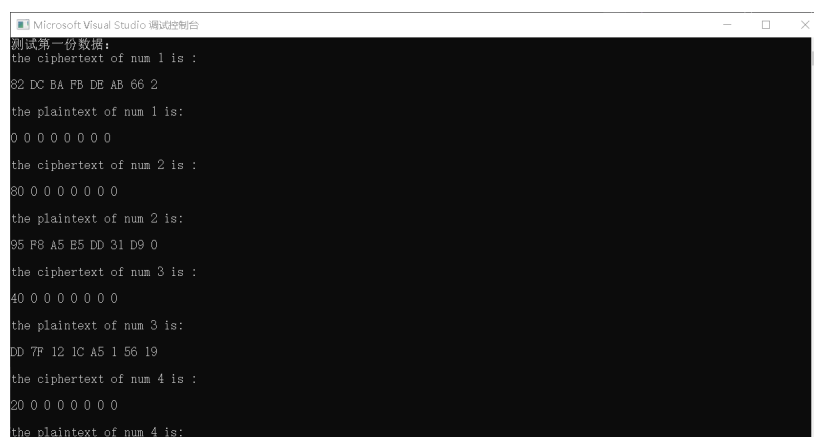
随后对于每一轮加密，我们首先对其进行左循环移位，随后使用 $PC - 2$ 表进行置换选择操作，得到这一轮对应的子密钥。其中左循环移位的位数使用左循环移位表得到。 $PC - 2$ 表与左循环移位表在我们的代码中定义如下：

```
1      int PC_2[] = { 14, 17, 11, 24, 1, 5,  
2                    3, 28, 15, 6, 21, 10,  
3                    23, 19, 12, 4, 26, 8,  
4                    16, 7, 27, 20, 13, 2,  
5                    41, 52, 31, 37, 47, 55,  
6                    30, 40, 51, 45, 33, 48,  
7                    44, 49, 39, 56, 34, 53,  
8                    46, 42, 50, 36, 29, 32 };  
9  
10     int shiftBits[] = { 1, 1, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 1  
11     };
```

至此，我们基本实现了DES加密算法所需的工具并了解了其运行的大致步骤。

2 DES运行结果

下面我们来查看所实现的DES算法的执行效果，我们将文档中提供的实验数据使用程序执行，结果如下：



```
Microsoft Visual Studio 调试控制台  
测试第一份数据:  
the ciphertext of num 1 is :  
82 DC BA FB DE AB 66 2  
the plaintext of num 1 is:  
0 0 0 0 0 0 0 0  
the ciphertext of num 2 is :  
80 0 0 0 0 0 0 0  
the plaintext of num 2 is:  
95 F8 A5 E5 DD 31 D9 0  
the ciphertext of num 3 is :  
40 0 0 0 0 0 0 0  
the plaintext of num 3 is:  
DD 7F 12 1C A5 1 56 19  
the ciphertext of num 4 is :  
20 0 0 0 0 0 0 0  
the plaintext of num 4 is:
```

图 5: DES执行效果

具体如下：

测试第一份数据:

the ciphertext of num 1 is :

82 DC BA FB DE AB 66 2

the plaintext of num 1 is:

0 0 0 0 0 0 0 0

the ciphertext of num 2 is :

80 0 0 0 0 0 0 0

the plaintext of num 2 is:

95 F8 A5 E5 DD 31 D9 0

the ciphertext of num 3 is :

40 0 0 0 0 0 0 0

the plaintext of num 3 is:

DD 7F 12 1C A5 1 56 19

the ciphertext of num 4 is :

20 0 0 0 0 0 0 0

the plaintext of num 4 is:

2E 86 53 10 4F 38 34 EA

the ciphertext of num 5 is :

10 0 0 0 0 0 0 0

the plaintext of num 5 is:

4B D3 88 FF 6C D8 1D 4F

the ciphertext of num 6 is :

8 0 0 0 0 0 0 0

the plaintext of num 6 is:

20 B9 E7 67 B2 FB 14 56

the ciphertext of num 7 is :

4 0 0 0 0 0 0 0

the plaintext of num 7 is:

55 57 93 80 D7 71 38 EF

the ciphertext of num 8 is :

2 0 0 0 0 0 0 0

the plaintext of num 8 is:

6C C5 DE FA AF 4 51 2F

the ciphertext of num 9 is :

1 0 0 0 0 0 0

the plaintext of num 9 is:

D 9F 27 9B A5 D8 72 60

the ciphertext of num 10 is :

0 80 0 0 0 0 0

the plaintext of num 10 is:

D9 3 1B 2 71 BD 5A A

测试第二份数据:

the plaintext of num 1 is :

0 0 0 0 0 0 0

the plaintext of num 2 is :

95 F8 A5 E5 DD 31 D9 0

the plaintext of num 3 is :

DD 7F 12 1C A5 1 56 19

the plaintext of num 4 is :

2E 86 53 10 4F 38 34 EA

the plaintext of num 5 is :

4B D3 88 FF 6C D8 1D 4F

the plaintext of num 6 is :

20 B9 E7 67 B2 FB 14 56

the plaintext of num 7 is :

55 57 93 80 D7 71 38 EF

the plaintext of num 8 is :

6C C5 DE FA AF 4 51 2F

the plaintext of num 9 is :

D 9F 27 9B A5 D8 72 60

the plaintext of num 10 is :

D9 3 1B 2 71 BD 5A A

可以看到，程序输出了正确的结果。

3 雪崩效应

雪崩效应就是一种不稳定的平衡状态也是加密算法的一种特征，它指明文或密钥的少量变化会引起密文的很大变化，就像雪崩前，山上看上去很平静，但是只要有一点问题，就会造成一片大崩溃。可以用在很多场合对于Hash码，雪崩效应是指少量消息位的变化会引起信息摘要的许多位变化。[§]

下面我们来测试DES算法的雪崩效应，我们不妨对文档数据中的第一组进行测试。这里我将第一组明文中的第四个数据修改为了“0x5A”，将第一组密文中的第七个数据修改为了“0x3B”。

```
1 //the plaintext after modification
2 int my_txt_2 [] = { 0x00, 0x00, 0x00, 0x5A, 0x00, 0x00, 0x00, 0x00 };
3
4 //the ciphertext after modification
5 int my_txt_3 [] = { 0x82, 0xDC, 0xBA, 0xFB, 0xDE, 0xAB, 0x3B, 0x02 };
6
```

我们对原有的DES算法代码稍作修改，得到其执行结果如下：

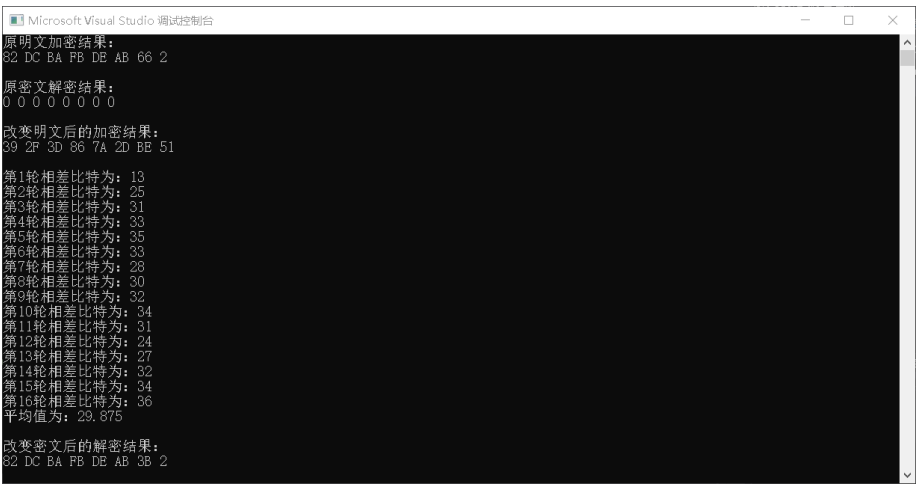


图 6: 雪崩效应分析结果-1

[§]摘自百度百科

```

Microsoft Visual Studio 调试控制台
第16轮相差比特为: 36
平均值为: 29.875

改变明文后的解密结果:
92 DC BA FB DE AB 38 2

第1轮相差比特为: 13
第2轮相差比特为: 25
第3轮相差比特为: 31
第4轮相差比特为: 33
第5轮相差比特为: 35
第6轮相差比特为: 33
第7轮相差比特为: 28
第8轮相差比特为: 30
第9轮相差比特为: 32
第10轮相差比特为: 34
第11轮相差比特为: 31
第12轮相差比特为: 24
第13轮相差比特为: 27
第14轮相差比特为: 32
第15轮相差比特为: 34
第16轮相差比特为: 36
平均值为: 29.875

C:\Users\DELL\source\repos\ConsoleApplication38\Debug\ConsoleApplication38.exe (进程 13616) 已退出, 返回代码为: 0.
若要在调试停止时自动关闭控制台, 请启用“工具”->“选项”->“调试”->“调试停止时自动关闭控制台”。
按任意键关闭此窗口...

```

图 7: 雪崩效应分析结果-2

我们不妨以表格的形式更直观地感受这一数据:

加密轮数	1	2	3	4	5	6	7	8
密文相差比特数	13	25	31	33	35	33	28	30
加密轮数	9	10	11	12	13	14	15	16
密文相差比特数	32	34	31	24	27	32	34	36
平均值	29.875							

表 1: 修改的明文与原明文的加密差异

解密轮数	1	2	3	4	5	6	7	8
明文相差比特数	13	25	31	33	35	33	28	30
解密轮数	9	10	11	12	13	14	15	16
明文相差比特数	32	34	31	24	27	32	34	36
平均值	29.875							

表 2: 修改的密文与原密文的解密差异

可以看到, 哪怕只是对明文或者密文进行微小的修改, 最终的结果也有巨大差异。因此, DES算法确实具有雪崩效应。

参考文献

- [1] 现代密码学/杨波编著. — 4版. — 北京: 清华大学出版社, 2017
(2018.8重印)
- [2] <https://blog.csdn.net/lisonglisonglisong/article/details/41777413>