西安电子科技大学

计算机与网络安全 课程实验报告

实验名称 加密算法实现

网络与信息安全 学院 1618019 班

姓名 曹寅峰 学号 16020610025

同作者 无

实验日期 2019 年 6 月 11 日

成 绩

指导教师评语：
指导教师： 年 月 日

实验报告内容

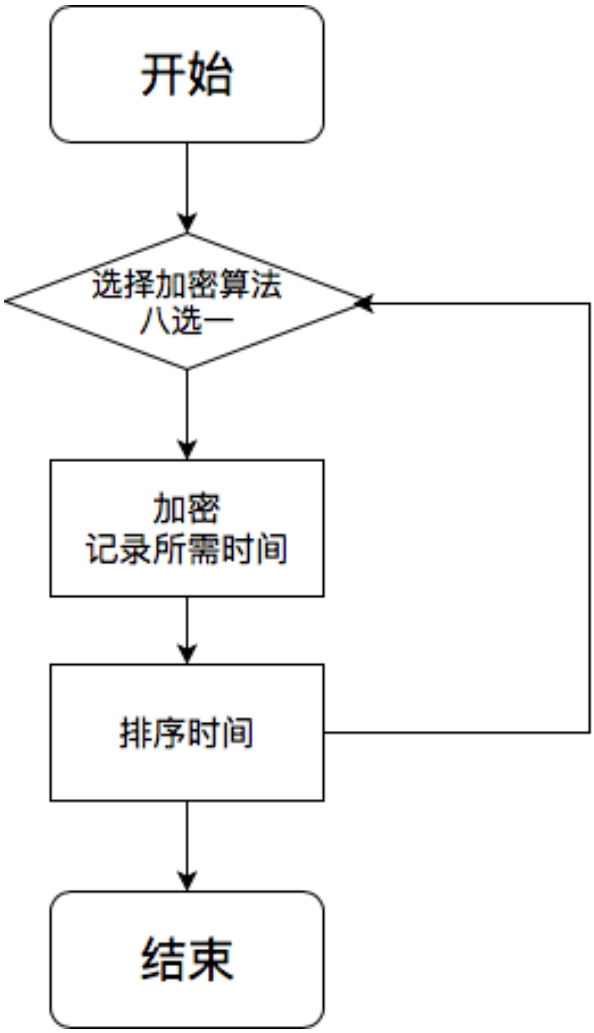
一、 实验目的

实现 RSA，CBC，PBE，MD5，DES，凯撒，RC4 和 A5 加密和解密，并且对比各个加密方案的时间消耗（**额外增加了 RC4 和 A5 加密算法**）

二、 实验所用仪器（或实验环境）

Python 2.7

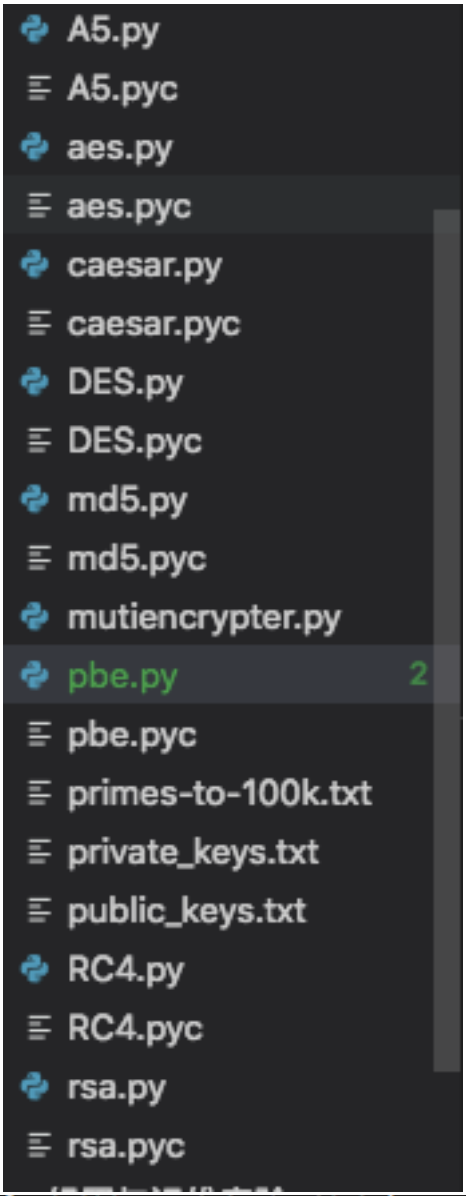
三、 实验基本原理及步骤（或方案设计及理论计算）



Mutiencrypter 的程序流程图如下

这里支持八种加密算法，且拥有时间记录，统计功能

由于功能较多，所以采用 Python 包的方式引用各个函数文件，这样可以使得主程序简单易懂，方便增加/减少功能，由于实现的密码算法较多，不一一列举原理，以节省篇幅




```
import time
import rsa
import caesar
import aes
import pbe
import DES
import md5
import RC4
import A5
costdic={}
```

```
(base) cyf:mutiencrypt/ $ python2 ./mutiencrypt.py
Welcome to mutiencrypt! by 16020610025 Caoyinfeng

Please choose encryption:
1 RSA
2 caesar
3 AES CBC
4 PBE
5 DES
6 MD5
7 RC4
8 A5
```

用户首先进入算法选择界面，可以选择 8 种加密模式，每种模式均有对应的加解密操作。

```
start = time.clock()
cipher=caesar.caesar(message,encryption_key)
end = time.clock()
print('ciphertext is '+cipher+"\n")
print('-------------------------caesar end-----

print ('Cost '+str(end-start)+'s\n')
```

利用 python 的 time 模块统计执行加密/解密前后的时间差值，从而得出加解密时间

1. RSA

RSA 加密算法是一种非对称加密算法。在公开密钥加密和电子商业中 RSA 被广泛使用

首先选择生成公私钥对，然后进行加密，这里用用钥加密

```
1
-------------------------rsa-------------------------

Do you want to generate new public and private keys? (y or n)
y
Would you like to encrypt or decrypt? (Enter e or d):
e
What would you like to encrypt?
abc
Do you want to encrypt using your own public key? (y or n)
y
433265 523973
-------------------------rsa end-------------------------

Cost 1.322488s
```

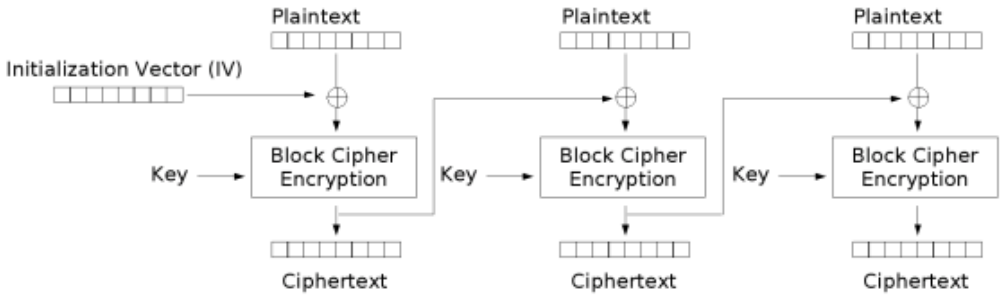
2. Caesar 密码

凯撒是一种替换加密的技术，明文中的所有字母都在字母表上向后（或向前）按照一个固定数目进行偏移后被替换成密文。例如，当偏移量是 3 的时候，所有的字母 A 将被替换成 D，B 变成 E，以此类推。

```
-------------------------caesar-------------------------

What is your encryption key?
3
What would you like to encrypt?
abc
ciphertext is def

-------------------------caesar end-------------------------

Cost 0.000146s
```

3. AES CBC 模式

1976 年，IBM 发明了密码分组链接（CBC，Cipher-block chaining）模式。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。



Cipher Block Chaining (CBC) mode encryption

4. PBE

PBE（Password Based Encryption，基于口令加密）算法是一种基于口令的加密算法，其特点在于口令是由用户自己掌握的，采用随机数杂凑多重加密等方法保证数据的安全性。

PBE 算法没有密钥的概念，密钥在其它对称加密算法中是经过算法计算得出来的，PBE 算法则是使用口令替代了密钥。

```
4
----------------------------PBE----------------------------

What would you like to encrypt?
abcdefgh
What is your encryption key?
abcdefgh
jFlEfR0wt9290e0z8gNEw7oPwMGcA5iS
abcdefgh
------------------------PBE end------------------------

Cost 0.005813s
```

5. DES

数据加密标准（英语：Data Encryption Standard，缩写为 DES）是一种对称密钥加密块密码算法，1976 年被美国联邦政府的国家标准局确定为联邦资料处理标准（FIPS），随后在国际上广泛流传开来。它基于使用 56 位密钥的对称算法。

```
5
----------------------------DES----------------------------

What is your encryption key?
abcdefgh
What would you like to encrypt?
abcdefgh
Cipher: '*\x8di\xde\x9d_\xdf\xf9'
('Deciphered: ', 'abcdefgh')
----------------------------DES end----------------------------
```

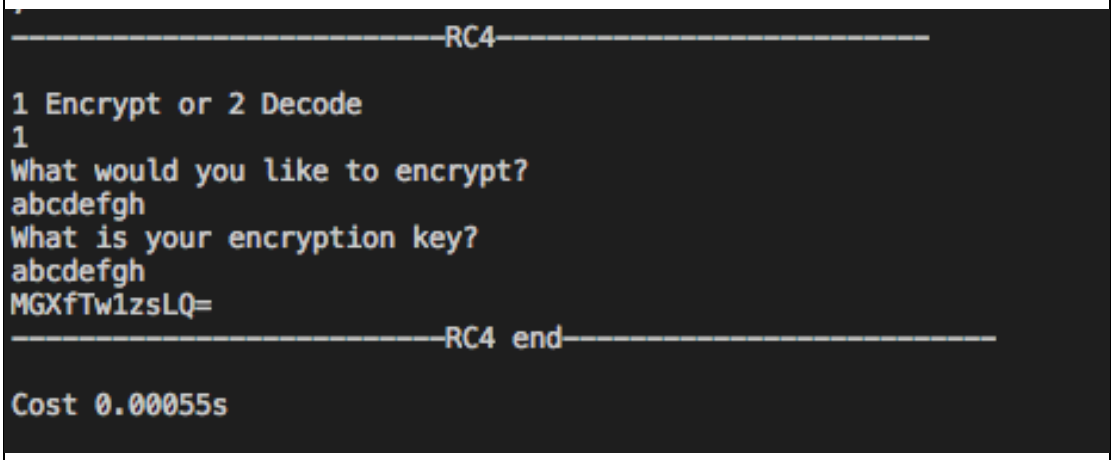
6. MD5

MD5 消息摘要算法（英语：MD5 Message-Digest Algorithm），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（hash value），用于确保信息传输完整一致。MD5 由美国密码学家罗纳德·李维斯特（Ronald Linn Rivest）设计，于 1992 年公开，用以取代 MD4 算法。这套算法的程序在 RFC 1321 中被加以规范

```
6
----------------------------MD5----------------------------

What would you like to hash?
abc
900150983cd24fb0d6963f7d28e17f72
----------------------------MD5 end----------------------------

Cost 0.003915s
```

7. RC4

在密码学中，RC4（来自 Rivest Cipher 4 的缩写）是一种流加密算法，密钥长度可变。它加解密使用相同的密钥，因此也属于对称加密算法。RC4 是有线等效加密（WEP）中采用的加密算法，也曾经是 TLS 可采用的算法之一。由美国密码学家罗纳德·李维斯特（Ronald Rivest）在 1987 年设计的。由于 RC4 算法存在弱点，2015 年 2 月所发布的 RFC 7465 规定禁止在 TLS 中使



用 RC4 加密算法

8. A5

A5 / 1 是用于在 GSM 蜂窝电话标准中提供无线通信隐私的流密码。它是为 GSM 使用指定的七种算法之一。最初是保密的，但通过泄漏和逆向工程成为公众的知识。 密码中存在一些严重的弱点。

```
8
------------------------A5--------------------------

1 Encrypt or 2 Decode
1
What would you like to encrypt?
abcdefgh
What is your encryption key?
abcdefgh
0x8088166c752dc0d5L
------------------------A5--------------------------

Cost 0.003609s
```

排序后的各加密算法耗时

```
1 RSA
2 caesar
3 AES CBC
4 PBE
5 DES
6 MD5
7 RC4
8 A5

[(1, 1.322488), (4, 0.005813000000000068), (3, 0.004662000000000166), (5, 0.00433899999999
9999999998284), (2, 0.0001460000000020152)]
```

四、 实验数据记录（或仿真及软件设计）

RSA（非对称）	1.322488
PBE（对称）	0.005813
AES（对称）	0.004662
DES（对称）	0.004339
MD5（哈希）	0.003915
A5（流密码）	0.003609
RC4（流密码）	0.00055
Caesar（置换密码）	0.000146

五、 实验结果分析及回答问题（或测试环境及测试结果）

经统计，对于同样的短密文，实现的各种算法时间如上（单位/s），基本符合预期。

一般情况下开销排序：

非对称加密>对称加密>哈希函数>流密码>简单置换加密

这是由于加密算法本身的函数决定的，如 RSA 的非对称加密涉及到幂指数运算，速度较慢；而流密码，置换密码开销则相当小，因为其往往只涉及到一些位运算，映射等线性预算，加密速度极快。

六、 主体源代码

这里仅列举主体源码，使用 IF_else 结构，方便更改，清晰易懂。
算法均具体实现，尽量减少直接调库，加深对算法的理解，具体见压缩包
使用命令
python2 ./mutiencrypter.py
即可运行（完整见电子版）

```
import time
import rsa
import caesar
import aes
import pbe
import DES
import md5
import RC4
import A5
costdic={}
while 1:
   print('Welcome to mutiencrypt! by 16020610025 Caoyinfeng\n')
   print('Please choose encryption:')
   print('1 RSA\n2 caesar\n3 AES CBC\n4 PBE\n5 DES\n6 MD5\n7
RC4\n8 A5\n')
   choose=int(raw_input())
   if choose==1:
      print('-------------------------rsa-----------------------
--\n')
      choose_again = raw_input('Do you want to generate new public
and private keys? (y or n)\n')
      if (choose_again == 'y'):
         rsa.chooseKeys()

      instruction = raw_input('Would you like to encrypt or
decrypt? (Enter e or d): \n')
      if (instruction == 'e'):
         message = raw_input('What would you like to encrypt?\n')
         option = raw_input('Do you want to encrypt using your own
public key? (y or n) \n')
```

```
            if (option == 'y'):
                start = time.clock()
                print(rsa.encrypt(message))
                end = time.clock()
            else:
                file_option = raw_input('Enter the file name that
stores the public key: \n')
                print(rsa.encrypt(message, file_option))

        elif (instruction == 'd'):
            message = raw_input('What would you like to decrypt?\n')
            print('Decryption...')
            print(rsa.decrypt(message))
        else:
            print('That is not a proper instruction.')
        print('--------------------------rsa end--------------------
------\n')

        print ('Cost '+str(end-start)+'s\n')
        costdic[choose]=end-start


    elif choose==2:

        print('--------------------------caesar---------------------
-----\n')
        encryption_key=int(raw_input('What is your encryption
key?\n'))
        message=raw_input('What would you like to encrypt?\n')
        start = time.clock()
        cipher=caesar.caesar(message,encryption_key)
        end = time.clock()
        print('ciphertext is '+cipher+"\n")
        print('--------------------------caesar end-----------------
---------\n')

        print ('Cost '+str(end-start)+'s\n')
        costdic[choose]=end-start
    elif choose==3:

        print('-------------------------AES CBC--------------------
------\n')
        moo = aes.AESModeOfOperation()
```

```
        cleartext = raw_input('What would you like to encrypt?\n')
        start = time.clock()
        cypherkey =
[143,194,34,208,145,203,230,143,177,246,97,206,145,92,255,84]
        iv =
[103,35,148,239,76,213,47,118,255,222,123,176,106,134,98,92]
        mode, orig_len, ciph = moo.encrypt(cleartext,
moo.modeOfOperation["CBC"],
                cypherkey, moo.aes.keySize["SIZE_128"], iv)
        print 'm=%s, ol=%s (%s), ciph=%s' % (mode, orig_len,
len(cleartext), ciph)
        end = time.clock()
        decr = moo.decrypt(ciph, orig_len, mode, cypherkey,
                moo.aes.keySize["SIZE_128"], iv)
        print decr
        aes.testStr(cleartext, 16, "CBC")
        print('--------------------------AES CBC end----------------
----------\n')

        print ('Cost '+str(end-start)+'s\n')
        costdic[choose]=end-start
    elif choose==4:

        print('--------------------------PBE----------------------
--\n')
        msg = raw_input('What would you like to encrypt?\n')
        passwd = raw_input('What is your encryption key?\n')
        start = time.clock()
        s = pbe.encrypt(msg, passwd)
        end = time.clock()
        print (s)
        print (pbe.decrypt(s, passwd))
        print('--------------------------PBE end--------------------
------\n')
        print ('Cost '+str(end-start)+'s\n')
        costdic[choose]=end-start
    elif choose==5:

        print('--------------------------DES----------------------
--\n')
        key = raw_input('What is your encryption key?\n')
        text= raw_input('What would you like to encrypt?\n')
        #
        start = time.clock()
```

```
        d = DES.des()
        r = d.encrypt(key,text)
        end = time.clock()
        r2 = d.decrypt(key,r)
        print("Cipher: %r" % r)

        print("Deciphered: ", str(r2))
        print('-------------------------DES end-------------------
------\n')
        print ('Cost '+str(end-start)+'s\n')
        costdic[choose]=end-start
    elif choose==6:
        print('-------------------------MD5-----------------------
--\n')

        mess = raw_input("What would you like to hash?\n")
        start = time.clock()
        md5.init_mess(mess)
        out_put = md5.hex_digest()
        print out_put
        end = time.clock()
        print('-------------------------MD5 end-------------------
------\n')
        print ('Cost '+str(end-start)+'s\n')
        costdic[choose]=end-start
    elif choose==7:
        print('-------------------------RC4-----------------------
--\n')
        mode = raw_input("1 Encrypt or 2 Decode \n")
        if mode == '1':
            start = time.clock()

            message = RC4.get_message()
            key = RC4.get_key()
            box = RC4.init_box(key)
            RC4.ex_encrypt(message,box,mode)
            end = time.clock()

        elif mode == '2':
            message = RC4.get_message()
            key = RC4.get_key()
            box = RC4.init_box(key)
            RC4.ex_encrypt(message, box, mode)
```

```
        print('-------------------------RC4 end-------------------
------\n')
        print ('Cost '+str(end-start)+'s\n')
        costdic[choose]=end-start
    elif choose==8:
        print('-------------------------A5-----------------------
-\n')

        choice = raw_input("1 Encrypt or 2 Decode \n")
        if choice == '1':
            message = raw_input('What would you like to encrypt?\n')
            start = time.clock()
            A5.a5_encode(message)
            end = time.clock()
        elif choice == '2':
            bin_message = raw_input('What would you like to
encrypt?\n')
            A5.a5_decode(bin_message)
        print('-------------------------A5-----------------------
-\n')
        print ('Cost '+str(end-start)+'s\n')
        costdic[choose]=end-start
    reslist = sorted(costdic.items(), key=lambda d:d[1],reverse =
True)
    print('1 RSA\n2 caesar\n3 AES CBC\n4 PBE\n5 DES\n6 MD5\n7
RC4\n8 A5\n')
    print(reslist)
```

1. RSA

```
import random

def gcd(a, b):

    if (b == 0):
        return a
    else:
        return gcd(b, a % b)

def xgcd(a, b):

    x, old_x = 0, 1
    y, old_y = 1, 0
```

```
    while (b != 0):
        quotient = a // b
        a, b = b, a - quotient * b
        old_x, x = x, old_x - quotient * x
        old_y, y = y, old_y - quotient * y

    return a, old_x, old_y

def chooseE(totient):

    while (True):
        e = random.randrange(2, totient)

        if (gcd(e, totient) == 1):
            return e

def chooseKeys():

    rand1 = random.randint(100, 300)
    rand2 = random.randint(100, 300)

    fo = open('primes-to-100k.txt', 'r')
    lines = fo.read().splitlines()
    fo.close()

    prime1 = int(lines[rand1])
    prime2 = int(lines[rand2])

    n = prime1 * prime2
    totient = (prime1 - 1) * (prime2 - 1)
    e = chooseE(totient)

    gcd, x, y = xgcd(e, totient)

    if (x < 0):
        d = x + totient
    else:
        d = x

    f_public = open('public_keys.txt', 'w')
    f_public.write(str(n) + '\n')
    f_public.write(str(e) + '\n')
    f_public.close()
```

```
    f_private = open('private_keys.txt', 'w')
    f_private.write(str(n) + '\n')
    f_private.write(str(d) + '\n')
    f_private.close()

def encrypt(message, file_name = 'public_keys.txt', block_size =
2):

    try:
        fo = open(file_name, 'r')


    except FileNotFoundError:
        print('That file is not found.')
    else:
        n = int(fo.readline())
        e = int(fo.readline())
        fo.close()

        encrypted_blocks = []
        ciphertext = -1

        if (len(message) > 0):
            ciphertext = ord(message[0])

        for i in range(1, len(message)):

            if (i % block_size == 0):
                encrypted_blocks.append(ciphertext)
                ciphertext = 0


            ciphertext = ciphertext * 1000 + ord(message[i])

        encrypted_blocks.append(ciphertext)


        for i in range(len(encrypted_blocks)):
            encrypted_blocks[i] = str((encrypted_blocks[i]**e) % n)

        encrypted_message = " ".join(encrypted_blocks)

        return encrypted_message
```

```
def decrypt(blocks, block_size = 2):


    fo = open('private_keys.txt', 'r')
    n = int(fo.readline())
    d = int(fo.readline())
    fo.close()

    # turns the string into a list of ints
    list_blocks = blocks.split(' ')
    int_blocks = []

    for s in list_blocks:
        int_blocks.append(int(s))

    message = ""

    # converts each int in the list to block_size number of
characters
    # by default, each int represents two characters
    for i in range(len(int_blocks)):

        int_blocks[i] = (int_blocks[i]**d) % n

        tmp = ""

        for c in range(block_size):
            tmp = chr(int_blocks[i] % 1000) + tmp
            int_blocks[i] //= 1000
        message += tmp

    return message

def main():
    # we select our primes and generate our public and private
keys,
    # usually done once
    choose_again = input('Do you want to generate new public and
private keys? (y or n) ')
    if (choose_again == 'y'):
        chooseKeys()

    instruction = input('Would you like to encrypt or decrypt?
(Enter e or d): ')
```

```
    if (instruction == 'e'):
        message = input('What would you like to encrypt?\n')
        option = input('Do you want to encrypt using your own public
key? (y or n) ')

        if (option == 'y'):
            print('Encrypting...')
            print(encrypt(message))
        else:
            file_option = input('Enter the file name that stores the
public key: ')
            print('Encrypting...')
            print(encrypt(message, file_option))

    elif (instruction == 'd'):
        message = input('What would you like to decrypt?\n')
        print('Decryption...')
        print(decrypt(message))
    else:
        print('That is not a proper instruction.')
```

2. Ceasar

```
import string
def encode_dict(encryption_key):
    encoding={}
    alphabet = string.ascii_lowercase + " "
    for i in range(len(alphabet)):
        encoding[alphabet[i]]=(i+encryption_key)%27
    return encoding

def caesar(message,encryption_key):
    # return the encoded message as a single string!
    alphabet = string.ascii_lowercase + " "

# create `letters` here!
    letters={}
    for i in range(len(alphabet)):
        letters[i]=alphabet[i]
    encoded_message = ''
    # use the function in Step 2 to get the encoding dictionary
    encoding = encode_dict(encryption_key)
    for char in message:
```

```
        encoded_message+=letters[encoding[char]]
    return encoded_message
    # your code is here
    # for each letter in message, get the encoded letter
```

3. AES CBC

```
import os
import sys
import math

class AES(object):
    '''AES funtions for a single block
    '''
    # Very annoying code:  all is for an object, but no state is
kept!
    # Should just be plain functions in a AES modlule.

    # valid key sizes
    keySize = dict(SIZE_128=16, SIZE_192=24, SIZE_256=32)

    # Rijndael S-box
    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]

    # Rijndael Inverted S-box
    rsbox = [0x52, 0x09, 0x6a, 0xd5, 0x30, 0x36, 0xa5, 0x38, 0xbf,
0x40, 0xa3,
          0x9e, 0x81, 0xf3, 0xd7, 0xfb , 0x7c, 0xe3, 0x39, 0x82,
0x9b, 0x2f,
          0xff, 0x87, 0x34, 0x8e, 0x43, 0x44, 0xc4, 0xde, 0xe9,
0xcb , 0x54,
          0x7b, 0x94, 0x32, 0xa6, 0xc2, 0x23, 0x3d, 0xee, 0x4c,
0x95, 0x0b,
          0x42, 0xfa, 0xc3, 0x4e , 0x08, 0x2e, 0xa1, 0x66, 0x28,
0xd9, 0x24,
          0xb2, 0x76, 0x5b, 0xa2, 0x49, 0x6d, 0x8b, 0xd1, 0x25 ,
0x72, 0xf8,
          0xf6, 0x64, 0x86, 0x68, 0x98, 0x16, 0xd4, 0xa4, 0x5c,
0xcc, 0x5d,
```

```
            0x65, 0xb6, 0x92 , 0x6c, 0x70, 0x48, 0x50, 0xfd, 0xed,
0xb9, 0xda,
            0x5e, 0x15, 0x46, 0x57, 0xa7, 0x8d, 0x9d, 0x84 , 0x90,
0xd8, 0xab,
            0x00, 0x8c, 0xbc, 0xd3, 0x0a, 0xf7, 0xe4, 0x58, 0x05,
0xb8, 0xb3,
            0x45, 0x06 , 0xd0, 0x2c, 0x1e, 0x8f, 0xca, 0x3f, 0x0f,
0x02, 0xc1,
            0xaf, 0xbd, 0x03, 0x01, 0x13, 0x8a, 0x6b , 0x3a, 0x91,
0x11, 0x41,
            0x4f, 0x67, 0xdc, 0xea, 0x97, 0xf2, 0xcf, 0xce, 0xf0,
0xb4, 0xe6,
            0x73 , 0x96, 0xac, 0x74, 0x22, 0xe7, 0xad, 0x35, 0x85,
0xe2, 0xf9,
            0x37, 0xe8, 0x1c, 0x75, 0xdf, 0x6e , 0x47, 0xf1, 0x1a,
0x71, 0x1d,
            0x29, 0xc5, 0x89, 0x6f, 0xb7, 0x62, 0x0e, 0xaa, 0x18,
0xbe, 0x1b ,
            0xfc, 0x56, 0x3e, 0x4b, 0xc6, 0xd2, 0x79, 0x20, 0x9a,
0xdb, 0xc0,
            0xfe, 0x78, 0xcd, 0x5a, 0xf4 , 0x1f, 0xdd, 0xa8, 0x33,
0x88, 0x07,
            0xc7, 0x31, 0xb1, 0x12, 0x10, 0x59, 0x27, 0x80, 0xec,
0x5f , 0x60,
            0x51, 0x7f, 0xa9, 0x19, 0xb5, 0x4a, 0x0d, 0x2d, 0xe5,
0x7a, 0x9f,
            0x93, 0xc9, 0x9c, 0xef , 0xa0, 0xe0, 0x3b, 0x4d, 0xae,
0x2a, 0xf5,
            0xb0, 0xc8, 0xeb, 0xbb, 0x3c, 0x83, 0x53, 0x99, 0x61 ,
0x17, 0x2b,
            0x04, 0x7e, 0xba, 0x77, 0xd6, 0x26, 0xe1, 0x69, 0x14,
0x63, 0x55,
            0x21, 0x0c, 0x7d]

    def getSBoxValue(self,num):
        """Retrieves a given S-Box Value"""
        return self.sbox[num]

    def getSBoxInvert(self,num):
        """Retrieves a given Inverted S-Box Value"""
        return self.rsbox[num]

    def rotate(self, word):
        """ Rijndael's key schedule rotate operation.
```

```
        Rotate a word eight bits to the left: eg, rotate(1d2c3a4f)
== 2c3a4f1d
        Word is an char list of size 4 (32 bits overall).
        """
        return word[1:] + word[:1]

    # Rijndael Rcon
    Rcon = [0x8d, 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80,
0x1b, 0x36,
            0x6c, 0xd8, 0xab, 0x4d, 0x9a, 0x2f, 0x5e, 0xbc, 0x63,
0xc6, 0x97,
            0x35, 0x6a, 0xd4, 0xb3, 0x7d, 0xfa, 0xef, 0xc5, 0x91,
0x39, 0x72,
            0xe4, 0xd3, 0xbd, 0x61, 0xc2, 0x9f, 0x25, 0x4a, 0x94,
0x33, 0x66,
            0xcc, 0x83, 0x1d, 0x3a, 0x74, 0xe8, 0xcb, 0x8d, 0x01,
0x02, 0x04,
            0x08, 0x10, 0x20, 0x40, 0x80, 0x1b, 0x36, 0x6c, 0xd8,
0xab, 0x4d,
            0x9a, 0x2f, 0x5e, 0xbc, 0x63, 0xc6, 0x97, 0x35, 0x6a,
0xd4, 0xb3,
            0x7d, 0xfa, 0xef, 0xc5, 0x91, 0x39, 0x72, 0xe4, 0xd3,
0xbd, 0x61,
            0xc2, 0x9f, 0x25, 0x4a, 0x94, 0x33, 0x66, 0xcc, 0x83,
0x1d, 0x3a,
            0x74, 0xe8, 0xcb, 0x8d, 0x01, 0x02, 0x04, 0x08, 0x10,
0x20, 0x40,
            0x80, 0x1b, 0x36, 0x6c, 0xd8, 0xab, 0x4d, 0x9a, 0x2f,
0x5e, 0xbc,
            0x63, 0xc6, 0x97, 0x35, 0x6a, 0xd4, 0xb3, 0x7d, 0xfa,
0xef, 0xc5,
            0x91, 0x39, 0x72, 0xe4, 0xd3, 0xbd, 0x61, 0xc2, 0x9f,
0x25, 0x4a,
            0x94, 0x33, 0x66, 0xcc, 0x83, 0x1d, 0x3a, 0x74, 0xe8,
0xcb, 0x8d,
            0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x1b,
0x36, 0x6c,
            0xd8, 0xab, 0x4d, 0x9a, 0x2f, 0x5e, 0xbc, 0x63, 0xc6,
0x97, 0x35,
            0x6a, 0xd4, 0xb3, 0x7d, 0xfa, 0xef, 0xc5, 0x91, 0x39,
0x72, 0xe4,
            0xd3, 0xbd, 0x61, 0xc2, 0x9f, 0x25, 0x4a, 0x94, 0x33,
0x66, 0xcc,
```

```
            0x83, 0x1d, 0x3a, 0x74, 0xe8, 0xcb, 0x8d, 0x01, 0x02,
0x04, 0x08,
            0x10, 0x20, 0x40, 0x80, 0x1b, 0x36, 0x6c, 0xd8, 0xab,
0x4d, 0x9a,
            0x2f, 0x5e, 0xbc, 0x63, 0xc6, 0x97, 0x35, 0x6a, 0xd4,
0xb3, 0x7d,
            0xfa, 0xef, 0xc5, 0x91, 0x39, 0x72, 0xe4, 0xd3, 0xbd,
0x61, 0xc2,
            0x9f, 0x25, 0x4a, 0x94, 0x33, 0x66, 0xcc, 0x83, 0x1d,
0x3a, 0x74,
            0xe8, 0xcb ]

    def getRconValue(self, num):
        """Retrieves a given Rcon Value"""
        return self.Rcon[num]

    def core(self, word, iteration):
        """Key schedule core."""
        # rotate the 32-bit word 8 bits to the left
        word = self.rotate(word)
        # apply S-Box substitution on all 4 parts of the 32-bit word
        for i in range(4):
            word[i] = self.getSBoxValue(word[i])
        # XOR the output of the rcon operation with i to the first
part
        # (leftmost) only
        word[0] = word[0] ^ self.getRconValue(iteration)
        return word

    def expandKey(self, key, size, expandedKeySize):
        """Rijndael's key expansion.

        Expands an 128,192,256 key into an 176,208,240 bytes key

        expandedKey is a char list of large enough size,
        key is the non-expanded key.
        """
        # current expanded keySize, in bytes
        currentSize = 0
        rconIteration = 1
        expandedKey = [0] * expandedKeySize

        # set the 16, 24, 32 bytes of the expanded key to the input
key
```

```
        for j in range(size):
            expandedKey[j] = key[j]
        currentSize += size

        while currentSize < expandedKeySize:
            # assign the previous 4 bytes to the temporary value t
            t = expandedKey[currentSize-4:currentSize]

            # every 16,24,32 bytes we apply the core schedule to t
            # and increment rconIteration afterwards
            if currentSize % size == 0:
                t = self.core(t, rconIteration)
                rconIteration += 1
            # For 256-bit keys, we add an extra sbox to the
calculation
            if size == self.keySize["SIZE_256"] and ((currentSize %
size) == 16):
                for l in range(4): t[l] = self.getSBoxValue(t[l])

            # We XOR t with the four-byte block 16,24,32 bytes before
the new
            # expanded key.  This becomes the next four bytes in the
expanded
            # key.
            for m in range(4):
                expandedKey[currentSize] = expandedKey[currentSize -
size] ^ \
                        t[m]
                currentSize += 1

        return expandedKey

    def addRoundKey(self, state, roundKey):
        """Adds (XORs) the round key to the state."""
        for i in range(16):
            state[i] ^= roundKey[i]
        return state

    def createRoundKey(self, expandedKey, roundKeyPointer):
        """Create a round key.
        Creates a round key from the given expanded key and the
        position within the expanded key.
        """
        roundKey = [0] * 16
```

```
        for i in range(4):
            for j in range(4):
                roundKey[j*4+i] = expandedKey[roundKeyPointer + i*4 +
j]
        return roundKey

    def galois_multiplication(self, a, b):
        """Galois multiplication of 8 bit characters a and b."""
        p = 0
        for counter in range(8):
            if b & 1: p ^= a
            hi_bit_set = a & 0x80
            a <<= 1
            # keep a 8 bit
            a &= 0xFF
            if hi_bit_set:
                a ^= 0x1b
            b >>= 1
        return p

    #
    # substitute all the values from the state with the value in
the SBox
    # using the state value as index for the SBox
    #
    def subBytes(self, state, isInv):
        if isInv: getter = self.getSBoxInvert
        else: getter = self.getSBoxValue
        for i in range(16): state[i] = getter(state[i])
        return state

    # iterate over the 4 rows and call shiftRow() with that row
    def shiftRows(self, state, isInv):
        for i in range(4):
            state = self.shiftRow(state, i*4, i, isInv)
        return state

    # each iteration shifts the row to the left by 1
    def shiftRow(self, state, statePointer, nbr, isInv):
        for i in range(nbr):
            if isInv:
                state[statePointer:statePointer+4] = \
                        state[statePointer+3:statePointer+4] + \
                        state[statePointer:statePointer+3]
```

```
            else:
                state[statePointer:statePointer+4] = \
                      state[statePointer+1:statePointer+4] + \
                      state[statePointer:statePointer+1]
        return state

    # galois multiplication of the 4x4 matrix
    def mixColumns(self, state, isInv):
        # iterate over the 4 columns
        for i in range(4):
            # construct one column by slicing over the 4 rows
            column = state[i:i+16:4]
            # apply the mixColumn on one column
            column = self.mixColumn(column, isInv)
            # put the values back into the state
            state[i:i+16:4] = column

        return state

    # galois multiplication of 1 column of the 4x4 matrix
    def mixColumn(self, column, isInv):
        if isInv: mult = [14, 9, 13, 11]
        else: mult = [2, 1, 1, 3]
        cpy = list(column)
        g = self.galois_multiplication

        column[0] = g(cpy[0], mult[0]) ^ g(cpy[3], mult[1]) ^ \
                    g(cpy[2], mult[2]) ^ g(cpy[1], mult[3])
        column[1] = g(cpy[1], mult[0]) ^ g(cpy[0], mult[1]) ^ \
                    g(cpy[3], mult[2]) ^ g(cpy[2], mult[3])
        column[2] = g(cpy[2], mult[0]) ^ g(cpy[1], mult[1]) ^ \
                    g(cpy[0], mult[2]) ^ g(cpy[3], mult[3])
        column[3] = g(cpy[3], mult[0]) ^ g(cpy[2], mult[1]) ^ \
                    g(cpy[1], mult[2]) ^ g(cpy[0], mult[3])
        return column

    # applies the 4 operations of the forward round in sequence
    def aes_round(self, state, roundKey):
        state = self.subBytes(state, False)
        state = self.shiftRows(state, False)
        state = self.mixColumns(state, False)
        state = self.addRoundKey(state, roundKey)
        return state
```

```
    # applies the 4 operations of the inverse round in sequence
    def aes_invRound(self, state, roundKey):
        state = self.shiftRows(state, True)
        state = self.subBytes(state, True)
        state = self.addRoundKey(state, roundKey)
        state = self.mixColumns(state, True)
        return state

    # Perform the initial operations, the standard round, and the
final
    # operations of the forward aes, creating a round key for each
round
    def aes_main(self, state, expandedKey, nbrRounds):
        state = self.addRoundKey(state,
self.createRoundKey(expandedKey, 0))
        i = 1
        while i < nbrRounds:
            state = self.aes_round(state,
                                   self.createRoundKey(expandedKey,
16*i))
            i += 1
        state = self.subBytes(state, False)
        state = self.shiftRows(state, False)
        state = self.addRoundKey(state,
                                 self.createRoundKey(expandedKey,
16*nbrRounds))
        return state

    # Perform the initial operations, the standard round, and the
final
    # operations of the inverse aes, creating a round key for each
round
    def aes_invMain(self, state, expandedKey, nbrRounds):
        state = self.addRoundKey(state,
                                 self.createRoundKey(expandedKey,
16*nbrRounds))
        i = nbrRounds - 1
        while i > 0:
            state = self.aes_invRound(state,
                                      self.createRoundKey(expandedKey,
16*i))
            i -= 1
        state = self.shiftRows(state, True)
        state = self.subBytes(state, True)
```

```
        state = self.addRoundKey(state,
self.createRoundKey(expandedKey, 0))
        return state

    # encrypts a 128 bit input block against the given key of size
specified
    def encrypt(self, iput, key, size):
        output = [0] * 16
        # the number of rounds
        nbrRounds = 0
        # the 128 bit block to encode
        block = [0] * 16
        # set the number of rounds
        if size == self.keySize["SIZE_128"]: nbrRounds = 10
        elif size == self.keySize["SIZE_192"]: nbrRounds = 12
        elif size == self.keySize["SIZE_256"]: nbrRounds = 14
        else: return None

        # the expanded keySize
        expandedKeySize = 16*(nbrRounds+1)

        # Set the block values, for the block:
        # a0,0 a0,1 a0,2 a0,3
        # a1,0 a1,1 a1,2 a1,3
        # a2,0 a2,1 a2,2 a2,3
        # a3,0 a3,1 a3,2 a3,3
        # the mapping order is a0,0 a1,0 a2,0 a3,0 a0,1 a1,1 ...
a2,3 a3,3
        #
        # iterate over the columns
        for i in range(4):
            # iterate over the rows
            for j in range(4):
                block[(i+(j*4))] = iput[(i*4)+j]

        # expand the key into an 176, 208, 240 bytes key
        # the expanded key
        expandedKey = self.expandKey(key, size, expandedKeySize)

        # encrypt the block using the expandedKey
        block = self.aes_main(block, expandedKey, nbrRounds)

        # unmap the block again into the output
        for k in range(4):
```

```
            # iterate over the rows
            for l in range(4):
                output[(k*4)+l] = block[(k+(l*4))]
        return output

    # decrypts a 128 bit input block against the given key of size
specified
    def decrypt(self, iput, key, size):
        output = [0] * 16
        # the number of rounds
        nbrRounds = 0
        # the 128 bit block to decode
        block = [0] * 16
        # set the number of rounds
        if size == self.keySize["SIZE_128"]: nbrRounds = 10
        elif size == self.keySize["SIZE_192"]: nbrRounds = 12
        elif size == self.keySize["SIZE_256"]: nbrRounds = 14
        else: return None

        # the expanded keySize
        expandedKeySize = 16*(nbrRounds+1)

        # Set the block values, for the block:
        # a0,0 a0,1 a0,2 a0,3
        # a1,0 a1,1 a1,2 a1,3
        # a2,0 a2,1 a2,2 a2,3
        # a3,0 a3,1 a3,2 a3,3
        # the mapping order is a0,0 a1,0 a2,0 a3,0 a0,1 a1,1 ...
a2,3 a3,3

        # iterate over the columns
        for i in range(4):
            # iterate over the rows
            for j in range(4):
                block[(i+(j*4))] = iput[(i*4)+j]
        # expand the key into an 176, 208, 240 bytes key
        expandedKey = self.expandKey(key, size, expandedKeySize)
        # decrypt the block using the expandedKey
        block = self.aes_invMain(block, expandedKey, nbrRounds)
        # unmap the block again into the output
        for k in range(4):
            # iterate over the rows
            for l in range(4):
                output[(k*4)+l] = block[(k+(l*4))]
```

```
        return output


class AESModeOfOperation(object):
    '''Handles AES with plaintext consistingof multiple blocks.
    Choice of block encoding modes:  OFT, CFB, CBC
    '''
    # Very annoying code:  all is for an object, but no state is
kept!
    # Should just be plain functions in an AES_BlockMode module.
    aes = AES()

    # structure of supported modes of operation
    modeOfOperation = dict(OFB=0, CFB=1, CBC=2)

    # converts a 16 character string into a number array
    def convertString(self, string, start, end, mode):
        if end - start > 16: end = start + 16
        if mode == self.modeOfOperation["CBC"]: ar = [0] * 16
        else: ar = []

        i = start
        j = 0
        while len(ar) < end - start:
            ar.append(0)
        while i < end:
            ar[j] = ord(string[i])
            j += 1
            i += 1
        return ar

    # Mode of Operation Encryption
    # stringIn - Input String
    # mode - mode of type modeOfOperation
    # hexKey - a hex key of the bit length size
    # size - the bit length of the key
    # hexIV - the 128 bit hex Initilization Vector
    def encrypt(self, stringIn, mode, key, size, IV):
        if len(key) % size:
            return None
        if len(IV) % 16:
            return None
        # the AES input/output
        plaintext = []
```

```
        iput = [0] * 16
        output = []
        ciphertext = [0] * 16
        # the output cipher string
        cipherOut = []
        # char firstRound
        firstRound = True
        if stringIn != None:
            for j in range(int(math.ceil(float(len(stringIn))/16))):
                start = j*16
                end = j*16+16
                if  end > len(stringIn):
                    end = len(stringIn)
                plaintext = self.convertString(stringIn, start, end,
mode)
                # print 'PT@%s:%s' % (j, plaintext)
                if mode == self.modeOfOperation["CFB"]:
                    if firstRound:
                        output = self.aes.encrypt(IV, key, size)
                        firstRound = False
                    else:
                        output = self.aes.encrypt(iput, key, size)
                    for i in range(16):
                        if len(plaintext)-1 < i:
                            ciphertext[i] = 0 ^ output[i]
                        elif len(output)-1 < i:
                            ciphertext[i] = plaintext[i] ^ 0
                        elif len(plaintext)-1 < i and len(output) < i:
                            ciphertext[i] = 0 ^ 0
                        else:
                            ciphertext[i] = plaintext[i] ^ output[i]
                    for k in range(end-start):
                        cipherOut.append(ciphertext[k])
                    iput = ciphertext
                elif mode == self.modeOfOperation["OFB"]:
                    if firstRound:
                        output = self.aes.encrypt(IV, key, size)
                        firstRound = False
                    else:
                        output = self.aes.encrypt(iput, key, size)
                    for i in range(16):
                        if len(plaintext)-1 < i:
                            ciphertext[i] = 0 ^ output[i]
                        elif len(output)-1 < i:
```

```
                        ciphertext[i] = plaintext[i] ^ 0
                    elif len(plaintext)-1 < i and len(output) < i:
                        ciphertext[i] = 0 ^ 0
                    else:
                        ciphertext[i] = plaintext[i] ^ output[i]
                for k in range(end-start):
                    cipherOut.append(ciphertext[k])
                iput = output
            elif mode == self.modeOfOperation["CBC"]:
                for i in range(16):
                    if firstRound:
                        iput[i] =  plaintext[i] ^ IV[i]
                    else:
                        iput[i] =  plaintext[i] ^ ciphertext[i]
                # print 'IP@%s:%s' % (j, iput)
                firstRound = False
                ciphertext = self.aes.encrypt(iput, key, size)
                # always 16 bytes because of the padding for CBC
                for k in range(16):
                    cipherOut.append(ciphertext[k])
        return mode, len(stringIn), cipherOut

    # Mode of Operation Decryption
    # cipherIn - Encrypted String
    # originalsize - The unencrypted string length - required for
CBC
    # mode - mode of type modeOfOperation
    # key - a number array of the bit length size
    # size - the bit length of the key
    # IV - the 128 bit number array Initilization Vector
    def decrypt(self, cipherIn, originalsize, mode, key, size, IV):
        # cipherIn = unescCtrlChars(cipherIn)
        if len(key) % size:
            return None
        if len(IV) % 16:
            return None
        # the AES input/output
        ciphertext = []
        iput = []
        output = []
        plaintext = [0] * 16
        # the output plain text character list
        chrOut = []
        # char firstRound
```

```
        firstRound = True
        if cipherIn != None:
            for j in range(int(math.ceil(float(len(cipherIn))/16))):
                start = j*16
                end = j*16+16
                if j*16+16 > len(cipherIn):
                    end = len(cipherIn)
                ciphertext = cipherIn[start:end]
                if mode == self.modeOfOperation["CFB"]:
                    if firstRound:
                        output = self.aes.encrypt(IV, key, size)
                        firstRound = False
                    else:
                        output = self.aes.encrypt(iput, key, size)
                    for i in range(16):
                        if len(output)-1 < i:
                            plaintext[i] = 0 ^ ciphertext[i]
                        elif len(ciphertext)-1 < i:
                            plaintext[i] = output[i] ^ 0
                        elif len(output)-1 < i and len(ciphertext) < i:
                            plaintext[i] = 0 ^ 0
                        else:
                            plaintext[i] = output[i] ^ ciphertext[i]
                    for k in range(end-start):
                        chrOut.append(chr(plaintext[k]))
                    iput = ciphertext
                elif mode == self.modeOfOperation["OFB"]:
                    if firstRound:
                        output = self.aes.encrypt(IV, key, size)
                        firstRound = False
                    else:
                        output = self.aes.encrypt(iput, key, size)
                    for i in range(16):
                        if len(output)-1 < i:
                            plaintext[i] = 0 ^ ciphertext[i]
                        elif len(ciphertext)-1 < i:
                            plaintext[i] = output[i] ^ 0
                        elif len(output)-1 < i and len(ciphertext) < i:
                            plaintext[i] = 0 ^ 0
                        else:
                            plaintext[i] = output[i] ^ ciphertext[i]
                    for k in range(end-start):
                        chrOut.append(chr(plaintext[k]))
                    iput = output
```

```
            elif mode == self.modeOfOperation["CBC"]:
                output = self.aes.decrypt(ciphertext, key, size)
                for i in range(16):
                    if firstRound:
                        plaintext[i] = IV[i] ^ output[i]
                    else:
                        plaintext[i] = iput[i] ^ output[i]
                firstRound = False
                if originalsize is not None and originalsize <
end:
                    for k in range(originalsize-start):
                        chrOut.append(chr(plaintext[k]))
                else:
                    for k in range(end-start):
                        chrOut.append(chr(plaintext[k]))
                iput = ciphertext
        return "".join(chrOut)


def append_PKCS7_padding(s):
    """return s padded to a multiple of 16-bytes by PKCS7
padding"""
    numpads = 16 - (len(s)%16)
    return s + numpads*chr(numpads)

def strip_PKCS7_padding(s):
    """return s stripped of PKCS7 padding"""
    if len(s)%16 or not s:
        raise ValueError("String of len %d can't be PCKS7-padded" %
len(s))
    numpads = ord(s[-1])
    if numpads > 16:
        raise ValueError("String ending with %r can't be PCKS7-
padded" % s[-1])
    return s[:-numpads]

def encryptData(key, data,
mode=AESModeOfOperation.modeOfOperation["CBC"]):
    """encrypt `data` using `key`

    `key` should be a string of bytes.

    returned cipher is a string of bytes prepended with the
initialization
```

```
    vector.

    """
    key = map(ord, key)
    if mode == AESModeOfOperation.modeOfOperation["CBC"]:
        data = append_PKCS7_padding(data)
    keysize = len(key)
    assert keysize in AES.keySize.values(), 'invalid key
size: %s' % keysize
    # create a new iv using random data
    iv = [ord(i) for i in os.urandom(16)]
    moo = AESModeOfOperation()
    (mode, length, ciph) = moo.encrypt(data, mode, key, keysize,
iv)
    # With padding, the original length does not need to be known.
It's a bad
    # idea to store the original message length.
    # prepend the iv.
    return ''.join(map(chr, iv)) + ''.join(map(chr, ciph))

def decryptData(key, data,
mode=AESModeOfOperation.modeOfOperation["CBC"]):
    """decrypt `data` using `key`

    `key` should be a string of bytes.

    `data` should have the initialization vector prepended as a
string of
    ordinal values.
    """

    key = map(ord, key)
    keysize = len(key)
    assert keysize in AES.keySize.values(), 'invalid key
size: %s' % keysize
    # iv is first 16 bytes
    iv = map(ord, data[:16])
    data = map(ord, data[16:])
    moo = AESModeOfOperation()
    decr = moo.decrypt(data, None, mode, key, keysize, iv)
    if mode == AESModeOfOperation.modeOfOperation["CBC"]:
        decr = strip_PKCS7_padding(decr)
    return decr
```

```
def generateRandomKey(keysize):
    """Generates a key from random data of length `keysize`.
    The returned key is a string of bytes.
    """
    if keysize not in (16, 24, 32):
        emsg = 'Invalid keysize, %s. Should be one of (16, 24, 32).'
        raise ValueError, emsg % keysize
    return os.urandom(keysize)

def testStr(cleartext, keysize=16, modeName = "CBC"):
    '''Test with random key, choice of mode.'''
    print 'Random key test', 'Mode:', modeName
    print 'cleartext:', cleartext
    key =  generateRandomKey(keysize)
    print 'Key:', [ord(x) for x in key]
    mode = AESModeOfOperation.modeOfOperation[modeName]
    cipher = encryptData(key, cleartext, mode)
    print 'Cipher:', [ord(x) for x in cipher]
    decr = decryptData(key, cipher, mode)
    print 'Decrypted:', decr


if __name__ == "__main__":
    moo = AESModeOfOperation()
    cleartext = "This is a test with several blocks!"
    cypherkey =
[143,194,34,208,145,203,230,143,177,246,97,206,145,92,255,84]
    iv =
[103,35,148,239,76,213,47,118,255,222,123,176,106,134,98,92]
    mode, orig_len, ciph = moo.encrypt(cleartext,
moo.modeOfOperation["CBC"],
           cypherkey, moo.aes.keySize["SIZE_128"], iv)
    print 'm=%s, ol=%s (%s), ciph=%s' % (mode, orig_len,
len(cleartext), ciph)
    decr = moo.decrypt(ciph, orig_len, mode, cypherkey,
           moo.aes.keySize["SIZE_128"], iv)
    print decr
    testStr(cleartext, 16, "CBC")
```

4. PBE

```
import base64
import hashlib
import re
import os
from Crypto.Cipher import DES

def get_derived_key(password, salt, count):
    key = password + salt
    for i in range(count):
        m = hashlib.md5(key)
        key = m.digest()
    return (key[:8], key[8:])

def decrypt(msg, password):
    msg_bytes = base64.b64decode(msg)
    salt = msg_bytes[:8]
    enc_text = msg_bytes[8:]
    (dk, iv) = get_derived_key(password, salt, 1000)
    crypter = DES.new(dk, DES.MODE_CBC, iv)
    text = crypter.decrypt(enc_text)
    # remove the padding at the end, if any
    return re.sub(r'[\x01-\x08]','',text)

def encrypt(msg, password):
    salt = os.urandom(8)
    pad_num = 8 - (len(msg) % 8)
    for i in range(pad_num):
        msg += chr(pad_num)
    (dk, iv) = get_derived_key(password, salt, 1000)
    crypter = DES.new(dk, DES.MODE_CBC, iv)
    enc_text = crypter.encrypt(msg)
    return base64.b64encode(salt + enc_text)

def main():
    msg = "hello, world"
    passwd = "mypassword"
    s = encrypt(msg, passwd)
    print (s)
    print (decrypt(s, passwd))

if __name__ == "__main__":
```

```
    main()
```

5. DES

```
#-*- coding: utf8 -*-

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

#Initial permut made on the key
CP_1 = [57, 49, 41, 33, 25, 17, 9,
       1, 58, 50, 42, 34, 26, 18,
       10, 2, 59, 51, 43, 35, 27,
       19, 11, 3, 60, 52, 44, 36,
       63, 55, 47, 39, 31, 23, 15,
       7, 62, 54, 46, 38, 30, 22,
       14, 6, 61, 53, 45, 37, 29,
       21, 13, 5, 28, 20, 12, 4]

#Permut applied on shifted key to get Ki+1
CP_2 = [14, 17, 11, 24, 1, 5, 3, 28,
       15, 6, 21, 10, 23, 19, 12, 4,
       26, 8, 16, 7, 27, 20, 13, 2,
       41, 52, 31, 37, 47, 55, 30, 40,
       51, 45, 33, 48, 44, 49, 39, 56,
       34, 53, 46, 42, 50, 36, 29, 32]

#Expand matrix to get a 48bits matrix of datas to apply the xor
with Ki
E = [32, 1, 2, 3, 4, 5,
    4, 5, 6, 7, 8, 9,
    8, 9, 10, 11, 12, 13,
    12, 13, 14, 15, 16, 17,
    16, 17, 18, 19, 20, 21,
    20, 21, 22, 23, 24, 25,
    24, 25, 26, 27, 28, 29,
```

```
    28, 29, 30, 31, 32, 1]

#SBOX
S_BOX = [

[[14, 4, 13, 1, 2, 15, 11, 8, 3, 10, 6, 12, 5, 9, 0, 7],
 [0, 15, 7, 4, 14, 2, 13, 1, 10, 6, 12, 11, 9, 5, 3, 8],
 [4, 1, 14, 8, 13, 6, 2, 11, 15, 12, 9, 7, 3, 10, 5, 0],
 [15, 12, 8, 2, 4, 9, 1, 7, 5, 11, 3, 14, 10, 0, 6, 13],
],

[[15, 1, 8, 14, 6, 11, 3, 4, 9, 7, 2, 13, 12, 0, 5, 10],
 [3, 13, 4, 7, 15, 2, 8, 14, 12, 0, 1, 10, 6, 9, 11, 5],
 [0, 14, 7, 11, 10, 4, 13, 1, 5, 8, 12, 6, 9, 3, 2, 15],
 [13, 8, 10, 1, 3, 15, 4, 2, 11, 6, 7, 12, 0, 5, 14, 9],
],

[[10, 0, 9, 14, 6, 3, 15, 5, 1, 13, 12, 7, 11, 4, 2, 8],
 [13, 7, 0, 9, 3, 4, 6, 10, 2, 8, 5, 14, 12, 11, 15, 1],
 [13, 6, 4, 9, 8, 15, 3, 0, 11, 1, 2, 12, 5, 10, 14, 7],
 [1, 10, 13, 0, 6, 9, 8, 7, 4, 15, 14, 3, 11, 5, 2, 12],
],

[[7, 13, 14, 3, 0, 6, 9, 10, 1, 2, 8, 5, 11, 12, 4, 15],
 [13, 8, 11, 5, 6, 15, 0, 3, 4, 7, 2, 12, 1, 10, 14, 9],
 [10, 6, 9, 0, 12, 11, 7, 13, 15, 1, 3, 14, 5, 2, 8, 4],
 [3, 15, 0, 6, 10, 1, 13, 8, 9, 4, 5, 11, 12, 7, 2, 14],
],

[[2, 12, 4, 1, 7, 10, 11, 6, 8, 5, 3, 15, 13, 0, 14, 9],
 [14, 11, 2, 12, 4, 7, 13, 1, 5, 0, 15, 10, 3, 9, 8, 6],
 [4, 2, 1, 11, 10, 13, 7, 8, 15, 9, 12, 5, 6, 3, 0, 14],
 [11, 8, 12, 7, 1, 14, 2, 13, 6, 15, 0, 9, 10, 4, 5, 3],
],

[[12, 1, 10, 15, 9, 2, 6, 8, 0, 13, 3, 4, 14, 7, 5, 11],
 [10, 15, 4, 2, 7, 12, 9, 5, 6, 1, 13, 14, 0, 11, 3, 8],
 [9, 14, 15, 5, 2, 8, 12, 3, 7, 0, 4, 10, 1, 13, 11, 6],
 [4, 3, 2, 12, 9, 5, 15, 10, 11, 14, 1, 7, 6, 0, 8, 13],
],

[[4, 11, 2, 14, 15, 0, 8, 13, 3, 12, 9, 7, 5, 10, 6, 1],
 [13, 0, 11, 7, 4, 9, 1, 10, 14, 3, 5, 12, 2, 15, 8, 6],
 [1, 4, 11, 13, 12, 3, 7, 14, 10, 15, 6, 8, 0, 5, 9, 2],
```

```
 [6, 11, 13, 8, 1, 4, 10, 7, 9, 5, 0, 15, 14, 2, 3, 12],
],

[[13, 2, 8, 4, 6, 15, 11, 1, 10, 9, 3, 14, 5, 0, 12, 7],
 [1, 15, 13, 8, 10, 3, 7, 4, 12, 5, 6, 11, 0, 14, 9, 2],
 [7, 11, 4, 1, 9, 12, 14, 2, 0, 6, 10, 13, 15, 3, 5, 8],
 [2, 1, 14, 7, 4, 10, 8, 13, 15, 12, 9, 0, 3, 5, 6, 11],
]
]

#Permut made after each SBox substitution for each round
P = [16, 7, 20, 21, 29, 12, 28, 17,
    1, 15, 23, 26, 5, 18, 31, 10,
    2, 8, 24, 14, 32, 27, 3, 9,
    19, 13, 30, 6, 22, 11, 4, 25]

#Final permut for datas after the 16 rounds
PI_1 = [40, 8, 48, 16, 56, 24, 64, 32,
       39, 7, 47, 15, 55, 23, 63, 31,
       38, 6, 46, 14, 54, 22, 62, 30,
       37, 5, 45, 13, 53, 21, 61, 29,
       36, 4, 44, 12, 52, 20, 60, 28,
       35, 3, 43, 11, 51, 19, 59, 27,
       34, 2, 42, 10, 50, 18, 58, 26,
       33, 1, 41, 9, 49, 17, 57, 25]

#Matrix that determine the shift for each round of keys
SHIFT = [1,1,2,2,2,2,2,2,1,2,2,2,2,2,2,1]

def string_to_bit_array(text):#Convert a string into a list of bits
    array = list()
    for char in text:
        binval = binvalue(char, 8)#Get the char value on one byte
        array.extend([int(x) for x in list(binval)]) #Add the bits
to the final list
    return array

def bit_array_to_string(array): #Recreate the string from the bit
array
    res = ''.join([chr(int(y,2)) for y in [''.join([str(x) for x in
_bytes]) for _bytes in  nsplit(array,8)]])
    return res
```

```
def binvalue(val, bitsize): #Return the binary value as a string of
the given size
    binval = bin(val)[2:] if isinstance(val, int) else
bin(ord(val))[2:]
    if len(binval) > bitsize:
        raise "binary value larger than the expected size"
    while len(binval) < bitsize:
        binval = "0"+binval #Add as many 0 as needed to get the
wanted size
    return binval

def nsplit(s, n):#Split a list into sublists of size "n"
    return [s[k:k+n] for k in range(0, len(s), n)]

ENCRYPT=1
DECRYPT=0

class des():
    def __init__(self):
        self.password = None
        self.text = None
        self.keys = list()

    def run(self, key, text, action=ENCRYPT, padding=False):
        if len(key) < 8:
            raise "Key Should be 8 bytes long"
        elif len(key) > 8:
            key = key[:8] #If key size is above 8bytes, cut to be
8bytes long

        self.password = key
        self.text = text

        if padding and action==ENCRYPT:
            self.addPadding()
        elif len(self.text) % 8 != 0:#If not padding specified data
size must be multiple of 8 bytes
            raise "Data size should be multiple of 8"

        self.generatekeys() #Generate all the keys
        text_blocks = nsplit(self.text, 8) #Split the text in blocks
of 8 bytes so 64 bits
        result = list()
        for block in text_blocks:#Loop over all the blocks of data
```

```
            block = string_to_bit_array(block)#Convert the block in
bit array
            block = self.permut(block,PI)#Apply the initial
permutation
            g, d = nsplit(block, 32) #g(LEFT), d(RIGHT)
            tmp = None
            for i in range(16): #Do the 16 rounds
                d_e = self.expand(d, E) #Expand d to match Ki size
(48bits)
                if action == ENCRYPT:
                    tmp = self.xor(self.keys[i], d_e)#If encrypt use
Ki
                else:
                    tmp = self.xor(self.keys[15-i], d_e)#If decrypt
start by the last key
                tmp = self.substitute(tmp) #Method that will apply
the SBOXes
                tmp = self.permut(tmp, P)
                tmp = self.xor(g, tmp)
                g = d
                d = tmp
            result += self.permut(d+g, PI_1) #Do the last permut and
append the result to result
        final_res = bit_array_to_string(result)
        if padding and action==DECRYPT:
            return self.removePadding(final_res) #Remove the padding
if decrypt and padding is true
        else:
            return final_res #Return the final string of data
ciphered/deciphered

    def substitute(self, d_e):#Substitute bytes using SBOX
        subblocks = nsplit(d_e, 6)#Split bit array into sublist of 6
bits
        result = list()
        for i in range(len(subblocks)): #For all the sublists
            block = subblocks[i]
            row = int(str(block[0])+str(block[5]),2)#Get the row with
the first and last bit
            column = int(''.join([str(x) for x in block[1:][:-1]]),2)
#Column is the 2,3,4,5th bits
            val = S_BOX[i][row][column] #Take the value in the SBOX
appropriated for the round (i)
            bin = binvalue(val, 4)#Convert the value to binary
```

```
            result += [int(x) for x in bin]#And append it to the
resulting list
        return result

    def permut(self, block, table):#Permut the given block using
the given table (so generic method)
        return [block[x-1] for x in table]

    def expand(self, block, table):#Do the exact same thing than
permut but for more clarity has been renamed
        return [block[x-1] for x in table]

    def xor(self, t1, t2):#Apply a xor and return the resulting
list
        return [x^y for x,y in zip(t1,t2)]

    def generatekeys(self):#Algorithm that generates all the keys
        self.keys = []
        key = string_to_bit_array(self.password)
        key = self.permut(key, CP_1) #Apply the initial permut on
the key
        g, d = nsplit(key, 28) #Split it in to (g->LEFT),(d->RIGHT)
        for i in range(16):#Apply the 16 rounds
            g, d = self.shift(g, d, SHIFT[i]) #Apply the shift
associated with the round (not always 1)
            tmp = g + d #Merge them
            self.keys.append(self.permut(tmp, CP_2)) #Apply the
permut to get the Ki

    def shift(self, g, d, n): #Shift a list of the given value
        return g[n:] + g[:n], d[n:] + d[:n]

    def addPadding(self):#Add padding to the datas using PKCS5
spec.
        pad_len = 8 - (len(self.text) % 8)
        self.text += pad_len * chr(pad_len)

    def removePadding(self, data):#Remove the padding of the plain
text (it assume there is padding)
        pad_len = ord(data[-1])
        return data[:-pad_len]

    def encrypt(self, key, text, padding=False):
        return self.run(key, text, ENCRYPT, padding)
```

```
    def decrypt(self, key, text, padding=False):
        return self.run(key, text, DECRYPT, padding)

if __name__ == '__main__':
    key = "secret_k"
    text= "Hello wo"
    d = des()
    r = d.encrypt(key,text)
    r2 = d.decrypt(key,r)
    print("Ciphered: %r" % r)
    print("Deciphered: ", r2)
```

6. MD5

```
import struct
import math
import binascii

lrot = lambda x,n: (x << n)|(x >> 32- n)
A, B, C, D = (0x67452301, 0xefcdab89, 0x98badcfe, 0x10325476)
# A, B, C, D = (0x01234567, 0x89ABCDEF, 0xFEDCBA98, 0x76543210)

r = [   7, 12, 17, 22, 7, 12, 17, 22, 7, 12, 17, 22, 7, 12, 17, 22,
        5,  9, 14, 20, 5,  9, 14, 20, 5,  9, 14, 20, 5,  9, 14, 20,
        4, 11, 16, 23, 4, 11, 16, 23, 4, 11, 16, 23, 4, 11, 16, 23,
        6, 10, 15, 21, 6, 10, 15, 21, 6, 10, 15, 21, 6, 10, 15, 21
        ]

k =  [int(math.floor(abs(math.sin(i + 1)) * (2 ** 32))) for i in
range(64)]


def init_mess(message):
    global A
    global B
    global C
    global D
    A, B, C, D = (0x67452301, 0xefcdab89, 0x98badcfe, 0x10325476)
    # A, B, C, D = (0x01234567, 0x89ABCDEF, 0xFEDCBA98, 0x76543210)
    length = struct.pack('<Q', len(message)*8)
    while len(message) > 64:
        solve(message[:64])
```

```
        message = message[64:]

    message += '\x80'
    message += '\x00' * (56 - len(message) % 64)
    #print type(length)
    message += length
    # print binascii.b2a_hex(message)
    solve(message[:64])


def solve(chunk):
    global A
    global B
    global C
    global D
    w = list(struct.unpack('<' + 'I' * 16, chunk))
    a, b, c, d = A, B, C, D

    for i in range(64):
        if i < 16:
            f = ( b & c)|((~b) & d)
            flag  = i
        elif i < 32:
            f = (b & d)|(c & (~d))
            flag = (5 * i +1) %16
        elif i < 48:
            f = (b ^ c ^ d)
            flag  = (3 * i + 5)% 16
        else:
            f = c ^(b |(~d))
            flag  = (7 * i ) % 16
        tmp = b + lrot((a + f + k[i] + w[flag])& 0xffffffff,r[i])
        a, b, c, d = d, tmp & 0xffffffff, b, c
        #print(hex(a).replace("0x","").replace("L",""),
hex(b).replace("0x","").replace("L","") ,
hex(c).replace("0x","").replace("L",""),
hex(d).replace("0x","").replace("L",""))
    A = (A + a) & 0xffffffff
    B = (B + b) & 0xffffffff
    C = (C + c) & 0xffffffff
    D = (D + d) & 0xffffffff



def digest():
```

```
    global A
    global B
    global C
    global D
    return struct.pack('<IIII',A,B,C,D)

def hex_digest():
    return binascii.hexlify(digest()).decode()
```

7. RC4

```
# -*- coding: utf-8 -*-

import base64

def get_message():
    s = raw_input("What would you like to encrypt?\n")
    return s

def get_key():
    key = raw_input("What is your encryption key?\n")
    if key == '':
        key = 'none_public_key'
    return key

def init_box(key):
    """
    S 盒
    """
    s_box = list(range(256)) #我这里没管秘钥小于 256 的情况，小于 256 应该不断重复填充即可
    j = 0
    for i in range(256):
        j = (j + s_box[i] + ord(key[i % len(key)])) % 256
        s_box[i], s_box[j] = s_box[j], s_box[i]
    #print(type(s_box)) #for_test
    return s_box

def ex_encrypt(plain,box,mode):
    """
    利用 PRGA 生成秘钥流并与密文字节异或，加解密同一个算法
    """
```

```
    if mode == '2':
        while True:
            c_mode = raw_input("输入你的解密模式:Base64 or ordinary\n")
            if c_mode == 'Base64':
                plain = base64.b64decode(plain)
                plain = bytes.decode(plain)
                break
            elif c_mode == 'ordinary':
                plain = plain
                break
            else:
                print("Something Wrong,请重新新输入")
                continue

    res = []
    i = j =0
    for s in plain:
        i = (i + 1) %256
        j = (j + box[i]) %256
        box[i], box[j] = box[j], box[i]
        t = (box[i] + box[j])% 256
        k = box[t]
        res.append(chr(ord(s)^k))

    cipher = "".join(res)
    #print(cipher)
    if  mode == '1':
        # 化成可视字符需要编码

        print(base64.b64encode(cipher))

    if mode == '2':

        print(cipher)
```

8. A5

```
# -*- coding: utf-8 -*-

X = ''
Y = ''
```

```python
Z = ''

import base64
import re


def str2bin(str_mess):
    res = ""
    for i in str_mess:
        tmp = bin(ord(i))[2:].zfill(8)
        res += tmp
    return res



def bin2str(bin_mess):
    res = ""
    tmp = re.findall(r'.{8}',bin_mess)
    for i in tmp:
        res += chr(int(i,2))
    return res


def LFSRinit(): #用 64bit 密钥初始 3 个移位寄存器,分别是 19,22,23 位
    global X
    global Y
    global Z
    key = raw_input('What is your encryption key?\n')
    while len(key) != 8:   #限定只能是 8 位，然后生成 64 位的二进制流
        key = raw_input('What is your encryption key?\n')
    key_bin_str = ""
    for i in key:
        tmp = bin(ord(i))[2:].zfill(8)
        key_bin_str += tmp
    X = key_bin_str[0:19]
    Y = key_bin_str[19:41]
    Z = key_bin_str[41:]
    # print("X"+X)
    # print("Y"+Y)
    # print("Z"+Z)



def xor(bin_str,bin_key): #输入的字符串的二进制流
    res = ""
```

```
    for i in range(len(bin_str)):
        if bin_str[i] == bin_key[i]:
            res += '0'
        else:
            res += '1'
    return res


def create_key():
    global X
    global Y
    global Z
    LFSRinit()
    res = ""
    for i in range(114):   # A5 / 1 用于为每个突发产生 114 比特的密钥流序列
        # a = X[-1]
        # b = Y[-1]
        # c = Z[-1]
        g  = int(X[-1]) ^ int(Y[-1]) ^ int(Z[-1])
        res += str(g)  #用最后一位异或产生秘钥流
        x =  str(int(X[13]) ^  int(X[16]) ^ int(X[17]) ^ int(X[18]) ^
1)              #候选位的值
        y =  str(int(Y[20]) ^ int(Y[21]) ^ 1)
        z = str(int(Z[7]) ^ int(Z[20]) ^ int(Z[21]) ^ int(Z[22]) ^
1)
        #选择的钟控位
        c_x = int(X[8])
        c_y = int(Y[10])
        c_z = int(Z[10])
        if (c_x + c_y + c_z) >= 2:#多数的占优
            choice = '1'
        else:
            choice = '0'
        if str(c_x) == choice:
            X = x + X[:-1] #隐式位移，这里是不包含最后一位的
        if str(c_y) == choice:
            Y = y + Y[:-1]
        if str(c_z) == choice:
            Z = z + Z[:-1]
        # print("X"+X)
        # print("Y"+Y)
        # print("Z"+Z)
    # print(res)
    return res
```

```
def a5_encode(mess):
    bin_mess = str2bin(mess)
    bin_key = create_key()
    bin_cipher = ""
    #print(len(bin_mess))
    if len(bin_mess) % 114 == 0:
        for i in range(0, len(bin_mess), 114):
            bin_cipher += xor(bin_mess, bin_key)
    elif len(bin_mess) > 114:
        j = 0
        for i in range(len(bin_mess)):
            bin_cipher += str(int(bin_mess[i]) ^ int(bin_key[i]))
            j += 1
            if j == 114:
                j = 0
    else:
        for i in range(len(bin_mess)):
            bin_cipher += str(int(bin_mess[i]) ^ int(bin_key[i]))
    print(hex(int(bin_cipher,2)))
    str_cipher = bin2str(bin_cipher)




def a5_decode(bin_mess):
    bin_key = create_key()
    bin_cipher = ""
    # print(len(bin_mess))
    if len(bin_mess) % 114 == 0:
        for i in range(0, len(bin_mess), 114):
            bin_cipher += xor(bin_mess, bin_key)
    elif len(bin_mess) > 114:
        j = 0
        for i in range(len(bin_mess)):
            bin_cipher += str(int(bin_mess[i]) ^ int(bin_key[i]))
            j += 1
            if j == 114:
                j = 0
    else:
        for i in range(len(bin_mess)):
            bin_cipher += str(int(bin_mess[i]) ^ int(bin_key[i]))
    str_cipher = bin2str(bin_cipher)
```

```
    print("解密后的结果:"+str_cipher)
```