3.7 Approaches to Nonlinearity for Feedback Shift Registers

LFSRs are popular—in particular among electrical engineers and military—for several reasons:

- very easy implementation,
- extreme efficiency in hardware,
- good qualification as random generators for statistical applications and simulations,
- unproblematic operation in parallel even in large quantities.

But unfortunately from a cryptological view they are completely insecure if used naively. To capitalize their positive properties while escaping their cryptological weakness there are several approaches.

Approach 1, Nonlinear Feedback

Nonlinear feedback follows the scheme from Figure 1.7 with a nonlinear Boolean function f. There is a general proof that in realistic use cases NLFSRs are cryptographically useless if used in the direct naive way [6]. We won't pursue this approach here.

Approach 2, Nonlinear Output Filter

The nonlinear ouput filter (nonlinear feedforward) realizes the scheme from Figure 3.7. The shift register itself is linear, the Boolean function f, nonlinear.

The nonlinear ouput filter is a special case of a nonlinear combiner.

Approach 3, Nonlinear Combiner

The nonlinear combiner uses a "battery" of n LFSRs—preferably of different lengths—operated in parallel. The output sequences of the LFSRs serve as input of a Boolean function $f\colon \mathbb{F}_2^n \longrightarrow \mathbb{F}_2$, see Figure 3.8. (Sometimes also called "nonlinear feedforward.") We'll see in Section 3.8 how to cryptanalyze this random generator.

Approach 4, Output Selection/Decimation/Clocking

There are different ways of controlling a battery of n parallel LFSRs by another LFSR:

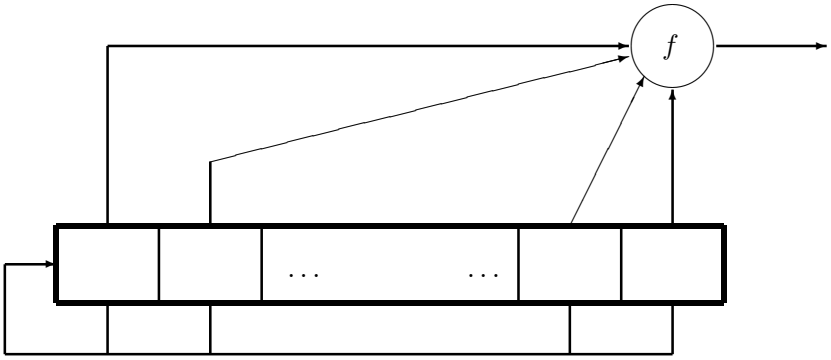

Figure 3.7: Nonlinear ouput filter for an LFSR

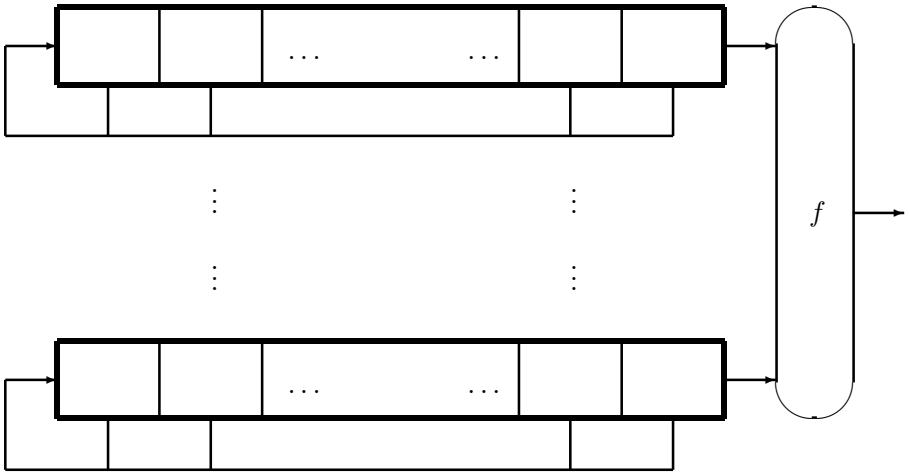

Figure 3.8: Nonlinear combiner

- **Output selection** takes the current output bit of exactly one of the LFSRs from the "battery", depending on the state of the auxiliary register, and outputs it as the next pseudorandom bit. More generally we could choose "r from n".
- For **decimation** one usually takes $n = 1$, and outputs the current bit of the one battery register only if the auxiliary register is in a certain state, for example its own current output is 1. Of course this kind of decimation applies to arbitrary bit sequences in an analogous way.
- For **clocking** we look at the state of the auxiliary register and depending on it decide which of the battery registers to step in the current cycle (and by how many positions), leaving the other registers in their current states (this mimics the control logic of rotor machines in classical cryptography).

These methods turn out to be special cases of nonlinear combiners if properly rewritten. Thus approach 3 represents the most important method of making the best of LFSRs.

The encryption standard A5/1 for mobile communications uses three LFSRs of lengths 19, 22 und 23, each with maximum possible period, and slightly differently clocked. It linearly (by simple binary addition) combines the three output streams. The—even weaker—algorithm A5/2 controls the clocking by an auxiliary register. Both variants can be broken on a standard PC in real-time.

The Bluetooth encryption standard E_0 uses four LFSRs and combines them in a nonlinear way. This method is somewhat stronger than A5, but also too weak for real security [7].

Example: The Geffe generator

The Geffe generator provides a simple example of output selection. Its description is in Figure 3.9. The output is x, if $z = 0$, and y, if $z = 1$. Expressed by a formula:

$$\begin{aligned} u &= \begin{cases} x, & \text{if } z = 0, \\ y, & \text{if } z = 1 \end{cases} \\ &= (1 - z)x + zy = x + zx + zy. \end{aligned}$$

This formula shows how to interpret the Geffe generator as a nonlinear combiner with a Boolean function $f\colon \mathbb{F}_2^3 \longrightarrow \mathbb{F}_2$ of degree 2. For later use we implement f in Sage sample 3.2.

For a concrete example we first choose three LFSRs of lengths 15, 16, 17, whose periods are $2^{15} - 1 = 32767$, $2^{16} - 1 = 65535$, and $2^{17} - 1 = 131071$. These are pairwise coprime. Combining their outputs (in each step)

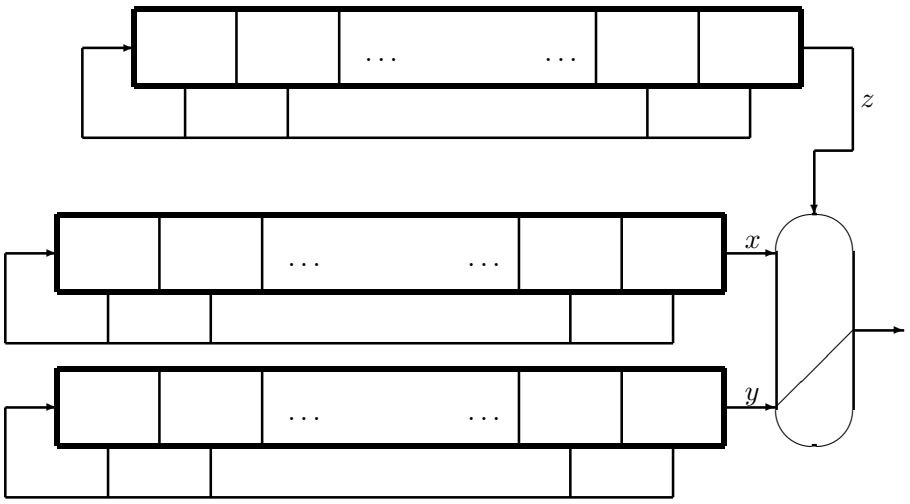

Figure 3.9: Geffe generator

Sage Example 3.2 The Geffe function

```
sage: geff = BoolF(str2bbl("00011100"),method="ANF")
sage: geff.printTT()
Value at 000 is 0
Value at 001 is 0
Value at 010 is 0
Value at 011 is 1
Value at 100 is 1
Value at 101 is 0
Value at 110 is 1
Value at 111 is 1
```

as bitblocks of length 3 yields a sequence with a period that has an impressive length of 281459944554495, about 300×10^{12} (300 European billions, for Americans this are 300 trillions).

Register 1 recursive formula $u_n = u_{n-1} + u_{n-15}$, taps `100000000000001`, initial state `011010110001001`.

Register 2 recursive formula $u_n = u_{n-2} + u_{n-3} + u_{n-5} + u_{n-16}$, taps `0110100000000001`, initial state `0110101100010011`.

Register 3 recursive formula $u_n = u_{n-3}+u_{n-17}$, taps `00100000000000001`, initial state `01101011000100111`.

Sage sample 3.3 defines the three LFSRs. We let each of the LFSRs generate a sequence of length 100, see Sage sample 3.4.

Sage Example 3.3 Three LFSRs

```
sage: reg15 = LFSR([1,0,0,0,0,0,0,0,0,0,0,0,0,0,1])
sage: reg15.setState([0,1,1,0,1,0,1,1,0,0,0,1,0,0,1])
sage: print(reg15)
Length: 15 | Taps: 100000000000001 | State: 011010110001001
sage: reg16 = LFSR([0,1,1,0,1,0,0,0,0,0,0,0,0,0,0,1])
sage: reg16.setState([0,1,1,0,1,0,1,1,0,0,0,1,0,0,1,1])
sage: print(reg16)
Length: 16 | Taps: 0110100000000001 | State: 0110101100010011
sage: reg17 = LFSR([0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,1])
sage: reg17.setState([0,1,1,0,1,0,1,1,0,0,0,1,0,0,1,1,1])
sage: print(reg17)
Length: 17 | Taps: 00100000000000001 | State: 01101011000100111
```

Sage Example 3.4 Three LFSR sequences

```
sage: nofBits = 100
sage: outlist15 = reg15.nextBits(nofBits)
sage: print(outlist15)
[1, 0, 0, 1, 0, 0, 0, 1, 1, 0, 1, 0, 1, 1, 0, 1, 1, 1, 0, 0,
 0, 0, 1, 0, 0, 1, 1, 0, 1, 1, 0, 1, 0, 0, 0, 0, 0, 1, 1, 1,
 0, 1, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 0, 1, 1, 0,
 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 1, 0, 1,
 0, 1, 1, 1, 0, 0, 0, 1, 1, 1, 0, 0, 1, 1, 0, 0, 1, 0, 1, 1]
sage: outlist16 = reg16.nextBits(nofBits)
sage: print(outlist16)
[1, 1, 0, 0, 1, 0, 0, 0, 1, 1, 0, 1, 0, 1, 1, 0, 0, 0, 1, 1,
 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 1, 1,
 1, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 1, 1, 0,
 1, 1, 1, 1, 0, 1, 0, 0, 1, 0, 0, 1, 1, 1, 1, 0, 0, 1, 0, 1,
 1, 0, 1, 1, 1, 1, 0, 0, 1, 0, 1, 1, 1, 0, 0, 1, 0, 0, 0, 1]
sage: outlist17 = reg17.nextBits(nofBits)
sage: print(outlist17)
[1, 1, 1, 0, 0, 1, 0, 0, 0, 1, 1, 0, 1, 0, 1, 1, 0, 0, 0, 1,
 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 0,
 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0,
 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1,
 0, 1, 0, 1, 0, 1, 0, 1, 1, 0, 0, 1, 0, 1, 1, 0, 0, 1, 1, 0]
```

The three sequences of length 100 are:

```
10010 00110 10110 11100 00100 11011 01000 00111 01101 10000
00101 10110 11111 11001 00100 10101 01110 00111 00110 01011

11001 00011 01011 00011 00111 10000 00001 11011 10001 11000
00100 01110 11110 10010 01111 00101 10111 10010 11100 10001

11100 10001 10101 10001 00000 01100 11111 10110 11000 00111
00001 10000 00001 11111 10010 01001 01010 10110 01011 00110
```

In Sage sample 3.5 the GEFFE function combines them to the output sequence

```
11010 00111 00011 01101 00100 10011 00001 10011 10101 10000
00100 00110 11110 10010 00110 10101 00110 10011 01100 01001
```

Sage Example 3.5 The combined sequence

```
sage: outlist = []
sage: for i in range(0,nofBits):
....:     x = [outlist15[i],outlist16[i],outlist17[i]]
....:     outlist.append(geff.valueAt(x))
....:
sage: print(outlist)
[1, 1, 0, 1, 0, 0, 0, 1, 1, 1, 0, 0, 0, 1, 1, 0, 1, 1, 0, 1,
 0, 0, 1, 0, 0, 1, 0, 0, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0, 1, 1,
 1, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0,
 1, 1, 1, 1, 0, 1, 0, 0, 1, 0, 0, 0, 1, 1, 0, 1, 0, 1, 0, 1,
 0, 0, 1, 1, 0, 1, 0, 0, 1, 1, 0, 1, 1, 0, 0, 0, 1, 0, 0, 1]
```