信息安全技术第二次作业

李钰

19335112

作业内容

学习一个能够实现典型分组密码，如DES和AES的密码软件库（任何编程语言皆可），简单介绍它的功能，以及它在分组密码的工作模式和填充模式上的设置方法，以AES加密为例，给出实现不同工作模式和填充模式的加解密源代码。

Python下密码软件库——Crypto算法库

【功能】

常见对称密码在 Crypto.Cipher 库下，主要有：DES 3DES AES RC4 Salsa20

对称密码AEC

```
from Crypto.Cipher import AES
import base64
key = bytes('this_is_a_key'.ljust(16,' '),encoding='utf8')
aes = AES.new(key,AES.MODE_ECB)
# encrypt
plain_text = bytes('this_is_a_plain'.ljust(16,' '),encoding='utf8')
text_enc = aes.encrypt(plain_text)
text_enc_b64 = base64.b64encode(text_enc)
print(text_enc_b64.decode(encoding='utf8'))
# decrypt
msg_enc = base64.b64decode(text_enc_b64)
msg = aes.decrypt(msg_enc)
print(msg.decode(encoding='utf8'))
```

对称密码DES

```
from Crypto.Cipher import DES
import base64
key = bytes('test_key'.ljust(8,' '),encoding='utf8')
des = DES.new(key,DES.MODE_ECB)
# encrypt
plain_text = bytes('this_is_a_plain'.ljust(16,' '),encoding='utf8')
text_enc = des.encrypt(plain_text)
text_enc_b64 = base64.b64encode(text_enc)
print(text_enc_b64.decode(encoding='utf8'))
# decrypt
msg_enc = base64.b64decode(text_enc_b64)
msg = des.decrypt(msg_enc)
print(msg.decode(encoding='utf8'))
```

非对称密码在 Crypto.PublicKey 库下，主要有：RSA ECC DSA
哈希密码在 Crypto.Hash 库下，常用的有：MD5 SHA-1 SHA-128 SHA-256
随机数在 Crypto.Random 库下
实用小工具在 Crypto.Util 库下，常用到 Util 中的pad()函数来填充密钥

数字签名在 Crypto.Signature 库下

【填充模式】

AES采用分组加密的方式，即将明文拆分为一个个独立的明文块，每个明文块长度为128bit。但是如果一个明文的长度在拆分时剩下的一部分不足128bit，这时就需要对铭文块进行填充。下面列出几种典型的填充方式。

1. **NoPadding**：不做任何填充，但是要求明文必须是16字节的整数倍。
2. **PKCS5Padding（默认）：** 如果明文块少于16个字节（128bit），在明文块末尾补足相应数量的字符，且每个字节的值等于缺少的字符数。 比如明文：{1,2,3,4,5,a,b,c,d,e},缺少6个字节，则补全为{1,2,3,4,5,a,b,c,d,e,6,6,6,6,6,6 }

```
def pad(self, text):
    """
    #填充函数，填充缺少字节数，使被加密数据的字节码长度是block_size的整数倍
    """
    count = len(text.encode('utf-8'))
    add = self.length - (count % self.length)
    entext = text + (chr(add) * add)
    return entext
```

3. **ISO10126Padding:** 如果明文块少于16个字节（128bit），在明文块末尾补足相应数量的字节，最后一个字符值等于缺少的字符数，其他字符填充随机数。比如明文：{1,2,3,4,5,a,b,c,d,e},缺少6个字节，则可能补全为{1,2,3,4,5,a,b,c,d,e,5,c,3,G,$,6}

```
#ISO10126Padding 填充
#在明文块末尾补足相应数量的字节，最后一个字符值等于缺少的字符数，其他字符填充随机数
def pad(text):
    byte=16-len(text)%16
    #print(byte)
    return(text+get_Random_String(byte-1)+chr(byte))
```

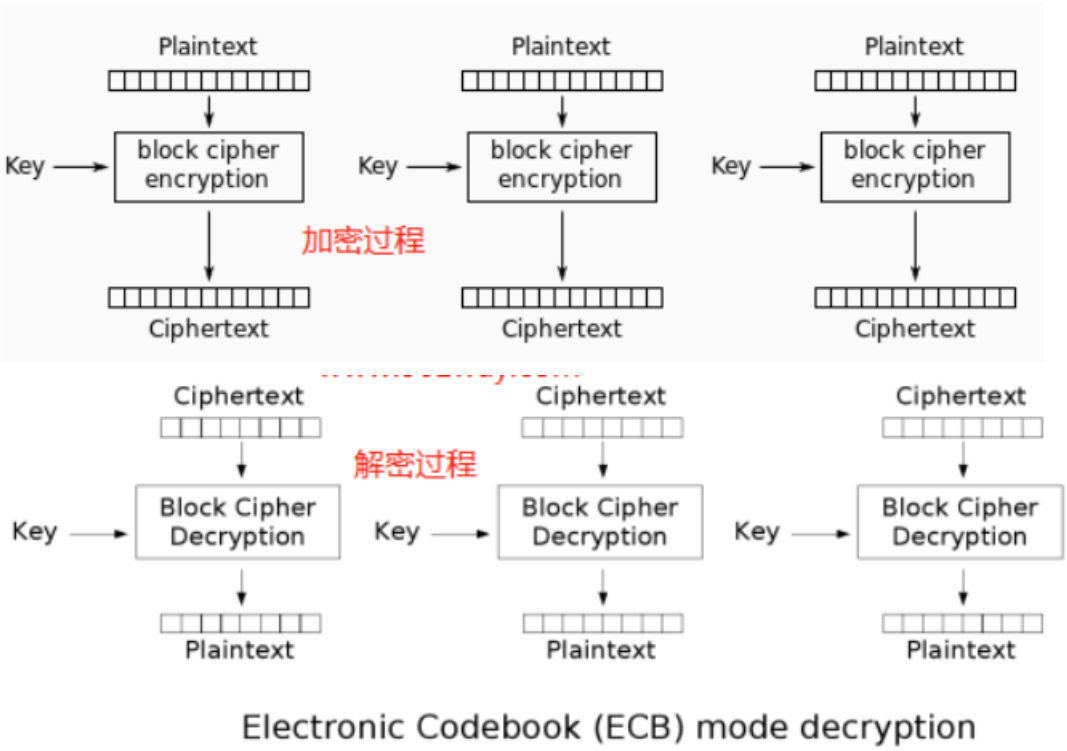
4. **PKCS7Padding:** 原理与PKCS5Padding相似，区别是PKCS5Padding的blocksize为8字节，而PKCS7Padding的blocksize可以为1到255字节

```
def __pad(self, text):
"""填充方式，加密内容必须为16字节的倍数，若不足则使用self.iv进行填充"""
text_length = len(text)
amount_to_pad = AES.block_size - (text_length % AES.block_size)
if amount_to_pad == 0:
amount_to_pad = AES.block_size
pad = chr(amount_to_pad)
return text + pad * amount_to_pad

def __unpad(self, text):
pad = ord(text[-1])
return text[:-pad]
```

【工作模式】

AES的的工作模式，体现在把明文块加密成密文块的处理过程中。AES加密算法提供了五种不同的工作模式：CBC,ECB,CTR,CFB,OFB

1. **ECB模式**：ECB是最简单的块密码加密模式，加密前根据加密块大小（如AES为128位）分成若干块，之后将**每块使用相同的密钥单独加密**，解密同理。具体见下图：

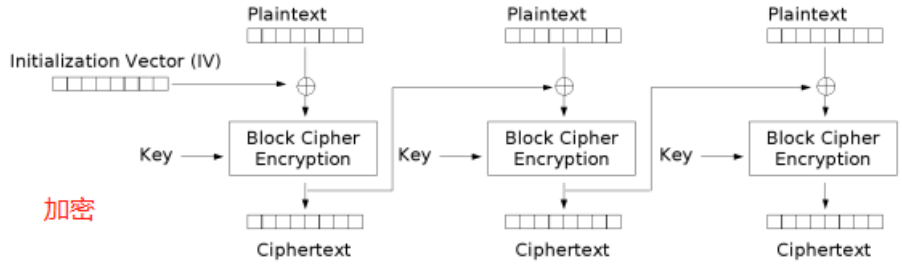


Electronic Codebook (ECB) mode decryption

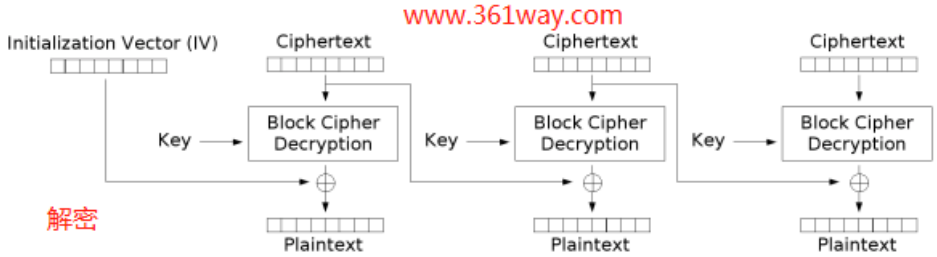
ECB模式由于每块数据的加密是独立的因此加密和解密都可以并行计算，ECB模式最大的缺点是相同的明文块会被加密成相同的密文块，这种方法在某些环境下不能提供严格的数据保密性。

```
from Cryptodome.Cipher import AES
from binascii import b2a_hex, a2b_hex

key = 'abcdefgh'    # 秘钥,此处需要将字符串转为字节
def pad(text):    # 加密内容需要长达16位字符，所以进行空格拼接
    while len(text) % 16 != 0:
        text += ' '
    return text

aes = AES.new(key.encode(), AES.MODE_ECB)  # 进行加密算法，模式ECB模式

text = 'hello'    # 加密内容,此处需要将字符串转为字节
#加密
encrypted_text = aes.encrypt(pad(text).encode())  # 进行内容拼接16位字符后传入加密类中，结果为字节类型
encrypted_text_hex = b2a_hex(encrypted_text)
print(encrypted_text_hex)
#用aes对象进行解密，将字节类型转为str类型，错误编码忽略不计
de = str(aes.decrypt(a2b_hex(encrypted_text_hex)), encoding='utf-8',errors="ignore")
print(de[:len(text)])    # 获取str从0开始到文本内容的字符串长度。
```

2. **CBC模式**：CBC模式对于每个待加密的密码块在加密前会先与前一个密码块的密文异或然后再用加密器加密。第一个明文块与一个叫初始化向量的数据块异或。如下图：



Cipher Block Chaining (CBC) mode encryption

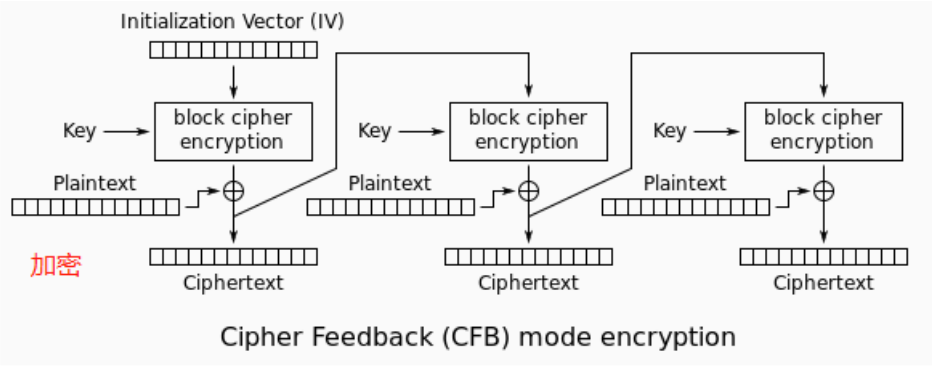


Cipher Block Chaining (CBC) mode decryption

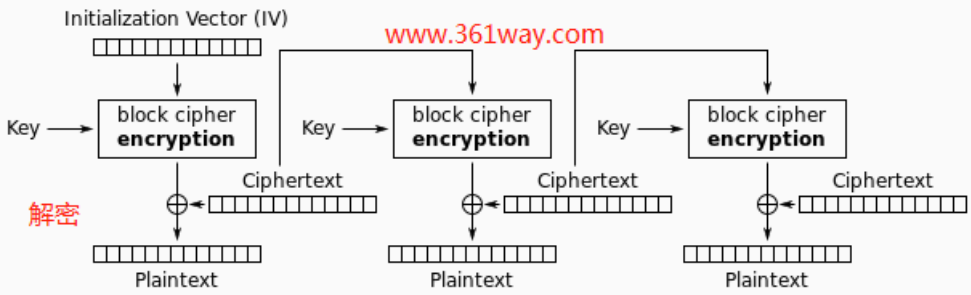
CBC模式允许length不是16(128位)的整数倍，不足的部分会用0填充，输出总是16的整数倍。完成加密或解密后会更新初始化向量IV。CBC模式相比ECB有更高的保密性，但由于对每个数据块的加密依赖与前一个数据块的加密所以加密无法并行。与ECB一样在加密前需要对数据进行填充，不是很适合对流数据进行加密。

```
#示例：
from Crypto.Cipher import AES
from binascii import b2a_hex, a2b_hex

def pad(text):
    """加密文本text必须为16的倍数!
    """
    while len(text) % 16 != 0:
        text += '\0'.encode('utf-8')    # \0 可以被decode()自动清除，并且不会影响本来的字符0
    return text
#加密
def encrypt(text):
    cryptor = AES.new(key.encode('utf-8')), AES.MODE_CBC, key.encode('utf-8'))    # 此变量是一次性的(第二次调用值会变)不能作为常量通用
    ciphertext = cryptor.encrypt(pad(text.encode('utf-8')))    # encode()转换是因为十六进制用的是字节码
    return b2a_hex(ciphertext).decode('utf-8')    # 因为AES加密时候得到的字符串不一定是ascii字符集的，所以使用十六进制转换才能print来储存
#解密
def decrypt(text):
    cryptor = AES.new(key.encode('utf-8'), AES.MODE_CBC, key.encode('utf-8'))
    plain_text = cryptor.decrypt(a2b_hex(text.encode('utf-8')))
    return plain_text.decode('utf-8').rstrip('\0')    # 去除凑数补全的\0
```

3. **CFB模式**：与ECB和CBC模式只能够加密块数据不同，CFB能够将块密文（Block Cipher）转换为流密文（Stream Cipher）。见下图：



Cipher Feedback (CFB) mode encryption



Cipher Feedback (CFB) mode decryption

CFB的加密工作分为两部分：1、将一前段加密得到的密文再加密；2、将第1步加密得到的数据与当前段的明文异或。

由于加密流程和解密流程中被块加密器加密的数据是前一段密文，因此即使明文数据的长度不是加密块大小的整数倍也是不需要填充的，这保证了数据长度在加密前后是相同的。以128位为例，128位的CFB是对前一段数据的密文用块加密器加密后保存在IV中，之后用这128位数据与后面到来的128位数据异或，num用来记录自上次调用加密器后已经处理的数据长度（字节），当num重新变为0的时候就会再调用加密器，即每处理128位调用一次加密器。

```
def AES_128_CFB(String):
    cryptor = AES.new(key=key, mode=AES.MODE_CFB, IV=iv,segment_size=128)
    ciphertext = cryptor.encrypt(String)
    return   base64.b64encode(ciphertext)

def AES_128_CFB_decode(String):
    decode = base64.b64decode(String)
    cryptor = AES.new(key=key, mode=AES.MODE_CFB, IV=iv,segment_size=128)
    plain_text = cryptor.decrypt(decode)
    return plain_text

def get_encode_data(message):
    cryptor = AES.new(key=key, mode=AES.MODE_CFB, IV=iv, segment_size=128)
    ciphertext = cryptor.encrypt(message)
    return base64.b64encode(ciphertext)
```

4. **3.计算器模式（Counter (CTR)）**

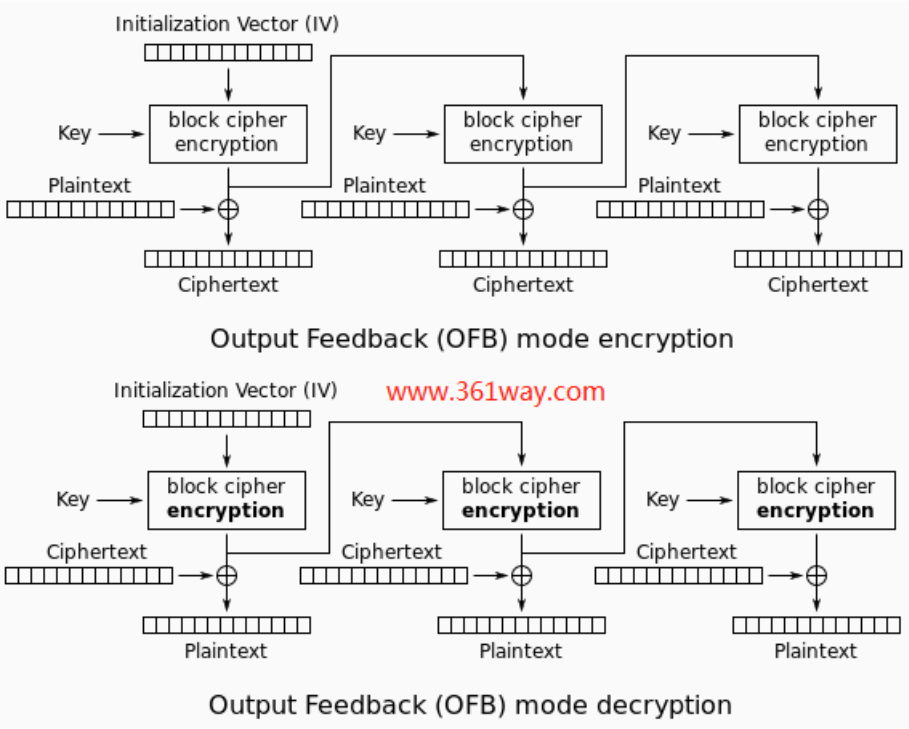
此模式不常见， 使用一个自增的算子，这个算子用加密之后的输出和明文异或的结果得到密文，相当于一次一密。这种加密方式简单快速，安全可靠，而且可以并行加密，但是在计算器不能维持很长的情况下，只能使用一次。

```
def _get_timers(self,iv, msgLen):
    #iv: 计时器初值
    #msgLen: 密文长度(明文)
    blockSZ = self.block_size
    blocks = int((msgLen + blockSZ - 1) //blockSZ)
```

```
    timer = int_from_bytes(iv)
    timers = iv
    for i in range(1, blocks):
    timer += 1
    timers += int_to_bytes(timer)
    return timers

def encrypt(self, plainText, count):
    count= bytes(count)
    #各个计数器值
    counters= self._get_timers(count,len(plainText))
    blocks = xor_block(self._cipher.encrypt(counters), plainText)
    ciphertext = bytes(blocks)
    return count+ciphertext[:len(plainText)]
```

```
def decrypt(self, cipherText):
    blockSZ = self.block_size
    # 加密和解密只有输入不同
    pt = self.encrypt(cipherText[blockSZ:], cipherText[:blockSZ])
    return pt[blockSZ:]
```

5. **OFB模式**：OFB是先用块加密器生成密钥流（Keystream），然后再将密钥流与明文流异或得到密文流，解密是先用块加密器生成密钥流，再将密钥流与密文流异或得到明文，由于异或操作的对称性所以加密和解密的流程是完全一样的。




```
from Crypto.Cipher import AES
from Crypto.Random import get_random_bytes

plainText = b"Lorem ipsum dolor sit amet, consectetur adipiscing e"
key = b"ANAAREMEREAAAAAA"

def ofbEnc(plainText, key):
    pos = 0
```

```
    cipherTextChunks = []
    iv = get_random_bytes(16)
    originalIV = iv
    cipher = AES.new(key, AES.MODE_ECB)
    if len(plainText) % 16 != 0:
        plainText += b"1"
    while len(plainText) % 16 != 0:
        plainText += b"0"
    while pos + 16 <= len(plainText):
        toXor = cipher.encrypt(iv)
        nextPos = pos + 16
        toEnc = plainText[pos:nextPos]
        cipherText = bytes([toXor[i] ^ toEnc[i] for i in range(16)])
        cipherTextChunks.append(cipherText)
        pos += 16
        iv = toXor
    return (originalIV, cipherTextChunks)


def ofbDec(cipherTextChunks, key, iv):
    plainText = b""
    cipher = AES.new(key, AES.MODE_ECB)
    for chunk in cipherTextChunks:
        toXor = cipher.encrypt(iv)
        plainText += bytes([toXor[i] ^ chunk[i] for i in range(15)])
        iv = toXor
    while plainText[-1] == 48:
        plainText = plainText[0:-1]
    if plainText[-1] == 49:
        plainText = plainText[0:-1]
    return plainText


iv, result = ofbEnc(plainText, key)
print(iv, result)

plain = ofbDec(result, key, iv)
print(plain)
```