Chapter 3

Feedback Shift Registers and Linear Complexity

As we saw in the last chapter LFSRs are cryptographically weak if naively used. Also nonlinear FSRs admit an efficient prediction algorithm via the Noetherian principle undermining their security.

In this chapter we'll look at LFSRs from the opposite direction: Given a bit sequence, how to generate it by an LFSR in an optimal way? The minimal length of such an LFSR will turn out to be a useful measure of predictability—even a very good measure except for a few outliers.

3.1 The Linear Complexity of a Bit Sequence

We consider bit sequences $u = (u_i)_{i \in \mathbb{N}} \in \mathbb{F}_2^{\mathbb{N}}$—for the moment infinite ones. We search an LFSR of smallest length that produces the sequence.

If the sequence is generated by an LFSR, it must be periodic. On the other hand every periodic sequence is generated by an LFSR whose length is the sum of the lengths of preperiod and period—namely by the **circular LFSR** that feeds back the bit where the period begins: If $u_{l+i} = u_{k+i}$ for $i \geq 0$, then the taps are $a_{l-k} = 1$, $a_i = 0$ else, as in Figure 3.1. This consideration shows:

Lemma 11 *A bit sequence $u \in \mathbb{F}_2^{\mathbb{N}}$ is generated by an LFSR if and only if it is (eventually) periodic.*

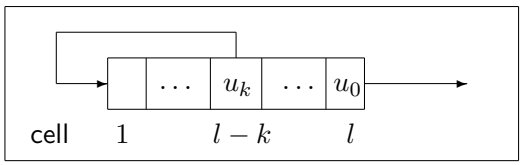


Figure 3.1: A circular LFSR

Definition The **linear complexity** $\lambda(u)$ of a bit sequence $u \in \mathbb{F}_2^{\mathbb{N}}$ is the minimal length of an LFSR that generates u.

For u constant 0 let $\lambda(u) = 0$, for a non-periodic u set $\lambda(u) = \infty$.

This concept of complexity uses the quite special machine model of an LFSR.

Remarks and examples

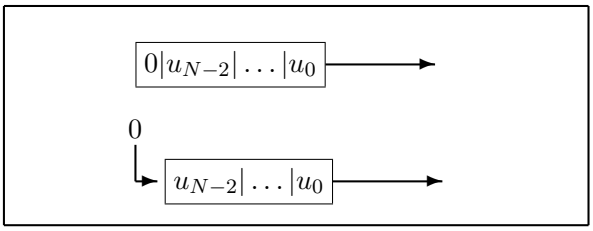
1. Let $\tau(u)$ be the sum of the lengths of the preperiod and the period of u. Assume that u is generated by an LFSR of length l. Then
$$\lambda(u) \leq \tau(u) \leq 2^l - 1 \quad \text{and} \quad \lambda(u) \leq l.$$
2. The periodically repeated sequence $0, \ldots, 0, 1$ ($l-1$ zeroes) has period l and linear complexity l. An LFSR of length $< l$ would start with the null vector as initial value and thus force the complete output sequence to zero.

For a finite bit sequence $u = (u_0, \ldots, u_{N-1}) \in \mathbb{F}_2^N$ the linear complexity is analogously defined. In particular $\lambda(u)$ is the minimum integer l for which there exist $a_1, \ldots, a_l \in \mathbb{F}_2$ with
$$u_i = a_1 u_{i-1} + \cdots + a_l u_{i-l} \quad \text{for } i = l, \ldots, N-1.$$

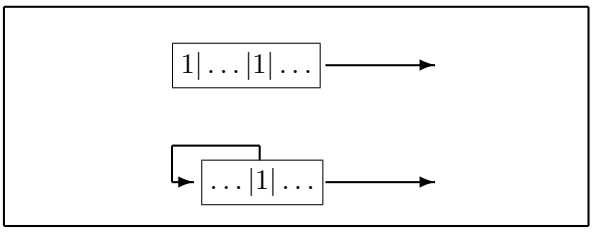
3. For $u \in \mathbb{F}_2^N$ we have $0 \leq \lambda(u) \leq N$.

4. $\lambda(u) = 0 \iff u_0 = \cdots = u_{N-1} = 0$.

5. $\lambda(u) = N \iff u = (0, \ldots, 0, 1)$. The implication "$\Longleftarrow$" follows as in remark 2. For the reverse direction assume $u_{N-1} = 0$. Then we can take the LFSR of length $N - 1$ with feedback constant 0—the two LFSRs



both generate the same output of length N. This contradiction shows that $u_{N-1} = 1$. Assume there is a 1 at an earlier position. Then we can take the LFSR of length $N - 1$ that feeds back exactly this position—the two LFSRs



both generate the same output up to length N.

6. From the first $2\lambda(u)$ bits of the sequence u all the following bits are predictable. (Note that the cryptanalyst who knows that many bits of the sequence, but no further bits, also doesn't know $\lambda(u)$. Therefore she doesn't know that her predictions will be correct from now on. This ignorance doesn't prevent her from correctly predicting bit for bit!)

3.2 Synthesis of LFSRs

In this section we treat the problem of how to find an LFSR of shortest length that generates a given finite bit sequence. In section 2.6 we described a method of finding linear relations for sequence elements from a quite general generator. This might result in an LFSR, but anyway the linear relations might change from step to step and there appears no easy way of getting an optimal LFSR.

Here we follow another approach that solves our problem in a surprisingly easy way: the BM-algorithm, named after Berlekamp (1968 in a different context) and Massey (1969).

We don't use any special properties of the field $\mathbb{F}_2$, so we work over an arbitrary field K. Our goal is to construct a homogeneous linear generator of the smallest possible recursion depth l that generates a given finite sequence $u \in K^N$.

We consider a homogeneous linear generator whose recursion formula is

$$u_k = a_1 u_{k-1} + \cdots + a_l u_{k-l} \quad \text{for } k = l, \ldots, N-1. \tag{1}$$

Its coefficient vector is $(a_1, \ldots, a_l) \in K^l$. The polynomial

$$\varphi = 1 - a_1 T - \cdots - a_l T^l \in K[T]$$

is called **feedback polynomial**.

Note Don't confuse this polynomial with the feedback function

$$s(u_0, \ldots, u_{l-1}) = a_1 u_{l-1} + \cdots + a_l u_0.$$

The feedback polynomial is the reciprocal polynomial of the characteristic polynomial

$$\chi = \mathrm{Det}(T \cdot 1 - A) = T^l - a_1 T^{l-1} - \cdots - a_l$$

of the companion matrix

$$A = \begin{pmatrix} 0 & 1 & \ldots & 0 \\ & \ddots & \ddots & \\ & & & 1 \\ a_l & a_{l-1} & \ldots & a_1 \end{pmatrix}.$$

These two polynomials are related by the formula

$$\varphi = T^l \cdot \chi(\frac{1}{T}).$$

Lemma 12 *Let the sequence $u = (u_0, \ldots, u_{n-1}) \in K^n$ be a segment of the output of the linear generator (1), but not the sequence $\hat{u} = (u_0, \ldots, u_n) \in K^{n+1}$. Then every homogeneous linear generator of length $m \geq 1$ that generates $\hat{u}$ has $m \geq n + 1 - l$.*

Proof. **Case 1:** $l \geq n$. Then obviously $l + m \geq n + 1$.

Case 2: $l \leq n - 1$. Assume $m \leq n - l$. We have

$$u_j = a_1 u_{j-1} + \cdots + a_l u_{j-l} \quad \text{for } l \leq j \leq n-1.$$

Let $(b_1, \ldots, b_m)$ be the coefficient vector of a homogeneous linear generator that produces $\hat{u}$. Then

$$u_j = b_1 u_{j-1} + \cdots + b_m u_{j-m} \quad \text{for } m \leq j \leq n.$$

We deduce

$$\begin{aligned} u_n &\neq a_1 u_{n-1} + \cdots + a_l u_{n-l} \\ &= \sum_{i=1}^{l} a_i \cdot \underbrace{\sum_{k=1}^{m} b_k u_{n-i-k}}_{u_{n-i}} \quad [\text{since } n - l \geq m] \\ &= \sum_{k=1}^{m} b_k \cdot \underbrace{\sum_{i=1}^{l} a_i u_{n-k-i}}_{u_{n-k}} = u_n, \end{aligned}$$

contradiction. ◇

Consider a sequence $u \in K^N$. For $0 \leq n \leq N$ let $\lambda_n(u) = \lambda_n$ be the smallest recursion depth for which a homogeneous linear generator exists that produces $(u_0, \ldots, u_{n-1})$.

Lemma 13 *For every sequence $u \in K^N$ we have:*

(i) $\lambda_{n+1} \geq \lambda_n$ *for all* n.

(ii) *There is a homogeneous linear generator of recursion depth λ_n that produces $(u_0, \ldots, u_n)$ if and only if $\lambda_{n+1} = \lambda_n$.*

(iii) *If there is no such generator, then*

$$\lambda_{n+1} \geq n + 1 - \lambda_n.$$

Proof. (i) Every generator that produces $(u_0, \ldots, u_n)$ a forteriori produces $(u_0, \ldots, u_{n-1})$.

(ii) follows from (i).

(iii) The precondition of Lemma 12 is true for every generator of $(u_0, \ldots, u_{n-1})$. ◇

Proposition 10 [Massey] *Let* $u \in K^N$ *and* $0 \le n \le N-1$. *Let* $\lambda_{n+1}(u) \neq \lambda_n(u)$. *Then*

$$\lambda_n(u) \le \frac{n}{2} \quad \textit{and} \quad \lambda_{n+1}(u) = n + 1 - \lambda_n(u).$$

Thus the linear complexity may jump only if λ_n (we often omit u in the notation) is "below the diagonal," and then it jumps to the symmetric position "above the diagonal." An illustration is in Figure 3.2.

Proof. First we consider the easy case $\lambda_n = 0$: Here $u_0 = \ldots = u_{n-1} = 0$. If $u_n = 0$, then $\lambda_{n+1} = \lambda_n = 0$, leaving nothing to prove. Otherwise $u_n \neq 0$, and then $\lambda_{n+1} = n + 1 = n + 1 - \lambda_n$ by remark 5 in 3.1.

In general the first statement follows from the second one: We have $\lambda_n < \lambda_{n+1}$, hence $2\lambda_n < \lambda_n + \lambda_{n+1} = n + 1$.

Now we prove the second statement by induction on n. In the case $n = 0$ we have $\lambda_0 = 0$—this case is already settled.

Now let $n \ge 1$. We may assume $l := \lambda_n \ge 1$. Let

$$u_j = a_1 u_{j-1} + \cdots + a_l u_{j-l} \quad \text{for } j = l, \ldots, n-1;$$

hence the feedback polynomial is

$$\varphi := 1 - a_1 T - \cdots - a_l T^l \in K[T].$$

Let the "n-th discrepancy" be defined as

$$d_n := u_n - a_1 u_{n-1} - \cdots - a_l u_{n-l}.$$

If $d_n = 0$, then the generator outputs u_n as the next element, and there is nothing to prove. Otherwise let $d_n \neq 0$. Let r be the length of the segment before the last increase of linear complexity, thus

$$t := \lambda_r < l, \quad \lambda_{r+1} = l.$$

By induction $l = r + 1 - t$. We have a relation

$$u_j = b_1 u_{j-1} + \cdots + b_t u_{j-t} \quad \text{for } j = t, \ldots, r-1,$$

the corresponding feedback polynomial is

$$\psi := 1 - b_1 T - \cdots - b_t T^t \in K[T],$$

and the corresponding r-th discrepancy,

$$d_r := u_r - b_1 u_{r-1} - \cdots - b_t u_{r-t} \neq 0.$$

In the case $t = 0$ we have $\psi = 1$ and $d_r = u_r$. Now we form the polynomial

$$\eta := \varphi - \frac{d_n}{d_r} \cdot T^{n-r} \cdot \psi = 1 - c_1 T - \cdots - c_m T^m \in K[T]$$

with $m = \deg \eta$. What is the output of the corresponding homogeneous linear generator? We have

$$\begin{aligned} u_j - \sum_{i=1}^{m} c_i u_{j-i} &= u_j - \sum_{i=1}^{l} a_i u_{j-i} - \frac{d_n}{d_r} \cdot \left[u_{j-n+r} - \sum_{i=1}^{t} b_i u_{j-n+r-i} \right] \\ &= 0 \quad \text{for } j = m, \ldots, n; \end{aligned}$$

for $j = m, \ldots, n-1$ this follows directly, for $j = n$ via the intermediate result $d_n - [d_n/d_r] \cdot d_r$. Hence the output is $(u_0, \ldots, u_n)$. Now we have

$$\lambda_{n+1} \leq m \leq \max\{l, n - r + t\} = \max\{l, n + 1 - l\}.$$

Since linear complexity grows monotonically we conclude $m > l$, and by Lemma 12 we get $m \geq n + 1 - l$. Hence $m = n + 1 - l$ and $\lambda_{n+1} = m$. This proves the proposition. ◇

Corollary 1 *If $d_n \neq 0$ and $\lambda_n \leq \frac{n}{2}$, then*

$$\lambda_{n+1} = n + 1 - \lambda_n > \lambda_n.$$

Proof. By Lemma 12 we have $\lambda_{n+1} \geq n + 1 - \lambda_n$, thus $\lambda_{n+1} \geq \frac{n}{2} + 1 > \lambda_n$. By Proposition 10 we conclude $\lambda_{n+1} = n + 1 - \lambda_n$. ◇

During the successive construction of a linear generator in the proof of the proposition, in each iteration step one of two cases occurs:

- $d_n = 0$: then $\lambda_{n+1} = \lambda_n$.
- $d_n \neq 0$: then
  - $\lambda_{n+1} = \lambda_n$ if $\lambda_n > \frac{n}{2}$,
  - $\lambda_{n+1} = n + 1 - \lambda_n$ if $\lambda_n \leq \frac{n}{2}$.

In particular we always have:

- If $\lambda_n > \frac{n}{2}$, then $\lambda_{n+1} = \lambda_n$.
- If $\lambda_n \leq \frac{n}{2}$, then $\lambda_{n+1} = \lambda_n$ or $\lambda_{n+1} = n + 1 - \lambda_n$.

By the way we found an alternative method of predicting LFSRs:

Corollary 2 *If $u \in \mathbb{F}_2^N$ is generated by an LFSR of length $\leq l$, then one such LFSR may be determined from $u_0, \ldots, u_{2l-1}$.*

Proof. Assume n is the first index $\geq 2l$ such that $d_n \neq 0$. Then $\lambda_n \leq l \leq \frac{n}{2}$, thus $\lambda_{n+1} = n + 1 - \lambda_n \geq l + 1$, contradiction. ◇

3.3 The Berlekamp-Massey Algorithm

The proof of Proposition 10 is constructive: It contains an algorithm that successively builds a linear generator. For the step from length n to length $n+1$ three cases (1, 2a, 2b) are possible:

Case 1 $d_n = 0$, hence the generator with feedback polynomial φ next outputs u_n: Then φ and l remain unchanged, and so remain ψ, t, r, d_r.

Case 2 $d_n \neq 0$, hence the generator with feedback polynomial φ doesn't output u_n as next element: Then we form a new feedback polynomial η whose corresponding generator outputs $(u_0, \ldots, u_n)$. We distinguish between:

a) $l > \frac{n}{2}$: Then $\lambda_{n+1} = \lambda_n$. We replace φ by η and leave l, ψ, t, r, d_r unchanged.

b) $l \leq \frac{n}{2}$: Then $\lambda_{n+1} = n + 1 - \lambda_n$. We replace φ by η, l by $n + 1 - l$, ψ by φ, t by l, r by n, d_r by d_n.

So a semi-formal description of the Berlekamp-Massey algorithm (or BM algorithm) is:

Input: A sequence $u = (u_0, \ldots, u_{N-1}) \in K^N$.

Output: The linear complexity $\lambda_N(u)$,

the feedback polynomial φ of a linear generator of length $\lambda_N(u)$ that produces u.

Auxiliary variables: n = current index, initialized by $n := 0$,

l = current linear complexity, initialized by $l := 0$,

φ = current feedback polynomial $= 1 - a_1 T - \cdots - a_l T^l$, initialized by $\varphi := 1$,

invariant condition: $u_i = a_1 u_{i-1} + \cdots + a_l u_{i-l}$ for $l \leq i < n$,

d = current discrepancy $= u_n - a_1 u_{n-1} - \cdots - a_l u_{n-l}$,

r = previous index, initialized by $r := -1$,

t = previous linear complexity,

ψ = previous feedback polynomial $= 1 - b_1 T - \cdots - b_t T^t$, initialized by $\psi := 1$,

invariant condition: $u_i = b_1 u_{i-1} + \cdots + b_t u_{i-t}$ for $t \leq i < r$,

d' = previous discrepancy $= u_r - b_1 u_{r-1} - \cdots - b_t u_{r-t}$, initialized by $d' := 1$,

η = new feedback polynomial,

m = new linear complexity.

Iteration steps: For $n = 0, \ldots, N-1$:

$d := u_n - a_1 u_{n-1} - \cdots - a_l u_{n-l}$
If $d \neq 0$
  $\eta := \varphi - \frac{d}{d'} \cdot T^{n-r} \cdot \psi$
  If $l \leq \frac{n}{2}$ [linear complexity increases]
    $m := n + 1 - l$
    $t := l$
    $l := m$
    $\psi := \varphi$
    $r := n$
    $d' := d$
  $\varphi := \eta$
Output: $\lambda_N(u) := l$ and φ

Of course we may output also the complete sequence (λ_n).

As an **example** we apply the algorithm to the sequence 001101110. The steps where $d \neq 0$, $l \leq \frac{n}{2}$, are tagged by "[!]".

preconditions of the step				actions
$n = 0$	$u_0 = 0$	$l = 0$	$\varphi = 1$	$d := u_0 = 0$
$r = -1$	$d' = 1$	$t =$	$\psi = 1$	
$n = 1$	$u_1 = 0$	$l = 0$	$\varphi = 1$	$d := u_1 = 0$
$r = -1$	$d' = 1$	$t =$	$\psi = 1$	
$n = 2$	$u_2 = 1$	$l = 0$	$\varphi = 1$	$d := u_2 = 1$ [!]
$r = -1$	$d' = 1$	$t =$	$\psi = 1$	$\eta := 1 - T^3$
				$m := 3$
$n = 3$	$u_3 = 1$	$l = 3$	$\varphi = 1 - T^3$	$d := u_3 - u_0 = 1$
$r = 2$	$d' = 1$	$t = 0$	$\psi = 1$	$\eta := 1 - T - T^3$
$n = 4$	$u_4 = 0$	$l = 3$	$\varphi = 1 - T - T^3$	$d := u_4 - u_3 - u_1 = -1$
$r = 2$	$d' = 1$	$t = 0$	$\psi = 1$	$\eta := 1 - T + T^2 - T^3$
$n = 5$	$u_5 = 1$	$l = 3$	$\varphi = 1 - T + T^2 - T^3$	$d := u_5 - u_4 + u_3 - u_2 = 1$
$r = 2$	$d' = 1$	$t = 0$	$\psi = 1$	$\eta := 1 - T + T^2 - 2T^3$

From now on the results differ depending on the characteristic of the base field K. First assume $\operatorname{char} K \neq 2$. Then the procedure continues as follows:

preconditions of the step	actions
$n = 6 \quad u_6 = 1 \quad l = 3$ $\varphi = 1 - T + T^2 - 2T^3$ $r = 2 \quad d' = 1 \quad t = 0 \quad \psi = 1$	$d := u_6 - u_5 + u_4 - 2u_3 = -2$ [!] $\eta = 1 - T + T^2 - 2T^3 + 2T^4$ $m := 4$
$n = 7 \quad u_7 = 1 \quad l = 4$ $\varphi = 1 - T + T^2 - 2T^3 + 2T^4$ $r = 6 \quad d' = -2 \quad t = 3$ $\psi = 1 - T + T^2 - 2T^3$	$d := u_7 - u_6 + u_5 - 2u_4 + 2u_3 = 3$ $\eta = 1 + \frac{1}{2}T - \frac{1}{2}T^2 - \frac{1}{2}T^3 - T^4$
$n = 8 \quad u_8 = 0 \quad l = 4$ $\varphi = 1 + \frac{1}{2}T - \frac{1}{2}T^2 - \frac{1}{2}T^3 - T^4$ $r = 6 \quad d' = -2 \quad t = 3$ $\psi = 1 - T + T^2 - 2T^3$	$d := u_8 + \frac{1}{2}u_7 - \frac{1}{2}u_6 - \frac{1}{2}u_5 - u_4 = -\frac{1}{2}$ [!] $\eta := 1 + \frac{1}{2}T - \frac{3}{4}T^2 - \frac{1}{4}T^3 - \frac{5}{4}T^4 + \frac{1}{2}T^5$ $m := 5$

The resulting sequence of linear complexities is

$$\lambda_0 = 0, \lambda_1 = 0, \lambda_2 = 0, \lambda_3 = 3, \lambda_4 = 3, \lambda_5 = 3, \lambda_6 = 3, \lambda_7 = 4, \lambda_8 = 4, \lambda_9 = 5,$$

and the generating formula is

$$u_i = -\frac{1}{2}u_{i-1} + \frac{3}{4}u_{i-2} + \frac{1}{4}u_{i-3} + \frac{5}{4}u_{i-4} - \frac{1}{2}u_{i-5} \quad \text{for } i = 5, \ldots, 8.$$

For char $K = 2$ the last three iteration steps look differently:

preconditions of the step	actions
$n = 6 \quad u_6 = 1 \quad l = 3$ $\varphi = 1 - T - T^2$ $r = 2 \quad d' = 1 \quad t = 0 \quad \psi = 1$	$d := u_6 - u_5 - u_4 = 0$
$n = 7 \quad u_7 = 1 \quad l = 3$ $\varphi = 1 - T - T^2$ $r = 2 \quad d' = 1 \quad t = 0 \quad \psi = 1$	$d := u_7 - u_6 - u_5 = 1$ [!] $\eta = 1 - T - T^2 - T^5$ $m := 5$
$n = 8 \quad u_8 = 0 \quad l = 5$ $\varphi = 1 - T - T^2 - T^5$ $r = 7 \quad d' = 1 \quad t = 3 \quad \psi = 1 - T - T^2$	$d := u_8 - u_7 - u_6 - u_3 = 1$ $\eta := 1 - T^3 - T^5$

In this case the sequence of linear complexities is

$$\lambda_0 = 0, \lambda_1 = 0, \lambda_2 = 0, \lambda_3 = 3, \lambda_4 = 3, \lambda_5 = 3, \lambda_6 = 3, \lambda_7 = 3, \lambda_8 = 5, \lambda_9 = 5,$$

and the generating formula is

$$u_i = u_{i-3} + u_{i-5} \quad \text{for } i = 5, \ldots, 8.$$

A Sage program for the char 2 case is in Sage Example 3.1. It uses the function `bmAlg` from Appendix C.2.

Figure 3.2 shows the growth of the linear complexities.

Sage Example 3.1 Applying the BM-algorithm

```
sage: u = [0,0,1,1,0,1,1,1,0]
sage: res = bmAlg(u)
sage: res
[[0, 0, 0, 3, 3, 3, 3, 3, 5, 5], T^5 + T^3 + 1]
```

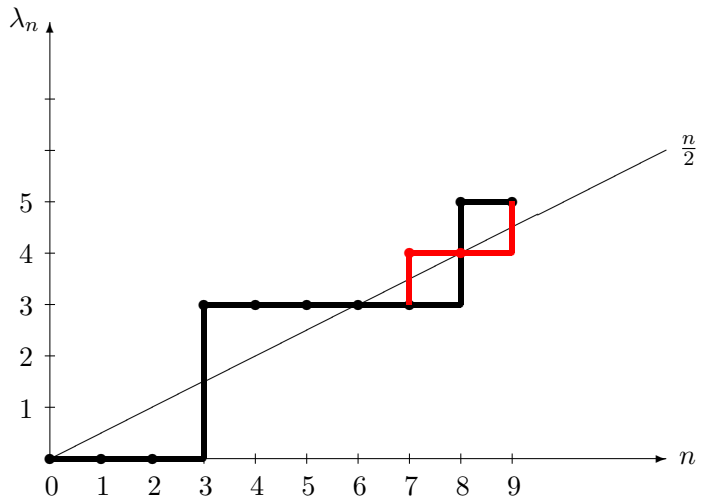


Figure 3.2: The sequence of linear complexities. The red line is for char $K \neq 2$.

The cost of the BM algorithm is $O(N^2 \log N)$.

The sequence $(\lambda_n)_{n\in\mathbb{N}}$ or (for finite output sequences) $(\lambda_n)_{0\leq n\leq N}$ is called the **linearity profile** of the sequence u.

Here is the linearity profile of the first 128 bits of the sequence that we generated by an LFSR in Section 1.10:

$$(0, 1, 1, 2, 2, 3, 3, 4, 4, 4, 4, 7, 7, 7, 7, 8, 8, 9, 9, 10, 10, 11, 11, 12,$$

$$12, 13, 13, 13, 13, 16, 16, 16, 16, \ldots),$$

its graphic representation is in Figure 3.3.

In Section 4.1 we'll generate a "perfect" pseudorandom sequence. The linearity profile of its first 128 bits is:

$$(0, 1, 1, 1, 1, 4, 4, 4, 4, 5, 5, 5, 5, 8, 8, 8, 8, 8, 8, 8, 8, 12, 12, 12, 12,$$

$$12, 12, 12, 12, 12, 17, 17, 17, 17, 17, 17, 18, 18, 18, 20, 20, 20, 21, 21,$$

$$22, 22, 22, 24, 24, 24, 24, 24, 24, 28, 28, 28, 28, 28, 29, 29, 30, 30, 31,$$

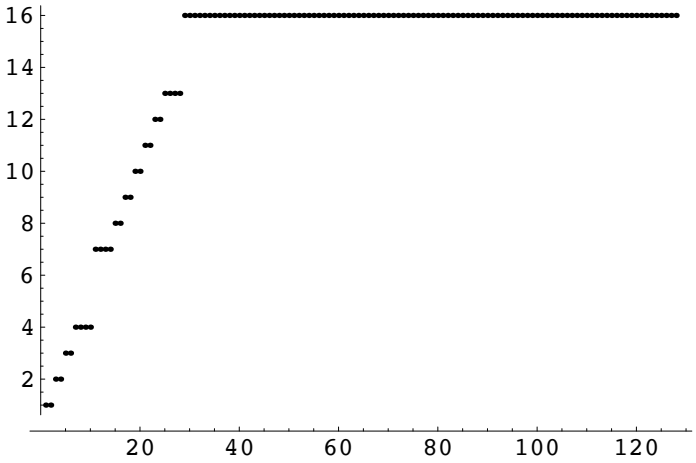


Figure 3.3: Linearity profile of an LFSR sequence

$$31, 32, 32, 32, 34, 34, 34, 34, 36, 36, 36, 37, 37, 38, 38, 39, 39, 40, 40,$$
$$41, 41, 41, 41, 41, 41, 46, 46, 46, 46, 46, 46, 47, 47, 48, 48, 49, 49, 50,$$
$$50, 50, 52, 52, 52, 53, 53, 54, 54, 54, 54, 54, 54, 54, 54, 61, 61, 61, 61,$$
$$61, 61, 61, 61, 61, 63, 63, 63, 64, 64),$$

graphically illustrated by Figure 3.4.

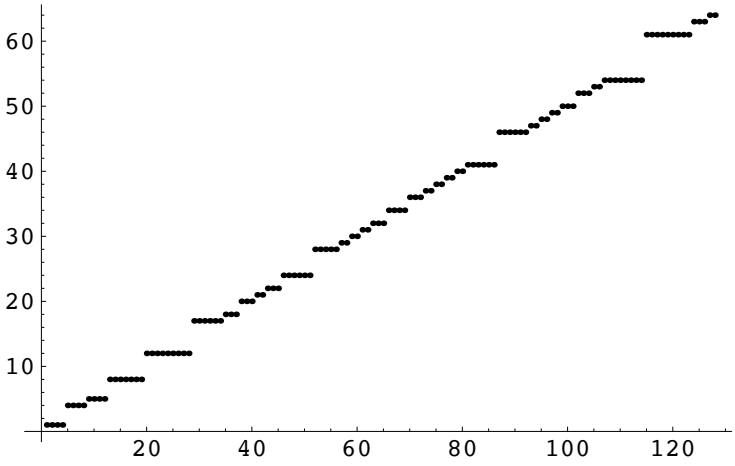


Figure 3.4: Linearity profile of a perfect pseudorandom sequence

In the second example we see a somewhat irregular oscillation around the diagonal, as should be expected for a "good" random sequence. The first example also shows a similar behaviour, but only until the linear complexity of the sequence is reached.

3.4 The BM Algorithm as a Cryptanalytic Tool

We revisit the cryptanalysis of an XOR ciphertext in Section 2.3 and explore how well the BM algorithm performs in this example following the cycle "construct – predict – adjust" as in Section 2.10. Remember the ciphertext:

```
10011100 10100100 01010110 10100110 01011101 10101110
01100101 10000000 00111011 10000010 11011001 11010111
00110010 11111110 01010011 10000010 10101100 00010010
11000110 01010101 00001011 11010011 01111011 10110000
10011111 00100100 00001111 01010011 11111101
```

For use with SageMath we provisionally fix its first 48 bits:

```
ciphtext = [1,0,0,1,1,1,0,0,1,0,1,0,0,1,0,0,0,1,0,1,0,
  1,1,0,1,0,1,0,0,1,1,0,0,1,0,1,1,1,0,1,1,0,1,0,1,1,1,0]
```

As in Section 2.3 we suspect that the cipher is XOR with a key stream from an LFSR, but now *of unknown length*. As before we guess that the text is in German and might begin with the word "Treffpunkt". To solve the cryptogram we need some bits of plaintext, say the first t letters (assumed in the 8-bit ISO 8859-1 character set), making up $8t$ bits of the key stream.

Let us tentatively start with two letters of plaintext: `Tr`, and the corresponding 16 keystream bits

```
          10011100 10100100  (ciphertext)
    Tr =  01010100 01110010  (assumed plaintext)
          -------- --------
          11001000 11010110  (keystream)
```

After attaching the Sage modules `Bitblock.sage`, `FSR.sage`, and `bmAlg.sage` from Appendix C (or Part II, Appendix E.1) we use the interactive commands

```
sage: kbits = [1,1,0,0,1,0,0,0,1,1,0,1,0,1,1,0]
sage: res = bmAlg(kbits)
sage: fbpol = res[1]; fbpol
T^8 + T^7 + T^5 + T^4 + T^3 + T^2 + T + 1
```

This result tells us that the shortest LFSR that generates our 16 keystream bits has length 8 and the taps 1, 2, 3, 4, 5, 7, 8 set. Next we initialize this LFSR in SageMath (note the reverse order of the bits in the initial state):

```
sage: coeff = [1,1,1,1,1,0,1,1]
sage: reg = LFSR(coeff)
sage: start = [0,0,0,1,0,0,1,1]
sage: reg.setState(start)
```

Using this LFSR we predict 32 more, hence altogether 48 tentative keystream bits:

```
sage: testkey = reg.nextBits(48); testkey
[1,1,0,0,1,0,0,0,1,1,0,1,0,1,1,0,1,0,0,1,1,1,0,0,
 0,0,0,0,0,1,1,0,0,0,0,1,0,1,1,1,0,1,1,1,1,0,0,0]
```

These tentative key bits yield 48 bits of experimental plaintext, represented by 6 bytes in decimal notation:

```
sage: testplain = xor(ciphtext,testkey)
sage: testtext = []
sage: for i in range(6):
    block = testplain[8*i:8*i+8]
    nr = bbl2int(block)
    testtext.append(nr)
sage: testtext
[84, 114, 202, 160, 74, 214]
```

or, written as ISO 8859-1 characters, "TrÊ␣JÖ" (where ␣ represents the non-breaking space)—a definitive failure.

So let us guess one more letter of plaintext: `Tre`, and use the corresponding 24 keystream bits

```
          10011100 10100100 01010110  (ciphertext)
   Tre =  01010100 01110010 01100101  (assumed plaintext)
          -------- -------- --------
          11001000 11010110 00110011  (keystream)
```

As above we apply the BM algorithm interactively and get an LFSR of length 12 with feedback polynomial $T^{12}+T^{10}+T^9+T^8+T^6+T^5+T^3+T+1$, hence taps $1, 3, 5, 6, 8, 9, 10, 12$. Setting up the LFSR and predicting 48 keystream bits:

```
sage: coeff = [1,0,1,0,1,1,0,1,1,1,0,1]
sage: reg = LFSR(coeff)
sage: start = [1,0,1,1,0,0,0,1,0,0,1,1]
sage: reg.setState(start)
sage: testkey = reg.nextBits(48); testkey
[1,1,0,0,1,0,0,0,1,1,0,1,0,1,1,0,0,0,1,1,0,0,1,1,
 0,1,0,1,0,0,0,0,1,0,0,0,1,0,1,1,0,1,0,1,0,0,0,1]
```

we again get 48 bits of experimental plaintext, as bytes in decimal notation: `[84, 114, 101, 246, 214, 255]`. The translation to ISO 8859-1 yields the next flop: "TreöÖÿ".

As next step we use four letters of known plaintext `Tref` (as in Section 2.3) and derive 32 tentative keystream bits:

```
         10011100 10100100 01010110 10100110  (ciphertext)
Tref =   01010100 01110010 01100101 01100110  (assumed plaintext)
         -------- -------- -------- --------
         11001000 11010110 00110011 11000000  (keystream)
```

The BM algorithm yields an LFSR of length 16 with feedback polynomial $T^{16} + T^5 + T^3 + T^2 + 1$, hence taps $2, 3, 5, 16$. It predicts 48 keystream bits:

```
sage: coeff = [0,1,1,0,1,0,0,0,0,0,0,0,0,0,0,1]
sage: reg = LFSR(coeff)
sage: start = [0,1,1,0,1,0,1,1,0,0,0,1,0,0,1,1]
sage: reg.setState(start)
sage: testkey = reg.nextBits(48); testkey
[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]
```

and the experimental plaintext `[84, 114, 101, 102, 102, 32]` that looks promising: "Treff␣" (where ␣ here represents simple space character).

Sure of victory we decipher the complete text:

```
sage: cstream = "10011100101...1111111101"
sage: fullcipher = str2bbl(cstream)
sage: start = [0,1,1,0,1,0,1,1,0,0,0,1,0,0,1,1]
sage: reg.setState(start)
sage: keystream = reg.nextBits(232)
sage: fullplain = xor(fullcipher,keystream)
sage: fulltext = []
sage: for i in range(232/8):
    block = fullplain[8*i:8*i+8]
    nr = bbl2int(block)
    fulltext.append(nr)
sage: fulltext
[84,114,101,102,102,32,109,111,114,103,101,110,32,56,32,85,104,114,
 32,66,97,104,110,104,111,102,32,77,90]
 T  r  e  f  f  _  m  o  r  g  e  n  _  8  _  U  h  r
 _  B  a  h  n  h  o  f  _  M  Z
```

"Meeting tomorrow at 8 p. m. train station Mainz".

Remark

The success of this cryptanalytic approach crucially depends on the LFSR scenario, or in other words on a linearity profile like that in Figure 3.3 for the keystream. If the keystream comes from another kind of source we expect a linearity profile as in Figure 3.4 and shall not be able to make a stable prediction before the plaintext is exhausted.

We could also try nonlinear FSRs in an analoguous way as in Appendix B. Unfortunately most trials—even if the recursive profile stabilizes—will find a trivial FSR that allows no prediction beyond the end of the already known partial key sequence, see Appendix B. Then the approach "construct – predict – adjust" cannot work better than by guessing more keystream bits in a purely random way.

3.5 The Distribution of Linear Complexity

The distribution of the linear complexities of bit sequences of a fixed length can be determined exactly.

A given sequence $u = (u_0, \ldots, u_{N-1}) \in \mathbb{F}_2^N$ has two possible extensions $\tilde{u} = (u_0, \ldots, u_N) \in \mathbb{F}_2^{N+1}$ by 1 bit. The relation between $\lambda(\tilde{u})$ and $\lambda(u)$ is given by the Massey recursion: Let

$$\delta = \begin{cases} 0 & \text{if the prediction is correct,} \\ 1 & \text{otherwise.} \end{cases}$$

Here "prediction" refers to the next outpit bit from the LFSR we constructed for u. Then

$$\lambda(\tilde{u}) = \begin{cases} \lambda(u) & \text{if } \delta = 0, \\ \lambda(u) & \text{if } \delta = 1 \text{ and } \lambda(u) > \frac{N}{2}, \\ N + 1 - \lambda(u) & \text{if } \delta = 1 \text{ and } \lambda(u) \leq \frac{N}{2}. \end{cases}$$

In the middle case we need a new LFSR, but of the same length.

From these relations we derive a formula for the number $\mu_N(l)$ of all sequences of length N that have a given linear complexity l. To this end let

$$\begin{aligned} M_N(l) &:= \{u \in \mathbb{F}_2^N \mid \lambda(u) = l\} \quad \text{for } N \geq 1 \text{ and } l \in \mathbb{N}, \\ \mu_N(l) &:= \#M_N(l). \end{aligned}$$

The following three statements are immediately clear:

- $0 \leq \mu_N(l) \leq 2^N$,
- $\mu_N(l) = 0$ for $l > N$,
- $\sum_{l=0}^{N} \mu_N(l) = 2^N$.

From these we find explicit rules for the recursion from $\mu_{N+1}(l)$ to $\mu_N(l)$:

Case 1, $0 \leq l \leq \frac{N}{2}$. Every $u \in \mathbb{F}_2^N$ may be continued in two different ways: $u_N = 0$ or 1. Exactly one of them matches the prediction and leads to $\tilde{u} \in M_{N+1}(l)$. The other one leads to $\tilde{u} \in M_{N+1}(N+1-l)$. Since there are no other contributions to $M_{N+1}(l)$ we conclude $\mu_{N+1}(l) = \mu_N(l)$.

Case 2, $l = \frac{N+1}{2}$ (may occur only for odd N). The correctly predicted u_N leads to $\tilde{u} \in M_{N+1}(l)$, however the same is true for the mistakenly predicted one because of the Massey recursion. Hence $\mu_{N+1}(l) = 2 \cdot \mu_N(l)$.

Case 3, $l \geq \frac{N}{2} + 1$. Both possible continuations lead to $\tilde{u} \in M_{N+1}(l)$. Additionally we have one element from each of of the wrong predictions of all $u \in M_{N+1-l}(l)$ from case 1. Hence $\mu_{N+1}(l) = 2 \cdot \mu_N(l) + \mu_{N+1-l}(l)$.

The following lemma summarizes these considerations:

Lemma 14 *The frequency $\mu_N(l)$ of bit sequences of length N and linear complexity l complies with the recursion*

$$\mu_{N+1}(l) = \begin{cases} \mu_N(l) & \text{if } 0 \le l \le \frac{N}{2}, \\ 2 \cdot \mu_N(l) & \text{if } l = \frac{N+1}{2}, \\ 2 \cdot \mu_N(l) + \mu_{N+1-l}(l) & \text{if } l \ge \frac{N}{2} + 1. \end{cases}$$

From this recursion we get an explicit formula:

Proposition 11 [Rueppel] *The frequency $\mu_N(l)$ of bit sequences of length N and linear complexity l is given by*

$$\mu_N(l) = \begin{cases} 1 & \text{if } l = 0, \\ 2^{2l-1} & \text{if } 1 \le l \le \frac{N}{2}, \\ 2^{2(N-l)} & \text{if } \frac{N+1}{2} \le l \le N, \\ 0 & \text{if } l > N. \end{cases}$$

Proof. For $n = 1$ we have $M_1(0) = \{(0)\}$, $M_1(1) = \{(1)\}$, hence $\mu_1(0) = \mu_1(1) = 1$.

Now we proceed by induction from N to $N+1$. The case $l = 0$ is trivial since $M_{N+1}(0) = \{(0, \ldots, 0)\}$, $\mu_{N+1}(0) = 1$. As before we distinguish three cases:

Case 1, $1 \le l \le \frac{N}{2}$. A forteriori $1 \le l \le \frac{N+1}{2}$, and

$$\mu_{N+1}(l) = \mu_N(l) = 2^{2l-1}.$$

Case 2, $l = \frac{N+1}{2}$ (N odd). Here $\mu_N(l) = 2^{2(N-l)}$, and the exponent is $2N - 2l = 2N - N - 1 = N - 1 = 2l - 2$, hence

$$\mu_{N+1}(l) = 2 \cdot 2^{2(N-l)} = 2^{2l-2+1} = 2^{2l-1}.$$

Case 3, $l \ge \frac{N}{2} + 1$. Again $\mu_N(l) = 2^{2(N-l)}$. For $l' = N + 1 - l$ we have $l' \le N + 1 - \frac{N}{2} - 1 = \frac{N}{2}$, hence $\mu_N(l') = 2^{2l'-1}$, and

$$\begin{aligned} \mu_{N+1}(l) &= 2\mu_N(l) + \mu_N(l') = 2^{2N-2l+1} + 2^{2N-2l+1} \\ &= 2^{2N-2l+2} = 2^{2(N+1-l)}. \end{aligned}$$

This completes the proof. ◇

Table 3.1 gives an impression of the distribution.

	1	2	3	4	5	6	7	8	9	10	$N \rightarrow$
0	1	1	1	1	1	1	1	1	1	1	
1	1	2	2	2	2	2	2	2	2	2	
2		1	4	8	8	8	8	8	8	8	
3			1	4	16	32	32	32	32	32	
4				1	4	16	64	128	128	128	
5					1	4	16	64	256	512	
6						1	4	16	64	256	
7							1	4	16	64	
8								1	4	16	
9									1	4	
10										1	
l											
$\downarrow$											

Table 3.1: The distribution of linear complexity

Observations

- Row l is constant from $N = 2l$ on (red numbers), the diagonals, from $N = 2l - 1$ on (blue numbers).
- Each column N, from row $l = 1$ to row $l = N$, contains the powers 2^k, $k = 0, \ldots, N - 1$, each one exactly once—first the odd powers in ascending order (red), followed by the even powers (blue) in descending order.
- For every length N there is exactly one sequence of linear complexity 0 and N each: From Section 3.1 we know that these are the sequences $(0, \ldots, 0, 0)$ and $(0, \ldots, 0, 1)$.

Figure 3.5 shows the histogram of this distribution for $N = 10$, Figure 3.6 for $N = 100$. The second histogram looks strikingly small. We'll clarify this phenomen in the following Section 3.6.

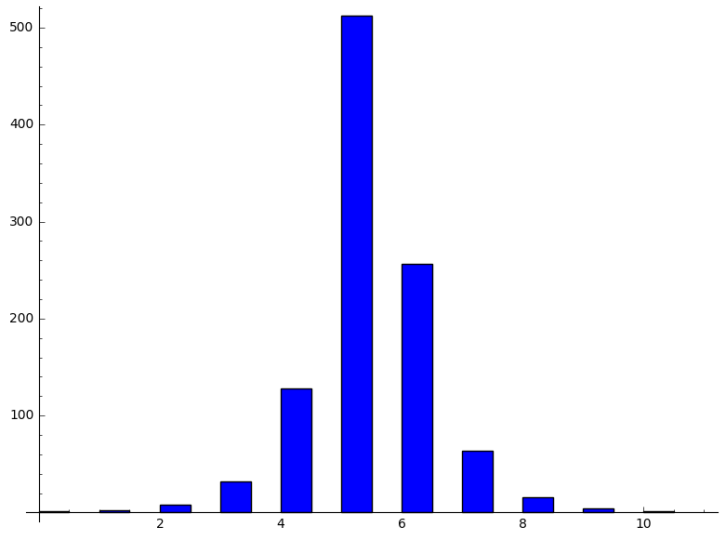


Figure 3.5: The distribution of linear complexity for bitsequences of length $N = 10$

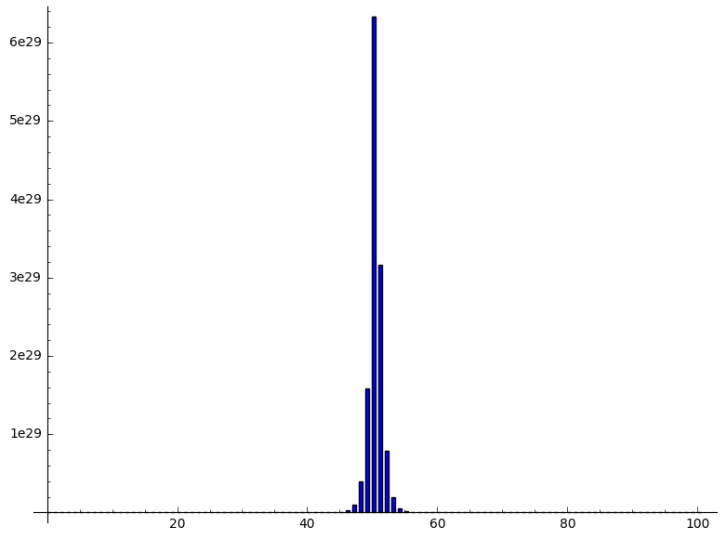


Figure 3.6: The distribution of linear complexity for bitsequences of length $N = 100$

3.6 The Mean Value of the Linear Complexity

From the exact distribution of the linear complexity we also can exactly determine the mean value and the variance (for fixed length N):

Theorem 3 (Rueppel) *Explicit formulas for the mean value*

$$E_N = \frac{1}{2^N} \cdot \sum_{u \in \mathbb{F}_2^N} \lambda(u)$$

and the variance V_N of the linear complexity of all bit sequences of length N are:

$$\begin{aligned} E_N &= \frac{N}{2} + \frac{2}{9} + \frac{\varepsilon}{18} - \frac{N}{3 \cdot 2^N} - \frac{2}{9 \cdot 2^N} \approx \frac{N}{2}, \\ V_N &= \frac{86}{81} - \frac{14 - \varepsilon}{27} \cdot \frac{N}{2^N} - \frac{82 - 2\varepsilon}{81} \cdot \frac{1}{2^N} - \frac{9N^2 + 12N + 4}{81} \cdot \frac{1}{2^{2N}} \approx \frac{86}{81} \end{aligned}$$

where $\varepsilon = 0$ for N even, $\varepsilon = 1$ for N odd (ε is the parity of N).

Remarkably the variance is almost independent of N. Thus almost all linear complexities vary around the mean value in a small strip only that is (almost) independent of N and becomes *relatively* more narrow with increasing N as illustrated by Figures 3.5 and 3.6.

For the proof we have to make a small detour. We'll encounter sums that have a nice expression using a well-known trick from calculus.

Lemma 15 *For the derivatives of the function*

$$f : \mathbb{R} - \{1\} \longrightarrow \mathbb{R}, \quad f(x) = \frac{x^{r+1} - x}{x - 1},$$

we have the formulas:

$$f'(x) = \frac{1}{(x-1)^2} \cdot \left[r x^{r+1} - (r+1)x^r + 1 \right],$$

$$f''(x) = \frac{1}{(x-1)^3} \cdot \left[(r^2 - r)x^{r+1} - 2(r^2 - 1)x^r + (r^2 + r)x^{r-1} - 2 \right],$$

$$x^2 f''(x) + x f'(x) = \frac{x}{(x-1)^3} \cdot \left[r^2 x^{r+2} - (2r^2 + 2r - 1)x^{r+1} + (r+1)^2 x^r - x - 1 \right].$$

Proof. By direct calculation. ◇

Using these formulas for f we explicitly calculate some sums:

Corollary 1 *For all $x \in \mathbb{R}$, $x \neq 1$, we have:*

$$\begin{aligned}
\sum_{i=1}^{r} x^i &= \frac{1}{x-1} \cdot \left[x^{r+1} - x\right], \\
\sum_{i=1}^{r} i x^i &= \frac{x}{(x-1)^2} \cdot \left[r x^{r+1} - (r+1)x^r + 1\right], \\
\sum_{i=1}^{r} i^2 x^i &= \frac{x}{(x-1)^3} \cdot \left[r^2 x^{r+2} - (2r^2+2r-1)x^{r+1} + (r+1)^2 x^r - x - 1\right].
\end{aligned}$$

Proof. From the sum formula for the geometric series we conclude

$$\sum_{i=1}^{r} x^i = x \cdot \sum_{i=0}^{r-1} x^i = x \cdot \frac{x^r - 1}{x-1} = f(x),$$

$$\sum_{i=1}^{r} i x^i = x \cdot \sum_{i=1}^{r} i x^{i-1} = x \cdot f'(x),$$

$$\sum_{i=1}^{r} i^2 x^i = \sum_{i=1}^{r} i(i-1)x^i + \sum_{i=1}^{r} i x^i = x^2 \cdot f''(x) + x \cdot f'(x).$$

Therefore the claimed formulas follow from Lemma 15. ◇

Corollary 2

$$\begin{aligned}
\sum_{i=1}^{r} i\, 2^{2i-1} &= \frac{3r-1}{9} \cdot 2^{2r+1} + \frac{2}{9}, \\
\sum_{i=1}^{r} i^2\, 2^{2i-1} &= \frac{3r^2-2r}{9} \cdot 2^{2r+1} + \frac{5}{27} \cdot 2^{2r+1} - \frac{10}{27}.
\end{aligned}$$

Proof.

$$\sum_{i=1}^{r} i 2^{2i-1} = \frac{1}{2} \cdot \sum_{i=1}^{r} i 4^i = \frac{1}{2} \cdot \frac{4}{9} \cdot \left[r 4^{r+1} - (r+1)4^r + 1\right] = \frac{2}{9} \cdot \left[3r 4^r - 4^r + 1\right],$$

$$\begin{aligned}
\sum_{i=1}^{r} i^2 2^{2i-1} = \frac{1}{2} \cdot \sum_{i=1}^{r} i^2 4^i &= \frac{1}{2} \cdot \frac{4}{27} \cdot \left[r^2 4^{r+2} - (2r^2+2r-1)4^{r+1} + (r+1)^2 4^r - 5\right] \\
&= \frac{2}{27} \cdot \left[(9r^2 - 6r + 5) \cdot 4^r - 5\right].
\end{aligned}$$

◇

Now the mean value of the linear complexity is

$$E_N = \frac{1}{2^N} \cdot \sum_{u \in \mathbb{F}_2^N} \lambda(u) = \frac{1}{2^N} \cdot \sum_{l=0}^{N} l \cdot \mu_N(l),$$

$$2^N E_N = \underbrace{\sum_{l=1}^{\lfloor \frac{N}{2} \rfloor} l \cdot 2^{2l-1}}_{S_1} + \underbrace{\sum_{l=\lceil \frac{N+1}{2} \rceil}^{N} l \cdot 2^{2(N-l)}}_{S_2}.$$

First let N be even. Then

$$\begin{aligned}
S_1 &= \sum_{l=1}^{\frac{N}{2}} l \cdot 2^{2l-1} = \frac{3N-2}{18} \cdot 2^{N+1} + \frac{2}{9} = \frac{N}{3} \cdot 2^N - \frac{2}{9} \cdot 2^N + \frac{2}{9}, \\
S_2 &= \sum_{l=\frac{N}{2}+1}^{N} l \cdot 4^{N-l} \overset{k=N-l}{=} \sum_{k=0}^{\frac{N}{2}-1} (N-k) \cdot 4^k = N \cdot \sum_{k=0}^{\frac{N}{2}-1} 4^k - \sum_{k=0}^{\frac{N}{2}-1} k \cdot 4^k \\
&= N \cdot \frac{4^{N/2}-1}{3} - \frac{4}{9} \cdot \left[(\frac{N}{2} - 1) \cdot 4^{\frac{N}{2}} - \frac{N}{2} \cdot 4^{\frac{N}{2}-1} + 1 \right] \\
&= \frac{N}{3} \cdot 2^N - \frac{N}{3} - \frac{4}{9} \cdot \left[\frac{N}{2} \cdot 2^N - 2^N - \frac{N}{8} \cdot 2^N + 1 \right] \\
&= \left(\frac{N}{6} + \frac{4}{9} \right) \cdot 2^N - \frac{N}{3} - \frac{4}{9}.
\end{aligned}$$

Taken together this yields

$$2^N E_N = \frac{N}{2} \cdot 2^N + \frac{2}{9} \cdot 2^N - \frac{N}{3} - \frac{2}{9},$$

proving the first formula of Theorem 3 for N even.

For odd N we have

$$\begin{aligned}
S_1 &= \sum_{l=1}^{\frac{N-1}{2}} l \cdot 2^{2l-1} = \frac{3(N-1)-2}{18} \cdot 2^N + \frac{2}{9} = \frac{3N-5}{18} \cdot 2^N + \frac{2}{9} \\
&= \frac{N}{6} \cdot 2^N - \frac{5}{18} \cdot 2^N + \frac{2}{9},
\end{aligned}$$

$$\begin{aligned}
S_2 &= \sum_{l=\frac{N+1}{2}}^{N} l \cdot 4^{N-l} \overset{k=N-l}{=} \sum_{k=0}^{\frac{N-1}{2}} (N-k) \cdot 4^k = N \cdot \sum_{k=0}^{\frac{N-1}{2}} 4^k - \sum_{k=0}^{\frac{N-1}{2}} k \cdot 4^k \\
&= N \cdot \frac{4^{(N+1)/2} - 1}{3} - \frac{4}{9} \cdot \left[\frac{N-1}{2} \cdot 4^{\frac{N+1}{2}} - \frac{N+1}{2} \cdot 4^{\frac{N-1}{2}} + 1 \right] \\
&= \frac{N}{3} \cdot 2^{N+1} - \frac{N}{3} - \frac{4}{9} \cdot \left[\frac{N-1}{2} \cdot 2^{N+1} - \frac{N+1}{2} \cdot 2^{N-1} + 1 \right] \\
&= \frac{2N}{3} \cdot 2^N - \frac{N}{3} - \frac{4N}{9} \cdot 2^N + \frac{4}{9} \cdot 2^N + \frac{N}{9} \cdot 2^N + \frac{1}{9} \cdot 2^N - \frac{4}{9} \\
&= \left(\frac{N}{3} + \frac{5}{9} \right) \cdot 2^N - \frac{N}{3} - \frac{4}{9},
\end{aligned}$$

$$2^N E_N = \frac{N}{2} \cdot 2^N + \frac{5}{18} \cdot 2^N - \frac{N}{3} - \frac{2}{9},$$

proving the first formula of Theorem 3 also for odd N.

Now let's calculate the variance V_N. We start with

$$\begin{aligned}
V_N + 2^N E_N^2 &= \frac{1}{2^N} \cdot \sum_{u \in \mathbb{F}_2^N} \lambda(u)^2 = \frac{1}{2^N} \cdot \sum_{l=0}^{N} l^2 \cdot \mu_N(l), \\
&= \underbrace{\sum_{l=1}^{\lfloor \frac{N}{2} \rfloor} l^2 \cdot 2^{2l-1}}_{S_3} + \underbrace{\sum_{l=\lceil \frac{N+1}{2} \rceil}^{N} l^2 \cdot 4^{N-l}}_{S_4}.
\end{aligned}$$

Again we first treat the case of even N. Then the first sum evaluates as

$$\begin{aligned}
S_3 &= \sum_{l=1}^{\frac{N}{2}} l^2 \cdot 2^{2l-1} = \frac{3 \cdot \frac{N^2}{4} - 2 \cdot \frac{N}{2}}{9} \cdot 2^{N+1} + \frac{5}{27} \cdot 2^{N+1} - \frac{10}{27} \\
&= \frac{N^2}{6} \cdot 2^N - \frac{2N}{9} \cdot 2^N + \frac{10}{27} \cdot 2^N - \frac{10}{27}.
\end{aligned}$$

We decompose the second sum:

$$\begin{aligned}
S_4 &= \sum_{l=\frac{N}{2}+1}^{N} l^2 \cdot 4^{N-l} \overset{k=N-l}{=} \sum_{k=0}^{\frac{N}{2}-1} (N-k)^2 \cdot 4^k \\
&= \underbrace{N^2 \cdot \sum_{k=0}^{\frac{N}{2}-1} 4^k}_{S_{4a}} \underbrace{-2\,N \cdot \sum_{k=0}^{\frac{N}{2}-1} k \cdot 4^k}_{S_{4b}} + \underbrace{\sum_{k=0}^{\frac{N}{2}-1} k^2 \cdot 4^k}_{S_{4c}}
\end{aligned}$$

and separately evaluate the summands:

$$\begin{aligned}
S_{4a} &= N^2 \cdot \frac{4^{\frac{N}{2}}-1}{3} = \frac{N^2}{3}\cdot 2^N - \frac{N^2}{3}\,,\\
S_{4b} &= N\cdot\frac{4}{9}\cdot\left[(\frac{N}{2}-1)\cdot 4^{\frac{N}{2}} - \frac{N}{2}\cdot 4^{\frac{N}{2}-1}+1\right]\\
&= \frac{4N}{9}\cdot\left[\frac{N}{2}\cdot 2^N - 2^N - \frac{N}{8}\cdot 2^N + 1\right] = \frac{N^2}{6}\cdot 2^N - \frac{4N}{9}\cdot 2^N + \frac{4N}{9}\,,\\
S_{4c} &= \frac{4}{27}\cdot\left[(\frac{N}{2}-1)^2\cdot 4^{\frac{N}{2}+1} - \left(2\cdot(\frac{N}{2}-1)^2 + 2\cdot(\frac{N}{2}-1) - 1\right)\cdot 4^{\frac{N}{2}}\right.\\
&\qquad \left. +(\frac{N}{2})^2\cdot 4^{\frac{N}{2}-1} - 5\right]\\
&= \frac{4}{27}\cdot\left[2\cdot(\frac{N^2}{4}-N+1)\cdot 2^N - N\cdot 2^N + 2\cdot 2^N + 2^N + \frac{N^2}{16}\cdot 2^N - 5\right]\\
&= \frac{1}{12}\cdot N^2\cdot 2^N - \frac{4}{9}\cdot N\cdot 2^N + \frac{20}{27}\cdot 2^N - \frac{20}{27}\,.
\end{aligned}$$

We have to subtract

$$\begin{aligned}
2^N\cdot E_N^2 &= \left[\frac{N}{2}+\frac{2}{9}-\frac{N}{3\cdot 2^N}-\frac{2}{9\cdot 2^N}\right]^2\cdot 2^N\\
&= \frac{N^2}{4}\cdot 2^N + \frac{2N}{9}\cdot 2^N + \frac{4}{81}\cdot 2^N - \frac{N^2}{3} - \frac{10N}{27} - \frac{8}{81}\\
&\quad + \frac{N^2}{9\cdot 2^N} + \frac{4N}{27\cdot 2^N} + \frac{4}{81\cdot 2^N}\,.
\end{aligned}$$

All this fragments together yield

$$2^N\cdot V_N = \frac{86}{81}\cdot 2^N - \frac{14N}{27} - \frac{82}{81} - \frac{N^2}{9\cdot 2^N} - \frac{4N}{27\cdot 2^N} - \frac{4}{81\cdot 2^N},$$

proving the second formula of Theorem 3 for even N.

The corresponding calculation for odd N is:

$$\begin{aligned}
S_3 &= \sum_{l=1}^{\frac{N-1}{2}} l^2\cdot 2^{2l-1} = \frac{N^2}{12}\cdot 2^N - \frac{5N}{18}\cdot 2^N + \frac{41}{108}\cdot 2^N - \frac{10}{27}\,,\\
S_{4a} &= N^2\cdot\sum_{k=0}^{\frac{N-1}{2}} 4^k = \frac{2N^2}{3}\cdot 2^N - \frac{N^2}{3}\,,\\
S_{4b} &= N\cdot\sum_{k=0}^{\frac{N-1}{2}} k\cdot 4^k = \frac{N^2}{3}\cdot 2^N - \frac{5N}{9}\cdot 2^N + \frac{4N}{9}\,,\\
S_{4c} &= \sum_{k=0}^{\frac{N-1}{2}} k^2\cdot 4^k = \frac{N^2}{6}\cdot 2^N - \frac{5N}{9}\cdot 2^N + \frac{41}{54}\cdot 2^N - \frac{20}{27}\,,
\end{aligned}$$

$$\begin{aligned} 2^N \cdot E_N^2 &= \left[\frac{N}{2} + \frac{5}{18} - \frac{N}{3 \cdot 2^N} - \frac{2}{9 \cdot 2^N}\right]^2 \cdot 2^N \\ &= \frac{N^2}{4} \cdot 2^N + \frac{5N}{18} \cdot 2^N + \frac{25}{324} \cdot 2^N - \frac{N^2}{3} - \frac{11N}{27} - \frac{10}{81} \\ &\quad + \frac{N^2}{9 \cdot 2^N} + \frac{4N}{27 \cdot 2^N} + \frac{4}{81 \cdot 2^N}\,. \end{aligned}$$

Putting the fragments together we get

$$\begin{aligned} 2^N \cdot V_N &= S_3 + S_{4a} - 2 \cdot S_{4b} + S_{4c} - 2^N \cdot E_N^2 \\ &= \frac{86}{81} \cdot 2^N - \frac{13N}{27} - \frac{80}{81} - \frac{9N^2 + 12N + 4}{81 \cdot 2^N}\,. \end{aligned}$$

This completes the proof of Theorem 3.

3.7 Linear Complexity and TURING Complexity

A **universal** TURING **machine** is able to simulate every other TURING machine by a suitable program. Let **M** be one, and let $u \in \mathbb{F}_2^n$ be a bit sequence of length n. Then the TURING-KOLMOGOROV-CHAITIN **(TKC) complexity** $\chi(u)$ is the length of the shortest program of **M** that outputs u. There is always one such program of length n: Simply take u as input sequence and output it unchanged. (Informally: Move the input tape forward by n steps and stop.)

Remark The function $\chi\colon \mathbb{F}_2^* \longrightarrow \mathbb{N}$ itself is not computable. This means there is no TURING machine that computes χ. Thus the TKC complexity is of low practical value as a measure of complexity. However in the recent years it gained some momentum in a more precise form by the work of VITANYI and others, see for example:

> Ming LI, Paul VITANYI: *An Introduction to Kolmogorov Complexity and Its Applications.* Springer, New York 1993, 1997.

A central result of the theory is:

$$\frac{1}{2^n} \cdot \#\{u \in \mathbb{F}_2^n \mid \chi(u) > n \cdot (1-\varepsilon)\} > 1 - \frac{1}{2^{n\varepsilon-1}}.$$

This result says that almost all sequences have a TKC complexity near the maximum value, there is no significantly shorter description of a sequence than to simply write it down. A common interpretation of this result is: "Almost all sequences are random." This corresponds quite well with the intuitive idea of randomness. Nobody would consider a sequence with a short description such as "alternate one million times between 0 and 1" as random.

> Thomas BETH, Zong-Duo DAI: On the complexity of pseudo-random sequences – or: If you can describe a sequence it can't be random. EUROCRYPT 89, 533–543.

This paper contains some small errors that are corrected in [9].

Also "linear complexity" λ measures complexity, using a quite special machine model: the LFSR. On first sight it suffers from severe deficits. The sequence "999999 times 0, then a single 1" has a very low TKC complexity—corresponding to a very low intuitive randomness—, but the linear complexity is 1 million.

Of course we could also try to use nonlinear FSRs for measuring complexity, see for instance the papers:

- Agnes Hui CHAN, Richard A. GAMES: On the quadratic span of periodic sequences. CRYPTO 89, 82–89.

- Cees J. A. Jansen, Dick E. Boekee: The shortest feedback shift register that can generate a given sequence. Crypto 89, 90–96.

and Appendix B. However, as we saw, *a short description by a nonlinear FSR also implies a small linear complexity.*

In any case linear complexity has the advantage of easy explicite computability, and "in general" it characterizes the randomness of a bit sequence very well. This vague statement admits a surprisingly precise wording (stated here without proof). To make a fair comparision note that the description of a sequence by an LFSR needs $2 \times \lambda$ bits: the taps of the register and the starting value. Therefore we should compare χ and $2 \cdot \lambda$:

Proposition 12 (Beth/Dai)

$$\begin{aligned}
\frac{1}{2^n} \cdot \#\{u \in \mathbb{F}_2^n \mid (1-\varepsilon)2\lambda(u) \leq \chi(u)\} &\geq 1 - \frac{8}{3 \cdot 2^{\frac{n\varepsilon}{2-\varepsilon}}}, \\
\frac{1}{2^n} \cdot \#\{u \in \mathbb{F}_2^n \mid (1-\varepsilon)\chi(u) \leq 2\lambda(u)\} &\geq 1 - \frac{1}{3} \cdot \frac{1}{2^{n\varepsilon-(1-\varepsilon)(1+\log n)+1}} - \frac{1}{3} \cdot \frac{1}{2^n}.
\end{aligned}$$

We interpret this as: "For almost all bit sequences the linear complexity and the TKC complexity coincide with only a negligeable discrepancy (up to the obvious factor 2)."

This result confirms that linear complexity—despite its simplicity—is a useful measure of complexity, and that in general bit sequences of high linear complexity have no short description in other machine models. Thus they are cryptographically useful. Every efficient prediction method—in the sense of cryptanalysis of bitstream ciphers—would provide a short description in the sense of TKC complexity. And conversely: If a sequence has a short description, then we even can generate it by a short LFSR. Thus we may summarize: *In general a bit sequence of high linear complexity is not efficiently predictable.*

Note that these results

- are "asymptotic" in character. For the "bounded" world we live in they only yield qualitative statements—a standard phenomen for results on cryptographic security.

- concern probabilities only. There might be $2^r \ll 2^n$ sequences of small TKC complexity that however have high linear complexity—*relatively* very few, but *absolutely* quite a lot! In Chapter 4 we'll construct such sequences, dependent on secret parameters, and show (up to one of the usual hardness assumptions for mathematical problems) that they don't allow an efficient prediction algorithm, in particular not by a "short" LFSR.

3.8 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 3.9 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:

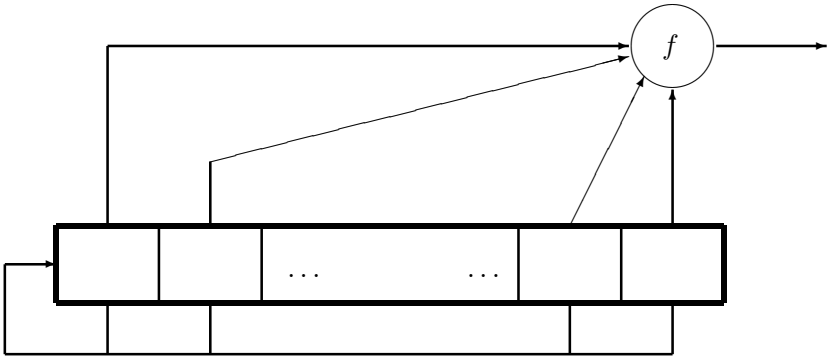


Figure 3.7: Nonlinear ouput filter for an LFSR

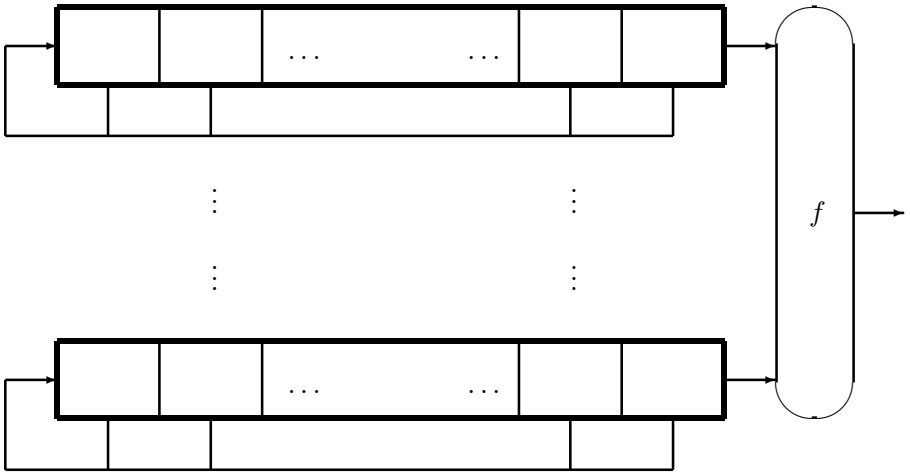


Figure 3.8: Nonlinear combiner

- **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)

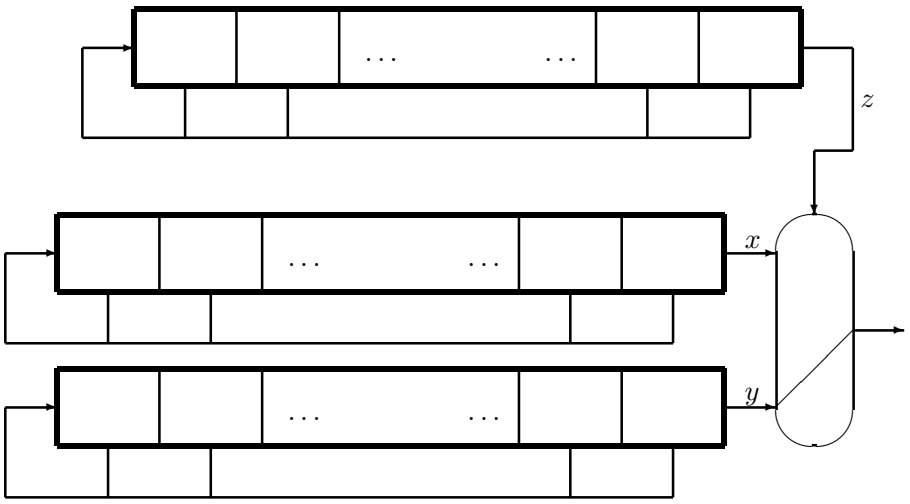


Figure 3.9: Geffe generator

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]
```

3.9 Correlation Attacks—the Achilles Heels of Combiners

Let $f\colon \mathbb{F}_2^n \longrightarrow \mathbb{F}_2$ be the combining function of a nonlinear combiner. The number

$$K_f := \#\{x = (x_1, \ldots, x_n) \in \mathbb{F}_2^n \mid f(x) = x_1\}$$

counts the coincidences of the value of the function with its first argument. If it is $> 2^{n-1}$, then the probability of a coincidence,

$$p = \frac{1}{2^n} \cdot K_f > \frac{1}{2},$$

is above average, and the combined output sequence "correlates" with the output of the first LFSR more then expected by random. If $p < \frac{1}{2}$, then the correlation deviates from the expected value in the other direction.

The cryptanalyst can exploit this effect in an attack with known plaintext. We suppose that she knows the "hardware", that is the taps of the registers, and also the combining function f. She seeks the initial states of all the LFSRs. We assume she knows the bits $k_0, \ldots, k_{r-1}$ of the key stream. For each of the 2^{l_1} initial states of the first LFSR she generates the sequence $u_0, \ldots, u_{r-1}$, and counts the coincidences. The expected values are

$$\frac{1}{r} \cdot \#\{i \mid u_i = k_i\} \approx \begin{cases} p & \text{for the correct initial state of LFSR 1,} \\ \frac{1}{2} & \text{otherwise.} \end{cases}$$

If r is large enough, she can determine the true initial state of LFSR 1 (with high probability) for a cost of $\sim 2^{l_1}$. She continues with the other registers, and finally identifies the complete key with a cost of $\sim 2^{l_1} + \cdots + 2^{l_n}$. Note that the cost is exponential, but significantly lower than the cost $\sim 2^{l_1} \cdots 2^{l_n}$ of the naive exhaustion of the key space.

In the language of linear cryptanalysis from Part II she made use of the linear relation

$$f(x_1, \ldots, x_n) \overset{p}{\approx} x_1$$

for f. Clearly she could use any linear relation as well to reduce the complexity of key search. (A more in-depth analysis of the situation leads to the notion of correlation immunity that is related with the linear potential.)

Correlations from the Geffe generator

From the truth table 3.2 we get the correlations produced by the Geffe generator. Thus the probabilities of coincidences are

$$p = \begin{cases} \frac{3}{4} & \text{for register 1 } (x), \\ \frac{3}{4} & \text{for register 2 } (y), \\ \frac{1}{2} & \text{for register 3 } (z = \text{control bit}). \end{cases}$$

x	0	0	0	0	1	1	1	1
y	0	1	0	1	0	1	0	1
z	0	0	1	1	0	0	1	1
$f(x,y,z)$	0	0	0	1	1	1	0	1

Table 3.2: Truth table of the GEFFE function

linear form	0	z	y	$y+z$	x	$x+z$	$x+y$	$x+y+z$
representation	000	001	010	011	100	101	110	111
potential λ	0	0	1/4	1/4	1/4	1/4	0	0
probability p	1/2	1/2	3/4	1/4	3/4	3/4	1/2	1/2

Table 3.3: Coincidence probabilities of the GEFFE function

A correlation attack easily detects the initial states of registers 1 and 2—the battery registers—given only a short piece of an output sequence. Afterwards exhaustion finds the initial state of register 3, the control register.

We exploit this weakness of the GEFFE generator for an attack in Sage sample 3.6 that continues Sage sample 3.2. Since we defined the linear profile for objects of the class `BoolMap` only, we first of all have to interpret the function `geff` as a Boolean map, that is a one-element list of Boolean functions. Then the linear profile is represented by a matrix of 2 columns and 8 rows. The first column `[64, 0, 0, 0, 0, 0, 0, 0]` shows the coincidences with the linear form 0 in the range. So it contains no useful information, except the denominator 64 that applies to all entries. The second row `[0, 0, 16, 16, 16, 16, 0, 0]` yields the list of coincidence probabilities p (after dividing it by 64) in Table 3.3, using the formula

$$p = \frac{1}{2} \cdot (\pm\sqrt{\lambda} + 1).$$

If $\lambda = 0$, then $p = 1/2$. If $\lambda = 1/4$, then $p = 1/4$ or $3/4$. For deciding between these two values for p we use Table 3.2.

Sage Example 3.6 Linear profile of the Geffe function

```
sage: g = BoolMap([geff])
sage: linProf = g.linProf(); linProf
[[64,0], [0,0], [0,16], [0,16], [0,16], [0,16], [0,0], [0,0]]
```

In Sage sample 3.7 we apply this finding to the 100 element sequence from Sage sample 3.5. The function `coinc` from the Sage module `Bitblock.sage` in Appendix E.1 of Part II counts the coincidences. For the first register we

find 73 coincidences, for the second one 76, for the third one only 41. This confirms the values 75, 75, 50 predicted by our theory.

Sage Example 3.7 Coincidences for the Geffe generator

```
sage: coinc(outlist15,outlist)
73
sage: coinc(outlist16,outlist)
76
sage: coinc(outlist17,outlist)
41
```

Cryptanalysis of the Geffe Generator

These results promise an effortless analysis of our sample sequence. For an assessment of the success probability we consider a bitblock $b \in \mathbb{F}_2^r$ and first ask how large is the probability that a random bitblock $u \in \mathbb{F}_2^r$ coincides with b at exactly t positions. For an answer we have to look at the symmetric binomial distribution (where $p = \frac{1}{2}$ is the probability of coincidence at a single position): The probability of exactly t coincidences is

$$B_{r,\frac{1}{2}}(t) = \frac{\binom{r}{t}}{2^r}.$$

Hence the cumulated probability of up to T coincidences is

$$\sum_{t=0}^{T} B_{r,\frac{1}{2}}(t) = \frac{1}{2^r} \cdot \sum_{t=0}^{T} \binom{r}{t}.$$

If r is not too large, then we may explicitly calculate this value for a given bound T. If on the other hand r is not too small, then we approximate the value using the normal distribution. The mean value of the number of coincidences is $r/2$, the variance, $r/4$, and the standard deviation, $\sqrt{r}/2$.

In any case for $r = 100$ the probability of finding at most (say) 65 coincidences is 0.999, the probability of surpassing this number is 1 ‰. For the initial state of register 1 we have to try $2^{15} = 32786$ possibilities (generously including the zero state $0 \in \mathbb{F}_2^{15}$ into the count). So we expect about 33 oversteppings with at least 66 coincidences. One of these should occur for the true initial state of register 1 that we expect to produce about 75 coincidences. Maybe it even produces the maximum number of coincidences.

Sage sample 3.8 shows that this really happens. However the maximum number of coincidences, 73, occurs twice in the histogram. The first occurrence happens at index 13705, corresponding to the initial state 011010110001001, the correct solution. The second occurrence, at index

Sage Example 3.8 Analysis of the Geffe generator—register 1

```
sage: clist = []
sage: histogr = [0] * (nofBits + 1)
sage: for i in range(0,2**15):
....:     start = int2bbl(i,15)
....:     reg15.setState(start)
....:     testlist = reg15.nextBits(nofBits)
....:     c = coinc(outlist,testlist)
....:     histogr[c] += 1
....:     clist.append(c)
....:
sage: print(histogr)
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 4, 12, 12, 37, 78, 116, 216,
 329, 472, 722, 1003, 1369, 1746, 1976, 2266, 2472, 2531, 2600,
 2483, 2355, 2149, 1836, 1574, 1218, 928, 726, 521, 343, 228, 164,
 102, 60, 47, 36, 13, 8, 7, 4, 2, 1, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
sage: mm = max(clist)
sage: ix = clist.index(mm)
sage: block = int2bbl(ix,15)
sage: print("Maximum =", mm, "at index", ix, ", start value", block)
Maximum = 73 at index 13705 , start value\
 [0, 1, 1, 0, 1, 0, 1, 1, 0, 0, 0, 1, 0, 0, 1]
```

Sage Example 3.9 Analysis of the Geffe generator—continued

```
sage: ix = clist.index(mm,13706); ix
31115
sage: print(int2bbl(ix,15))
[1, 1, 1, 1, 0, 0, 1, 1, 0, 0, 0, 1, 0, 1, 1]
```

31115, see Sage sample 3.9, yields the false solution 111100110001011 that eventually leads to a contradiction.

Sage sample 3.10 shows the analogous analysis of register 2. Here the maximum of coincidences, 76, is unique, occurs at index 27411 corresponding to the initial state 0110101100010011, and provides the correct solution.

To complete the analysis we must yet determine the initial state of register 3, the control register. The obvious idea is to exhaust the 2^{17} different possibilities. There is a shortcut since we already know 51 of the first 100 bits of the control register: At a position where the values of registers 1 and

Sage Example 3.10 Analysis of the Geffe generator—register 2

```
sage: clist = []
sage: histogr = [0] * (nofBits + 1)
sage: for i in range(0,2**16):
....:     start = int2bbl(i,16)
....:     reg16.setState(start)
....:     testlist = reg16.nextBits(nofBits)
....:     c = coinc(outlist,testlist)
....:     histogr[c] += 1
....:     clist.append(c)
....:
sage: print(histogr)
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 1, 0, 2, 3, 4, 8, 17, 25, 51, 92, 171,
 309, 477, 750, 1014, 1423, 1977, 2578, 3174, 3721, 4452, 4821,
 5061, 5215, 5074, 4882, 4344, 3797, 3228, 2602, 1974, 1419,
 1054, 669, 434, 306, 174, 99, 62, 38, 19, 10, 3, 0, 1, 0, 0,
 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0]
sage: mm = max(clist)
sage: ix = clist.index(mm)
sage: block = int2bbl(ix,16)
sage: print("Maximum =", mm, "at index", ix, ", start value", block)
Maximum = 76 at index 27411 , start value\
 [0, 1, 1, 0, 1, 0, 1, 1, 0, 0, 0, 1, 0, 0, 1, 1]
```

2 differ, the control bit is necessarily 0 if the final output coincides with register 1, and 1 otherwise. Only at positions where registers 1 and 2 coincide the corresponding bit of register 3 is undetermined.

```
 register 1: 10010001101011011100001001101101000001110110110000
 register 2: 11001000110101100011001111000000001110111000111000
 register 3: -1-00--0-1101-110001---00-1-00-1--1101--110---0---
bitsequence: 11010001110001101101001001001100001100111010110000

        ... 00101101101111111001001001010101110001110011001011
        ... 00100011101111010010011110010110111100101110010001
        ... ----110-------1-1-11-0-100----01--01-1-001-1-00-1-
        ... 00100001101111010010001101010100110100110110001001
```

In particular we already know 11 of the 17 initial bits, and are left with only $2^6 = 64$ possibilities to try.

$u_{17} = u_{14} + u_0$	$0 = 1 + u_0$	$u_0 = 1$	
$u_{19} = u_{16} + u_2$	$1 = 0 + u_2$	$u_2 = 1$	
$u_{20} = u_{17} + u_3$	$u_{20} = 0 + 0$	$u_{20} = 0$	
$u_{22} = u_{19} + u_5$	$u_{22} = u_5 + 1$	$u_5 = u_{22} + 1$	
$u_{23} = u_{20} + u_6$	$0 = u_{20} + u_6$	$u_6 = u_{20}$	$u_6 = 0$
$u_{25} = u_{22} + u_8$	$u_{25} = u_{22} + u_8$	$u_8 = u_{22} + u_{25}$	$u_8 = u_{22}$
$u_{27} = u_{24} + u_{10}$	$u_{27} = 0 + 1$	$u_{27} = 1$	
$u_{28} = u_{25} + u_{11}$	$0 = u_{25} + 0$	$u_{25} = 0$	
$u_{30} = u_{27} + u_{13}$	$u_{30} = u_{27} + u_{13}$	$u_{13} = u_{27} + u_{30}$	$u_{13} = u_{30} + 1$
$u_{33} = u_{30} + u_{16}$	$u_{33} = u_{30} + 0$	$u_{30} = u_{33}$	$u_{30} = 1$
$u_{36} = u_{33} + u_{19}$	$0 = u_{33} + 1$	$u_{33} = 1$	
$u_{39} = u_{36} + u_{22}$	$u_{39} = 0 + u_{22}$	$u_{22} = u_{39}$	
$u_{42} = u_{39} + u_{25}$	$0 = u_{39} + u_{25}$	$u_{39} = u_{25}$	$u_{39} = 0$

Table 3.4: Determination of the control register's initial state

But even this may be further simplified, since the known and the unknown bits obey linear relations of the type $u_n = u_{n-3} + u_{n-17}$. The unknown bits of the initial state are u_0, u_2, u_5, u_6, u_8, u_{13}. The solution follows the columns of Table 3.4, that immediately give

$$u_0 = 1,\ u_2 = 1,\ u_6 = 0.$$

The remaining solutions are

$$u_8 = u_{22} = u_{39} = 0,\ u_5 = u_{22} + 1 = u_8 + 1 = 1,\ u_{13} = u_{30} + 1 = 0.$$

Hence the initial state of the control register is `01101011000100111`, and we know this is the correct solution. We don't need to bother with the second possible solution for register 1 since we already found a constellation that correctly reproduces the sequence.

3.10 Design Criteria for Nonlinear Combiners

From the forgoing discussion we derive design criteria for nonlinear combiners:

- The battery registers should be as long as possible.
- The combining function f should have a low linear potential.

How long should the battery registers be? There are some algorithms for "fast" correlation attacks using the Walsh transformation, in particular against sparse linear feedback functions (that use only a small number of taps) [4]. These don't reduce the complexity class of the attack ("exponential in the length of the shortest register") but reduce the cost by a significant factor. So they are able to attack registers with up to 100 coefficients 1 in the feedback function. As a consequence

- The single LFSRs should have a length of at least 200 bits, and use about 100 taps each.

To assess the number n of LFSRs we bear in mind that the combining function should be "correlation immune", in particular have a low linear potential. A well-chosen Boolean function of 16 variables should suffice, but there are no known recommendations in the literature.

Rueppel found an elegant way out to make the correlation attack break down: Use a "time-dependent" combining function, that is a family $(f_t)_{t\in\mathbb{N}}$. The bit u_t of the key stream is calculated by the function f_t. We won't analyze this approach here.

Observing that the correlation attack needs knowledge of the taps, the security could be somewhat better if the taps are secret. Then the attacker has to perform additional exhaustions that multiply the complexity by factors such as 2^{l_1} for the first LFSR alone. This scenario allows choosing LFSRs of somewhat smaller lengths. But bear in mind that for a hardware implementation the taps are parts of the algorithm, not of the key, that is they are public parameters in the sense of Figure 2.1.

Efficiency

LFSRs and nonlinear combiners allow efficient realizations by special hardware that produces one bit per clock cycle. This rate can be enlarged by parallelization. From this point of view estimating the cost of execution on a usual PC processor is somewhat inadequate. Splitting each of the ≥ 200 bit registers into 4 parts of about 64 bits shifting a single register requires at least 4 clock cycles, summing up to 64 clock cycles for 16 registers. Add some clock cycles for the combining function. Thus one single bit would take about 100 clock cycles. A 2-GHz processor, even with optimized implementation, would produce at most $2 \cdot 10^9/100 = 20$ million bits per second.

As a summary we note:

Using LFSRs and nonlinear combining functions we can build useful and fast random generators, especially in hardware.

Unfortunately there is no satisfying theory for the cryptologic security of this type of random generators, even less a mathematical proof. Security is assessed by plausible criteria that—as for bitblock ciphers—are related to the nonlinearity of Boolean functions.