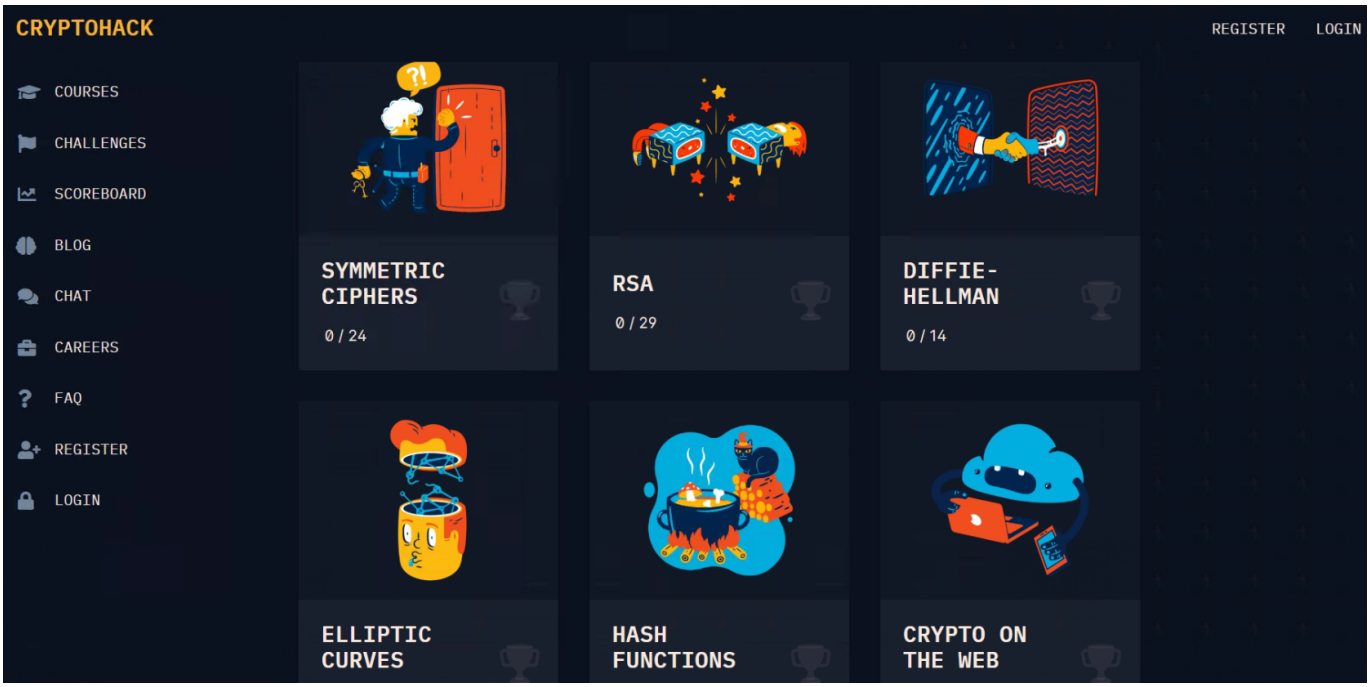
Crypto 专题1

- 对称加密
  - AES实现
  - 分组模式
- 非对称加密
  - RSA攻击
- 格基规约
  - 格基规约
  - Learning With Errors
  - RSA Coppersmith 攻击
- 作业概述

写在前面：CryptoHack是个比较好的密码学习题练习平台，会有基础练习题帮助了解基础知识，也有较高难度的挑战题



而且AAA传奇选手4老师全球排名第11😍 😍 😍

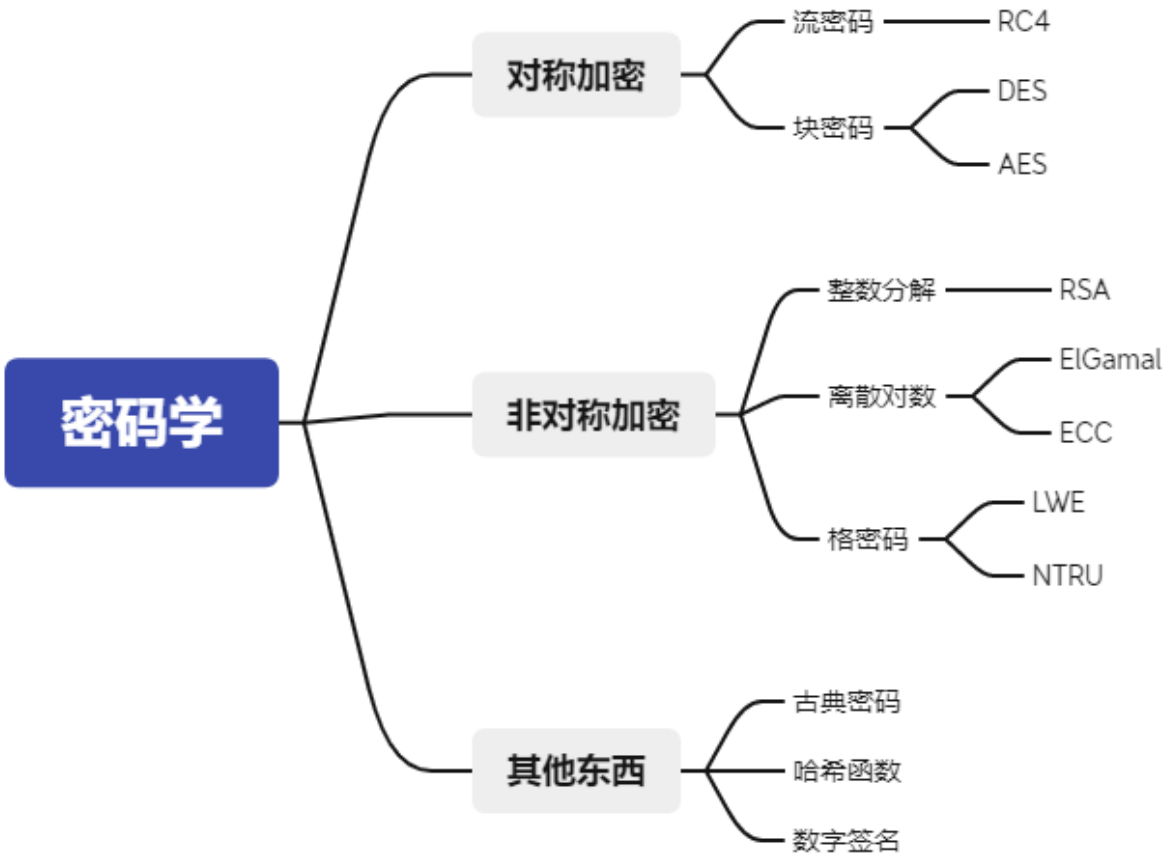
还有，本次部分题目是CryptoHack上的原题，可以注册账号直接在这上面做，当然也会在作业文件中丢上一份

以及，小白老师的密码学讲义对加密/解密算法的底层实现讲解的十分透彻，可以去看看密码学讲义

首先先来简单回顾一下什么是密码学

- 明文：你看得懂的要被加密的信息
- 密文：加密后的“乱码”
- 密钥：加密用的密码

下面这张图是对密码学的部分总结



对称密码

> 回顾：对称密码就是加密和解密使用同一个密钥的加密算法

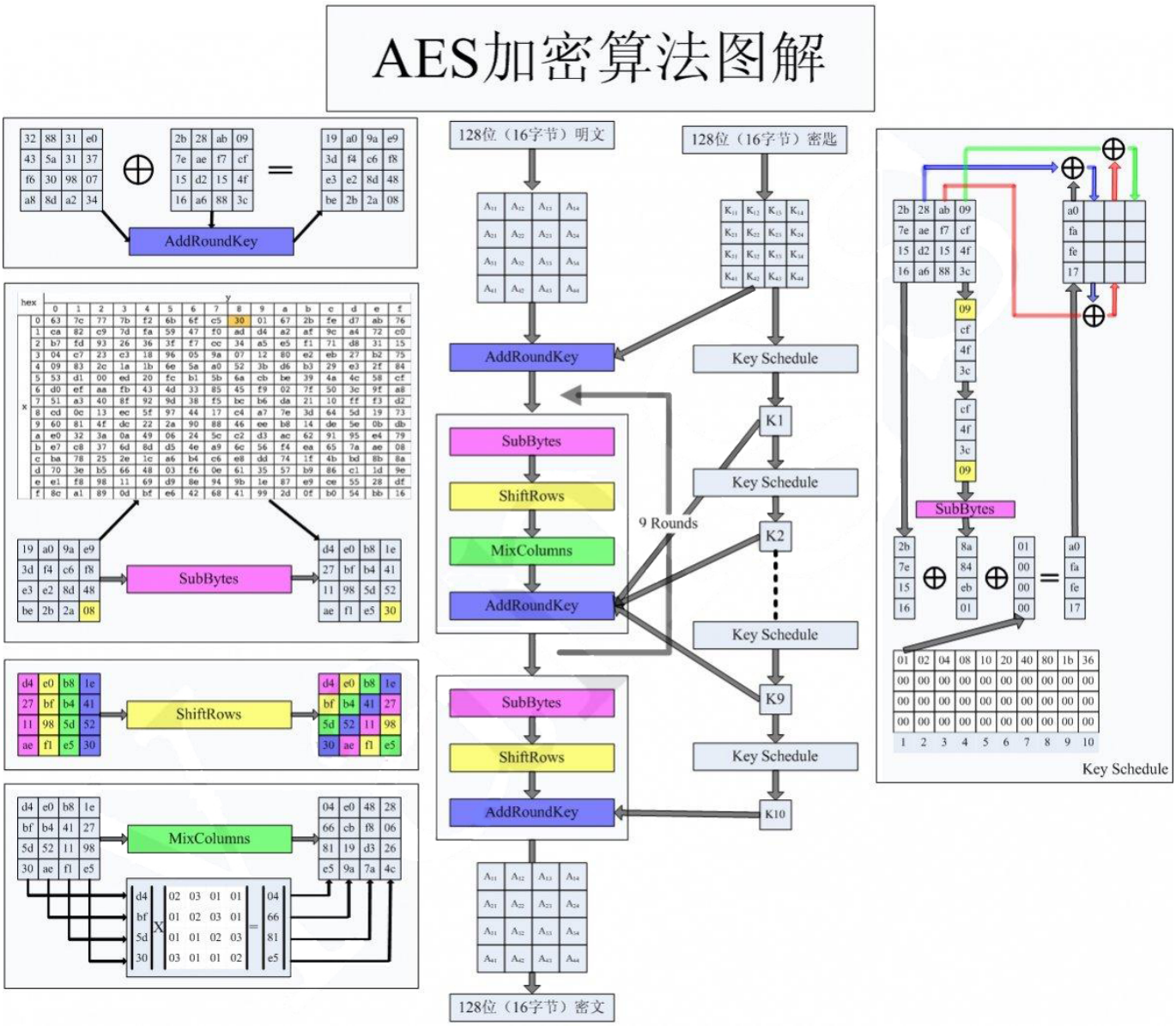
AES实现

AES的全称是高级加密标准(Advanced Encryption Standard)，是为了取代安全性较低的DES加密算法而发明的

DES是一种**分组加密**算法，**64位**构成一组，密钥为~~64位?~~虽然让你输入64位，但是只是方便起见，实际用的是**56位**，剩下8位要么直接扔掉，要么当作校验

AES也是一种**分组加密**算法，**128位**构成一组，密钥长度可以是**128/192/256位**

那么，AES长啥样呢？如图：

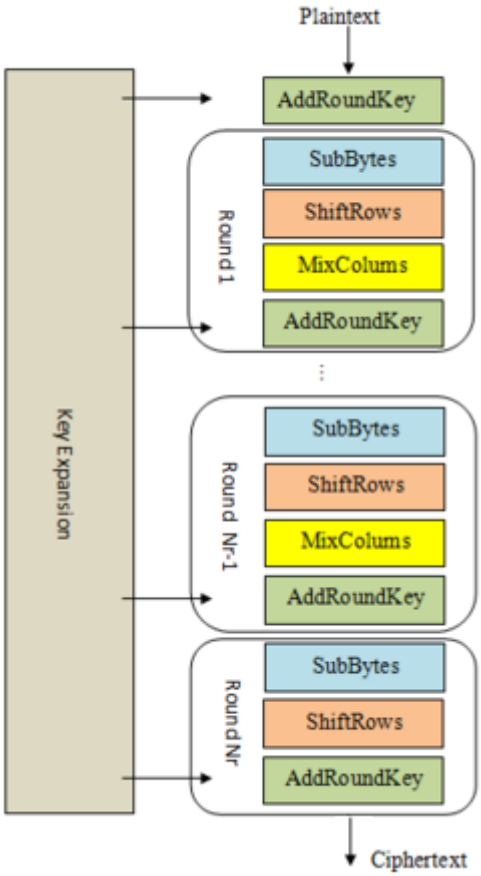


emm……好复杂~~（然而不复杂一下就被黑客们攻破了🤣）~~

所以，还是把这个东西全部拆解开，一步步了解AES到底是什么吧

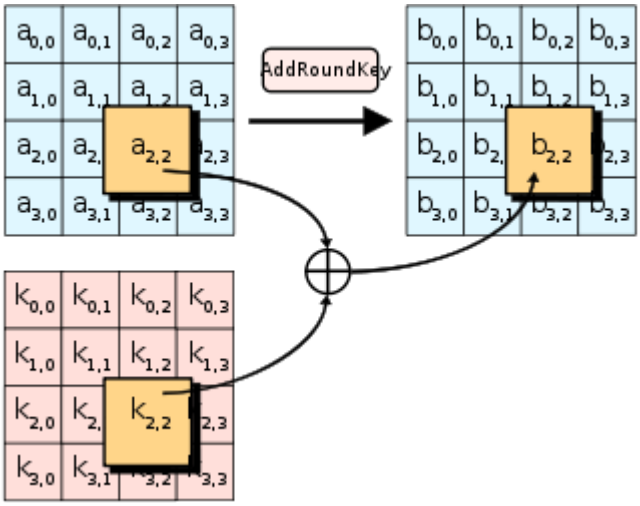
首先先是对整体的AES做一个叙述吧（以128bit密钥为例）：

- 将原来的128bit密钥变成11个128bit的密钥
- 初始先让明文和原始密钥进行一次“轮密钥加”
- 进行10次循环，每次循环进行：
  - 字节替换
  - 行移位
  - 列混淆
  - 轮密钥加

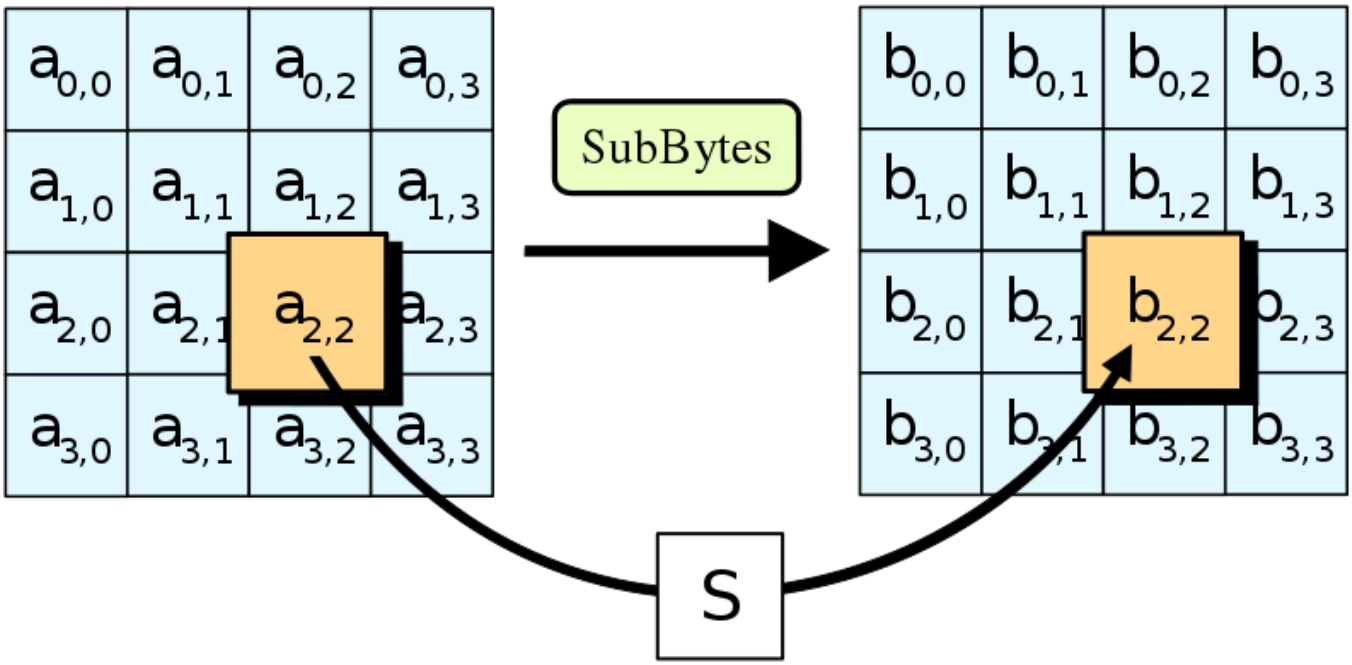


接下来简要讲一下各个板块的具体操作：

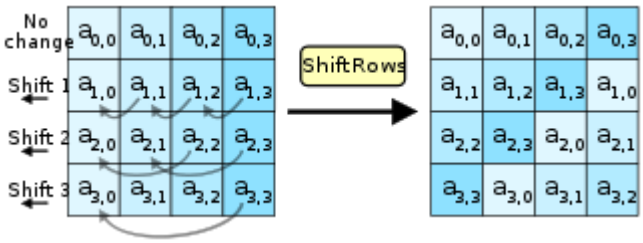
AddRoundKey 轮密钥加：对于两个16字节的数据，将每个字节作异或操作即可



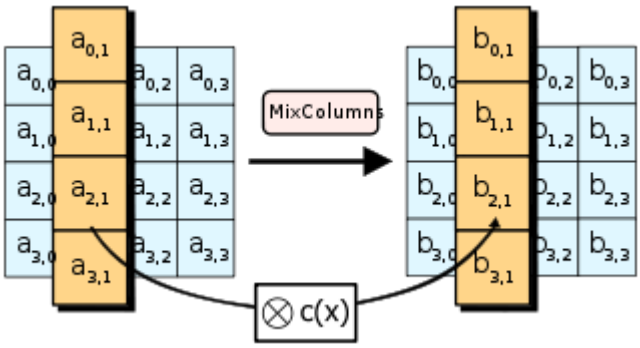
SubBytes 字节替换：通过非线性sbox，将目前16字节数据替换成其他数据，原来的值作为sbox数组的下标（混淆）



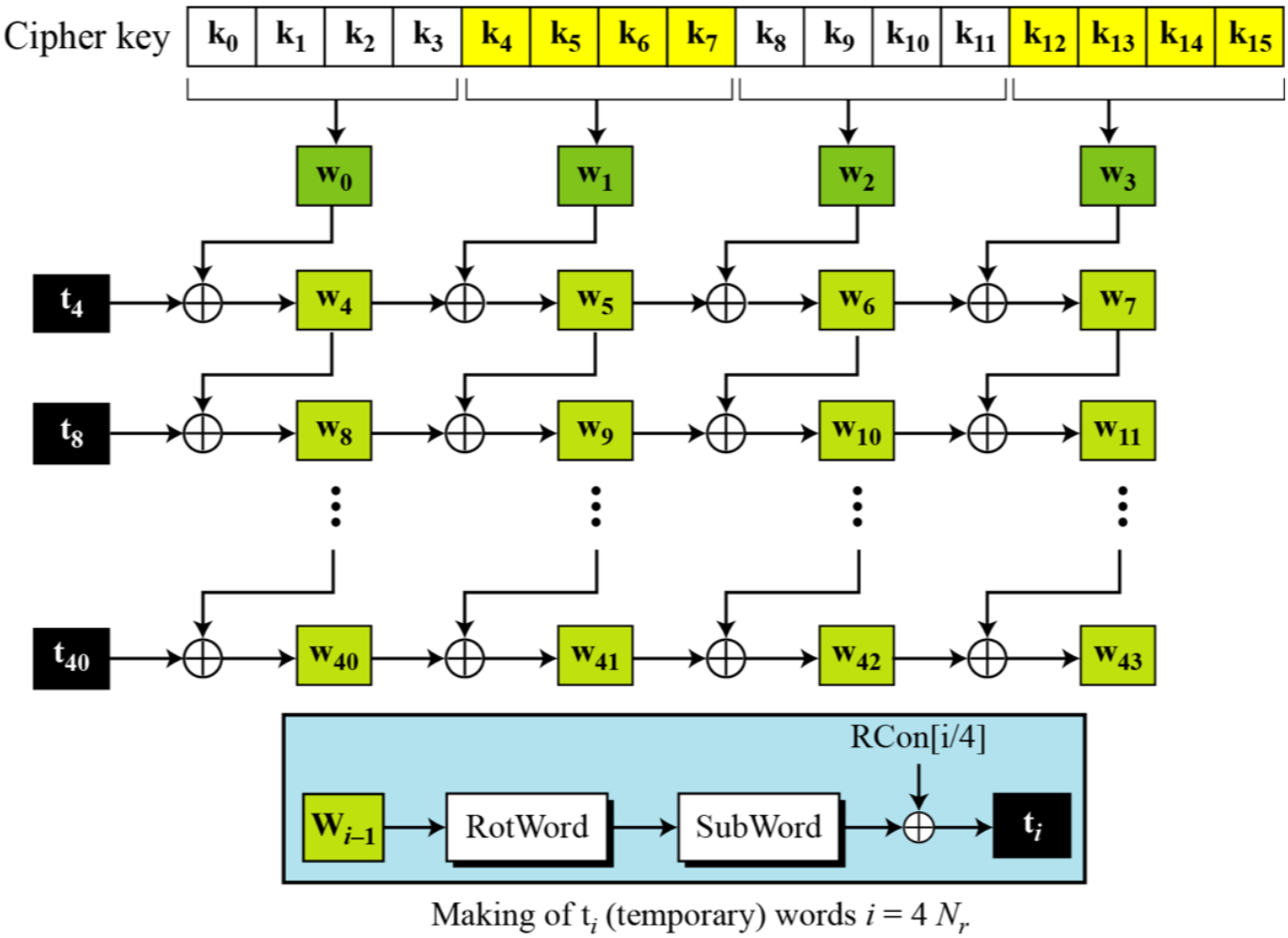
ShiftRows 行移位：对于目前的4x4矩阵，第k行（从0开始）左移k个单位（扩散）



MixColumns 列混淆：将混淆矩阵与原来的矩阵进行$GF(2^8)$上的乘法（扩散）



KeyExpansion 密钥拓展：将原密钥经过10轮操作变为11个128位密钥，其中每轮操作，上一轮结尾的4字节需经过左移，sbox和rcon异或操作，然后所有的4字节组均进行异或操作



*** 上面的加密算法，除了Sbox这一步，其他的操作全都是线性计算，这一点非常重要**

分组模式

如上所述，AES是一种分组加密的算法，其一次只能处理128位的信息加密，那么，如果你需要加密的信息远超于128位，该怎么办呢？

ECB

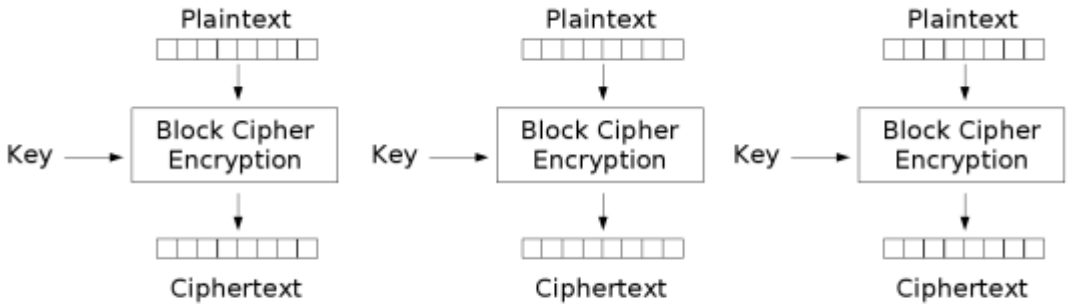
我们最先想到的肯定是，将需要加密的信息分成16字节一组，然后每一组用同一个算法来进行加密运算，这样就完成了加密

至于不满16字节的小组，随便往后面加点东西凑满16字节不就好了吗？😊

这个操作叫做padding 填充，最容易想到的便是填0，即Zero Padding

而还有一种较为常见也稍微安全一些的填充方法是PKCS7 Padding，简单来说，如果你还剩k个字节凑满16字节，那么剩下需要填充的字节值都是k，而本来就正好填满16字节的信息容易被破解，所以对于本就填满的信息，它会假设你没填满，那你就还需要填上16字节的0x10

这样的分组密码加密模式被称作**ECB (Electronic CodeBook)**，是一种简便、快速的加密算法（放隔壁超算就可以直接并行计算了）



Electronic Codebook (ECB) mode encryption

ECB Oracle

> 然而，这样的分组模式安全吗？

以下面的题为例

```python
from Crypto.Cipher import AES
from Crypto.Util.Padding import pad, unpad

KEY = ?
FLAG = ?

def encrypt(username):
    username = bytes.fromhex(username)

    padded = pad(b"Hello " + username + b" , Here is the flag: " + FLAG.encode(), 16)
    cipher = AES.new(KEY, AES.MODE_ECB)
    try:
        encrypted = cipher.encrypt(padded)
    except ValueError as e:
        print("Error: " + str(e))

    print("ciphertext: ", encrypted.hex())

while(1):
    encrypt(input())
```

让我们先简单玩一玩

```
4141414141
ciphertext:
6156faef7751cde3ccc4d3203ed487c9650ca5191edc0f550aa0d5469710c436751af52aaa08d55fa4a78e
848e918512ad548fcd829967cce3ee13f756b5749949883cd30f909605335cc3e187d54b89
414141414141
ciphertext:
3358e607c5c0393e23f35fd958765f1dc139310b153ea0d2a705652e0d10f09b9293df49a6f7a4c77efeaa
a609deefb93af7731d873debbdf5983fb1c82b17c22e7d0950d8d46406a36e59e47d2bc9d9
41414141414141
ciphertext:
0ada8d19904fb3456032550542816a50444f2172912533e3759080190b371f79f6ba177ac703fe3ab334f7
d4545728256b7e2ea97f5559c0bccc1c016c9029b6976da6b43545afe7bf1a351f58307af1
4141414141414141
ciphertext:
2d5cf6f6ed55392f80e7b7328a857264e24a93a83e05e8bae9d7b9dec5200ad4733b5f12dc1d3782ed2dbf
d5d28cb6fa05dade80cc0f1611bd4b7db85c753fe48e50c0c0d94ff62c873d8a22e24a878fbb0cfd3dac7f
5b0106a0270215ae48a5
```

由此可知，FLAG的长度是？

```
48 65 6C 6C 6F 20 41 41 41 41 41 41 41 20 2C 20 | Hello AAAAAAA ,
48 65 72 65 20 69 73 20 74 68 65 20 66 6C 61 67 | Here is the flag
3A 20 78 78 78 78 78 78 78 78 78 78 78 78 78 78 | : xxxxxxxxxxxxxx
78 78 78 78 78 78 78 78 78 78 78 78 78 78 78 78 | xxxxxxxxxxxxxxx
78 78 78 78 78 78 78 78 78 78 78 78 78 78 78 01 | xxxxxxxxxxxxxx\x01
```

在知道长度后，有什么神奇的操作可以使得flag被泄露出来呢?

48 65 6C 6C 6F 20 41 41 41 41 20 2C 20 48 65 72 | Hello AAAA , Her 65 20 69 73 20 74 68 65 20 66 6C 61 67 3A 20 78 | e is the flag: x ……

然后，可不可以，我的名字其实长这样(•`ω •´)✧

48 65 6C 6C 6F 20 41 41 41 41 20 2C 20 48 65 72 | Hello AAAA , Her

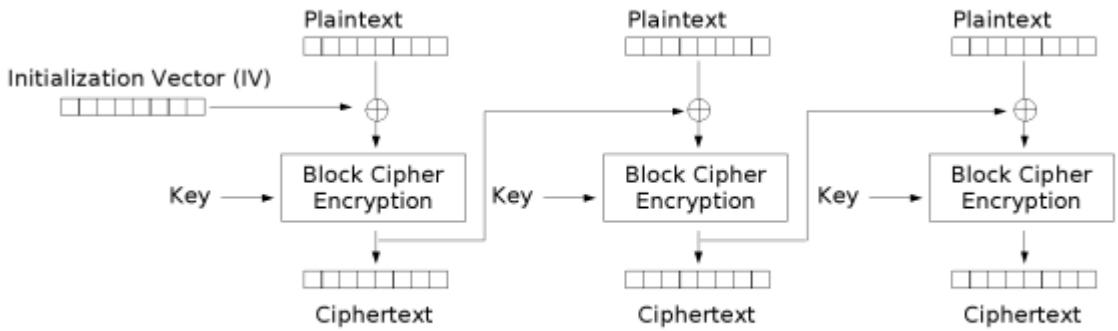
65 20 69 73 20 74 68 65 20 66 6C 61 67 3A 20 78 | e is the flag: x

……

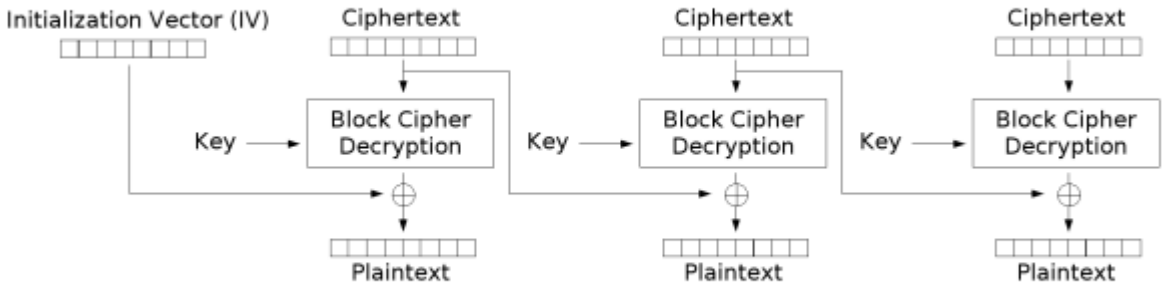
那现在，只需要将用户名的最后一位遍历一遍，就能一位一位的爆破出flag的具体内容

CBC

CBC即Cipher Block Chaining，用一种类似于链表的方式串起所有的分组，使得当前分组的改变会影响后续分组的加解密



Cipher Block Chaining (CBC) mode encryption



Cipher Block Chaining (CBC) mode decryption

这很好的避免了相同内容的块被加密成相同密文而导致的统计学攻击，然而，CBC还是有一些问题的

CBC Byte Flip

```python
from Crypto.Cipher import AES
from Crypto.Util.Padding import pad, unpad
from os import urandom

KEY = urandom(16)
FLAG = ?

OriginString = "Welcome to SECURE Crypto System\nNo way to hack😜"
print(OriginString)
assert len(OriginString.encode()) == 50
HackString = "Your Crypto System is HACKED BY AAA🤣🤣🤣"
print(HackString)
assert len(HackString.encode()) == 47

def encrypt(enc_str = OriginString):
    enc_str = enc_str.encode()
    padded = pad(enc_str, 16)
    IV = urandom(16)
    cipher = AES.new(KEY, AES.MODE_CBC, IV)
    try:
        encrypted = IV + cipher.encrypt(padded)
    except ValueError as e:
        print("Error: " + str(e))

    print("ciphertext: ", encrypted.hex())

def decrypt(dec_str):
    dec_str = bytes.fromhex(dec_str)
    IV = dec_str[:16]
    dec_str = dec_str[16:]
    cipher = AES.new(KEY, AES.MODE_CBC, IV)
    try:
        padded = cipher.decrypt(dec_str)
        decrypted = unpad(padded, 16)
    except ValueError as e:
        print("Error: " + str(e))

    print("plaintext: ", decrypted.hex())
    return decrypted.decode()

encrypt()
while(1):
    if decrypt(input()) == HackString:
        print(FLAG)
```

简要分析一下

57 65 6c 63 6f 6d 65 20 74 6f 20 53 45 43 55 52 | Welcome to SECUR

45 20 43 72 79 70 74 6f 20 53 79 73 74 65 6d 0a | E Crypto System\n

4e 6f 20 77 61 79 20 74 6f 20 68 61 63 6b f0 9f | No way to hack\xf0\x9f

需要变成

59 6f 75 72 20 43 72 79 70 74 6f 20 53 79 73 74 | Your Crypto Syst

65 6d 20 69 73 20 48 41 43 4b 45 44 20 42 59 20 | em is HACKED BY

41 41 41 f0 9f a4 a3 f0 9f a4 a3 f0 9f a4 a3 01 | AAA🤣🤣🤣\x01

注意到CBC解密时每个块其实是先进行解密，再与前一块的密文异或

那么，回忆一下异或，如果原本的解密结果D1和D1异或，那结果是0，再和你需要的指定解密结果D2异或，结果就变回D2了

因此，可以得出，对于两个密文组E1+E2解密结果是D1+D2，如果E1变为E1^D2^A，则现在解密结果的第二块就变成A了

但是，现在D1也被破坏了，怎么办呢？注意到CBC的一个密文块只影响自己和下一个块的解密，因此这个操作完全可以迭代，直到第一块密文块

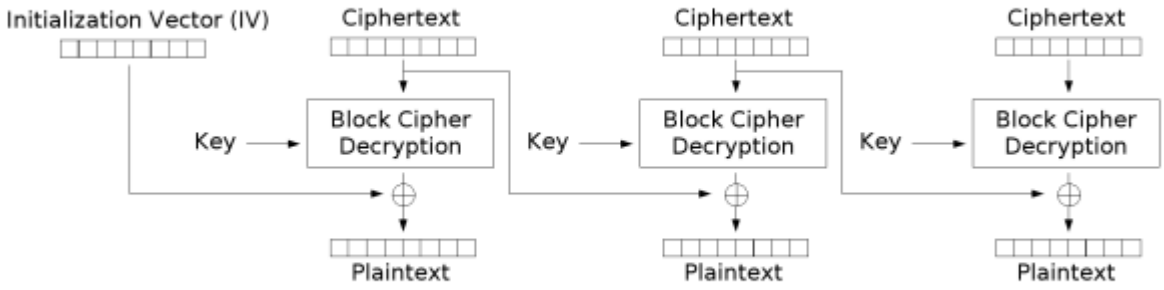
那么，第一块密文块如何控制其解密结果呢？很简单，因为你能控制IV这个量，因此你可以把IV当作第0个密文组，通过修改IV修改第一块密文块的解密结果

Padding Oracle Attack

```python
from Crypto.Cipher import AES
from Crypto.Util.Padding import pad, unpad
from os import urandom

KEY = urandom(16)
FLAG = ?
IV = urandom(16)

def pad(msg: bytes) -> bytes:
    pad_length = 16 - len(msg) % 16
    return msg + (chr(pad_length) * pad_length).encode()

def unpad(msg: bytes):
    if bytes([msg[-1]]) * msg[-1] != msg[-msg[-1]:]:
        return b"padding error!?"
    return msg[:-msg[-1]]

def encrypt(msg: bytes):
    cipher = AES.new(KEY, AES.MODE_CBC, IV)
    msg = pad(msg)
    encrypted = cipher.encrypt(msg)
    return IV + encrypted

def decrypt(msg: bytes):
    cipher = AES.new(KEY, AES.MODE_CBC, IV)
    decrypted = cipher.decrypt(msg)
    decrypted = unpad(decrypted)
    return decrypted

print(encrypt(FLAG).hex())
while(1):
    decrypted = decrypt(bytes.fromhex(input()))
    if(decrypted == b"padding error!?"):
        print(500)
    elif(decrypted == FLAG):
        print(200)
    else:
        print(403)
```

这次的加密系统看似天衣无缝，当你的输入解密为flag时返回成功请求，当输入解密不是flag时就返回错误请求，连解密的结果都不给，怎么获取flag的值呢?

这里有一个小小的漏洞可以用来泄露信息，就是当程序解密并unpad时，若unpad失败，程序会抛出不同的返回，但却不会直接终止程序



Cipher Block Chaining (CBC) mode decryption

现在，让我们再次回忆一下CBC的加密模式以及Byte Flip的攻击方式，假设我们将原先的C1放在某个块的后面，对前一块的最后一个字节进行遍历，当该字节与IV^P1的最后一个字节异或结果为0x01时，其结果一定是unpad成功，这样我们就能获取到IV^P1最后一个字节的结果了，同理，一位一位进行爆破，构造合法的unpad结果并获取到整个IV^P1的值，你便能得到P1的值了

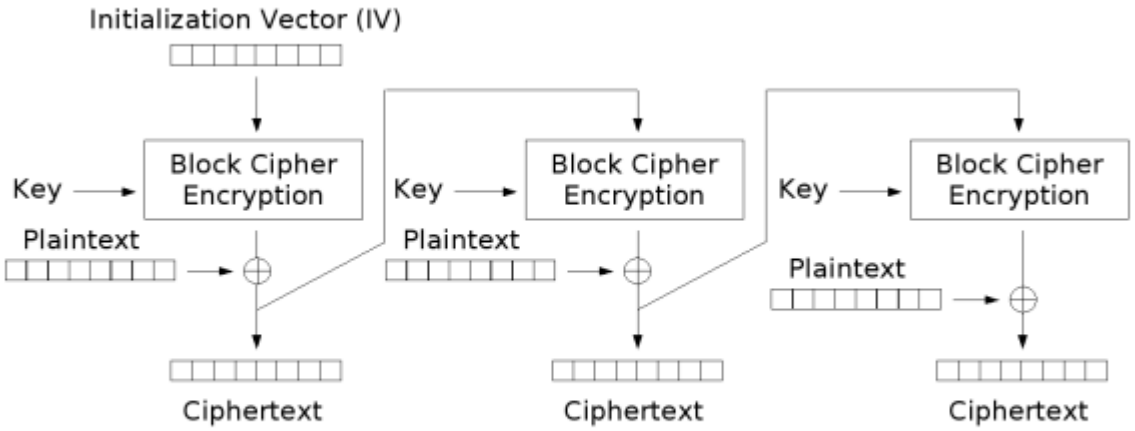
接着便是一个块一个块的进行破解，最后得到整个密文的信息

> 上面的攻击思路真的没问题吧

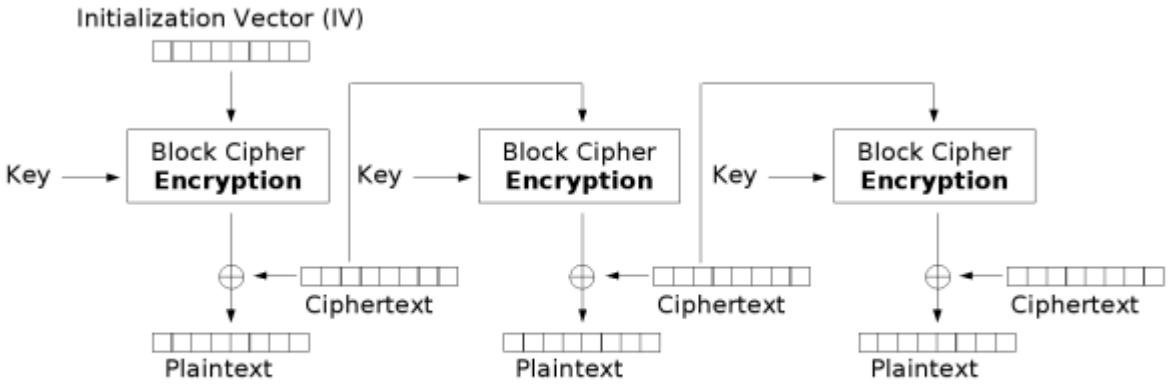
上面的攻击方法仍存在一个小小的漏洞，就是每个块爆破最后一个字节时，若解密结果刚好为xxx\x02\x02，也会被认为是pad正确，3个\x03同理，至于怎么处理这个问题，就交给同学们自己思考吧

其他分组模式

CFB (Cipher FeedBack)

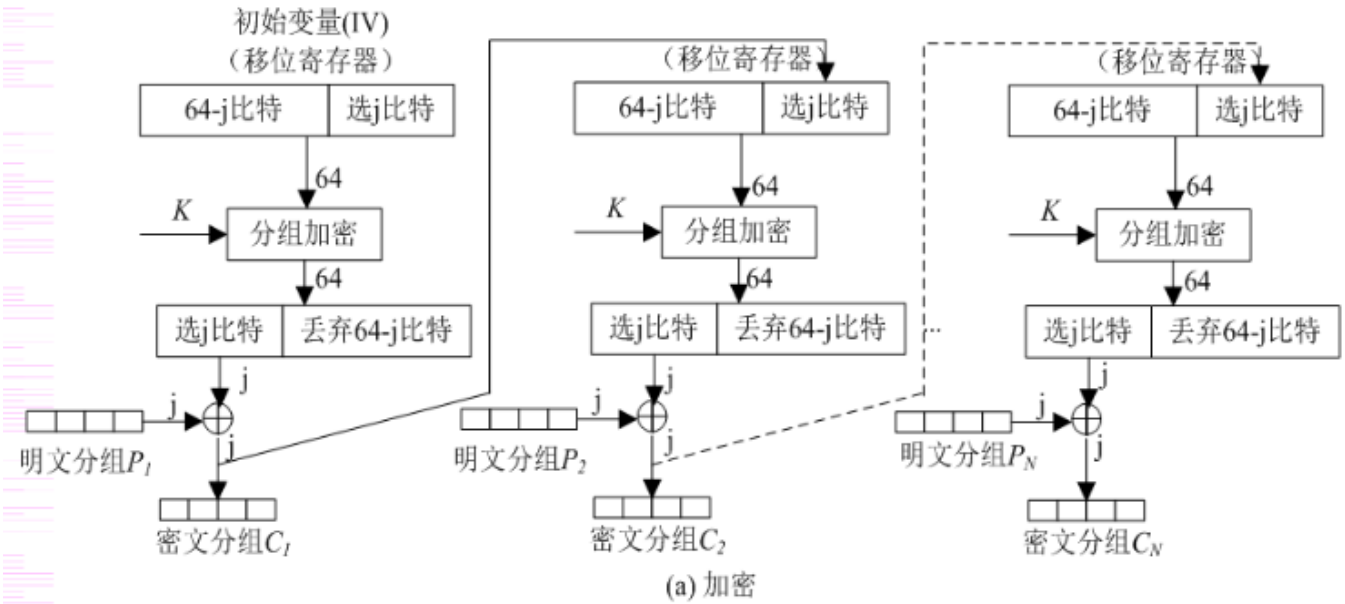


Cipher Feedback (CFB) mode encryption



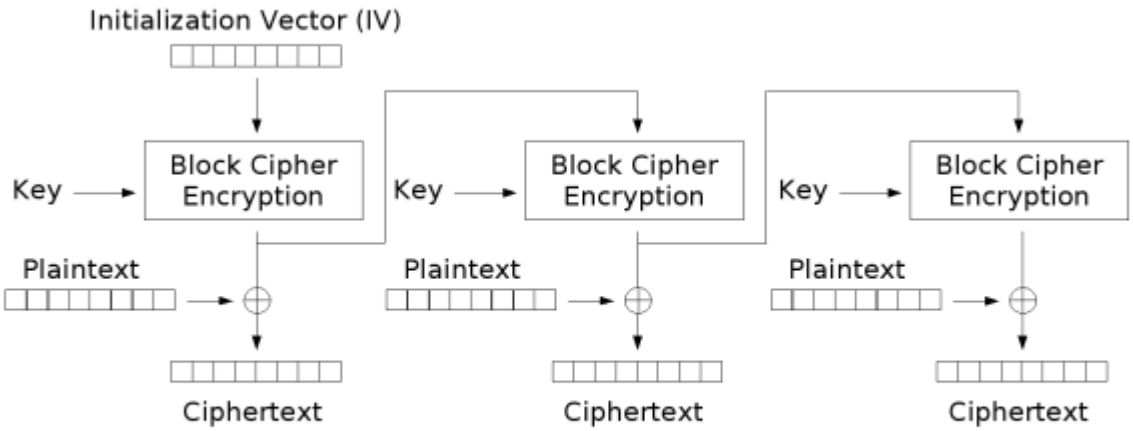
Cipher Feedback (CFB) mode decryption

不过，在实际应用中，CFB模式往往会使用下面的分组方式，以j比特为单位（j多为8），每次进行加密

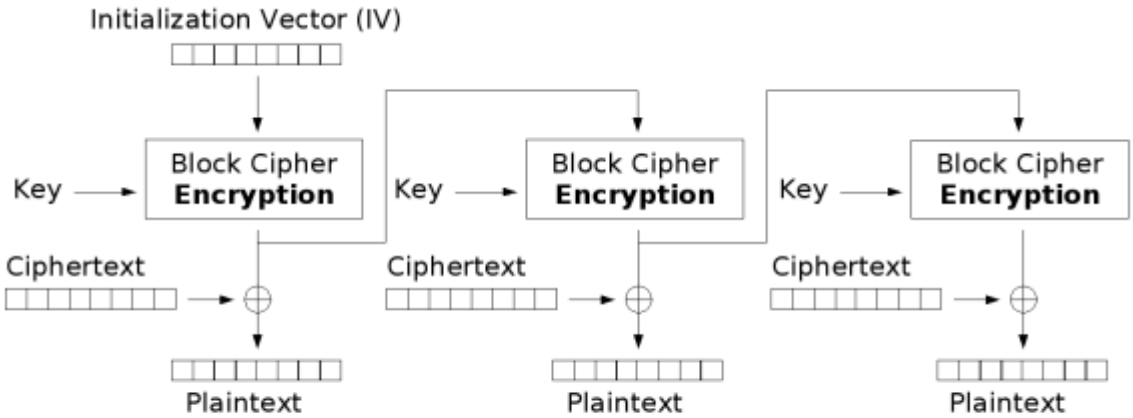


(a) 加密

OFB (Output Feedback)

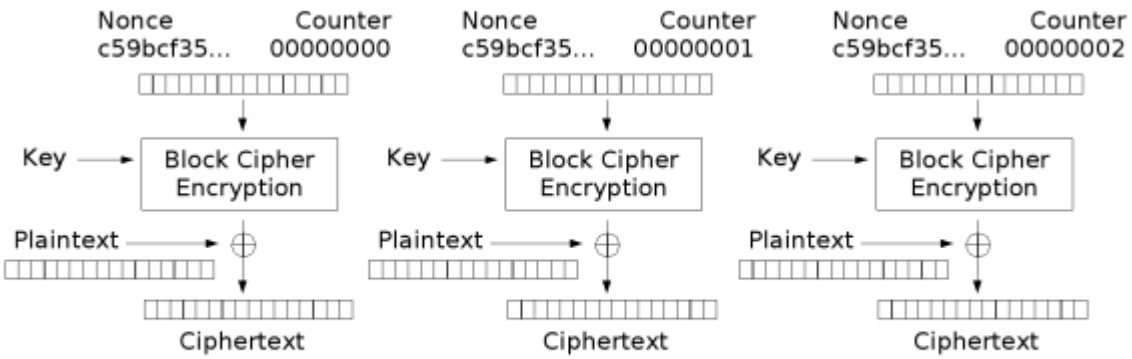


Output Feedback (OFB) mode encryption



Output Feedback (OFB) mode decryption

CTR (CounTeR)



Counter (CTR) mode encryption

上面的这些分组模式也有许多好玩的特性，也有一些安全性的缺陷，大家可以自己研究一下

非对称密码

> 回顾：非对称密码就是加密和解密使用不同密钥（公钥和私钥）的加密算法

RSA攻击

在这里首先简单回忆一下RSA的加密和解密步骤

- 选取大质数p和q
- 计算N=pq
- 计算Φ(N)=(p-1)(q-1)
- 选取一个与Φ(N)互质的e
- 加密密文m，明文$c \equiv m^e (mod\ n)$
- 计算e在模Φ(N)同余类下的逆元d
- 解密明文c，$c^d \equiv m (mod\ n)$

然而，有时候，一些特殊的参数构造会使得RSA存在一定风险

N分解攻击

某些特殊的p,q取值会使得N很容易被质因数分解

比如，当N较小时(小于256bit)可以以较短时间直接暴力分解

另外，p，q差值太大或者太小时，可以使用Fermat或Pollard rho法进行分解

在这里推荐一些工具or网址

yafu:该工具可以用于自适应整数因式分解，基本覆盖全平台，内置有Fermat、Pollard rho、SIQS、GNFS等多种算法

factordb:在线数据库，但大多数都查不到

Integer factorization calculator:可以当作是网页版的yafu，有时处理大数质数分解会比yafu还好，强烈推荐

低加密指数小明文攻击

当e极小(如e=3)，且m也较小时

$$c \equiv m^3 (mod\ n)$$

则可以写成

$$m^3 = c + kn$$

进一步

$$m = \sqrt[3]{c + kn}$$

m很小时只需暴力枚举k的值，也有可能会使密文被爆破

举例：

```
import gmpy2
n =
0xb99a120d9b8ec040e1bcea45959d2f0076439c44cc898658144af9f3701532420cfcbceb03bb32d99f9e
051703bec8bfa50a10d39942597ab4cff3aa42842f45e981975c0bd0cf0560845a0930a094fd52e4cc1d3f
d350babf2864ee677d8f7059872f0add2b0993df764fda8cf468dc2077531e21151b309337b62a512c0ab7
e = 0x3
c =
0x9d6a3d450e03a161f8849877080571ca3828fbe158e4fc27bd687d6a75150266d90e54dbade7592b4672
45011797cdc2bcf0b0d1bc245cfa87b4c0c32f9bee969e2283ab0895ffd0a1e27f09499f80b5e3bcf95bd9
aa9fe941b8bb64d7cbd9acfb216b1f8e391ab801ec0f4d82aae27bfd93ac9a37b4d098d7c66c3e8c4ef795
i = 0
while 1:
    res = gmpy2.iroot(c+i*n,3)
    if(res[1] == True):
        m = res[0]
        print(hex(m))
        break
    print("i="+str(i))
    i = i+1
```

共模攻击

在RSA的使用中,使用了相同的模n、不同的模e对相同的明文m进行了加密，那么就可以在不分解n的情况下还原出明文m的值

已知

$$m^{e_1} \equiv c_1 (mod\ n)$$
$$m^{e_2} \equiv c_2 (mod\ n)$$

那么，可以想办法构造$m^{ae_1+be_2}$使得$ae_1 + be_2 = 1$，此时回忆一下extgcd，当两个e互素时，可以求出a和b的值

```
import gmpy2
from Crypto.Util.number import long_to_bytes
n =
0xa44bc0ae5e054a5ead7eba9ff6844807c89f212475f70c93cd70745570eac8f7fc4c97e853c47ebd9e82
7409a029176c83c072e980ba96ba6070045777d09a554b32dcae81d03b2bfe39c87da7460c3da4b7fe2b90
b7d43adf17c00d95e3249e27643654124a0ed659fe8ea072003df4ed1d9f8fa6b6a6e31c984096e2a3228d
e1 = 0xb733349f37e8547ae9
e2 = 0xff80413c40bbcb1ad3
c1 =
0x18bbc1c4ef4ed1e5f9c7f1fa26d82de5b56e632bd6db8e5336ced7504bea71914d2a0a822860ba2c90ae
936ac6c113e0f1f49986d9610187e52ac9a4ba4ffd5e48da9fa89d5239feec1bd5a527fe30ea96497a0504
6a6c722a7d13f68d0a0f28d7030e108d0dbbb4c6b60bc62ac417d43e9a88f84ccbad3cadfcfc83e31612ad
c2 =
0x2888a1d502de6dad3d600fa297c9b35ce8ddc2a8d5d242d9a4eccec29d94981346aa8ff44f91f8ed9afa
02b4c539c47b966e285655e0aa0c17c663a4d2ae3e96dc597a7c77d0cb8e02edd84ca863e181a1123b71f8
1713b075d9f39e6a8d2a332cca259af9abeaae65f5faaf181945a609f989039cebfff7f808e107eb6a9b3d
gcd, s, t = gmpy2.gcdext(e1, e2)
m = gmpy2.powmod(c1, s, n) * gmpy2.powmod(c2, t, n) % n
print(long_to_bytes(m))
```

低指数广播攻击

首先回顾一下中国剩余定理

中国剩余定理（CRT）：

对于两两互素的正整数$m_1, m_2, \cdots, m_k$，同余方程组

$$\begin{aligned} x &\equiv a_1 (mod\ m_1) \\ x &\equiv a_2 (mod\ m_2) \\ &\vdots \\ x &\equiv a_k (mod\ m_k) \end{aligned}$$

在模$M = m_1 m_2 \cdots m_k$同余类下有且仅有唯一正整数解

其解为$x \equiv \sum_{i=1}^{k} a_i M_i^{-1} M_i$，其中$M_i = M/m_i$，$M_i M_i^{-1} \equiv 1 (mod\ m_i)$

再看回低指数广播攻击，其特点是，使用极小的e（通常为3），对相同的m，多次使用不同的n进行加密

通过CRT，就可以得出m^e在模$n_1 n_2 n_3$同余类下的解，因为e极小，所以大概率$m^e < n_1 n_2 n_3$（当e≤3时是一定的）

```
n1 =
0x87e9810feb6ecc7a4e55233a76626965eba9242667db3b3aa1be75c8240262d18543600149c9e90a49bc
d5262679883c37a9b65a8b05b48de1a0696576e79df32e6a2e6685d732972cf9c9da7897f6a1b87cb26176
75b71ec67e48f8cdca7d78c92c99a32681bb7e2eafc61c65637356119498a72dc671e829e5cf8d0a79ddb7
n2 =
0x81e247044f044a0cba17906b829c650b388814e757d4f45657ad8782a2bfe3e740d87c1f0ba9e00662c6
a5ec0bffa9cc2cc2974f374733fc4bd6a743fd2c222b395f7da5009fad4e748dabba0aaaf793b1b31134c6
c85f7274e30bc7ef1f8f1530eb947002e7529395e2af892cc2ddbf829989cb0f9f77a431df46388e2ca25f
n3 =
0x8533bfb4a4e5106c72f9bdf58e9b6459af80e74e1f82c9725efa30f92c584a1f506c977efda553153bbc
778c3eee7dcff2f3252a988700a8fd11be160834b8740387d94d2d197232886da6b787f00030f783011fe1
f662ce56cb66eafd1f115dc9348c43caf9a419dd01c1249c773f1106be76f3c26a02446c156374531aabaf
c1 =
0x4213640633166627acd96336c4abc12de583e852b7afb2ae83a7ef802ff3412e77cd0fd5ed18f94178e5
6157cf6c6b46adf66e545c904c0fad52b09210c598f3c0e554dbe2984a5f00d713d85e57ef73ca5184821a
648d5c41ba2b27046421c88f67d1833cf4110dcd0e3dcb3b91776676a717c352619b9e1c564ecdcf75fe26
c2 =
0x7418e32b2273d227fb36fbd1c794e609f0e72167e9e46021728799c7718b19d4f9215537a28269c3ce67
5df156c4281a9660b41307cd49211342009c5557d78dfbe47d5f59326398f8e1a9e535ff76f537f83ba9ac
fa6667a33de60afe9423643111af0ffeffdb14c81108b51000070ac9c8062c07367bc3127cef9c03ab16f4
c3 =
0x39147c85bb76babf38b27b02708ee4d00d349ecbe173c6ec20937902e012d3afbc4ce6e7b0dfa4b0bd49
8fc5130a2988253f4832d97004fd25b39a8cf38661871fc14503593800e84988fbb33d5eb3dbb1c180a11b
a39673f82ef0cc9a6ff56a1b1c37ef9405d32ccaa76f048d1a202ae95870176b893e08ddf0f3e95e13a7d4
e = 3
N1 = n2*n3
N2 = n1*n3
N3 = n1*n2
N = n1*n2*n3
t1 = gmpy2.invert(N1,n1)
t2 = gmpy2.invert(N2,n2)
t3 = gmpy2.invert(N3,n3)
m3 = (c1*t1*N1+c2*t2*N2+c3*t3*N3)%N
mi = gmpy2.iroot(m3,e)
m = mi[0]
print(long_to_bytes(m))
```

选择明/密文攻击

> 有一个自动RSA加/解密的机器，使用者不知道它的n和e，但是它真的是安全的吗？

现在，对自动RSA加密机，输入2、4、8

$$
\begin{aligned}
c_2 &= 2^e \pmod{n} \\
c_4 &= 4^e \pmod{n} \\
c_8 &= 8^e \pmod{n}
\end{aligned}
$$

可以计算出$c_2^2 \equiv c_4 \pmod n$，$c_2^3 \equiv c_8 \pmod n$，这样的话$c_2^2 - c_4$和$c_2^3 - c_8$就都是n的倍数，可以多用几组求出n的具体值

这次RSA的作业之一就是校巴上的Republican Signature Agency这道题，有不会的地方随时来交流(✿◡‿◡)

这边稍微补充一下如何过PoW（power of work），可以直接使用python的itertools进行遍历爆破，当然，也可以直接调用pwntools里的bruteforce或者mbruteforce方法，后者并行计算速度会快一些

```
from pwn import *
from pwnlib.util.iters import mbruteforce
from hashlib import *

context.log_level = 'debug'

p = remote('10.214.160.13', 12505)

p.recvuntil(b' == ')
s = p.recvn(6)
p.recvuntil(b'Give me str: ')
print(s)

ans = mbruteforce(lambda x: sha256(x.encode()).hexdigest()[-6:] == s.decode(),
string.ascii_letters + string.digits, length = 4)
print(ans)

p.sendline(ans.encode())

p.interactive()
```

更多的RSA攻击方法可以查看前年的课件（这次的RSA部分纯纯的Copy & Paste），较为有趣的是利用连分数趋近实现的维纳攻击

格基规约

Lattice学习笔记01：格的简介

格基规约相关 - Coinc1dens' Lotus Land

格基规约算法：数学基础

格概述 - CTF Wiki

Coppersmith 相关攻击 - CTF Wiki

格基规约

线性代数

来点简单的线代回忆

基：线性无关向量组，其线性组合可以张成整个线性空间

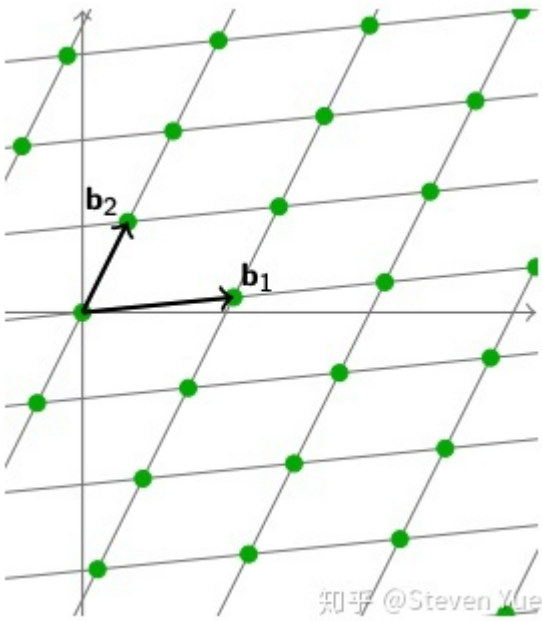
Gram-Schmidt正交化：一种让线性无关向量组等价变为标准正交向量组的方法，通过这种方法求出某个向量空间的标准正交基，能大幅减少后续计算的运算量

$$
\begin{aligned}
&\beta_1 = \alpha_1 \\
&\textbf{for } i \ in \ (2, n] \textbf{ do} \\
&\quad \mu_{ij} = \frac{(\alpha_i, \beta_j)}{(\beta_j, \beta_j)}, 0 < j < i \\
&\quad \beta_i = \alpha_i - \sum_{j=1}^{i-1} \mu_{ij} \beta_j \\
&\textbf{end}
\end{aligned}
$$

格

而格则是若干线性无关向量$\mathbf{b}_1, \ldots, \mathbf{b}_n \in \mathbb{R}^m$整系数线性组合构成的集合，即$\mathcal{L} = \sum_{i=1}^{n} \mathbf{b}_i \cdot \mathbb{Z} = \{\sum_{i=1}^{n} c_i \mathbf{b}_i : c_i \in \mathbb{Z}\}$，而$\mathbf{B} = (\mathbf{b}_1, \ldots, \mathbf{b}_n)$就称为格基

简而言之，下图就能概括出格的特征



格中NP问题

最短向量问题（SVP）

给定格L及其基B，找到格L中的最短非零向量v

最近向量问题（CVP）

给定格L和目标向量t，找到一个格中的非零向量v，使得对于格中的任意非零向量u满足$\|v-t\| \leq \|u-t\|$

连续最小长度问题（SMP）

求出给定秩为n的格L的连续最小长度，即找到格L中n个线性无关向量s_i，对于任意1≤i≤n，满足格中i个线性无关的向量$\|v_j\| \leq \|s_i\| = \lambda_i, 1 \leq j \leq i$的最小值

另外还有最短线性无关向量问题（SIVP）等格上问题也是计算困难的

格基规约

我们对把一个Lattice的基进行变换，找到一组非常接近垂直的基的过程成为格基规约，完成格基规约后就能快速求解SVP、CVP等问题

高斯格基规约算法

这是一种较为古老的二维格基规约算法，比较像GCD和施密特正交化的结合

LLL算法和BKZ算法

现代格基规约算法主要使用LLL算法以及其改良算法BKZ算法，BKZ的具体的算法不在这里叙述了，有兴趣的同学可以在参考网站中自学

LLL(Lenstra–Lenstra–Lovász)算法，可以看作是一种高斯算法在高维格中的推广算法

若格基B满足以下两个性质

(1) 对于任意的j < i ≤ n，有$|\mu_{i,j}| = |\frac{(\mathbf{b}_i, \mathbf{b}_j^*)}{(\mathbf{b}_j^*, \mathbf{b}_j^*)}| \leq \frac{1}{2}$ (2) 任取i有$\delta\|\mathbf{b}_i^*\|^2 \leq \|\mathbf{b}_{i+1}^* + \mu_{i+1,i}\mathbf{b}_i^*\|^2$（Lovász condition）

则称其为格L的一组δ-LLL约化基

而LLL算法的大致思路是，按顺序对基做施密特正交化（取整），再判断与前一个基是否满足Lovász condition，若满足则继续处理下一个，不满足则交换两者并回退到已处理过的基中最大的那个，直至Lovász condition全部满足

Learning With Errors

正因格中许多计算是属于计算困难性问题，因此可以用来作为非对称加密，便称之为格密码，格密码最引人注目的特点就是它是抗量子计算攻击的。在这里，我们着重介绍LWE这种格密码

从CryptoHack上的一道题说起

```python
from utils import listener
from sage.all import *


FLAG = b"crypto{??????????????????????}"

# dimension
n = 64
# plaintext modulus
p = 257
# ciphertext modulus
q = 0x10001

V = VectorSpace(GF(q), n)
S = V.random_element()

def encrypt(m):
    A = V.random_element()
    b = A * S + m
    return A, b


class Challenge:
    def __init__(self):
        self.before_input = "Would you like to encrypt your own message, or see an encryption of a character in the flag?\n"

    def challenge(self, your_input):
        if 'option' not in your_input:
            return {'error': 'You must specify an option'}

        if your_input['option'] == 'get_flag':
            if "index" not in your_input:
                return {"error": "You must provide an index"}
                self.exit = True

            index = int(your_input["index"])
            if index < 0 or index >= len(FLAG) :
                return {"error": f"index must be between 0 and {len(FLAG) - 1}"}
                self.exit = True

            A, b = encrypt(FLAG[index])
            return {"A": str(list(A)), "b": str(int(b))}

        elif your_input['option'] == 'encrypt':
            if "message" not in your_input:
                return {"error": "You must provide a message"}
                self.exit = True

            message = int(your_input["message"])
```

```
            if message < 0 or message >= p:
                return {"error": f"message must be between 0 and {p - 1}"}
                self.exit = True

            A, b = encrypt(message)
            return {"A": str(list(A)), "b": str(int(b))}

        return {'error': 'Unknown action'}


listener.start_server(port=13411)
```

简单来说，上面的代码每次随机一个向量，又留存了一个向量作为私钥，每次加密都将两个向量相乘，再加上明文，变为密文，这样的题目如何求解呢?

首先，简单回忆一下线性代数课上求线性方程组的经历，线性代数中我们将线性方程组表示为了矩阵的形式，通过构造$A\cdot X = B$，利用高斯消元法或者矩阵求逆得到X的值

那么，上面的题目其实也就是一个线性方程组求解的题目，进行多次加密后，将A拼成满秩的矩阵，就能求解出S的值了

然而，密码学家们找到了一种神奇的后量子密码的构造方法，只要对上面的操作稍作改进即可使得“线性方程组”的求解成为一个几乎不可能的问题

```
# dimension
n = 64
# plaintext modulus
p = 257
# ciphertext modulus
q = 0x10001
# bound for error term
error_bound = int(floor((q/p)/2))
# message scaling factor
delta = int(round(q/p))


V = VectorSpace(GF(q), n)
S = V.random_element()
print("S = ", S, "\n")

m = ?

A = V.random_element()
error = randint(-error_bound, error_bound)
b = A * S + m * delta + error

print("A = ", A)
print("b = ", b)
```

简单来说，将A作为公钥，S作为私钥，m是一个较小的值，对于每次求出来的值，我们都给它加上一个小扰动e，此时先前对于线性方程组的求解完全被破坏了，如果不知道S就基本不可能完成密码的破解

这里，$A\cdot S+E=B$就是LWE的核心思路，那么，这种密码如何才能解密呢?

对于上面的加密方法$A\cdot S+\Delta\cdot m+e=b$，将误差error限定在$[-\frac{\Delta}{2},\frac{\Delta}{2}]$中，这样，在解密的时候，$\Delta\cdot m+e=b-A\cdot S$，在此时，如果将等号两边除以Δ，再进行四舍五入，就能得到m的值了

```
# dimension
n = 64
# plaintext modulus
p = 257
# ciphertext modulus
q = 0x10001
# bound for error term
error_bound = int(floor((q/p)/2))
# message scaling factor
delta = int(round(q/p))


V = VectorSpace(GF(q), n)
S =  V((55542, 19411, 34770, 6739, 63198, 63821, 5900, 32164, 51223, 38979, 24459,
10936, 17256, 20215, 35814, 42905, 53656, 17000, 1834, 51682, 43780, 22391, 33012,
61667, 37447, 16404, 58991, 61772, 44888, 43199, 32039, 26885, 17206, 62186, 58387,
57048, 38393, 29306, 58001, 57199, 33472, 56572, 53429, 62593, 14134, 40522, 25106,
34325, 37646, 43688, 14259, 24197, 33427, 43977, 18322, 38877, 55093, 12466, 16869,
25413, 54773, 59532, 62694, 13948) )

A =  V((13759, 12750, 38163, 63722, 39130, 22935, 58866, 48803, 15933, 64995, 60517,
64302, 42432, 32000, 22058, 58123, 53993, 33790, 35783, 61333, 53431, 43016, 60795,
25781, 28091, 11212, 64592, 11385, 24690, 40658, 35307, 63583, 60365, 60359, 32568,
35417, 22078, 38207, 16331, 53636, 28734, 30436, 18170, 15939, 966, 48519, 41621,
36371, 41836, 4026, 33536, 57062, 52428, 59850, 476, 43354, 61614, 32243, 42518,
19733, 63464, 29357, 56039, 15013))
b =  44007
x = int(b - (A * S) % q)
m = int(round(x / delta))
print(m)
```

当然，还有另外一种信息低比特存储的LWE加密方法，$A\cdot S+p\cdot m+e=b$，具体可以参照CryptoHack

LWE的安全性非常高，但是不当的参数选取仍旧会导致被攻破，最常见的就是error的值过小

回忆一下前面刚讲的格基规约，用格基规约求取的问题似乎都是些“小问题”（如求最短向量），那对于LWE，我们要求的其实是一个CVP问题，即求出A格中最接近b的格向量

对于

$$
\begin{aligned}
A_n &= [A_{n,1}\ A_{n,2}\ \cdots\ A_{n,m}] \\
S &= [S_1\ S_2\ \cdots\ S_m] \\
b_n &= A_n\cdot S+e_n(mod\ q)
\end{aligned}
$$

构建这么一个格矩阵（未写明部分为0）

$$(-1\ S_1\ S_2\ \cdots S_m\ k_1\ k_2\ \cdots\ k_n)\begin{bmatrix} b_1 & b_2 & \cdots & b_n & BIGNUM \\ A_{1,1} & A_{2,1} & \cdots & A_{n,1} & \\ A_{1,2} & A_{2,2} & & & \\ \vdots & & \ddots & & \\ A_{1,m} & & & A_{n,m} & \\ q & & & & \\ & q & & & \\ & & \ddots & & \\ & & & q & \end{bmatrix} = (e_1\ e_2\ \cdot$$

经过前面向量的线性变换，能使得构造出的向量尽可能的小，从而达成e的求解，再通过正常的线性代数知识求得S

```
# dimension
n = 64
# plaintext modulus
p = 257
# ciphertext modulus
q = 1048583
# round
m = 100

V = VectorSpace(GF(q), n)
S = V.random_element()

A = Matrix(GF(q), [V.random_element() for i in range(m)])
e = vector(GF(q), [randint(-1,1) for i in range(m)])
b = A*S+e

#attack
mat = []
BIGNUM = q

for i in range(100):
    mat.append([int(b[i])]+list(map(int,A[i])))
mat.append([BIGNUM] + [0] * n)
M = Matrix(ZZ, mat)
M = M.transpose()
M = M.stack(diagonal_matrix(m + 1, [q] * m))
M = M.dense_matrix()
res = M.LLL()
error = list(res[-1])
print("error", error)
if error[-1] < 0:
    error = [e * -1 for e in error]
assert error[-1] == BIGNUM
assert all([abs(e) <= 1 for e in error[:-1]])

error = vector(GF(q),error[:-1])
assert e == error

Secret = A.solve_right(b-error)
print("Secret", Secret)
assert S == Secret
```

RSA Coppersmith 攻击

前面说了，LLL是一个对找“小东西”的很有效的工具，那么，对于RSA，LLL是否也能发挥自己的作用呢?

Don Coppersmith所构建的Coppersmith攻击，是通过构造包含小整数根的模多项式方程，在多项式时间内找到所有方程的小整数根，这样对某些特殊情况RSA的破解会有很大的作用，这里具体的推导不再赘述~~（太难了）~~，感兴趣的可以参考这篇文献Twenty Years of Attacks on the RSA Cryptosystem或者去看看前年的课程课件和录播

这么经典的攻击方法，当然可以调库啦😋

sage里自带了small_roots方法，只需一行代码就能求出小根

`sage.rings.polynomial.polynomial_modn_dense_ntl.small_roots(self, X=None, beta=1.0, epsilon=None, **kwds)`

其中$X = ceil(\frac{1}{2}N^{\frac{\beta^2}{\delta}-\epsilon})$，β的意义是n的某个因数b使得$b \geq n^{\beta}$，所以0<β≤1

这里补充一点，small_roots方法不仅能找到当前模下的小根，还能找到其因数的模下的小根

下面是coppersmith攻击的常见例题

已知明文高位

```
n =
0x2680048649769800c79ec1f3ceab3909b8dc665c2f44dea93678a3c604c1e87cd7500a59e126bbeed103
18c83807c7701d8968448200a43b7f64e697129c8fa88b52f0e62ce3166a0d817f195452e0de6fe27e1d29
39a88756e2b38694837513199aa6d92ba88dd895020a53ef007b0f57478c7f363d4d262db4c2da5e8b1575
c =
0xbdf9abbdf60072d7421a33bd7294c76fcea8f5cac435e552b003d9192de4391a049fc8911b15378091bd
4f87a6173671abcd13e92d353d5b1f0798d49538d4da98931b5f021ca7743036b996893b8d81910859edba
619c28ec237e5f7abc0d9a4fdb1a85ba44c22b9031d3317998eb41d4d11ab34ab1f731ba50697317029cd
mbar =
0x472796951dbf2909faba23a804ff52df69a44e166d7a7bd2c57eeba22e4ea2dcb99123fdb17db038a13a
f1061c41daab206a88881c8023000000000000000000
kbits = 72
e = 3
PR.<x> = PolynomialRing(Zmod(n))
f = (mbar + x)^e - c
x0 = f.small_roots(X=2^kbits, beta=1)[0]  # find root < 2^kbits with factor = n
print(hex(mbar + x0))
#
0x472796951dbf2909faba23a804ff52df69a44e166d7a7bd2c57eeba22e4ea2dcb99123fdb17db038a13a
f1061c41daab206a88881c8023cd25397dfaff8641ec
```

已知p，q其中一个的高位

```
n =
0x63d1b2ecd4fd3da6ce2afe6c7b2e9086920a461f1b6a0d0c5782563c1e6af681cd03ea7ff4b4cb5d6f98
0fca6f605db0376c337e0921798f395fb0088b895eebe4b977251c9000c9c80055edbdc89068e0e88f3116
41407c8082aaab9497ee685005063af64b735143d18bbb0c72ee02220bd68c8250e5bcfde8d3d7e66bf2b5
p =
0x8b8a9cc1a5373864eaf7671d0a12547163c83408f1e99dbfd531f86dc7af585bae265a749f63d05016b4
596ecdc5308500000000000000000000000000000000
pbits = 512  #p的位数
kbits = 128     #未知位数
pbar = p & (2^pbits-2^kbits)
print("upper %d bits (of %d bits) is given" % (pbits-kbits, pbits))
PR.<x> = PolynomialRing(Zmod(n))
f = x + pbar
# f = (x + pbar) % p = 0
x0 = f.small_roots(X=2^kbits, beta=0.4)[0]  # find root < 2^kbits with factor >= n^0.4
print(hex(x0 + pbar))
#
0x8b8a9cc1a5373864eaf7671d0a12547163c83408f1e99dbfd531f86dc7af585bae265a749f63d05016b4
596ecdc53085ca1dbee39771f2c4258f31116c6d7003
```

已知d低位求p,q

```
def partial_p(p0, kbits, n):
    PR.<x> = PolynomialRing(Zmod(n))
    nbits = n.nbits()

    f = 2^kbits*x + p0
    f = f.monic()
    roots = f.small_roots(X=2^(nbits//2-kbits), beta=0.3)  # find root < 2^(nbits//2-
kbits) with factor >= n^0.3
    if roots:
        x0 = roots[0]
        p = gcd(2^kbits*x0 + p0, n)
        return ZZ(p)

def find_p(d0, kbits, e, n):
    X = var('X')

    for k in range(1, e+1):
        results = solve_mod([k*X^2 + (e*d0 - k*(n+1) - 1)*X + k*n == 0], 2^kbits)
        for x in results:
            p0 = ZZ(x[0])
            p = partial_p(p0, kbits, n)
            if p:
                return p


if __name__ == '__main__':
    n =
0x5c6b8458509910007051be15d867161643d8c62a3032200261b8fef53403d4a7639e4f810fc34ddbce0c
49156bab8686ecea8a2cfeba2349304535e483fd40cfedb8c45163f130f98d23edb8bca3c049f34f11c92c
9ec7df9a9096221564a59f9e165106144b4b2f890d19e1bc82d24b4e2fe27b34428fd635f49f3e589fb5a7
    e = 3
    d =
0xc54fb4b61153358ca26946efd6c472b248715edb73d853900400e2488535847b5030dd97a619693fc9bf
8151a1562b291193f87518fae5c1674266cd6f0a91bb

    nbits = n.nbits()
    kbits = floor(nbits*(0.5))
    d0 = d & (2^kbits-1)
    print("lower %d bits (of %d bits) is given" % (kbits, nbits))

    p = find_p(d0, kbits, e, n)
    print(hex(p))
```

对于coppersmith的利用，还有许多其它复杂的攻击方法，如Broadcast Attack with Linear Padding，Related Message Attack，Boneh and Durfee attack等，有兴趣的同学可以自行查阅论文资料学习，通常这些复杂的攻击方法都是直接套模板的，没有太大意义，题目中通常要让你自己通过数学推导找到一个“小根方程”（比如上面的已知d低位求p,q）

作业概述

实验需要提交实验报告。每道做出来的题均需要写在实验报告中，否则无法给分。**实验报告需要写出每道题的思路并贴上攻击脚本（payload）**。对于没法完整做出的题，也可以叙述自己的思路和解题过程，会酌情给分。

今年仍旧有**15分的保底分**（可能也会调整），只要提交作业，成功做出任意一题就能拿到，今年的保底分直接加到最后分数中，因此其实你**只需拿到85分就能满了**。本次虽然所有作业总分为155分，不过最多只能获得15分的bonus，加满为止，所以可以合理选取作业题目。

本次crypto lab对python以及sage的要求会比较高，如果认为自己对python的了解还是不够的话，请务必善用搜索引擎，并积极向助教们提问（对于密码学库的问题尽量咨询密码学方向助教，不过其它python相关问题可以询问所有助教）

本次有两道题目部署在ZJU::CTF平台，有两道为校巴上的题目，另外还有一道CryptoHack上的题目，可以在上面提交flag验证是否正确。

* **声明：由于前两年抄袭现象较为严重，本次作业所有题目都会进行查重，查到就不仅仅是这次Lab得0分了**😳

AES部分

- 完成课上例题CBC Byte Flip，题目部署在ZJUCTF平台，本题分值为20分
- 完成课上例题Padding Oracle，部署在ZJUCTF平台上，本题分值为30分

RSA部分

- 完成校巴上的Republican Signature Agency这道题，学习RSA选择明/密文攻击，地址为10.214.160.13:12505，分值25分
- 来道简单的Coppersmith攻击练练手，题目名是Crush On Proust，在校巴上可以看到，题目不算太难，但对数学要求较高，本题分值35分

格密码

- 来点CryptoHack，CryptoHack - Post Quantum - Learning With Errors - Noise Cheap，本题分值30分