
中国矿业大学计算机学院

2019 级本科生课程报告

课程名称	密码学课程设计
------	---------

学 号	05191643
-----	----------

专 业	信息安全
-----	------

任课教师	汪楚娇
------	-----

学生姓名	许万鹏
------	-----

报告时间	2022 年 1 月 10 日
------	-----------------

报告说明

1. 请认真查看教务系统，封面内容请准确填写；
2. 目录为 Word 自动生成，不能使用手动填写目录；
3. 课程报告打印上交（建议双面打印），并提交电子版；
4. 报告内容须有详细步骤（包含解题程序代码）和重要步骤截图，
截图要清晰，报告格式要规范；具体格式参考本科毕业设计论文
格式，主要要求如下：
 - a) 一级标题：黑体小 2 加粗，单倍行距，段前 0.5 行，段后 0 行；
 - b) 二级标题：黑体小 3，行距固定值 20 磅，段前 0.5 行，段后 0.5 行；
 - c) 三级标题：黑体 4 号，行距固定值 20 磅，段前 0.5 行，段后 0.5 行；
 - d) 正文内容：宋体小 4 号，行距固定值 20 磅，英文用 Times New Roman，小 4 号；
 - e) 图标题置于图的下方，居中；宋体 5 号字，单倍行距；
5. 严禁抄袭，如有发现，两份报告课程成绩均按零分处理。

目 录

1 古典密码.....	6
1.1 维吉尼亚密码.....	6
1.1.1 算法原理.....	6
1.1.2 代码实现.....	6
1.1.3 正确性检验与性能分析.....	7
1.1.4 安全性与可用性分析.....	8
1.1.5 破解或攻击方式分析.....	8
1.2 Hill 密码	10
1.2.1 算法原理.....	10
1.2.2 代码实现.....	10
1.2.3 正确性检验与性能分析.....	12
1.2.4 安全性与可用性分析.....	13
1.2.5 破解或攻击方式分析.....	13
2 序列密码.....	16
2.1 线性反馈移位寄存器.....	16
2.1.1 算法原理.....	16
2.1.2 代码实现.....	16
2.1.3 正确性检验与周期分析.....	17
2.1.4 安全性与可用性分析.....	19
2.1.5 破解或攻击方式分析.....	19
2.2 LFSRXOR.....	20
2.2.1 分析.....	20
2.2.2 脚本.....	21
2.2.3 运行结果.....	23
3 分组密码.....	24
3.1 DES	24
3.1.1 算法原理.....	24
3.1.2 代码实现.....	24
3.1.3 正确性检验与性能分析.....	26
3.1.4 安全性与可用性分析.....	26
3.1.5 破解或攻击方式分析.....	27
3.2 工作模式.....	27
3.2.1 电子密码本模式（ECB）	27

3.2.2	电子密码本模式 (ECB)	31
3.2.3	密码反馈模式 (CFB)	35
3.2.4	输出反馈模式 (OFB)	39
3.2.5	计数器模式 (CTR)	43
3.3	RC6	47
3.3.1	算法原理	47
4.2.2.	代码实现	48
4.2.3.	正确性检验与性能分析	49
4.2.4.	安全性与可用性分析	49
3.4	AES2	50
3.5	DES3	50
4	公钥密码	51
4.1	RSA 加密	51
4.1.1	算法原理	51
4.1.2	Miller-Rabin 素性检验	51
4.1.3	代码实现	51
4.1.4	正确性检验与性能分析	54
4.1.5	安全性与可用性分析	54
4.1.6	破解或攻击方式分析	55
4.2	生成 pem 文件	55
4.2.1	代码实现	55
4.2.2	正确性验证	56
4.3	RSA	57
4.3.1	分析	57
4.3.2	脚本	57
4.3.3	运行结果	61
4.4	mediumRSA	61
4.4.1	分析	61
4.4.2	脚本	62
4.4.3	运行结果	62
4.5	hardRSA	63
4.2.1.	分析	63
4.2.2.	脚本	63
4.2.3.	运行结果	64

4.6 veryhardRSA.....	64
4.6.1 分析.....	64
4.6.2 脚本.....	64
4.6.3 运行结果.....	65
4.7 cipher.....	65
4.7.1 分析.....	65
4.7.2 脚本.....	66
4.7.3 运行结果.....	66
5 Hash 函数	67
5.1 MD5	67
5.1.1 算法原理.....	67
5.1.2 代码实现.....	67
5.1.3 正确性检验与性能分析.....	69
5.1.4 安全性与可用性分析.....	72
5.1.5 破解或攻击方式分析.....	73
5.2 md51	73
5.2.1 分析.....	73
5.2.2 脚本.....	74
5.2.3 运行结果.....	74
6 综合实践.....	75
6.1 设计要求.....	75
6.2 设计原理.....	75
6.3 代码实现.....	77
6.4 功能检验.....	80

1 古典密码

1.1 维吉尼亚密码

1.1.1 算法原理

维吉尼亚密码是使用一系列凯撒密码组成密码字母表的加密算法，属于多表替换密码的一种简单形式。

设 m 为某一固定的正整数， P 、 C 和 K 分别为明文空间、密文空间和密钥空间，并且 $P = K = C = (Z_{26})^m$ ，对一个密钥 $k = (k_1, k_2, \dots, k_m)$ ，维吉尼亚密码的加密函数为

$$e_k(x_1, x_2, \dots, x_m) = (x_1 + k, x_2 + k, \dots, x_m + k)$$

若写作同余式则为

$$C_i \equiv P_i + K_i \pmod{26}$$

同理，其解密文法可以写作

$$P_i \equiv C_i - K_i \pmod{26}$$

1.1.2 代码实现

1) 加密函数

```
def encrypt(self, plaintext: str) -> None:
    """
    维吉尼亚加密
    @param plaintext: 明文
    @return: 无
    """
    self.clear(False)
    self.p = plaintext
    len_k = len(self.k)
    j = 0
    for i, _p in enumerate(self.p):
        if _p.isupper():
            self.c += chr(ord('A') + ((ord(_p) - ord('A')) +
            (ord(self.k[j % len_k]) - ord('A')) % 26))
        elif _p.islower():
            self.c += chr(ord('a') + ((ord(_p) - ord('a')) +
            (ord(self.k[j % len_k]) - ord('A')) % 26))
        else:
            self.c += _p
```

```
        continue
    j += 1
```

2) 解密函数

```
def decrypt(self, ciphertext: str) -> None:
    """
    维吉尼亚解密
    @param ciphertext: 密文
    @return: 无
    """
    self.clear(False)
    self.c = ciphertext
    len_k = len(self.k)
    j = 0
    for i, _c in enumerate(self.c):
        if _c.isupper():
            self.p += chr(ord('A') + ((ord(_c) - ord('A')) -
            (ord(self.k[j % len_k]) - ord('A')))) % 26)
        elif _c.islower():
            self.p += chr(ord('a') + ((ord(_c) - ord('a')) -
            (ord(self.k[j % len_k]) - ord('A')))) % 26)
        else:
            self.p += _c
            continue
        j += 1
```

1.1.3 正确性检验与性能分析

1) 正确性验证

请选择密码机模式：

[E]加密Encrypt [D]解密Decrypt [C]破解Crack [R]重置Reset [Q]关闭Quit
E

请输入key以创建密码机（若无key可留空）：cunt

请输入明文：china

密文：ebugc

请选择密码机模式：

[E]加密Encrypt [D]解密Decrypt [C]破解Crack [R]重置Reset [Q]关闭Quit
D

请输入密文：ebugc

明文：china

请选择密码机模式：

[E]加密Encrypt [D]解密Decrypt [C]破解Crack [R]重置Reset [Q]关闭Quit
q

密码机已关闭

2) 性能分析

维吉尼亚密码是一种加解密简洁的密码，只需要一层 for 循环便可完成加解密，时间复杂度为 $O(n)$ ，也可以通过预先建立二维表以空间换时间。

1.1.4 安全性与可用性分析

1) 安全性

维吉尼亚是一种多表替换密码，相对于单表替换密码而言打破了字符出现规律，一定程度上掩盖了明文的统计特性，无法直接使用频率分析法，同时密钥空间大大增大，安全性大大提高。

但由于实际上密钥的长度常常小于明文的长度，也就是说需要重复使用密钥，无法做到一次一密，所以维吉尼亚密码仍然可以用重合指数法等方法进行唯密文攻击，但是前提是密文足够的长，较短的密文几乎是不可破译的。

2) 可用性

维吉尼亚密码加解密算法简单，时间耗费少，适合于资源有限的情况下对短信息的加密。而对于较重要的信息以及大量的明文情况下，则不适合使用维吉尼亚密码进行，相对于对于现代的计算能力来说，其实际上已无太大实际用途。

1.1.5 破解或攻击方式分析

实际上，通过分析可以知道维吉尼亚密码有着比较明显的漏洞，即如果敌手知道了密钥的长度，那密文就可以被看作是经过列混合的多组恺撒密码，而

其中的每一组都可以单独破解。所以要想破解维吉尼亚密码，首要的步骤就是得出密钥的长度。

确定密钥的长度，可以使用 Kasiski 测试法或重合指数法，我在这里使用重合指数法。

接下来要获取每组单表替换密码（凯撒密码）的密钥，因为密钥空间只有 26-1，所以可以使用暴力破解或重合指数法。

具体代码实现如下，已省略其他函数：

```
def crack(self, ciphertext: str) -> None:
    """
    维吉尼亚唯密文攻击
    @param ciphertext: 密文
    @return: 无
    """
    self.clear()
    # 确定密钥长度
    self.c = ciphertext
    self.alpha_c = self.c_alpha(self.c)
    m = self.pre_10(self.alpha_c)
    key_len = self.gcd(m[1:3] + 1) # 取 1~3 为高可信密钥长度，取他们的
    # 最大公因数为密钥长度
    # key_len = int(input()) # 输入猜测的秘钥长度
    # 确定密钥
    self.one_key(ciphertext, key_len)
    self.decrypt(ciphertext)
```

正确性验证：

```
请选择密码机模式：
[E]加密Encrypt [D]解密Decrypt [C]破解Crack [R]重置Reset [Q]关闭Quit
C
请输入密文：CHREE VOAHM AERAT BIAXX WTNXB EEOPH BSBQM QEGER BWRVX UOAKK AOSXX WEAHB WGJMM QMNKG RFVGX WTRZX WIAKL XFPSK AUTEM NDCMG TSXNX BTUJA
DNGMG PSREL XNJEL XVRVP RTULH DNUWT WDTYG BPHXT FALJH ASVBF XNGLL CHRZB WELEK MSJIK NBHWR JGNMG JSGLX FEYPH AGNRB IEQJT AMRVL CRREM NDGLX
RRIMG NSNRW CHRQH AEYEV TAQEB BIPEE WEVKA KOEWA DRENX NTBHH CHRTK DNVZ CHRL QHPW QAIW XNRMG WOIF KEE
密码机已获取密钥：JANET
明文：THEAL MONDT REEWA SINTE NTATI VEBLO SSOMT HEDAY SWERE LONGR ROFTE NENDI NGWIT HMAGN IFICE NTEVE NINGS OFCOR RUGAT EDPIN KSKJE STHEH UNTIN
GSEAS ONWAS OVERW ITHHO UNHSA NDGUN SPUTA WAYFO RSIXM ONTHS THEVI MEYAR DSWER EBUSY AGAIN ASTHE WELLO RGANI ZEDFA RMERS TREAT EDTHE IRVIN
ESAND THEM0 RELAC KADAI SICAL NEIGH BORSH URRJE ETODO THEPR UNING THEYS HOULD HAVED ONEIN NOVEN BER
明文：the almond tree was in tentative blossom the days were long rr often ending with magnificent evenings of corrugated pink sk jes the
hunting season was over with hou nhs and guns put away for six months the vineyards were busy again as the well organized farmers treated
their vines and the more lackadaisical neighbors hur r jeet odo the pruning they should have done in november
请选择密码机模式：
[E]加密Encrypt [D]解密Decrypt [C]破解Crack [R]重置Reset [Q]关闭Quit
Q
密码机已关闭
```

1.2 Hill 密码

1.2.1 算法原理

Hill 密码的基本思想是利用 Z_{26} 上的线性变换把 n 个连续的明文字母替换为 n 密文字母，它的密钥是一个变换矩阵，解密时只需对密文做一次逆变换即可，本质上是通过一个变换矩阵把明文变换为密文的一种密码体制。

加密文法：

$$(c_1, c_2, \dots, c_n) \equiv (p_1, p_2, \dots, p_n) \begin{pmatrix} k_{11} & \cdots & k_{1n} \\ \vdots & \ddots & \vdots \\ k_{n1} & \cdots & k_{nn} \end{pmatrix} (\text{mod } 26)$$

也可写作：

$$c_{1 \times n} \equiv p_{1 \times n} \times k_{n \times n} (\text{mod } 26)$$

解密文法：

$$(p_1, p_2, \dots, p_n) \equiv (c_1, c_2, \dots, c_n) \begin{pmatrix} k_{11} & \cdots & k_{1n} \\ \vdots & \ddots & \vdots \\ k_{n1} & \cdots & k_{nn} \end{pmatrix}^{-1} (\text{mod } 26)$$

也可写作：

$$p_{1 \times n} \equiv c_{1 \times n} \times k_{n \times n}^{-1} (\text{mod } 26)$$

要注意的是密钥 k 在 Z_{26} 是一个非奇异矩阵，且满足 $\gcd(\det(k), 26) = 1$ ，也就是其逆矩阵存在。

1.2.2 代码实现

1) 加密函数

```
def encrypt(self, plaintext: str) -> None:
    """
    希尔密码加密方法
    @param plaintext: 明文
    @return: 无
    """
    self.p = plaintext.lower()
    blocks = [self.p[i:i + self.o] for i in range(0, len(self.p), self.o)]
    # 每 阶数个 明文作为一组
    blocks[-1] = blocks[-1].ljust(self.o, 'a') # 对最后一组进行填充以进行矩阵乘法
    temp = np.array([list(map(ord, block)) for block in blocks]) - ord('a')
    # 将每组的字母转换为字母序号
    encrypted_code = (temp @ self.k) % 26 + ord('A')
    decryption_matrix_text = ''.join(map(chr, encrypted_code.ravel()))
    self.c = decryption_matrix_text[:len(self.p)] # 取出密文，忽略填充字符
```

2) 解密函数

```
def decrypt(self, ciphertext: str) -> None:
    """
    希尔密码解密方法
    @param ciphertext: 密文
    @return: 无
    """
    self.c = ciphertext.lower()
    self.k_inv = self.solve_mod26_inverse_matrix(self.k) # 求密钥矩阵的逆矩阵
    blocks = [self.c[i:i + self.o] for i in range(0, len(self.c), self.o)]
    blocks[-1] = blocks[-1].ljust(self.o, 'a')
    temp = np.array([list(map(ord, block)) for block in blocks]) - ord('a')
    encrypted_code = ((temp @ self.k_inv) % 26) + ord('a')
    decryption_matrix_text = ''.join(map(lambda x: chr(int(x)),
    encrypted_code.ravel()))
    self.p = decryption_matrix_text[:len(self.c)]
```

两段代码最关键的地方在于如何求解 Z_{26} 上的（而不是 R 上的）逆矩阵

3) 求逆矩阵函数及其必须函数

```
def ex_gcd(self, a, b, arr) -> int: # 扩展欧几里得
    if b == 0:
        arr[0], arr[1] = 1, 0
        return a
    g = self.ex_gcd(b, a % b, arr)
    arr[0], arr[1] = arr[1], arr[0] - int(a / b) * arr[1]
    return g

def mod_reverse(self, a, n) -> int: # ax=1(mod m) 求a模m的乘法逆元
    x(a-1)
    arr = [0, 1]
    gcd = self.ex_gcd(a, n, arr)
    if gcd == 1:
        return (arr[0] % n + n) % n
    else:
        return -1

def solve_mod26_inverse_matrix(self, matrix) -> None:
    det = int(np.linalg.det(matrix)) # det(p)
    inverse_matrix = np.linalg.inv(matrix) # p在实数域的逆矩阵
    adjoint_matrix = (inverse_matrix * det) # p的伴随矩阵=det(p)*p-1
    adjoint_matrix_mod26 = adjoint_matrix % 26 # p的模26伴随矩阵=p的伴随
```

```

矩阵模 26
    modular_multiplicative_inverse = self.mod_reverse(det, 26) #
det(p)^(-1) mod 26
    inverse_matrix = np.around((modular_multiplicative_inverse *
adjoint_matrix_mod26)).astype('int') % 26
    return inverse_matrix

```

1.2.3 正确性检验与性能分析

1) 加解密正确性

请选择密码机模式:

[E]加密Encrypt [D]解密Decrypt [C]破解Crack [R]重置Reset [Q]关闭Quit

E

请输入key的阶数n以创建密码机: 5

请输入key以创建密码机:

10 5 12 0 0

3 14 21 0 0

8 9 11 0 0

0 0 0 11 8

0 0 0 3 7

请输入明文: cryptologyinformationsets

密文: DWVOTZMHIIDHIXMPAGIPGDS

请选择密码机模式:

[E]加密Encrypt [D]解密Decrypt [C]破解Crack [R]重置Reset [Q]关闭Quit

D

请输入密文: DWVOTZMHIIDHIXMPAGIPGDS

明文: cryptologyinformationsets

请选择密码机模式:

[E]加密Encrypt [D]解密Decrypt [C]破解Crack [R]重置Reset [Q]关闭Quit

Q

密码机已关闭

2) 性能分析

事实上, 由矩阵乘法的 $O(n^3)$ 时间复杂度, 考虑 c 为 $1 \times n$ 的矩阵, 那么加密只需 $O(n^2)$ 的时间复杂度。又求解 Z_{26} 上的逆矩阵需要 $O(n^3)$ 时间复杂度, 解密需 $O(n^2)$ 时间复杂度, 那么解密共需 $O(n^3) + O(n^2) = O(n^3)$ 时间复杂度。

1.2.4 安全性与可用性分析

希尔密码将长消息分组，分组的长度由矩阵的维数决定，更好地隐藏了单字母的统计特性，所以希尔密码能较好地抵抗统计分析法，对抗唯密文攻击的强度较高，但易受到已知明文攻击。

1.2.5 破解或攻击方式分析

由上所述，我们考虑已知明文攻击的情况。

假设攻击者已经确定正在使用的希尔密码的矩阵维数为 n ，并且知道至少有 n 个不同的明文-密文对：

$$\vec{p}_i = (p_{1,i}, p_{2,i}, \dots, p_{n,i}), i = 1, 2, \dots, n$$

$$\vec{c}_i = (c_{1,i}, c_{2,i}, \dots, c_{n,i}), i = 1, 2, \dots, n$$

则由希尔密码的定义可得：

$$(c_{ij})_{n \times n} \equiv ((p_{ij})_{n \times n} \times (k_{ij})_{n \times n}) \pmod{26}$$

$$(k_{ij})_{n \times n} \equiv (p_{ij})_{n \times n}^{-1} \times (c_{ij})_{n \times n} \pmod{26}$$

即

$$\begin{pmatrix} k_{11} & \cdots & k_{1n} \\ \vdots & \ddots & \vdots \\ k_{n1} & \cdots & k_{nn} \end{pmatrix} \equiv \begin{pmatrix} p_{11} & \cdots & p_{1n} \\ \vdots & \ddots & \vdots \\ p_{n1} & \cdots & p_{nn} \end{pmatrix}^{-1} \times \begin{pmatrix} c_{11} & \cdots & c_{1n} \\ \vdots & \ddots & \vdots \\ c_{n1} & \cdots & c_{nn} \end{pmatrix} \pmod{26}$$

上述结论成立的前提条件是 \vec{p}^{-1} 存在，这就要求明文 \vec{p} 是一个非奇异方阵，即要求 $\det(\vec{p}) \neq 0$ ，且满足 $\gcd(\det(\vec{p}), 26) = 1$ 。所以已知明文-密文对攻击推导希尔密码密钥时至少要知道与矩阵维数相等个数的明文-密文对。如果明文矩阵不可逆，则尝试其他已知的明文-密文对重新组成明文矩阵再分析，直到找到满足可逆的明文矩阵为止。

按照以上方法，我们设计破解代码：

```
def crack(self, plaintext, ciphertext, order) -> None:
    self.p = plaintext
    self.c = ciphertext
    self.o = order
    p_matrix = self.text2matrix(self.p)[0:self.o:1] # 切片为方阵以计算行列式
    c_matrix = self.text2matrix(self.c)[0:self.o:1]
    p_inverse_matrix = self.solve_mod26_inverse_matrix(p_matrix)
    self.k = (p_inverse_matrix @ c_matrix) % 26

    det_p = int(np.linalg.det(p_matrix)) % 26
```

```
if (det_p != 0 and self.ex_gcd(det_p, 26, [0, 1]) == 1): # 可逆检测
    return True
else:
    return False
```

正确性验证:

前文例

请选择密码机模式:

[E]加密Encrypt [D]解密Decrypt [C]破解Crack [R]重置Reset [Q]关闭Quit

C

请输入明文: cryptologyinformationsets

请输入密文: DWVOTZMHIIDHIXMPAGIPGDS

请输入key的阶数n: 5

密码机已获取密钥:

```
[[10  5 12  0  0]
 [ 3 14 21  0  0]
 [ 8  9 11  0  0]
 [ 0  0  0 11  8]
 [ 0  0  0  3  7]]
```

请选择密码机模式:

[E]加密Encrypt [D]解密Decrypt [C]破解Crack [R]重置Reset [Q]关闭Quit

Q

密码机已关闭

P65 例题改

请选择密码机模式:

[E]加密Encrypt [D]解密Decrypt [C]破解Crack [R]重置Reset [Q]关闭Quit

C

请输入明文: friday

请输入密文: PQCFKU

请输入key的阶数n: 2

密码机已获取密钥:

```
[[ 7 19]
 [ 8  3]]
```

请选择密码机模式:

[E]加密Encrypt [D]解密Decrypt [C]破解Crack [R]重置Reset [Q]关闭Quit

Q

密码机已关闭

例 3.4 假设明文 friday 利用 $m = 2$ 的希尔密码加密，得到密文为 PQCFKU。

首先我们有

$$e_K(5, 17) = (15, 16), e_K(8, 3) = (2, 5), e_K(0, 24) = (10, 20)$$

使用头两个明一密文对，可得矩阵方程

$$\begin{pmatrix} 15 & 16 \\ 2 & 5 \end{pmatrix} = \begin{pmatrix} 5 & 17 \\ 8 & 3 \end{pmatrix} K$$

利用引理 1.4，容易计算

$$\begin{pmatrix} 5 & 17 \\ 8 & 3 \end{pmatrix}^{-1} = \begin{pmatrix} 9 & 1 \\ 2 & 15 \end{pmatrix}$$

因此

$$K = \begin{pmatrix} 9 & 1 \\ 2 & 15 \end{pmatrix} \begin{pmatrix} 15 & 16 \\ 2 & 5 \end{pmatrix} = \begin{pmatrix} 7 & 19 \\ 8 & 3 \end{pmatrix}$$

与手算结果一致。

完整性验证：

请选择密码机模式：

[E]加密Encrypt [D]解密Decrypt [C]破解Crack [R]重置Reset [Q]关闭Quit

C

请输入明文：informationsecuritycenter

请输入密文：DHIXMXPAGIPGEAIEJKYXJKRV

请输入key的阶数n：5

明文矩阵不可逆！

请选择密码机模式：

[E]加密Encrypt [D]解密Decrypt [C]破解Crack [R]重置Reset [Q]关闭Quit

Q

密码机已关闭

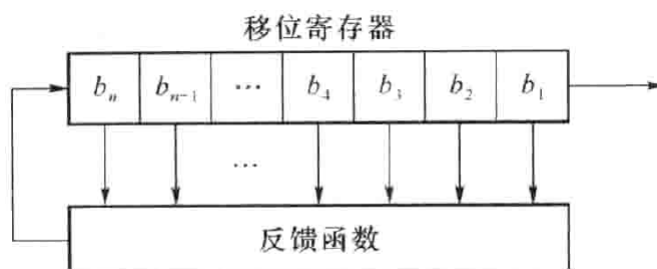
当明文矩阵不可逆时不计算密钥。

2 序列密码

2.1 线性反馈移位寄存器

2.1.1 算法原理

LFSR 是属于 FSR（反馈移位寄存器）的一种，除了 LFSR 之外，还包括 NFSR（非线性反馈移位寄存器）。FSR 是流密码产生密钥流的一个重要组成部分，在 $GF(2)$ 上的一个 n 级 FSR 通常由 n 个二元存储器和一个反馈函数组成，如下图所示：

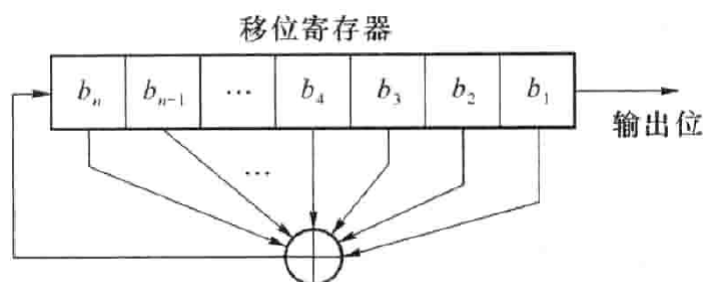


如果这里的反馈函数是线性的，我们则将其称为 LFSR，此时该反馈函数可以表示为：

$$f(a_1, a_2, \dots, a_n) = c_n a_1 \oplus c_{n-1} a_2 \oplus \dots \oplus c_1 a_n$$

其中 $c_i \in \{0,1\}$ ， \oplus 表示异或。

LFSR 如下图所示：



2.1.2 代码实现

1) 反馈函数

```
@staticmethod
def feedback(reg, taps) -> int:
    res = reg[taps[0] - 1]
    for i in range(1, len(taps)):
        res = int(res) ^ int(reg[taps[i] - 1])
    return res
```


2) 运行函数

```
def run(self) -> None:
    for i in range(pow(2, self._reg_len) - 1):
        self._output.append(self._shift_reg[-1])
        input = str(self.feedback(self._shift_reg,
self._coefficients))
        self._shift_reg = input + self._shift_reg[:-1]
        self._regs.append(self._shift_reg)
```

2.1.3 正确性检验与周期分析

首先我们可以通过网站快速地寻找本原多项式，这里推荐
http://wims.unice.fr/wims/cn_tool~algebra~primpoly.cn.html

在 \mathbb{F}_2 上总共有 24000 个 20 次本原多项式.

搜索结果: [往后>>](#)

1	$x^{20} + x^3 + 1$
2	$x^{20} + x^6 + x^4 + x + 1$
3	$x^{20} + x^6 + x^5 + x^2 + 1$
4	$x^{20} + x^6 + x^5 + x^3 + 1$
5	$x^{20} + x^6 + x^5 + x^4 + x^3 + x + 1$

我们选择一个 20 次的本原多项式 $x^{20} + x^6 + x^4 + x + 1$ 。

请输入本原多项式x的系数(以空格分隔):

20 6 4 1

输入的本原多项式为:

$x^{20}+x^6+x^4+x^1+1$

其理论最大周期为: 1048575

确认使用这个本原多项式吗?

[Y]确定 [N]重新输入 [Q]退出

Y

周期为: 1048575

是否查看输出序列?

[Y]是 [N]否

N

是否输出周期内寄存器各状态?

[Y]是 [N]否

Y

成功! 周期内寄存器各状态保存在regs.txt

因为 $1048575 = 2^{20} - 1$, 所以程序正确。

1	00000000000000000001
2	10000000000000000000
3	11000000000000000000
4	11100000000000000000
5	11110000000000000000
6	01111000000000000000
7	10111100000000000000
8	11011110000000000000
9	11101111000000000000
10	01110111100000000000

1048566	0000000001000000000
1048567	0000000001000000000
1048568	0000000001000000000
1048569	0000000001000000000
1048570	0000000001000000000
1048571	0000000001000000000
1048572	0000000001000000000
1048573	0000000001000000000
1048574	0000000001000000000
1048575	<u>0000000001000000000</u>

可以看到第 1048575 个寄存器状态的下一个状态即为初始状态，为一个周期。

2.1.4 安全性与可用性分析

如果反馈多项式是稀疏的，即其项数较少，这常常会使算法变弱，稠密的反馈多项式能够增强算法的安全性。

2.1.5 破解或攻击方式分析

m 序列（周期达到最大值 $2^n - 1$ 的序列）本身是适宜的伪随机序列产生器，但在已知明文攻击下可破译反馈多项式。

假设敌手已知 $2n$ 位明密文对，则可确定一段 $2n$ 位长的密钥序列，由此可完全确定出反馈多项式的系数。

如果用 a_i 表示反馈多项式的系数，而且已知 $2n$ 位输出密钥序列，如 k_1, k_2, \dots, k_{2n} ，则可以得出下面 n 个式子

$$\begin{cases} k_{n+1} = k_1 a_n + k_2 a_{n-1} + \dots + k_n a_1 \\ k_{n+2} = k_2 a_n + k_3 a_{n-1} + \dots + k_{n+1} a_1 \\ \vdots \\ k_{2n} = k_n a_n + k_{n+1} a_{n-1} + \dots + k_{2n-1} a_1 \end{cases}$$

上式中的加为模 2 加，下同。现将上述 n 个式子写作矩阵的形式，可得

$$\begin{bmatrix} k_{n+1} \\ k_{n+2} \\ \vdots \\ k_{2n} \end{bmatrix} = \begin{bmatrix} k_1 & k_2 & \dots & k_n \\ k_2 & k_3 & \dots & k_{n+1} \\ \vdots & \vdots & & \vdots \\ k_n & k_{n+1} & \dots & k_{2n-1} \end{bmatrix} \begin{bmatrix} a_n \\ a_{n-1} \\ \vdots \\ a_1 \end{bmatrix} = \mathbf{K} \begin{bmatrix} a_n \\ a_{n-1} \\ \vdots \\ a_1 \end{bmatrix}$$

上面 n 个式子相当于 n 个线性方程，并且只有 n 个未知数： a_1, a_2, \dots, a_n ，可以证明如果矩阵 \mathbf{K} 是可逆的，那么可唯一解出 $a_i (i = 1, \dots, n)$ ：

$$\begin{pmatrix} a_n \\ a_{n-1} \\ \vdots \\ a_1 \end{pmatrix} = \mathbf{K}^{-1} \begin{pmatrix} k_{n+1} \\ k_{n+2} \\ \vdots \\ k_{2n} \end{pmatrix} = \begin{pmatrix} k_1 & k_2 & \cdots & k_n \\ k_2 & k_3 & \cdots & k_{n+1} \\ \vdots & \vdots & \ddots & \vdots \\ k_n & k_{n+1} & \cdots & k_{2n-1} \end{pmatrix}^{-1} \begin{pmatrix} k_{n+1} \\ k_{n+2} \\ \vdots \\ k_{2n} \end{pmatrix}$$

由此可见，采用线性寄存器产生的序列密码在已知明文攻击下是可以破译的，所以密钥序列产生器还需要非线性组合部分。

2.2 LFSRXOR

2.2.1 分析

题目用两个 LFSR 的输出序列作为两个密钥，后面又打乱了密钥的排列顺序。

通过 LFSR 的构造语句，可以使用我们自己的 LFSR 去获取周期

请输入本原多项式 x 的系数(以空格分隔)：

4 3

输入的本原多项式为：

$x^4 + x^3 + 1$

其理论最大周期为：15

确认使用这个本原多项式吗？

[Y]确定 [N]重新输入 [Q]退出

Y

周期为：15

请输入本原多项式 x 的系数(以空格分隔)：

5 4 2 1

输入的本原多项式为：

$x^5 + x^4 + x^2 + x + 1$

其理论最大周期为：31

确认使用这个本原多项式吗？

[Y]确定 [N]重新输入 [Q]退出

Y

周期为：31

那么生成密钥的长度也分别为 15 和 31。

在 `encode` 函数中，将 `content` 的每一个字节与密钥 `key` 中的对应字节进行异或，密钥 `key` 循环使用，得到密文 `enc`。

而 `content` 是一段随机字符串连上 `flag`，因此 `content` 的最后一个字节一定是右花括号`}`。

那么，将 `enc` 的最后一个字节与`}`异或后，就可以得到密钥中的对应字节。

题目将明文使用两个密钥加密得到两个密文，根据异或关系，可以从一个密钥得到另一个密钥。推导过程如下：

$$m_i \oplus k_{4i \% 15} = enc1_i$$

$$m_i \oplus k_{5i \% 31} = enc2_i$$

$$k_{4i \% 15} \oplus k_{5i \% 31} = enc1_i \oplus enc2_i$$

$$k_{4i \% 15} = k_{5i \% 31} \oplus enc1_i \oplus enc2_i$$

那么我们就可以得到 k_4 的所有字节。

2.2.2 脚本

```
# from pylfsr import LFSR
import string

# 这里用自己的 LFSR 看长度
# L4 = LFSR(fpoly=[4, 3], initstate='random', verbose=True)
# L4.info()
#
# L5 = LFSR(fpoly=[5, 4, 2, 1], initstate='random', verbose=True)
# L5.info()

chars = string.ascii_letters + string.digits + '{} '
enc1 =
b'\xbb\xd3\x08\x15\xc6:\x08\xb2\xb2\x9f\xe4p\xc7\xec\x7f\xfd)\xf6f\x9c\xe4\x
d12\xaeJ\x81\xb1\x88\xab\xa5V\xa9\x88\x14\xdf`~\xf6\xdbJ\xb4\x06S!0\xbb\xe4\
x1a\xe6R\x8e\x84X\x19K\x95\x07C\xe8\xb2'\xa9\x80\x15\xec\x8f\x8dY\nK\x85\x9
9\xb7!\x134\xa9\xb6\x15\xcf&\r\x9b\xe1\x99\xe4]3h~\xf0\xa9\xa5\x14\xee}\xd19
l\x14h\x07v
*a0\x12\x14\xfe\x0f\x05\xdem\x1d\xe4s2J\x7f\xc28\xf6RR\x8e\xba\xb2m\x18M\xf1
\xef!4\x17\xa8\xb4\x14\xc2\x8f\xb9Y:K\xaa\x06T!\x1b\xbb\xfd\xf6Gv\x8e\x9a\xe
b\xd9K\xbb\x06N\x9a\x82c\xa9\xa0\x14\xed!\x04\xdbm\x13\xe5w3B\x7f\xd0\xa9\xb
f\xb7\x9c\xe3\xd00\x83K\x86\xab3\x7f\xc1\xbb\xfd\x11\x15\xdf\x8e\x80Y\x07\xd
8\xe5]2m\xe9\xbb\xce`\x91o\x8f\x8cY!\x81\xe4J\x92\x8c\xa7T\x16E\x15\xf1WMY(\
```

xb8[\x8e2y~\xcbM\x10\x15\xc7\x1fWY\x0cK\x87\xce\xe5 !b\xa8\x83\x14\xec6\xd1!
\xc8\x905\xe52L\xf1\xba\xcf\n\x9d\x9d\xe7u\xadm\x06\xe4n2r\xd8\xba\xed\xf6\x
7f\x9d\xd8\xd02m\x12G\x07Y\x89\x7f\x00\xa8\xa4\x15\xe5\x043Y\x1eJ\xae\x07n\x
94\x87\xbb\xcf_\x8d\x9d\xd1\x14Y,\x9e\xe5b\xd7\x8c\x7f\xf7\xa8\x8f\x14\xc7\x
8f\xb3\xb6\xf1\x93\xe40\xdd\xc4\xdb\xba\xf6!\x15\xfd.\xd1\x18\xcf\xf6\x03\xe
a2E\x7f\xe1\xa9\xa5\xfe\x9d\xc9\xd1;\xd9\xee\x05\x06z\xc8\xb2\xbb\xe2\xf7{JW
4\xcdm\x1a\xe5U\x8d
\x0f&\x14\x7f\xf6\x9d\xd4E\xbf\xc3\xdb\xe4L\xe1\xf7\x90\xbb\xdaZ\xf4\x9d\xd1
3\xb8m3\xe2D3o~\xf8H\xf6U*\x07LY\x03K\xab\x07~\xa3\x87\xbb\xc9\xf7sAQ\x08Y6J
\x86\x07Y\xec\xf7\xbb\xc6s\x15\xc6\x7fEY\x02J\x95\x07Z
\x11\xbb\xc6T\x15\xfc-\xd0\x06\xe6\x9f-\x07^
\x15\xbb\xccz\x14\xf3\x8f\x97\xd4l9t\x85\xe8\x8a\xbe\xbb\xf9\xf6f\x9d\xf2\xd
19\xa2K\xb6\xcd\xcf\xf6~\xd5\xa9\xaa\x15\xd8\x8e\xb3\x81m9\xe4f\xb2!\x1e\xba
\xd8s\xfd\x11\x08W\xa1l;\x01\x07_!\x11\xbb\xdd\xf6x\x9d\xf0\x17Y\x15\xfe\x02
\xc7\xa0!.W\xa9\xa5\x8f\x9c\xe8\xd1\x12m\x04\xe5s3Q~\xdd\xa9\xa3\x15\xdb\x8f
\xac\xaf\xec\xbb\x10\xde2_\xba\xba\xe8\xf6f.\x1e\xd1\x17l\x06\xe4U\xdd\xf0\x
d6~\x0fA\x14\xcb\x8e\xb0Y\x1fJ\xb2\xe4\xb3!"\xba\xfeU\x14\xedY\xd0>l-~\x06P
1\xbb\xf2\xf6wad\xd1(m\x12`\x06@\xb6~\xfa\xa9\xb1\xb0\x9d\xfb\x18\xfbm&\xe4v
2w\xce\xba\xcb0\xd5\x07\x11QX<J\xbd\xb220\x7f\xd8x>\xc8\x9c\xd3\xd03\x9d\xb5
\x1e\xd72S\xf2ry\xf1W\x9c\x89Y\rK\x8f\xff\x8a\xe0\xb5{\xa9\xae\xb1\x9d\xdd\
xd1=\xbeK\xa3\x06e!\x08\xba\xd2\xf6j\x9c\xf6\xd0\x0fl#\xe5o\xf5\xaa~\xc2\xa9
\x99\x15\xea6\xd1:\xe7\xa8\xe4n\xbb
\nV\xa9\x91\x14\xf9}\xd0!m/\xe5|2o\x81\xba\xf8\r\x14\xeb\tR\x09\xec\xdd` \xbf
\xc6\x81\xdfKXW\xb3o.%\xa9\xcd\xb9\x14\xfd\x97\x83\x8e0\n\x03\xb6iuu\xab\x9d
\xbc\x15\xf4\xc3\xd6\xc1'
enc2 =
b'p\xfd\x1ff\xcaB\xa5\xe6`\x87\xa8\x8ci\x855\x9208P\xa5}^ \xd8\xed\x1a\x88=c\
xe0\x9f\xedq\xf8\xe1%\x7fX\xd2\xba\xbe\x03\xa8\x9a\x9c\x075\x98"\xca\xed\xa4
C^\xc6.j\xec\xfa\x10\xa7\xd9\x01\x06\x87\x90f\xcc\xf6\x1b\x0c\xde\xcc,\xfb\x
f0\xc74\x94\xcfj\x8ay\xd5\xd2`.@\xed\xc2\xd8!Dsp\xf5\x12f\xf1\xf6#\x80\xbe\x
16\xa8\xaeF\xd0\xd1\xd4\xad\xb9\xf7#\x16\x08\xb2[\x1a\x87\x8b\xa0\xfaEF\xbf\
x86\x8b\x8c\x90\xa4\xd5\xfbcr\xe2W\x9c\n5\x8b\xcfQ"\xf2\x16\x10\xb2I\x1a\x88
\x8b\x8cj\x16\xebp\xccS\xd2\x90\xa8|q\x05\xafq\xfa\xcaHE{\x1a\xba#\xfd\x17\
xb2L\x1a\x87\x8a\x90\x9Dmp\xef\x0ef\xf2Z|S\x00R\xfc\x1c\x9d\n5\x84\xceS\xb0
\xa4M_\xff\xb9\x1a\x8a\x1d\\ \x98D\|p\xcb*f\xdcV\xd0\xd5Q\xec\x1a\xfa\xf0\x91
\xa8\xd4\x8a\xca\x9c-
\x17\x07\xb2_\xff\n\x8a\x83\xfb\xc2\x00\x10\x87\x83\xaeF\xf7#\xd4\xbe'\xa9\
x8a\$Imp\x14\xe8\x00\xa4z\xd1\xb2H\xe6e\x8b\xb0\xcf\xb1\x01<\x87\x88g\xc2Q|H\
xbe9\xa9\xad\x9c#4\x8cl8I\x0c\x17\$\xb3}\x1b\x94\x01:j7\x00;\x86\xbd\xd2i\xf6
\x1a\xa4' R\xf6?\x9c\x08\xe1\xd4\xab\xdd\x8f\xa4[_\xca/@\xed\xe86\xf7\x9c\x0
18i\x04\xc3\x90\xa8\xaa\x0c\xde\xf2\xa8\xba?\xf4\xd39\xce\\ "\xfe\x16\x0cY/]\
xed\xe9l\xce\xa5\x018o,g\xdb\xf7\x12\xdag\xb6=\xfa\xccHgk\xcfH\xbf\x18\x9e\x
bd\xb3u\x8f\n\$Hk\x0e\xd3\xa6i\xe1\x15=\x16}R]\xb3\xa8\x82\x9b\x0b4\x9a\xcf{\

```
xc2\xa4V\xe8:\x93\x1a\x83\x8a\x97j\t\x82\x88\x86\x80f\xf6*\xa2\xd5\xbe\x08\x
a9\x98\x9c#\xf8\\\xceV\xa7\xa5L\xae&/t\xec\xfb\xd9\x02Dnp\xe8Cf\xf0U}R4\x87a
\xfb\xf0I_\xd4\xaa\xb4"\xca\x16\x18>/i}\t\x03\xc1\x84\x00!\x86\x93g\xed\xf7\
x1d\xc3\xbf\x01c\x06KI[\xd5\x929g\xa4t\x87\xb2\\\x1b\x8d\x0b\xd9\x0bDp\xf5om
\xe1\x16\x0e}|ZR\xc4\xfb\xf2H@\xd4\xa28\\c\x17&\x07\xc8\xda~\x8b\x88\x86DS\x
eb\x87\x87f\xda\xf73\r\xcaS\xd9\xfa\xfaI`\xd5\x889^R\x97\xaeF\xf6\x1a\x92N\x
d8*Er\xc3\x16\xe0)\x91\xba|_Q\x83\x00>;\xff5\x82\xceX"\xd7\x17\x08P\xae\x1a\
xb1\x8a\x8f\xc9Ep\xa7\x86\x86g\xf6m|o\xbf\x1c\xa9\xa1\x9c+\xc9\x1e\xcfI#\xfc
\x92^\xc1\xb8\x1b\xad\x8a\x9e\xceEu\xb8$\xe0\x0b\x90\x87}[\x0fS\xcab]\xd2\xa
aU\xcfh"\xfc\xa2_\xdd/y<C\x05k\x18\x00\x1aw\x1e\x9cA\xf6\x0f\x80w\x83\xae\xb
8\x9d\x0e\xdc\xd4\xaf9H\\\xaf\x9ey\xef\x1b\xb4.\xd99Dd\xa2\x87\xa7f\xc6\xf6\
n\x0c\xc4R\xd7\xfa\xe4Hc\xd4\xa78Jc\x9c^\xca.u\xed\xfcak&\x8b\x92\x87\x88\xe
e\x90\x83\x90\x0c\xd9R\xcd\x08\x9c04\xb1\xceC"\xea\xe9^\xe3\xd4\x1a\x9a\x0c[
\xfa\xc5\x97\xf5>\x15\xc71\x06\x8d\xac\x19\xa0\t\x0e1\xe9\xc6%4\x9d\x80U\xe3
\xfdF\x8d\xee\x17.+ \x9b\xb3\xf0\x83w\x16\xd9'
```

```
k4_len = 15
```

```
k5_len = 31
```

```
k4 = [0] * k4_len
```

```
k5 = [0] * k5_len
```

```
# 当然这里用 enc1 推也是一样的
```

```
enc2_length = len(enc2)
```

```
k5_known = enc2_length % k5_len - 1
```

```
k5[k5_known] = ord('}') ^ enc2[enc2_length - 1]
```

```
for i in range(k5_known, enc2_length, k5_len):
```

```
    k4[i % k4_len] = k5[k5_known] ^ enc1[i] ^ enc2[i]
```

```
for i in range(enc2_length):
```

```
    if chr(k4[i % k4_len] ^ enc1[i]) in chars:
```

```
        print(chr(k4[i % k4_len] ^ enc1[i]), end='')
```

```
    else:
```

```
        print('.', end='')
```

2.2.3 运行结果

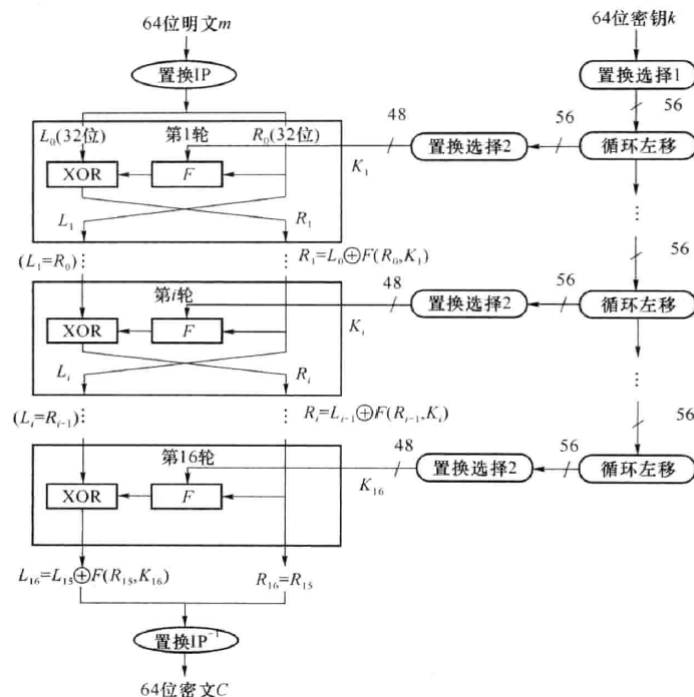
```
.....6...B.....uyI.....mlt.....1.....F..L...k..N.....0...N.....B.D.....S.....k5...o.....qv...
X.....L.A.....o...}.....Q...U..UF1.....cn..}p.....1}J.....RD.....{..g...M...J..f7.....d...B...K.....I.....v...jZ...
n.....g...C.....vg.....W.....J...V.lR.....o.....tJ.....A.....K..F..2V.....a.....I.....0...
N...RA.h.....C...F..0ED.....w..Q.....X.....5026.....pS.....j..dt.....B.....X.....F..T...
g...a.....r..Z.Y.B...v.....2.8...N.....{.....f.....H.....{..H..J.....0.....8...DASCTF{7cc33bd1c63b029fa27a6a78f1253024}
进程已结束, 退出代码0
```

3 分组密码

3.1 DES

3.1.1 算法原理

DES 使用的分组密码种常见的 Feistel 网络，分组长度 64 比特，密钥长度 64 比特（其中有效长度为 56 比特），第 8、16、24、40、48、56 和 64 位是奇偶校验位。DES 加密流程大致如下图所示，其中右半部分的每轮 48 位子密钥由密钥编排算法生成：



3.1.2 代码实现

秉持着一切密码皆有其对应的密码机这样的原则，我在这里仍使用面向对象编程，将作为基本部件的各种盒子作为 DES 类的属性，加密、解密与工作模式作为其方法。

1) 加密

```
def encrypt(self, message, key):
    """
    DES 中每组的加密算法
    @param message: 64 位明文
```



```

    @param key: 密钥
    @return: 64 位密文
    """
    self.message_process(message, 64)
    key = self.key_process(key)
    key_list = self.get_key(key)
    ip_out = self.ip(message)
    l_part = ip_out[:32]
    r_part = ip_out[32:]
    for i in range(15):
        l_part, r_part = r_part, self.xor(l_part, self.func_f(r_part,
key_list[i]))
        l_16, r_16 = self.xor(l_part, self.func_f(r_part, key_list[15])),
r_part
    cipher = self.ip_inverse(l_16 + r_16)
    return cipher

```

2) 解密

```

# Feistel 网络的优点在于加解密相似性，这里可以简化掉
def decrypt(self, cipher, key):
    """
    DES 中每组的解密算法
    @param cipher: 64 位密文
    @param key: 密钥
    @return: 64 位明文
    """
    key = self.key_process(key)
    key_list_inverse = self.get_key(key)[::-1]
    ip_out = self.ip(cipher)
    l_part = ip_out[:32]
    r_part = ip_out[32:]
    for i in range(15):
        l_part, r_part = r_part, self.xor(l_part, self.func_f(r_part,
key_list_inverse[i]))
        l_16, r_16 = self.xor(l_part, self.func_f(r_part,
key_list_inverse[15])), r_part
    message = self.ip_inverse(l_16 + r_16)
    return message.strip('\x00')

```

3) 用到的几个方法说明：

`self.message_process`: 预处理，保证明文的长度能够完整的分组

`self.key_process`: 预处理，保证密钥的长度为 64 位。

`self.get_key`: 密钥生成算法

`self.ip`: IP 置换

`self.xor`: 异或

`self.func_f`: F 函数

`self.ip_inverse`: IP 逆置换

3.1.3 正确性检验与性能分析

与其它分组密码相似，只能对单个分组加密的 DES 自身并不是实用的算法，而必须以某种工作模式进行实际操作，在下一节中我将使用 5 种工作模式分别对 DES 进行加解密并验证，所以这里暂不验证。

3.1.4 安全性与可用性分析

1) 安全性

① 互补性：利用这一性质，如果用某个密钥加密一个明文分组得到一个密文分组，那么用该密钥的补密钥加密该明文分组的补便会得到该密文分组的补。互补性会使 DES 在选择明文攻击下所需的工作量减半，仅需要测试 256 个密钥的一半，及 255 个密钥。

② 弱密钥：DES 的解密过程需要用到 56 位初始密钥生成的 16 个子密钥，如果给定初始密钥 K ，经过子密钥生成器得到的各个子密钥都相同，即有 $K_1 = K_2 = \dots = K_{16}$ ，则称给定的初始密钥 K 为弱密钥。如果 K 为弱密钥，则对任意的 64 位数据 M ，有

$$E_K(E_K(M)) = M$$

$$D_K(D_K(M)) = M$$

这说明以 K 对 M 加密两次或解密两次相当于恒等映射，结果仍是 M ，这也意味着加密运算和解密运算没有区别。除此之外，还有半弱密钥、四分之一弱密钥和八分之一弱密钥，他们的总和为 256 位。

③ 密钥空间大小：随着现代计算机运算能力的不断提升，DES 实际 56 位的密钥已经远远不够了，甚至可以用穷举法进行攻击。

2) 可用性

尽管 DES 自诞生起就经历了许多很强的分析攻击，但是至今并未发现一种能够高效破解它的攻击方式。不过，由于现代计算能力利用穷举法就可能容易的破解传统 DES。因此，对大多数实际应用场景来说，传统 DES 都已经不再适用。

3.1.5 破解或攻击方式分析

除了最传统的暴力破解之外，还有如下两种已知方法可以对 DES 进行攻击：

- ① 差分分析：即系统地研究明文中一个细小变化是如何影响密文的。
- ② 线性分析：即寻找明文、密文和密钥间的有效线性逼近，当该逼近的线性偏差足够大时，就可以由一定量的明密文对推测出部分密钥信息。

3.2 工作模式

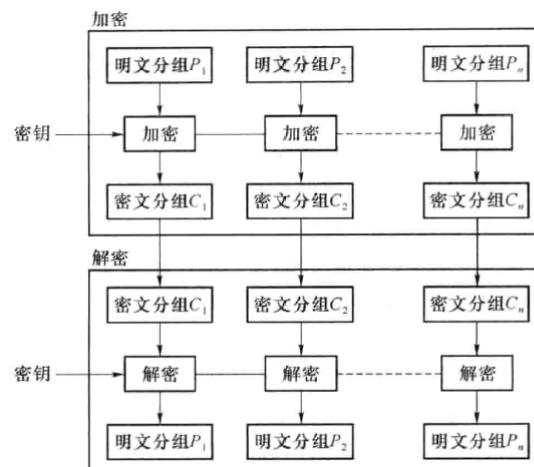
本节在上一节的基础上，分别实现了在 ECB、CBC、CFB、OFB、CTR 五种工作模式下的 DES 加解密，并验证正确性，然后再对这五种工作模式的可用性、安全性与性能效率进行分析。

3.2.1 电子密码本模式（ECB）

3.2.1.1 原理

这是分组加密最简单的一种模式，即明文的每一个块加密成密文的每一个块。明文被分为

若干块($M_1, M_2, M_3, M_4, \dots, M_n$)，通过加密方法 E_k ，得到密文($C_1, C_2, C_3, C_4, \dots, C_n$)，当最后一个明文分组小于分组长度时，需要用一些特定的数据进行填充。



3.2.1.2 代码实现

1) 加密

```
def ecb_encrypt(self, message, key, output_mode='h'):
    """
```

使用电子密码本(ECB) 模式进行DES 加密,即使用相同的密钥分别对明文分组独立加密

```
@param message: 明文(字符串)
@param key: 密钥(字符串)
@param output_mode: 密文输出方式
@return: 密文
"""
# 转换为 2 进制,并补全到 64 位的倍数
message_bin = self.message_process(self.str2bin(message), 64)
# 将密钥转为 2 进制
key_bin = self.str2bin(key)
# 将明文分组
message_list = re.findall(r'.{64}', message_bin)

cipher_bin = ''
for m_i in message_list:
    cipher_bin += self.encrypt(m_i, key_bin)

cipher = ''
if output_mode in ('H', 'h'):
    cipher = self.str2hex(self.bin2str(cipher_bin))
elif output_mode in ('B', 'b'):
    cipher =
base64.b64encode(self.bin2str(cipher_bin).encode('latin')).decode('latin')
return cipher
```

2) 解密

```
def ecb_decrypt(self, cipher, key, input_mode='b'):
    """
    对使用电子密码本(ECB) 模式的DES 进行解密
    @param cipher: 密文(base64/hex)
    @param key: 密钥(字符串)
    @param input_mode: 密文输入方式
    @return: 明文(字符串)
    """
    cipher_bin = ''
    if input_mode in ('h', 'H'):
        cipher_bin = self.str2bin(self.hex2str(cipher))
    elif input_mode in ('b', 'B'):
        cipher_bin = self.str2bin(base64.b64decode(cipher).decode('latin'))
    # 将密钥转为 2 进制
    key_bin = self.str2bin(key)
    # 将密文分组
    cipher_list = re.findall(r'.{64}', cipher_bin)
```

```

message_bin = ''
for c_i in cipher_list:
    message_bin += self.decrypt(c_i, key_bin)

# 明文 2 进制转字符串
message = self.bin2str(message_bin).strip("\x00")
return message

```

3.2.1.3 正确性验证

DES加密/解密

china

模式	ECB ▾	填充	ZeroPadding ▾	偏移量	ECB模式不需要	密文编码	Base64 ▾
密钥	abc				↓ 加密	↑ 解密	清空

4D6IKbLoYEM=

```
Please choose a mode:
[E]Encrypt [D]Decrypt [Q]Quit
E
Please input the message:
china
Please input the key:
abc
Please choose a working mode:
[1]ECB [2]CBC [3]CFB [4]OFB [5]CTR
1
Please choose the output way:
[B]Base64 [H]Hex
B
The cipher is:
4D6IKbLoYEM=

Please choose a mode:
[E]Encrypt [D]Decrypt [Q]Quit
D
Please input the cipher:
4D6IKbLoYEM=
Please input the key:
abc
Please choose a working mode:
[1]ECB [2]CBC [3]CFB [4]OFB [5]CTR
1
Please choose your input way:
[B]Base64 [H]Hex
b
The message is:
china
```

3.2.1.4 安全性与可用性分析

1) 安全性

ECB 模式最大的问题在于它的加密是高度确定的，因此只要密钥不变，相同的明文分组总是产生相同的密文分组，这也就使得攻击者容易实现统计分析

攻击、分组重放攻击和代换攻击。在许多实际情形中，尤其是消息较长的时候，ECB 可能是不安全的。

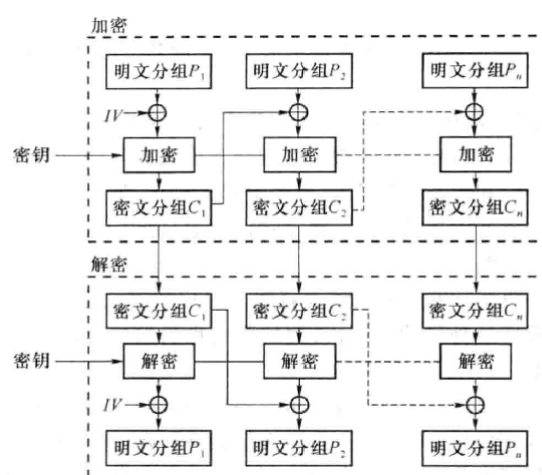
2) 可用性

ECB 模式拥有很多优点，加密方和解密方之间的分组同步不是必须的，每个明文分组可以独立地进行加密。ECB 还具有良好的差错控制，位错误仅仅影响对应的分组，其对后面的分组没有任何影响。所以，ECB 特别适合数据随机且较少的情况，比如加密密钥。

3.2.2 电子密码本模式（ECB）

3.2.2.1 原理

在 CBC 模式中，每个明文块先与前一个密文块进行异或后，再进行加密。在这种方法中，每个密文块都依赖于它前面的所有明文块。同时，为了保证每条消息的唯一性，在第一个块中需要使用初始化向量。



3.2.2.2 代码实现

1) 加密

```
def cbc_encrypt(self, message, key, iv, output_mode='h'):  
    """  
    使用密码分组链接(CBC)模式进行加密, 它将前一个分组的加密结果反馈到当前分组的加密中  
    @param message: 明文(字符串)  
    @param key: 密钥(字符串)  
    @param iv: 初始向量(字符串)  
    @param output_mode: 输出方式  
    @return: 密文  
    """  
    # 将密钥转为 2 进制
```

```

key_bin = self.str2bin(key)
# iv 处理方式与 key 相同(少于 8 字节补 0, 大于 8 字节截断)
iv_bin = self.key_process(self.str2bin(iv))
# 将明文进行预处理
message_bin = self.message_process(self.str2bin(message), 64)
# 将明文分组
message_list = re.findall(r'.{64}', message_bin)
# 第一组的反馈即为 iv

reg = iv_bin
cipher_bin = ''
for m_i in message_list:
    # 与反馈进行异或
    encrypt_m_i = self.xor(reg, m_i)
    reg = self.encrypt(encrypt_m_i, key_bin)
    # 链接密文分组
    cipher_bin += reg

cipher = ''
if output_mode in ('H', 'h'):
    cipher = self.str2hex(self.bin2str(cipher_bin))
elif output_mode in ('B', 'b'):
    cipher =
base64.b64encode(self.bin2str(cipher_bin).encode('latin')).decode('latin
')
return cipher

```

2) 解密

```

def cbc_decrypt(self, cipher, key, iv, input_mode='b'):
    """
    对使用密码分组链接(CBC)模式的DES 进行解密
    @param cipher: 密文
    @param key: 密钥
    @param iv: 初始向量
    @param input_mode: 输出方式
    @return: 密文
    """
    # 将密钥转为 2 进制
    key_bin = self.str2bin(key)
    # 对 iv 进行处理
    iv_bin = self.key_process(self.str2bin(iv))
    # 密文转 2 进制

```



```

cipher_bin = ''
if input_mode in ('h', 'H'):
    cipher_bin = self.str2bin(self.hex2str(cipher))
elif input_mode in ('b', 'B'):
    cipher_bin =
self.str2bin(base64.b64decode(cipher).decode('latin'))
# 对密文进行分组
cipher_list = re.findall(r'.{64}', cipher_bin)

reg = iv_bin
message_bin = ''
for c_i in cipher_list:
    decrypt_c_i = self.decrypt(c_i, key_bin)
    m = self.xor(reg, decrypt_c_i)
    message_bin += m
    # 保存当前密文分组,用于下一个分组的解密异或
    reg = c_i

# 明文转换为字符串
message = self.bin2str(message_bin).strip("\x00")
return message

```

3.2.2.3 正确性验证

china

模式

CBC ▾

填充

ZeroPadding ▾

偏移量

256

密文编码

Base64 ▾

密钥

abc

↓ 加密

↑ 解密

清空

2QospNonf5w=

```
Please choose a mode:
[E]Encrypt [D]Decrypt [Q]Quit
E
Please input the message:
china
Please input the key:
abc
Please choose a working mode:
[1]ECB [2]CBC [3]CFB [4]OFB [5]CTR
2
Please input the IV:
256
Please choose the output way:
[B]Base64 [H]Hex
b
The cipher is:
2QospNonf5w=

Please choose a mode:
[E]Encrypt [D]Decrypt [Q]Quit
D
Please input the cipher:
2QospNonf5w=
Please input the key:
abc
Please choose a working mode:
[1]ECB [2]CBC [3]CFB [4]OFB [5]CTR
2
Please input the IV:
256
Please choose your input way:
[B]Base64 [H]Hex
b
The message is:
china
```

3.2.2.4 安全性与可用性分析

1) 安全性

由于引入了反馈，CBC 可以将重复的明文分组加密成不同的密文，克服了 ECB 的弱点，并且使用了随机化向量 IV，有效防止了重放攻击，但是前提是 IV 应该同密钥一样被加以保护，否则攻击者可以欺骗接收者，让他使用不同的 IV，从而使接收者解密出第一块明文分组的某些位取反。

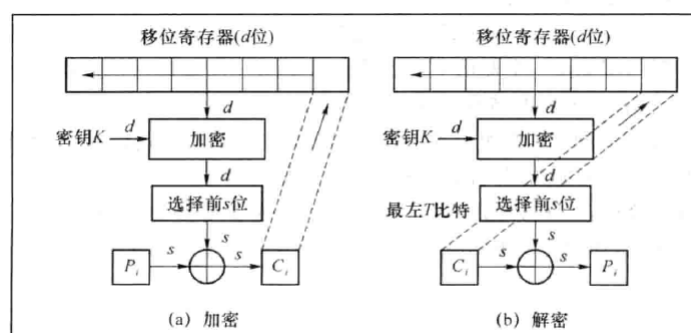
2) 可用性

CBC 模式的没有明文错误扩散，密文错误扩散很小，是最为常用的工作模式，确保互联网安全的通信协议之一 IPsec，就是使用 CBC 模式来确保通信机密性的，此外，CBC 模式还被用于 Kerberos version 5 的认证系统中。

3.2.3 密码反馈模式（CFB）

3.2.3.1 原理

CFB 模式可以看作是一种使用分组密码来实现流密码的方式，CFB 模式中由密码算法所生成的比特序列称为密钥流。在 CFB 模式中，密码算法就相当于用来生成密钥流的伪随机数生成器，而初始化向量就相当于伪随机数生成器的“种子”，它的明文数据可以被逐比特加密。



3.2.3.2 代码实现

1) 加密

```
def cfb_encrypt(self, message, key, iv, output_mode='h'):
```

```
    """
```

密文反馈模式(CFB): 这里选择传输单元为8位(即8位一个分组), 移位寄存器64位, 只需要用到des加密算法。

先在寄存器中填充IV(64位), 然后对寄存器中内容进行加密, 加密结果取出前8位与明文分组(8位)异或得到一个密文分组,

然后寄存器左移8位后, 将前一个密文分组(8位)填充在寄存器最后

@param message: 明文(字符串)

@param key: 密钥(字符串)

@param iv: 初始向量(字符串)

@param output_mode: 密文输出方式(base64/hex)

```

@return: 密文
"""
# 对参数进行预处理
key_bin = self.str2bin(key)
iv_bin = self.key_process(self.str2bin(iv))
message_bin = self.message_process(self.str2bin(message), 8)
# 对明文进行分组(8 位一组)
message_list = re.findall(r'.{8}', message_bin)
cipher_bin = ''
# 初始化移位寄存器
reg = iv_bin
for i in message_list:
    # 对寄存器 64 位进行加密
    enc_out = self.encrypt(reg, key_bin)
    # 选择加密结果的前 8 位与明文分组异或
    c = self.xor(i, enc_out[:8])
    # 寄存器左移 8 位,并在最右边填充前一密文分组
    reg = reg[8:] + c
    # 连接密文分组
    cipher_bin += c
cipher = ''
if output_mode in ('H', 'h'):
    cipher = self.str2hex(self.bin2str(cipher_bin))
elif output_mode in ('B', 'b'):
    cipher =
base64.b64encode(self.bin2str(cipher_bin).encode('latin')).decode('latin')
return cipher

```

2) 解密

```

def cfb_decrypt(self, cipher, key, iv, input_mode='b'):
    """
    对 DES-CFB 进行解密: 先在寄存器中填充 IV(64 位), 然后对寄存器中内容进行加密, 加密结果取出前 8 位与密文分组(8 位)异或得到一个明文分组,
    然后将移位寄存器左移 8 位, 用前一密文分组(8 位)填充再最右边
    @param cipher: 密文(base64/hex)
    @param key: 密钥(字符串)
    @param iv: 初始向量(字符串)
    @param input_mode: 密文输入方式
    @return: 明文
    """
    # 参数预处理

```

```

key_bin = self.str2bin(key)
iv_bin = self.key_process(self.str2bin(iv))
cipher_bin = ''
if input_mode in ('h', 'H'):
    cipher_bin = self.str2bin(self.hex2str(cipher))
elif input_mode in ('b', 'B'):
    cipher_bin =
self.str2bin(base64.b64decode(cipher).decode('latin'))
# 对密文进行分组(8 位一组)
cipher_list = re.findall(r'.{8}', cipher_bin)
message_bin = ''
# 初始化移位寄存器
reg = iv_bin
for c_i in cipher_list:
    # 对移位寄存器进行加密
    enc_out = self.encrypt(reg, key_bin)
    # 选择加密结果的前 8 位与密文分组异或
    m_i = self.xor(c_i, enc_out[:8])
    # 寄存器左移 8 位,并在最右边填充前一密文分组
    reg = reg[8:] + c_i
    # 连接明文分组
    message_bin += m_i
# 明文转换为字符串
message = self.bin2str(message_bin).strip("\x00")
return message

```

3.2.3.3 正确性验证

DES加密模式: CFB
填充: zeropadding
密码: cumt
偏移量: 05191643
输出: base64
字符集: gb2312编码

待加密、解密的文本:

This is WanpengXu's test.

↑ 将你电脑文件直接拖入试试 ^-^

DES加密
DES解密

DES加密、解密转换结果(base64了):

vjSDwORmjikkLt4alvatJp2yS1SemmDeWr+018pYR88=

```
Please choose a mode:
[E]Encrypt [D]Decrypt [Q]Quit
E
Please input the message:
This is WanpengXu's test.
Please input the key:
cumt
Please choose a working mode:
[1]ECB [2]CBC [3]CFB [4]OFB [5]CTR
3
Please input the IV:
05191643
Please choose the output way:
[B]Base64 [H]Hex
b
The cipher is:
vjSDw0RmjikkLt4aIvatJp2yS1SemnDeWg==

Please choose a mode:
[E]Encrypt [D]Decrypt [Q]Quit
D
Please input the cipher:
vjSDw0RmjikkLt4aIvatJp2yS1SemnDeWr+018pYR88=
Please input the key:
cumt
Please choose a working mode:
[1]ECB [2]CBC [3]CFB [4]OFB [5]CTR
3
Please input the IV:
05191643
Please choose your input way:
[B]Base64 [H]Hex
b
The message is:
This is WanpengXu's test.
```

最后一个字符的区别为换行符的区别，对结果无影响

3.2.3.4 安全性与可用性分析

1) 安全性

CFB 模式可以通过重放攻击进行攻击。例如，如果使用其他密文替换新的密文，则用户将获得错误的数 据，但他不会知道数据是错误的。为确保安全性，需要为每 $2^{\frac{n+1}{2}}$ 个加密块更改此模式下的密钥。

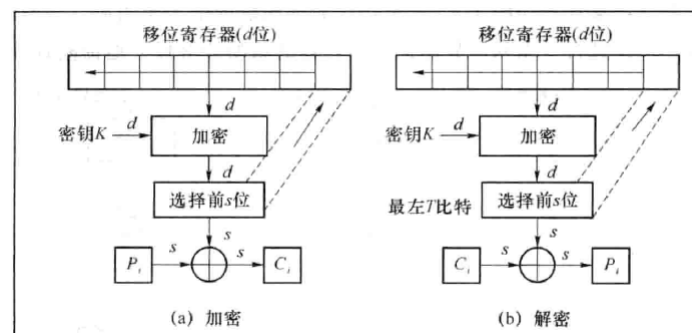
2) 可用性

CFB 模式不会直接加密明文，只是使用密文与明文进行 XOR 来获得密文。所以在 这种模式下，它不需要填充数据，以将块密码变为自同步的流密码，且具有有限的错误传播，这种可以进行实时流操作的模式，在某些情境下具有很大的优势，如面向数据流的通用传输及认证。

3.2.4 输出反馈模式（OFB）

3.2.4.1 原理

在 OFB 模式中，密码算法的输出会反馈到密码算法的输入中。OFB 模式不是通过密码算法对明文直接进行加密的，而是通过将明文分组和密码算法的输出进行 XOR 来产生密文分组的。



3.2.4.2 代码实现

1) 加密

```
def ofb_encrypt(self, message, key, iv, output_mode='h'):  
    """  
    输出反馈模式(OFB): 与CFB类似, 不过这里把将加密算法的输出的前8位反馈到移位寄存器, 而不是前一个密文分组  
    @param message: 明文(字符串)  
    @param key: 密钥(字符串)  
    @param iv: 初始向量(字符串)  
    @param output_mode: 密文输出方式(base64/hex)  
    @return: 密文  
    """
```

```

"""
# 对参数进行预处理
key_bin = self.str2bin(key)
iv_bin = self.key_process(self.str2bin(iv))
message_bin = self.message_process(self.str2bin(message), 8)
# 对明文进行分组(8位一组)
message_list = re.findall(r'.{8}', message_bin)
cipher_bin = ''
# 初始化移位寄存器
reg = iv_bin
for m_i in message_list:
    # 对寄存器 64 位进行加密
    enc_out = self.encrypt(reg, key_bin)
    # 选择加密结果的前 8 位与明文分组异或
    c_i = self.xor(m_i, enc_out[:8])
    # 寄存器左移 8 位,并在最右边填充加密结果的前 8 位
    reg = reg[8:] + enc_out[:8]
    # 连接密文分组
    cipher_bin += c_i
cipher = ''
if output_mode in ('H', 'h'):
    cipher = self.str2hex(self.bin2str(cipher_bin))
elif output_mode in ('B', 'b'):
    cipher =
base64.b64encode(self.bin2str(cipher_bin).encode('latin')).decode('latin')
return cipher

```

2) 解密

```

def ofb_decrypt(self, cipher, key, iv, input_mode='b'):
    """
    对 DES-CFB 进行解密:
    @param cipher: 密文(base64/hex)
    @param key: 密钥(字符串)
    @param iv: 初始向量(字符串)
    @param input_mode: 密文输入方式
    @return: 明文
    """
    # 参数预处理
    key_bin = self.str2bin(key)
    iv_bin = self.key_process(self.str2bin(iv))
    cipher_bin = ''

```



```

    if input_mode in ('h', 'H'):
        cipher_bin = self.str2bin(self.hex2str(cipher))
    elif input_mode in ('b', 'B'):
        cipher_bin =
self.str2bin(base64.b64decode(cipher).decode('latin'))
    # 对密文进行分组(8位一组)
    cipher_list = re.findall(r'.{8}', cipher_bin)
    message_bin = ''
    # 初始化移位寄存器
    reg = iv_bin
    for i in cipher_list:
        # 对移位寄存器进行加密
        enc_out = self.encrypt(reg, key_bin)
        # 选择加密结果的前8位与密文分组异或
        m = self.xor(i, enc_out[:8])
        # 寄存器左移8位,并在最右边填充加密结果的前8位
        reg = reg[8:] + enc_out[:8]
        # 连接明文分组
        message_bin += m
    # 明文转换为字符串
    message = self.bin2str(message_bin).strip("\x00")
    return message

```

3.2.4.3 正确性验证

DES加密模式:
填充:
密码:
偏移量:
输出:
字符集:

待加密、解密的文本:

This is a test of WanpengXu.

↑ 将你电脑文件直接拖入试试 ^-^

DES加密
DES解密

DES加密、解密转换结果(base64了):

vgcMv9XHng2q8HGZSb1Yf74KAWbkXqRatiYMMf+d+uI=

```
Please choose a mode:
[E]Encrypt [D]Decrypt [Q]Quit
E
Please input the message:
This is a test of WanpengXu.
Please input the key:
cumt
Please choose a working mode:
[1]ECB [2]CBC [3]CFB [4]OFB [5]CTR
4
Please input the IV:
05191643
Please choose the output way:
[B]Base64 [H]Hex
b
The cipher is:
vgcMv9XHng2q8HGZSb1Yf74KAWbkXqRatiYMMQ==

Please choose a mode:
[E]Encrypt [D]Decrypt [Q]Quit
D
Please input the cipher:
vgcMv9XHng2q8HGZSb1Yf74KAWbkXqRatiYMMQ==
Please input the key:
cumt
Please choose a working mode:
[1]ECB [2]CBC [3]CFB [4]OFB [5]CTR
4
Please input the IV:
05191643
Please choose your input way:
[B]Base64 [H]Hex
b
The message is:
This is a test of WanpengXu.
```

3.2.4.4 安全性与可用性分析

1) 安全性

OFB 抗消息流篡改攻击的能力不如 CFB，因为密文中的某位取反，恢复出的明文的相应位也取反，所以攻击者有办法控制恢复明文的改变。这也，攻击者可以根据消息的改动而改动校验和，以使改动不被纠错码发现。

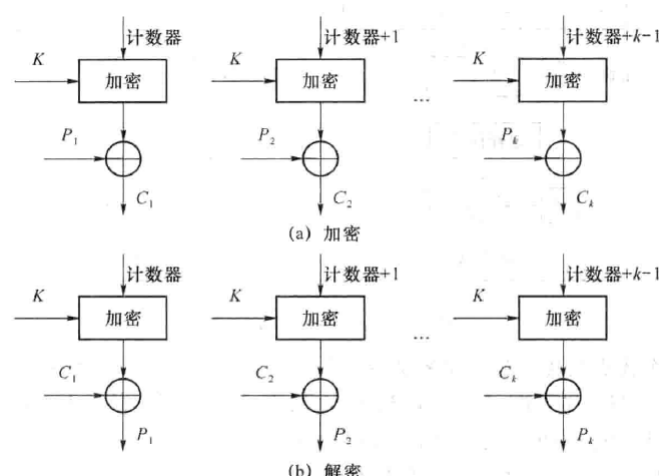
2) 可用性

OFB 模式与 CFB 模式相似，但是将前一次加密产生的 s 比特分组中非密文分组送入移位寄存器的最右边，是一种“内部反馈”机制，其优点是传输过程中密文在某位上发生的错误不会影响解密后明文的其他未，但是在 OFB 中，失去同步是致命的。

3.2.5 计数器模式 (CTR)

3.2.5.1 原理

CTR 模式是一种通过将逐次累加的计数器进行加密来生成密钥流的流密码。CTR 模式中，每个分组对应一个逐次累加的计数器，并通过对计数器进行加密来生成密钥流。也就是说最终的密文分组是通过将计数器加密得到的比特序列，与明文分组进行 XOR 而得到的。



3.2.5.2 代码实现

1) 加密

```
def ctr_encrypt(self, message, key, nonce, output_mode='h'):
    """
    计数器模式(CTR): 使用与明文分组长度相同(这里选择64位为一组)的计数器,对计数器进行加密,得到的结果与明文分组进行异或,可以并行化的加解密
    初始计数器值是一个随机数,后一个计数器值是前一个计数器值加1
    @param message: 明文(字符串)
    @param key: 密钥(字符串)
    @param nonce: 初始化计数器值的随机数
```

```

    @param output_mode: 密文输出方式(base64/hex)
    @return: 密文
    """
    # 参数预处理
    key_bin = self.str2bin(key)
    nonce_bin = self.key_process(self.str2bin(nonce))
    message_bin = self.message_process(self.str2bin(message), 64)
    # 明文分组
    message_list = re.findall(r'.{64}', message_bin)
    cipher_bin = ''
    # 初始化计数器
    counter = nonce_bin # 一直是二进制字符串
    for m_i in message_list:
        enc_out = self.encrypt(counter, key_bin)
        cipher_bin += self.xor(m_i, enc_out)
        # 计数器加 0b1(mod 2^64)
        counter = bin(eval(f'0b{counter} + 0b1 % pow(2,
64)'))[2:].zfill(64)
    cipher = ''
    if output_mode in ('H', 'h'):
        cipher = self.str2hex(self.bin2str(cipher_bin))
    elif output_mode in ('B', 'b'):
        cipher =
base64.b64encode(self.bin2str(cipher_bin).encode('latin')).decode('la
tin')
    return cipher

```

2) 解密

```

def ctr_decrypt(self, cipher, key, nonce, input_mode='b'):
    """
    对DES-CTR 进行解密
    @param cipher: 密文
    @param key: 密钥
    @param nonce: 初始化计数器值的随机数
    @param input_mode: 密文输入方式(base64/hex)
    @return: 明文
    """
    # 参数预处理
    key_bin = self.str2bin(key)
    nonce_bin = self.key_process(self.str2bin(nonce))
    cipher_bin = ''
    if input_mode in ('h', 'H'):

```

```

        cipher_bin = self.str2bin(self.hex2str(cipher))
    elif input_mode in ('b', 'B'):
        cipher_bin =
self.str2bin(base64.b64decode(cipher).decode('latin'))
    # 对密文进行分组(8 位一组)
    cipher_list = re.findall(r'.{64}', cipher_bin)
    message_bin = ''
    # 初始化计数器
    counter = nonce_bin
    for c_i in cipher_list:
        enc_out = self.encrypt(counter, key_bin)
        message_bin += self.xor(c_i, enc_out)
        # 计数器加 1(mod 2^64)
        counter = bin(eval(f'0b{counter} + 0b1 % pow(2,
64)'))[2:].zfill(64)
    # 明文转换为字符串
    message = self.bin2str(message_bin).strip("\x00")
    return message

```

3.2.5.3 正确性验证

DES加密模式: CTR 填充: zeropadding 密码: cumt 偏移量: 05191643 输出: base64 字符集: gb2312编码

待加密、解密的文本:

This is a test of WangepengXu.

↑ 将你电脑文件直接拖入试试 ^-^

DES加密 DES解密

DES加密、解密转换结果(base64了):

vpxJfg5gUSbRVFaaqTkVQggbu7zDBq/wqG+ztQPIRTM=

```
Please choose a mode:
[E]Encrypt [D]Decrypt [Q]Quit
E
Please input the message:
This is a test of WanpengXu.
Please input the key:
cumt
Please choose a working mode:
[1]ECB [2]CBC [3]CFB [4]OFB [5]CTR
5
Please input the nonce:
05191643
Please choose the output way:
[B]Base64 [H]Hex
b
The cipher is:
vpxJfg5gUSbRVFaaqTkVQgqbu7zDBq/wqG+ztQPIRTM=

Please choose a mode:
[E]Encrypt [D]Decrypt [Q]Quit
D
Please input the cipher:
vpxJfg5gUSbRVFaaqTkVQgqbu7zDBq/wqG+ztQPIRTM=
Please input the key:
cumt
Please choose a working mode:
[1]ECB [2]CBC [3]CFB [4]OFB [5]CTR
5
Please input the nonce:
05191643
Please choose your input way:
[B]Base64 [H]Hex
b
The message is:
This is a test of WanpengXu.
```

3.2.5.4 安全性与可用性分析

1) 安全性

CTR 也具有一定的可证明安全性，能够证明 CTR 至少和上述 4 中工作模式一样安全。

2) 可用性

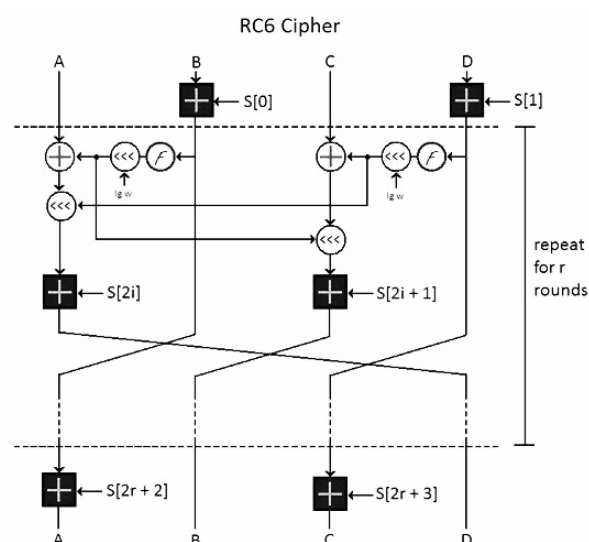
CTR 模式的加密和解密使用了完全相同的结构，因此在软硬件实现上比较容易。

此外，CTR 模式中可以以任意顺序对分组进行加密和解密，能够以任意顺序处理分组，就意味着能够实现并行计算。在支持并行计算的系统中，CTR 模式的速度是非常快的，具有很大的优势。

3.3 RC6

3.3.1 算法原理

我们可以先用一张流程图简单地概括 RC6 算法。



接下来，对图中的各个部件进行说明。

首先，RC6 算法需要 4 个 32 位寄存器 A, B, C, D。

其次，RC6 算法需要四种基本运算，分别是：

- 1) 整数模 2^w 加/减，记为 $+/-$
- 2) 整数模 2^w 乘，记为 \times
- 3) 按位模2加，记为 \oplus
- 4) 循环左移/循环右移，记作 ROL/ROR

这四种基本运算中，前两种可以视为与运算，第三种实质上是异或运算，由此可见，算法的效率不会很低。

4.2.2. 代码实现

1) 加密

```
def encrypt(plaintext, k):
    ciphertext = ''
    S = extend_key(k)
    bin_plaintext = ''.join([bin(ord(ch))[2:].zfill(8) for ch in plaintext])
    bin_plaintext += '0' * (128 - len(bin_plaintext) % 128)
    for l in range(0, len(bin_plaintext), 128):
        A = int(bin_plaintext[l + 00:l + 32], 2)
        B = int(bin_plaintext[l + 32:l + 64], 2)
        C = int(bin_plaintext[l + 64:l + 96], 2)
        D = int(bin_plaintext[l + 96:l + 128], 2)

        B = (B + S[0]) % MOD
        D = (D + S[1]) % MOD
        for i in range(1, r + 1):
            t = ROL(B * (2 * B + 1), int(log2(w)))
            u = ROL(D * (2 * D + 1), int(log2(w)))
            A = (ROL(A ^ t, u) + S[2 * i]) % MOD
            C = (ROL(C ^ u, t) + S[2 * i + 1]) % MOD
            (A, B, C, D) = (B, C, D, A)
        A = (A + S[2 * r + 2]) % MOD
        C = (C + S[2 * r + 3]) % MOD

        ciphertext += f'{A:032b}{B:032b}{C:032b}{D:032b}'
    res = hex(int(ciphertext, 2))[2:]
    print(res)
```

2) 解密

```
def decrypt(ciphertext, k):
    S = extend_key(k)
    plaintext = ''
    for l in range(0, len(ciphertext), 128):
        A = int(ciphertext[l + 00:l + 32], 2)
        B = int(ciphertext[l + 32:l + 64], 2)
        C = int(ciphertext[l + 64:l + 96], 2)
        D = int(ciphertext[l + 96:l + 128], 2)
        C = (C - S[2 * r + 3]) % MOD
        A = (A - S[2 * r + 2]) % MOD
        for i in range(r, 1 - 1, -1):
```



```
(A, B, C, D) = (D, A, B, C)
u = ROL(D * (2 * D + 1), 5)
t = ROL(B * (2 * B + 1), 5)
C = ROR(C - S[2 * i + 1], t) ^ u
A = ROR(A - S[2 * i], u) ^ t
D = (D - S[1]) % MOD
B = (B - S[0]) % MOD
plaintext += f'{A:032b}{B:032b}{C:032b}{D:032b}'

res = ''
for i in range(0, len(plaintext), 8):
    res += chr(int(plaintext[i:i + 8], 2))
print(res)
```

4.2.3. 正确性检验与性能分析

请选择: [E]加密Encrypt [D]解密Decrypt

E

请输入明文:

china

请输入密钥(小于32字节):

abc

f7d2a64ef2c17c46314de739baf68f01

请选择: [E]加密Encrypt [D]解密Decrypt

D

请输入密文:

f7d2a64ef2c17c46314de739baf68f01

请输入密钥(小于32字节):

abc

china

4.2.4. 安全性与可用性分析

首先我们知道分组密码的本质是混淆和扩散。混淆是指打乱密文、明文、密钥之间的依赖关系。扩散是指明文的统计特性消散在密文中，每个明文比特尽可能的影响多个密文，密文每个比特受多个明文比特影响。

作为 RC5 强化版的 RC6 通过引入乘法运算来决定循环移位次数的方法，对 RC5 进行改进，弥补了 RC5 在扩散速度上的不足，并且 RC6 中的非线性部分是由多个部件共同实现的，这都大大增强了 RC6 的安全性。

接着，我们考虑弱密钥的可能性。RC6 的扩展算法将用户密钥用一个伪随机过程加以扩展，尽管还没有证明两个密钥不会产生相同的循环密钥表，但是这种可能微乎其微。

当 $w/r/b$ 三者为典型值，即 $w = 32$, $r = 20$, $b = 32$ 时，密钥为 256 位，两个 256 位的密钥产生一个相同的 44 个 32 位循环密钥的概率，根据密钥长度的全部可能性对应的集合大小进行比较运算，约为 $\frac{2^{2*256}}{2^{44*32}} = 2^{2*256-44*32} = 2^{-896} \approx 10^{-270}$ ，可见概率很低。

至今 RC6 还未发现类似 DES 中的“弱密钥”。

最后，我们探讨一下常见密码破解方法对 RC6 算法的效果。

穷举攻击：穷举 b 字节的用户密钥或扩展密钥，密钥长度为 2^{8b} 位，当 $w = 32$, $r = 20$, $b = 32$ 时，扩展密钥长度为 2^{8*4*32} 位，即 2^{1024} 位，这种穷举法需要 $\min\{2^{8b}, 2^{1024}\}$ 次操作，可行性较低^[1]。

中间相遇攻击：RC6 多密钥重复加密时候，如果对其进行中间相遇攻击，则需要 2^{700} 次计算，这样要恢复扩展密钥最少需要 $\min\{2^{86}, 2^{704}\}$ 次操作^[6]。

差分攻击和线性攻击：对 RC6 的差分分析和线性分析只有在迭代轮数较少时有效，对 20 轮循环的 RC6，用线性分析法至少需要 2^{155} 个明文，用差分分析法至少需要 2^{238} 个明文^[1]。

时间攻击：因为 RC6 的加解密中只有移位和异或运算，运算过程比较固定，都与数据无关，这样可以有效地避免“时间攻击”。

3.4 AES2

略

3.5 DES3

略

4 公钥密码

4.1 RSA 加密

4.1.1 算法原理

RSA 是 1977 年由罗纳德·李维斯特 (Ron Rivest)、阿迪·萨莫尔 (Adi Shamir) 和伦纳德·阿德曼 (Leonard Adleman) 一起提出的, 故取他们的姓氏首字母得以命名。

1) 密钥生成算法:

选取两个保密的大素数 p 和 q , 满足 $p \neq q$, 计算 $n = p \times q$, $\varphi(n) = (p - 1)(q - 1)$, $\varphi(n)$ 为 n 的欧拉函数。

随机选取整数 e , 满足 $1 < e < \varphi(n)$ 且 $\gcd(e, \varphi(n)) = 1$, 即 e 与 $\varphi(n)$ 互素。

计算 d , 满足 $ed \equiv 1 \pmod{\varphi(n)}$, 则公钥为 (e, n) , 私钥为 d 。

2) 加密:

对明文进行比特串分组, 使每个分组十进制小于 n , 然后对每个分组 $m (0 \leq m < n)$, 计算 $c \equiv m^e \pmod{n}$, 则得到密文 c 。

3) 解密

对于密文 $c (0 \leq c < n)$, 计算 $m \equiv c^d \pmod{n}$, 得到对应明文 m 。

4.1.2 Miller-Rabin 素性检验

除了密钥生成以及加密解密算法之外, RSA 加密体制还有一个重要的问题, 那就使大素数的生成, 这就要求了还需要掌握素性检验的算法。在本实验中, 我使用了 Miller-Rabin 素性检验结合随机数的生成来得到所需要的大素数, 它是一个基于概率的算法, 是费马小定理的一个改进。简单来说, 要测试 n 是否为素数, 首先将 $n - 1$ 分解为 $2^s d$ 。在每次测试开始时, 先随机选一个介于 $[1, n - 1]$ 的整数 a , 之后如果对所有的 $r \in [0, s - 1]$, 若 $a^d \not\equiv 1 \pmod{n}$ 且 $a^{2^r d} \not\equiv -1 \pmod{n}$, 则 n 是合数。否则, n 有 $3/4$ 的概率为素数, 随着增加测试的次数, 是素数的概率会越来越高, 当达到某一次数时, 我们有理由相信它是一个素数。

4.1.3 代码实现

1) Miller-Rabin

```
def miller_rabin(self, n, k=10):  
    """  
    用 Miller-Rabin 算法进行素性检验
```

```

:param n: 被检验的数
:param k: 检验的次数, 默认为10 次
:return: 是否通过检验
"""
# 偶数直接不通过
if n % 2 == 0:
    return False
s, d = 0, n - 1
# 将 p-1 分解为(2**s)d
while d % 2 == 0:
    s += 1
    d //= 2
# 进行 k 次检验
for i in range(k):
    # 每次测试时, 随机选取一个[1, n-1]的整数 a
    a = randint(1, n - 1)
    x = pow(a, d, n) # x = a**d mod(n)
    # 如果 a**d(mod n)=1, 说明当次检验通过(不是合数), 进行下一轮检验
    if x == 1 or x == n - 1:
        continue
    else:
        flag = 0
        # 对所有的 r∈[0, s-1], 判断 a**((2**r)*d) (mod n) 是否等于-1,
        for r in range(s):
            # x**pow(2, r) == a**d**pow(2, r)
            x = pow(x, 2, n)
            if x == n - 1:
                flag = 1
                break
        # 若 a**d≠1(mod n) 且 a**pow(2, r)**≠
        if flag == 0:
            return False
return True

```

2) 素数生成

```

def get_prime(self, n):
    """
    得到一个n 位的素数(10 进制表示)
    :param n: 二进制位数
    :return: n 位素数(10 进制表示)
    """
    while True:

```

```

        # 最高位为 1, 保证是 n 位 (若随机生成可能为 0, 即 n-1 位)
        # 随机生成 n-2 位数
        # 最低位为 1, 保证是奇数
        num = '1' + ''.join([str(randint(0, 1)) for _ in range(n - 2)]) + '1'
        num = int(num, 2)
        if self.miller_rabin(num):
            return num

```

3) 密钥生成

```

def get_keys(self, nbits):
    """
    :param nbits: 密钥长度(512/1024/2048...)
    :return: 公钥(e, n), 私钥 d
    """
    nbits = int(nbits)
    while True:
        p = self.get_prime(nbits)
        q = self.get_prime(nbits)
        # 保证 p != q
        if p == q:
            continue

        n = p * q
        phi_n = (p - 1) * (q - 1)
        e = randint(500, 10000)
        # 保证 e 与 phi_n 互素
        if self.extended_gcd(e, phi_n)[0] == 1:
            # 计算私钥
            d = self.mod_inverse(e, phi_n)
            return e, n, d

```

4) 加密

```

def encrypt(self, m, e, n):
    e = int(e)
    n = int(n)
    c = pow(m, e, n)
    return hex(c)

```

5) 解密

```
def decrypt(self, c, d, n):
    d = int(d)
    n = int(n)
    m = pow(c, d, n)
    return m
```

4.1.4 正确性检验与性能分析

请选择模式：
[E]加密Encrypt [D]解密Decrypt [C]破解Crack

E

请输入明文：
china

请输入位数（k*512）：
512

公共模数n为：
88418158261687840981888980589704020754483934588836892821535093373616051725788829707437714176137453453418692404321615683739789142816510705251742495883658450132966
95279025461262879433241575536779021514170398402443414220818936354249018110577970605572336794849110328690421118855309167367854872052786249552456199

公钥e为：4487

私钥d为：
680230626965336365209451216838997725973876849121161029685623227826437732465841408847926968873640848968397888607024340838036494789615020047811235813637260601828835
10659727726158023563309717795673091885388095558591283478504536138846717740883967687943728467585486881867574738223565654217859994996306361624740663

密文为：
0x70012c1336ded21f5832841210a258db52bf39ac15066d94df20a5eb0c675677172ebdc5647f3d3d6b37ee11d5c7c489e5c425f7c812f90a66d80b2963d3b2df46576d521ab66d48a1057f326ea461af87d
87df35c6f290cd4ebdc3687b9bbd2fb8d0561fa19dbd329fa49efef45bce617d44f244a268a4d7aa5afcb54a1694a80e

请选择模式：
[E]加密Encrypt [D]解密Decrypt [C]破解Crack

D

请输入密文（16进制）：
70012c1336ded21f5832841210a258db52bf39ac15066d94df20a5eb0c675677172ebdc5647f3d3d6b37ee11d5c7c489e5c425f7c812f90a66d80b2963d3b2df46576d521ab66d48a1057f326ea461af87d
f35c6f290cd4ebdc3687b9bbd2fb8d0561fa19dbd329fa49efef45bce617d44f244a268a4d7aa5afcb54a1694a80e

请输入私钥d：
680230626965336365209451216838997725973876849121161029685623227826437732465841408847926968873640848968397888607024340838036494789615020047811235813637260601828835
0659727726158023563309717795673091885388095558591283478504536138846717740883967687943728467585486881867574738223565654217859994996306361624740663

请输入公共模数n：
884181582616878409818889805897040207544839345888368928215350933736160517257888297074377141761374534534186924043216156837397891428165107052517424958836584501329669
5279025461262879433241575536779021514170398402443414220818936354249018110577970605572336794849110328690421118855309167367854872052786249552456199

密文为：b'china'

4.1.5 安全性与可用性分析

1) 安全性

RSA 算法的安全性取决于对大整数的因子分解难题，对一极大整数做因数分解越困难，RSA 算法的安全性越可靠，假如有人找到一种快速因数分解的算法的话，那么 RSA 的安全性将会不复存在。然而目前来看，只要其密钥的长度足够长，用 RSA 加密的信息实际上是不能被破解的。

但是，RSA 的安全性很多时候也受到各参数选择的影响，具体来说：

- ① p 和 q 的长度相差不能太大，以避免椭圆曲线因子分解法。
- ② p 和 q 的差值不能太小，以防止被所有接近 \sqrt{n} 的奇整数试除而被有效分解。
- ③ $\gcd(p-1, q-1)$ 应该尽可能小
- ④ $p-1$ 和 $q-1$ 都应有大的素因子，以避免循环攻击。

-
- ⑤ e 和 d 不能选取太小，以避免低指数攻击。
 - ⑥ 多个用户不应使用相同模数 n ，以避免共模攻击。

2) 可用性

RSA 是第一个安全、实用的公钥加密算法，已经成为国际标准，是目前应用最广泛的公钥加密体制，其算法数学原理可靠，实现简单。虽然 RSA 与 DES 等分组密码相比加密速度要慢很多，但是仍可应用在密钥加密、数字签名等领域，而不适合直接对大量明文进行加密。

4.1.6 破解或攻击方式分析

如前所述，RSA 加密在计算上本身是难破解的，我们所说的攻击 RSA 加密，经常是针对参数选择的攻击，详见下面的几道例题。

4.2 生成 pem 文件

为了在最后的综合实践中让 Alice 和 Bob 能够顺利交换公钥，我为 RSA 算法加入了生成.pem 文件的功能

4.2.1 代码实现

1) 公钥生成

```
def derRSAPublicKey(self, n, e):
    """
    生成RSA 公钥的der 格式字节流并转换为pem 格式。
    @param n: 公钥n
    @param e: 公钥e
    @return: pem 文件的列表
    """
    data = self.derBitString(self.derTag(self.derInteger(n) +
self.derInteger(e)))

    data =
b'\x30\x0d\x06\x09\x2a\x86\x48\x86\xf7\x0d\x01\x01\x01\x05\x00' +
data
    # print(showbytes(derTag(data)))
    rsapk = base64.b64encode(self.derTag(data))
    ret = []
    ret.append(b'-----BEGIN PUBLIC KEY-----')
    for i in range(0, len(rsapk), 64):
        ret.append(rsapk[i:i + 64])
```

```
ret.append(b'-----END PUBLIC KEY-----')
return ret
```

2) 私钥生成

```
def derRSAPrivateKey(self, n, e, d, p, q):
    """
    生成RSA 私钥的der 格式字节流并转换为pem 格式。
    @param n: 公钥n
    @param e: 公钥e
    @param d: 私钥d
    @param p: 大素数p
    @param q: 大素数q
    @return: pem 文件的列表
    """
    para1 = d % (p - 1)
    para2 = d % (q - 1)
    para3 = gmpy2.invert(q, p)
    data = self.derTag(
        self.derInteger(0) + self.derInteger(n) + self.derInteger(e) +
        self.derInteger(d) + self.derInteger(p) +
        self.derInteger(q) + self.derInteger(para1) +
        self.derInteger(para2) + self.derInteger(para3))

    data = self.derTag(self.derInteger(
        0) +
        b'\x30\x0d\x06\x09\x2a\x86\x48\x86\xf7\x0d\x01\x01\x01\x05\x00' +
        self.derOctetString(data))
    # print(showbytes(data))
    rsapk = base64.b64encode(data)
    ret = []
    ret.append(b'-----BEGIN PRIVATE KEY-----')
    for i in range(0, len(rsapk), 64):
        ret.append(rsapk[i:i + 64])
    ret.append(b'-----END PRIVATE KEY-----')
    return ret
```

4.2.2 正确性验证

```
def show(self):
    self.get_keys(512)
    self.outputPublicKey('key')
    self.outputPrivateKey('key')
```



```

self.inputKey(r'key\private.txt')
print('hello'.encode('utf-8'))
x = self.encrypt(libnum.s2n('hello'), self.e, self.n)
print(x)
y = self.decrypt(int(x, 16), self.d, self.n)
print(libnum.n2s(y))

```

私钥加载成功，当前运行模式为：加密解密。

```

b'hello'
0x5d3f51bd4a51a7c2d297872690ba2c4196ac1685ac7aca2f
b'hello'

```

4.3 RSA

4.3.1 分析

打开题目看到了公钥 (n, e) 和密文 c 。

其中 e 很大，那么相应地， d 会很小，符合维纳攻击（Wiener's attack）的情况。以密码学家 Michael J. Wiener 命名的 Wiener 攻击是一种针对 RSA 的密码攻击。该攻击使用连分数法在 d 较小时暴露私钥 d 。

其内容为：当 $n = pq$ ， $q < p < 2q$ ， $d < \frac{1}{3}N^{\frac{1}{4}}$ 时，若有 (n, e) 和 $ed \equiv 1 \pmod{\varphi(n)}$ ，那么攻击者可以有效地恢复 d 。

其证明如下：

Fact 1 Let $\langle N, e \rangle$ be an RSA public key. Given the private key d , one can efficiently factor the modulus $N = pq$. Conversely, given the factorization of N , one can efficiently recover d .

Proof A factorization of N yields $\varphi(N)$. Since e is known, one can recover d . This proves the converse statement. We now show that given d one can factor N . Given d , compute $k = de - 1$. By definition of d and e we know that k is a multiple of $\varphi(N)$. Since $\varphi(N)$ is even, $k = 2^t r$ with r odd and $t \geq 1$. We have $g^k = 1$ for every $g \in \mathbb{Z}_N^*$, and therefore $g^{k/2}$ is a square root of unity modulo N . By the Chinese Remainder Theorem, 1 has four square roots modulo $N = pq$. Two of these square roots are ± 1 . The other two are $\pm x$ where x satisfies $x = 1 \pmod p$ and $x = -1 \pmod q$. Using either one of these last two square roots, the factorization of N is revealed by computing $\gcd(x - 1, N)$. A straightforward argument shows that if g is chosen at random from \mathbb{Z}_N^* then with probability at least $1/2$ (over the choice of g) one of the elements in the sequence $g^{k/2}, g^{k/4}, \dots, g^{k/2^t} \pmod N$ is a square root of unity that reveals the factorization of N . All elements in the sequence can be efficiently computed in time $O(n^3)$ where $n = \log_2 N$. \square

4.3.2 脚本

```

import string

from Crypto.Util.number import long_to_bytes

```

```

def bitlength(x):
    """
    Calculates the bitlength of x
    """
    assert x >= 0
    n = 0
    while x > 0:
        n = n + 1
        x = x >> 1
    return n

def isqrt(n):
    """
    Calculates the integer square root
    for arbitrary large nonnegative integers
    """
    if n < 0:
        raise ValueError('square root not defined for negative numbers')

    if n == 0:
        return 0
    a, b = divmod(bitlength(n), 2)
    x = 2 ** (a + b)
    while True:
        y = (x + n // x) // 2
        if y >= x:
            return x
        x = y

def rational_to_contfrac(x, y):
    """
    Converts a rational x/y fraction into
    a list of partial quotients [a0, ..., an]
    """
    a = x // y
    pquotients = [a]
    while a * y != x:
        x, y = y, x - a * y
        a = x // y
        pquotients.append(a)
    return pquotients

```

```

# TODO: efficient method that calculates convergents on-the-go, without
doing partial quotients first
def convergents_from_contfrac(frac):
    '''
    computes the list of convergents
    using the list of partial quotients
    '''
    convs = [];
    for i in range(len(frac)):
        convs.append(contfrac_to_rational(frac[0:i]))
    return convs

def contfrac_to_rational(frac):
    '''Converts a finite continued fraction [a0, ..., an]
    to an x/y rational.
    '''
    if len(frac) == 0:
        return (0, 1)
    num = frac[-1]
    denom = 1
    for _ in range(-2, -len(frac) - 1, -1):
        num, denom = frac[_] * num + denom, num
    return (num, denom)

def is_perfect_square(n):
    '''
    If n is a perfect square it returns sqrt(n),
    otherwise returns -1
    '''
    h = n & 0xF; # last hexadecimal "digit"

    if h > 9:
        return -1 # return immediately in 6 cases out of 16.

    # Take advantage of Boolean short-circuit evaluation
    if (h != 2 and h != 3 and h != 5 and h != 6 and h != 7 and h != 8):
        # take square root if you must
        t = isqrt(n)

```

```

        if t * t == n:
            return t
        else:
            return -1

    return -1

def hack_RSA(e, n):
    '''
    Finds d knowing (e,n)
    applying the Wiener continued fraction attack
    '''
    frac = rational_to_contfrac(e, n)
    convergents = convergents_from_contfrac(frac)

    for (k, d) in convergents:

        # check if d is actually the key
        if k != 0 and (e * d - 1) % k == 0:
            phi = (e * d - 1) // k
            s = n - phi + 1
            # check if the equation x^2 - s*x + n = 0
            # has integer roots
            discr = s * s - 4 * n
            if (discr >= 0):
                t = is_perfect_square(discr)
                if t != -1 and (s + t) % 2 == 0:
                    print("Hacked!")
                    return d

chars = string.ascii_letters + string.digits + '_@' + '{}'

e =
1843761357024744573770463077615077573550924452563330353292181312299754995474
1828855898842356900537746647414676272022397989161180996467240795661928117273
8376666154151535719592588478295281315194234862617575694540119403188495897301
5203152832357699780178820645754853180266383441838106155122754493741273477658
1781
n =
1472825736119845803849657279768393513560094656160534754280398517945538808331
7787721132331813084326784730326473008842455265712931429511761422263032658194

```

```

3132950689147833674506592824134135054877394753008169629583742916853056999371
9853071387752980809868017429428332127279492775176913113150987225362821198886
05701

c =
1408966982676704801757398175398986386570990871970968367342430168242041134529
8761761094498674291979350602489263885133901501570616441299451459856498937403
7762836439262224649359411190187875207060663509777017529293145434535056275850
5553310991306332328440547670571750765987412339885331810358712384440083663069
56934

d = hack_RSA(e, n)
m = pow(c, d, n)
m = long_to_bytes(m).decode()
for ch in m:
    if ch in chars:
        print(ch, end='')
    else:
        print('.', end='')

```

4.3.3 运行结果

```

Hacked!
unctf{wi3n3r_Att@ck}
进程已结束,退出代码0

```

```
unctf{wi3n3r_Att@ck}
```

4.4 mediumRSA

4.4.1 分析

RSA 常见的入门题目，不过 n 的值并没有直接给出，我们可以使用 `openssl` 读取它，也可以使用 `python` 中的 `RSA` 库。

分解 n 可以使用 `yafu` 或 `factordb`，这里使用 `yafu`。

```

***factors found***

P39 = 319576316814478949870590164193048041239
P39 = 275127860351348928173285174381581152299

ans = 1

```

4.4.2 脚本

```
import string
import gmpy2
from Crypto.Util.number import long_to_bytes
from Crypto.PublicKey import RSA

with open('pubkey.pem') as f:
    key = RSA.import_key(f.read())
    # print(f'e={key.e}')
    # print(f'n={key.n}')

chars = string.ascii_letters + string.digits + '_' + '{}'
n = key.n #
8792434826413240687527614051449993714505089366560259299241817164704249165846
1
e = key.e # 65537

p = 275127860351348928173285174381581152299
q = 319576316814478949870590164193048041239

phi_n = (p - 1) * (q - 1)
d = gmpy2.invert(e, phi_n)
c = int(open('flag.enc', 'rb').read().hex(), 16)
m = pow(c, d, n)
m = long_to_bytes(m).decode('latin')
for ch in m:
    if ch in chars:
        print(ch, end='')
    else:
        print('.', end='')
```

4.4.3 运行结果

.....PCTF{256b_i5_m3dium}.

进程已结束,退出代码0

PCTF{256b_i5_m3dium}

4.5 hardRSA

4.2.1. 分析

低指数攻击

公钥 e 很小，一般为 2，没有逆元， N 可被分解

4.2.2. 脚本

```
import gmpy2
import string
from Crypto.PublicKey import RSA

chars = string.ascii_letters + string.digits + '_' + '{}'

with open('pubkey.pem') as f:
    key = RSA.import_key(f.read())
    # print(f'e={key.e}')
    # print(f'n={key.n}')

n = key.n #
8792434826413240687527614051449993714505089366560259299241817164704249165846
1
e = key.e # 2

p = 319576316814478949870590164193048041239
q = 275127860351348928173285174381581152299

assert (p % 4 == 3 and q % 4 == 3)
cipher = int(open('flag.enc', 'rb').read().hex(), 16)
p_inv = gmpy2.invert(p, q)
q_inv = gmpy2.invert(q, p)

m_p = pow(cipher, (p + 1) // 4, p)
m_q = pow(cipher, (q + 1) // 4, q)

a = (p_inv * p * m_q + q_inv * q * m_p) % n
b = n - a
c = ((p_inv * p * m_q - q_inv * q * m_p) % n + n) % n
d = n - c

for s in (a, b, c, d):
    s = format(s, 'x')
```

```

if len(s) % 2 != 0:
    s = '0' + s
hx = bytes.fromhex(s).decode('latin')
for ch in hx:
    if ch in chars:
        print(ch, end='')
    else:
        print('.', end='')

```

4.2.3. 运行结果

D.....P....e.cb.....P...V...}.Y....S..Zv...F.....C.....gq...2.....PCTF{sp3ci4l_rsa}.....8....X.....i..{Y.....W....
 进程已结束, 退出代码0

PCTF{sp3ci4l_rsa}

4.6 veryhardRSA

4.6.1 分析

共模攻击

明文 m 、模数 n 相同，公钥指数 e 、密文 c 不同， e_1 和 e_2 互质，即
 $\gcd(e_1, e_2) == 1$

$$\text{encrypt1} = \text{pow}(\text{data_num}, e_1, N)$$

$$\text{encrypt2} = \text{pow}(\text{data_num}, e_2, N)$$

对同一明文的多次加密使用相同的模数和不同的公钥指数。

首先可以从题目中得到 n 、 e_1 、 e_2

4.6.2 脚本

```

import gmpy2
import string
from Crypto.Util.number import long_to_bytes, bytes_to_long

chars = string.ascii_letters + string.digits + '_' + '{}'

n =
0x00b0bee5e3e9e5a7e8d00b493355c618fc8c7d7d03b82e409951c182f398dee3104580e7ba
70d383ae5311475656e8a964d380cb157f48c951adfa65db0b122ca40e42fa709189b719a4f0
d746e2f6069baf11cebd650f14b93c977352fd13b1eea6d6e1da775502abff89d3a8b3615fd0
db49b88a976bc20568489284e181f6f11e270891c8ef80017bad238e363039a458470f174910
1bc29949d3a4f4038d463938851579c7525a69984f15b5667f34209b70eb261136947fa123e5

```



```

49dfff00601883afd936fe411e006e4e93d1a00b0fea541bbfc8c5186cb6220503a94b241311
0d640c77ea54ba3220fc8f4cc6ce77151e29b3e06578c478bd1bebe04589ef9a197f6f806db8
b3ecd826cad24f5324ccdec6e8fead2c2150068602c8dcdc59402ccac9424b790048ccdd9327
068095efa010b7f196c74ba8c37b128f9e1411751633f78b7b9e56f71f77a1b4daad3fc54b5e
7ef935d9a72fb176759765522b4bbc02e314d5c06b64d5054b7b096c601236e6ccf45b5e611c
805d335dbab0c35d226cc208d8ce4736ba39a0354426fae006c7fe52d5267dcfb9c3884f51fd
dfdf4a9794bcfe0e1557113749e6c8ef421dba263aff68739ce00ed80fd0022ef92d3488f76d
eb62bdef7bea6026f22a1d25aa2a92d124414a8021fe0c174b9803e6bb5fad75e186a946a172
80770f1243f4387446ccceb2222a965cc30b3929

e1 = 17
e2 = 65537

with open('flag.enc1', 'rb') as enc1:
    c1 = bytes_to_long(enc1.read())
with open('flag.enc2', 'rb') as enc2:
    c2 = bytes_to_long(enc2.read())

gcd, r, s = gmpy2.gcdext(e1, e2)
m = gmpy2.powmod(c1, r, n) * gmpy2.powmod(c2, s, n) % n
m = long_to_bytes(m).decode('latin')
for ch in m:
    if ch in chars:
        print(ch, end='')
    else:
        print('.', end='')

```

4.6.3 运行结果

```

..7.e..e.B.....Y.7.....6..Tnh0..4.....j...2.....3..i...h...XL3.....C..rD...b....Yp.....SQjE.....J..H.S.....Y.....g...A.v.R.C
.X.R.C.....Y...s.....L.Gw.X0.e...1.....g.....{0.....WWQ.Au.....a...l.....b.....z6g.....n.....u.3...e..Y.....X.....Z
.q...o...v.Z.....r.N.....V.....6...Z...0...v...V...Sg..h.....E.u...BcY.....e...e0.Qg.....9.X.....1.....
.PCTF{M4st3r_oF_Number_Th3ory}.

```

进程已结束, 退出代码0

PCTF{M4st3r_oF_Number_Th3ory}

4.7 cipher

4.7.1 分析

一个简单的 RSA 共享素数攻击，在生成 p 和 q 的时候，难免会有 2 个 n 共享 1 个素数

所以我们用 \gcd 遍历 n ，相应的脚本如下，即可分解出 p 和 q

4.7.2 脚本

```
import gmpy2
import string
from Crypto.Util.number import long_to_bytes

chars = string.ascii_letters + string.digits + '_' + '{}'
f = open('cipher', 'rb')
cipher = f.readlines()[6:-1]
ns = []
cs = []
e = 65537
cipher = [line.decode() for line in cipher]
for line in cipher:
    if 'n' in line:
        ns.append(int(line[4:].replace('\n', '')))
    elif 'c' in line:
        cs.append(int(line[4:].replace('\n', '')))
for i in range(len(ns)):
    for j in range(i + 1, len(ns)):
        if gmpy2.gcd(ns[i], ns[j]) != 1:
            n = ns[i]
            c = cs[i]
            p = gmpy2.gcd(ns[i], ns[j])
            q = n // p
            phi_n = (p - 1) * (q - 1)
            d = gmpy2.invert(e, phi_n)
            m = long_to_bytes(pow(c, d, n)).decode('latin')
            for ch in m:
                if ch in chars:
                    print(ch, end='')
                else:
                    print('.', end='')
```

4.7.3 运行结果

flag{abdcbe5fd94e23b3de429223ab9c2fdf}

进程已结束,退出代码0

flag{abdcbe5fd94e23b3de429223ab9c2fdf}

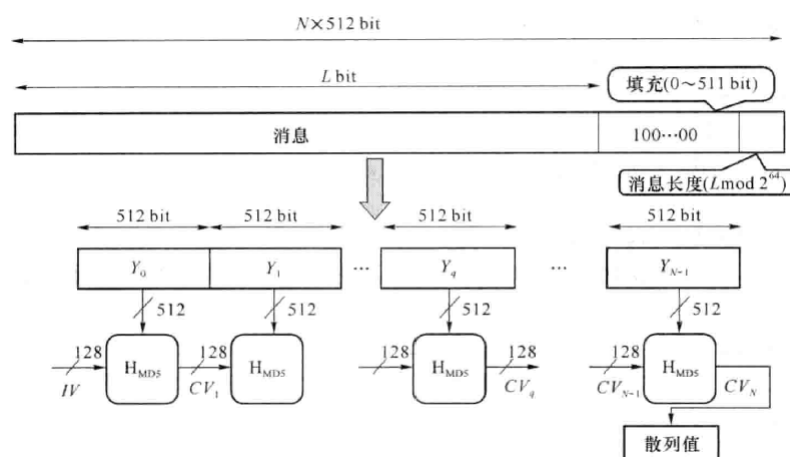
5 Hash 函数

5.1 MD5

5.1.1 算法原理

MD5 消息摘要算法（MD5），一种被广泛使用的密码散列函数，可以产生出一个 128 位（16 字节）的散列值，用于确保信息传输完整一致。

MD5 算法输入不定长度信息，输出固定长度 128-bits 的算法。经过程序流程，生成四个 32 位数据，最后联合起来成为一个 128-bits 散列。基本方式有求余、取余、调整长度、与链接变量进行循环运算，得出结果。基本流程如下图所示：



5.1.2 代码实现

1) 压缩函数

```
# 压缩函数(对每个 512bit 分组进行处理)
def compress_func(self, a, b, c, d, m_i):
    """
    压缩函数函数, 对每 512bit 得分组进行处理, 包括 4 轮, 每轮 16 步
    :param a, b, c, d: 输入链接变量(即前一个分组的输出链接变量)
    :param m_i: 512bit 的消息分组
    :return: A, B, C, D 输出链接变量
    """
    # 对每一分组的初始链接变量进行备份
    A, B, C, D = a, b, c, d
    # 将 512bit 分为 16 组, 每组 32bit
    m_list_32 = re.findall(r'\.{32}', m_i)
    # 每个分组经过 4 轮函数
```

```

for round_index in range(4):
    # 每轮有 16 步
    for step_index in range(16):
        # 对每一步的链接变量进行备份
        AA, BB, CC, DD = A, B, C, D
        # 每一轮选择不同的非线性函数
        match round_index:
            case 0:
                func_out = self.F(B, C, D)
            case 1:
                func_out = self.G(B, C, D)
            case 2:
                func_out = self.H(B, C, D)
            case 3:
                func_out = self.I(B, C, D)
        A, C, D = D, B, C
        # B 模加非线性函数的输出
        temp = self.mod_add(AA, func_out)
        # 模加消息分组(注意为大端序)
        temp = self.mod_add(temp,
int(self.bin2little(m_list_32[self._M_index[round_index][step_index]]
), 2))

        # print(type(B))
        # 模加伪随机常数
        temp = self.mod_add(temp, self.T(16 * round_index +
step_index + 1))
        # 循环左移 s 位
        temp = self.rol(temp, self._shift[round_index][step_index])
        # 模加 BB
        B = self.mod_add(temp, BB)
        #     print(str(16 * round_index + step_index + 1).zfill(2),
end=" ")
        #     print(hex(A).replace("0x", "").replace("L", "").zfill(8),
end=" ")
        #     print(hex(B).replace("0x", "").replace("L", "").zfill(8),
end=" ")
        #     print(hex(C).replace("0x", "").replace("L", "").zfill(8),
end=" ")
        #     print(hex(D).replace("0x", "").replace("L", "").zfill(8))
        # print("*" * 38)
        # 与该分组的初始链接变量异或
        A = self.mod_add(A, a)
        B = self.mod_add(B, b)

```

```
C = self.mod_add(C, c)
D = self.mod_add(D, d)
```

2) 运算流程

```
def hash(self, m):
    # 转为 2 进制
    m = self.str2bin(m)
    # 消息填充
    m = self.message_padding(m)
    # 对消息分组, 每组 512 位
    m_list_512 = re.findall(r'.{512}', m)
    # 初始链接变量
    A, B, C, D = self.IV_A, self.IV_B, self.IV_C, self.IV_D
    # 对每 512bit 进行分组处理, 前一组的 4 个输出连接变量作为下一组的 4 个输入链接变量
    for m_i in m_list_512:
        A, B, C, D = self.compress_func(A, B, C, D, m_i)
    # 把最后一次的分组 4 个输出连接变量再做一次大端小端转换
    A = self.hex2little(hex(A)[2:]).zfill(8)
    B = self.hex2little(hex(B)[2:]).zfill(8)
    C = self.hex2little(hex(C)[2:]).zfill(8)
    D = self.hex2little(hex(D)[2:]).zfill(8)
    # 拼接到一起的得到最终的 md5 值
    return f'{A}{B}{C}{D}'
```

5.1.3 正确性检验与性能分析

请输入要进行md5的内容:

china

第一轮:

```

01 10325476 d9d418ab efcdab89 98badcfe
02 98badcfe 6f64c4d9 d9d418ab efcdab89
03 efcdab89 6229d229 6f64c4d9 d9d418ab
04 d9d418ab 22d53239 6229d229 6f64c4d9
05 6f64c4d9 5bd4d3d8 22d53239 6229d229
06 6229d229 f5a8a176 5bd4d3d8 22d53239
07 22d53239 4cd35dd5 f5a8a176 5bd4d3d8
08 5bd4d3d8 f27145e7 4cd35dd5 f5a8a176
09 f5a8a176 8a1a886c f27145e7 4cd35dd5
10 4cd35dd5 790c2ce7 8a1a886c f27145e7
11 f27145e7 7ee1db7f 790c2ce7 8a1a886c
12 8a1a886c 01fed588 7ee1db7f 790c2ce7
13 790c2ce7 47c8543f 01fed588 7ee1db7f
14 7ee1db7f 2fa47f47 47c8543f 01fed588
15 01fed588 172cd9b2 2fa47f47 47c8543f
16 47c8543f c551bf80 172cd9b2 2fa47f47
*****

```

第二轮:

```

17 2fa47f47 2348f5ca c551bf80 172cd9b2
18 172cd9b2 a19d0930 2348f5ca c551bf80
19 c551bf80 c0f08159 a19d0930 2348f5ca
20 2348f5ca 514a65d0 c0f08159 a19d0930
21 a19d0930 ad5b55d7 514a65d0 c0f08159
22 c0f08159 2461fe41 ad5b55d7 514a65d0
23 514a65d0 1b6a4fb9 2461fe41 ad5b55d7
24 ad5b55d7 2dfe784e 1b6a4fb9 2461fe41
25 2461fe41 02f22eef 2dfe784e 1b6a4fb9
26 1b6a4fb9 217c8b0c 02f22eef 2dfe784e
27 2dfe784e fc77d7fa 217c8b0c 02f22eef
28 02f22eef 4ed1256b fc77d7fa 217c8b0c
29 217c8b0c 73fee33c 4ed1256b fc77d7fa
30 fc77d7fa 3a235e5e 73fee33c 4ed1256b
31 4ed1256b e26fc623 3a235e5e 73fee33c
32 73fee33c e7e42cd8 e26fc623 3a235e5e
*****

```

第三轮:

```
33 3a235e5e 22013f13 e7e42cd8 e26fc623
34 e26fc623 23577e5c 22013f13 e7e42cd8
35 e7e42cd8 b833b51b 23577e5c 22013f13
36 22013f13 68834cc7 b833b51b 23577e5c
37 23577e5c 13066052 68834cc7 b833b51b
38 b833b51b 7a42f9e9 13066052 68834cc7
39 68834cc7 503aaa9f 7a42f9e9 13066052
40 13066052 7deb0b3d 503aaa9f 7a42f9e9
41 7a42f9e9 b13e8176 7deb0b3d 503aaa9f
42 503aaa9f 969754f7 b13e8176 7deb0b3d
43 7deb0b3d 5077d463 969754f7 b13e8176
44 b13e8176 62f4fcf8 5077d463 969754f7
45 969754f7 5571deaa 62f4fcf8 5077d463
46 5077d463 849a4dd5 5571deaa 62f4fcf8
47 62f4fcf8 457c710e 849a4dd5 5571deaa
48 5571deaa 2cda8da8 457c710e 849a4dd5
*****
```

第四轮:

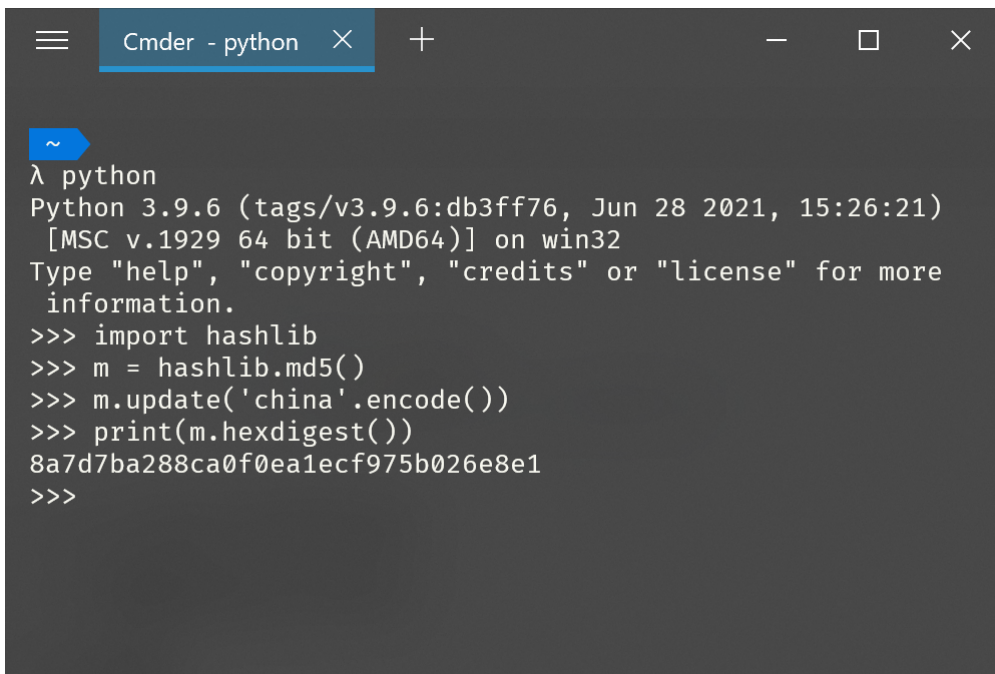
```
49 849a4dd5 cee88b24 2cda8da8 457c710e
50 457c710e a827b18b cee88b24 2cda8da8
51 2cda8da8 af13c4fb a827b18b cee88b24
52 cee88b24 193bd8c9 af13c4fb a827b18b
53 a827b18b 643b9412 193bd8c9 af13c4fb
54 af13c4fb 47d386a5 643b9412 193bd8c9
55 193bd8c9 3e62201c 47d386a5 643b9412
56 643b9412 c8d4fef0 3e62201c 47d386a5
57 47d386a5 699190d6 c8d4fef0 3e62201c
58 3e62201c 8fcbc273 699190d6 c8d4fef0
59 c8d4fef0 6b389e61 8fcbc273 699190d6
60 699190d6 20a990aa 6b389e61 8fcbc273
61 8fcbc273 3b365a89 20a990aa 6b389e61
62 6b389e61 d1b5d23a 3b365a89 20a990aa
63 20a990aa dd3f0fa3 d1b5d23a 3b365a89
64 3b365a89 1e421eff dd3f0fa3 d1b5d23a
*****
```

结果:

md5后的散列值为:

8a7d7ba200a8fce0a1ecf975b026e8e1

与内建库结果一致:



```
~
λ python
Python 3.9.6 (tags/v3.9.6:db3ff76, Jun 28 2021, 15:26:21)
[MSC v.1929 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more
information.
>>> import hashlib
>>> m = hashlib.md5()
>>> m.update('china'.encode())
>>> print(m.hexdigest())
8a7d7ba288ca0f0ea1ecf975b026e8e1
>>>
```

5.1.4 安全性与可用性分析

1) 安全性

本质上，对于任何一个哈希函数来说，碰撞是无可避免的，从一个规模较大的集合映射到一个规模较小的集合，必然会存在相同映射的情况。对于 MD5 而言，由于其抗密码分析能力较弱以及计算机运算能力的不断提升，在有限时间实现 MD5 的碰撞已经被证明为可能，这使得 MD5 算法在目前的安全环境下有一点落伍。但是从实践角度，不同信息具有相同 MD5 的可能性还是非常低的，通过碰撞的方法也很难碰撞出复杂信息的 MD5 数值。

2) 可用性

MD5 不依赖任何密码系统和假设条件，算法简洁、计算速度快，特别适合 32 位计算机软件实现。当前，MD5 算法因其普遍、稳定、快速的特点，广泛应用于普通数据的错误检查、数字签名、密码存储等领域，但是，因为其已经被发现可以在有限时间内找到其碰撞，安全性保障大大下降，在需要对重要消息进行摘要的情况下，应选择安全系数更高的其他哈希函数。

5.1.5 破解或攻击方式分析

5.1.5.1 生日攻击

生日攻击是一种密码学攻击手段，所利用的是概率论中生日问题的数学原理。这种攻击手段可用于滥用两个或多个集团之间的通信。此攻击依赖于在随机攻击中的高碰撞概率和固定置换次数（鸽巢原理）。使用生日攻击，攻击者可在 $\sqrt{2^n} = 2^{\frac{n}{2}}$ 中找到散列函数碰撞， 2^n 为原像抗性安全性。

5.1.5.2 学术界成果

- 1) T. Berson（1992）已经证明，对单轮的 MD5 算法，利用差分密码分析，可以在合理的时间内找出散列值相同的两条消息。这一结果对 MD5 四轮运算的每一轮都成立。但是，目前尚不能将这种攻击推广到具有四轮运算的 MD5 上。
- 2) B. Boer 和 A. Bosselaers（1993）说明了如何找到消息分组和 MD5 两个不同的初始值,使它们产生相同的输出. 也就是说, 对于一个 512 位的分组, MD5 压缩函数对缓冲区 ABCD 的不同值产生相同的输出,这种情况称为伪碰撞（pseudo-collision），但是目前尚不能用该方法成功攻击 MD5 算法。
- 3) H. Dobbertin（1996）找到了 MD5 无初始值的碰撞(pseudo-collision)。给定一个 512 位的分组，可以找到另一个 512 位的分组，对于选择的初始值 IV0，它们的 MD5 运算结果相同。到目前为止，尚不能用这种方法对使用 MD5 初始值 IV 的整个消息进行攻击。
- 4) 我国山东大学王小云教授（2004）提出的攻击对 MD5 最具威胁。对于 MD5 的初始值 IV，王小云找到了许多 512 位的分组对，它们的 MD5 值相同，即利用差分分析，只需 1 小时就可找出 MD5 的碰撞。
- 5) 国际密码学家 Lenstra 利用王小云等提供的 MD5 碰撞，伪造了符合 X.509 标准的数字证书。

5.2 md51

5.2.1 分析

明文有三位未知，数量很小，我们可以直接爆破，可以用到一点技巧
限定爆破的字符集 chars 为数字和大写字母，比遍历课件 ASCII 字符要快得多。

验证爆破的 MD5 值是否相同，只需验证开头和结尾即可，因为 hash 函数的抗弱碰撞性，很难出现 hash 值相同但原像不同的情况。

这样做的根本目的在于自己写的 MD5 没有自动的 MD5 快，内建库中的 MD5 用来多线程等各种手段，我目前没有这么高的水平，所以通过限制密文空间也可以在短时间内获取爆破结果。

5.2.2 脚本

```
from main import *
import hashlib
import string

chars = string.digits + string.ascii_uppercase
plaintext = "TASC?03RJM?WDJKX?ZM"
ciphertext = "e9032???da???08???911513?0???a2"

machine = MD5()
for i in chars:
    for j in chars:
        for k in chars:
            temp_plaintext = f'TASC{i}03RJM{j}WDJKX{k}ZM'
            m = hashlib.md5()
            m.update(temp_plaintext.encode())
            temp_ciphertext = m.hexdigest()
            # temp_ciphertext = machine.hash(temp_plaintext)
            if temp_ciphertext.startswith('e9032') and
temp_ciphertext.endswith('a2'):
                print(temp_plaintext)
                print(temp_ciphertext)
                break
```

5.2.3 运行结果

```
plaintext = TASCJ03RJMVKWDJKXLZM
ciphertext = e9032994dabac08080091151380478a2
```

进程已结束, 退出代码0

```
e9032994dabac08080091151380478a2
```

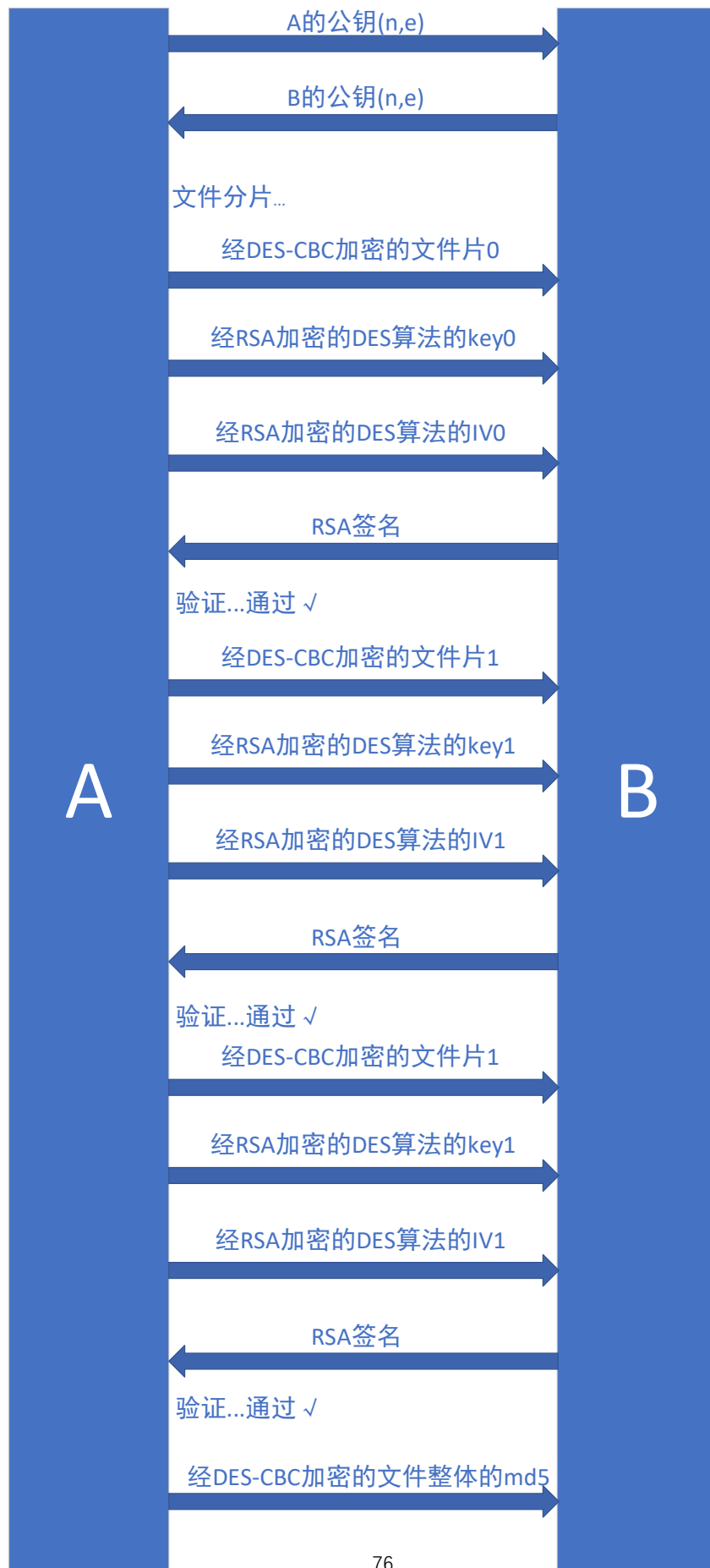
6 综合实践

6.1 设计要求

Alice 想通过公共信道给 Bob 传输一份秘密文件（文件非常大），又知道，很多人和机构想得到这份文件。如果你是 Alice，你应该怎样做？请设计一个方案，并编程实现。

6.2 设计原理

首先题设文件非常大，那么我们要考虑分片，并且考虑到要在公共信道传输，需要对分片的文件进行加密，而 RSA 加密速度略慢，在这里只需使用它进行签名即可，分片文件的加密工作交由 DES 完成。下面用一张图简单地说明主要算法原理



6.3 代码实现

1) 双方密钥生成

```
random_generator = Random.new().read
rsa = RSA.generate(1024, random_generator)
# 生成 Alice 的公钥和私钥
private_pem = rsa.exportKey()
# print(private_pem)
public_pem = rsa.publickey().exportKey()
with open(r'key\alice_private.pem', 'wb') as f:
    f.write(private_pem)
with open(r'key\alice_public.pem', 'wb') as f:
    f.write(public_pem)

# 生成 Bob 的公钥和私钥
private_pem = rsa.exportKey()
public_pem = rsa.publickey().exportKey()
with open(r'key\bob_private.pem', 'wb') as f:
    f.write(private_pem)
with open(r'key\bob_public.pem', 'wb') as f:
    f.write(public_pem)
```

2) DES-CBC 加密

```
def transfer_encrypt(message):
    """
    对分组传输的每一个分组进行 des_cbc 加密, 且每次使用不同的 key 和 iv
    @param message: 被加密的分组
    @return: key, iv, cipher
    """
    des_key, iv = get_des_param()
    des_cipher = DES().cbc_encrypt(message, des_key, iv, 'b')
    with open(r'key\bob_public.pem') as f:
        key = f.read()
        pubkey = RSA.importKey(key) # 将 bob_public.pem 中的公钥导出
        cipher = Cipher_pkcs1_v1_5.new(pubkey) # 按照 pkcs1_v1_5 的标准对
        公钥进行处理, 如填充字段等
        # 对 message 使用 bob 的公钥进行加密, 加密结果再用 base64 进行处理生成最终
        密文
        # 用 base64 进行处理的原因是为了防止原来的二进制数据某些字符被当做控制指
        令, 或者一些不可见的二进制字符(ascii 值在 128~255 之间的)在路由传输过程中被滤过
        # 使用 base64 可将所有不可见二进制字符显示
        key_encrypted =
```

```

base64.b64encode(cipher.encrypt(des_key.encode(encoding="utf-8")))
    iv_encrypted =
base64.b64encode(cipher.encrypt(iv.encode(encoding="utf-8")))
    return key_encrypted, iv_encrypted, des_cipher

```

3) 循环发送分片文件

```

filepath = r'Alice\file'
# 确定文件路径
print("Send: " + filepath)

if os.path.isfile(filepath):
    file_size = str(os.path.getsize(filepath))
    # 总共要传输的次数
    count = (int(file_size) // BUFSIZE) + 1

    # 获得原文件的 md5
    file_digest = get_file_md5(filepath)
    send_size = 0
    f1 = open(filepath, 'rb')
    print("Start transferring...")
    i = 0
    shutil.rmtree('Bob')
    os.mkdir('Bob')
    # 循环发送文件
    while count:
        print('=' * 45 + f'round{i}' + '=' * 45 + '\n')
        print('*' * 45 + "Alice->Bob:" + '*' * 45)
        filedata = base64.b64encode(f1.read(BUFSIZE)).decode()
        key_encrypted, iv_encrypted, allfiles_encrypted =
transfer_encrypt(filedata)
        # 发送 des-cbc 加密后的内容(16 进制)
        print(f'encrypted_file_part{i} is:')
        print(allfiles_encrypted)
        with open(rf'Bob\file.ency{i}', 'wb') as f:
            f.write(allfiles_encrypted.encode())
        # 发送 RSA 加密后的 key(10 进制)
        print('encrypted_des_key is:')
        print(key_encrypted.decode())
        # 发送 RSA 加密后的 iv
        print('encrypted_des_iv is:')
        print(iv_encrypted.decode())
        # time.sleep(0.2)

```

```

# 计算已发送的大小
send_size += BUFSIZE
if send_size > int(file_size):
    send_size = file_size
count -= 1
i += 1
print('*' * 100 + '\n')
# print("Bob:")
print('*' * 45 + "Bob->Alice" + '*' * 45)
"""对 Bob 身份进行验证, Bob 用自己的私钥进行签名, Alice 收到签名后, 用 Bob
公钥进行解密验签"""
# 签名
with open(r'key\bob_private.pem') as f:
    key = f.read()
    signkey = RSA.importKey(key)
    signer = Signature_pkcs1_v1_5.new(signkey) # 按照
pkcs1_v1_5 的标准对私钥进行处理, 用于签名计算, 将处理后的私钥存入 signer 变量
    digest = SHA.new()
    digest.update(allfiles_encrypted.encode("utf8")) # 对
message 用 SHA 进行散列计算, 得到摘要, 防止篡改
    sign = signer.sign(digest) # 对摘要用私钥进行处理
    signature = base64.b64encode(sign)
    print('signature is:')
    print(signature)
print('*' * 100 + '\n')
print('*' * 45 + "Alice" + '*' * 45)
# 验签
with open(r'key\bob_public.pem') as f:
    key = f.read()
    unsignkey = RSA.importKey(key)
    unsigner = Signature_pkcs1_v1_5.new(unsignkey) # 按照
pkcs1_v1_5 的标准对公钥进行处理, 用于签名解密, 将处理后的公钥存入 unsigner 变量
    digest = SHA.new()
    digest.update(allfiles_encrypted.encode("utf8"))
    is_verify = unsigner.verify(digest,
                                base64.b64decode(signature)) # 将签
名数据用 unsigner 进行解密, 然后与原文 digest 的摘要进行比较是否一致
    print('verify result is:', is_verify)
print('*' * 100 + '\n')
print("Get the file Successfully.")
f.close()
print('*' * 45 + "Alice->Bob" + '*' * 45)
# 发送 md5

```

```

print("file's md5:")
print(file_digest.encode())
print('*' * 100 + '\n')
else:
    print("[Error]: Can't find the file")

```

其中，BUFSIZE 是 Alice 处路由器的缓冲区容量，当欲发送文件大小大于这个容量时需要进行分片。

6.4 功能检验

我们先在 Alice 处新建 file 文件，为了检测分片效果，其最好大于 BUFSIZE。

Send: Alice/file

Start transferring...

第一轮发送

Alice->Bob

加密文件片

加密DES密钥

加密DES初始向量

Bob->Alice

Bob->Alice

Alice

verify result is: True

第二轮发送

Alice->Bob

加密文件片

加密DES密钥

加密DES初始向量

Bob->Alice

Bob->Alice

Alice

verify result is: True

可见流程正确。

接下来看一下 Bob 处的情况，查看项目结构树（其余项目已手动删减），有

```

D:.
├── file_transfer_system
│   ├── file_transfer_system.py
│   ├── file
│   └── Bob
│       ├── file.encyr0
│       └── file.encyr1

```

80

```
|      file.ency2
|
└─key
      alice_private.pem
      alice_public.pem
      bob_private.pem
      bob_public.pem
```

发现 Bob 处接收到了发送的三个加密后的文件片，那么证明我们作为 Alice 的实践完成了，而 Bob 只需将他们拼接后使用 md5 验证完整性，若通过则把他们分别 DES-CBC 解密再拼接即可。