LSFR流密码小结

原创

M3ng@L 于 2022-03-19 00:09:00 发布 437 收藏

分类专栏： CTF比赛复现 密码学知识总结 文章标签： Crypto python lsfr

本文链接：https://blog.csdn.net/qq_51999772/article/details/123587428

CTF比赛复现 同时被 2 个专栏收录

31 篇文章 0 订阅

订阅专栏

密码学知识总结

18 篇文章 2 订阅

订阅专栏

LSFR流密码小结

keywords: `LSFR`

Introduction

关于`lfsr`，可以简单地说做，使用明文二进制位作为密钥流发生器的种子，生成密文

观察`lfsr`的`python`实现代码

```
def lfsr(R,mask):
    output = (R << 1) & 0xffffffff
    i = (R & mask) & 0xffffffff
    lastbit = 0
    while i != 0:
        lastbit ^= (i & 1)
        i = i >> 1
    output ^= lastbit
    return (output,lastbit)
```

首先看到`& 0xffffffff`这个与运算，大家都知道和`1`做与运算是不会改变原来的值的，但是如果用大于`0xffffffff`的数与其做与运算就会缩小到与`0xffffffff`同二进制位的数值；所以ta的作用是防止运算结果溢出到大于`0xffffffff`（当然在不同的场合中这个数也会改变，一般是会与需要加密的明文同二进制位数）

这里的`R`相当于本次加密的种子（第一次的`R`当然是明文，之后会不断改变），看到`output = (R << 1) & 0xffffffff`；将`R`二进制全部左移一位，且将最高位舍去，低位用0补齐（刚才说了`0xffffffff`这个数的取值一般与`R`同二进制位数）

`i = (R & mask) & 0xffffffff`；故`i`是`R`与掩码`mask`进行与运算得到之后进行异或的有效位

```
    while i != 0:
        lastbit ^= (i & 1)
        i = i >> 1
```

`i & 1`就是取`i`得低位的第一位，与`lastbit`进行异或（`lastbit`首先赋值是0，所以第一次运算的结果就是`lastbit = i & 1`）；进行完这步之后，`i >> 1`，会不断将`i`右移，所以最后的结果是每一位`i`都进行了与`lastbit`的异或运算

整个`while`循环就是实现了将`i`中的每一位进行相互异或（而`i`又是`R`和掩码`mask`进行与运算之后的结果，所以实质上就是`R`的有效位相互异或），得到结果`lastbit`，退出循环（注意这里的`lastbit`是只有一位二进制的）

`output ^= lastbit`，联系到之前生成`output`的过程，是`R`二进制全部左移一位低位补0生成的，这里就是相当于将`output`变成`R`二进制全部左移一位低位补`lastbit`

最后返回`output`和`lastbit`，但CTF中一般不会已知`output`，而是知道`lastbit`的数值

关于这样的`lfsr`函数一般遇见的使用方式

```
f=open("enc","w")
sum_enc = # 需要生成的密文总长
R = plaintext
mask = # mask需要保密，且与R同二进制位数
for i in range(sum_enc):
    tmp=0
    for j in range(8): # 进行8次循环是因为，8次循环得到的8个二进制位恰好是一个字节的长度
        (R,out) = lfsr(R,mask)
        tmp = (tmp << 1) ^ out
    f.write(chr(tmp))
```

`tmp = (tmp << 1) ^ out`，就是将`lsfr`得到的所有`lastbit`都组合起来生成密文

CTF Problems

下面会举几道例题进行分析

有强网杯2018的streamgame系列和CISCN2018的oldstream

题目链接：2018-QWB-CTF/02Crypto-5题 at master · jas502n/2018-QWB-CTF · GitHub

BUUCTF在线评测 (buuoj.cn)

第二届强网杯全国网络安全挑战赛 | 闲言语 (ret2neo.cn)

streamgame1-2

keywords: 爆破明文空间

Description

这里以streamgame2为例（因为两道题的方式是一样的）

```python
from flag import flag
assert flag.startswith("flag{")
assert flag.endswith("}")
assert len(flag)==27 # streamgame是len(flag) == 21;除mask以外其他都一样，都在可爆破的范围，所以解法一样

def lfsr(R,mask):
    output = (R << 1) & 0xffffff
    i=(R&mask)&0xffffff
    lastbit=0
    while i!=0:
        lastbit^=(i&1)
        i=i>>1
    output^=lastbit
    return (output,lastbit)

R=int(flag[5:-1],2)
mask=0x100002

f=open("key","ab")
for i in range(12):
    tmp=0
    for j in range(8):
        (R,out)=lfsr(R,mask)
        tmp=(tmp << 1)^out
    f.write(chr(tmp))
f.close()
```

Analysis

首先看明文空间（与`flag`参与`lsfr`加密的长度相关）

这里的`flag`是二进制，一共有

$27-6=21$ 位二进制参与了`lsfr`加密，所以明文空间就是2^{21}，这是一般电脑可以接受的爆破范围

爆破方法就是将明文空间中的每一位数值代入整个`lsfr`加密过程，得到的结果如果与已知的密文一致，则为`flag`（不用担心是否会出现不同种子出现相同密文的情况，流密码确实是有生成循环，但是循环大到不可能跑完一次循环）

Solving code

```python
from tqdm import tqdm
from Crypto.Util.number import *

f = open("key","rb")
enc = f.read()
mask = 0x100002

def lfsr(R,mask):
    output = (R << 1) & 0xffffff
    i=(R&mask)&0xffffff
    lastbit=0
    while i!=0:
        lastbit^=(i&1)
        i=i>>1
    output^=lastbit
    return (output,lastbit)
# print(len(enc))
for flag in tqdm(range(2**21)):
    now_enc = []
    R = flag
    judge = 1
    for i in range(len(enc)):
        tmp = 0
        for j in range(8):
            (R,out) = lfsr(R,mask)
            tmp = (tmp << 1) ^ out
        if tmp != enc[i]:
            judge = 0
            break
    if judge == 1:
        print(flag)
        print("flag{" + str(bin(flag))[2:] + "}")
        break
```

streamgame3

keywords: 相关攻击， 非线性反馈移位寄存器， Geffe生成器

Description

```
from flag import flag
assert flag.startswith("flag{")
assert flag.endswith("}")
assert len(flag)==24

def lfsr(R,mask):
    output = (R << 1) & 0xffffff
    i=(R&mask)&0xffffff
    lastbit=0
    while i!=0:
        lastbit^=(i&1)
        i=i>>1
    output^=lastbit
    return (output,lastbit)


def single_round(R1,R1_mask,R2,R2_mask,R3,R3_mask):
    (R1_NEW,x1)=lfsr(R1,R1_mask)
    (R2_NEW,x2)=lfsr(R2,R2_mask)
    (R3_NEW,x3)=lfsr(R3,R3_mask)
    return (R1_NEW,R2_NEW,R3_NEW,(x1*x2)^((x2^1)*x3))

R1=int(flag[5:11],16)
R2=int(flag[11:17],16)
R3=int(flag[17:23],16)
assert len(bin(R1)[2:])==17
assert len(bin(R2)[2:])==19
assert len(bin(R3)[2:])==21
R1_mask=0x10020
R2_mask=0x4100c
R3_mask=0x100002

for fi in range(1024):
    print fi
    tmp1mb=""
    for i in range(1024):
        tmp1kb=""
        for j in range(1024):
            tmp=0
            for k in range(8):
                (R1,R2,R3,out)=single_round(R1,R1_mask,R2,R2_mask,R3,R3_mask)
                tmp = (tmp << 1) ^ out
            tmp1kb+=chr(tmp)
        tmp1mb+=tmp1kb
    f = open("./output/" + str(fi), "ab")
    f.write(tmp1mb)
    f.close()
```

Analysis

看得出来关键函数是`single_round`

```
def single_round(R1,R1_mask,R2,R2_mask,R3,R3_mask):
    (R1_NEW,x1)=lfsr(R1,R1_mask)
    (R2_NEW,x2)=lfsr(R2,R2_mask)
    (R3_NEW,x3)=lfsr(R3,R3_mask)
    return (R1_NEW,R2_NEW,R3_NEW,(x1*x2)^((x2^1)*x3))
```

由不同的种子通过`lsfr`函数生成的密钥流`x1,x2,x3`，再参与了一个非线性组合运算

对比之前的`lsfr`题目，这里的爆破范围达到了`pow(2,len(R1)) * pow(2,len(R2)) * pow(2,len(R3))`

而`R1,R2,R3`是`flag`的各个部分

```
R1=int(flag[5:11],16)
R2=int(flag[11:17],16)
R3=int(flag[17:23],16)
```

最后输出的结果是非线性组合运算的结果逐个组成的数据

```
            tmp=0
            for k in range(8):
                (R1,R2,R3,out)=single_round(R1,R1_mask,R2,R2_mask,R3,R3_mask)
                tmp = (tmp << 1) ^ out
            tmp1kb+=chr(tmp)
```

所以显然这里无法直接爆破,但是我们可以使用**分治**的方法将总的爆破时间减小

这里就涉及到了**相关攻击** (一般是针对流密码的)

相关攻击的应用点就在于最后的非线性组合运算的结果和参与最后非线性组合运算的`x1,x2,x3`之间的关系

姑且先不分析这个非线性组合运算，直接看这个非线性组合运算的结果构成的**真值表**

x1	x2	x3	result
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

我们可以看到，结果`result`与`x1,x3`相同的概率都是`0.75`

而一般组合运算的要求是输入数据和输出数据的相同的概率是`0.5`，这样才能避免相关攻击

我们也可以不找到真值表，可以直接对`x1,x2,x3,result`进行假设分析

以当需要满足`result=x3`为例

假设`x2=1`，那么`result=x1`（而当`x1==x3`时也就是`result=x1=x3`，这也是`0.5`的概率）；当`x2=0`，那么`result=x3`

所以`p(result = 3) = 0.5 + 0.5 * 0.5 = 0.75`

同理`result = 2`时候的概率

综上，我们可以分别爆破`R1,R3`，经过`lsfr`函数计算出来的`x1,x3`来对比整个组合运算得到的结果；如果对比的结果相同对的概率越接近`0.75`那么此时爆破的`R`一般就是真实的`R`（当对比的`x`与`result`的位数过少时可能得到不正确的答案，但是对比的位数越大越有可能是真实的`R`）

当爆破完`R1,R3`之后，由于`x2`与`result`之间相同的概率是`0.5`，无法使用相关攻击；但是此时`R1,R2,R3`已知两个，可以爆破`R2`，代入`single_round`得出的结果与原来的密文进行比较

尽量选取多个位数比较（当然不可能是所有位数比较，因为这样所需时长会非常长），当比较一致的时候一般就是真实的`R2`了

Solving code

```
from Crypto.Util.number import *
import gmpy2
from tqdm import tqdm

f = open("output","rb")
ori_cipher = f.read()

def lfsr(R,mask):
    output = (R << 1) & 0xffffff
    i=(R&mask)&0xffffff
    lastbit=0
    while i!=0:
        lastbit^=(i&1)
        i=i>>1
    output^=lastbit
    return (output,lastbit)
def single_round(R1,R1_mask,R2,R2_mask,R3,R3_mask):
    (R1_NEW,x1)=lfsr(R1,R1_mask)
    (R2_NEW,x2)=lfsr(R2,R2_mask)
    (R3_NEW,x3)=lfsr(R3,R3_mask)
    return (R1_NEW,R2_NEW,R3_NEW,(x1*x2)^((x2^1)*x3))

def get_relation(ori_cipher,now_cipher):
    cipher = ""
    for i in range(len(ori_cipher)):
        cipher += str(bin(ori_cipher[i]))[2:].zfill(8)
    count = 0.0
    assert len(cipher) == len(now_cipher)
    for i,j in zip(cipher,now_cipher):
        count += (i == j)
    return count/len(now_cipher)

def correlation_attack(R_range,mask,required_bin):
    global ori_cipher
    real_R = 0
    relation = 0
    for R in tqdm(range(R_range)):
        now_cipher = ""
        temp_R = R
        for _ in range(required_bin):
            R, lastbit = lfsr(R, mask)
            now_cipher += str(lastbit)
        temp_relation = get_relation(ori_cipher[:len(now_cipher)//8],now_cipher)
        if temp_relation > relation:
            relation = temp_relation
            real_R = temp_R
    return(real_R,relation)

R1_mask = 0x10020
R2_mask = 0x4100c
R3_mask = 0x100002
# R1 = correlation_attack(2**17,R1_mask,17*8)
```

```python
# print(R1)
R1 = 113099
# R3 = correlation_attack(2**21,R3_mask,21*8)
# print(R3)
R3 = 1487603

for R2 in tqdm(range(2**19)):
    temp_hex = 10
    tempR1 = R1
    tempR2 = R2
    tempR3 = R3
    tmp = 0
    for _ in range(temp_hex * 8):
        (tempR1,tempR2,tempR3,out) = single_round(tempR1,R1_mask,tempR2,R2_mask,tempR3,R3_mask)
        tmp = (tmp << 1) ^ out
    tmp1kb = str(tmp)
    cipher = ori_cipher[:temp_hex]
    # print(cipher)
    assert len(cipher) == len(long_to_bytes(tmp1kb))
    if cipher == long_to_bytes(tmp1kb):
        print(hex(R2))
        print("flag{" + str(hex(R1))[2:] + str(hex(R2))[2:] + str(hex(R3))[2:] + "}")
        break
```

Reference

非线性反馈移位寄存器 - CTF Wiki (ctf-wiki.org)

第二届强网杯全国网络安全挑战赛 | 闲言语 (ret2neo.cn)

streamgame4

keywords: 截取部分密文爆破， 非线性反馈发生器

Description

```
from flag import flag
assert flag.startswith("flag{")
assert flag.endswith("}")
assert len(flag)==27

def nlfsr(R,mask):
    output = (R << 1) & 0xffffff
    i=(R&mask)&0xffffff
    lastbit=0
    changesign=True
    while i!=0:
        if changesign:
            lastbit &= (i & 1)
            changesign=False
        else:
            lastbit^=(i&1)
        i=i>>1
    output^=lastbit
    return (output,lastbit)

R=int(flag[5:-1],2)
mask=0b110110011011001101110

f=open("key","ab")
for i in range(1024*1024):
    tmp=0
    for j in range(8):
        (R,out)=nlfsr(R,mask)
        tmp=(tmp << 1)^out
    f.write(chr(tmp))
f.close()
```

Analysis

加密函数是`nlfsr`，与`lfsr`函数不同的是

```
    changesign=True
    while i!=0:
        if changesign:
            lastbit &= (i & 1)
            changesign=False
        else:
            lastbit^=(i&1)
```

也就是在每次进行加密时，`lastbit`首先是与`i`的低位第一位进行**与运算**，之后才进行和`lsfr`函数一样的把剩余的有效位相异或

$$key \quad = (i \ \& \ i \ ) \oplus i \oplus i \oplus$$

这是**非线性滤波生成器**，对一个 LFSR 的内容使用一个非线性组合函数

但是这道题实际上不用管加密函数的具体实现，依旧使用**爆破**，只是这里给出的`key`很大，如果将爆破的各个种子生成的密钥流等长地比较就太慢了，实际上我们呢只需要开头的几个字节是相同的即可（也可以逐个字节判断，若不同则`break`）

Solving code

```
from tqdm import tqdm
with open("key","rb") as f:
    enc = f.read()
    f.close()
enc = enc[:5]
# enc = [209,217,64,67,147]
def nlfsr(R,mask):
    output = (R << 1) &0xffffff
    i=(R&mask)&0xffffff
    lastbit=0
    changesign=True
    while i!=0:
        if changesign:
            lastbit &= (i & 1)
            changesign=False
        else:
            lastbit^=(i&1)
        i=i>>1
    output^=lastbit
    return (output,lastbit)

mask = 0b11011001101100110111O
for flag in tqdm(range(2**21)):
    R = flag
    judge = 1
    for i in range(5):
        tmp = 0
        for j in range(8):
            (R,out) = nlfsr(R,mask)
            tmp = (tmp << 1) ^ out
        if enc[i] != tmp:
            judge = 0
            break
    if judge == 1:
        print()
        print('flag{' + str(bin(flag))[2:] + '}')
        break
```

Oldstream

keywords: 已知掩码，逆向运算流密码

Description

```
assert flag.startswith("flag{")
assert flag.endswith("}")
assert len(flag)==14

def lfsr(R,mask):
    output = (R << 1) & 0xffffffff
    i=(R&mask)&0xffffffff
    lastbit=0
    while i!=0:
        lastbit^=(i&1)
        i=i>>1
    output^=lastbit
    return (output,lastbit)

R=int(flag[5:-1],16)
mask = 0b10100100000010000000100010010100

f=open("key","w")
for i in range(100):
    tmp=0
    for j in range(8):
        (R,out)=lfsr(R,mask)
        tmp=(tmp << 1)^out
    f.write(chr(tmp))
f.close()
```

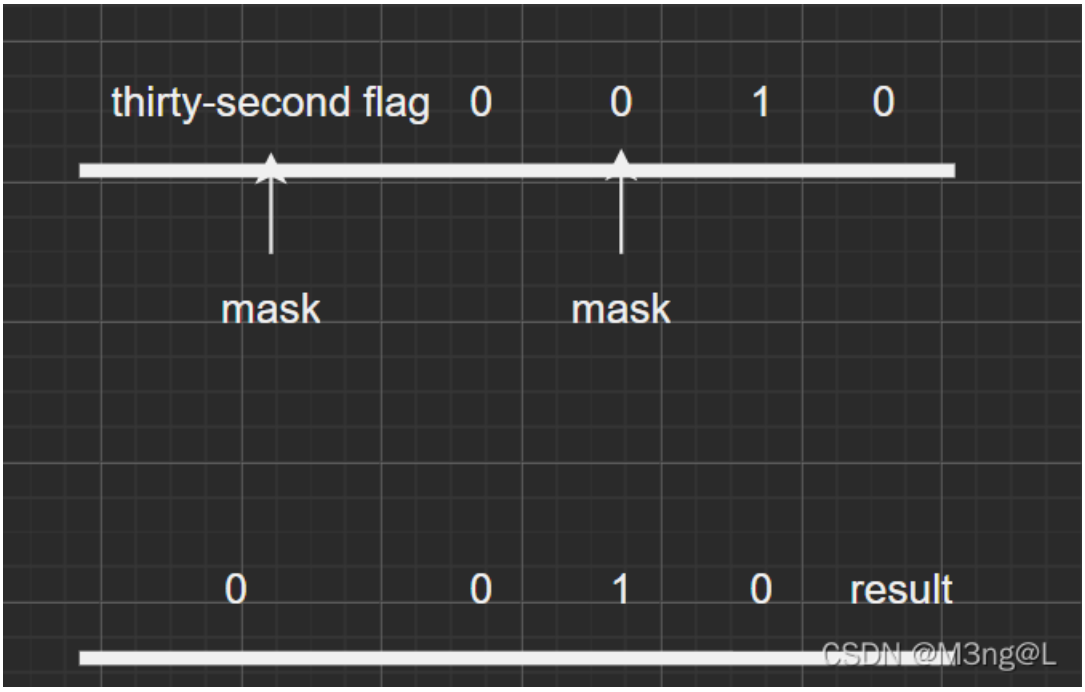
Analysis

首先可以分析一下爆破强度，这里的`flag`是有8位参与流密码运算（8位十六进制，所以实际上是32位二进制参与了运算），所以爆破强度是2 ；可以爆破，但是时间太长没有必要

接着看加密过程是正常的流密码加密，生成的密文有100个十六进制，而实际上我们不需要所有的密文就可以解出这道题。CTF破解流密码一般需要的是把加密过程表现出来，就可以知道解法了

`lsfr`函数中使用的是`&0xffffffff`，所以一次只能有32位二进制参与了实际运算；而这里**掩码已知**，所以我们知道在32位二进制中哪些位置的二进制确切地参与了之后的异或运算，进而我们可以从流密码输出的密文中使用异或的方式一步步推算出之前二进制位，也就是说可以从现有的密文推导出`flag`

拿第一步（恢复`flag`低位的第一位）举个例子



在加密过程中，恰好进行到对`flag`的第32位作为`stream`的第一位时进行加密的过程后，生成的`result`填充到`stream`低位，将`stream`高位第一位`thirty-second flag`去掉

由于这个`result`是掩码运算后的所有有效位相异或得到的结果（有效位也就是在这幅图中有`mask`箭头指向的二进制位），正好`flag`的二进制位也参与了运算（一定会参与运算），而我们现在已知了`result`和其他除`flag`二进制位外的所有二进制位，就可以通过异或进行恢复`flag`了

Solving code

```python
from tqdm import tqdm
from Crypto.Util.number import *
from re import *

f = open("key.enc","rb")

mask = 10100100000010000000100010010100
# print(len(str(mask)))
whole_enc = f.read()[:4]
temp = list(whole_enc)
temp_enc = ""
for i in list(whole_enc):
    # print(bin(i))
    temp_enc += bin(i)[2:].zfill(8)
temp_enc = temp_enc[:32]
# print(temp_enc)
enc = []
for i in range(len(temp_enc)):
    enc.append(temp_enc[i])
enc = list(map(int,enc))
print(enc)

# 寻找与掩码运算后有效位的index
mask_index = []
for i in finditer("1",str(mask)):
    mask_index.append(i.start())
# 逆向进行流密码过程，开始恢复flag
stream = enc
for _ in range(len(str(mask))):
    temp = 0
    for index in mask_index:
        if index != 0:
            temp = temp ^ int(stream[index - 1])
            # 由于流总体左移，所以上一次加密过程使用的二进制应该是这一次的index - 1
    part_flag = temp ^ int(stream[-1])
    # stream[:-1]也就是上一次加密生成的result
    stream.insert(0,part_flag)
    stream = stream[:32]
    print(part_flag,stream)
# print(str("".join(list(map(str,stream)))))
# print(hex(int(str("".join(list(map(str,stream)))),2)))
print("flag{" + str(hex(int(str("".join(list(map(str,stream)))),2)))[2:] + "}")
```