

tags: ctf

# HW1 writeup

## LSB

server 會 decrypt 任何給它的 ciphertext，但每次只會回傳 plaintext % 3 的結果回來

因為每次都會回傳 % 3 的結果回來，因此可以做 LSB Oracle attack，只是變成三進制

一開始直接送 server 回傳給我們的 enc，來取得 plaintext 的最後一個 bit

要取得倒數第二個 bit 的話，我們必須計算  $(3^{-1})^e c$ , where  $c$  is ciphertext，因為  $(3^{-1})^e c$  decrypt 後為  $3^{-1}m$

Inference

$$3^{-1}m = 3y_2 + x_1 + 3^{-1}x_0$$

$$r = [3y_2 + x_1 + 3^{-1}x_0]_{\text{mod } n} \pmod{3}, \text{ where } r \text{ is return of server}$$

$$r = [3^{-1}x_0]_{\text{mod } n} + x_1 \pmod{3}$$

$$x_1 = r - [3^{-1}x_0]_{\text{mod } n} \pmod{3}$$

我們有  $x_0$ ，因此可以計算出倒數第二個 bit， $x_1$

接著利用  $(3^{-2})^e c \rightarrow 3^{-2}m$  來推算倒數第三個 bit，之後便以此類推解下去

我們保留  $x_1 + [3^{-1}x_0]_{\text{mod } n}$  的結果，因為在推算倒數第三個 bit 時，會需要減去  $[3^{-1}x_1 + 3^{-2}x_0]_{\text{mod } n}$ ，這樣只需要乘上  $[3^{-1}]_{\text{mod } n}$ ，不需要每回合都重新計算。因此在之後的推算，都會做這樣的紀錄：

$$x_i + [3^{-1}x_{i-1} + \dots + 3^{-i}x_0]，減少計算的時間$$

payload

```
1 from Crypto.Util.number import *
2 from pwn import *
3 from sage.all import *
4
5 p = remote( 'edu-ctf.zoolab.org', 10102 )
6
7 n = int( p.recvline().decode().strip() )
8 e = int( p.recvline().decode().strip() )
9 enc = int( p.recvline().decode().strip() )
10 print( n, e, enc )
11
12 inv_3 = inverse( 3, n )
13 i = 0
14 b = 0
15 pt = 0
16 while True:
17     # calculate oracle
18     oracle = ( pow( inv_3, e * i, n ) * enc ) % n
19     p.sendline( str( oracle ).encode() )
20
21     # calculate i-th bit
```

```

22     r = int( p.recvline().decode().strip() )
23     x_i = ( r - ( inv_3 * b ) % n ) % 3
24
25     # record to use for next bit
26     b = ( inv_3 * b + x_i ) % n
27
28     # restore
29     pt += x_i * ( 3 ** i )
30     print( r, x_i, i, pt )
31
32     i += 1
33
34     m = long_to_bytes( pt )
35     if b'flag{' in m or b'FLAG{' in m:
36         print( m )
37         break

```

## XOR-revenge

一開始我的想法是，因為我們有  $\text{len(flag)} + 70$  個輸出，其中前  $\text{len(flag)}$  個是被汙染過的，但我們還有 70 個 LFSR 的輸出，這樣足夠我們列出 64 個方程式，可以解聯立拿到 initial state。

但題目 LFSR 的形式和上課時教的不太一樣，因此我查了一下 wiki，發現上課教的是 Fabonacci LFSRs，而題目所用的是 Galois LFSRs。

- Fabonacci LFSRs: 對硬體較友善的實作；Galois LFSRs: 軟體友善的實作

Galois LFSRs 的 Companion matrix

$$\begin{bmatrix} c_1 & 1 & 0 & \dots & 0 & 0 & 0 \\ c_2 & 0 & 1 & \dots & 0 & 0 & 0 \\ c_3 & 0 & 0 & \dots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ c_{n-1} & 0 & 0 & \dots & 0 & 0 & 1 \\ 1 & 0 & 0 & \dots & 0 & 0 & 0 \end{bmatrix}$$

有了 Galois LFSRs 的 Companion matrix，我們便可以列出 64 個方程式，來解聯立求得 initial state

因為 output.txt 前  $\text{len(flag)}$  個是與 flag xor 的結果，因此我們跳過這些，拿後面 64 個 output。

在題目中，取得每個 output 前，都會先 `getbit()` 36 次，再 `getbit()` 一次，並當作 output，因此我們可以將這樣的行為模式寫成數學式：

$C^{37} * a = a'$ , where  $C$  is Companion matrix,  $a$  is initial state，並拿  $a'$  最高位的 bit 當作 output。

因此每個 output 都是先乘上  $C^{37}$  後，拿最高位的 bit 當作 output。

於是我們可以列出聯立方程，反解 initial state，有了 initial state 後，利用這個 initial state 來產生前  $\text{len(flag)}$  個 output，並和 output.txt 中的結果 xor 反解出 flag

payload

因為在 sage 中， $\wedge$  代表的是 python 中的  $**$ (次方)，我 google 後發現 sage 中的 xor 是  $\wedge\wedge$ ，於是我將  $\wedge$  替換成  $\wedge\wedge$ ，但卻還是會報錯，因此我將 xor 還原 flag 的 code 獨立出來，在 python 執行。

```

1 from sage.all import *
2 from Crypto.Util.number import *
3
4 output = [1, 0, 1, 0, 0, 1, 1, 1, 1, 1, 0, 0, 1, 0, 1, 1, 1, 0, 1, 1, 0, 0,
0, 1, 0, 1, 1, 0, 0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 1, 1, 1, 1, 0, 0, 1,
0, 0, 1, 0, 0, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 1, 0, 0,
1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 0, 1, 1, 0, 1, 1, 1, 0, 0, 1,
0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1,
1, 1, 0, 1, 1, 1, 0, 1, 0, 0, 1, 1, 1, 1, 0, 1, 1, 0, 1, 0, 1, 0, 0, 1, 0,
0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 0, 0, 1, 1, 1, 0, 0, 1, 0,
0, 0, 1, 0, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 0, 1, 1, 0, 0, 0, 1, 0,
1, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1,
1, 0, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 1, 0, 1, 0, 0, 1, 0, 1, 0, 1, 0, 1, 0,
0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0,
1, 0, 1, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 1, 0, 0, 1, 1, 0, 0,
1, 0, 0, 1, 0, 1, 0, 0, 1, 0, 0, 0, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1,
1, 1, 1, 0, 1, 0, 0, 0, 1, 0, 0, 1, 1, 0, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 1,
0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 1, 0, 1, 1, 0, 0, 1, 0, 1, 0, 0,
1, 1, 1, 0, 0, 0, 0, 1, 0, 0, 1, 0, 1, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 0,
1, 0, 0, 0, 1, 1, 1, 1, 0]

5
6 taps_raw = f"{0x1da785fc480000001:0>64b}"
7
8 # create characteristic equation
9 P.<x> = PolynomialRing( GF(2) )
10 P = 0
11 for i, t in enumerate( taps_raw ):
12     if t == '1':
13         P += x**( 64 - i )
14
15 # create companion matrix
16 C = companion_matrix( P, format='left' )
17
18 # create simultaneous equations
19 M = Matrix( GF(2), 64 )
20 V = vector( GF(2), 64 )
21 D = C ** 37
22
23 for i in range( 64 ):
24     M[i] = ( D ** ( i + 1 + 336 ) )[0]
25     V[i] = output[336+i]
26
27 # calculate initial state
28 init_state = M ** -1 * V
29
30 # collect output from initial state
31 re_output = []
32 for _ in range( 336 ):
33     init_state = D * init_state
34     re_output.append( init_state[0] )
35
36 print( output[:] )
37 print( re_output[:] )

```

```

1  a = [1, 0, 1, 0, 0, 1, 1, 1, 1, 1, 0, 0, 1, 0, 1, 1, 1, 0, 1, 1, 0, 0, 0, 1,
      0, 1, 1, 0, 0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 1, 1, 1, 1, 0, 0, 1, 0, 0,
      1, 0, 0, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 1, 0, 0, 1, 1,
      0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 0, 1, 1, 0, 1, 1, 1, 0, 0, 1, 0, 0,
      1, 1, 0, 0, 1, 1, 0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 1, 1,
      0, 1, 1, 1, 0, 1, 0, 0, 1, 1, 1, 1, 0, 1, 1, 0, 1, 0, 1, 0, 0, 1, 0, 0, 0,
      1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 0,
2      0, 1, 1, 1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 1, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1,
      1, 0, 1, 1, 0, 0, 0, 1, 0, 1, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1,
      0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 1, 0, 1, 0,
      0, 1, 0, 1, 0, 1, 0, 1, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0,
      0, 1, 0, 0, 0, 0, 1, 0, 0, 1, 0, 1, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1,
      0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 1, 0, 0, 0, 1, 1, 1, 1,
      1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 0, 1, 0,
3      0, 0, 1, 0, 0, 1, 1, 0, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 1, 0,
      0, 0, 0, 1, 1, 1, 0, 0, 0, 1, 0, 1, 1, 0, 0, 1, 0, 1, 0, 0, 1, 1, 1, 0, 0,
      0, 0, 1, 0, 0, 1, 0, 1, 1, 0, 1, 0, 1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1,
      1, 1, 1, 0]
4
5  b = [1, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0,
      0, 0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 1, 1, 0, 0, 1, 1, 0, 1, 0, 1, 1, 0,
      1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 0, 0, 1, 0, 0, 1, 0, 1, 1, 1, 0,
      1, 1, 1, 1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 1, 1, 0,
      0, 0, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 1, 0, 1, 0, 0,
      0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 1, 1, 0,
      1, 1, 0, 0, 1, 1, 1, 0, 1, 0, 0, 0, 1, 0, 1,
6      0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1,
      1, 1, 0, 1, 0, 0, 1, 0, 0, 1, 1, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 1, 1,
      0, 1, 1, 0, 1, 1, 0, 1, 0, 1, 1, 1, 0, 0, 1, 1, 1, 0, 0, 1, 0, 0, 1, 1, 1,
      0, 1, 0, 0, 0, 1, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 1, 0, 0,
      1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0,
      0, 0, 1, 0, 1, 0, 1, 1, 0, 1, 1, 1, 1, 0, 0, 1, 0, 1, 1, 0, 0, 1, 1, 0, 1,
      1, 0, 0, 0, 0, 0, 0, 1, 1, 0, 1, 1, 0, 1, 1,
7      0, 1, 0, 1, 1, 0, 1, 1]
8
9  flag = []
10 for i in range( 336 ):
11     flag.append( a[i] ^ b[i] )
12
13 f = ""
14 for i in range( 42 ):
15     tmp = 0
16     for j in range( 8 ):
17         tmp |= flag[ 8 * i + j ] << ( 8 - j - 1 )
18     f += chr( tmp )
19 print( f )

```

# DH

首先，題目給了一個 1024 bit 的質數  $p$ ，並可以自行輸入  $g$ ，接著會過濾掉  $g = 1 \pmod{p}$  or  $g = p - 1 \pmod{p}$  的情況，並生成一個 random number  $a$ ，計算  $A = g^a \pmod{p}$ ，若  $A$  等於  $1 \pmod{p}$  or  $p - 1 \pmod{p}$  程式會終止；反之，生成另一個 random number  $b$ ，並計算  $(g^a)^b * \text{flag} \pmod{p}$  將 flag 加密，這樣的加密方式看起來是 Elgamal encryption

因為  $g$  是我們自行輸入的，所以我們可以輸入一個 order 很小的  $g$ ，ex. order = 2，但因為在程式當中，有排除掉  $g = 1 \pmod{p}$  or  $g = p - 1 \pmod{p}$  的輸入，因此 order = 2 是不可行的

因此我們這邊使用 order = 4 的  $g$ ，因為 order = 2 的  $g$  有 1 和  $p - 1$ ，所以我們可以確認看看  $p - 1$  有沒有平方根，如果有，將  $p - 1$  開根號，這樣我們就得到了一個 order = 4 的  $g = \sqrt{p - 1}$

Inference

$$p - 1 \equiv p - 1 \pmod{p}, (p - 1)^2 \equiv 1 \pmod{p} \\ (\sqrt{p - 1})^2 \equiv p - 1 \pmod{p}, (\sqrt{p - 1})^4 \equiv 1 \pmod{p}$$

因為  $g = \sqrt{p - 1}$ , order = 4，因此可以 bypass 第一個過濾  $g = 1 \pmod{p}$  or  $g = p - 1 \pmod{p}$ ，但在第二個過濾  $g^a \pmod{p} \equiv 1 \text{ or } p - 1$  有一半的機率被擋下來，因為當  $a$  為偶數的時候， $\pmod{p}$  的結果可能是 1 or  $p - 1$ ，但在  $a$  是奇數的情況下，便可以通過檢查，拿到用  $g = \sqrt{p - 1}$  加密的結果。

拿到加密的結果時，可以直接 long\_to\_bytes 看看是否是 flag，因為在計算  $(g^a)^b$  時，沒有做  $\pmod{p}$  的結果可能是 1 or  $p - 1$  的檢查，如果拿到的結果並不是 flag，就將其  $* g \pmod{p}$ ，因為當  $g$  的次方是 4 的倍數時，會同餘 1，就能得到  $1 * \text{flag} \pmod{p}$ ，拿到 flag

payload

```
1  from pwn import *
2  from sage.all import *
3  from Crypto.Util.number import long_to_bytes
4
5  enc = 0
6  while True:
7      r = remote( 'edu-ctf.zoolab.org', 10104 )
8      p = int( r.recvline().strip() )
9      print( 'p=', p )
10
11     # check whether p - 1 is square
12     if Mod( p - 1, p ).is_square():
13         # order 2 to order 4
14         g = Mod( p - 1, p ).sqrt()
15         print( 'g=', g )
16         # restart
17     else:
18         r.close()
19         continue
20
21     r.sendline( str( g ).encode() )
22     ret = r.recvline().strip()
23     # if a is even, restart
24     if b'Bad :(' in ret:
25         r.close()
26         continue
27
28     enc = int( ret )
29     print( 'enc=', enc )
```

```

30     break
31
32     flag = b''
33     while True:
34         flag = long_to_bytes( int( enc ) )
35         if b'flag{' in flag or b'FLAG{' in flag:
36             print( flag )
37             break
38         # if ab is not a multiple of 4
39         else:
40             enc = enc * g % p

```

## node

這題是一個 ECDLP(Elliptic Curve Discrete Logarithm Problem) 的題目，題目中使用 double and add 的方式，計算  $flag \cdot G$ ，並將  $G$  和  $flag \cdot G$  的結果給我們。

在投影片中有提到，一個橢圓曲線  $y^2 = x^3 + ax + b \bmod p$  如果它是 singular:

$4a^3 + 27b^2 = 0 \bmod p$  的話，ECDLP 可以被轉換為 DLP on  $(\mathbb{F}_p, \times)$  或是 DLP on  $(\mathbb{F}_p, +)$

其中，若橢圓曲線可以分解為  $y^2 = (x - \alpha)^2(x - \beta)$  (Node)，則可以定義一個

$\phi(P(x, y)) = \frac{y + \sqrt{\alpha - \beta}(x - \alpha)}{y - \sqrt{\alpha - \beta}(x - \alpha)}$ ，確認它是否有 homomorphism， $\phi(P + Q) = \phi(P) \times \phi(Q)$ ，若是 homomorphism，則可以化簡 ECDLP 到 DLP on  $(\mathbb{F}_p, \times)$ ， $\phi(dP) = \phi(P)^d$

因此先來確認題目所使用的橢圓曲線是否為 singular，其中橢圓曲線的參數為  $a = -3, b = 2$

$$4 * -3^3 + 27 * 2^2 \equiv 0 \bmod p$$

確認完是 singular 後，可以來計算橢圓曲線的根， $\alpha$  及  $\beta$ ，並定義上述的  $\phi(P(x, y))$ ，確認 homomorphism。

題目中的橢圓曲線是 singular 且 homomorphism，因此可以將 ECDLP 轉成 DLP on  $(\mathbb{F}_p, \times)$ ， $\phi(dP) = \phi(P)^d$  並用 sage 的 discrete\_log() 來得出 flag

payload

```

1  from collections import namedtuple
2  from sage.all import *
3  from Crypto.Util.number import long_to_bytes
4  from chal import *
5
6  p =
1439347494057702678080391095332416717831615681366794991423769071711253367841
7633573178282302940945362269687132727837373091481050096454083379083647152529
5291332255885782612535793955727295077649715977839675098393245636668277194569
9642843910855001472647561367694613650577664546895409254178984894650442674939
55801
7  a = -3
8  b = 2
9
10 # check singular
11 singular = 4 * a**3 + 27 * b**2 % p
12 print( singular == 0 )
13
14 # calculate alpha and beta
15 F.<x> = PolynomialRing( Zmod( p ) )
16 roots = ( x**3 + a*x + b ).roots()

```

```

17 print( roots )
18
19 # validate alpha and beta
20 alpha = Zmod( p )( roots[0][1] )
21 beta = Zmod( p )( roots[0][0] )
22 res1 = ( alpha**3 + a*alpha + b )
23 res2 = ( beta**3 + a*beta + b )
24 print( res1, res2 )
25
26 def phi( P ):
27     return ( P.y + ( alpha - beta ).sqrt() * ( P.x - alpha ) ) / ( P.y - (
alpha - beta ).sqrt() * ( P.x - alpha ) )
28
29 x, y =
1018060571407808505447145304436447838257851670751471959006969666283489444474
9208525254009067924130172134098597551922414433142547762838657401604035864875
2353263802400527250163297781189749285392087154377684890287451078937692380556
1921269716690690156736626355614257355937957438521412327110661815422506703872
03333,
2107087706104714044822399433786361530649941274328852484740588692929521276499
9318872250771845966630538832460153205159221566590942573559588219757767072634
0725646459999590846534514050370793114900897670107649554189296242764912800345
7815036358401291333758803508050942113922971057834226101744135304443709297711
9013
30 G = Point( x, y )
31 B = Point( x =
9801549593290707686409625840798896200737632884989981025032200232562535973592
2937686533359455570369291999900476297694445557845368802830788062976760815467
2396612831570944251853375405788428518434971777806024153227062264262655158466
3337920374458882948817604579460285884786440213715075196182653652426530813993
4971, y =
8716613605429927265853459298243036167552031920609949999252923766393524661756
1944716447831162561604277568397630920048376392806047558420891922813475124718
9678890743220617473417803689224253960614688514601858619644323924085617695884
6852418786817138656457836292377782427939669809385755009193109198389309243686
4205 )
32
33 # whether have homomorphism or not
34 print( phi( point_addition( B, G ) ) == phi( B ) * phi( G ) )
35
36 flag = discrete_log( phi( B ), phi( G ) )
37 flag = long_to_bytes( flag )
38 print( flag )

```

## AES

題目有提供 plaintext、ciphertext 以及 power trace，我們可以透過這些資訊來執行 DPA attack，我的 DPA attack 是利用 plaintext

在 stm32f0\_aes.json 中，有提供 plaintext 及其對應的 power trace，首先將這些 plaintext 與 power trace，寫成兩個矩陣，size 分別為 (50, 16)、(50, 1806)，其中 50 為 records 的數目，16 為 plaintext 的長度，1806 為 power trace 的 sample points 數目

接著，拿出 plaintext matrix 的第一個 column，分別與一個 key byte(0x00-0xFF，256 種可能)，經過 S-Box 的中間值運算( $Sbox(p \oplus k)$ )，得到一個 50 x 256 的大矩陣，並將這個大矩陣經過 Power model(Hamming weight)

最後，將經過 Power model 的 output 的每個 column 與 power trace matrix 使用 Pearson's correlation coefficient 計算 correlation，得到一個 256 x 1806 的矩陣。這時候我們可以看，correlation 最高的是 0x00-0xFF 哪個數值，就猜它為這 AES 128 Key 的第一個 byte value

接著再拿出 plaintext matrix 的第二個 column，重複以上的計算，推出 key 的第二個 byte，之後依此類推，可以推算出 16 bytes 的 key

Pearson's correlation coefficient:

$$r = \frac{\sum_{i=1}^n ((x_i - \bar{x})(y_i - \bar{y}))}{\sqrt{\sum_{i=1}^n (x_i - \bar{x})^2 \sum_{i=1}^n (y_i - \bar{y})^2}}$$

payload

```
1 import json
2 import numpy as np
3 from tqdm import tqdm
4
5 sbox = np.array([ 0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5, 0x30,
6 0x01, 0x67, 0x2b, 0xfe, 0xd7, 0xab, 0x76,
7 0xca, 0x82, 0xc9, 0x7d, 0xfa, 0x59, 0x47, 0xf0, 0xad, 0xd4, 0xa2, 0xaf,
8 0x9c, 0xa4, 0x72, 0xc0,
9 0xb7, 0xfd, 0x93, 0x26, 0x36, 0x3f, 0xf7, 0xcc, 0x34, 0xa5, 0xe5, 0xf1,
10 0x71, 0xd8, 0x31, 0x15,
11 0x04, 0xc7, 0x23, 0xc3, 0x18, 0x96, 0x05, 0x9a, 0x07, 0x12, 0x80, 0xe2,
12 0xeb, 0x27, 0xb2, 0x75,
13 0x09, 0x83, 0x2c, 0x1a, 0x1b, 0x6e, 0x5a, 0xa0, 0x52, 0x3b, 0xd6, 0xb3,
14 0x29, 0xe3, 0x2f, 0x84,
15 0x53, 0xd1, 0x00, 0xed, 0x20, 0xfc, 0xb1, 0x5b, 0x6a, 0xcb, 0xbe, 0x39,
16 0x4a, 0x4c, 0x58, 0xcf,
17 0xd0, 0xef, 0xaa, 0xfb, 0x43, 0x4d, 0x33, 0x85, 0x45, 0xf9, 0x02, 0x7f,
18 0x50, 0x3c, 0x9f, 0xa8,
19 0x51, 0xa3, 0x40, 0x8f, 0x92, 0x9d, 0x38, 0xf5, 0xbc, 0xb6, 0xda, 0x21,
20 0x10, 0xff, 0xf3, 0xd2,
21 0xcd, 0x0c, 0x13, 0xec, 0x5f, 0x97, 0x44, 0x17, 0xc4, 0xa7, 0x7e, 0x3d,
22 0x64, 0x5d, 0x19, 0x73,
23 0x60, 0x81, 0x4f, 0xdc, 0x22, 0x2a, 0x90, 0x88, 0x46, 0xee, 0xb8, 0x14,
24 0xde, 0x5e, 0x0b, 0xdb,
25 0xe0, 0x32, 0x3a, 0x0a, 0x49, 0x06, 0x24, 0x5c, 0xc2, 0xd3, 0xac, 0x62,
26 0x91, 0x95, 0xe4, 0x79,
27 0xe7, 0xc8, 0x37, 0x6d, 0x8d, 0xd5, 0x4e, 0xa9, 0x6c, 0x56, 0xf4, 0xea,
28 0x65, 0x7a, 0xae, 0x08,
29 0xba, 0x78, 0x25, 0x2e, 0x1c, 0xa6, 0xb4, 0xc6, 0xe8, 0xdd, 0x74, 0x1f,
30 0x4b, 0xbd, 0x8b, 0x8a,
31 0x70, 0x3e, 0xb5, 0x66, 0x48, 0x03, 0xf6, 0x0e, 0x61, 0x35, 0x57, 0xb9,
32 0x86, 0xc1, 0x1d, 0x9e,
33 0xe1, 0xf8, 0x98, 0x11, 0x69, 0xd9, 0x8e, 0x94, 0x9b, 0x1e, 0x87, 0xe9,
34 0xce, 0x55, 0x28, 0xdf,
35 0x8c, 0xa1, 0x89, 0x0d, 0xbf, 0xe6, 0x42, 0x68, 0x41, 0x99, 0x2d, 0x0f,
36 0xb0, 0x54, 0xbb, 0x16 ])
```

```
21
22 hamming_weight = np.array([
23 0,1,1,2,1,2,2,3,1,2,2,3,2,3,3,4,
24 1,2,2,3,2,3,3,4,2,3,3,4,3,4,4,5,
25 1,2,2,3,2,3,3,4,2,3,3,4,3,4,4,5,
26 2,3,3,4,3,4,4,5,3,4,4,5,4,5,5,6,
27 1,2,2,3,2,3,3,4,2,3,3,4,3,4,4,5,
```



```

28 2,3,3,4,3,4,4,5,3,4,4,5,4,5,5,6,
29 2,3,3,4,3,4,4,5,3,4,4,5,4,5,5,6,
30 3,4,4,5,4,5,5,6,4,5,5,6,5,6,6,7,
31 1,2,2,3,2,3,3,4,2,3,3,4,3,4,4,5,
32 2,3,3,4,3,4,4,5,3,4,4,5,4,5,5,6,
33 2,3,3,4,3,4,4,5,3,4,4,5,4,5,5,6,
34 3,4,4,5,4,5,5,6,4,5,5,6,5,6,6,7,
35 2,3,3,4,3,4,4,5,3,4,4,5,4,5,5,6,
36 3,4,4,5,4,5,5,6,4,5,5,6,5,6,6,7,
37 3,4,4,5,4,5,5,6,4,5,5,6,5,6,6,7,
38 4,5,5,6,5,6,6,7,5,6,6,7,6,7,7,8
39 ], np.uint8)
40
41 def corr_coef( x, y ):
42     #print( x.shape, y.shape )
43     x_bar = np.mean( x )
44     y_bar = np.mean( y )
45
46     n, d1, d2 = 0., 0., 0.
47     for i in range( 50 ):
48         n += ( x[i] - x_bar ) * ( y[i] - y_bar )
49         d1 += ( x[i] - x_bar )**2
50         d2 += ( y[i] - y_bar )**2
51     return n / ( ( d1 * d2 )**(0.5) )
52
53 # read plaintext and power trace
54 f = open('stm32f0_aes.json')
55 datas = json.load(f)
56 f.close()
57
58 pt, pm = [], []
59 for data in datas:
60     pt.append( data['pt'] )
61     pm.append( data['pm'] )
62
63 pt, pm = np.array( pt ), np.array( pm )
64 #print( pt.shape, pm.shape )
65
66 keys = np.zeros( 256, np.uint8 )
67 for i in range( 256 ):
68     keys[i] = i
69 #print( keys )
70
71 arr = np.zeros( ( 16, 50, 256 ), np.uint8 )
72 for i in range( 16 ):
73     for j in range( 256 ):
74         arr[i, :, j] = hamming_weight[ sbox[ pt[:, i] ^ keys[j] ] ]
75         #print( arr[i, :, j] )
76 #print( arr )
77
78 ret = np.zeros( ( 16, 256, 1806 ), np.float32 )
79
80 r_keys = ""
81 for i in tqdm( range( 16 ) ):
82     for j in range( 256 ):
83         for k in range( 1806 ):
84             ret[i, j, k] = corr_coef( arr[i, :, j], pm[:, k] )
85             #print( np.argmax( ret[i, :, :] ) // 1806 )

```

```
86     r_keys += chr( np.argmax( ret[i, :, :] ) // 1806 )
87
88     print( r_keys )
```