Name: 何秉學　Student ID: R11921A16
:::spoiler TOC

:::

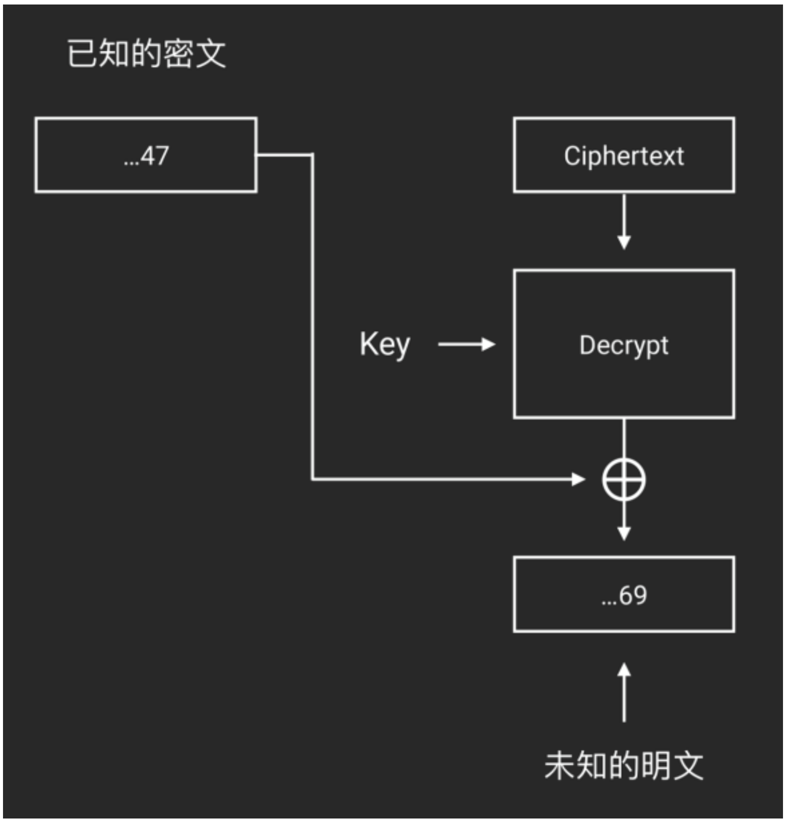
Lab-COR

Flag: `FLAG{Corre1ati0n_Attack!_!}`

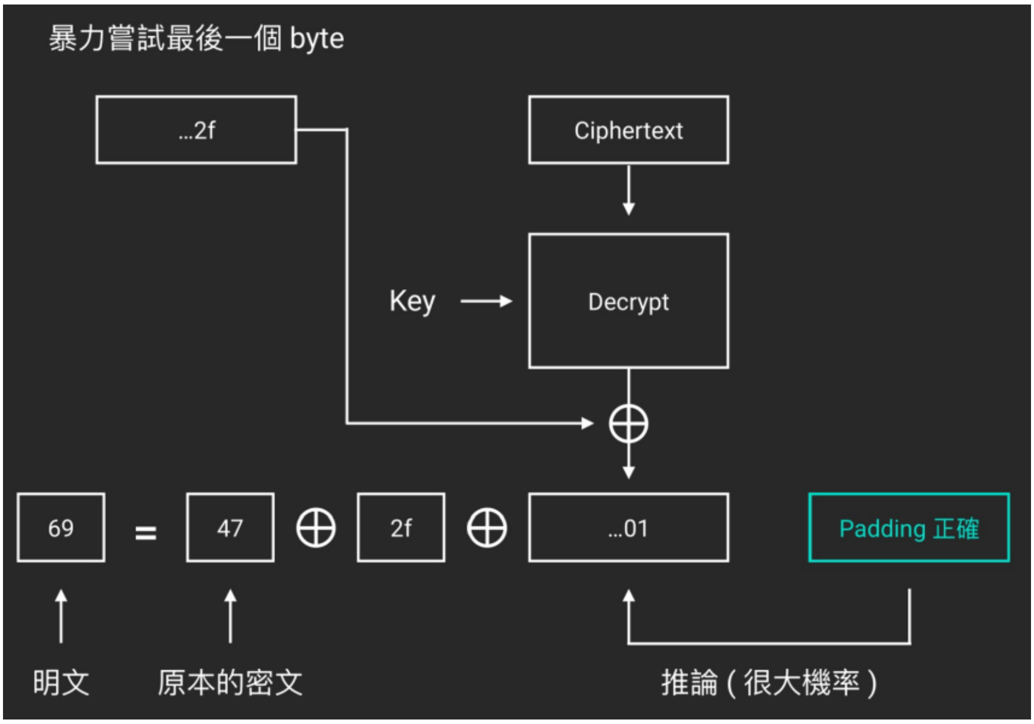
解題流程與思路

這一題是簡單的padding oracle attack，他一樣是應用在CBC mode上，只是他padding的方式和上課教的有一點不一樣，他會先在最後放一個0x80然後接續放0x00直到長度%16==0，同樣的，我們可以用上課教的方式:

- What we have: 我們有的東西就是密文，所以可以利用它動一些手腳
- Our Goal 1: 目標是要取得原本和47進行XOR的數字是多少
- Our Goal 2: 這樣才可以取得最後的明文69



- How to achieve: 我們可以簡單猜一個byte，從0x00開始，把密文換成猜測的byte，這樣256種組合和原本的Goal 1所求的byte進行XOR後會padding正確(也就是0x01)，此時假設我們已經猜到目前是0x2f符合padding正確的目標，代表現在的假明文是0x01，則原本和0x47進行XOR的數字就是0x01⊕0x2f，然後我們就可以回到原本解密的流程，也就是原本的密文0x47⊕剛剛得知的(0x01⊕0x2f)，就會得到想要的正確的明文0x69



所以套用到今天的lab意思也是一樣，如果要知道padding是否正確可以問oracle，反正只要符合明文+0x80+(0...15)*0x00，這一題的flag長度可以從題目給的ciphertext看出來，顯然扣掉16bytes的initial vector後，flag的長度是32 bytes，也就是說我們從第二個block開始解，我們可以單獨把第一個ciphertext block當成第二個ciphertext block的initial vector，合併後再一起送出去，然後不斷變化IV的最後一個byte，如果oracle回傳`Well received :)`代表第一個bytes猜對了，我們就可以把flag的最後一個bytes求出來$\to$我們猜的byte⊕原本ciphertext的最後一個byte⊕0x80(0x80是我們判斷padding正確的依據)，當然找到真正的plaintext byte後要把我們猜測的block恢復原狀，接著繼續找下一個byte

Lab-LSB

Flag: `FLAG{Viycx_qsklsjgmeld_fgd_spkgjo}`

解題流程與思路

這一題是變形過的Lease Significant Bit，上課教的例子是mod 2下的結果，而看source code可以知道目前他是mod 3下的結果，但換湯不換藥，只要把上課教的部分全部換成mod 3就可以了

1. 首先計算$3^{-1},3^{-2},3^{-3},3^{-4},...,3^{-(log_3^n)}\ (mod\ 3)$，並建立一個table
2. 依序執行上課教的流程
    1. 密文*$(3^{-1})^e$
    2. 合併要減掉的部分，也就是把之前已知道所有部分都乘以table上對應的反元素
    3. 再把oracle回傳的假明文減掉上面合併的部分(記得mod)，就是我們要的bit

Lab-POA

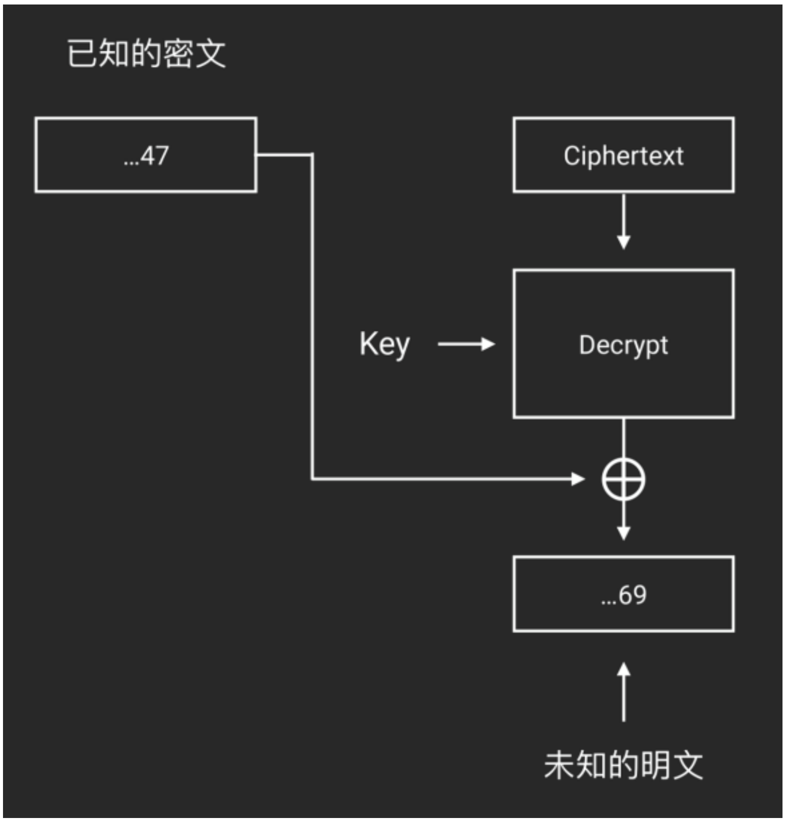
Flag: `FLAG{pAdd1NG_0rAcL3_A77aCK}`

解題流程與思路

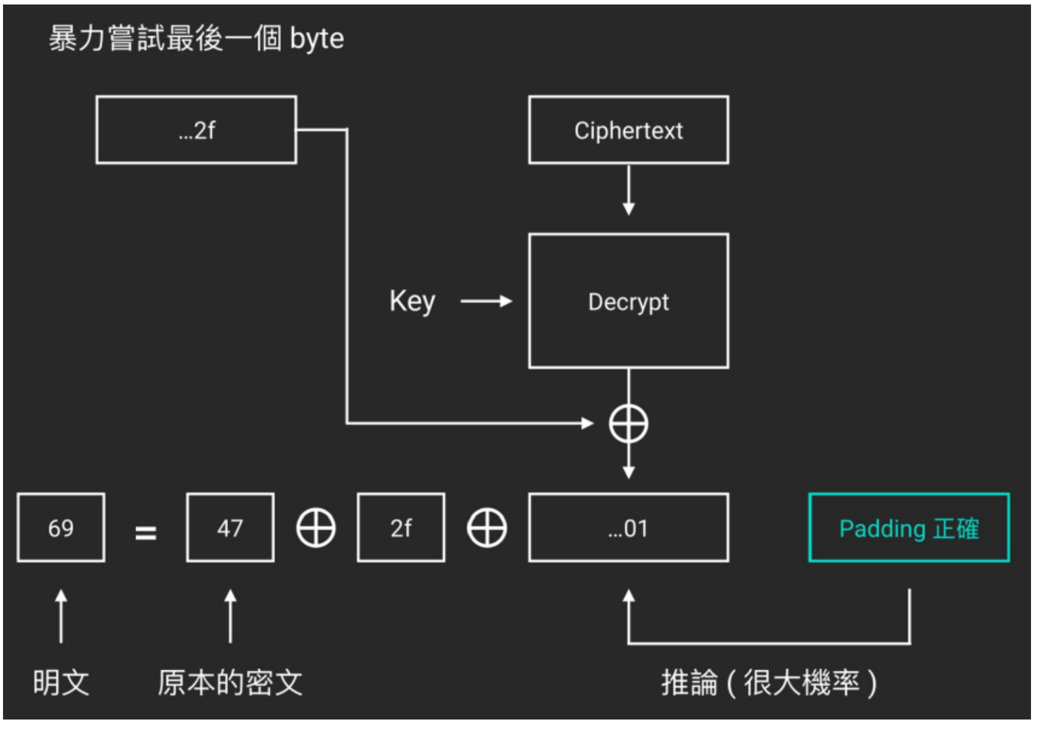
這一題是簡單的padding oracle attack，他一樣是應用在CBC mode上，只是他padding的方式和上課教的有一點不一樣，他會先在最後放一個0x80然後接續放0x00直到長度%16==0，同樣的，我們可以用上課教的方式:

- What we have: 我們有的東西就是密文，所以可以利用它動一些手腳
- Our Goal 1: 目標是要取得原本和47進行XOR的數字是多少

- Our Goal 2: 這樣才可以取得最後的明文69



- How to achieve: 我們可以簡單猜一個byte，從0x00開始，把密文換成猜測的byte，這樣256種組合和原本的Goal 1所求的byte進行XOR後會padding正確(也就是0x01)，此時假設我們已經猜到目前是0x2f符合padding正確的目標，代表現在的假明文是0x01，則原本和0x47進行XOR的數字就是0x01⊕0x2f，然後我們就可以回到原本解密的流程，也就是原本的密文0x47⊕剛剛得知的(0x01⊕0x2f)，就會得到想要的正確的明文0x69



所以套用到今天的lab意思也是一樣，如果要知道padding是否正確可以問oracle，反正只要符合明文+0x80+(0...15)*0x00，這一題的flag長度可以從題目給的ciphertext看出來，顯然扣掉16bytes的initial vector後，flag的長度是32 bytes，也就是說我們從第二個block開始解，我們可以單獨把第一個ciphertext block當成第二個ciphertext block的initial vector，合併後再一起送出去，然後不斷變化IV的最後一個byte，如果oracle回傳 `Well received :)` 代表第一個bytes猜對了，我們就可以把flag的最後一個bytes求出來$\to$我們猜的byte⊕原本ciphertext的最後一個byte⊕0x80(0x80是我們判斷padding正確的依據)，當然找到真正的plaintext byte後要把我們猜測的block恢復原狀，接著繼續找下一個byte

HW-LFSR

Flag: `FLAG{Lf5r_15_50_eZZzZzZZZzzZzzz}`

解題流程與思路

這一題和前面的triLFSR不一樣的地方在於他只有一層的LFSR，但他只會每個70個才會給一個state，換句話說我們只能拿到$S{71*0+70},\ S{711+70},\ S_{712+70},\ S_{71*3+70}...$(從0開始算)，而前面256個拿到的State最後會和flag進行XOR，只有最後70個是最純粹的State

- What we have
  我們有的東西就是Companion Matrix，因為題目有給taps，所以可以建出上課提到的矩陣；另外我們還有最後出現的70個State，雖然是每格70個出現一次，換句話說就是$State{71*256+70},\ State{71257+70},\ State_{71258+70},\ ...State_{71*325+70}$(從0開始算)
- Goal
  既然我們知道了State的公式為$s_m = p_0s_0 + p_1s_1 + ... + p{m-1}s{m-1}$，也就是companion matrix的最後一列$*$那64個initial state就會是新的state，換句話說，繼續往下做，其實就只是把companion matrix多乘幾次，然後還是一樣乘以initial state，然後我們只要取得companion matrix乘完之後的最後一列，就是下一個新的state的特徵，如下圖所示:

  在Round 0時，companion matrix的最後一列當然就是$S{64}$的特徵，再往下做，也就是Round 1時，companion matrix的平方後，再取最後一列就是$S{65}$的特徵，而題目給我們的ouptut[0]以state來說就是第70個(以0來說)，所以companion matrix的7次方，再取最後一列，以此類推，我們陸續算到output256，也就是companion matrix的$71256+7=18183$次方再取最後一列，就是$S_{71256+70}$的特徵，自此開始，我們就可以開始把這些特徵存起來，存滿64個後，再取反矩陣，乘上原本得到的那64個state，就可以得到一開始的initial state
- 完整的對應關係如下圖

HW-Oracle

解題流程與思路

這一題真的非常難，而且要通靈很久，首先Oracle.py的工作是把一張flag image用AES加密，並且把AES會用到的key/iv都用RSA再加密，然後通通傳給Alice，而Alice.py的工作才是本次作業實際上的Oracle，他會吃key/iv/ciphertext，前兩者是decimal，後者是hex形式，一開始可以先試看看把這三者傳過去，理論上只要格式對了就會回傳`OK! Got it.`

```
encrypted_key =
65690013242775728459842109842683020587149462096059598501313133592635945234121561534622365974927219223034823754673718159579772056712404749324225325531206903216411508240699572153162745754564955215041783396329242482406426376133687186983187563217156659178000486342335478915053049498619169740534463504372971359692
encrypted_iv =
35154524936059729204581782839781987236407179504895959653768093617367549802652967862418906182387861924584809825831862791349195432705129622783580000716829283234184762744224095175044663151370869751957952842383581513986293064879608592662677541628813345923397286253057417592725291925603753086190402107943880261658
enc_png = open('./Crypto/HW/Oracle/encrypted_flag_d6fbfd5306695c4a.not_png', 'rb').read()

r = remote("10.113.184.121", 10031)
r.sendlineafter(b'key: ', str(encrypted_key).encode())
r.sendlineafter(b'iv: ', str(encrypted_iv).encode())
r.sendlineafter(b'ciphertext: ', enc_png.hex().encode())
print(r.recvline().decode().strip())
```

解題的手法經@Yaan的小提示，完整如下:

1. 首先我們手上可控的地方，就是key/iv/ciphertext，一開始的想法是，由於此次的flag是一張png，所以一開始的magic header一定都一樣，所以可以透過這個magic header推測出IV是多少，但這樣的作法卻沒辦法知道key，所以這個方法行不通
2. 正確的作法是控制key/iv，變成自己設定的東西，然後試圖加密plaintext(同樣也是自己設定)，然後把自己設定的ciphertext/key以及原本題目給的encrypted_key或是encrypted_iv丟到oracle，要解密的部分(也就是encrypted_key/encrypted_iv)就當作是iv的部分輸入，這樣神奇的操作如下圖所示
3. 為甚麼這樣可以解出我們想要解的東西?那就要取決於如何控制plaintext/iv，key可以隨便控，而plaintext則是從零開始，iv也是全部都是零，這樣的好處是pt用AES加密前的部份是我們知道的，換句話說，在解密的時候和iv XOR前的數值也是知道的，此時我們可以從oracle output知道padding正確與否，我們又知道和iv XOR的數值是多少，則我們一定可以利用POA的方式推出原本的IV是多少
4. 舉個例子
   若
   `encrypted_iv=b'0123456789abcdef'` $\to$ unknown(也是我們想知道的部分)
   `self_pt=b'0000000000000000'` $\to$ self defined
   `self_iv=b'0000000000000000'` $\to$ self defined
   則我們開始改變`self_pt`的最後一個byte，也就是`b'0...00'`,`b'0...01'`,`b'0...02'`...，讓他和`encrypted_iv`進行XOR之後判斷padding正確與否

   如果padding正確也就代表目前的padding結果是`0x01`，而此時的`self_pt=b'0...0e'`，所以想當然`encrypted_iv=XXX...f`，而換到下一round，我們也改造一下`self_pt`，首先原本最後一個byte(0xe)要改成$0xe\oplus 0x2=0xc$，因為下一round的padding必須要是`0x0202`才會正確，然後我們就可以改變倒數第二個byte(一樣從零開始)，也就是`b'0...0c'`,`b'0...1c'`,`b'0...2c'`...，以此類推就可以得出真正的IV是多少了，而`encrypted_key`的做法也和IV一模一樣

Lab-dlog

Flag: `FLAG{YouAreARealRealRealRealDiscreteLogMaster}`

解題流程與思路

基本上這一題和上一個學期上的CNS中，作業二的Little Knowledge Proof概念一模一樣，當時還不知道這是啥騷操作，現在覺得非常簡單，就是套用了Pohlig-Hellman的原理進行破解

1. 首先看source code需要我們提供一個prime(N)，然後跟一個不重要的底數g，接著題目return一個hint就是$hint=g^{flag}\ mod(N)$，因此按照discrete log的難度，我們很難針對hint進行brute force，縱使我們知道N,g,hint也一樣，但因為N是我們提供的，所以我可以故意給他一個smooth prime，也就是$N-1$是由多個prime相乘而得

2. 我們可以用上課教過的Pohlig-Hellman原理去思考，也就是先把群的範圍縮小，再利用BSGS的方法找到x_i，這時雖然得到x_i但由於是mod p_i的結果，就不是真正的x，要利用CRT把多個x_i還原成原本的x，幸虧以上操作sage都做好了

Lab-signature

Flag: `FLAG{EphemeralKeyShouldBeRandom}`

解題流程與思路

這一題主要就是利用上課提到的nonce k不隨機的問題，因為k只能用一次，也就代表他需要夠隨機，如果像LCG這樣的psudo random generator產生的話，一但被compromise，就會被推導出private key d，而這個lab就是有這樣的問題

1. 觀察source code會發現不同的nonce k之間會產生一個1337倍數的關係，然後如果request `Give me the FLAG.` 的signature會被拒絕，所以只能自己產生`Give me the FLAG.` 的signature再丟給server檢查，如果過了就可以拿到flag，但重點是要怎麼偽造signature假裝是server簽的?就是要想辦法拿到server產生的private key d，可以詳細看一下source code中提到，通常public key都一樣，所以重點是d才能產生private key，然後用private key簽署message

```
E = SECP256k1
G, n = E.generator, E.order
d = randint(1, n)
pubkey = Public_key(G, d*G)
prikey = Private_key(pubkey, d)
↓
sig = prikey.sign(bytes_to_long(h), k)
```

2. 已知(題目給的部分)
只要我們給兩次要簽章的message，總共可以得到以下資訊

$$\begin{gathered} coordinate\ (x_0,\ y_0), \\ hash\ H_1,\ hash\ H_2, \\ signature\ (s_1,\ r_1),\ (s_2,\ r_2) \end{gathered}$$

3. 推導
假設$msg=b'a'$

$$\begin{gathered} H_1 = H_2 = sha256(msg) \\ k_1 = {s_1}^{-1} \cdot (H_1 + d \cdot r_1) = {s_1}^{-1} \cdot H_1 + d \cdot r_1 \cdot {s_1}^{-1} \\ k_2 = {s_2}^{-1} \cdot (H_2 + d \cdot r_2) = 1337 \times k_1 = \\ = {s_2}^{-1} \cdot H_2 + {s_2}^{-1} \cdot d \cdot r_2 \\ = 1337 \cdot {s_1}^{-1} \cdot H_1 + 1337 \cdot d \cdot r_1 \cdot {s_1}^{-1} \\ \downarrow \\ d \cdot (H_2 \cdot {s_2}^{-1} - 1337 \cdot H_1 \cdot {s_1}^{-1}) = 1337 \cdot r_1 \cdot {s_1}^{-1} - r_2 \cdot {s_2}^{-1} \\ \hookrightarrow d = \frac{1337 \cdot r_1 \cdot {s_1}^{-1} - r_2 \cdot {s_2}^{-1}}{H_2 \cdot {s_2}^{-1} - 1337 \cdot H_1 \cdot {s_1}^{-1}} \end{gathered}$$

4. 得到原本的private key d之後就可以直接選一個亂數nonce k，然後重新自己簽署`Give me the FLAG.` 的signature

Lab-coppersmith

Flag: `FLAG{RandomPaddingIsImportant}`

解題流程與思路

這一題看到`e=3`直覺會想到小明文攻擊，但是前提除了e要很小以外，明文也不能太大，要不然會找很久，他的原理是(假設`e=3`):

$$\begin{gathered} \because C \equiv m^3\ mod\ N \\ \therefore m^3 = C + k \times N \\ \hookrightarrow m = \sqrt[3]{C + k \times N} \end{gathered}$$

所以可以枚舉很多的k，並且依次開三次方，直到開出整數為止，但就像前面的前提，明文不能太大，不然也會找的很痛苦，此時就可以用到上課教到的coppersmith，解出這樣的問題

- Review Coppersmith Attack
問題：如果有一個${f(x)\equiv 0\ (mod\ N)\ |x=r, N\in \mathbb{Z}, f(x)\in \mathbb{Z}{[x]}}$，當$x=r$的時候會同餘$0$
想求：r是多少能符合以上的式子

首先這個問題因為mod是一個循環，所以正常情況下很難知道r多少能符合，因此我們可以簡化一下問題，或者說增加一些限制，這樣在尋找r的時候會比較好找一點

1. 首先構造一個

$$\{Q(x) = s(x) \cdot f(x) + t(x) \cdot N\ (mod\ N) \mid Q(r) \equiv 0\ (mod\ N), r \in \mathbb{Z}\}$$

在這裡可以先把r帶進去這個構造的式子，就會發現其實跟一開始求的問題，也就是$f(x)\equiv 0\ (mod\ N)$其實一樣，但為甚麼要這樣做呢?是因為把問題拉到實數域中求解後比較好做，等我們拿到r在實數域得到的root之後就可以帶回去$f(x)$中。

我們可以把r想像成是一個flag，然後flag會有一個最大可能性的上界，也就是R，假設flag有32個字元，代表256個bits，我們可以想像$R=2^{256}$，我們不知道flag是多少，但一定在R的這個範圍中，且flag一定是整數(換算成int的話)

2. 所以我們就可以重新寫一個bounded equation

$$Q(r) = |Q_n r^n + ... + Q_2 r^2 + Q_1 r^1 + Q_0| \le |Q_n| R^n + ... + |Q_2| R^2 + |Q_1| R + |Q_0|$$

有了這個bound equation後，我們就可以說

$$\because |Q(r)| < |Q(R)| < N 且 Q(r) \equiv 0 \; mod \; N$$
$$\therefore Q(r) = 0$$

有了以上條件和說明，此時我們確定把問題拉到實數域上了，現在還不知到r為多少

3. 而要知道r就必須知道$Q(r)$，只要得到$Q(r)$再利用找root的sage method就可以直接得到r為多少，但在得到$Q(r)$之前我們要先得到$Q(R)$，我們可以利用前面提到的$s(x)\cdot f(x)+t(x)\cdot N\ (mod\ N)$建一個多項式，然後用matrix表示並把$R$帶入，再利用LLL求shortest vector，此時的shortest vector是以$x=R$為條件帶入，所以只要在各個term把R除掉，就可以得到$Q(r)$各個term的係數，然後就求得r為多少了，舉例來說：

   在RSA中，已知$c= m^e\ (mod\ N)$，當我們今天拿到一個有padding明文(當然我們拿到的是密文，只是知道明文有經過padding，且padding的部分我們知道，另外flag的大小也不能太大，具體能多大可以看影片)，且$e=3$，我們可以rewrite整個式子(假設padding的部分為a，flag的部分為x)

   $$m = padding + flag$$
   $$c = m^3 = (padding + flag)^3 \; (mod \; N) \downarrow s(x) \cdot f(x) + t(x) \cdot N \; (mod \; N) = c_3(x^3 + 3ax^2 + 3a^2x + (a^3 - c)) + (c_2x^2 + c_1x + c_0) \cdot$$
   $$f(x) = (padding + flag)^3 - c \; (mod \; N)$$

   $s(x)=c_3$，如果把$f(x)$乘開就會是$x^3 + 3ax^2 + 3a^2x + (a^3 - c)$，而$t(x)=c_2x^2 + c_1x + c_0$。此時把矩陣的$x$帶入上界$R$再利用LLL求shortest vector，也就是

   $$\begin{bmatrix} c_3R^3 \\ (c_3 3a + c_2N) * R^2 \\ (c_3 3a^2 + c_1N) * R \\ (c_3(a^3 - c) + c_0N) \end{bmatrix}^T$$

   詳細過程如下:

   $$M = \begin{bmatrix} R^3 & 3aR^2 & 3a^2R & a^3 - c \\ 0 & NR^2 & 0 & 0 \\ 0 & 0 & NR & 0 \\ 0 & 0 & 0 & N \end{bmatrix} |x = R$$
   $$LLL(M) = \begin{bmatrix} c_3R^3 \\ (c_3 3a + c_2N) * R^2 \\ (c_3 3a^2 + c_1N) * R \\ (c_3(a^3 - c) + c_0N) \end{bmatrix}^T \hookrightarrow Q(x) = \begin{bmatrix} c_3 \\ c_3 3a + c_2N \\ c_3 3a^2 + c_1N \\ c_3(a^3 - c) + c_0N \end{bmatrix}^T \begin{bmatrix} x^3 \\ x^2 \\ x^1 \\ x^0 \end{bmatrix} \le Q(R) = \begin{bmatrix} Q_3 \\ Q_2 \\ Q_1 \\ Q_0 \end{bmatrix}^T \begin{bmatrix} R^3 \\ R^2 \\ R \\ 1 \end{bmatrix} \le N$$

4. 求flag(也就是求得$Q(x)$的root x_0)
   由以上過程，我們已經取得了$Q(x)$，則我們就可以在實數域中求$Q(x)$的根x_0

- 基本上這一題就是按照上面講的這樣解就可以了

HW-invalid_curve_attack

Flag: `FLAG{YouAreARealECDLPMaster}`

解題流程與思路

1. 觀察source code會發現maple實作了一個沒有檢查我們傳送的點是否在一開始創的橢圓曲線上的elliptiv curve class，然後他把我們給的point當作參數，創立一個初始點，可以看一下下面裡個範例，如果是maple的實作，給予一個根本不在該Elliptic Curve的點他還是會算一個G+G的點給你，只是該點其實是在別的曲線上的2G這個點，反觀正常的sage中的實作會發現只要給予的點不在該曲線上就會直接報錯

   :::spoiler maple 實作的Elliptic Curve

   ```
   >>> from elliptic_curve_97cadb52fbd7b2cd import Curve, Point
   >>> p=23
   >>> a=5
   >>> b=1
   >>> E = Curve(p, a, b)
   >>> G = Point(E, 4, 4)
   >>> print(G)
   (4, 4)
   >>> print(G+G)
   (19, 3)
   >>> fake_G = Point(E, 4, 3)
   >>> print(fake_G+fake_G)
   (17, 1)
   ```

   :::

   :::spoiler 正常的Elliptic Curve

   ```
   >>> from sage.all import *
   >>> p=23
   >>> a=5
   >>> b=1
   >>> E = EllipticCurve(Zmod(p), [a, b])
   ```

```
>>> G = E(4, 4)
>>> print(G)
(4 : 4 : 1)
>>> fake_G = E(4, 3)
Traceback (most recent call last):
  File "sage/structure/category_object.pyx", line 839, in
sage.structure.category_object.CategoryObject.getattr_from_category
(build/cythonized/sage/structure/category_object.c:7216)
KeyError: 'point_homset'

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "/home/sbk6401/anaconda3/envs/sageenv/lib/python3.11/site-
packages/sage/schemes/projective/projective_subscheme.py", line 122, in point
    return self._point(self.point_homset(), v, check=check)
           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "/home/sbk6401/anaconda3/envs/sageenv/lib/python3.11/site-
packages/sage/schemes/elliptic_curves/ell_point.py", line 259, in __init__
    point_homset = curve.point_homset()
                   ^^^^^^^^^^^^^^^^^^
  File "sage/structure/category_object.pyx", line 833, in
sage.structure.category_object.CategoryObject.__getattr__
(build/cythonized/sage/structure/category_object.c:7135)
  File "sage/structure/category_object.pyx", line 848, in
sage.structure.category_object.CategoryObject.getattr_from_category
(build/cythonized/sage/structure/category_object.c:7301)
  File "sage/cpython/getattr.pyx", line 356, in sage.cpython.getattr.getattr_from_other_class
(build/cythonized/sage/cpython/getattr.c:2717)
AttributeError: 'IntegerModRing_generic_with_category' object has no attribute '__custom_name'

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/sbk6401/anaconda3/envs/sageenv/lib/python3.11/site-
packages/sage/schemes/elliptic_curves/ell_generic.py", line 582, in __call__
    return plane_curve.ProjectivePlaneCurve.__call__(self, *args, **kwds)
           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "/home/sbk6401/anaconda3/envs/sageenv/lib/python3.11/site-packages/sage/schemes/generic/scheme.py",
line 266, in __call__
    return self.point(args)
           ^^^^^^^^^^^^^^^^
  File "/home/sbk6401/anaconda3/envs/sageenv/lib/python3.11/site-
packages/sage/schemes/projective/projective_subscheme.py", line 124, in point
    return self._point(self, v, check=check)
           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "/home/sbk6401/anaconda3/envs/sageenv/lib/python3.11/site-
packages/sage/schemes/elliptic_curves/ell_point.py", line 298, in __init__
    raise TypeError("Coordinates %s do not define a point on %s" % (list(v), curve))
TypeError: Coordinates [4, 3, 1] do not define a point on Elliptic Curve defined by y^2 = x^3 + 5*x + 1
over Ring of integers modulo 23
```

:::

2. 有了這個性質就可以回去參考一下maple在github上的說明，我們要解決的問題是$hint=Gflag$中的flag到底是甚麼，如果是像前面舉例的那樣($p=23/a=5/b=1/order=31$)很小的order，其實只要直接算`discrete_log(K, G, operation='+')`就可以了，範例如下，可以看到我先定義`K = E(19, 3)`，算出`discete log=28`，事後驗證也證明$K=28G$。但是，像題目中這樣這麼大的order，如果要計算discrete_log的話會非常非常久的時間，總之我先往smooth order的方向思考，也就是說order被factor後其實是由好幾個小的prime所組成，我是直接調整b這個不會被Elliptic Curve Multiplication運算使用到的參數(代表其他參數p, a要照舊)，然後factor曲線的order看夠不夠smooth，但這樣找也一樣要非常非常久，或者說找到的b所得到的order都不夠smooth，最大的prime都還是超過2^{65}(e.g. 範例如下)

```
>>> G = E.gen(0)
>>> print(G)
(15 : 1 : 1)
>>> K = E(19, 3)
>>> discrete_log(K, G, operation='+')
28
>>> 28 * G
(19 : 3 : 1)
```

```
sage: p = 0xffffffff00000001000000000000000000000000ffffffffffffffffffffffff
....: a = 0xffffffff00000001000000000000000000000000fffffffffffffffffffffffc
....: b = 56
sage: E = EllipticCurve(Zmod(p), [a, b])
sage: factor(E.order())
3^3 * 13967 * 67679 * 559243 * 11024719 * 127273871 * 1213196727283 * 171447020014729 *
27796463802665410393
sage: 27796463802665410393.bit_length()
65
```

3. 所以我開始朝maple的說明繼續前進，如果有invalid curve的問題就可以考慮用Pohlig–Hellman algorithm的方法求出flag為多少，就如同maple在background中提到的，我們選擇不同的b所產生的Elliptic Curve Order被factor後不一定有一個超大prime存在，因此我們就可以把問題簡化(n就是改變b之後取得的Elliptic Curve Order)

$$
\begin{gathered}
hint = flag * G \\
\hookrightarrow \frac{n}{prime}hint = flag' \times \frac{n}{prime}G \\
flag' = discrete\_log(\frac{n}{prime}hint, \frac{n}{prime}G, operation =' +')
\end{gathered}
$$

4. 等我們找到很多個b就可以找到很多不同的$flag'$，最後我們再用CRT找出真正的$flag$為何就可以了，也就是

$$
\begin{gathered}
flag \equiv flag' \ (mod\ prime_1) \\
flag \equiv flag'' \ (mod\ prime_2) \\
flag \equiv flag''' \ (mod\ prime_3) \\
\cdots
\end{gathered}
$$

所以重點在於要找到足夠多的$flag'$和$prime_n$組合

HW-signature_revenge

解題流程與思路

這一題沒有做出來，但跟一些朋友討論有得出解題的思路

1. 首先k_1, k_2是很特別的組合，他們符合以下式子

$$
\begin{gathered}
k_1 = magic_1 * 2^{128} + magic_2 \\
k_2 = magic_2 * 2^{128} + magic_1
\end{gathered}
$$

2. 所以我們可以改寫一下原本的公式

$$
\begin{gathered}
k_1 + tk_2 + u \equiv 0 \ (mod\ n) \\
\rightarrow magic_1 * 2^{128} + magic_2 + t(magic_2 * 2^{128} + magic_1) + u \equiv 0(mod\ n) \\
\rightarrow (t + 2^{128})magic_1 + (1 + t * 2^{128}) * magic_2 + u \equiv 0(mod\ n) \\
\rightarrow magic_1 + (1 + t * 2^{128})(t + 2^{128})^{-1}magic_2 + (t + 2^{128})^{-1}u \equiv 0(mod\ n)
\end{gathered}
$$

   此時新的t,u

$$
\begin{gathered}
new_t = (1 + t * 2^{128})(t + 2^{128})^{-1} \\
new_u = (t + 2^{128})^{-1}u
\end{gathered}
$$

3. 建立B matrix
4. 解LLL找最小的vector
5. 有了$magic_1, magic_2$之後就可以爆搜找d，並還原出原本的flag

HW-Power Anaylysis

Flag: `FLAG{W0ckAwocKaWoCka1}`

解題流程與思路

這一題全部都是刻出來的，也包含算correlation coefficient，後面才知道numpy有這東西，但反正根據老師上課的作法一步一步跟著做是絕對沒有問題的，包含以下步驟:

1. Preprocessing
   也就是把pt, ct, pm都按照簡報上的方式排列(各個trace的第一個byte都蒐集在一起，第二個byte都蒐集再一起...)
2. 計算和sbox key XOR的結果
3. 查表sbox
4. 計算hamming weight model
5. 計算和trace的correlation coefficient
6. 看哪一個結果的數值最大，並把index結果記錄下來算它的ascii
7. repeat以上操作後共可得16 bytes的flag

- 加速的方法:
  可以把整個trace的圖片plot出來看看，會發現題目給的json file是把整段加密的過程記錄下來，所以我們可以只取前一兩百個point就可以完成key的還原

Reference

LFSR

[Easiest way to perform modular matrix inversion with Python?]