SHANDONG UNIVERSITY

密码工程第二次实验报告

姓名: 谢子洋
学院: 网络空间安全学院 (研究院)
专业: 网络空间安全
学号:202100460116

2023 年 11 月 28 日

目录

1 实验原理

1.1 Grain128 生成密钥流

Grain128 流密码主要由三部分组成, 分别为 NFSR,LFSR, 输出函数.

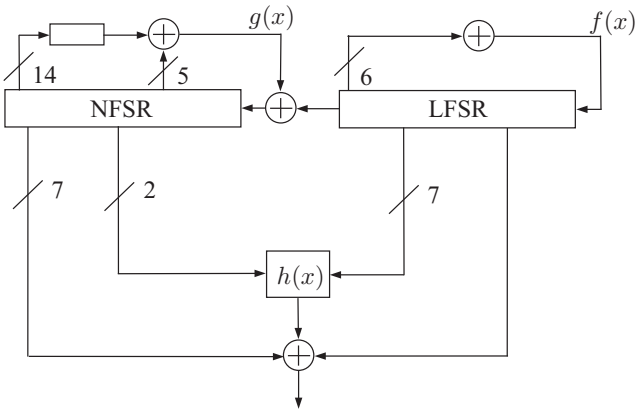

图 1: Grain128

1) NFSR 由 $b_i, b_{i+1}, ..., b_{i+127}$ 组成, 其非线性反馈生成式为:

$$\begin{aligned} b_{i+128} = {} & s_i + b_i + b_{i+26} + b_{i+56} + b_{i+91} + b_{i+96} \\ & + b_{i+3}b_{i+67} + b_{i+11}b_{i+13} + b_{i+17}b_{i+18} + \\ & + b_{i+27}b_{i+59} + b_{i+40}b_{i+48} + b_{i+61}b_{i+65} + \\ & + b_{i+68}b_{i+84}. \end{aligned} \tag{1.1}$$

2) LFSR 由 $s_i, s_{i+1}, ..., s_{i+127}$ 组成, 其线性反馈生成式为:

$$s_{i+128} = s_i + s_{i+7} + s_{i+38} + s_{i+70} + s_{i+81} + s_{i+96}. \tag{1.2}$$

3) 函数 $h(x)$ 定义为

$$h(x) = x_0x_1 + x_2x_3 + x_4x_5 + x_6x_7 + x_0x_4x_8. \tag{1.3}$$

其中 $x_0, x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8$ 依次对应 $b_{i+12}, s_{i+8}, s_{i+13}, s_{i+20}, b_{i+95}, s_{i+42}, s_{i+60}, s_{i+79}, s_{i+95}$

4) 输出函数定义为:

$$z_i = \sum_{j \in \mathscr{A}} b_{i+j} + h(x) + s_{i+93}. \tag{1.4}$$

其中 $\mathscr{A} = \{2, 15, 36, 45, 64, 73, 89\}$

1.2 Grain128 初始化

Grain128 算法在产生有效密钥流之前需要先进行初始化: 将 key 和 IV 加载到寄存器并进行 256 轮更新, 过程中输出函数要反馈到 LFSR 和 NFSR.

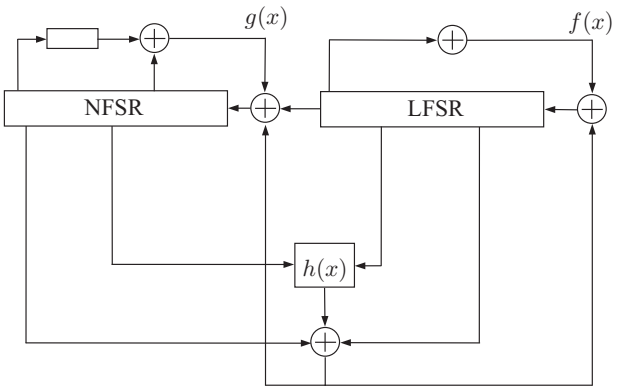

图 2: Grain128

1) **初始化寄存器初值**将 key 各 bit 以 $k_i, 0 \le i \le 127$ 表示, 将 IV 各 bit 以 $IV_i, 0 \le i \le 95$ 表示.

对于 128bit 的 NFSR, 令

$$b_i = k_i \ , 0 \le i \le 127 \tag{1.5}$$

对于 128bit 的 LFSR, 令

$$s_i = \begin{cases} IV_i, & 0 \le i \le 95 \\ 1, & 96 \le i \le 127 \end{cases} \tag{1.6}$$

2) **进行初始化循环**

在加载 key 和 IV 比特后, 密码算法进行 256 次而不产生任何密钥流。输出函数被反馈到 LFSR 和 NFSR, 并与输入异或产生新 bit.

$$\begin{aligned} &for \ \ i = 0 \, to \, 255 \\ &\quad NFSR \Leftarrow \{b_{i+1}, b_{i+2}, ..., b_{i+128} \oplus z_i\} \\ &\quad LFSR \Leftarrow \{s_{i+1}, s_{i+2}, ..., s_{i+128} \oplus z_i\} \end{aligned} \tag{1.7}$$

2 实验过程

2.1 模块输入输出

输入时钟 clk 控制密钥流生成速度, 每当时钟上升沿模块产生新密钥 bit. 输入 key 和 IV 唯一确定密钥流. 因为算法存在初始化阶段, 因此并非任意时刻的密钥流都是可用的, 本文通过添加有效标志 valid 输出加以区分, 只有当 valid 为 1 时输出密钥流才是有效的.

```
module Grain128(
  input clk,          // 输入时钟
  input [0:127] key,  // 输入key
  input [0:95] IV,    // 输入IV
  output wire stream,// 输出密钥流
  output valid        // 密钥流是否有效
  );
endmodule
```

2.2 寄存器设初值

设计两个 128bit 寄存器分别代表 NFSR 和 LFSR。

```
reg [0:127]NFSR;
reg [0:127]LFSR;
wire[0:127]b,b_128;
wire[0:127]s,s_128;
assign b=NFSR;
assign s=LFSR;
```

Grain128 流密码算法要求 NFSR 和 LFSR 在算法开始时分别设定规定初始值, 因此我们在 initial 语句块中为寄存器设定初始值, 并设定时延保证寄存器初始化完全.

```
initial begin
    #10;
    NFSR[0:127] =key[0:127];
    LFSR[0:95]  =IV[0:95];
    LFSR[96:127]={32{1'b1}};
end
```

2.3 算法初始化

设定寄存器初始值过程中时序电路不可运行, 否则会导致算法流程错误.

因此使用阻塞语句 wait() 延后时序电路的执行, 保证在寄存器赋完初值后时序电路才开始运行.

```
reg init=1'b0;
initial begin
    ......
    init=1'b1;
end
always @(posedge clk) begin
    wait (init) ;
    ......
end
```

寄存器设定初始值后需要进行 256 轮以初始化, 过程中算法输出值要与寄存器输入值异或, 并作为新的输入更新寄存器. 因此本文添加寄存器"count" 用以计数, 初值设为 256, 每经一个时钟上升沿将该寄存器数值减 1, 直到寄存器数值为 0.

初期"valid" 为 0 表示输出密钥流无效, 每轮输出的 bit 要与反馈函数输出异或, 作为寄存器的新输入. 当"count" 寄存器数值为 0 时, 数字电路完成初始化, 输出信号"valid" 变为 1, 开始输出有效密钥流.

```
reg [8:0] count;//计数
assign valid=(¬(count[0]|count[1]|count[2]|
                count[3]|count[4]|count[5]|
                count[6]|count[7]|count[8]));
initial begin
    ......
    count[8:0]=9'b1_0000_0000;
end
always @(posedge clk) begin
    if(¬valid) count=count-1;
    ......
    NFSR[127]=b_128^(valid?1'b0:resultBit);
    LFSR[127]=s_128^(valid?1'b0:resultBit);
end
```

2.4 产生加密流

将电路设计抽象并简化, 使用软件绘制如下设计图.

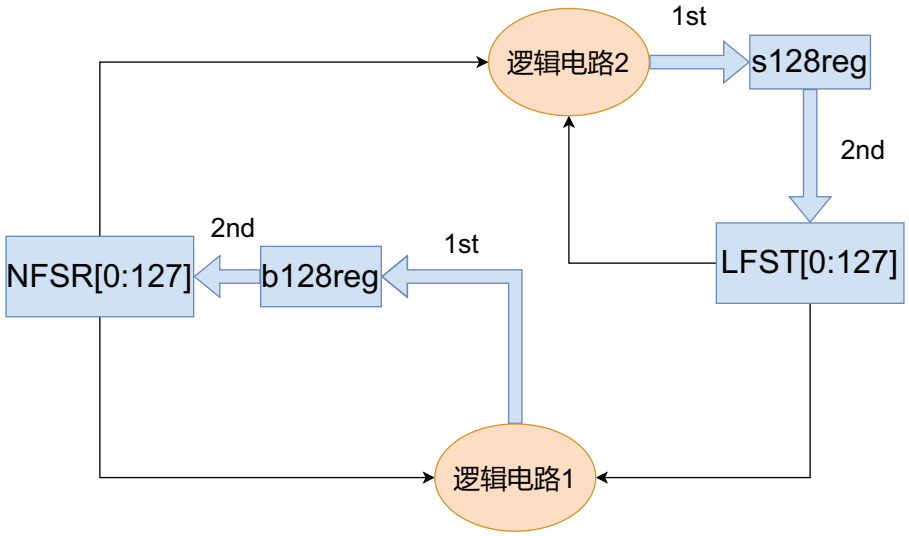

图 3: Grain128 内部更新设计

除 NFSR 和 LFSR 使用寄存器外, 其他部分均可用逻辑电路表示. 因此可使用 assign 语句将电路中所有逻辑电路表示出来.

```
assign b=NFSR;
assign s=LFSR;
assign b_128=s[0]^b[0]^b[26]^b[56]^b[91]^b[96]
             ^(b[3]&b[67])^(b[11]&b[13])^(b[17]&b[18])
             ^(b[27]&b[59])^(b[40]&b[48])^(b[61]&b[65])
             ^(b[68]&b[84]);
assign s_128=s[0]^s[7]^s[38]^s[70]^s[81]^s[96];
assign x={b[12],s[8],s[13],s[20],b[95],s[42],s[60],s[79],s[95]};
assign hx=(x[0]&x[1])^(x[2]&x[3])^(x[4]&x[5])^(x[6]&x[7])^(x[0]&x[4]&x[8]);
assign resultBit=hx^s[93]^b[2]^b[15]^b[36]^b[45]^b[64]^b[73]^b[89];
assign stream=resultBit&valid;
```

模块输入包含时钟 clk, 每当检测到 clk 的上升沿就进行一次寄存器的更新.

```
always @(posedge clk) begin
  NFSR[0:126]=NFSR[1:127];
  LFSR[0:126]=LFSR[1:127];
  NFSR[127]=b_128^(valid?1'b0:resultBit);
  LFSR[127]=s_128^(valid?1'b0:resultBit);
end
```

2.5 完整代码

最后给出完整代码:

```
module Grain128(
  input clk, // 输入时钟
  input [0:127] key,
  input [0:95] IV,
  output wire stream,
  output valid
  );
  reg init=1'b0;
  reg [8:0] count;//计数
  reg [0:127]NFSR;
  reg [0:127]LFSR;
  wire[0:127]b,b_128;
  wire[0:127]s,s_128;
  wire [0:8]x;
  wire hx,resultBit;

  assign b=NFSR;
  assign s=LFSR;
  assign b_128=s[0]^b[0]^b[26]^b[56]^b[91]^b[96]
               ^(b[3]&b[67])^(b[11]&b[13])^(b[17]&b[18])
               ^(b[27]&b[59])^(b[40]&b[48])^(b[61]&b[65])
               ^(b[68]&b[84]);
  assign s_128=s[0]^s[7]^s[38]^s[70]^s[81]^s[96];
  assign x={b[12],s[8],s[13],s[20],b[95],s[42],s[60],s[79],s[95]};
  assign hx=(x[0]&x[1])^(x[2]&x[3])^(x[4]&x[5])^(x[6]&x[7])^(x[0]&x[4]&x[8]);
  assign resultBit=hx^s[93]^b[2]^b[15]^b[36]^b[45]^b[64]^b[73]^b[89];
  assign stream=resultBit&valid;
  assign valid=(¬(count[0]|count[1]|count[2]|
               count[3]|count[4]|count[5]|
               count[6]|count[7]|count[8]));

  initial begin
      #10;
      NFSR[0:127] =key[0:127];
      LFSR[0:95]  =IV[0:95];
      LFSR[96:127]={32{1'b1}};
      count[8:0]=9'b1_0000_0000;
      init=1'b1;
  end
  always @(posedge clk) begin
      wait (init) ;
      if(¬valid) count=count-1;
      NFSR[0:126]=NFSR[1:127];
      LFSR[0:126]=LFSR[1:127];
      NFSR[127]=b_128^(valid?1'b0:resultBit);
      LFSR[127]=s_128^(valid?1'b0:resultBit);
  end
endmodule
```

3 实验结果

论文 [1] 中给出了部分测试实例, 具体数值如下.

表 1: 测试实例

Key	00000000000000000000000000000000
IV	000000000000000000000000
Keystream	0fd9deefeb6fad437bf43fce35849cfe
Key	0123456789abcdef123456789abcdef0
IV	0123456789abcdef12345678
Keystream	db032aff3788498b57cb894fffb6bb96

本文以此作为测试数据并编写了 Grain128 模块的 testbench, 具体代码如下.

```
`timescale  1ns / 1ps
`include"Grain128.v"
module tb_Grain128;
  parameter PERIOD  = 10;
  reg   clk                                  = 0 ;
  reg   [0:127]  key                         = 0 ;
  reg   [0:95]  IV                           = 0 ;
  wire  stream                               ;
  wire  valid                                ;
  Grain128  u_Grain128 (
      .clk                    ( clk             ),
      .key                    ( key     [0:127] ),
      .IV                     ( IV      [0:95]  ),

      .stream                 ( stream          ),
      .valid                  ( valid           )
  );
  always#30 clk=¬clk;
  initial begin
      $dumpfile("wave.vcd");
      $dumpvars;
      clk=0;

      key[0:127]={128{1'b0}};
      IV[0:95]={96{1'b0}};
      #30000;

      // key[0:127]=128'h0123456789abcdef123456789abcdef0;
      // IV[0:95]=96'h0123456789abcdef12345678;
      // #30000;

      $finish;
  end
endmodule
```

运行 testbench, 得到测试结果如下:

(1) 测试数据 1.

设定 key 和 IV 为全 0, 编译 testbench 并运行, 输出波形图如下:

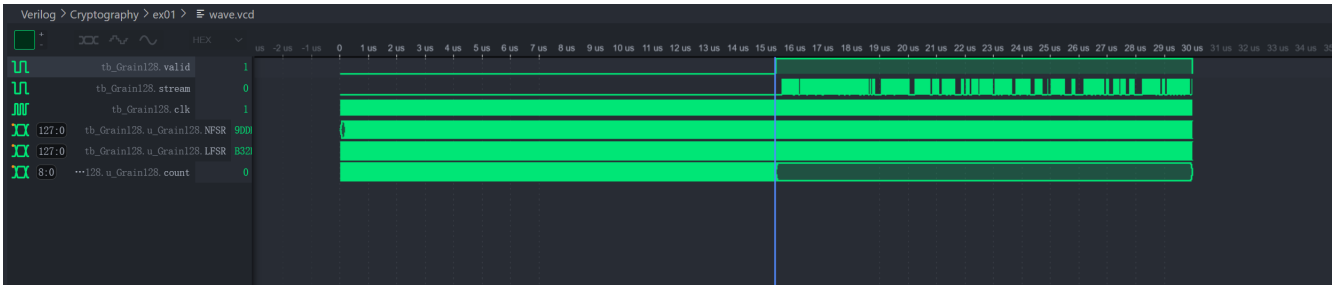

图 4: 波形图

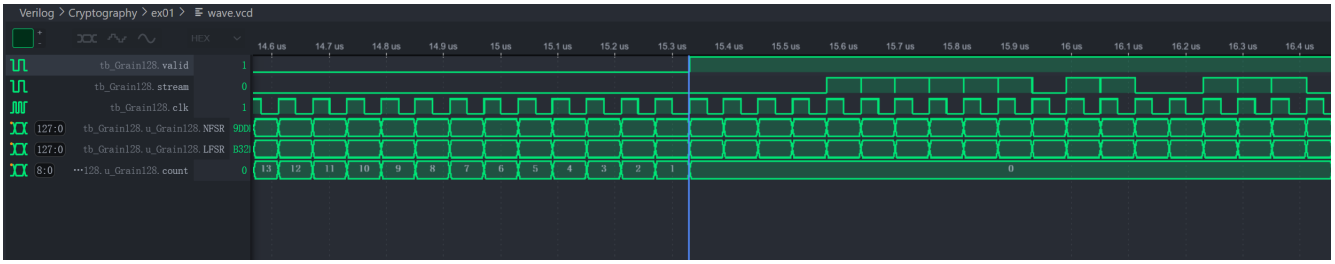

图 5: 波形图

初始条件下 valid 为 0, 表明该时段算法正在初始化. 可观察到当 count 寄存器数值变为 0 时, valid 变为 1, 表明该时段正在产生有效的秘钥流。观察"stream" 变量, 可知生成的密钥流为:

$$0b\ 0000\ 1111\ 1101\ 1001\cdots$$

与正确结果保持一致.

(2) 测试数据 2.

设定

Key=0123456789abcdef123456789abcdef0

IV =0123456789abcdef12345678

编译并运行 testbench, 得到波形图如下

图 6: 波形图

观察可知产生的密钥流为:

$$0b\ 1101\ 1011\ 0000\ 0011\cdots$$

与正确结果一致.

参考文献

[1] M. Hell, T. Johansson, A. Maximov and W. Meier, "A Stream Cipher Proposal: Grain-128," 2006 IEEE International Symposium on Information Theory, Seattle, WA, USA, 2006, pp. 1614-1618, doi: 10.1109/ISIT.2006.261549.

A Code

module.

```
module Grain128(
  input clk, // 输入时钟
  input [0:127] key,
  input [0:95] IV,
  output wire stream,
  output valid
  );
  reg init=1'b0;
  reg [8:0] count;//计数

  reg [0:127]NFSR;
  reg [0:127]LFSR;
  wire[0:127]b,b_128;
  wire[0:127]s,s_128;
  wire [0:8]x;
  wire hx,resultBit;

  assign b=NFSR;
  assign s=LFSR;
  assign b_128=s[0]^b[0]^b[26]^b[56]^b[91]^b[96]
              ^(b[3]&b[67])^(b[11]&b[13])^(b[17]&b[18])
              ^(b[27]&b[59])^(b[40]&b[48])^(b[61]&b[65])
              ^(b[68]&b[84]);
  assign s_128=s[0]^s[7]^s[38]^s[70]^s[81]^s[96];
  assign x={b[12],s[8],s[13],s[20],b[95],s[42],s[60],s[79],s[95]};
  assign hx=(x[0]&x[1])^(x[2]&x[3])^(x[4]&x[5])^(x[6]&x[7])^(x[0]&x[4]&x[8]);
  assign resultBit=hx^s[93]^b[2]^b[15]^b[36]^b[45]^b[64]^b[73]^b[89];
  assign stream=resultBit&valid;
  assign valid=(¬(count[0]|count[1]|count[2]|
              count[3]|count[4]|count[5]|
              count[6]|count[7]|count[8]));

  initial begin
      #10;
      NFSR[0:127] =key[0:127];
      LFSR[0:95]  =IV[0:95];
      LFSR[96:127]={32{1'b1}};
      count[8:0]=9'b1_0000_0000;
      init=1'b1;
  end
  always @(posedge clk) begin
      wait (init) ;
      if(¬valid) count=count-1;
      NFSR[0:126]=NFSR[1:127];
      LFSR[0:126]=LFSR[1:127];
      NFSR[127]=b_128^(valid?1'b0:resultBit);
      LFSR[127]=s_128^(valid?1'b0:resultBit);
  end
endmodule
```

testbench.

```
`timescale  1ns / 1ps
`include"Grain128.v"
module tb_Grain128;
  parameter PERIOD  = 10;
  reg   clk                                = 0 ;
  reg   [0:127]  key                       = 0 ;
  reg   [0:95]  IV                         = 0 ;
  wire  stream                             ;
  wire  valid                              ;
  Grain128  u_Grain128 (
      .clk                    ( clk             ),
      .key                    ( key     [0:127] ),
      .IV                     ( IV      [0:95]  ),

      .stream                 ( stream          ),
      .valid                  ( valid           )
  );
  always#30 clk=¬clk;
  initial begin
      $dumpfile("wave.vcd");
      $dumpvars;
      clk=0;

      key[0:127]={128{1'b0}};
      IV[0:95]={96{1'b0}};
      #30000;

      // key[0:127]=128'h0123456789abcdef123456789abcdef0;
      // IV[0:95]=96'h0123456789abcdef12345678;
      // #30000;

      $finish;
  end
endmodule
```