SPN线性密码分析【附code】

D-A-X 于 2021-01-20 11:59:33 发布

综述

SPN 线性 密码分析：

- 基于S盒子逼近的分析方法；
- 是已知明文的分析方法，需要较多的名密文对；
- 此方法只能分析最后一轮子密钥，缩小了密钥穷举范围（当分析出最后一轮密钥时，由于密钥生成算法固定，因此也为分析第一轮密钥提供了可能）。

相关概念

- 定义对于$\{0,1\}$上的离散随机变量X_i，$p_i = P(X_i = 0)$。
- 定义X_i的偏差为：$\epsilon_i = p_i - \frac{1}{2}$。
- **堆积引理：** 设$X_{i_1}, X_{i_2}, \cdots, X_{i_k}$是**独立**随机变量，$\epsilon_{i_1}, \epsilon_{i_2}, \cdots, \epsilon_{i_k}$分别表示随机变量$X_{i_1}, X_{i_2}, \cdots, X_{i_k}$的偏差，那么对变量$X_{i_1} \oplus X_{i_2} \oplus \cdots \oplus X_{i_k}$的偏差，则有：$\epsilon_{i_1,i_2,\cdots,i_k} = 2^{k-1} \prod_{j=1}^{k} \epsilon_{i_j}$。

线性分析过程

- 收集大量明密文对；
- 选择一个固定的输入位$in(x,y) = (\sum_{i=1}^{4} \oplus a_i x_i) \oplus (\sum_{i=1}^{4} \oplus b_i y_i)$，如：选择$x_1 \oplus x_3 \oplus y_2$；
- 通过收集的明密文对，根据上面构造的$in(x,y)$来构建线性逼近表；
- 根据输入，选取线性逼近表中偏差最大的作为输出，找到对应位置进行输入；再根据线性逼近表找到输出；以此类推，构建出线性逼近链（偏差越大，越具有线性关系）；
- 根据该线性逼近链，通过将中间过程中的输入输出相异或来化简过程，最终化简为只剩下输入和最后一轮输入异或的关系式；

- 遍历所有可能的密钥，遍历所有收集到的明密文对，通过y和当前测试的密钥计算出最后一轮输入，将该值带入关系式$test()$中，若该关系式为0，则该密钥对应$count$值加1；
- 遍历结束后输出$count$值最大的密钥。

线性分析原理

线性分析基于三个事实：

- 找到一组S盒子，对于输入明文X在经过该S盒子后，将得到固定的密文Y，我们可以列出所有明文X对应的密文Y的情况。选出需要的“逼近中的活动S盒”，计算这些活动S盒中输入随机变量和输出随机变量的异或的偏差。并利用堆积引理计算这些“逼近中的活动S盒”的总偏差。
- 根据输入随机变量经过这些S盒子的过程可以推算出仅包含明文比特和S盒子最后一轮输入比特的关系式。而通过已知的密文Y即可反推回最后一轮的输出值，再反推回最后一轮的输入值。这样我们便可以计算该关系式。
- 令$X=(x_1,x_2,\cdots,x_n)$，其对应的加密值$Y=(y_1,y_2,\cdots,y_n)$，即$Y=\pi_s(X)$。
  因此必有$X\oplus Y=0$。换言之，对于任意情况下的变量X、Y，若存在关系$X\oplus Y=0$，则必有Y为X的对应密文。

因此，当我们遍历可能的密钥空间，并计算相应的关系式的值，当为0时我们进行计数（为0表示此时明文与密文对应），结束后输出计数值最大的密钥，密钥越正确，对应的明密文则越多。
实际上真正的密钥计数器值应接近$\frac{1}{2}\pm\epsilon$，因为$\frac{1}{2}\pm\epsilon=p_i=P(X\oplus Y=0)$。

代码

线性分析

描述

内容：根据已知明密文对分析原始SPN的密钥。
要求：
（1）实现教材所给算法。
（2）能根据所给8000对明密文对分析对应位置密钥。
（3）分析出所有32比特密钥。

输入

第一行：一个正整数n，表示破解n组密钥
接下来的8000n行依次为十六进制的明文-密文对，形如“xxxx xxxx\n”。

第k个数据点满足：n = 10k

输出

输出n行，每行为8个字符表示的十六进制密钥

首先使用书中给出的线性分析链，分析出第5轮第2、4部分的密钥。再选择新的线性分析链，在第2、4部分密钥已知的基础上分析出第1、3部分的密钥。接着在已知起始密钥低16位的基础上，穷举高16位密钥对给出的8000个明密文对进行验证。由于在加密过程中进行了5轮的S代换和P置换，导致相同明文在不同密钥下得到相同密文的概率极低，因此在实际验证过程中并不需要验证8000个明密文对是否对应，而仅需验证3个即可判断出密钥是否合适。

需要注意的是：第一，由于虽然可以选取到仅包含第5轮第1、3部分的线性分析链，但是由于偏差不大，因此代表性较差，可能导致对于1、3部分密钥可能性较大部分的遍历过多，而导致时间开销较大，所以自选的线性分析链应该首先选择偏差较大的链条（尽管该链条可能还包含第5轮第2或第4部分密钥）。第二，由于线性分析是基于概率的分析，因此count值最大的并不一定是真正密钥，因此需要在一定范围内遍历count值较大的所有可能密钥并进行验证。

基于以上两点，需要有两层主循环，第一层主循环遍历可能的第5轮第2、4部分密钥，第二层在已知第5轮第2、4部分密钥的基础上生成并遍历可能的第5轮第1、3部分密钥，同时穷举高16位密钥并进行验证。

综上程序流程为：读入明密文对并存入数组；根据明密文对和书中已知线性分析链计算第5轮第2、4部分密钥对应的count值并存起来；开始两层循环；找到后退出循环并输出结果。

读入和输出采用快速读入和输出。

```
#include <stdio.h>
#include <string.h>
#define abs(a) {if(a >= 4000) a -= 4000; else a = 4000 - a;}
typedef unsigned short ushort;
typedef unsigned int uint;

const int MAX = 8005;
int n, count13[2][16][16], cnt13[16][16], cnt24[16][16];
ushort plaintext[MAX], ciphertext[MAX], tail_key;
int key51, key52, key53, key54;
uint key;

//SPN statement
const unsigned short sBox_4[16] = {0xe, 0x4, 0xd, 0x1, 0x2, 0xf, 0xb, 0x8, 0x3, 0xa, 0x6, 0xc, 0x5, 0x9, 0x0, 0x7};
const unsigned short inverse_sBox[16] = {0xe, 0x3, 0x4, 0x8, 0x1, 0xc, 0xa, 0xf, 0x7, 0xd, 0x9, 0x6, 0xb, 0x2, 0x0, 0x5};
const unsigned short pos[17] = {0x0,
                                0x8000, 0x4000, 0x2000, 0x1000,
                                0x0800, 0x0400, 0x0200, 0x0100,
                                0x0080, 0x0040, 0x0020, 0x0010,
                                0x0008, 0x0004, 0x0002, 0x0001};
const unsigned short pBox[17] = {0x0,
                                 0x8000, 0x0800, 0x0080, 0x0008,
                                 0x4000, 0x0400, 0x0040, 0x0004,
                                 0x2000, 0x0200, 0x0020, 0x0002,
                                 0x1000, 0x0100, 0x0010, 0x0001};
unsigned short sBox_16[65536], spBox[65536];

void get_spBox(){  //获得spBox
    for(int i = 0; i < 65536; ++i){
        sBox_16[i] = (sBox_4[i >> 12] << 12) | (sBox_4[(i >> 8) & 0xf] << 8) | (sBox_4[(i >> 4) & 0xf] << 4) | sBox_4[i & 0xf];
        spBox[i] = 0;
        for(int j = 1; j <= 16; ++j){
```

```
            if(sBox_16[i] & pos[j]) spBox[i] |= pBox[j];
        }
    }
}

inline ushort read(){
    char ch;
    ushort buf = 0;
    for(int i = 0; i < 4; ){
        ch = getchar();
        if(ch >= '0' && ch <= '9'){
            buf = (buf << 4) | (ch ^ 48);
            i++;
        }
        else if(ch >= 'a' && ch <= 'z'){
            buf = (buf << 4) | (ch - 'a' + 10);
            i++;
        }
    }
    return buf;
}

void output(){
    char buf[8];  //输出缓冲区
    for(int i = 0; i < 8; ++i){
        buf[7 - i] = key & 0xf;
        if(buf[7 - i] < 10) buf[7 - i] += '0';
        else buf[7 - i] = (buf[7 - i] - 10) + 'a';
        key >>= 4;
    }
    for(int i = 0; i < 8; ++i) putchar(buf[i]);
    putchar('\n');
}

inline void input(){
    for(int i = 1; i <= 8000; ++i){
        plaintext[i] = read();
        ciphertext[i] = read();
    }
```

```
}

int main(){
    get_spBox();

    scanf("%d", &n);
    bool flag;
    ushort u1, u2, u3, u4;
    ushort x_init, y_init, z;
    ushort x[13], y[5];
    for(int i = 0; i < n; ++i){
        input();
        flag = false;
        //linear24();  //先分析第2、4位
        memset(cnt24, 0, 256 * sizeof(int));
        for(int group = 1; group <= 8000; ++group){
            //提前处理要用到的对应位值
            x_init = plaintext[group];
            for(int pos = 1; pos <= 12; ++pos){
                x[pos] = (x_init >> (16 - pos)) & 0x1;
            }
            y_init = ciphertext[group];
            for(int pos = 1, k = 12; pos <= 4; ++pos, k -= 4){
                y[pos] = (y_init >> k) & 0xf;
            }

            for(int L1 = 0; L1 < 16; ++L1){
                for(int L2 = 0; L2 < 16; ++L2){
                    u2 = inverse_sBox[L1 ^ y[2]];
                    u4 = inverse_sBox[L2 ^ y[4]];
                    z = (x[5] ^ x[7] ^ x[8] ^ (u2 >> 2) ^ u2 ^ (u4 >> 2) ^ u4) & 0x1;
                    if(!z) cnt24[L1][L2]++;
                }
            }
        }

        //处理count相加值
        for(int L1 = 0; L1 < 16; ++L1){
            for(int L2 = 0; L2 < 16; ++L2){
```

```
            abs(cnt24[L1][L2]);
        }
    }

    //外循环
    for(int round24 = 0; round24 < 64; ++round24){
        //计算最大的2、4位对应密钥值
        int max24 = -1;
        for(int L1 = 0; L1 < 16; ++L1){
            for(int L2 = 0; L2 < 16; ++L2){
                if(cnt24[L1][L2] > max24){
                    max24 = cnt24[L1][L2];
                    key52 = L1;
                    key54 = L2;
                }
            }
        }
        cnt24[key52][key54] = 0;

        //根据2、4位对应密钥值计算1、3位对应密钥值;
        //linear13();
        memset(count13, 0, 512 * sizeof(int));
        for(int group = 1; group <= 8000; ++group){
            //提前处理要用到的对应位值
            x_init = plaintext[group];
            for(int pos = 1; pos <= 12; ++pos){
                x[pos] = (x_init >> (16 - pos)) & 0x1;
            }
            y_init = ciphertext[group];
            for(int pos = 1, k = 12; pos <= 4; ++pos, k -= 4){
                y[pos] = (y_init >> k) & 0xf;
            }
            //开始计算count
            for(int L1 = 0; L1 < 16; ++L1){
                for(int L2 = 0; L2 < 16; ++L2){
                    u1 = inverse_sBox[y[1] ^ L1];
                    u2 = inverse_sBox[y[2] ^ key52];
                    u3 = inverse_sBox[y[3] ^ L2];
                    u4 = inverse_sBox[y[4] ^ key54];
```

```
                z = (x[1] ^ x[2] ^ x[4] ^ (u1 >> 3) ^ (u2 >> 3) ^ (u3 >> 3) ^ (u4 >> 3)) & 0x1;
                if(!z) count13[0][L1][L2]++;
                z = (x[9] ^ x[10] ^ x[12] ^ (u1 >> 1) ^ (u2 >> 1) ^ (u3 >> 1) ^ (u4 >> 1)) & 0x1;
                if(!z) count13[1][L1][L2]++;
            }
        }
    }

    //处理count相加值
    for(int L1 = 0; L1 < 16; ++L1){
        for(int L2 = 0; L2 < 16; ++L2){
            abs(count13[0][L1][L2]);
            abs(count13[1][L1][L2]);
            cnt13[L1][L2] = count13[0][L1][L2] + count13[1][L1][L2];
        }
    }


    for(int round13 = 0; round13 < 2; ++round13){
        int max13 = -1;
        for(int L1 = 0; L1 < 16; ++L1){
            for(int L2 = 0; L2 < 16; ++L2){
                if(cnt13[L1][L2] > max13){
                    max13 = cnt13[L1][L2];
                    key51 = L1;
                    key53 = L2;
                }
            }
        }
        cnt13[key51][key53] = 0;
        tail_key = (key51 << 12) | (key52 << 8) | (key53 << 4) | key54;

        //穷举
        int count, fore_key, k1, k2, k3, k4, k5;
        for(fore_key = 0; fore_key < 65536; ++fore_key){
            key = (fore_key << 16) | tail_key;
            //get_roundKey();
            k5 = tail_key;
```

```
                    k4 = (key >> 4) & 0xffff;
                    k3 = (key >> 8) & 0xffff;
                    k2 = (key >> 12) & 0xffff;
                    k1 = (key >> 16) & 0xffff;
                    for(count = 1; count < 4; ++count){
                        //encrypt: ciphertext = sBox_16[spBox[spBox[spBox[plaintext ^ k1] ^ k2] ^ k3] ^ k4] ^ k5;
                        if((sBox_16[spBox[spBox[spBox[plaintext[count] ^ k1] ^ k2] ^ k3] ^ k4] ^ k5) != ciphertext[count]) break;
                    }
                    if(count == 4){
                        flag = true;
                        break;
                    }
                }
                if(flag) break;
            }
            if(flag) break;
        }
        output();
    }

    return 0;
}
```

相关资料

差分分析【附code】