EzSPN

[题目考点]

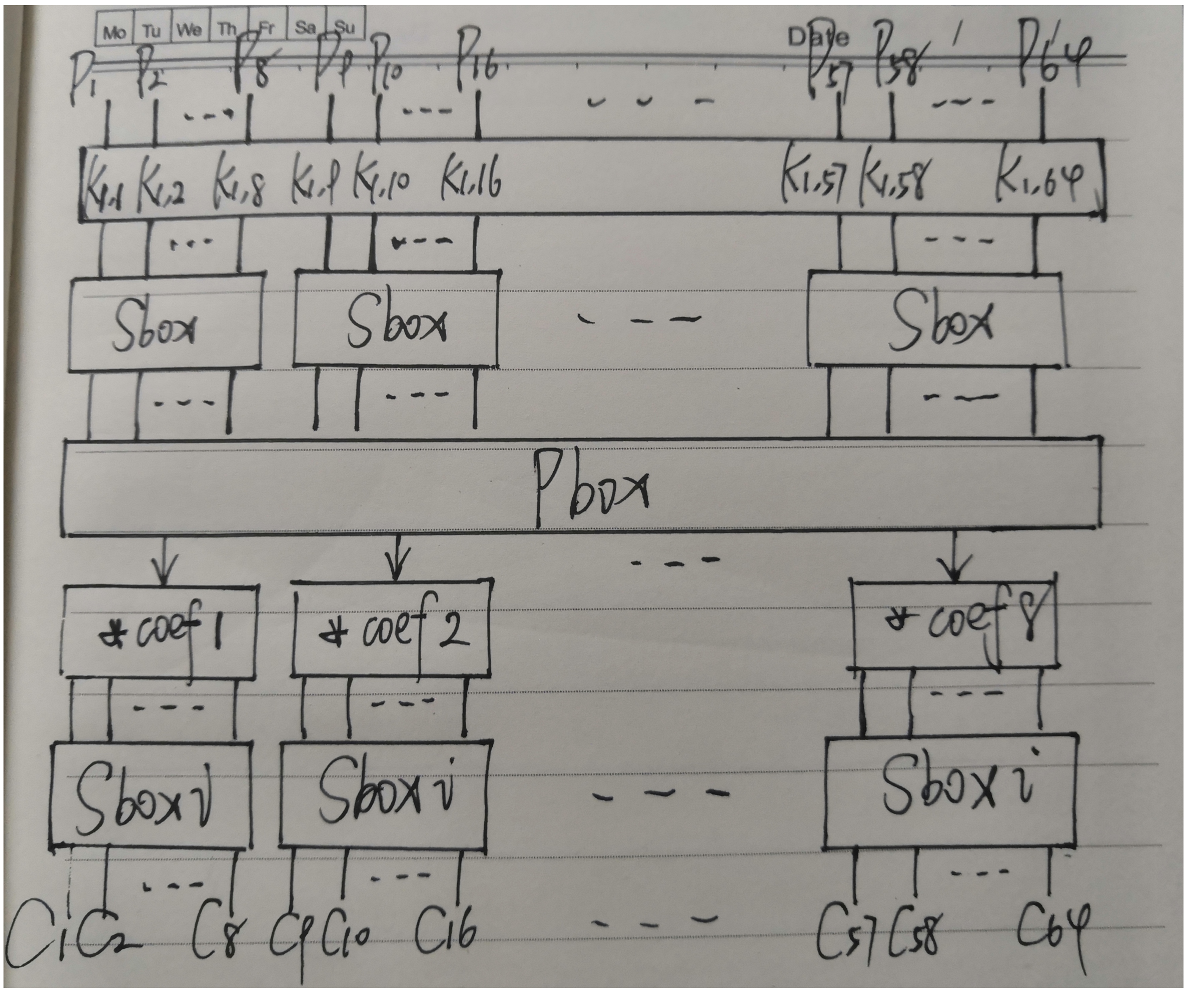
```
线性分析/差分分析
```

[flag]

```
npuctf{7inySPN-c4n-b3-3@si1y-cr4ck3d!!}
```

[题目分析]

加密所用SPN网络如下图所示



P_1 P_2 ... P_8 P_9 P_{10} ... P_{16} ... P_{57} P_{58} ... P_{64}
$K_{1,1}$ $K_{1,2}$ $K_{1,8}$ $K_{1,9}$ $K_{1,10}$ $K_{1,16}$ $K_{1,57}$ $K_{1,58}$ $K_{1,64}$
Sbox
Pbox
* coef 1
* coef 2
* coef 8
Sbox i
C_1 C_2 C_8 C_9 C_{10} C_{16} C_{57} C_{58} C_{64}

关于P盒的代码如下

```python
bits = [bin(x)[2:].rjust(8,'0') for x in blk[k:k+8]]
for i in range(8):
    res.append(int(''.join([x[(i+1) % 8] for x in bits]),2))
```

即为

$$\begin{bmatrix} 57 & 1 & 9 & 17 & 25 & 33 & 41 & 49 \\ 58 & 2 & 10 & 18 & 26 & 34 & 42 & 50 \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ 64 & 8 & 16 & 24 & 32 & 40 & 48 & 56 \end{bmatrix}$$

除P盒外还有在256模域上乘coef的操作 $(gcd(coef_i, 256) = 1)$

```
coef = [239, 163, 147, 71, 163, 75, 219, 73]
...
res[k:k+8] = [(coef[i] * res[k+i]) % 256 for i in range(8)]
```

上述两个变换均为线性且可简单求逆，因此主要着眼于S盒的线性估计（这里不赘述原理，基本思路在前置知识中给出）

但在找合适的线性逼近前，我们要知道以下两点：

- 每个线性逼近对应的密钥子集猜测空间不能过大（这里我们使得每个线性逼近表达式里的C_i满足$8j < i \leq 8(j+1)$，j在每次猜测下各自固定）
- 涉及线性估计的S盒不能过多（堆积引理$\epsilon_{1,2,\ldots,n} = 2^{n-1}\prod_{i=1}^{n}\epsilon_i$可知，总偏移量的绝对值会随着n的增大而减小，总偏移量过小会使得在猜测密钥时的成功率极低）

因此我们选择分别对应Sbox八个输出的八个Sbox的最大偏差线性估计，来进行八个第二轮子密钥的猜测，总的猜测空间为256*8

至于提供线性分析的数据足够多即可（exp中取10000，$bias = |\frac{N-5000}{10000}|$，很大概率下，$bias_{max}$对应的即为正确密钥）

[前置知识]

线性分析的基本方法即为寻找一个给定密码系统下，具有如下形式的有效线性表达式

$$P[i_1, \ldots, i_a] \oplus C[j_1, \ldots, j_b] = K[k_1, \ldots, k_c]$$

$(i_1, \ldots, i_a; j_1, \ldots, j_b; k_1, \ldots, k_c$均为固定比特位)

设$X_1, X_2, \ldots, X_k$为$\{0,1\}$上的独立随机变量，$Pr(X_i = 0) = p_i, Pr(X_i = 1) = 1 - p_i$，则用分布偏差来表示其概率分布，定义如下

$$\epsilon_i = p_i - \frac{1}{2}$$

[堆积引理] - $Pr(X_1 \oplus \ldots \oplus X_n = 0) = \frac{1}{2} + 2^{n-1}\prod_{i=1}^{n}\epsilon_i$，或表示为$\epsilon_{1,2,\ldots,n} = 2^{n-1}\prod_{i=1}^{n}\epsilon_i$.

利用堆积引理，我们即可以对每轮变换中偏差最大的线性逼近式进行组合，组合后的线性逼近式也将拥有最佳的偏差，即为要找的最佳线性逼近式

SPN结构种非线性结构只有S盒，因此只对其作偏差估计，具体方法如下（假设S盒为16*16）：

将输入的线性近似表示为$a_1 \cdot X_1 \oplus a_2 \cdot X_2 \oplus \ldots \oplus a_8 \cdot X_8 (a_i \in \{0,1\})$，对应的，我们也将输出线性近似表示为$b_1 \cdot Y_1 \oplus b_2 \cdot Y_2 \oplus \ldots \oplus b_8 \cdot Y_8 (b_i \in \{0,1\})$，穷举所有组合$(a,b)$对应的使得线性近似输入$\oplus$线性近似输出=0的偏差 (256*256)

偏差计算公式$\epsilon(a,b) = (N_L(a,b) - 128)/256$（$N_L(a,b)$为固定(a,b)下，满足上述条件的(X,Y)的个数）

我们的目的就是要找到$|\epsilon(a,b)|_{max}$对应的线性表达式，将其作为该单个S盒下的最佳线性估计

而找到最佳线性估计并组合后，我们就可以有效通过bias来进行子密钥子集的猜测（不正确的随机密钥会使得测试的明密文对中，满足线性表达式的概率趋于$\frac{1}{2}$，即偏差减小）

exp

```python
import os, sys
from binascii import hexlify, unhexlify
from pwn import *


SZ = 8
offset = [[0 for i in range(256)] for i in range(256)]  #Sbox线性估计offset
linearInput = []
sbox, sboxi, plain, cipher = [], [], [], []
enc_flag = None
coef = [15, 11, 155, 119, 11, 99, 83, 249]


def getData(ip, port):
    global enc_flag, sbox, sboxi
    io = remote(ip, port)
    sbox_str = io.recvline()
    sbox = eval(sbox_str)
    for i in range(256):
        sboxi.append(sbox.index(i))
    enc_flag = io.recvline().strip().decode("utf8")
    for i in range(10000):
        print("[+] Getting data...({}/10000)".format(i))
        pt = hexlify(os.urandom(8)).decode("utf8")
        plain.append(pt)
        io.sendline(pt)
        ct = io.recvline().strip().decode("utf8")
        cipher.append(ct)
    io.close()


def doxor(l1, l2):
    return [x[0] ^ x[1] for x in zip(l1, l2)]

# 线性层
def trans(blk):
    res = []
    for k in range(0, SZ, 8):
        cur = blk[k:k+8]
        cur = [(cur[i] * coef[i]) % 256 for i in range(8)]
        bits = [bin(x)[2:].rjust(8, '0') for x in cur]
        bits = bits[-1:] + bits[:-1]
        for i in range(8):
            res.append(int(''.join([x[i] for x in bits]), 2))
    return res


def bitxor(n, mask):
    bitlist = [int(x) for x in bin(n & mask)[2:]]
    return bitlist.count(1) % 2

# Sbox线性估计
def linearSbox():
    global linearInput
    for i in range(256):
        si = sbox[i]
        for j in range(256):
            for k in range(256):
                a = bitxor(i, j) # 线性估计输入
                b = bitxor(si, k) # 线性估计输出
                if a == b:
```

```
                offset[j][k] += 1
    for i in range(256):
        offset[i] = [abs(x - 128) / 256 for x in offset[i]]
    for linearOutput in range(256):
        cur = [x[linearOutput] for x in offset]
        linearInput.append(cur.index(max(cur)))


def calcOffset(pt, ct, j, guessed_key):  # 猜测第j段子密钥
    pt = list(unhexlify(pt))
    ct = list(unhexlify(ct))
    ct[j] ^= guessed_key
    ct[j] = sbox[ct[j]] # sbox即为sboxi的逆
    ct[j] = (ct[j] * coef[j]) % 256
    u1 = bitxor(pt[0], linearInput[1 << ((6 - j) % 8)])
    u2 = bitxor(ct[j], 0b10000000)
    if u1 == u2:
        return True
    else:
        return False


def linearAttack():
    key2 = []
    for i in range(8): # 第二轮子密钥的第i段
        count = [0 for _ in range(256)]
        for guessed_key in range(256):
            print("[+] Cracking key...({}-{})".format(i, guessed_key))
            for j in range(10000):
                if calcOffset(plain[j], cipher[j], i, guessed_key) == True:
                    count[guessed_key] += 1
        bias = [abs(x - 5000) / 10000 for x in count]
        key2.append(bias.index(max(bias)))
    return key2


def getkey(key2):
    ct = list(unhexlify(cipher[0]))
    pt = list(unhexlify(plain[0]))
    cur = doxor(ct, key2)
    cur = [sbox[x] for x in cur]
    cur = trans(cur)
    cur = [sboxi[x] for x in cur]
    key = doxor(cur, pt) + key2
    return key


def decrypt_block(ct, key):
    cur = doxor(ct, key[SZ:])
    cur = [sbox[x] for x in cur]
    cur = trans(cur)
    cur = [sboxi[x] for x in cur]
    cur = doxor(cur, key[:SZ])
    return cur


def decrypt(ct, key):
    pt = b''
    for i in range(0, len(ct), SZ * 2):
        block_ct = list(unhexlify(ct[i : i + SZ * 2]))
        block_pt = decrypt_block(block_ct, key)
        pt += bytes(block_pt)
    return pt


if __name__ == "__main__":
    getData(sys.argv[1], sys.argv[2])
    linearSbox()
    key2 = linearAttack()
    key = getkey(key2)
    print(key)
    flag = decrypt(enc_flag, key)
    print(flag)
```

More

线性分析也可处理SPN网络，但这里不赘述