CS HW1 Writeup

B08901164 林霈瑀
Username: lpy

1.XOR-Revenge

We are given a mask = 0x1da785fc480000001 = {m0, m1, …m63, m64}. Also, we have a state = {s0, s1, …, s63}.

The recurrence relationship:

$$\begin{cases} sk' = sk-1 \oplus (mk\, s63) & 1 \leq k \leq 63 \\ sk' = s63 & k = 0 \end{cases}$$

Therefore, we know the matrix relationship(under modulo 2) would look like:

$$M \cdot S = \begin{bmatrix} 0 & 0 & \dots & 0 & m0 \\ 1 & 0 & 0 & 0 & m1 \\ 0 & 1 & 0 & 0 & m2 \\ 0 & 0 & \dots & 0 & \dots \\ 0 & 0 & \dots & 1 & m63 \end{bmatrix} \begin{bmatrix} s0 \\ s1 \\ s2 \\ \dots \\ s63 \end{bmatrix} = \begin{bmatrix} s0' \\ s1' \\ s2' \\ \dots \\ s63' \end{bmatrix} = S'$$

The output of 1 round(under modulo 2):

$$\begin{bmatrix} 0 & 0 & \dots & 1 & m63 \end{bmatrix} \begin{bmatrix} s0 \\ s1 \\ \dots \\ s62 \\ s63 \end{bmatrix} = s63'$$

Therefore, we can calculate $\left\{M^{36}, M^{(36+37)}, M^{(36+37\cdot 2)} \dots\right\}$, get the last row of these matrices and form a system of linear equations(under modulo 2).

Output = {o0, o2, …}

$$\begin{bmatrix} M^{36}[63] \\ M^{(36+37)}[63] \\ M^{(36+37\cdot 2)}[63] \\ M^{(36+37\cdot 3)}[63] \\ \dots \end{bmatrix} \begin{bmatrix} s0 \\ s1 \\ s2 \\ \dots \\ s63 \end{bmatrix} = \begin{bmatrix} o0 \\ o1 \\ o2 \\ \dots \\ \dots \end{bmatrix}$$

Then, we can use sage to solve the system for the last 70 output bits, get the initial state, regenerate len(output) bits and get our flag.

Code:

```
from sage.all import *

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, 0, 1, 0, 1, 1, 0, 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, 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, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 1,
1, 1, 0]
MASK = 0x1da785fc480000001
state = 0
print("output: ", output[-70:])

def create_relation_matrix(R, number):
   global MASK
   mask = MASK
   rows = []
   for i in range(64):
       v = [0 for i in range(64)]
       v[63] = mask & 1
       mask >>= 1
       if i != 0:
           v[i-1] = 1
       rows.append(v)
   A = matrix(R, rows)
   I = matrix.identity(R, 64)
   for _ in range(number):
       I *= A
   return I

def get_last_row(M):
   return M[63]

def getbit():
   global state
   state <<= 1
   if state & (1 << 64):
       state ^= MASK
       return 1
   return 0

R = IntegerModRing(2)
M = matrix.identity(R, 64)

k = 0
rows = []
A = create_relation_matrix(R, 36)
M = A * M
v = get_last_row(M)
rows.append(v)
k += 1

A = create_relation_matrix(R, 37)
while k < len(output):
   M = A * M
   v = get_last_row(M)
   rows.append(v)
   k += 1
```

```
b = vector(R, output[-70:])
M = Matrix(R, rows[-70:])
seed = M.solve_right(b)

seed = list(seed)
for i in range(len(seed)):
   state += (int(seed[i]) << i)

out = []
for i in range(len(output)):
   for __ in range(36):
       getbit()
   a = getbit()
   out.append(a)
   output[i] ^= a

count = 0
flag_str = ""
for i in range(len(output)-70):
   val = output[i] << (7 - (i % 8))
   count += val
   if i % 8 == 7:
       flag_str += chr(count)
       count = 0

print("output: ", out[-70:])
print(flag_str)
```

Discussed with: b08901162

2.DH

Using factordb to factor p - 1, we can see that there are some small subgroups to exploit:

14299679064942095422216472867759219807100753053992282482112627127052320797011051556458400819572: Factorize!

Result:		
status (?)	digits	number
CF *	309 (show)	1429967906...00<309> = 2^2 · 3 · 5^2 · 4766559688...41<306>

Choose 5 as the order of the subgroup we want to generate. Find the generator for this subgroup using the following code:

```
p =
142996790649420954222164728677592198071007530539922824821126271270523207970110515564584008195723642124899896463843343427510806813124830678075983060663170191187324854031294041875387916499471130750967026466292812838422126149086627079858637978341343164315392615344366312548410846471761199219711056362616042332301
factor = 5
g = 1
while g == 1 or g == (p-1):
   g = pow(getRandomInteger(1024), (p-1)//factor, p)
print(g)
```

We get the corresponding ciphertext after manually pasting our generator to terminal:

```
~/Uni/cs/hw1/DH                                                12s  cnsenv Py  17:11:26
❯ nc edu-ctf.zoolab.org 10104
142996790649420954222164728677592198071007530539922824821126271270523207970110515
5645840081957236421248998964638433434275108068131248306780759830606631701911873
2485403129404187538791649947113075096702646629281283842212614908662707985863797
8341343164315392615344366312548410846471761199219711056362616042332301
13404880262847890184021281690202767153260770172595834585245159935003641332026502
98524465229335682220403551915593759542739880570170393346962248400371163408241654
40664809623823038470661056872583825161764909755430814926778063276757100806160008
74569727389330553762575982028774810239876536279472660005131591591435
7738110138752399428694006971695853723056333680673244187451506571447569659708402
2303387805630528709148989993547802088672668484293479806057899048786424598121652
212856311984941141488912499949527799887821357441744087064396686273517416727727132
6031737186394442084751626619873260522784222964726081698365705444741841
```

Then, we search the subgroup that g generates(i = 0 ~4), and try to retrieve the original flag by doing: `flag = pow(g, -i, p)*ciphertext % p`

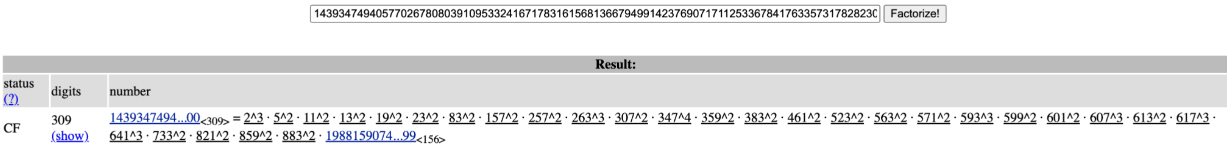
```
from Crypto.Util.number import long_to_bytes

p =
1429967906494209542221647286775921980710075305399228248211262712705232079701105155645840
0819572364212489989646384334342751080681312483067807598306066317019118732485403129404187
5387916499471130750967026466292812838422126149086627079858637978341343164315392615344366
312548410846471761199219711056362616042332301
g =
1340488026284789018402128169020276715326077017259583458524515993500364133202650298524465
2293356822204035519155937595427398805701703933469622484003711634082416544066480962382303
8470661056872583825161764909755430814926778063276757100806160008745697273893305537625759
82028774810239876536279472660005131591591435
ciphertext =
7738110138752399428694006971695853723056333680673244187451506571447569659708402230338780
5630528709148989993547802088672668484293479806057899048786424598121652212856311984941141
4889124999495277998878213574417440870643966862735174167277271326031737186394442084751626
61987326052278422296472608169836570544474184
factor=5

for i in range(factor):
   shared_key = pow(g, i, p)
   inv = pow(shared_key, -1, p)
   flag = (ciphertext * inv) % p
   flag = long_to_bytes(flag)
   print(flag)
```

```
~/Uni/cs/hw1/DH                                                cnsenv Py  17:44:39
❯ python3 findflag.py
b'n1\xc0\xab \xc0\x92N\x91\xa9z\\\x03,\xb4\xe3G\xf0"r\x84\x8fq\x0fO\x9e\xf7p\x94z4\xbf\x07\xd7\xf9D\xd9\x86\xddr\xb0N\xbe
\xab\x8a+\x0b/\x18I\xb8\x12\xe5\x98\xfc\xc7\xc9#.\x8fK\xfd\xe4\xc7\xaf+\xc1\xdaW\x7f\xb1\x967\xf7\xcb\x01\xf0\xe0>\xf5\\\
xab\x9c3\xa1\x92\x8aDu\x0e\xea0\xe1/\xc9\xc5&).\x1f\x1f<"\xfd8\xb4\x0c>\xbb7i\x11\x8bp^NS\xcef\t\xa9\x95\xe4h%\xac\x88H'
b'FLAG{M4yBe_i_N33d_70_checK_7he_0rDEr_OF_G}'
b'\xa2-3\xd3\x8a\xa1\x8b\xde\xfa\xbe,\xd0]\xfc\x94\xf7H\'\x13\xf5\xd0\xfa"\xfb\x0c*\x84\x88\xc5\xc5\xb1\xc6\xd0\xbd\r\x1e
{Nz\xe9\xf5\xday\xc8\x03e\xfd{\x1c$\xb1\x86}\x16kx\xc5\x11\xc1p(\x04\x19l)\x8c\xd8["\xec\xc3\x8b\xce\xde\xcd8\xbd\xbd\xba
u\x98\xa286\x98\xca\xc5M\x1eu&\xf4\xd2\x89h\xfb2\x80PJ\x89hVO\x16\x8c\xd0\x8fB\x1e5D\xe8\x08\x1d\x12\xa9\xa3x\xa1\x19\x86
\x9b\xd0T^\x14\x93'
b"\xab\xd4;\xd9\xe3g:\xa3\xe7\xe7\x0eq(\x8e/\xa1\xc3D]+S\x87a\xd3\xb5Z\xeb-\xe2\xc9\xde\xa8\xe0.d\xdf\x98\xf5p\xd2bF\x83\
x88:\x13\xf8S\xc2\xcd\x0f\xbfb\xdd\xben\x9b\xbb\xa0\x07i\xec\xf3\\e\x87\xc9l\xc4\xe7kw\xc6<\xfb\x9c\xccqi[*\x1am\x1e\xa3\
xc5|\xadd\xbb\xdc\x11l\x1e;`'\xf1\xfdp\x9d\x13[#A\x06?\x10l[\xb2h\ry\xd3y\xab\x81x\xc7T\x02Y/S\xebc}"
b'\xa6\xb3\xdeV9\xc5;\x85\x08\x03\x8a5wq\xae\xb8-\xfd\x8du\x11&H\xc4\x17`$w\x80\xc3\x87rmA{\x96V\xb3<a)x\xf6\x94\xb5!\\\x
c6\x92h\xbeM9\xeat\xf5p\xd4\xb0\xf6N\xd0\xd6\xa8\x16W\xd5:\x89\xd1%IJLJ-\xf6\xbfi\x9a\x0c2\xcb\xe0\x81\x91\xf8\xf2\xfb\x1
34,y\xb6g\xcaR\x08\xe2\xed\xc8$\xf9\xd1\xaa\x05\x02\x15\xaaR\xab\x13!\x12\x90\xcd\x02\xa4cH\xfe\x0b\xa0\xc4(\xf0\x99\xd2'
```

References: https://sasdf.github.io/ctf/tasks/2018/ais3Final/crypto/300-xorlnarmoni'akda/

3.Node

We find out that the curve is singular : $4a^3 + 27b^2 = 0$, with the type node. Therefore, we can transform the DLP to (Fp, x).
By checking p - 1 with factordb, we can see that it has a lot of small factors, meaning that it's a smooth curve. We can use the Pohlig Hellman method to solve the DLP by calling discrete_log() in sage.




```python
from math import sqrt
from sage.all import *
from collections import namedtuple
from Crypto.Util.number import long_to_bytes

x, y =
1018060571407808505447145304436447838257851670751471959006969666283489444474920852525400906792413017213409859755192241443314254776283865740160403586487523532638024005272501632977811897492853920871543776848902874510789376923805561921269716690690156736626355614257355937957438521412327110661815422506703872033 33,
2107087706104714044822399433786361530649941274328852484740588692929521276499931887225077184596663053883246015320515922156659094257355958821975776707263407256464599995908465345140503707931149008976701076495541892962427649128003457815036358401291333758803508050942113922971057834226101744135304443709297711 9013
fx =
9801549593290707686409625840798896200737632884989981025032200232562535973592293768653335945557036929199990047629769444555784536880283078806297676081546723966128315709442518533754057884285184349717778060241532270622642626551584663337920374458882948817604579460285884786440213715075196182653652426530813993 4971
fy =
8716613605429927265853459298243036167552031920609949999252923766393524661756194471644783116256160427756839763092004837639280604755842089192281347512471896788907432206174734178036892242539606146885146018586196443239240856176958846852418786817138656457836292377782427939669809385755009193109198389309243686 4205
p =
1439347494057702678080391095332416717831615681366794991423769071711253367841763357317828230294094536226968713272783737309148105009645408337908364715252952913322558857826125357939557272950776497159778396750983932456366682771945699642843910855001472647561367694613650577664546895409254178984894650442674939 55801

# y**2 = (x - 1)**2 * (x + 2)
a = 1
b = -2

Point = namedtuple("Point", "x y")
O = 'Origin'

def map_to_phi(P, a, b, p):
    upper = (P.y + Mod(a - b, p).sqrt()*(P.x - a))
    lower = P.y - Mod(a - b, p).sqrt()*(P.x - a)
    return (upper * inverse_mod(lower, p)) % p
```

```
G = Point(x, y)
F = Point(fx, fy)

base = map_to_phi(G, a, b, p)
ciphertext = map_to_phi(F, a, b, p)
flag = discrete_log(ciphertext, base)
flag = long_to_bytes(flag)
print(flag)
```

4.LSB

Take the first 3 LSBs for example:

To get x0:

$m = 3x1 + x0$

send: $c = m^e (mod\, n)$

return: $r1 = x0$

To get x1:

$3^{-1}m = 3x2 + x1 + 3^{-1}x0$

send: $3^{-e}c = \left(3^{-1}m\right)^e (mod\, n)$

return: $r2 - 3^{-1}x0 = x1$

To get x2:

$3^{-2}m = 3x3 + x2 + 3^{-1}x1 + 3^{-2}x0$

send: $3^{-2e}c = \left(3^{-2}m\right)^e (mod\, n)$

return: $r3 - \left(3^{-1}x1 + 3^{-2}x0\right) = x2$

It is also important to remember to mod n or mod 3 accordingly when implementing the algorithm.

Code:

```
from pwn import *
from Crypto.Util.number import long_to_bytes, getPrime
import math

r = remote('edu-ctf.zoolab.org', 10102)

n = int(r.recvline().decode().strip())
e = int(r.recvline().decode().strip())
c = int(r.recvline().decode().strip())

found_bits = []

count = int(math.log(n, 3))

for i in range(0, count):
   cipher = (c * pow(3, -e*i, n)) % n
   r.sendline(str(cipher).encode())
   lsb = int(r.recvline().decode().strip())
```

```
    subtract = 0
    for j in range(1, len(found_bits)+1):
        subtract += ((pow(3, -1*j, n) * found_bits[j-1]) % n)
    subtract %= n
    subtract %= 3
    lsb = (lsb - subtract) % 3
    found_bits = [lsb] + found_bits

found_bits.reverse()
m = 0
for i in range(len(found_bits)):
    m += pow(3, i) * found_bits[i]
    m %= n

flag = long_to_bytes(m)
print(flag)

#FLAG{LE4ST_519Nific4N7_Bu7_m0S7_1MporT4Nt}
```

5.AES

The given plaintexts and traces are in `stm32f0_aes.json`. We use CPA(Correlation power analysis) attack, as in the course slides to get back the key. We use SBOX(plaintext ^ key) as the intermediate value and Hamming Weight(number of 1s) as our power model.
Code:

```
import json
import numpy as np

POWER_TRACE = "./stm32f0_aes.json"

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
]

def getSboxValue(num):
    global sbox
    return sbox[num]

def HW(num):
    # hamming weight
    return bin(num).count("1")

def powerModel(M):
    # get the power model of our intermediate value matrix
    for i in range(len(M)):
        for j in range(len(M[i])):
            M[i][j] = HW(M[i][j])
    return M

def getIntermediate(k, plaintext):
    # get the intermediate value matrix for the kth byte
    M = []
    for i in range(50):
        v = []
        for j in range(256):
            byte = getSboxValue(plaintext[k][i] ^ j)
            v.append(byte)
        M.append(v)
    return M

def correlation_analysis(power_model, trace):
    power_model = np.array(power_model) # 50 * 256
    trace = np.array(trace) # 50 * 1806
    power_model = np.flip(np.rot90(power_model), 0)
    trace = np.flip(np.rot90(trace), 0)

    corr = []
    for i in range(256):
        v = power_model[i]
          x = []
        for j in range(1806):
            c = np.abs(np.corrcoef(v, trace[j])[0][1])
            x.append(c)
        corr.append(x)
    corr = np.array(corr)
    return corr

def getmaxindex(M):
    # get the index with the largest correlation coefficient
    max = 0
```

```
    mi = 0
    mj = 0
    for i in range(len(M)):
        for j in range(len(M[i])):
            if M[i][j] > max:
                max = M[i][j]
                mi = i
                mj = j
    return (mi, mj)

# parse real traces
json_file = open(POWER_TRACE, 'r')
power_trace = json.load(json_file)

plaintext = []
powermodel = []
for i in range(len(power_trace)):
    plaintext.append(power_trace[i]['pt'])
    powermodel.append(power_trace[i]['pm'])

plaintext = np.rot90(np.array(plaintext))
plaintext = np.flip(plaintext, 0)

key = [i for i in range(256)]

cipherkey = []
for k in range(16):
    M = getIntermediate(k, plaintext)
    pm = powerModel(M)
    corr = correlation_analysis(pm, powermodel)
    mi, mj = getmaxindex(corr)
    print(mi, mj)
    cipherkey.append(mi)

flag = "FLAG{"
for i in range(len(cipherkey)):
    flag += chr(cipherkey[i])
flag += "}"
print(flag)
```