线性攻击算法

姓名：齐明杰　学号：2113997 班级：信安2班

1　算法原理

SPN (Substitution-Permutation Network) 是一种块密码结构，由多轮的替代 (substitution) 和置换 (permutation) 操作组成。这种加密方式有如下的特点：

1. **替代 (Substitution)**：这是通过 S-box 实现的，它是一个预定义的表，用于将输入映射到输出。S-box 的选择对于算法的安全性至关重要。
2. **置换 (Permutation)**：这是通过 P-box 实现的，它重新排列其输入位。
3. **密钥混合**：每一轮都会有一个子密钥与数据进行异或操作。

线性攻击是一种针对块密码的密码分析方法，目的是找到明文和密文之间的线性关系。这种关系可以用于减少恢复密钥所需的尝试次数。

不妨设最后一轮密钥$K = K_1K_2K_3K_4$，则

根据课本知识，K_2和K_4可以由以下线性分析链得出：

$$z_1 = x_5 \oplus x_7 \oplus x_8 \oplus u_6^4 \oplus u_8^4 \oplus u_{14}^4 \oplus u_{16}^4$$

另查阅资料，发现K_1和K_3可以由另一个线性分析链得出：

$$z_2 = x_1 \oplus x_2 \oplus x_4 \oplus u_1^4 \oplus u_5^4 \oplus u_9^4 \oplus u_{13}^4$$
$$z_3 = x_9 \oplus x_{10} \oplus x_{12} \oplus u_3^4 \oplus u_7^4 \oplus u_{11}^4 \oplus u_{15}^4$$
$$z^* = z_2 + z_3$$

2　代码

我使用*Python*进行编程，完成了两份代码的编写：

- **util.py**

这份代码主要编写SPN加密算法的实现，用于**产生密钥以及明文-密文对**。

```python
import random

l = 4   # 每组比特数
m = 4   # 组数
Nr = 4  # 轮数
Sbox = [0xE, 0x4, 0xD, 0x1, 0x2, 0xF, 0xB, 0x8,
        0x3, 0xA, 0x6, 0xC, 0x5, 0x9, 0x0, 0x7]    # S盒
Pbox = [1, 5, 9, 13, 2, 6, 10, 14, 3, 7, 11, 15, 4, 8, 12, 16]  # P盒


```

```python
def reSbox(Sbox):
    '''
    生成逆S盒
    param Sbox: S盒
    '''
    reSbox = [0 for _ in Sbox]
    for i in range(len(Sbox)):
        reSbox[Sbox[i]] = i
    return reSbox


def xor(w, K):
    '''
    异或运算
    param w: 二进制字符串
    param K: 二进制字符串
    '''
    assert len(w) == len(K)
    u = ''
    for _w, _k in zip(w, K):
        u += str(int(_w) ^ int(_k))
    return u

def sbox(u):
    '''
    S盒运算
    param u: 二进制字符串
    '''
    v = ''
    for i in range(m):
        ui = u[i * l : i * l + l]
        vi = format(Sbox[int(ui, 2)], '04b')
        v += vi
    return v

def pbox(v):
    '''
    P盒运算
    param v: 二进制字符串
    '''
    assert len(v) == l * m
    w = ''
    for i in range(len(v)):
        w += v[Pbox[i] - 1]
    return w

```

```python

def spn(plain, key):
    '''
    SPN加密算法
    param plain: 明文
    param key: 密钥
    '''
    keys = [key[i * Nr : i * Nr + l * m] for i in range(Nr + 1)]
    w = plain
    for r in range(Nr - 1):
        u = xor(w, keys[r])
        v = sbox(u)
        w = pbox(v)
    u = xor(w, keys[Nr - 1])
    v = sbox(u)
    y = xor(v, keys[Nr])
    return y


def generate_key(n):
    '''
    生成密钥
    param n: 密钥个数
    '''
    keys = []
    for _ in range(n):
        key = ''
        for _ in range(l * m + 4 * Nr):
            key += str(random.randint(0, 1))
        keys.append(key)
    return keys


def generate_data(T, key):
    '''
    生成明文-密文对
    param T: 明文-密文对个数
    '''
    data = []
    for _ in range(T):
        plain = bin(random.randint(0, 2 ** (l * m) - 1))[2:].zfill(l * m)
        y = spn(plain, key)
        data.append((plain, y))
    return data
```

代码解析:

1. `reSbox()`：计算逆 S-box，使得我们可以从输出值回到输入值。
2. `xor()`：这是一个简单的异或操作，用于两个字符串。
3. `sbox()`：应用 S-box 到输入字符串。
4. `pbox()`：应用 P-box 到输入字符串。
5. `spn()`：SPN 的主要实现，函数使用子密钥、S-box 和 P-box 对明文进行加密。
6. `generate_key()`：输入要生成的数量，生成密钥。
7. `generate_data()`：输入要生成的数量，生成随机的明文和对应的密文。

- **spn_re.py**

这份代码为核心代码，实现了对SPN算法最后一轮密钥的线性攻击。

```
from util import generate_data, generate_key, reSbox
import numpy as np
from concurrent.futures import ProcessPoolExecutor

Sbox = [0xE, 0x4, 0xD, 0x1, 0x2, 0xF, 0xB, 0x8,
        0x3, 0xA, 0x6, 0xC, 0x5, 0x9, 0x0, 0x7]     # S盒

reSbox = reSbox(Sbox)   # 逆S盒
T = 10000     # 每个密钥生成的明文-密文对个数
N = 50        # 待攻击密钥个数

def reverse_spn(data):
    '''
    线性攻击算法
    param data: 明文-密文对
    return: 最后一轮的密钥
    '''
    lens = len(data)
    counter = np.zeros((16, 16), dtype=int)
    for pair in data:
        x, y = pair
        y2, y4 = int(y[4:8], 2), int(y[12:], 2)
        x = [None] + [int(i) for i in x]    # 获取明文的每一位
        for L1 in range(16):
            for L2 in range(16):
                u2 = [int(i) for i in format(reSbox[L1 ^ y2], '04b')]
                u4 = [int(i) for i in format(reSbox[L2 ^ y4], '04b')]
                u = [None] * 5 + u2 + [None] * 4 + u4
                z = x[5] ^ x[7] ^ x[8] ^ u[6] ^ u[8] ^ u[14] ^ u[16]
                if z & 1 == 0:
                    counter[L1][L2] += 1
    counter = np.abs(counter - 0.5 * lens)
    k2, k4 = map(lambda x: x[0], np.where(counter == np.max(counter)))  # 找到最大值的位置
```

```
34
35      counter1 = np.zeros((16, 16), dtype=int)
36      counter2 = np.zeros((16, 16), dtype=int)
37      for pair in data:
38          x, y = pair
39          y1, y2, y3, y4 = int(y[:4], 2), int(y[4:8], 2), int(y[8:12], 2),
    int(y[12:], 2)
40          x = [None] + [int(i) for i in x]    # 获取明文的每一位
41          for L1 in range(16):
42              for L2 in range(16):
43                  u = [None] + [int(i) for i in format(reSbox[L1 ^ y1],
    '04b') + \
44                                                format(reSbox[k2 ^ y2],
    '04b') + \
45                                                format(reSbox[L2 ^ y3],
    '04b') + \
46                                                format(reSbox[k4 ^ y4],
    '04b')]
47                  z = x[1] ^ x[2] ^ x[4] ^ u[1] ^ u[5] ^ u[9] ^ u[13]
48                  if z & 1 == 0:
49                      counter1[L1][L2] += 1
50                  z = x[9] ^ x[10] ^ x[12] ^ u[3] ^ u[7] ^ u[11] ^ u[15]
51                  if z & 1 == 0:
52                      counter2[L1][L2] += 1
53      counter1 = np.abs(counter1 - 0.5 * lens)
54      counter2 = np.abs(counter2 - 0.5 * lens)
55      counter = counter1 + counter2
56      k1, k3 = map(lambda x: x[0], np.where(counter == np.max(counter)))  #
    找到最大值的位置
57
58      key = format(k1, '04b') + format(k2, '04b') + format(k3, '04b') +
    format(k4, '04b')
59      return key
60
61
62  def process_key(key):
63      '''
64      并行处理
65      param key: 密钥
66      '''
67      data = generate_data(T=T, key=key)
68      reverse_key = reverse_spn(data)
69      return key, reverse_key
70
71
72  def print_result(future):
```

```
    '''
    打印结果
    '''
    key, reverse_key = future.result()
    msg = 'success' if key[16:] == reverse_key else 'fail'
    key = ' '.join([key[i:i+4] for i in range(0, len(key), 4)])
    reverse_key = ' '.join([reverse_key[i:i+4] for i in range(0,
len(reverse_key), 4)])
    print(f'key = {key}\nreverse_key = {reverse_key}\nresult: {msg}\n')


if __name__ == '__main__':
    keys = generate_key(n=N)                    # 生成密钥
    with ProcessPoolExecutor() as executor:  # 多进程处理
        futures = [executor.submit(process_key, key) for key in keys]
        for future in futures:
            future.add_done_callback(print_result)
```

- **代码流程**:

1. **初始化参数和S盒**:
    - 初始化S盒（代替置换表）。
    - 计算逆S盒。
    - 设置明文-密文对个数`T`和待攻击密钥个数`N`。
2. **线性攻击算法（`reverse_spn`）**:
    - 首先，初始化一个计数器矩阵。
    - 遍历所有的明文-密文对。
    - 对于每个明文-密文对，遍历所有可能的L1和L2值，进行线性逼近。
    - 根据计算的线性逼近统计信息，得到可能的k2和k4。
    - 再次遍历所有的明文-密文对，使用已知的k2和k4，以及所有可能的L1和L2值来进行线性逼近。
    - 根据新的线性逼近统计信息，得到可能的k1和k3。
    - 最后，返回由k1, k2, k3, k4组成的密钥。
3. **主函数**:
    - 生成待攻击的密钥列表。
    - 使用多进程并行处理每个密钥。
    - 为每个进程任务添加回调函数以打印结果。

3 运行结果

我采用了**随机生成密钥**的方式，在本次我令`n=50`，即生成50组密钥来攻击，但由于速度较慢，我**使用了并行来加速**，最终结果如下：

```
key = 0000 0101 1100 1111 0011 0111 1111 0110
reverse_key = 0011 0111 1111 1011
result: fail

key = 0101 0001 0010 1110 1001 1010 0011 1010
reverse_key = 1001 1010 0011 1010
result: success

key = 1000 0111 1100 1100 1001 0100 1101 1100
reverse_key = 1001 1001 1101 1100
result: fail

key = 0111 1001 1101 0111 1011 0001 0000 0011
reverse_key = 1011 0001 0000 0011
result: success

key = 0011 1001 1010 1101 1000 1010 0100 1000
reverse_key = 1000 1010 0100 1000
result: success

key = 1100 1111 1010 0100 1011 1000 1011 0010
reverse_key = 1011 0101 1011 0010
result: fail

key = 1100 1110 0011 0110 0011 0101 0000 0000
reverse_key = 0011 0101 0000 0000
result: success

key = 0101 0101 1000 0000 1100 1001 1110 1010
reverse_key = 1100 1001 1110 1010
result: success

key = 0010 1110 1000 0111 1011 0101 0001 1100
reverse_key = 1011 0101 0001 1100
result: success

key = 0101 0111 0101 0010 0010 1011 0110 1100
reverse_key = 0010 1011 0110 1100
result: success

key = 0001 0000 1101 1110 1111 0001 1100 1000
```

```
reverse_key = 1111 0001 1100 1000
result: success

key = 0101 1101 1100 1001 0100 0111 0110 0101
reverse_key = 0100 0111 0110 0101
result: success

key = 1111 0010 1010 1001 0011 0011 1110 0010
reverse_key = 0011 0011 1110 0010
result: success

key = 0101 1101 0101 1001 1111 1110 0011 0000
reverse_key = 1111 1110 0011 0000
result: success

key = 0001 1111 1111 0000 0101 0111 1111 0100
reverse_key = 0101 0111 1111 0100
result: success

key = 1111 1110 0011 1001 1101 1000 0011 0110
reverse_key = 1101 0100 0001 0110
result: fail

key = 0001 0111 1001 1110 1001 0101 1000 1001
reverse_key = 1001 0101 1000 1001
result: success

key = 1101 1100 1011 0011 0100 0100 0110 1001
reverse_key = 0100 0110 0110 1001
result: fail

key = 0110 0001 0011 1110 1010 1010 1001 1010
reverse_key = 1010 1010 1001 1000
result: fail

key = 0100 1101 1000 1100 1111 0000 1100 1111
reverse_key = 1111 0000 1100 1111
result: success

key = 0011 0000 0011 1100 1000 1111 1001 0111
reverse_key = 1000 1111 1001 0111
result: success

key = 1100 1111 1011 0011 1110 1001 1111 0111
reverse_key = 1100 1000 1111 0011
result: fail
```

```
key = 0100 1110 0000 0101 0111 0011 1000 1100
reverse_key = 0111 0011 1000 1100
result: success

key = 0101 1111 0010 0111 1111 1000 0001 0011
reverse_key = 1111 1000 0001 0001
result: fail

key = 1111 1101 0100 0100 1001 0100 0000 0101
reverse_key = 1001 0100 0000 0101
result: success

key = 1010 0101 0101 1000 1010 1011 0000 0011
reverse_key = 1010 1011 0000 0011
result: success

key = 0100 0011 1011 1000 1000 0001 0010 1001
reverse_key = 1000 0001 0010 1001
result: success

key = 0011 1010 0001 0110 1000 0011 0000 1111
reverse_key = 1000 0011 0000 1111
result: success

key = 0010 1110 1001 0011 0011 1011 1000 0011
reverse_key = 0011 1011 1000 0011
result: success

key = 1110 1111 0101 0110 1011 0000 1100 1111
reverse_key = 1011 0000 1100 1111
result: success

key = 0000 0010 1001 0010 0101 0011 1101 1100
reverse_key = 0101 1101 1101 0010
result: fail

key = 1010 0100 0101 1110 0101 1101 1010 1110
reverse_key = 0101 1101 1010 1110
result: success

key = 1101 1000 1010 0111 1001 1111 0011 0101
reverse_key = 0001 0010 0011 0101
result: fail

key = 1001 1000 1010 0010 1011 0101 0011 1000
```

```
reverse_key = 1011 0101 0011 1000
result: success

key = 0100 0011 1001 0000 0011 0001 1001 0101
reverse_key = 0011 0001 1001 0101
result: success

key = 1111 0000 1110 1011 0001 0001 0000 1011
reverse_key = 0001 0001 0000 1011
result: success

key = 1111 0011 1110 1100 0001 0101 1000 0110
reverse_key = 0001 1000 1000 0110
result: fail

key = 1011 1011 1100 0011 0011 1110 1011 0110
reverse_key = 0011 1000 1011 0011
result: fail

key = 1111 0001 1100 0000 0110 1101 1000 0101
reverse_key = 0110 1101 1000 0101
result: success

key = 1110 1100 0111 1101 1101 1100 0101 0011
reverse_key = 1101 1100 0101 0011
result: success

key = 1000 0101 0001 0000 0101 1110 0110 1001
reverse_key = 0101 1110 0110 1001
result: success

key = 1000 0100 1110 1100 1111 0000 0010 1010
reverse_key = 1111 0000 0010 1010
result: success

key = 1010 1101 1111 0111 0001 0001 0110 1001
reverse_key = 1101 1100 0111 0100
result: fail

key = 0100 1111 0111 1011 0011 0000 1011 0000
reverse_key = 0011 0000 1011 0000
result: success

key = 1110 1000 1001 0011 1010 0011 1100 0111
reverse_key = 1010 0011 1100 0111
result: success
```

```
key = 0110 1100 1111 1110 1110 1011 1010 0100
reverse_key = 1110 1011 1010 0100
result: success

key = 0010 1101 0101 1010 0001 0111 1110 1011
reverse_key = 0001 0111 1110 1011
result: success

key = 0000 1100 0010 1010 1000 1001 0000 1001
reverse_key = 1000 1010 0000 1001
result: fail

key = 1110 1110 1100 1011 0100 0000 0100 0001
reverse_key = 0100 0000 0100 0001
result: success

key = 0011 0001 1011 0110 0110 0001 0001 1001
reverse_key = 0110 0001 0001 1001
result: success
```

*可以看到在**50组密钥攻击中，其中36组密钥攻击成功，14组失败**，准确率并不高，继续增加明文-密文对(即大于8000对)可以提高准确率。*

因此令`T = 10000`，再次运行结果如下：

```
key = 0101 1111 0100 0110 0011 1101 1110 0010
reverse_key = 0011 1101 1110 0010
result: success

key = 1011 1010 0011 1000 0111 1000 0111 0101
reverse_key = 0111 1000 0111 0101
result: success

key = 0110 0000 1010 1000 1100 1101 0110 0001
reverse_key = 1100 1101 0110 0001
result: success

key = 0110 0100 1111 1011 0110 0110 1010 1111
reverse_key = 0110 0110 1010 1111
result: success

key = 1011 1110 1010 1011 1001 0101 1101 1101
reverse_key = 1001 0101 1101 1110
result: fail
```

```
key = 1100 0100 1100 0000 0010 0010 0100 0100
reverse_key = 0010 0010 0100 0100
result: success

key = 0111 1100 1000 0000 0110 0001 0100 1101
reverse_key = 0110 0001 0100 1101
result: success

key = 0000 0101 0101 1110 0010 1101 1010 1100
reverse_key = 0010 1101 1010 1100
result: success

key = 0101 0101 0010 0101 1011 0110 1000 1101
reverse_key = 1011 0110 1000 1101
result: success

key = 0011 1011 1110 0100 1011 1111 1001 0001
reverse_key = 1011 1111 1001 0001
result: success

key = 1001 0110 1010 1110 1001 1000 1001 0011
reverse_key = 1001 1000 1001 0011
result: success

key = 0001 1110 0101 0100 1001 0101 1100 1100
reverse_key = 1001 0101 1100 1100
result: success

key = 1101 1111 1011 1001 1001 1101 1111 0001
reverse_key = 1001 1101 1111 1100
result: fail

key = 1111 1011 0110 0101 1101 0100 0010 1010
reverse_key = 1101 0100 0010 1010
result: success

key = 0101 0000 0001 0101 1011 1111 1001 0111
reverse_key = 1011 1111 1001 0111
result: success

key = 0100 1111 0100 1101 1101 0101 0110 1111
reverse_key = 1101 0101 0110 1111
result: success

key = 0010 0010 0110 1001 0010 0110 0110 1010
reverse_key = 0010 1011 0110 1010
```

```
result: fail

key = 1010 1010 1101 0100 0000 1001 0011 0010
reverse_key = 0000 1001 0011 0010
result: success

key = 1010 0101 1101 0100 1111 1101 1110 1100
reverse_key = 1111 1101 1110 1100
result: success

key = 1100 1111 1000 1011 0011 0011 0010 1010
reverse_key = 0011 0011 0010 1010
result: success

key = 0101 0000 1010 1100 0100 1000 1111 1010
reverse_key = 0100 1000 1111 1010
result: success

key = 0010 0100 0110 1110 1000 0111 1111 1100
reverse_key = 1000 1010 1111 0001
result: fail

key = 1111 1111 0001 0101 1101 1101 1100 0101
reverse_key = 1101 1101 1100 0101
result: success

key = 0010 0110 1110 0010 0111 1101 1101 1001
reverse_key = 0111 1101 1101 1001
result: success

key = 1100 1011 1000 0100 0000 1110 0001 1011
reverse_key = 0000 1110 0001 1011
result: success

key = 0001 0011 1110 0011 1111 0111 0011 0011
reverse_key = 1111 1001 0011 0011
result: fail

key = 1001 1100 0101 0111 1001 1111 1101 0110
reverse_key = 1001 1111 1101 0110
result: success

key = 1101 1001 0011 0000 0001 1001 1100 0011
reverse_key = 0001 1001 1100 1110
result: fail

```

```
key = 0110 0100 1011 1000 1001 1100 1100 1110
reverse_key = 1001 1100 1100 1110
result: success

key = 1101 0001 0000 1001 1100 0101 0001 1011
reverse_key = 1100 0101 0001 1011
result: success

key = 1100 1011 0000 1100 0011 1000 1111 1011
reverse_key = 0011 1000 1111 1011
result: success

key = 1111 1000 0101 1100 1111 0011 1000 0110
reverse_key = 1111 0011 1000 0110
result: success

key = 1010 0101 1011 1011 0000 0011 0010 1101
reverse_key = 0000 0011 0010 1101
result: success

key = 1011 1101 0010 1110 1110 1110 0000 1111
reverse_key = 1110 1110 0000 1111
result: success

key = 0010 0101 1001 1111 0011 0100 0000 1111
reverse_key = 0011 0100 0000 1111
result: success

key = 1000 0011 0001 0101 0010 1100 0100 0111
reverse_key = 0010 1100 0100 0111
result: success

key = 0000 1101 0011 0000 1101 0010 0100 1101
reverse_key = 1101 0010 0100 0000
result: fail

key = 0010 0111 0101 1000 1101 0001 1111 1100
reverse_key = 1101 0001 1111 1100
result: success

key = 0110 0101 0000 1010 1011 1100 0000 0000
reverse_key = 1011 1100 0000 0000
result: success

key = 1001 0110 0011 0000 1100 1011 1011 0001
reverse_key = 1100 1011 1011 0001
```

```
result: success

key = 1001 1101 0111 0010 1000 0011 0000 0111
reverse_key = 1000 0011 0000 0111
result: success

key = 0101 1000 1000 1001 1111 1101 1101 1001
reverse_key = 1111 1101 1101 1001
result: success

key = 1100 1001 0111 0100 0101 0000 1001 0011
reverse_key = 0101 0000 1001 0011
result: success

key = 0101 0010 0101 0011 0100 1110 1100 1000
reverse_key = 0100 1110 1100 1000
result: success

key = 1010 0100 1001 0000 1010 1100 1110 1001
reverse_key = 1010 1100 1110 1001
result: success

key = 1100 0010 1111 1011 1101 1010 1111 1010
reverse_key = 1101 1010 1111 1010
result: success

key = 0101 1111 0110 1010 1001 0100 1011 1101
reverse_key = 1001 0100 1011 1101
result: success

key = 1001 1101 0110 1111 0110 1111 1010 0110
reverse_key = 0110 1111 1010 0110
result: success

key = 0000 1001 1100 0100 1011 0010 0101 1101
reverse_key = 1011 1111 0101 1110
result: fail

key = 0100 0001 1010 1000 0000 0010 1110 1000
reverse_key = 0000 0010 1110 1000
result: success
```

可以看到**50组成功42组，失败8组**，准确率有所提升。