

武汉大学国家网络安全学院

密码学实验报告

学 号 2021302181156

姓 名 赵伯侯

实验名称 序列密码

指导教师 何琨

一、实验名称: 序列密码

二、实验目的及要求:

2.1 实验目的

- (1) 掌握序列密码的基本概念
- (2) 掌握线性移位寄存器的结构及其序列的伪随机性
- (3) 熟悉非线性序列的概念与基本产生方法
- (4) 了解常用伪随机性评价方法
- (5) 掌握一种典型流密码（如 RC4 或 ZUC 等）

2.2 实验要求

- (1) 掌握序列密码的实现方案
- (2) 掌握线性移位寄存器的构造
- (3) 熟悉序列伪随机性的基本测试方法
- (4) 实现 RC4 或 ZUC 算法

三、实验设备环境及要求:

Windows 操作系统，python 高级语言开发环境

四、实验内容与步骤:

4.1 序列密码实现方案

4.1.1 种子密钥 K 输入到密钥流发生器

该步骤是序列密码的起始步骤。在这里，一个预先定义的密钥（称为种子密钥 K）被输入到一个专门的算法或设备中，即密钥流发生器。这个种子密钥通常是一串固定长度的二进制数字，它的作用是为接下来的密钥流生成提供基础。种子密

钥的选择非常重要，因为它直接影响到生成的密钥流的随机性和安全性。

4.1.2 产生一系列密钥流

在种子密钥输入之后，密钥流发生器开始产生一个长序列的密钥流。这个密钥流实际上是一个很长的伪随机二进制数字序列，它用于后续的加密过程。这个过程的关键在于生成的密钥流必须是高度随机的，以确保加密的安全性。密钥流的长度通常取决于明文的长度，以确保每个明文单元都有一个对应的密钥单元。

4.1.3 通过与同一时刻的一个字节或者一位明文流进行异或操作产生密文流

最后一步是将密钥流与明文数据进行组合。这通常是通过异或（XOR）操作完成的。在异或操作中，密钥流中的每一位或字节与明文流中相应的位或字节组合。如果明文和密钥流的相应位相同，则结果为 0；如果不同，则结果为 1。这样生成的序列就是密文流，它可以被发送到接收方。接收方在拥有相同密钥流的前提下，可以通过相同的异或操作将密文解密回原始明文。

4.2 线性移位寄存器的构造

4.2.1 选择连接多项式（一般选择本原多项式作为连接多项式）

连接多项式在 LFSR 中起着核心作用。它决定了反馈逻辑，从而影响生成的序列的特性。本原多项式是一种特殊的多项式，它具有生成最长周期序列的能力。在选择本原多项式时，它能确保 LFSR 产生的序列具有最大的周期，这对于加密应用是非常重要的。这些多项式是在有限域上定义的，通常表示为一系列系数，这些系数对应于 LFSR 中的反馈点。

4.2.2 根据连接多项式的反馈系数得出反馈函数

反馈函数是 LFSR 的核心，它定义了如何从当前的寄存器状态生成下一个状态。这个函数是基于连接多项式的系数来构建的。这些系数决定了在 LFSR 中哪些位会被用于计算反馈值。反馈函数通常是通过将选定的寄存器位进行异或（XOR）操作来实现的。这些选定的位是由连接多项式的非零系数位置决定的。

4.2.3 根据反馈函数得出每个节拍的寄存器状态

在 LFSR 的每个时钟周期（节拍）中，根据反馈函数计算新的输出位，并将其反馈到寄存器的最末端。同时，寄存器中的其他位将根据寄存器的工作方式向前移动一位。这个过程不断重复，每个新的时钟周期都会根据反馈函数更新寄存器的状态，从而生成一个伪随机序列。

4.2.4 实验（1）

使用本原多项式 $g_1(x) = x^4 + x + 1$ 为连接多项式组成线性移位寄存器。画出逻辑图，写出输出序列及状态变迁。

将本原多项式 $g_1(x) = x^4 + x + 1$ 选定为连接多项式得到的线性移位寄存器的逻辑图如下图所示

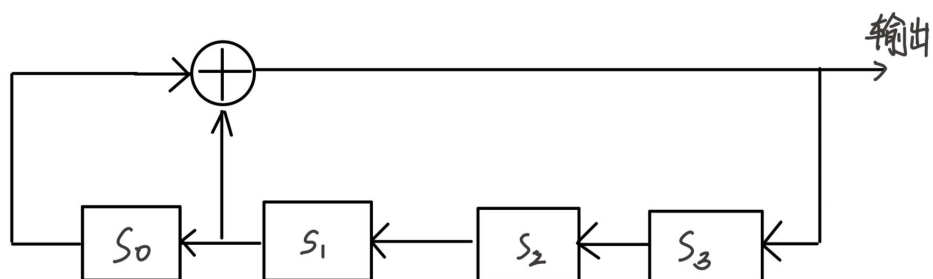


图 1: 线性移位寄存器 1

将寄存器的初值设置为 1001, 进行运算后得到的状态变迁表如下表所示

表 1: 序列 1 状态变迁表

状态序号	S0	S1	S2	S3
1	1	0	0	1
2	0	0	1	1
3	0	1	1	0
4	1	1	0	1
5	1	0	1	0
6	0	1	0	1
7	1	0	1	1
8	0	1	1	1
9	1	1	1	1
10	1	1	1	0
11	1	1	0	0
12	1	0	0	0
13	0	0	0	1
14	0	0	1	0
15	1	0	0	1
16	1	0	0	1
17	0	0	1	1
18	1	1	1	0

由状态变迁的过程得到的输出序列为 101011110001001..., 该序列是一个周期为 $2^4 - 1 = 15$ 的 m 序列。

4.2.5 实验 (2)

使用本原多项式 $g_1(x) = x^4 + x^3 + 1$ 为连接多项式组成线性移位寄存器。画出逻辑图, 写出输出序列及状态变迁。

将本原多项式 $g_1(x) = x^4 + x^3 + 1$ 选定为连接多项式得到的线性移位寄存器的逻辑图如下图所示

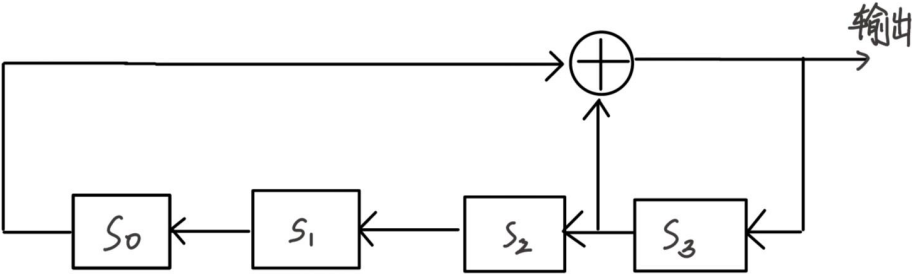


图 2: 线性移位寄存器 2

将寄存器的初值设置为 1001, 进行运算后得到的状态变迁表如下表所示

表 2: 序列 2 状态变迁表

状态序号	S0	S1	S2	S3
1	1	0	0	1
2	0	0	1	0
3	0	1	0	0
4	1	0	0	0
5	0	0	0	1
6	0	0	1	1
7	0	1	1	1
8	1	1	1	1
9	1	1	1	0
10	1	1	0	1
11	1	0	1	0
12	0	1	0	1
13	1	0	1	1
14	0	1	1	0
15	1	1	0	0
16	1	0	0	1
17	0	0	1	0

由状态变迁的过程得到的输出序列为 000111101011001..., 该序列是一个周期为 $2^4 - 1 = 15$ 的 m 序列。

4.2.6 输出序列关系

对比两组输出序列的关系

提示：（1）与（2）中的多项式是互反多项式，所谓互反多项式是指 $f(x)$ 与 $x^n f(\frac{1}{x})$

（1）中所得的输出序列为 101011110001001...，（2）中所得的输出序列为

000111101011001...，将序列 1 反向输出可以得到 100100011110101，将序列 2 调整顺序后将第 12 位作为周期的首位可得输出序列位 100100011110101，与序列 1 反向输出后的结果相同，由此可以得到结果两组输出序列相反，即为互反多项式构成的线性移位寄存器的输出序列相反。

4.3 随机性测试

随机性假设测试准则

4.3.1 准则 1（频率测试）

在 S 的周期 SN 中，1 的个数与 0 的个数至多相差 1。

（1）序列 1：

统计序列 1 一个周期中的 1 的个数为 8，0 的个数为 7，由于无论周期内的序列顺序如何选取，周期中的 1 和 0 的总量都不会发生改变，所以可以得出序列 1 中 1 的个数与 0 的个数相差 1，满足频率测试

（2）序列 2：

统计序列 2 一个周期中的 1 的个数为 8，0 的个数为 7 由于无论周期内的序列顺序如何选取，周期中的 1 和 0 的总量都不会发生改变，所以可以得出序列 2 中 1 的个数与 0 的个数相差 1，满足频率测试

4.3.2 准则 2（游程测试）

在 S 的周期 SN 中，至少有 $1/2$ 的游程长度为 1，至少有 $1/4$ 的游程长度为 2，至少有 $1/8$ 的游程长度为 3，以此类推，并且 0 和 1 游程的个数近似相等。

(1) 序列 1:

统计序列 1 的游程长度与数量的关系得到结果如下表所示

表 3: 序列 1 游程长度与数量关系

游程长度	0 游程个数	1 游程个数	总个数
1	2	2	4
2	1	1	2
3	1	0	1
4	0	1	1

游程长度为 1 的个数为 4, 占总游程数的比例 $\geq \frac{1}{2}$, 游程长度为 2 的个数为 2, 占总游程数的比例 $\geq \frac{1}{4}$ 游程长度为 3 的个数为 1, 占总游程数的比例 $\geq \frac{1}{8}$, 游程长度为 4 的个数为 1, 占总游程数的比例 $\geq \frac{1}{16}$ 并且 0 游程与 1 游程的个数相等, 所以序列 1 满足游程测试。

(2) 序列 2:

统计序列 2 的游程长度与数量的关系得到结果如下表所示

表 4: 序列 2 游程长度与数量关系

游程长度	0 游程个数	1 游程个数	总个数
1	2	2	4
2	1	1	2
3	1	0	1
4	0	1	1

游程长度为 1 的个数为 4, 占总游程数的比例 $\geq \frac{1}{2}$, 游程长度为 2 的个数为 2, 占总游程数的比例 $\geq \frac{1}{4}$ 游程长度为 3 的个数为 1, 占总游程数的比例 $\geq \frac{1}{8}$, 游程长度为 4 的个数为 1, 占总游程数的比例 $\geq \frac{1}{16}$ 并且 0 游程与 1 游程的个数相等, 所以序列 2 满足游程测试。

4.3.3 准则 3（自相关测试）

自相关函数 $R(t)$ 是双值的。即对某个整数 K ，有：

$$N \cdot R(t) = \sum_{i=0}^{N-1} (2s_i - 1)(2s_{i+t} - 1) = \begin{cases} N, t = 0 \\ K, 1 \leq t \leq N - 1 \end{cases}$$

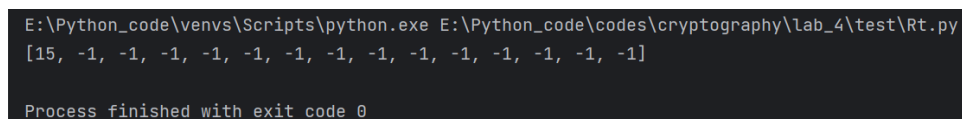
编写程序验证该公式的程序代码如下所示：

```
1 # S = [1, 0, 1, 0, 1, 1, 1, 1, 0, 0, 0, 1, 0, 0, 1]
2 S = [0, 0, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 0, 0, 1]
3 N = 15
4 R = []
5
6 for t in range(N):
7     Rt = 0
8     for i in range(N):
9         Rt += (2 * S[i] - 1) * (2 * S[(i + t) % N] - 1)
10    R.append(Rt)
11 print(R)
```

代码 1: 序列 1 验证程序

(1) 序列 1:

在序列 1 中输出序列为 101011110001001..., 且周期 $N=15$, 程序的运行结果如下图所示



```
E:\Python_code\venvs\Scripts\python.exe E:\Python_code\codes\cryptography\lab_4\test\Rt.py
[15, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1]

Process finished with exit code 0
```

图 3: 序列 1 验证结果

观察结果可得，整数 K 为-1，当 t 为 0 时公式的结果为 $N=15$ ， t 不为 0 时程序的运行结果为 $K=-1$ ，因此序列 1 满足自相关性测试

(2) 序列 2: 在序列 1 中输出序列为 000111101011001..., 且周期 $N=15$ ，程序

的运行结果如下图所示

```
E:\Python_code\venvs\Scripts\python.exe E:\Python_code\codes\cryptography\lab_4\test\Rt.py
[15, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1]

Process finished with exit code 0
```

图 4: 序列 2 验证结果

观察结果可得，整数 K 为-1，当 t 为 0 时公式的结果为 $N=15$ ， t 不为 0 时程序的运行结果为 $K=-1$ ，因此序列 2 满足自相关性测试

4.4 编程实现 ZUC 算法

4.4.1 算法原理

祖冲之密码算法在逻辑上分为上中下三层，上层是 16 级线性反馈移位寄存器（LFSR）、中层是比特重组（BR）、下层是非线性函数（F），算法的整体结构如下图所示

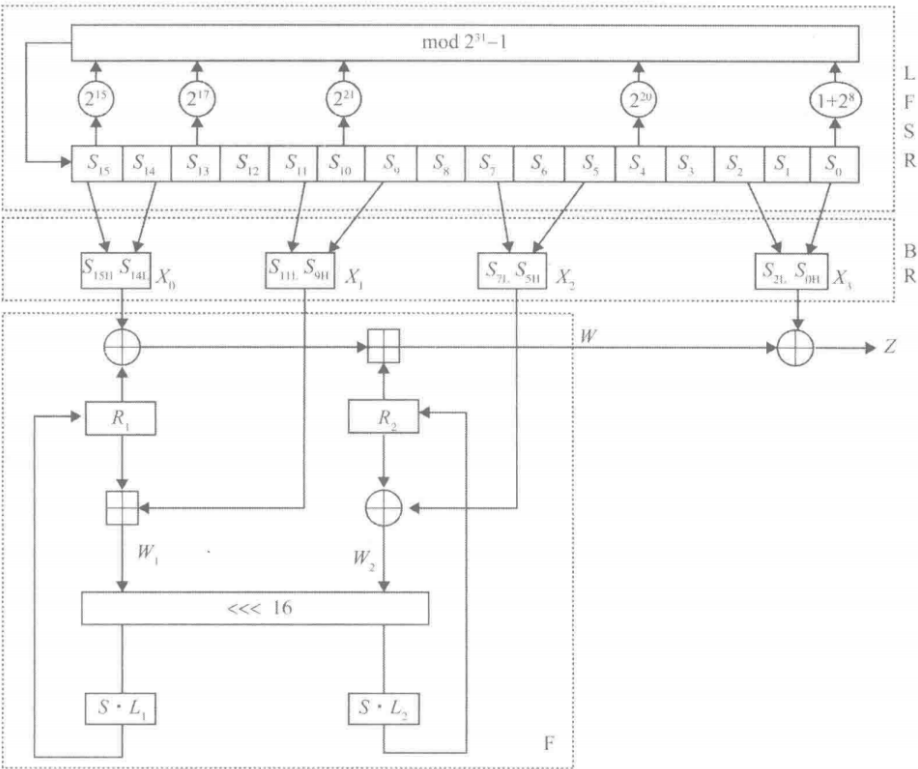


图 5: 祖冲之密码算法结构图

(1) 线性反馈移位寄存器 (LFSR) 以一个有限域 $GF(2^{31} - 1)$ 上的 16 次本原多项式为连接多项式, 因此, 其输出为 $GF(2^{31} - 1)$ 上的 m 序列, 具有良好的随机性。线性反馈移位寄存器 (LFSR) 的输出作为中层比特重组 (BR) 的输入

(2) 比特重组 (BR) 从线性反馈移位寄存器 (LFSR) 的状态中取出 128 位, 拼凑成 4 个 32 位字 (X_0, X_1, X_2, X_3), 供下层的非线性函数 F 和输出密钥序列使用。

(3) 非线性函数 (F) 从中层的比特重组 (BR) 接收 3 个 32 位字 (X_0, X_1, X_2, X_3) 作为输入, 经过内部的异或、循环移位和模 2^{32} 运算, 以及两个非线性 S 盒变换, 最后输出一个 32 位字 W 。由于非线性函数 F 是祖冲之密码算法中唯一的非线性部件, 所以非线性函数 F 就成为了确保祖冲之密码安全性的关键。

(4) 最后, 非线性函数 F 输出的 W 与比特重组 BR 输出 X_3 进行异或操作, 形成祖冲之密码的输出密钥字序列 Z 。

4.4.2 算法步骤

(1) 线性反馈移位寄存器:

线性移位寄存器的初始化模式下的程序代码片段如下所示

```
1 def LFSRWithInitMode(u):
2     """线性反馈移位寄存器初始化模式"""
3     v = (2 ** 15 * S[15] + 2 ** 17 * S[13] + 2 ** 21 * S[10] + 2 ** 20 * S[4] +
4         (1 + 2 ** 8) * S[0]) % (2 ** 31 - 1)
5     S.append((v + u) % (2 ** 31 - 1))
6     if S[16] == 0:
7         S[16] = 2 ** 31 - 1
8     S.pop(0)
```

代码 2: LFSR 初始化模式

在初始化模式下, LFSR 接受一个 31 比特字 u , u 是由非线性函数 F 的 32 比特输出 W 通过舍弃最低为比特得到的。

程序首先计算 v 的值, 将 S 中第 15、13、10、4、0 项乘对应的系数后与 $2^{31} - 1$

取模。

然后计算 S_{16} 的值，将计算得到的 v 和输入的 U 相加后与 $2^{31} - 1$ 取模。

然后判断 S_{16} 的值，若其值为 0，则将其值置为 $2^{31} - 1$ 。

最后将每一项的 S_i 的值赋值给其低位 S_{i-1} 。

线性反馈移位寄存器的工作模式下的程序代码片段如下所示

```
1 def LFSRWithWorkMode():
2     """线性反馈移位寄存器工作模式"""
3     S.append(
4         (2 ** 15 * S[15] + 2 ** 17 * S[13] + 2 ** 21 * S[10] + 2 ** 20 * S[4] +
5          (1 + 2 ** 8) * S[0]) % (2 ** 31 - 1))
6     if S[16] == 0:
7         S[16] = 2 ** 31 - 1
8     S.pop(0)
```

代码 3: LFSR 工作模式

在工作模式下相比于初始化模式，程序直接将计算得到的 V 作为 S_{16} 的值进行之后的操作，其余部分与初始化模式相同，这样的目的在于使得线性反馈移位寄存器的状态随机化

(2) 比特重组 BR:

比特重组部分的程序片段如下所示

```
1 def BitReconstruction():
2     """比特重组"""
3     X[0] = splicing_word(H_15(S[15]), L(S[14]))
4     X[1] = splicing_word(L(S[11]), H_15(S[9]))
5     X[2] = splicing_word(L(S[7]), H_15(S[5]))
6     X[3] = splicing_word(L(S[2]), H_15(S[0]))
```

代码 4: 比特重组 BR

比特重组部分的作用是从 LFSR 的寄存器单元中抽取 128 比特组成 4 个 32 比特字

X_0, X_1, X_2, X_3 。其具体的计算过程是将 S_{15} 的高 16 位与 S_{14} 的低 16 位进行首尾拼接得到 X_0 ; 然后将 S_{11} 的低 16 位与 S_9 的高 16 位进行首尾拼接得到 X_1 ; 然后将 S_7 的低 16 位与 S_5 的高 16 位进行首尾拼接得到 X_2 ; 最后将 S_2 的低 16 位与 S_0 的高 16 位进行首尾拼接得到 X_3 。

(3) 非线性函数 F

非线性函数 F 的内部包含 2 个 32bit 存储单元 R1, R2, F 的输入为来自比特重组的三个 32bit 字 X_0, X_1, X_2 , 输出为一个 32bit 字 W。

非线性函数 F 的实现代码如下所示

```
1 def F(X0, X1, X2):
2     """非线性函数F"""
3     global R1, R2, W
4     W = mod32(X0 ^ R1, R2)
5     W1 = mod32(R1, X1)
6     W2 = R2 ^ X2
7
8     R1 = S_box(L1(splicing_word(L(W1), H_16(W2))))
9     R2 = S_box(L2(splicing_word(L(W2), H_16(W1))))
```

代码 5: 非线性函数 F

该函数首先将输入的 X_0 与其内部的 R1 进行异或操作然后与 R2 进行 $\text{mod}(2^{32})$ 加法, 得到结果 W

然后将 R1 与 X_1 进行 $\text{mod}(2^{32})$ 加法, 得到结果 W1, 然后将 R2 与 X_2 进行异或操作得到 W2。

最后将 W1 的低 16 位与 W2 的高 16 位首尾拼接之后进行比特线性变换 1, 然后进行 S 盒变换得到结果 R1, 将 W2 的低 16 位与 W1 的高 16 位首尾拼接之后进行比特线性变换 2, 然后进行 S 盒变换得到结果 R2。

(4) S 盒变换 S_box

在 ZUC 算法中的 S 盒变换由 4 个并置的 8 进 8 出的单个 S 盒构成, S 盒变换

接受一个 4 字节的输入，对于第 0、2 字节采用 S0 盒，对于第 1, 3 字节采用 S1 盒，最后输出一个 4 字节的输出。在单个的 S 盒中接收一个字节的输入，将字节高位作为行号，字节低位作为列号，在 S 盒表中查找对应项进行输出。

S 盒的实现代码如下所示

```
1 def S_0(x_byte):
2     """使用S0盒进行替换"""
3     row = x_byte >> 4
4     nuw = x_byte & 0xf
5     result = S0[row][nuw]
6     # print(int_hex(result))
7     return result
8
9
10 def S_1(x_byte):
11     """使用S1盒进行替换"""
12     row = x_byte >> 4
13     nuw = x_byte & 0xf
14     result = S1[row][nuw]
15     # print(int_hex(result))
16     return result
17
18
19 def S_box(X):
20     """S盒变换"""
21     bytes = [0, 0, 0, 0]
22     result = [0, 0, 0, 0]
23
24     # 对于每一个字节进行S盒替换
25     bytes[0] = X >> 24
```

```

26     result[0] = S_0(bytes[0])
27
28     bytes[1] = (X >> 16) & 0xff
29     result[1] = S_1(bytes[1])
30
31     bytes[2] = (X >> 8) & 0xff
32     result[2] = S_0(bytes[2])
33
34     bytes[3] = X & 0xff
35     result[3] = S_1(bytes[3])
36
37     ans = (result[0] << 24) | (result[1] << 16) | (result[2] << 8) | result[3]
38     # print(int_hex(ans))
39     return ans

```

代码 6: S 盒变换

(5) 32 比特线性变换 L

线性比特变换的实现代码如下所示

```

1 def L1(X):
2     """比特线性变换"""
3     return X ^ rot(X, 2) ^ rot(X, 10) ^ rot(X, 18) ^ rot(X, 24) & 0xffffffff
4
5
6 def L2(X):
7     """比特线性变换"""
8     return X ^ rot(X, 8) ^ rot(X, 14) ^ rot(X, 22) ^ rot(X, 30) & 0xffffffff

```

代码 7: 比特线性变换

线性比特变换接收一个 32 位操作数 X ，对 X 操作的公式如下所示

$$L_1(x) = X \oplus (X \lll 2) \oplus (X \lll 10) \oplus (X \lll 18) \oplus (X \lll 24)$$

$$L_2(x) = X \oplus (X \lll 8) \oplus (X \lll 14) \oplus (X \lll 22) \oplus (X \lll 30)$$

(6) 密钥装入

密钥装入部分的实现代码如下所示

```

1 def get_in_key(key, iv):
2     """密钥装入"""
3     global S
4     for i in range(16):
5         S[i] = splicing_s(key[i], D[i], iv[i])

```

代码 8: 密钥装入

在密钥装入过程中是将 128 位的初始密钥 KEY 和 128 位的初始向量 IV 拓展为 16 个 31 比特字作为 LFSR 寄存器单元变量 S 的初始状态，在每一次拓展中选取 KEY 中的 8 位，再选取 240 比特常量 D 中的 15 位，最后选取 IV 中的 8 位拼凑成 S 中的一项。

4.4.3 算法运行

算法首先进行初始化阶段，执行密钥装入模块将 128 位的初始密钥 KEY 和 128 位的初始向量 IV 拓展为 16 个 31 比特字装入 LFSR 的寄存器单元变量 S 中之后作为 LFSR 的初态然后将非线性函数 F 中的 32 位存储单元 R1 和 R2 置为 0，随后重复 32 次依次进行比特重组、非线性函数变换 F 和线性反馈移位寄存器 LFSR 的初始化模式。算法的初始化代码如下所示

```

1 def init(key, iv):
2     """算法运行初始化阶段"""
3     global R1, R2, W
4     # 密钥装入
5     get_in_key(key, iv)
6     # for s in S:
7     #     print(s)
8

```



```

9     for i in range(32):
10         BitReconstruction()
11         F(X[0], X[1], X[2])
12         LFSRWithInitMode(W >> 1)

```

代码 9: 算法初始化阶段

然后算法进入工作阶段，首先依次执行比特重组、非线性函数变换 F 和线性反馈移位寄存器 LFSR 之后将 F 的输出 W 舍弃，然后进入密钥输出阶段，将比特重组、非线性函数 F 运行后得到的 W 与 X3 异或得到 Z、LFSR 操作定义为一个节拍，每运行一个节拍将 Z 输出，最后得到的 Z 即为最终的密钥。算法工作阶段的代码如下所示

```

1 def work():
2     """工作阶段"""
3     BitReconstruction()
4     F(X[0], X[1], X[2])
5     LFSRWithWorkMode()
6
7     for i in range(keylen):
8         BitReconstruction()
9         F(X[0], X[1], X[2])
10        Z = int_hex(W ^ X[3])
11        print("第" + str(i) + "个节拍输出的密钥为:" + str(Z))
12        Key_result.append(Z)
13        LFSRWithWorkMode()

```

代码 10: 算法工作阶段

4.5 思考与拓展阅读

4.5.1 定理

GF(2) 上的 n 级移位寄存器有 2^n 个状态，有 2^{2^n} 种不同的反馈函数

对于移位寄存器而言，每一个反馈系数都可以选择 0 或者 1，所以所有的反馈系数都参与排列组合可以有 2^n 中组合形式，所以也就有 2^n 个状态，对于每一个状态的输出都可以映射为 0 或 1，所以不同的反馈函数的数量应为 2^{2^n}

4.5.2 定理

GF(2) 上的 n 级移位寄存器中线性反馈函数只有 2^{n-1} 种

若 n 级线性移位寄存器的系数 g_n 为 0，则输出反馈不到 S_{n-1} ，因此在实际应用时应该保持其永远为 1，所以线性反馈函数应该只有 2^{n-1} 种

4.5.3 定理

n 次本原多项式的个数是 $\frac{\phi(2^n-1)}{n}$ ，其中 ϕ 是欧拉函数；（欧拉函数：小于或等于 x 的数中与 x 互质的数的数目）例如对于 $n=4$ ， $\frac{\phi(2^4-1)}{4} = 2$ 因此对于 4 元的线性移位寄存器，只有 2 个 LFSR 可以生成 m 序列，即实验（1）和（2）

一个本原多项式的根在 GF(2) 的扩展域中产生一个周期为 $2^n - 1$ 的序列。这意味着序列中的元素在达到 $2^n - 1$ 次幂时第一次回到起点。这相当于说，这个周期是与 $2^n - 1$ 互素的所有数的集合

欧拉函数 $\phi(2^n - 1)$ 给出了与 $2^n - 1$ 互素的数的个数。在本原多项式的上下文中，这表示 $2^n - 1$ 的周期内不同的状态数量

每个本原多项式在其周期内产生 n 个不同的状态（因为它是 n 次的），这些状态是彼此旋转等价的。因此 $\phi(2^n - 1)$ 给出的是周期内所有不同状态的总数，而我们需要将这个数字除以 n ，以排除那些由于旋转产生的重复状态。

所以可得 n 次本原多项式的个数是 $\frac{\phi(2^n-1)}{n}$

4.5.4 分析原因

在实际的序列密码设计中，人们往往选择项数较少的本原多项式构造 LFSR，例如三项式或五项式。试分析选择的原因。

(1) 硬件实现的简便性：项数较少的本原多项式意味着更简单的硬件实现。在硬件中，每一个额外的项都需要额外的逻辑门和连接。三项式或五项式的本原多项式可以显著减少所需的硬件资源，降低复杂性和成本。

(2) 计算效率：在软件实现中，项数较少的多项式意味着更少的计算步骤。这对性能有严格要求的应用中尤其重要，如实时加密通信或大容量数据加密。

(3) 可靠的统计性质：尽管本原多项式的项数较少，但如果正确选择，它们仍然可以产生具有良好统计性质的伪随机序列。这对于加密应用来说至关重要，因为它保证了序列的不可预测性和均匀分布。

(4) 易于分析和验证：简单的本原多项式使得对生成的序列进行数学分析和安全性验证更为容易。在密码学中，能够证明加密算法的强度是非常重要的，而复杂的算法往往更难以分析。

(5) 抵抗某些攻击的能力：虽然简单的本原多项式可能看起来更容易受到攻击，但如果正确使用，它们可以提供足够的安全性。此外，可以通过将多个简单的 LFSR 结合起来（如在非线性反馈移位寄存器中那样）来提高安全性，而不是增加单个 LFSR 中的项数。

(6) 可调性和灵活性：简单的本原多项式提供了一种在保持硬件和软件效率的同时调整和优化性能的方法。例如，通过改变反馈点的位置，可以生成不同的序列，而不需要改变整体设计。

4.5.5 定理

本原多项式一定是不可约多项式；反之则不一定成立。

假设存在一个本原多项式 $p(x)$ 不是不可约的，即它可以被分解为两个或更多次数较低的多项式的乘积。

由于 $p(x)$ 是本原的，其根 α 生成 $GF(2)$ 扩展域的循环群，具有阶 $2^n - 1$

如果 $p(x)$ 可分解，那么 α 也是这些较低次数多项式的根。这意味着 α 的阶必须小于 $2^n - 1$ ，因为较低次数多项式的根的阶最多是它们次数的 2 的幂减 1。

这与 α 的阶是 $2^n - 1$ 的事实矛盾。因此，本原多项式 $p(x)$ 必须是不可约的。

考虑不可约多项式 $q(x)$ 在 $GF(2)$ 中的一个根 β

β 的阶可能小于 $2^n - 1$ 。例如，如果 β 的阶是 $2^n - 1$ 的一个因子，那么 $q(x)$ 就不是本原的。

因此，尽管 $q(x)$ 是不可约的，它并不一定是本原的，因为其根的阶可能不是最大的。

4.5.6 常用的判定 $GF(2)$ 上的不可约多项式的方法

- (a) 如果 $f(x)$ 的常数项为 0，除 $f(x) = x$ 之外， $f(x)$ 一定可约
- (b) 如果 $f(x)$ 的项数为偶数，除 $f(x) = x+1$ 之外， $f(x)$ 一定可约
- (c) 如果 $f(x)$ 中各项的 x 的幂次都是 2 的倍数， $f(x)$ 一定可约
- (d) 如果 $\gcd(f(x), f'(x)) \neq 1$ ， $f(x)$ 一定可约
- (e) 如果 $f(x+1)$ 可约， $f(x)$ 一定可约
- (f) 如果 $x^n f(\frac{1}{x})$ 可约， $f(x)$ 一定可约

4.5.7 密码设计的初步尝试

(a) 设计 AES 类的 S 盒选择不同的不可约多项式，按照 AES S 盒的相同计算过程即可

按照 AES 的 S 盒的计算过程设计 S 盒的过程如下所示：

1. 选择不可约多项式。假设我们选择一个不同于 AES 的不可约多项式用于 $GF(2^8)$ 。我们可以选用 $m(x) = x^8 + x^6 + x^5 + x^2 + 1$ 。
2. 构建有限域。使用选定的多项式，我们可以构建一个 256 元素的伽罗瓦域

($GF(2^8)$)。在这个域中，每个元素都可以表示为一个多项式，其次数小于 8，并且所有的系数都是在 $GF(2)$ 中，即它们可以是 0 或 1。

3. 定义逆元素。对于域中的每个非零元素，我们需要找到它的乘法逆。这可以通过扩展的欧几里得算法来完成。逆元素是用于后续的仿射变换。

4. 设计仿射变换。仿射变换是一个线性变换加上一个常量偏移。假设我们选择一个简单的线性变换和一个随机的偏移，例如，我们可以将每个字节与一个固定的矩阵相乘并加上一个常量向量。

5. 组合所有步骤。最后，我们的 S 盒将是每个输入字节映射到其在 $GF(2^8)$ 中的逆，然后应用我们设计的仿射变换。

(b) 生成 m 序列选择不同的本原多项式，按照 LFSR 的构造过程即可

编写的计算 LFSR 的程序如下所示

```
1 import functools
2
3
4 def generate_m_sequence(primitive_poly, length):
5     # 初始化 LFSR 状态 (不全为 0)
6     lfsr = [1] * length
7
8     # 生成序列
9     sequence = []
10    for _ in range(2 ** length - 1):
11        # 将 LFSR 的当前状态作为序列的下一个元素
12        sequence.append(lfsr[-1])
13
14        # 计算新的反馈位
15        feedback_bit = functools.reduce(lambda x, y: x ^ y, [lfsr[i] for i in
16            range(length) if primitive_poly[i]])
```

```

17         # 更新LFSR状态
18         lfsr = [feedback_bit] + lfsr[:-1]
19
20     return sequence
21
22
23 # 本原多项式
24 primitive_poly = [1, 0, 0, 1, 1]
25
26 # 生成m序列
27 m_sequence = generate_m_sequence(primitive_poly, len(primitive_poly) - 1)
28
29 print(m_sequence)

```

代码 11: 计算 m 序列

选定本原多项式为 $x^4 + x + 1$ 运行该程序得到的结果如下图所示

```

E:\Python_code\venvs\Scripts\python.exe E:\Python_code\codes\cryptography\lab_4\get_m.py
[1, 1, 1, 1, 0, 1, 0, 1, 1, 0, 0, 1, 0, 0, 0]

Process finished with exit code 0

```

图 6: 计算 m 序列结果

五、实验结果与数据处理

运行祖冲之代码程序得到的结果如下图所示

```

E:\Python_code\venvs\Scripts\python.exe E:\Python_code\codes\cryptography\lab_4\ZUC\ZUC.py
第0个节拍输出的密钥为:14f1c272
第1个节拍输出的密钥为:3279c419
最终得到的所有的密钥为:
['14f1c272', '3279c419']

Process finished with exit code 0

```

图 7: 祖冲之密码算法结果

六、分析与讨论

(1) 通过学习序列密码的基本概念，我更深入地理解了密码学的基础知识，尤其是序列密码在安全通信中的重要性。

(2) 实验中，我掌握了线性移位寄存器的构造和操作，这让我了解到伪随机序列在加密中的应用和重要性。

(3) 非线性序列的学习让我认识到密码设计的复杂性和创新性。

(4) 通过熟悉伪随机性评价方法，我学会了如何评估一个密码算法的安全性。

(5) 实现 ZUC 算法的过程加深了我对流密码实现原理的理解。

七、教师评语