

# 第二次作业报告

## 一、利用线性密码分析确定 SPN 分组解密算法轮密钥

### 1.1、实验分析

根据课本上的知识，我们可以知道，可以定义如下四个随机变量：

$$\begin{aligned}T_1 &= U_5^1 \oplus U_7^1 \oplus U_8^1 \oplus V_6^1 \\T_2 &= U_6^2 \oplus V_6^2 \oplus V_8^2 \\T_3 &= U_6^3 \oplus V_6^3 \oplus V_8^3 \\T_4 &= U_{14}^3 \oplus V_{14}^3 \oplus V_{16}^3\end{aligned}$$

并且 $T_1, T_2, T_1, T_3, T_4$ 分别具有偏差 $1/4, -1/4, -1/4$ 和 $-1/4$ 。根据堆积引理，我们可以知道四个随机变量的乘积具有偏差 $-1/32$ 。

之后我们把等式右边根据S盒公式展开异或后便可以得到如下等式：

$$T_1 \oplus T_2 \oplus T_3 \oplus T_4 = X_5 \oplus X_7 \oplus X_8 \oplus U_6^4 \oplus U_8^4 \oplus U_{14}^4 \oplus U_{16}^4 \oplus k_5^1 \oplus k_7^1 \oplus k_8^1 \oplus k_6^2 \oplus k_6^3 \oplus k_{14}^3 \oplus k_6^4 \oplus k_8^4 \oplus k_{14}^4 \oplus k_{16}^4$$

因为本次实验假设密钥比特固定，所以随机变量 $k_5^1 \oplus k_7^1 \oplus k_8^1 \oplus k_6^2 \oplus k_6^3 \oplus k_{14}^3 \oplus k_6^4 \oplus k_8^4 \oplus k_{14}^4 \oplus k_{16}^4$ 具有固定的值0或1，所以随机变量 $X_5 \oplus X_7 \oplus X_8 \oplus U_6^4 \oplus U_8^4 \oplus U_{14}^4 \oplus U_{16}^4$ 具有偏差 $-1/32$ 。而正因为具有偏差0的偏差，所以允许我们对此进行线性密码攻击。具体算法如下：

算法 3.2 线性攻击( $T, T, \pi_S^{-1}$ )

for  $(L_1, L_2) \leftarrow (0, 0)$  to  $(F, F)$

do Count $[L_1, L_2] \leftarrow 0$

for each  $(x, y) \in T$

do {  
for  $(L_1, L_2) \leftarrow (0, 0)$  to  $(F, F)$   
do {  
 $v_{<2>}^4 \leftarrow L_1 \oplus y_{<2>}$   
 $v_{<4>}^4 \leftarrow L_2 \oplus y_{<4>}$   
 $u_{<2>}^4 \leftarrow \pi_S^{-1}(v_{<2>}^4)$   
 $u_{<4>}^4 \leftarrow \pi_S^{-1}(v_{<4>}^4)$   
 $z \leftarrow x_5 \oplus x_7 \oplus x_8 \oplus u_6^4 \oplus u_8^4 \oplus u_{14}^4 \oplus u_{16}^4$   
if  $z = 0$   
then Count $[L_1, L_2] \leftarrow \text{Count}[L_1, L_2] + 1$

### 第 3 章 分组密码与高级加密标准

max  $\leftarrow -1$

for  $(L_1, L_2) \leftarrow (0, 0)$  to  $(F, F)$

do {  
Count $[L_1, L_2] \leftarrow |\text{Count}[L_1, L_2] - T/2|$   
if Count $[L_1, L_2] > \text{max}$   
then {  
max  $\leftarrow \text{Count}[L_1, L_2]$   
maxkey  $\leftarrow (L_1, L_2)$

output(maxkey)

而要想实现攻击，我们需要准备由同一个密钥生成的8000多对明密文，况且由课本上的算法得到的只是第五轮的二号和四号块密钥，要想得到一号和三号块密钥，我们需要使用得到的二号和四号密钥和别的线性表达式来逼近得到。翻阅资料后可知对于一号和三号块密钥我们可以构造如下线性表达式：

$$Z_1 = X_1 \oplus X_2 \oplus X_4 \oplus U_1^4 \oplus U_5^4 \oplus U_9^4 \oplus U_{13}^4$$

$$Z_2 = X_9 \oplus X_{10} \oplus X_{12} \oplus U_3^4 \oplus U_7^4 \oplus U_{11}^4 \oplus U_{15}^4$$

$$Z_* = Z_1 + Z_2$$

## 1.2、实验过程

实验过程可分为如下几步

- 使用 SPN 算法生成数据集
- 利用线性分析算法得到第五轮的二号和四号块密钥
- 使用得到的密钥接着计算一号和三号块密钥

接下来我将结合自己代码介绍实验过程，因为需要大量数据集，所以本次实验我用的是python语言。

首先要生成大约8000多个数据集，因为在上机作业中已经用C实验了 SPN 加密算法，所以可以把 SPN 算法封装为一个函数，输入一个明文，就可以得到一个密文。所以我们只要执行8000多次该函数就可以得到一个大数据集。

```
def generate_binary_numbers(n):
    # 生成从 0 到 2^n - 1 的十进制数
    number=8000 #数据集大小
    list=[]
    for i in range(2 ** n):
        # 将十进制数转换为 n 位的二进制数，使用 zfill(n) 来确保它是 n 位
        binary_str = bin(i)[2:].zfill(n)
        list.append(binary_str)
    list=random.sample(list,number) #从全部数据集中随机选取8000个数据
    return list

# 生成明文
plaintexts=generate_binary_numbers(16)
#保存明密文的字典
Plain_ciphertext_pairs={}
for i in range(len(plaintexts)):
    ciphertext=cal_ciphertext(plaintexts[i]) #把SPN算法封装成cal_ciphertext函数
    Plain_ciphertext_pairs.update({plaintexts[i]: ciphertext})
```

之后使用课本上的线性分析算法得到2和4号块密钥。

```
#定义要使用的一些变量
count_l2l4 = np.zeros((16, 16), dtype=int) #计数矩阵
count_l1l3 = np.zeros((16, 16), dtype=int) #计数矩阵
Max1=-1
Max2=-1
#密钥
Maxkey1=''
Maxkey2=''
Maxkey3=''
Maxkey4=''

#线性分析算法
for x,y in Plain_ciphertext_pairs.items():
    for i in range(16):
        for j in range(16):
            L1=intToBinary(i)
            L2=intToBinary(j)
            # 提取密文
            Y2=[int(Y[4]),int(Y[5]),int(Y[6]),int(Y[7])]
            Y4=[int(Y[12]), int(Y[13]), int(Y[14]), int(Y[15])]
            V2=[a ^ b for a, b in zip(L1, Y2)]
            V4=[a ^ b for a, b in zip(L2, Y4)]
            U2 = intToBinary(S_reverse[int(''.join(map(str, V2)), 2)])
            U4 = intToBinary(S_reverse[int(''.join(map(str, V4)), 2)])
            # 线性表达式
            Z=int(x[4])^int(x[6])^int(x[7])^U2[1]^U2[3]^U4[1]^U4[3]
            if(Z==0):
```

```

        count_1214[i][j]+=1

for i in range(16):
    for j in range(16):
        count_1214[i][j]=abs(count_1214[i][j]-T/2)
        if(count_1214[i][j]>Max1):
            Max1=count_1214[i][j]
            max2=intToBinary(i)
            max4=intToBinary(j)
            Maxkey2=''.join(map(str, max2))
            Maxkey4= ''.join(map(str, max4))

#保存密钥
key2= [int(char) for char in Maxkey2]
key4= [int(char) for char in Maxkey4]

```

接着使用得到的部分密钥和线性表达式来计算1号和3号块密钥。

```

#属于1和3的线性分析算法
for x,Y in Plain_ciphertext_pairs.items():
    for i in range(16):
        for j in range(16):
            L1 = intToBinary(i)
            L3 = intToBinary(j)

            # 提取密文
            Y1 = [int(Y[0]), int(Y[1]), int(Y[2]), int(Y[3])]
            Y3 = [int(Y[8]), int(Y[9]), int(Y[10]), int(Y[11])]
            Y2 = [int(Y[4]), int(Y[5]), int(Y[6]), int(Y[7])]
            Y4 = [int(Y[12]), int(Y[13]), int(Y[14]), int(Y[15])]

            V1 = [a ^ b for a, b in zip(L1, Y1)]
            V2 = [a ^ b for a, b in zip(key2, Y2)] #使用已经得到的密钥
            V3 = [a ^ b for a, b in zip(L3, Y3)]
            V4 = [a ^ b for a, b in zip(key4, Y4)] #使用已经得到的密钥

            U1 = intToBinary(S_reverse[int(''.join(map(str, V1)), 2)])
            U2 = intToBinary(S_reverse[int(''.join(map(str, V2)), 2)])
            U3 = intToBinary(S_reverse[int(''.join(map(str, V3)), 2)])
            U4 = intToBinary(S_reverse[int(''.join(map(str, V4)), 2)])

            #新的线性表达式
            z1 = int(X[0]) ^ int(X[1]) ^ int(X[3]) ^ (U1[0]) ^ (U2[0]) ^ (U3[0]) ^ (U4[0])
            if z1 == 0:
                count_1113[i][j] += 1

            z2 = int(X[8]) ^ int(X[9]) ^ int(X[11]) ^ (U1[2]) ^ (U2[2]) ^ (U3[2]) ^ (U4[2])
            if z2 == 0:
                count_1113[i][j] += 1

for i in range(16):
    for j in range(16):
        count_1113[i][j]=abs(count_1113[i][j]-T/2)
        if count_1113[i][j] > Max2:
            Max2 = count_1113[i][j]
            max1 = intToBinary(i)
            max3 = intToBinary(j)
            Maxkey1 = ''.join(map(str, max1))
            Maxkey3 = ''.join(map(str, max3))

```

最后拼接四个密钥输出即可

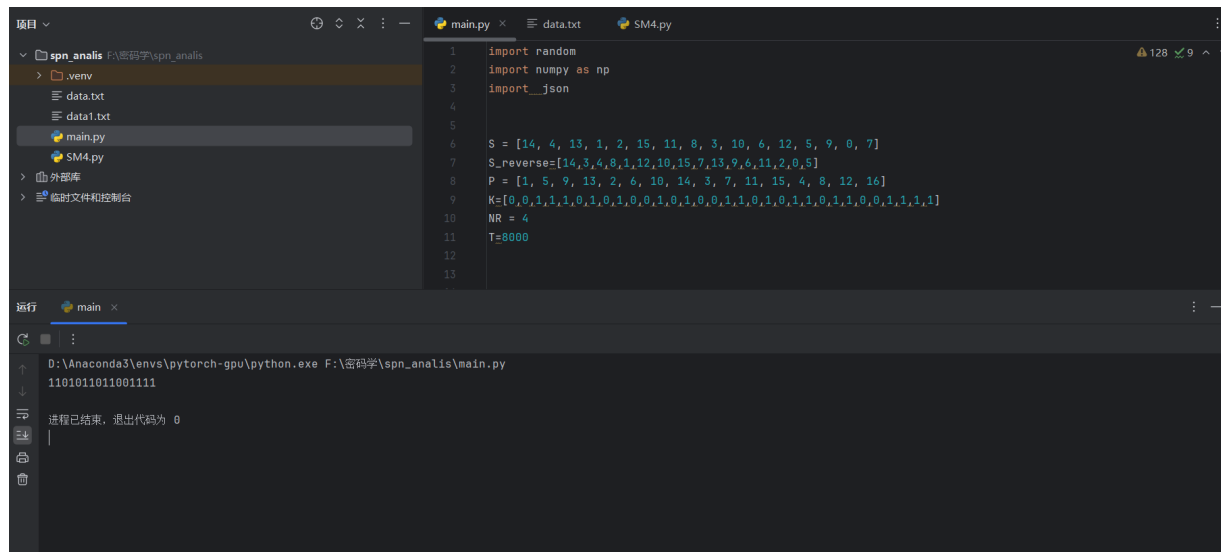
```

Maxkey='' +Maxkey1+Maxkey2+Maxkey3+Maxkey4
print(Maxkey)

```

## 1.3、实验结果

本次实验中我们使用32位密钥K=00111010100101001101011001111，通过运行代码可以发现在 $2^{16}$ 个数据集中随机选取8000个数据的情况下大部分情况中可以正确输出第五轮的全部密钥，即最后的16位密钥：k=1101011011001111。



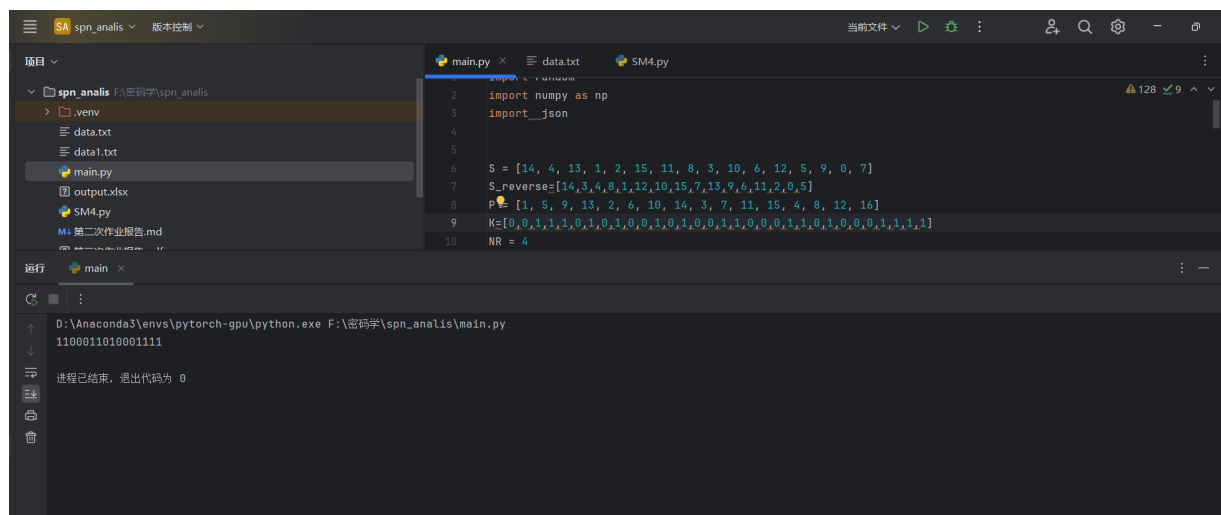
```
1 import random
2 import numpy as np
3 import json
4
5
6 S = [14, 4, 13, 1, 2, 15, 11, 8, 3, 10, 6, 12, 5, 9, 0, 7]
7 S_reverse=[14,3,4,8,1,12,10,15,7,13,9,6,11,2,0,5]
8 P = [1, 5, 9, 13, 2, 6, 10, 14, 3, 7, 11, 15, 4, 8, 12, 16]
9 K=[0,0,1,1,1,0,1,0,1,0,0,1,0,1,0,0,1,1,0,1,0,1,1,0,1,1,0,0,1,1,1,1]
10 NR = 4
11 T=8000
12
13
```

运行 main

D:\Anaconda3\envs\pytorch-gpu\python.exe F:\密码学\spn\_analis\main.py  
1101011011001111

进程已结束，退出代码为 0

当我换密钥K的时候也能正确输出



```
1 import random
2 import numpy as np
3 import json
4
5
6 S = [14, 4, 13, 1, 2, 15, 11, 8, 3, 10, 6, 12, 5, 9, 0, 7]
7 S_reverse=[14,3,4,8,1,12,10,15,7,13,9,6,11,2,0,5]
8 P = [1, 5, 9, 13, 2, 6, 10, 14, 3, 7, 11, 15, 4, 8, 12, 16]
9 K=[0,0,1,1,1,0,1,0,1,0,0,1,0,1,0,0,1,1,0,1,0,1,1,0,1,1,0,0,1,1,1,1]
10 NR = 4
```

运行 main

D:\Anaconda3\envs\pytorch-gpu\python.exe F:\密码学\spn\_analis\main.py  
1100011010001111

进程已结束，退出代码为 0

## 二、计算 SM4 的 SBox 差分分布表

### 2.1、实验分析

首先我们需要知道定义：

$$\Delta(x^1) = \{(x, x \oplus x^1) : x \in \{0, 1\}^m\}$$
$$N_D(x^1, y^1) = |\{x, x^* \in \Delta(x^1) : \pi_s(x) \oplus \pi_s(x^*) = y^1\}|$$

而差分分布表存的正是 $N_D(a, b)$ 的值，也就是当 $x^1 = a$ 时有多少种情况最后能计算出 $b$ 。

不同于 SPN 加密算法的四位输入和四位输出，SM4 加密算法的输入和输出都是八位，而除了这个，如下图所示，SM4 也有它独有的S盒映射。

S 盒中数据均采用 16 进制表示。

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0	d6	90	e9	fe	cc	e1	3d	b7	16	b6	14	c2	28	fb	2c	05
1	2b	67	9a	76	2a	be	04	c3	aa	44	13	26	49	86	06	99
2	9c	42	50	f4	91	ef	98	7a	33	54	0b	43	ed	cf	ac	62
3	e4	b3	1c	a9	c9	08	e8	95	80	df	94	fa	75	8f	3f	a6
4	47	07	a7	fc	f3	73	17	ba	83	59	3c	19	e6	85	4f	a8
5	68	6b	81	b2	71	64	da	8b	f8	eb	0f	4b	70	56	9d	35
6	1e	24	0e	5e	63	58	d1	a2	25	22	7c	3b	01	21	78	87
7	d4	00	46	57	9f	d3	27	52	4c	36	02	e7	a0	c4	c8	9e
8	ea	bf	8a	d2	40	c7	38	b5	a3	f7	f2	ce	f9	61	15	a1
9	e0	ae	5d	a4	9b	34	1a	55	ad	93	32	30	f5	8c	b1	e3
a	1d	f6	e2	2e	82	66	ca	60	c0	29	23	ab	0d	53	4e	6f
b	d5	db	37	45	de	fd	8e	2f	03	ff	6a	72	6d	6c	5b	51
c	8d	1b	af	92	bb	dd	bc	7f	11	d9	5c	41	1f	10	5a	d8
d	0a	c1	31	88	a5	cd	7b	bd	2d	74	d0	12	b8	e5	b4	b0
e	89	69	97	4a	0c	96	77	7e	65	b9	f1	09	c5	6e	c6	84
f	18	f0	7d	ec	3a	dc	4d	20	79	ee	5f	3e	d7	cb	39	48

例：输入‘ef’，则经 S 盒后的值为表中第 e 行和第 f 列的值， $Sbox('ef') = '84'$ 。

###

## 2.2、实验过程

整个实验过程可以分为以下几步

- 定义 sm4 的 S 盒
- 计算  $x^i = \alpha$  时的全部情况，并相应计数器加一
- 输出表格

同样的，本次实验我也是使用 python 语言完成的，我将结合代码描述实验过程。

首先，我们要定义给定的 S 盒，为了方便取用，我将 S 盒写成了相应的字典形式。

```
#定义S盒
s_box = {
    "00": "d6", "01": "90", "02": "e9", "03": "fe", "04": "cc", "05": "e1", "06": "3d", "07":
    "b7",
    "08": "16", "09": "b6", "0a": "14", "0b": "c2", "0c": "28", "0d": "fb", "0e": "2c", "0f":
    "05",
    "10": "2b", "11": "67", "12": "9a", "13": "76", "14": "2a", "15": "be", "16": "04", "17":
    "c3",
    "18": "aa", "19": "44", "1a": "13", "1b": "26", "1c": "49", "1d": "86", "1e": "06", "1f":
    "99",
    "20": "9c", "21": "42", "22": "50", "23": "f4", "24": "91", "25": "ef", "26": "98", "27":
    "7a",
    "28": "33", "29": "54", "2a": "0b", "2b": "43", "2c": "ed", "2d": "cf", "2e": "ac", "2f":
    "62",
    "30": "e4", "31": "b3", "32": "1c", "33": "a9", "34": "c9", "35": "08", "36": "e8", "37":
    "95",
    "38": "80", "39": "df", "3a": "94", "3b": "fa", "3c": "75", "3d": "8f", "3e": "3f", "3f":
    "a6",
    "40": "47", "41": "07", "42": "a7", "43": "fc", "44": "f3", "45": "73", "46": "17", "47":
    "ba",
    "48": "83", "49": "59", "4a": "3c", "4b": "19", "4c": "e6", "4d": "85", "4e": "4f", "4f":
    "a8",
    "50": "68", "51": "6b", "52": "81", "53": "b2", "54": "71", "55": "64", "56": "da", "57":
    "8b",
    "58": "f8", "59": "eb", "5a": "0f", "5b": "4b", "5c": "70", "5d": "56", "5e": "9d", "5f":
    "35",
```

```

        "60": "1e", "61": "24", "62": "0e", "63": "5e", "64": "63", "65": "58", "66": "d1", "67":
"a2",
        "68": "25", "69": "22", "6a": "7c", "6b": "3b", "6c": "01", "6d": "21", "6e": "78", "6f":
"87",
        "70": "d4", "71": "00", "72": "46", "73": "57", "74": "9f", "75": "d3", "76": "27", "77":
"52",
        "78": "4c", "79": "36", "7a": "02", "7b": "e7", "7c": "a0", "7d": "c4", "7e": "c8", "7f":
"9e",
        "80": "ea", "81": "bf", "82": "8a", "83": "d2", "84": "40", "85": "c7", "86": "38", "87":
"b5",
        "88": "a3", "89": "f7", "8a": "f2", "8b": "ce", "8c": "f9", "8d": "61", "8e": "15", "8f":
"a1",
        "90": "e0", "91": "ae", "92": "5d", "93": "a4", "94": "9b", "95": "34", "96": "1a", "97":
"55",
        "98": "ad", "99": "93", "9a": "32", "9b": "30", "9c": "f5", "9d": "8c", "9e": "b1", "9f":
"e3",
        "a0": "1d", "a1": "f6", "a2": "e2", "a3": "2e", "a4": "82", "a5": "66", "a6": "ca", "a7":
"60",
        "a8": "c0", "a9": "29", "aa": "23", "ab": "ab", "ac": "0d", "ad": "53", "ae": "4e", "af":
"6f",
        "b0": "d5", "b1": "db", "b2": "37", "b3": "45", "b4": "de", "b5": "fd", "b6": "8e", "b7":
"2f",
        "b8": "03", "b9": "ff", "ba": "6a", "bb": "72", "bc": "6d", "bd": "6c", "be": "5b", "bf":
"51",
        "c0": "8d", "c1": "1b", "c2": "af", "c3": "92", "c4": "bb", "c5": "dd", "c6": "bc", "c7":
"7f",
        "c8": "11", "c9": "d9", "ca": "5c", "cb": "41", "cc": "1f", "cd": "10", "ce": "5a", "cf":
"d8",
        "d0": "0a", "d1": "c1", "d2": "31", "d3": "88", "d4": "a5", "d5": "cd", "d6": "7b", "d7":
"bd",
        "d8": "2d", "d9": "74", "da": "d0", "db": "12", "dc": "b8", "dd": "e5", "de": "b4", "df":
"b0",
        "e0": "89", "e1": "69", "e2": "97", "e3": "4a", "e4": "0c", "e5": "96", "e6": "77", "e7":
"7e",
        "e8": "65", "e9": "b9", "ea": "f1", "eb": "09", "ec": "c5", "ed": "6e", "ee": "c6", "ef":
"84",
        "f0": "18", "f1": "f0", "f2": "7d", "f3": "ec", "f4": "3a", "f5": "dc", "f6": "4d", "f7":
"20",
        "f8": "79", "f9": "ee", "fa": "5f", "fb": "3e", "fc": "d7", "fd": "cb", "fe": "39", "ff":
"48"
    }
}

```

接着，我们要计算 $x^{\text{`}}$ 从0到255变化过程中的最后结果的值，并且要把表里相应的数字加1。

```

#计数数组也是差分分布表
dec_table = np.zeros((256, 256), dtype=int)

#计算函数
def cal_delta(delta_x):

    for x in range(256):
        x=decimal_to_8bit_binary(x) #把十进制整数转换成八位二进制字符串
        x_ = ''.join('1' if a != b else '0' for a, b in zip(delta_x, x)) #x和x`做异或运算得到x*
        x=binary_to_hex(x) #把八位二进制字符串转化成两位十六进制字符串
        x_=binary_to_hex(x_) #把八位二进制字符串转化成两位十六进制字符串
        y=hex_to_8bit_binary(s_box[x]) #取出y
        y_=hex_to_8bit_binary(s_box[x_]) #取出y`
        y_result = int(''.join('1' if a != b else '0' for a, b in zip(y,y_)),2) #计算y*
        x_result = int(delta_x, 2) #转换成十进制
        dec_table[x_result][y_result] += 1 #计数器加一

for i in range(256):
    i=decimal_to_8bit_binary(i) #把十进制整数转换成八位二进制字符串

```

```
cal_delta(i)
```

因为定义的时候使用的是十进制数，所以最后我们把表格稍微处理一下转成十六进制，然后输出就可以得到差分分布表。

```
np.set_printoptions(threshold=np.inf)

#把数组转成表格
df = pd.DataFrame(dec_table)

# 设置行和列的索引名称
df.index = [f'{i:02x}' for i in range(256)]
df.columns = [f'{i:02x}' for i in range(256)]
#保存差分分布表
df.to_excel('output.xlsx', index=True, header=True)
# 打印表格
print(df.to_string())
```

## 2.3、实验结果

运行程序，我们就可以得到差分分布表。

The image shows a Jupyter Notebook interface with a file explorer on the left and a code cell containing a 256x256 difference distribution table (DDF) for the SM4 cipher. The table is displayed in hexadecimal format, with both indices and values ranging from 00 to ff. The values are mostly 0, with some non-zero values scattered throughout, indicating the distribution of differences.

	00	01	02	03	04	05	06	07	08	09	0a	0b	0c	0d	0e	0f	10	11	12	13	14	15	16	17	18	19	1a	1b	1c	1d	1e	1f	20	21	22	23	24	25	26	27	28	29	2a	
00	256	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
01	0	2	2	2	2	0	0	2	0	2	2	0	0	0	2	2	0	2	0	2	0	2	2	0	0	2	2	0	0	2	2	0	2	2	2	2	0	2	2	0	2	0	2	0
02	0	0	2	2	2	2	0	2	2	0	2	2	2	0	0	0	2	2	0	2	0	2	0	2	0	2	2	0	2	2	0	2	2	2	2	0	2	2	0	2	0	2	0	
03	0	0	0	0	0	2	0	0	2	0	2	0	2	0	0	0	0	2	0	2	0	2	0	2	0	2	0	0	2	2	2	0	0	0	0	0	0	0	0	0	2	0	2	0
04	0	2	0	2	2	2	2	0	0	0	0	2	2	2	2	2	0	0	0	2	0	2	0	0	0	2	2	0	0	2	0	2	2	2	2	2	2	2	2	2	0	2	0	0
05	0	2	0	0	2	2	2	4	2	0	2	2	2	2	2	0	2	0	0	2	0	2	0	2	2	0	2	2	0	2	2	2	2	0	2	2	0	0	0	2	2	2	0	
06	0	2	0	0	2	0	2	2	2	0	0	2	2	2	0	0	0	0	0	2	2	0	2	0	0	2	2	2	2	2	2	2	0	0	2	2	0	2	2	2	2	0	0	
07	0	2	2	0	0	0	2	0	2	0	0	0	0	0	0	2	2	0	2	2	0	2	0	0	0	2	0	0	2	2	0	2	2	0	2	2	0	2	2	2	2	0	2	0
08	0	2	2	0	0	0	2	0	0	0	0	0	0	2	0	2	0	2	2	0	2	2	4	2	0	2	0	2	2	2	2	0	2	2	2	2	2	2	2	2	2	0	2	0
09	0	2	2	2	2	0	2	0	0	2	2	2	2	2	0	0	0	2	2	0	2	2	0	2	2	0	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	0	2	0
0a	0	2	2	2	2	2	0	2	0	2	2	2	0	0	0	0	2	2	2	0	0	2	0	2	0	2	2	2	2	2	2	2	2	2	0	2	2	2	2	2	0	2	2	2
0b	0	2	2	0	2	2	0	0	2	0	0	2	0	0	2	0	0	2	2	0	2	2	0	2	2	0	2	2	4	0	0	2	2	0	2	2	2	2	2	2	2	2	2	0
0c	0	0	2	0	2	2	2	0	2	0	0	2	0	0	2	2	0	2	2	2	2	2	0	2	2	2	0	0	2	2	0	2	2	2	2	2	2	2	2	2	2	2	2	0
0d	0	2	0	2	2	2	0	0	2	0	0	0	0	0	2	2	2	2	2	2	2	2	2	0	2	2	0	0	2	2	0	2	2	2	2	2	2	2	2	2	2	2	0	2
0e	0	2	0	0	2	2	0	2	2	2	2	2	0	0	0	2	0	2	0	0	2	0	0	0	0	0	0	0	2	0	2	2	2	2	2	2	2	2	2	2	2	2	0	
0f	0	0	0	2	2	2	2	0	0	0	0	0	2	2	2	2	0	2	2	0	0	2	2	0	2	2	2	2	2	2	2	2	2	0	2	2	2	0	0	0	2	2	0	
10	0	0	0	0	0	0	0	2	0	2	2	0	2	0	2	2	2	2	0	2	2	0	2	2	0	2	2	2	0	2	0	0	0	2	2	0	2	0	2	0	0	2	0	
11	0	2	2	2	2	0	0	0	0	2	0	0	2	2	0	2	2	0	2	0	2	2	0	2	2	0	2	2	0	0	2	2	0	0	2	0	0	2	2	0	2	2	0	
12	0	0	2	2	0	0	0	0	0	2	2	0	2	2	0	0	0	2	2	0	0	2	0	2	0	2	2	0	0	0	2	2	0	2	2	0	0	2	2	0	2	2	0	
13	0	0	4	2	2	2	2	2	2	2	0	2	2	2	2	2	0	0	2	2	0	2	2	2	2	0	2	2	0	0	2	2	2	2	2	2	2	2	2	2	2	0	2	2
14	0	2	0	0	0	0	0	0	2	0	2	2	2	2	2	2	2	0	2	0	2	2	0	0	0	2	2	2	2	2	2	0	0	2	2	2	2	2	2	2	2	2	0	
15	0	0	2	2	2	2	0	2	2	2	2	2	0	0	0	0	2	2	0	0	0	2	2	0	0	0	2	2	0	0	0	2	2	2	2	2	2	2	2	2	2	2	2	
16	0	0	2	2	0	2	0	2	0	0	2	0	2	0	2	2	2	2	0	2	0	2	0	0	0	0	2	2	2	2	2	0	2	2	0	2	2	0	2	2	0	0	0	
17	0	0	0	2	0	0	0	0	0	2	0	0	2	2	2	0	0	2	0	0	2	0	0	2	0	0	0	0	0	2	2	0	0	0	2	2	0	0	2	2	0	0	2	0
18	0	0	2	2	0	0	0	0	2	2	2	0	2	2	2	0	0	2	2	0	2	2	0	2	2	0	2	2	2	0	2	2	2	2	2	2	2	2	2	2	2	0	0	
19	0	2	0	2	2	0	2	0	0	0	0	2	2	0	0	2	0	2	0	0	2	0	2	0	2	0	2	0	2	0	2	2	0	0	2	2	2	2	2	2	2	0	0	
1a	0	0	2	0	2	2	2	2	0	2	2	0	2	0	2	2	0	2	0	0	2	2	0	2	2	0	2	2	0	0	2	2	0	0	2	0	0	2	2	2	2	0	0	