

Chapter 2

Linear Feedback Shift Registers

2.1 Basic Definitions

In a hardware realization of a finite state machine it is attractive to use flip-flops to store the internal state. With n flip-flops we can realize a machine with up to 2^n states. The update function is a Boolean function from $\{0, 1\}^n$ to $\{0, 1\}^n$. We can simplify both the implementation and the description if we restrict ourselves to feedback shift registers.

In a feedback shift register (see Fig. 2.1) we number the flip-flops F_0, \dots, F_{n-1} . In each time step F_i takes the value of F_{i-1} for $i > 0$ and F_0 is updated according to the feedback function $f : \{0, 1\}^n \rightarrow \{0, 1\}$. We will always assume that the value of F_{n-1} is the output of the shift register.

Feedback shift registers are useful tools in coding theory, in the generation of pseudo-random numbers and in cryptography. In this chapter we will summarize all results on linear feedback shift registers relevant to our study of stream ciphers. For other applications of feedback shift registers I recommend the classical book of Solomon W. Golomb [115].

Mathematically the sequence $(a_i)_{i \in \mathbb{N}}$ generated by a shift register is just a sequence satisfying the n -term recursion

$$a_{i+n} = f(a_i, \dots, a_{i+n-1}). \quad (2.1)$$

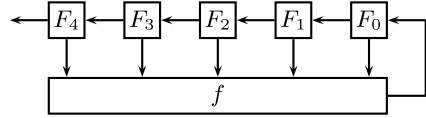
This definition is, of course, not restricted to binary sequences and most of our results will hold for shift register sequences defined over any (finite) field or sometimes even for sequences defined over rings.

We will call a shift register linear if the feedback function is linear. Thus:

Definition 2.1 A *linear feedback shift register* (LFSR) sequence is a sequence $(a_i)_{i \in \mathbb{N}}$ satisfying the recursion

$$a_{i+n} = \sum_{j=0}^{n-1} c_j a_{i+j}. \quad (2.2)$$

Fig. 2.1 A feedback shift register



Since the next value depends only on the preceding n values, the sequence must become periodic. The state $(a_i, \dots, a_{i+n-1}) = (0, \dots, 0)$ leads to the constant sequence 0, thus the period of an LFSR sequence over \mathbb{F}_q can be at most $q^n - 1$. If in addition $c_0 \neq 0$, we can extend the sequence backwards in time via

$$a_i = c_0^{-1} \left(a_{i+n} - \sum_{j=1}^{n-1} c_j a_{i+j+n} \right)$$

which proves that it is ultimately periodic.

As we have already seen in the introduction, a necessary condition for the security of a system is that the generated pseudo-random sequence has a large period. Thus the sequences of maximal period are of special interest.

Definition 2.2 An LFSR sequence over \mathbb{F}_q with period $q^n - 1$ is called an *m-sequence* (maximal sequence).

2.2 Algebraic Description of LFSR Sequences

In this section we develop an algebraic description of LFSR sequences. We especially want to find a closed formula for an LFSR sequence. One way to reach this goal is to study the *companion matrix* of the LFSR sequence. We have

$$\begin{pmatrix} a_{k+1} \\ \vdots \\ a_{k+n-1} \\ a_{k+n} \end{pmatrix} = \begin{pmatrix} 0 & 1 & & 0 \\ \vdots & & \ddots & \\ 0 & 0 & & 1 \\ c_0 & c_1 & \dots & c_{n-1} \end{pmatrix} \begin{pmatrix} a_k \\ \vdots \\ a_{k+n-2} \\ a_{k+n-1} \end{pmatrix} \quad (2.3)$$

and thus

$$\begin{pmatrix} a_k \\ \vdots \\ a_{k+n-2} \\ a_{k+n-1} \end{pmatrix} = \begin{pmatrix} 0 & 1 & & 0 \\ \vdots & & \ddots & \\ 0 & 0 & & 1 \\ c_0 & c_1 & \dots & c_{n-1} \end{pmatrix}^k \begin{pmatrix} a_0 \\ \vdots \\ a_{n-2} \\ a_{n-1} \end{pmatrix}. \quad (2.4)$$

Transforming the companion matrix to Jordan normal form makes it easy to compute the k -th power and transforming it back gives a closed formula for the LFSR sequence.

In the next section we will take another approach that is based on generating functions.

2.2.1 Generating Functions

This section contains the part of the theory of generating functions that we need, but for those who want to learn more about generating functions, I recommend [119].

Definition 2.3 The *generating function* $A(z)$ associated to a sequence $(a_i)_{i \in \mathbb{N}}$ is the formal power series $A(z) = \sum_{i=0}^{\infty} a_i z^i$.

A generating function is useful because it describes an entire sequence with a single algebraic object.

By the recursion (2.2) we find:

$$\begin{aligned} A(z) - \sum_{j=0}^{n-1} c_j A(z) z^{n-j} &= g(z) \\ \iff A(z) \left(1 - \sum_{j=0}^{n-1} c_j z^{n-j} \right) &= g(z) \end{aligned} \quad (2.5)$$

for some polynomial $g(z)$ of degree at most $n-1$.

The polynomial $1 - \sum_{j=0}^{n-1} c_j z^{n-j}$ is important enough to deserve a name.

Definition 2.4 For an LFSR sequence with recursion formula (2.2) we call

$$f(z) = z^n - \sum_{j=0}^{n-1} c_j z^j \quad (2.6)$$

the *feedback polynomial* of the LFSR. The *reciprocal polynomial*¹ is denoted by

$$f^*(z) = z^n f\left(\frac{1}{z}\right) = 1 - \sum_{j=0}^{n-1} c_j z^{n-j}. \quad (2.7)$$

From Eq. (2.5) we derive a closed formula for the generation function of an LFSR sequence:

$$A(z) = \frac{g(z)}{f^*(z)}. \quad (2.8)$$

For the derivation of the closed form of a_i it is useful to begin with the case where the feedback polynomial $f(z)$ has no multiple roots.

¹ $f^*(z)$ is sometimes called the feedback polynomial. As the literature has not adopted a unique notation, it is important to check which notation is being used.

2.2.2 Feedback Polynomials Without Multiple Roots

Let $f(z)$ be a feedback polynomial without multiple roots and let ξ_1, \dots, ξ_n be the different zeros of $f(z)$. Then $f^*(z) = \prod_{j=1}^n (1 - z\xi_j)$ and thus we get the partial fraction decomposition

$$A(z) = \frac{g(z)}{f^*(z)} = \sum_{j=1}^n \frac{b_j}{1 - z\xi_j}. \quad (2.9)$$

All we need to obtain a closed formula from the partial fraction decomposition is the geometric sum

$$\sum_{i=0}^{\infty} z^i = \frac{1}{1 - z}$$

and thus

$$\begin{aligned} A(z) &= \sum_{j=1}^n \frac{b_j}{1 - z\xi_j} \\ &= \sum_{j=1}^n b_j \sum_{i=0}^{\infty} (\xi_j z)^i \\ &= \sum_{i=0}^{\infty} \left(\sum_{j=1}^n b_j \xi_j^i \right) z^i. \end{aligned} \quad (2.10)$$

This gives us the closed formula

$$a_i = \sum_{j=1}^n b_j \xi_j^i \quad (2.11)$$

for the LFSR sequence.

Formula (2.11) holds if the feedback polynomial has no multiple roots. For separable irreducible feedback polynomials we can transform (2.11) to the following theorem. Note that over finite fields and fields with characteristic 0 every polynomial is separable. We will not deal with other fields in this book.

Theorem 2.1 *Let $(a_i)_{i \in \mathbb{N}}$ be an LFSR sequence over \mathbb{F}_q and let ξ be a zero of the irreducible feedback polynomial. Then*

$$a_i = \text{Tr}_{\mathbb{F}_{q^n}/\mathbb{F}_q}(\alpha \xi^i) \quad (2.12)$$

for some $\alpha \in \mathbb{F}_{q^n}$.

Proof We have already proved that the sequence $(a_i)_{i \in \mathbb{N}}$ has a unique closed form (2.11). Since the feedback polynomial is irreducible, its zeros have the form ξ^θ where θ is an automorphism of $\mathbb{F}_{q^n}/\mathbb{F}_q$. But $a_i^\theta = a_i$ for all i . Thus Equation (2.11) is invariant under θ . Therefore the coefficients b_j are conjugated, i.e.

$$a_i = \sum_{\theta \in \text{Aut}(\mathbb{F}_{q^n}/\mathbb{F}_q)} \alpha^\theta (\xi^\theta)^i = \text{Tr}_{\mathbb{F}_{q^n}/\mathbb{F}_q} (\alpha \xi^i). \quad \square$$

Corollary 2.1 *Under the conditions of Theorem 2.1 the period of the sequence is the multiplicative order $o(\xi)$ of ξ .*

As already mentioned in the previous section, the period $q^n - 1$ is of special interest. Thus the following feedback polynomials are special.

Definition 2.5 An irreducible polynomial of degree n over \mathbb{F}_q is *primitive* if the order of its zeros is $q^n - 1$.

2.2.3 Feedback Polynomials with Multiple Roots

Now we want to determine all possible periods of LFSR sequences.

First we consider the easy case where the feedback polynomial is reducible, but has no multiple roots. In this case we can factor the feedback polynomial f and write the generating function (see Eq. (2.8)) of $(a_i)_{i \in \mathbb{N}}$ as

$$A(z) = \frac{g(z)}{f^*(z)} = \sum_{j=1}^k \frac{g_j(z)}{f_j^*(z)}$$

where the polynomials f_j are the different irreducible factors of the feedback polynomial f .

Thus the sequence $(a_i)_{i \in \mathbb{N}}$ can be represented as a sum of k LFSR sequences $(a_i^{(j)})_{i \in \mathbb{N}}$ with irreducible feedback polynomial. By Corollary 2.1 the period of $(a_i^{(j)})_{i \in \mathbb{N}}$ divides $q^{n_j} - 1$ where $n_j = \deg f_j$ and hence the sequence $(a_i)_{i \in \mathbb{N}} = \sum_{j=1}^k (a_i^{(j)})_{i \in \mathbb{N}}$ has period

$$p = \text{lcm}(\pi_1, \dots, \pi_k)$$

where π_j is the period of $(a_i^{(j)})_{i \in \mathbb{N}}$.

To analyze the case of multiple roots we need an additional tool. In this case the partial fraction decomposition of the generation function yields:

$$A(z) = \frac{g(z)}{f^*(z)} = \sum_{j=1}^{n_1} \frac{b_{j,1}}{1 - z\xi_j} + \sum_{j=1}^{n_2} \frac{b_{j,2}}{(1 - z\xi_j)^2} + \dots + \sum_{j=1}^{n_r} \frac{b_{j,r}}{(1 - z\xi_j)^r}$$

with $n_1 \geq n_2 \geq \dots \geq n_r \geq n_{r+1} = 0$ where $\xi_{n_k+1}, \dots, \xi_{n_k}$ are roots of f of multiplicity k . So to get a closed formula we need in addition the power series of $\frac{1}{(1-z)^k}$.

We can find the power series either by computing the $(k-1)$ th derivative of $\frac{1}{1-z} = \sum_{i=0}^{\infty} z^i$ or we use the binomial theorem

$$(1+x)^r = \sum_{i=0}^{\infty} \binom{r}{i} x^i.$$

For a negative integer we get

$$\begin{aligned} \frac{1}{(1-z)^k} &= \sum_{i=0}^{\infty} \binom{-k}{i} (-1)^i z^i \\ &= \sum_{i=0}^{\infty} \binom{k+i-1}{i} z^i \\ &= \sum_{i=0}^{\infty} \binom{k+i-1}{k-1} z^i. \end{aligned}$$

This leads to the closed formula

$$\begin{aligned} a_i &= \sum_{j=0}^{n_1} b_{j,1} \zeta_j^i + \sum_{j=0}^{n_2} b_{j,2} \binom{i+1}{1} \zeta_j^i + \dots + \sum_{j=0}^{n_k} b_{j,k} \binom{i+k-1}{k-1} \zeta_j^i \\ &= \sum_{j=0}^{n_1} b'_{j,1} \zeta_j^i + \sum_{j=0}^{n_2} b'_{j,2} i \zeta_j^i + \dots + \sum_{j=0}^{n_k} b'_{j,k} i^{k-1} \zeta_j^i \end{aligned} \quad (2.13)$$

where the last transformation uses the fact that $\binom{k-1+i}{k-1}$, $k = 1, \dots, n$, is a basis for the polynomials of degree less than k . Note that the converse is also true. Given a sequence in the form of Eq. (2.13) we can reverse all previous steps and find the linear recurrence satisfied by that sequence.

From Eq. (2.13) we can immediately see the period of the sequence $(a_i)_{i \in \mathbb{N}}$. The power series $(\zeta_j^i)_{i \in \mathbb{N}}$ has a period π_i where $\pi_i | q^{n_j} - 1$ and n_j is the degree of the minimal polynomial of ζ_j . And since we are working in \mathbb{F}_q , the period of a polynomial series $(i^k)_{i \in \mathbb{N}}$ is the characteristic p of \mathbb{F}_q . Thus

$$\pi = p \operatorname{lcm}(\pi_1, \dots, \pi_k)$$

where π_1, \dots, π_k are the different orders of $\zeta_1, \dots, \zeta_{n_1}$.

We summarize the results of this section in the following theorem.

Theorem 2.2 *Let $(a_i)_{i \in \mathbb{N}}$ be an LFSR sequence over \mathbb{F}_q , $q = p^e$. Then the period π of $(a_i)_{i \in \mathbb{N}}$ is either*

$$\pi = \operatorname{lcm}(\pi_1, \dots, \pi_k) \quad (2.14)$$

where $\pi_j | q^{n_j} - 1$ and $\sum_{j=1}^k n_j \leq n$ or

$$\pi = p \operatorname{lcm}(\pi_1, \dots, \pi_k) \quad (2.15)$$

where $\pi_j | q^{n_j} - 1$ and $n_1 + \sum_{j=1}^k n_j \leq n$.

Proof We have already proved that the period π must have either the form (2.14) or (2.14). Now we prove the converse that for each such π there is an LFSR sequence with period πa .

Let π be of the form (2.14). Choose $\zeta_j \in \mathbb{F}_{q^{n_j}}$ such that ζ_j has order π_j . Without loss of generality we may assume that $\mathbb{F}_q(\zeta_j) = \mathbb{F}_{q^{n_j}}$, if not just replace n_j by a smaller n'_j . The sequence

$$x_i = \sum_{j=1}^k \operatorname{Tr}_{\mathbb{F}_{q^{n_j}}/\mathbb{F}_q}(\zeta_j^i)$$

is a linear shift register sequence with feedback polynomial

$$f(z) = \prod_{j=1}^k \prod_{l=0}^{n_j-1} (1 - z\zeta_j^{q^l}).$$

The sequence ζ_j has period π since the “subsequences” ζ_j^i and hence $\operatorname{Tr}_{\mathbb{F}_{q^{n_j}}/\mathbb{F}_q}(\zeta_j^i)$ have period π_j ($1 \leq j \leq k$).

If π is of the form (2.15), we find that the sequence

$$x_i = i \operatorname{Tr}_{\mathbb{F}_{q^{n_j}}/\mathbb{F}_q}(\zeta_1^i) + \sum_{j=2}^k \operatorname{Tr}_{\mathbb{F}_{q^{n_j}}/\mathbb{F}_q}(\zeta_j^i)$$

is a linear shift register sequence with feedback polynomial

$$f(z) = \left(\prod_{l=0}^{n_1-1} (1 - z\zeta_1^{q^l})^2 \right) \left(\prod_{j=1}^k \prod_{l=0}^{n_j-1} (1 - z\zeta_j^{q^l}) \right)$$

and period $\pi = p \operatorname{lcm}(\pi_1, \dots, \pi_k)$. The additional factor p is for the period of the polynomial i in \mathbb{F}_q . \square

2.2.4 LFSR Sequences as Cyclic Linear Codes

Another description of LFSR sequences is based on coding theory.

The LFSR defines a linear mapping from its initial state (a_0, \dots, a_{n-1}) to its output sequence $(a_i)_{i \in \mathbb{N}}$. For fixed N we may interpret the mapping

$$C : (a_0, \dots, a_{n-1}) \mapsto (a_0, \dots, a_{N-1})$$

as a linear code of length N and dimension n .

A parity check matrix of the code is

$$H = \begin{pmatrix} c_0 & \dots & c_{n-1} & -1 & 0 & \dots & 0 \\ 0 & c_0 & \dots & c_{n-1} & -1 & 0 & \dots & 0 \\ & & & \ddots & & \ddots & \ddots & \\ 0 & & \dots & 0 & c_0 & \dots & c_{n-1} & -1 \end{pmatrix}. \quad (2.16)$$

If we look at a full period of the LFSR, i.e. if we choose $N = p$, then the resulting linear code is cyclic and $f^*(z)$ is its parity check polynomial.

The code C also has a unique *systematic generator matrix*

$$G = \left(\begin{array}{cc|ccc} 1 & & 0 & c_{n,0} & \dots & c_{N-1,0} \\ & \ddots & & \vdots & & \vdots \\ 0 & & 1 & c_{n,n-1} & \dots & c_{N-1,n-1} \end{array} \right). \quad (2.17)$$

We have $(a_0, \dots, a_{N-1}) = (a_0, \dots, a_{n-1})G$, i.e.

$$a_k = \sum_{i=0}^{n-1} c_{k,i} a_i. \quad (2.18)$$

We will use this linear representation of the element a_k in terms of the initial state in several attacks.

2.3 Properties of m-Sequences

2.3.1 Golomb's Axioms

Linear shift register sequences of maximal length (m-sequences) have many desirable statistical properties.

The best known of these properties is that they satisfy Golomb's axioms for pseudo-random sequences [115].

We study a periodic binary sequence $(a_i)_{i \in \mathbb{N}}$ with period length p . Then the three axioms for $(a_i)_{i \in \mathbb{N}}$ to be a pseudo-random sequence are:

- (G1) In every period the number of ones is nearly equal to the number of zeros, more precisely the difference between the two numbers is at most 1:

$$\left| \sum_{i=1}^p (-1)^{a_i} \right| \leq 1.$$

- (G2) For any k -tuple b , let $N(b)$ denote the number of occurrences of the k -tuple b in one period.

Then for any k with $1 \leq k \leq \log_2 p$ we have

$$|N(b) - N(b')| \leq 1$$

for any k -tuples b and b' .

- (G2') A sequence of consecutive ones is called a *block* and a sequence of consecutive zeros is called a *gap*. A *run* is either a block or a gap.

In every period, one half of the runs has length 1, one quarter of the runs has length 2, and so on, as long as the number of runs indicated by these fractions is greater than 1.

Moreover, for each of these lengths the number of blocks is equal to the number of gaps.

- (G3) The auto-correlation function

$$C(\tau) = \sum_{i=0}^{p-1} (-1)^{a_i} (-1)^{a_{i+\tau}}$$

is two-valued.

Axiom (G1) is called the *distribution test*, Axiom (G2) is the *serial test* and Axiom (G3) is the *auto-correlation test*. In [115] Golomb uses (G2') instead of (G2). Axiom (G2) was introduced in [169] and is in some respects more useful than the original axiom.

The distribution test (G1) is a special case of the serial test (G2). However, (G1) is retained for historical reasons, and sequences which satisfy (G1) and (G3), but not (G2), are also important.

Theorem 2.3 (Golomb [115]) *Every m-sequence satisfies (G1)–(G3).*

Proof An m-sequence is characterized by the fact that the internal state of the linear feedback shift register runs through all elements of $\mathbb{F}_2^n \setminus \{(0, \dots, 0)\}$. Since at any time the next n output bits form the current internal state, this means that (a_t, \dots, a_{t+n-1}) runs over all elements of $\mathbb{F}_2^n \setminus \{(0, \dots, 0)\}$ where t runs from 0 to $2^n - 1$. This proves

$$N(a_1, \dots, a_k) = \begin{cases} 2^{n-k} - 1 & \text{for } a_1 = \dots = a_k = 0, \\ 2^{n-k} & \text{otherwise.} \end{cases}$$

Thus an m-sequence passes the serial test for blocks of length up to n and hence it satisfies (G2) and (G1).

A run of length k is just a subsequence of the form $1, 0, 0, \dots, 0, 1$ with k zeros and a block of length k is a subsequence of the form $0, 1, 1, \dots, 1, 0$. We have already proved that an m-sequence contains exactly 2^{n-k-2} subsequences of type $k \leq n-2$. This is the statement of (G2').

We find $C(0) = 2^n - 1$ as one value of the auto-correlation function. We now prove $C(\tau) = -1$ for $0 < \tau < 2^n - 1$. By Theorem 2.1 we have $a_i = \text{Tr}_{\mathbb{F}_{2^n}/\mathbb{F}_2}(\alpha \xi^i)$ for a primitive element ξ of \mathbb{F}_{2^n} and $a_{i+\tau} = \text{Tr}_{\mathbb{F}_{2^n}/\mathbb{F}_2}(\alpha' \xi^i)$. Note that we have the same ξ in both equations, since $(a_i)_{i \in \mathbb{N}}$ and $(a_{i+\tau})_{i \in \mathbb{N}}$ satisfy the same recurrence. Thus $a_i + a_{i+\tau} = \text{Tr}_{\mathbb{F}_{2^n}/\mathbb{F}_2}((\alpha + \alpha')\xi^i)$ and hence $(a_i + a_{i+\tau})_{i \in \mathbb{N}}$ is also an m-sequence. By (G1) we have

$$C(\tau) = \sum_{i=0}^{p-1} (-1)^{a_i + a_{i+\tau}} = -1.$$

Thus the auto-correlation function takes just the two values $2^n - 1$ and -1 . □

Besides the Golomb axioms, m-sequences also satisfy other interesting equations:

Theorem 2.4 *Every m-sequence satisfies:*

(a) *For every $0 < k < 2^n - 1$ there exists a δ for which*

$$a_i + a_{i+k} = a_{i+\delta}$$

for all $i \in \mathbb{N}$. This is called the shift-and-add property.

(b) *There exists a τ such that*

$$a_{i2^j + \tau} = a_{i+\tau}$$

for all $i, j \in \mathbb{N}$. This is called the constancy on cyclotomic cosets.

Proof We have already used and proved the shift-and-add property when we demonstrated that an m-sequence satisfies the auto-correlation test.

By Theorem 2.1 we know that $a_i = \text{Tr}_{\mathbb{F}_{2^n}/\mathbb{F}_2}(\alpha \xi^i)$ for some $\alpha \in \mathbb{F}_{2^n}$ and a primitive $\xi \in \mathbb{F}_{2^n}$. We choose τ such that $\xi^\tau = \alpha^{-1}$.

Then

$$\begin{aligned} a_{i+\tau} &= \text{Tr}_{\mathbb{F}_{2^n}/\mathbb{F}_2}(\alpha \xi^{i+\tau}) \\ &= \text{Tr}_{\mathbb{F}_{2^n}/\mathbb{F}_2}(\xi^i) \\ &= \text{Tr}_{\mathbb{F}_{2^n}/\mathbb{F}_2}(\xi^{i2^j}) \quad \text{since } x \mapsto x^{2^j} \text{ is an automorphism of } \mathbb{F}_{2^n}/\mathbb{F}_2 \\ &= \text{Tr}_{\mathbb{F}_{2^n}/\mathbb{F}_2}(\alpha \xi^{i2^j + \tau}) \\ &= a_{i2^j + \tau}. \end{aligned}$$

□

The shift-and-add property is of special interest since it characterizes the m-sequences uniquely.

Theorem 2.5 *Every sequence which satisfies the shift-and-add property is an m-sequence.*

Proof Let $A = (a_i)_{i \in \mathbb{N}}$ be a sequence of period p which has the shift-and-add property. Then the p shifts of the sequence, together with the zero sequence, form an elementary Abelian group. It follows that $p + 1 = 2^n$ for some $n \in \mathbb{N}$. Let A_k denote the sequence $(a_{i+k})_{i \in \mathbb{N}}$. Any n successive shifts of the sequence A form a basis of the elementary Abelian group, thus we can write A_n as a linear combination of A_0, \dots, A_{n-1} , i.e.

$$A_n = \sum_{k=0}^{n-1} c_k A_k.$$

Reading the last equation element-wise gives

$$a_{i+n} = \sum_{k=0}^{n-1} c_k a_{i+k},$$

i.e. the sequence A satisfies a linear recurrence. Since the period of A is $p = 2^n - 1$, it is an m-sequence. \square

2.3.2 Sequences with Two Level Auto-Correlation

It is a natural question whether the converse of Theorem 2.3 holds. Golomb conjectured that it does and indicated in a passage of his book (Sect. 4.7 in [115]) that he had a proof, but the actual answer turns out to be negative (see also [114]).

To put this answer in a bigger context we will study sequences which satisfy Axiom (G3), which have a strong connection to design theory. We make the following definition.

Definition 2.6 Let G be an additive group of order v and let D be a k -subset of G .

D is called a (v, k, λ) -difference set of G , if for every element $h \neq 0$ in G the equation

$$h = d - d'$$

has exactly λ solutions with $d, d' \in D$. If $G = \mathbb{Z}/v\mathbb{Z}$ is a cyclic group we speak of a cyclic (v, k, λ) -difference set.

The connection between sequences satisfying (G3) and difference sets is given by the following theorem.

Theorem 2.6 *The following statements are equivalent.*

- (1) *There exists a periodic sequence of period length v over \mathbb{F}_2 with two level auto-correlation and k ones in its period.*
- (2) *There exists a cyclic (v, k, λ) -difference set.*

Proof Let a_0, \dots, a_{v-1} be the period of a sequence with two level auto-correlation. This means the auto-correlation function satisfies $C(0) = v$ and $C(\tau) = x < v$ for $1 \leq \tau \leq v - 1$.

For $1 \leq \tau \leq v - 1$ let $\lambda_\tau = |\{i \mid a_i = a_{i+\tau} = 1, 0 \leq i \leq v - 1\}|$. Then

$$\begin{aligned} |\{i \mid a_i = 1, a_{i+\tau} = 0, 0 \leq i \leq v - 1\}| &= k - \lambda_\tau, \\ |\{i \mid a_i = 0, a_{i+\tau} = 1, 0 \leq i \leq v - 1\}| &= k - \lambda_\tau, \\ |\{i \mid a_i = a_{i+\tau} = 0, 0 \leq i \leq v - 1\}| &= v + \lambda_\tau - 2k. \end{aligned}$$

Thus $x = \lambda_\tau + (v + \lambda_\tau - 2k) - 2(k - \lambda_\tau) = v - 4(k - \lambda_\tau)$, i.e. $\lambda_\tau = \lambda$ independent of τ .

Let $D = \{i \mid a_i = 1\}$. Then $h = d - d'$ has exactly $\lambda = \lambda_h$ solutions with $d, d' \in D$.

For the converse, let D be a cyclic (v, k, λ) -difference set and define the sequence (a_i) by $a_i = 1$ if $i \in D$ and $a_i = 0$ for $i \notin D$. The definition of a difference set says that there are λ indices with $a_i = a_{i+\tau} = 1$ and, as above, we obtain $C(\tau) = \lambda_\tau + (v + \lambda_\tau - 2k) - 2(k - \lambda_\tau) = v - 4(k - \lambda_\tau)$ for $1 \leq \tau \leq v - 1$, i.e. the auto-correlation function is two leveled. \square

If we apply Theorem 2.6 to an m-sequence we obtain a difference set with parameters $v = 2^n - 1$, $k = 2^{n-1}$ and $\lambda = 2^{n-3}$. This is an example of a Hadamard difference set.

Definition 2.7 A $(4n - 1, 2n - 1, n - 1)$ -difference set is called a *Hadamard difference set*.

Hadamard difference sets are of course strongly connected with the well-known Hadamard matrices.

Definition 2.8 A *Hadamard matrix* of order n is an $n \times n$ matrix H with entries ± 1 which satisfies $HH^t = nI$.

The connection is that we can construct a $(4n) \times (4n)$ Hadamard matrix from a $(4n - 1, 2n - 1, n - 1)$ -difference set. Let D be a $(4n - 1, 2n - 1, n - 1)$ -difference set over the group $G = \{g_1, \dots, g_{4n-1}\}$ and define $H = (h_{i,j})_{i,j=0,\dots,4n-1}$ by

$$h_{ij} = \begin{cases} 1 & \text{if } i = 0 \text{ or } j = 0, \\ 1 & \text{if } i, j \geq 1 \text{ and } g_i + g_j \in D, \\ 0 & \text{if } i, j \geq 1 \text{ and } g_i + g_j \notin D. \end{cases}$$

Then a short calculation shows that H is a Hadamard matrix.

The opposite implication is false since the Hadamard difference set needs the group G acting on it, whereas a Hadamard matrix need not have any symmetries. Nevertheless, we see that there are many links between pseudo-random sequences and other interesting combinatorial objects. (Besides those we have mentioned here, there are also strong links to designs and coding theory.)

This is not the place to go deeper into the theory of Hadamard Matrices, but to answer the question we posed at the beginning of this section we mention that Cheng and Golomb [51] use the Hadamard $(127, 63, 31)$ -difference set of type E (given by Baumert [17]) to construct the sequence:

```
111110111100111111001001011101010111100011000001001101110011000
110110111010010001101000010101001101001010001110110000101000000
```

which satisfies (G1), (G2) and (G3) but is not an m-sequence.

2.3.3 Cross-Correlation of m-Sequences

Sequences with low correlation are intensively studied in the literature (see [126] for an overview). Since m-sequences have ideal auto-correlation properties, it is interesting to study the cross-correlation function for pairs of m-sequences. Many families of low correlation functions have been constructed in this way.

We use Eq. (2.12) to represent the m-sequence. We can shift it so that we get the form

$$a_i = \text{Tr}(\xi^i).$$

The second m-sequence can be assumed, without loss of generality, to be

$$b_i = \text{Tr}(\xi^{di})$$

for some d with $\gcd(d, q^n - 1) = 1$. The cross-correlation function depends only on d . This motivates the following definition:

Definition 2.9 Let ξ be a primitive element of \mathbb{F}_{q^n} and let $\omega \in \mathbb{C}$ be a q -th root of unity. For each d with $\gcd(d, q^n - 1) = 1$ we define the *cross-correlation function*

$$\theta_{1,d}(\tau) = \sum_{x \in \mathbb{F}_{q^n}^*} \omega^{\text{Tr}(\xi^\tau x - x^d)}.$$

We will not need $\theta_{1,d}$ in the following, so we keep this section short and just sketch a link to bent functions, which are important in cryptography. We state the following without proof.

Theorem 2.7 (Gold [106], Kasami [144]) *Let $q = 2$ and $\omega = -1$. For $1 \leq k \leq n$ let $e = \gcd(n, k)$. If n/e is odd and $d = 2^k + 1$ or $d = 2^{2k} - 2^k + 1$, then $\theta_{1,d}$ takes the following three values:*

- $-1 + 2^{(n+e)/2}$ is taken $2^{n-e-1} + 2^{(n-e-2)/2}$ times.
- -1 is taken $2^n - 2^{n-e} - 1$ times.
- $-1 - 2^{(n+e)/2}$ is taken $2^{n-e-1} - 2^{(n-e-2)/2}$ times.

A pair of m-sequences whose cross-correlation function takes only the three values -1 , $-1 + 2^{\lfloor (n+2)/2 \rfloor}$ and $-1 - 2^{\lfloor (n+2)/2 \rfloor}$ is called a *preferred pair*. Theorem 2.7 allows the construction of preferred pairs for n not divisible by 4. For $n \equiv 0 \pmod 4$ preferred pairs do not exist (see [185]).

What makes this interesting for cryptographic purposes is the following. To avoid attacks such as differential cryptanalysis [26] or linear cryptanalysis [179], the S-box of a block cipher must be as far from a linear mapping as possible. The appropriate measure of distance between two functions is provided by the Walsh transform.

Definition 2.10 Let $f : \mathbb{F}_{2^n} \rightarrow \mathbb{F}_2$. The *Walsh transform* of f is

$$f^W(a) = \sum_{x \in \mathbb{F}_{2^n}} (-1)^{f(x) + \text{Tr}(ax)}.$$

The image of f^W is the *Walsh spectrum* of f .

A linear function will contain $\pm 2^n$ in its Walsh spectrum. A function provides the strongest resistance against a linear cryptanalysis if its Walsh spectrum contains only values of small absolute value. One can prove that the Walsh spectrum of f must contain at least a value of magnitude $2^{\lceil n/2 \rceil}$ (see, for example, [48]).

Definition 2.11 For even n , a function $f : \mathbb{F}_{2^n} \rightarrow \mathbb{F}_2$ is *bent* if $f^W(x) \leq 2^{n/2}$ for all $x \in \mathbb{F}_{2^n}$.

For odd n , we call a function *almost bent* if $f^W(x) \leq 2^{(n+1)/2}$ for all $x \in \mathbb{F}_{2^n}$.

From two m-sequences $a_i = \text{Tr}(\xi^i)$ and $b_i = \text{Tr}(\xi^{di})$ we construct a Boolean function $f : \mathbb{F}_2^n \rightarrow \mathbb{F}_2$ by defining $f(\xi^i) = \xi^{di}$ and $f(0) = 0$. Applying this construction to a preferred pair with odd n results in an almost bent function.

2.4 Linear Complexity

2.4.1 Definition and Basic Properties

We have seen in the previous section that m-sequences have very desirable statistical properties. However, as linear functions, LFSR sequences are unusable as cryptographic pseudo-random generators. First note that the first n output bits of an LFSR

form its initial state, thus a LFSR fails against a known plaintext attack. Even if the feedback polynomial of the LFSR is unknown to the cryptanalyst, the system is still insecure. The first n output bits give us the initial state and the next n output bits give us n equations of the form

$$\begin{pmatrix} a_{n-1} & a_{n-2} & \cdots & a_0 \\ a_n & a_{n-1} & \ddots & \\ \vdots & & \ddots & \\ a_{2n-2} & a_{2n-3} & \cdots & a_{n-1} \end{pmatrix} \begin{pmatrix} c_{n-1} \\ c_{n-2} \\ \vdots \\ c_0 \end{pmatrix} = \begin{pmatrix} a_n \\ a_{n+1} \\ \vdots \\ a_{2n-1} \end{pmatrix}. \quad (2.19)$$

The determination of the unknown feedback coefficients c_i therefore only requires the solution of a system of n linear equations. A matrix of the form given in Eq. (2.19) is called a *Toeplitz matrix*. Toeplitz matrices appear in many different contexts. A lot is known about the solution of Toeplitz systems (see, for example, [142]).

In Sect. 2.4.2 we will learn a quadratic algorithm that computes not only the solution of the system (2.19), but gives us a lot of extra information. So an LFSR is not secure, even if its feedback polynomial is secret.

On the other hand, it is clear that every periodic sequence can be generated by a linear feedback shift register—simply take an LFSR of the same size as the period. It is therefore natural to use the length of the shortest LFSR that generates the sequence as a measure of its cryptographic security.

Definition 2.12 The *linear complexity* $\mathcal{L}((a_i)_{i=0,\dots,n-1})$ of a finite sequence a_0, \dots, a_{n-1} is the length of the shortest linear feedback shift register that produces that sequence.

The following theorem summarizes some very basic properties of linear complexity which follows directly from the definition.

Theorem 2.8

(a) A sequence of length n has linear complexity at most n :

$$\mathcal{L}((a_i)_{i=0,\dots,n-1}) \leq n.$$

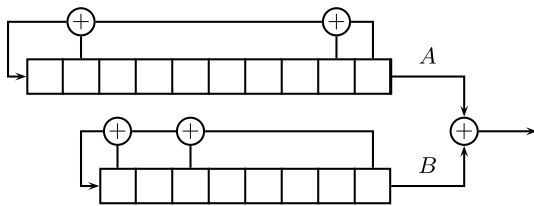
(b) The linear complexity of the subsequence $(a_i)_{i=0,\dots,k-1}$ of $(a_i)_{i=0,\dots,n-1}$ satisfies

$$\mathcal{L}((a_i)_{i=0,\dots,k-1}) \leq \mathcal{L}((a_i)_{i=0,\dots,n-1}).$$

Proof

(a) The first n output bits of a shift register of length n is simply its initial state. Thus every shift register of length n can produce any finite sequence of length n as output.

Fig. 2.2 The sum of two LFSRs



- (b) Any shift register that produces the sequence $(a_i)_{i=0,\dots,n-1}$ also produces the subsequence $(a_i)_{i=0,\dots,k-1}$. □

It is noteworthy that the bound of Theorem 2.8 (a) is sharp, as the following example shows.

Lemma 2.1 *The sequence $0, \dots, 0, 1$ ($n - 1$ zeros) has linear complexity n .*

Proof Assume that the sequence is generated by a shift register of length $k < n$. Since the first k symbols are 0 the shift register must be initialized with 0. But any LFSR initialized with 0 generates the constant sequence 0. □

Lemma 2.1 demonstrates that a high linear complexity is just a necessary but not sufficient condition for a good pseudo-random sequence.

Next we study the sum of two LFSR sequences, which is generated by the device shown in Fig. 2.2.

Theorem 2.9 *For two sequences (a_i) and (b_i) we have*

$$\mathcal{L}(a_i + b_i) \leq \mathcal{L}(a_i) + \mathcal{L}(b_i).$$

Proof Consider the generating functions

$$A(z) = \frac{g_A(z)}{f_A^*(z)} \quad \text{and} \quad B(z) = \frac{g_B(z)}{f_B^*(z)}$$

of the two LFSR sequences. Then the sum $(a_i + b_i)$ has the generating function $S(z) = A(z) + B(z)$.

Thus

$$S(z) = \frac{g_A(z)f_B^*(z) + g_B(z)f_A^*(z)}{f_A^*(z)f_B^*(z)}$$

which implies by Sect. 2.2.1 that $(a_i + b_i)$ can be generated by an LFSR with feedback polynomial $f_A(z)f_B(z)$, i.e.

$$\mathcal{L}(a_i + b_i) \leq \mathcal{L}(a_i) + \mathcal{L}(b_i).$$

(Note that $\text{lcm}(f_A(z), f_B(z))$ is the feedback polynomial of the minimal LFSR that generates $(a_i + b_i)$.) \square

Theorem 2.9 shows that a linear combination of several linear feedback shift registers does not result in a cryptographic improvement. In fact there is a much more general theorem: Any circuit of n flip-flops which contains only XOR-gates can be simulated by an LFSR of size at most n (see [46]).

Theorem 2.9 has a corollary, which will be crucial for the next section.

Corollary 2.2 *If*

$$\mathcal{L}((a_i)_{i=0,\dots,n-2}) < \mathcal{L}((a_i)_{i=0,\dots,n-1})$$

then

$$\mathcal{L}((a_i)_{i=0,\dots,n-1}) \geq n - \mathcal{L}((a_i)_{i=0,\dots,n-2}). \quad (2.20)$$

Proof Let (b_i) be the sequence $0, \dots, 0, 1$ ($n-1$ zeros) and let $a'_i = a_i + b_i$.

Since $\mathcal{L}((a_i)_{i=0,\dots,n-2}) < \mathcal{L}((a_i)_{i=0,\dots,n-1})$, the minimal LFSR that produces the sequence $(a_i)_{i=0,\dots,n-2}$ will produce $a_{n-1} + 1$ as n th output, i.e.

$$\mathcal{L}((a_i)_{i=0,\dots,n-2}) = \mathcal{L}((a'_i)_{i=0,\dots,n-1}).$$

Applying Theorem 2.9 to the sequences (a_i) and (a'_i) we obtain

$$\mathcal{L}((a_i)_{i=0,\dots,n-1}) + \mathcal{L}((a'_i)_{i=0,\dots,n-1}) \leq \mathcal{L}((b_i)_{i=0,\dots,n-1}) = n,$$

where the last equality is due to Lemma 2.1. \square

In the next section we will prove that we even have equality in (2.20).

2.4.2 The Berlekamp-Massey Algorithm

In 1968 E.R. Berlekamp [20] presented an efficient algorithm for decoding BCH-codes (an important class of cyclic error-correcting codes). One year later, Massey [178] noticed that the decoding problem is in its essential parts equivalent to the determination of the shortest LFSR that generates a given sequence.

We present the algorithm, which is now known as Berlekamp-Massey algorithm, in the form which computes the linear complexity of a binary sequence.

In cryptography we are interested only in binary sequences, in contrast to coding theory where the algorithm is normally used over larger finite fields. To simplify the notation we specialize Algorithm 2.1 to the binary case.

Theorem 2.10 *In the notation of Algorithm 2.1, L_i is the linear complexity of the sequence x_0, \dots, x_{i-1} and f_i is the feedback polynomial of the minimal LFSR that generates x_0, \dots, x_{i-1} .*

Algorithm 2.1 The Berlekamp-Massey algorithm

```

1: {initialization}
2:  $f_0 \leftarrow 1, L_0 \leftarrow 0$ 
3:  $f_{-1} \leftarrow 1, L_{-1} \leftarrow 0$ 
4: {Compute linear complexity}
5: for  $i$  from 0 to  $n - 1$  do
6:    $L_i = \deg f_i$ 
7:    $d_i \leftarrow \sum_{j=0}^{L_i} \text{coeff}(f_i, L_i - j)x_{i-j}$ 
8:   if  $d_i = 0$  then
9:      $f_{i+1} \leftarrow f_i$ 
10:  else
11:     $m \leftarrow \begin{cases} \max\{j \mid L_j < L_{j+1}\} & \text{if } \{j \mid L_j < L_{j+1}\} \neq \emptyset \\ -1 & \text{if } \{j \mid L_j < L_{j+1}\} = \emptyset \end{cases}$ 
12:    if  $m - L_m \geq i - L_i$  then
13:       $f_{i+1} \leftarrow f_i + X^{(m-L_m)-(i-L_i)} f_m$ 
14:    else
15:       $f_{i+1} \leftarrow X^{(i-L_i)-(m-L_m)} f_i + f_m$ 
16:    end if
17:  end if
18: end for

```

Proof As a first step we will prove by induction on i that f_i is a feedback polynomial of an LFSR that produces x_0, \dots, x_{i-1} . In the second step we prove the minimality.

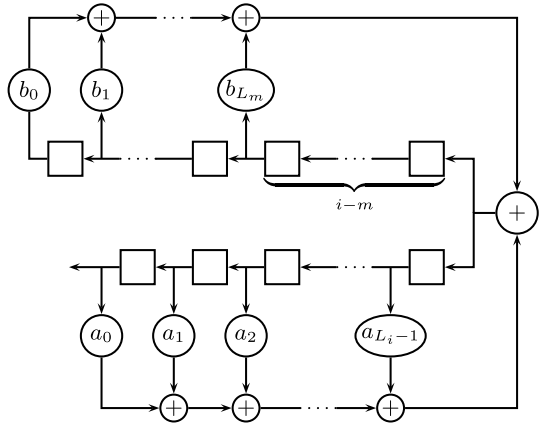
We start the induction with $i = 0$. The empty sequence has, by definition, linear complexity 0 and the “generating LFSR” has feedback polynomial 1.

Now suppose that f_i is the feedback polynomial of an LFSR which generates x_0, \dots, x_{i-1} . We prove that f_{i+1} is the feedback polynomial of an LFSR which generates x_0, \dots, x_i . In line 7, Algorithm 2.1 tests if the sequence x_0, \dots, x_i is also generated by the LFSR with feedback polynomial f_i . If this is the case we can keep the LFSR.

Now consider the case that the LFSR with feedback polynomial f_i fails to generate x_0, \dots, x_i . In this case we need to modify the LFSR. To this we use in addition to the LFSR with feedback polynomial $f_i(x) = \sum_{i=1}^{L_i} a_i x^i$ the latest time step m in which the linear complexity of the sequence was increased. Let $f_m(x) = \sum_{j=0}^{L_m} b_j x^j$ be the feedback polynomial for that time step. For the first steps in which no such time step is present, we use the conventional values $m = -1$, $f_{-1} = 1$, $L_{-1} = 0$. The reader can easily check that the following argument works with this definition.

With these two shift registers we construct the automaton described in Fig. 2.3.

In the lower part we see the feedback shift register with feedback polynomial f_i . It generates the sequence $x_0, \dots, x_i, x_{i+1} + 1$. So at time step $i - L_i$ it computes the wrong feedback $x_{i+1} + 1$. We correct this error with the feedback register shown in the upper part. We use the feedback polynomial f_m to test the sequence x_0, \dots . In the first time steps this test will output 0. At time step $m - L_m$ we will use at first

Fig. 2.3 Construction for the Berlekamp-Massey algorithm

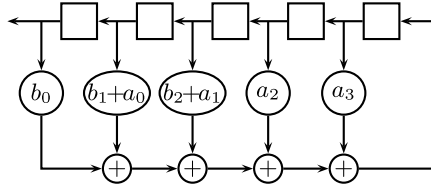
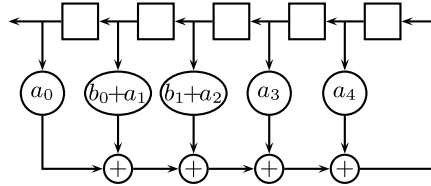
the value x_{m+1} and the test will output 1. We will feed back this 1 in such a way that it cancels with the 1 in the wrong feedback $x_{i+1} + 1$ of the first LFSR. To achieve this goal we have to delay the upper part by $i - m$.

We can combine the two registers into one LFSR by reflecting the upper part to the lower part. Here we must distinguish between the cases $m - L_m \geq i - L_i$ and $m - L_m \leq i - L_i$. (For $m - L_m = i - L_i$ both cases give the same result.)

If $L_m + (i - m) \geq L_i$, then the upper part contains more flip-flops. In this case the combination of both LFSRs looks like Fig. 2.4 (a).

If $L_m + (i - m) \leq L_i$, then the lower part contains more flip-flops. In this case the combination of both LFSRs looks like Fig. 2.4 (b).

In both cases the reader can check that the diagrams in Fig. 2.4 are described algebraically by the formulas given in lines 13 and 15 of the algorithm.

(a) The case $L_m + (i - m) \geq L_i$ (drawing with $L_i = 4$, $L_m = 2$, $i - m = 3$)(b) The case $L_m + (i - m) \leq L_i$ (drawing with $L_i = 5$, $L_m = 1$, $i - m = 3$)**Fig. 2.4** Combination of the two LFSRs of Fig. 2.3

At this point we have proved that f_i is a feedback polynomial of an LFSR that generates x_0, \dots, x_i . Now we prove the minimality.

At time $t = 0$ we have the empty sequence and the empty shift register, which is clearly minimal. Now look at the step $i \rightarrow i + 1$. If $\deg f_{i+1} = \deg f_i$ then f_{i+1} is clearly minimal (part (b) of Theorem 2.8).

The only interesting case is when $\deg f_{i+1} > \deg f_i$. This happens if and only if $d_i \neq 0$ and $m - L_m < i - L_i$.

In this case we may apply Corollary 2.2 to conclude

$$\mathcal{L}(x_0, \dots, x_i) \geq i + 1 - L_i.$$

To prove $\deg f_{i+1} = L_{i+1} = \mathcal{L}(x_0, \dots, x_i)$ it suffices to show $L_{i+1} = i + 1 - L_i$.

But

$$\begin{aligned} L_{i+1} &= (i - L_i) + (m - L_m) + \deg f_i \\ &= i - m + L_m \\ &= (i + 1) - (m + 1 - L_m). \end{aligned}$$

Since we have chosen m in such a way that $\deg f_m < \deg f_{m+1}$, we may use the induction hypothesis on that step and conclude $L_{m+1} = m + 1 - L_i$. But again, by choice of m , we have $\deg f_{m+1} = \deg f_{m+2} = \dots = \deg f_i$. So we get

$$\begin{aligned} L_{i+1} &= (i + 1) - (m + 1 - L_m) \\ &= (i + 1) - L_i. \end{aligned} \tag{2.21}$$

By Corollary 2.2 this proves $L_{i+1} = \mathcal{L}(x_0, \dots, x_i)$. \square

As part of the proof we have improved the statement of Corollary 2.2.

Corollary 2.3 *If*

$$\mathcal{L}((a_i)_{i=0,\dots,n-2}) < \mathcal{L}((a_i)_{i=0,\dots,n-1})$$

then

$$\mathcal{L}((a_i)_{i=0,\dots,n-1}) = n - \mathcal{L}((a_i)_{i=0,\dots,n-2}).$$

Proof The only case in which $\mathcal{L}((a_i)_{i=0,\dots,n-2}) < \mathcal{L}((a_i)_{i=0,\dots,n-1})$ is treated in line 15 of Algorithm 2.1. In the proof of Theorem 2.10 we have shown (Eq. (2.21)) that in this case $L_n = n - L_{n-1}$. \square

In each loop of the Berlekamp-Massey algorithm we need $O(L_i) = O(i)$ time steps, thus the computation of the linear complexity of a sequence of length n takes just $O(n^2)$ time steps. In comparison, the solution of a system of n linear equations, which we used in the beginning of this section, needs $O(n^3)$ time steps. In addition it has the advantage that it computes the linear complexity of all prefixes of the sequence. This is helpful if we want to study the linear complexity profile of the sequence (see Sect. 2.5).

2.4.3 Asymptotic Fast Computation of Linear Complexity

The Berlekamp-Massey algorithm is already very fast, so it may come as a surprise that we can compute the linear complexity even more rapidly.

One can lower the complexity of the computation of $\mathcal{L}(a_0, \dots, a_{n-1})$ from $O(n^2)$ to $O(n \log n (\log \log n)^2)$ (see, for example, [142]). The following algorithm is my own development and has the advantage that it has a small constant in the O -term and it computes, in addition to the feedback polynomial f_n , the whole sequence $\mathcal{L}(a_0, \dots, a_{i-1}), i \leq n$.

The idea is a divide and conquer variant of the original Berlekamp-Massey algorithm. The main part is the function $\text{Massey}(i, i')$ which allows us to go from time step i directly to time step i' .

We use the following notation: For any time step i , let f_i be the minimal feedback polynomial of the sequence a_0, \dots, a_{i-1} . The degree of f_i is L_i and by $m(i)$ we denote the largest $m < i$ with $L_m < L_i$. If such an m does not exist we use the conventional values $m(i) = -1$ and $f_{-1} = 1$.

By $A(z) = \sum_{j=0}^N a_j z^j$ we denote the generating function of the sequence A_i and by $A^*(z) = z^N A(1/z)$ we denote the reciprocal polynomial.

To simplify the presentation of Algorithm 2.2 we take the values $m(i)$, L_i and so on as known. We will see later how to compute these values quickly.

Lines 3 and 4 need some further explanation. As we can see in the final recursion step (line 8), we access only the coefficient $N - i + L_i$ of D_i . So there is no need to compute the full polynomial $D_i = A^* f_i$. We will prove in Theorem 2.13 that only the coefficients from $z^{N-i'+L_m+2}$ to $z^{\max\{L_i-i, L_{m(i)}-m(i)\}}$ (inclusive) are needed when we call $\text{Massey}(i, i')$. Since for a sequence with typical complexity profile $m \approx i$ and $L_i \approx L_m$, this means that we need only about $i' - i$ coefficients. For the complexity of the algorithm it is crucial to implement this optimization.

Before we prove the correctness of the algorithm we look at its running time. Let $T(d)$ be the running time of $\text{Massey}(i, i + d)$.

For $d > 1$ we have two recursive calls of the function Massey and the computations in lines 3, 4 and 6. This leads to the recursion

$$T(d) = 2T(d/2) + 4M(d/2, d/4) + 8M(d/4, d/4).$$

We neglect the time for the additions and memory access in this analysis. Asymptotically it does not matter anyway, and computer experiments show that even for small d it has very little impact on the constants.

$M(k, l)$ denotes the complexity of multiplying a polynomial of degree k by a polynomial of degree l .

Even with “school multiplication” $M(k, l) = kl$ we get $T(n) = 2n^2$ which is not too bad in comparison with the original Berlekamp-Massey algorithm, which needs $\approx n^2/4$ operations.

With multiplication based on the fast Fourier transform we get $T(n) = O(n \log^2 n \log \log n)$. However, the constant is not so good in this case.

Algorithm 2.2 Massey(i, i')

Require: The check polynomials $D_i = A^* f_i$ and $D_{m(i)} = A^* f_{m(i)}$, $L_i = \deg f_i$, $L_{m(i)} = \deg f_{m(i)}$

Ensure: Polynomials g_{00}, g_{01}, g_{10} and g_{11} with $f_{m(i')} = f_{m(i)}g_{00} + f_i g_{01}$ and $f_{i'} = f_{m(i)}g_{10} + f_i g_{11}$.

```

1: if  $i' - i > 1$  then
2:   Call Massey( $i, \frac{i+i'}{2}$ ) to get polynomials  $g'_{00}, g'_{01}, g'_{10}$  and  $g'_{11}$ 
3:    $D_{m(\frac{i+i'}{2})} := D_i g'_{00} + D_{m(i)} g'_{01}$  {Compute just the coefficients needed later!}
4:    $D_{\frac{i+i'}{2}} := D_i g'_{10} + D_{m(i)} g'_{11}$  {Compute just the coefficients needed later!}
5:   Call Massey( $\frac{i+i'}{2}, i'$ ) to get polynomials  $g''_{00}, g''_{01}, g''_{10}$  and  $g''_{11}$ 
6:   Compute  $g_{kj} = g'_{0j} g''_{k0} + g'_{1j} g''_{k1}$  for  $j, k \in \{0, 1\}$ 
7: else
8:   if  $\text{coeff}(D_i, N - i + L_i) = 1$  then
9:      $g_{00} = g_{11} = 1, g_{01} = g_{10} = 0$ 
10:  else
11:    if  $m(i) - L_{m(i)} > i - L_i$  then
12:       $g_{00} = 1, g_{01} = 0, g_{10} = x^{(m(i)-L_{m(i)})-(i-L_i)}, g_{11} = 1$ 
13:    else
14:       $g_{00} = 0, g_{01} = 1, g_{10} = 1, g_{11} = x^{(i-L_i)-(m(i)-L_{m(i)})}$ 
15:    end if
16:  end if
17: end if

```

With different multiplication algorithms such as, for example, Karatsuba (see also Sect. 11.3), we can get $T(n) = O(n^{1.59})$ with a very good constant. Fortunately there are plenty of good libraries which implement fast polynomial arithmetic, and the designers of the libraries have performed timings to choose the best multiplication algorithm for each range of n . So we can just implement Algorithm 2.2 and be sure that the libraries will guarantee an asymptotic running time of $O(n \log^2 n \log \log n)$ and select special algorithms with good constants for small n . For the timings at the end of this section we choose NTL [244] as the underling library for polynomial arithmetic.

We prove by induction that the polynomials g_{00}, g_{01}, g_{10} and g_{11} have the desired property.

Theorem 2.11 Algorithm 2.2 computes polynomials g_{00}, g_{01}, g_{10} and g_{11} with $f_{m(i')} = f_{m(i)}g_{00} + f_i g_{01}$ and $f_{i'} = f_{m(i)}g_{10} + f_i g_{11}$.

Proof We prove the theorem by induction on $d = i' - i$.

For $d = 1$ lines 8–16 are just a reformulation of lines 8–17 of Algorithm 2.1. Note that we flipped the sequence A_i and thus we have to investigate the bit at position $N - i + L_i$ instead of the bit at position i .

Now we have to check the case $d > 1$. By induction the call of $\text{Massey}(i, \frac{i+i'}{2})$ gives us polynomials $g'_{00}, g'_{01}, g'_{10}$ and g'_{11} with $f_{m(\frac{i+i'}{2})} = f_{m(i)}g'_{00} + f_i g'_{01}$ and $f_{\frac{i+i'}{2}} = f_{m(i)}g'_{10} + f_i g'_{11}$.

To prove that the call of $\text{Massey}(\frac{i+i'}{2}, i')$ in line 5 gives the correct result, we have to show that the values $D_{m(\frac{i+i'}{2})}$ and $D_{\frac{i+i'}{2}}$ computed in lines 3 and 4 satisfy the requirements of the algorithm.

In line 3 we compute $D_{m(\frac{i+i'}{2})}$ as

$$\begin{aligned} D_{m(\frac{i+i'}{2})} &= D_i g_{10} + D_{m(i)} g_{11} \\ &= A^* f_i g_{01} + A^* f_{m(i)} g_{01} \quad (\text{required input form}) \\ &= A^* f_{m(\frac{i+i'}{2})} \quad (\text{induction}) \end{aligned}$$

and similarly $D_{\frac{i+i'}{2}} = A^* f_{\frac{i+i'}{2}}$. Thus we meet the requirement for $\text{Massey}(\frac{i+i'}{2}, i')$ and by induction we obtain polynomials $g''_{00}, g''_{01}, g''_{10}$ and g''_{11} with $f_{m(i')} = f_{m(\frac{i+i'}{2})}g''_{00} + f_{\frac{i+i'}{2}}g''_{01}$ and $f_{i'} = f_{m(\frac{i+i'}{2})}g''_{10} + f_{\frac{i+i'}{2}}g''_{11}$.

Thus

$$\begin{aligned} f_{m(i')} &= f_{m(\frac{i+i'}{2})}g''_{00} + f_{\frac{i+i'}{2}}g''_{01} \\ &= (f_{m(i)}g'_{00} + f_i g'_{01})g''_{00} + (f_{m(i)}g'_{10} + f_i g'_{11})g''_{01}. \end{aligned}$$

This proves $g_{00} = g'_{00}g''_{00} + g'_{10}g''_{01}$ and so on, i.e. the polynomials computed in line 6 satisfy the statement of the theorem. \square

In order to do the test in line 11 of Algorithm 2.2 we need to know $m(i)$, L_i and $L_{m(i)}$. We assume that the algorithm receives these numbers as input and we must prove that we can rapidly compute $m(i')$, $L_{i'}$ and $L_{m(i')}$.

In Algorithm 2.2 we have at any time $f_{i'} = f_{m(i)}g_{10} + f_i g_{11}$. We must compute $L_{i'} = \deg f_{i'}$. The problem is that we cannot compute $f_{i'}$ for all i' , since it will take $O(n^2)$ steps just to write the results. However, we don't have to compute $f_{i'}$ to determine $\deg f_{i'}$, as the following theorem shows.

Theorem 2.12 *Using the notation of Algorithm 2.2*

$$\deg f_{i'} = \deg f_i + \deg g_{11}$$

and if $m(i') > m(i)$ then

$$\deg f_{m(i')} = \deg f_i + \deg g_{01}.$$

Proof It is enough to prove $\deg f_{i'} = \deg f_i + \deg g_{11}$, since the second part follows from the first simply by changing i to $m(i)$.

We will prove $\deg(f_{m(i)}g_{10}) < \deg(f_i g_{11})$, which implies the theorem. The proof is by induction on $d = i' - i$. This time we do not go directly from d to $2d$, but instead we will go from d to $d + 1$.

For $d = 1$ the only critical part is line 12. In all other cases we have $\deg f_i > \deg f_{m(i)}$ and $\deg g_{11} > \deg g_{10}$.

However, if we use line 12, then $\deg f_{i+1} = \deg f_i$, since $m(i) - L_{m(i)} > i - L_i$ and hence $\deg f_{i+1} = \deg f_i + \deg g_{11}$.

Now look at the step d to $d + 1$ as described in the algorithm. Let $g'_{00}, g'_{01}, g'_{10}$ and g'_{11} be the polynomials with $f_{m(i+d)} = f_{m(i)}g'_{00} + f_i g'_{01}$ and $f_{i+d} = f_{m(i)}g'_{10} + f_i g'_{11}$. By induction, $\deg f_{i+d} = \deg(f_i g'_{11}) > \deg f_{m(i)}g'_{10}$. Now observe how f_{i+d+1} is related to f_i . If we use line 9 in Algorithm 2.1 then $f_{i+d+1} = f_{i+d}$ and there is nothing left to prove.

If we use line 13 in Algorithm 2.1, then $\deg f_{i+d+1} = \deg f_{i+d}$ (since $m - L_m > i - L_i$) and hence $g'_1 = 1$ and $\deg f_{i+d+1} = \deg(f_{i+d}g'_1) = \deg(f_i g'_1 g'_{11}) = \deg f_i + \deg g_{11}$.

If we are in the case of line 15 in Algorithm 2.1 then $\deg g'_1 > \deg g'_0 = 0$ and hence $\deg f_i g'_1 > \deg f_{m(i)}g'_0$ and thus $\deg f_{i+d+1} = \deg(f_{i+d}g'_1) = \deg f_i + \deg g_{11}$.

This proves the theorem. \square

Theorem 2.12 allows us to compute $L_{i'} = \deg f_{i'}$ and $L_{m(i')} = \deg f_{m(i')}$ with just four additions from L_i and $L_{m(i)}$. Since $\deg f_i \leq i \leq n$ the involved numbers have at most $\log(n)$ bits, i.e. we need just $O(\log n)$ extra time steps per call of $\text{Massey}(i, i')$.

The last thing we have to explain is how the algorithm computes the function $m(i)$. At the beginning $m(0) = -1$ is known. If $i' - i = d = 1$ we obtain $m(i + 1)$ as follows: If we use line 9 or line 12 then $m(i + 1) = m(i)$, and if we use line 14 then $m(i + 1) = i$.

If $d > 1$ then we obtain the value $m(\frac{i+i'}{2})$ needed for the call of $\text{Massey}(\frac{i+i'}{2}, i')$ resulting from the call of $\text{Massey}(i, \frac{i+i'}{2})$.

Similarly the algorithm can recursively compute $L_i = \deg f_i$ and $L_{m(i)} = \deg f_{m(i)}$, which is even faster than using Theorem 2.12.

Finally, we show that we can trim the Polynomials D_i and $D_{m(i)}$. We used these to get the sub-quadratic bound for the running time.

Theorem 2.13 *The function $\text{Massey}(i, i')$ needs only the coefficients between $z^{N-i'+L_m+2}$ and $z^{\max\{L_i-i, L_{m(i)}-m(i)\}}$ (inclusive) from the polynomials D_i and $D_{m(i)}$.*

Proof In the final recursion steps the algorithm will have to access $\text{coeff}(D_j, N - j + L_j)$ for each j in $\{i, \dots, i' - 1\}$. So we have to determine which parts of the polynomials D_i and $D_{m(i)}$ are needed to compute the relevant coefficients.

Let $g_0^{(j)}$ and $g_1^{(j)}$ be the polynomials with $D_j = D_{m(i)}g_0^{(j)} + D_i g_1^{(j)}$.

By Theorem 2.12 we know that $\deg f_i + \deg g_1^{(j)} = \deg_{f_j}$ and $\deg f_m + \deg g_0^{(j)} < \deg_{f_j}$.

Thus $\max\{\deg g_0^{(j)}, \deg g_1^{(j)}\} \leq L_j - L_m - 1$. Therefore we need only the coefficients from $z^{(N-j+L_j)-(L_j-L_m-1)} = z^{N-j+L_m+1}$ to z^{N-j+L_j} of D_i and D_m to compute $\text{coeff}(D_j, N-j+L_j)$. (Note that in the algorithm we compute D_j not directly, but in intermediate steps so as not to change the fact that $\text{coeff}(D_j, N-j+L_j)$ is affected only by this part of the input.)

Since $j < i'$ we see that we need no coefficient below $z^{N-i'+L_m+2}$.

To get an upper bound we have to bound $L_j - j$ in terms of L_i , L_m and i . By induction on j we prove

$$\max\{L_j - j, L_{j(m)} - m(j)\} \leq \max\{L_i - i, L_{m(i)} - m(i)\}.$$

For $j = i$ this is trivial. Now we prove

$$\max\{L_{j+1} - (j+1), L_{m(j+1)} - m(j+1)\} \leq \max\{L_j - j, L_{m(j)} - m(j)\}.$$

To this end, we study what happens in lines 9, 13 and 15 of Algorithm 2.1.

In the first two cases we have $L_{j+1} = L_j$, $m(j+1) = m(j)$ and the inequality is trivial. In the last case $L_{j+1} = L_j + (j - L_j) - (m(j) - L_{m(j)})$ and $m(j+1) = j$ thus

$$\max\{L_{j+1} - (j+1), L_{m(j+1)} - m(j+1)\} = \max\{L_{m(j)} - m(j) - 1, L_j - j\}.$$

This proves $L_j - j < \max\{L_i - i, L_{m(i)} - m(i)\}$, which gives the desired upper bound $N - j + L_j$. \square

We have seen that the algorithm keeps track of the values $m(i)$, L_i and $L_{m(i)}$. So the only thing we have to do to get the full sequence L_1, \dots, L_n is to output $L_{\frac{i+i'}{2}}$ when the algorithm reaches line 5. This costs no extra time.

If we are interested in the feedback polynomials we have to do more work. We need an array R of size n . Each time the algorithm reaches its end (line 17) we store the values $(i, g_{00}, g_{01}, g_{10}, g_{11})$ at $R[i']$.

When the computation of the linear complexity is finished the array R is completely filled. Now we can compute the feedback polynomials by the following recursive function (Algorithm 2.3).

Finally, we can also use the algorithm in an iterative way. If we have observed N bits, we can call $\text{Massey}(0, N)$ to compute the linear complexity of the sequence a_0, \dots, a_{N-1} . We will remember $m(N)$ (computed by the algorithm), $f_N = g_{10} + g_{11}$ and $f_{m(N)} = g_{00} + g_{01}$. If we later observe N' extra bits of the sequence, we can call $\text{Massey}(N, N + N')$ to get the linear complexity of $a_0, \dots, a_{N+N'-1}$.

In the extreme case we can always stop after one extra bit. In this case the algorithm will of course need quadratic time, since it must compute all intermediate feedback polynomials. Computer experiments show that the new algorithm beats

Algorithm 2.3 feedback(i')

```

1: Get the values  $i, g_{00}, g_{01}, g_{10}, g_{11}$  from  $R[i']$ .
2: if  $i = 1$  then
3:    $f_{i'} = g_{10} + g_{11}, f_{m(i')} = g_{00} + g_{01}$ 
4: else
5:   Obtain  $f_i$  and  $f_{m(i')}$  by calling feedback( $i$ ).
6:    $f_{i'} = f_{m(i)}g_{10} + f_i g_{11}, f_{m(i')} = f_{m(i)}g_{00} + f_i g_{01}$ 
7: end if

```

Table 2.1 Tests for the algorithms

	n					
	100	500	1000	5000	10000	100000
Algorithm 2.1	0.00002	0.00065	0.0036	0.098	0.39	36.11
Algorithm 2.2	0.001	0.00157	0.0042	0.049	0.094	0.94

the original Berlekamp-Massey algorithm if it receives at least 5000 bits at once. The full speed is reached only if we receive the input in one step.

Table 2.1 shows that the asymptotic fast Berlekamp-Massey algorithm beats the classical variant even for very small n .

2.4.4 Linear Complexity of Random Sequences

Since we want to use linear complexity as a measure of the randomness of a sequence, it is natural to ask what the expected linear complexity of a random sequence is.

Theorem 2.14 (Rueppel [228]) *Let $1 \leq L \leq n$. The number $N(n, L)$ of binary sequences of length n having linear complexity exactly L is $N(n, L) = 2^{\min\{2n-2L, 2L-1\}}$.*

Proof We are going to find a recursion for $N(n, L)$. For $n = 1$ we have $N(1, 0) = 1$ (the sequence 0) and $N(1, 1) = 1$ (the sequence 1).

Now consider a sequence a_0, \dots, a_{n-1} of length n and linear complexity L . Let $f(z)$ be a feedback polynomial of a minimal LFSR generating a_0, \dots, a_{n-1} .

We have one way to extend the sequence a_0, \dots, a_{n-1} by an a_n without changing the feedback polynomial $f(z)$. (This already proves $N(n+1, L) \geq N(n, L)$.)

Now consider the second possible extension $a_0, \dots, a_{n-1}, \overline{a_n}$. By Corollary 2.3 we have either

$$\mathcal{L}(a_0, \dots, a_{n-1}, \overline{a_n}) = \mathcal{L}(a_0, \dots, a_{n-1}) = L$$

or

$$\mathcal{L}(a_0, \dots, a_{n-1}, \overline{a_n}) = n + 1 - L > L.$$

If $L \geq \frac{n+1}{2}$, the second case is impossible, i.e. we have $N(n+1, L) \geq 2N(n, L)$ for $L \geq \frac{n+1}{2}$.

Now let $L < \frac{n+1}{2}$ and let m be the largest index with $L_m = \mathcal{L}(a_0, \dots, a_{m-1}) < L$. Then by Corollary 2.3 we find

$$L_m = m - \mathcal{L}(a_0, \dots, a_{m-2}) > \mathcal{L}(a_0, \dots, a_{m-2})$$

and hence $L_m > m/2$. Therefore $n - L > \frac{n+1}{2} > m/2 > m - L_m$, which means by Algorithm 2.1 that

$$\mathcal{L}(a_0, \dots, a_{n-1}, \overline{a_n}) > \mathcal{L}(a_0, \dots, a_{n-1}) = L.$$

This proves the recursion

$$N(n+1, L) = \begin{cases} 2N(n, L) + N(n, n+1-L) & \text{if } 2L > n+1, \\ 2N(n, L) & \text{if } 2L = n+1, \\ N(n, L) & \text{if } 2L < n+1. \end{cases}$$

With this recursion it is just a simple induction to prove

$$N(n, L) = 2^{\min\{2n-2L, 2L-1\}}.$$

□

With Theorem 2.14 we need only elementary calculations to obtain the expected linear complexity of a finite binary sequence.

Theorem 2.15 (Rueppel [228]) *The expected linear complexity of a binary sequence of length n is*

$$\frac{n}{2} + \frac{4 + (n \& 1)}{18} - 2^{-n} \left(\frac{n}{3} + \frac{2}{9} \right).$$

Proof For even n we get

$$\begin{aligned} \sum_{i=1}^n i N(n, i) &= \sum_{i=1}^{n/2} i \cdot 2^{2i-1} + \sum_{i=0}^{n/2-1} (n-i) 2^{2i} \\ &= \left[n \frac{2^n}{3} + \frac{2^{n+1}}{9} + \frac{2}{9} \right] + \left[n \frac{2^n}{6} + \frac{2^{n+2}}{9} - \frac{n}{3} - \frac{4}{9} \right] \\ &= n 2^{n-1} + \frac{2^{n+1}}{9} - \frac{n}{3} - \frac{2}{9} \end{aligned}$$

and similarly for odd n we get

$$\begin{aligned} \sum_{i=1}^n i N(n, l) &= \sum_{i=1}^{(n-1)/2} i \cdot 2^{2i-1} + \sum_{i=0}^{(n-1)/2} (n-i) 2^{2i} \\ &= n 2^{n-1} + \frac{5}{18} 2^n - \frac{n}{3} - \frac{2}{9}. \end{aligned}$$

Multiplying by the probability 2^{-n} for a sequence of length n we get the expected value. \square

It is also possible to determine the expected linear complexity of a periodic sequence of period n with random elements. However, since in cryptography a cipher stream is broken if we are able to observe more than one period (see the Vigenère cipher in Sect. 1.1), this kind of result is of less interest. We state the following without proof.

Theorem 2.16 *A random periodic sequence with period n has expected linear complexity:*

- (a) $n - 1 + 2^{-n}$ if n is power of 2;
- (b) $(n - 1)(1 - \frac{1}{2^{o(2,n)}}) + \frac{1}{2}$ if n is an odd prime and $o(2, n)$ is the order of 2 in \mathbb{F}_n^\times .

Proof

- (a) See Proposition 4.6 in [228].
- (b) See Theorem 3.2 in [186].

\square

2.5 The Linear Complexity Profile of Pseudo-random Sequences

2.5.1 Basic Properties

We have introduced linear complexity as a measure of the cryptographic strength of a pseudo-random sequence. However, a high linear complexity is only a necessary but not sufficient condition for cryptographic strength. Take for example the sequence

1010111100010011010111100

which is generated by an LFSR with feedback polynomial $z^4 + z + 1$. It is a weak key stream and its linear complexity is 4. By changing just the last bit of the sequence the linear complexity rises to 22 (see Corollary 2.3), but changing just one bit does not make a keystream secure.

One way to improve linear complexity as a measure of the randomness of a sequence is to look at the linear complexity profile.

Fig. 2.5 The linear complexity profile of 1010111100010011010111101

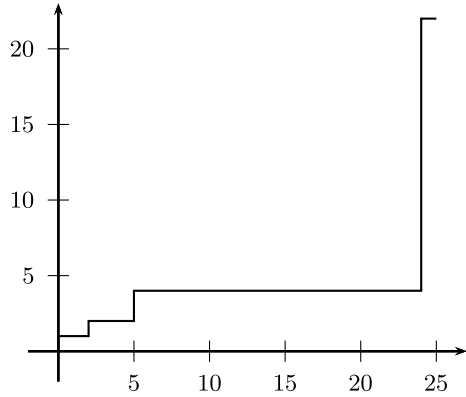
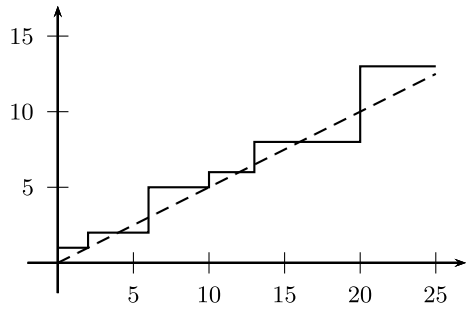


Fig. 2.6 A typical linear complexity profile



Definition 2.13 The linear *complexity profile* of a binary sequence $(a_n)_{n \in \mathbb{N}}$ is the function $LP : \mathbb{N}^+ \rightarrow \mathbb{N}$ with $n \mapsto \mathcal{L}(a_0, \dots, a_{n-1})$.

If we draw the linear complexity profile for the sequence

1010111100010011010111101

we see (Fig. 2.5) that the linear complexity jumps with the last bit to the high value 22. Prior to this we have the low value 4, which means that the sequence is a weak key stream.

By Theorem 2.15 the expected linear complexity of a sequence of length n is about $\frac{n}{2}$, i.e. the linear complexity profile of a good pseudo-random sequence should lie around the line $n \mapsto n/2$ as shown in Fig. 2.6.

In the remaining part of this section we will study sequences with a linear complexity profile which is “as good as possible”.

Definition 2.14 A sequence $(a_n)_{n \in \mathbb{N}}$ has a *perfect linear complexity profile* if

$$\mathcal{L}(a_0, \dots, a_{n-1}) = \left\lfloor \frac{n+1}{2} \right\rfloor.$$

The linear complexity profile is *good* if

$$\left| \mathcal{L}(a_0, \dots, a_{n-1}) - \frac{n}{2} \right| = O(\log(n)).$$

H. Niederreiter [198] classified all sequences with a good linear complexity profile by means of continued fractions. We will follow his proof in the remaining part of the section.

2.5.2 Continued Fractions

In this section we classify all sequences with a good linear complexity profile. To that end we establish a connection between the continued fraction expansion of the generation function and the complexity profile.

Consider the field $F((z^{-1})) = \{\sum_{i \geq n} a_i z^{-i} \mid n \in \mathbb{Z}, a_i \in F\}$ of formal Laurent series in z^{-1} . For $S = \sum_{i \geq n} a_i z^{-i} \in F((z^{-1}))$ we denote by $[S] = \sum_{0 \leq i \leq n} a_i z^{-i}$ the polynomial part of S .

A continued fraction is an expression of the form

$$b_0 + \frac{a_1}{b_1 + \frac{a_2}{b_2 + \frac{a_3}{b_3 + \ddots}}}$$

For continued fractions we use the compact notation of Pringsheim:

$$b_0 + \frac{a_1 |}{|b_1} + \frac{a_2 |}{|b_2} + \frac{a_3 |}{|b_3} + \dots$$

For $S \in F((z^{-1}))$ recursively define

$$A_i = [R_{i-1}^{-1}], \quad R_i = R_{i-1}^{-1} - A_i \quad \text{for } i \geq 0 \quad (2.22)$$

with $R_{-1} = S_0$. This gives the continued fraction expansion

$$S = A_0 + \frac{1 |}{|A_1} + \frac{1 |}{|A_2} + \dots \quad (2.23)$$

of S .

The term

$$A_0 + \frac{1 |}{|A_1} + \dots + \frac{1 |}{|A_k} = \frac{P_k}{Q_k}$$

with $P_k, Q_k \in F[z]$ is called the k th *convergent fraction* of f .

Let us recall some basic properties of continued fractions (see, for example, [211]).

The polynomials P_k, Q_k satisfy the recursion

$$6P_{-1} = 1, \quad P_0 = A_0, \quad P_k = A_k P_{k-1} + P_{k-2}, \quad (2.24)$$

$$Q_{-1} = 0, \quad Q_0 = 1, \quad Q_k = A_k Q_{k-1} + Q_{k-2}. \quad (2.25)$$

In addition we have the identities

$$P_{k-1} Q_k - P_k Q_{k-1} = (-1)^k, \quad (2.26)$$

$$\gcd(P_k, Q_k) = 1, \quad (2.27)$$

$$S = \frac{P_k + R_k P_{k-1}}{Q_k + R_k Q_{k-1}}. \quad (2.28)$$

The above identities hold for every continued fraction. The next identities use the degree function and hold only for continued fractions defined over $F((z^{-1}))$. Using the recursion (2.25) we get $\deg Q_i = \sum_{j=1}^i \deg A_j$ for $j \geq 1$.

Lemma 2.2 *For all $j \in \mathbb{N}$ we have*

$$\deg(Q_j S - P_j) = -\deg(Q_{j+1}).$$

Proof We prove this by induction on j . For $j = 0$ this follows immediately from Eq. (2.28) with

$$-\deg R_0 = \deg R_0^{-1} = \deg[R_0^{-1}] = \deg A_1 = \deg Q_1.$$

Now let $j \geq 1$. By Eq. (2.28) we have

$$SQ_j - P_j = B_j(SQ_{j-1} - P_{j-1}).$$

By induction $\deg SQ_{j-1} - P_{j-1} = \deg Q_j$ and since $\deg B_j = -\deg A_{j+1}$ and $\deg Q_{j+1} = \deg A_{j+1} + \deg Q_j$ we get

$$\deg(SQ_j - P_j) = -\deg Q_{j+1}. \quad \square$$

The connection of linear complexity and the Berlekamp-Massey algorithm with continued fractions and the Euclidean algorithm has been observed several times. The formulation of the following theorem is from [197].

Theorem 2.17 *Let $(a_n)_{n \in \mathbb{N}}$ be a sequence over the field F and let $S(z) = \sum_{j=0}^{-\infty} a_j z^{-j-1}$. Let $\frac{P_k}{Q_k}$ be the k th convergent fraction of the continued fraction expansion of S .*

Then for every $n \in \mathbb{N}^+$ the linear complexity $L_n = \mathcal{L}(a_0, \dots, a_{n-1})$ is given by $L_n = 0$ for $n < \deg Q_0$ and $L_n = \deg Q_j$ where $j \in \mathbb{N}$ is determined by

$$\deg Q_{j-1} + \deg Q_j \leq n < \deg Q_j + \deg Q_{j+1}.$$

Proof By Lemma 2.2 we have

$$\deg\left(S - \frac{P_j}{Q_j}\right) = -\deg Q_j - \deg Q_{j+1}.$$

This means that the first $\deg Q_j + \deg Q_{j+1}$ elements of the sequence with the rational generating function $\frac{P_j}{Q_j}$ coincide with $(a_n)_{n=0, \dots, \deg Q_j + \deg Q_{j+1} - 1}$.

But the rational generating function $\frac{P_j}{Q_j}$ belongs to an LFSR with feedback polynomial Q_j^* , which proves that

$$L_n \leq \deg Q_j \quad \text{for } n < \deg Q_j + \deg Q_{j+1}. \quad (2.29)$$

This already establishes one part of the theorem. Now we prove the equality. That $L_n = 0$ if $n < \deg Q_0$ is just a reformulation of the fact that $\deg Q_0$ denotes the number of leading zeros in the sequence $(a_n)_{n \in \mathbb{N}}$.

By induction we know now that $L_n = \deg Q_j$ for $\deg Q_{j-1} + \deg Q_j \leq n < \deg Q_j + \deg Q_{j+1}$.

If k is the smallest integer with $L_k > \deg Q_j$ then by Corollary 2.3 we have $L_k = k - \deg Q_j$. The only possible value of k for which L_k satisfies Eq. (2.29) is $k = \deg Q_j + \deg Q_{j+1}$. Thus $k = \deg Q_j + \deg Q_{j+1}$ and $L_k = \deg Q_{j+1}$. By Eq. (2.29) we get $L_n = \deg Q_{j+1}$ for $\deg Q_j + \deg Q_{j+1} \leq n < \deg Q_{j+1} + \deg Q_{j+2}$, which finishes the induction. \square

2.5.3 Classification of Sequences with a Perfect Linear Complexity Profile

By Theorem 2.17 it is easy to characterize sequences with a good linear complexity profile in terms of continued fractions (see [197, 198]). As a representative of all results of this kind, we present Theorem 2.18 which treats the case of a perfect linear complexity profile.

Theorem 2.18 (see [197]) *A sequence $(a_n)_{n \in \mathbb{N}}$ has a perfect linear complexity profile if and only if the generating function $S(z) = \sum_{j=0}^{-\infty} a_j z^{-j-1}$ is irrational and has a continued fraction expansion*

$$S = \frac{1}{|A_1|} + \frac{1}{|A_2|} + \frac{1}{|A_3|} + \dots$$

with $\deg A_i = 1$ for all $i \geq 1$.

Proof A perfect linear complexity profile requires that the linear complexity grows at most by 1 at each step. By Theorem 2.17 this means that the sequence $\deg Q_i$ contains all positive integers, i.e. $\deg A_i = 1$ for all continued fraction denominators A_i .

On the other hand, $\deg A_i = 1$ for all i implies $\deg Q_i = i$ and by Theorem 2.17 we get $L_i = \lfloor \frac{i+1}{2} \rfloor$. \square

By Theorem 2.18 we can deduce a nice characterization of sequences with a perfect linear complexity profile.

Theorem 2.19 (see [274]) *The binary sequence $(a_i)_{i \in \mathbb{N}}$ has a perfect linear complexity profile if and only if it satisfies $a_0 = 1$ and $a_{2i} = a_{2i-1} + a_i$ for all $i \geq 1$.*

Proof Define the operation $D : \mathbb{F}_2((z^{-1})) \rightarrow \mathbb{F}_2((z^{-1}))$ by

$$D : T \mapsto z^{-1}T^2 + (1 + z^{-1})T + z^{-1}.$$

A short calculation reveals the following identities:

$$D(T + U + V) = D(T) + D(U) + D(V) \quad \text{for } T, U, V \in \mathbb{F}_2((z^{-1})),$$

$$D(T^{-1}) = D(T)T^{-2} \quad \text{for } T \in \mathbb{F}_2((z^{-1})),$$

$$D(z) + D(c) = c + 1 \quad \text{for } c \in \mathbb{F}_2.$$

Now assume that the sequence $(a_i)_{i \in \mathbb{N}}$ has a perfect linear complexity profile and let

$$S(z) = \sum_{j=0}^{\infty} a_j z^{-j-1} = \frac{1}{|A_1|} + \frac{1}{|A_2|} + \frac{1}{|A_3|} + \dots$$

be the corresponding generating function with its continued fraction expansion.

By Theorem 2.18 we have $A_i = z + a_i$ with $a_i \in \mathbb{F}_2$. By Eq. (2.22) we have $R_i^{-1} = R_{i-1} - A_i$ and hence

$$D(R_{i-1})R_{i-1}^{-2} = D(R_{i-1}^{-1}) = D(R_i + z + a_i) = D(R_i) + 1 + a_i.$$

By definition, $S = R_{-1}$, and by induction on i we have

$$D(S) = \sum_{j=0}^{i-1} (a_j + 1) \prod_{k=-1}^{j-1} R_k^2 + \prod_{k=-1}^{i-1} R_k^2 D(R_i).$$

We can turn $\mathbb{F}_2((z^{-1}))$ into a metric space by defining $d(Q, R) = 2^{-\deg(Q-R)}$. Since $\deg R_i < 0$ for all i we get

$$\lim_{i \rightarrow \infty} \prod_{k=-1}^{i-1} R_k^2 D(R_i) = 0$$

and hence

$$D(S) = \sum_{j=0}^{\infty} (a_j + 1) \prod_{k=-1}^{j-1} R_k^2.$$

Since all summands lie in $\mathbb{F}_2((z^{-2}))$, we get $D(S) = U^2$ for some $U \in \mathbb{F}_2((z^{-1}))$ or equivalently

$$S^2 + (z + 1)S + 1 = zU^2. \quad (2.30)$$

Comparing the coefficients of z^0 we get $a_0 = 1$, and comparing the coefficients of z^{2i} ($i \in \mathbb{N}^+$) we get $a_i + a_{2i-1} + a_{2i} = 0$.

For the opposite direction note that the recursion $a_0 = 1$ and $a_i + a_{2i-1} + a_{2i} = 0$ imply that Eq. (2.30) is satisfied for some suitable $U \in \mathbb{F}_2((z^{-1}))$.

Assume that the linear complexity profile of the sequence is not perfect.

Then we find an index j with $\deg A_j > 1$ and by Lemma 2.2 we have

$$\deg(SQ_j - P_j) = -\deg Q_{j+1} < -\deg Q_j - 1.$$

It follows that

$$\begin{aligned} & \deg(P_j^2 + (x + 1)P_jQ_j + Q_j^2 + xU^2) \\ &= \deg(Q^2S^2 - P_j^2 + (x + 1)Q_j(SQ_j - P_j)) \\ &\leq \max\{\deg(Q^2S^2 - P_j^2), \deg((x + 1)Q_j(SQ_j - P_j))\} \\ &< 0. \end{aligned} \quad (2.31)$$

In particular the constant term $P_j(0)^2 + P_j(0)Q_j(0) + Q_j(0)^2$ is 0, but this implies $P_j(0) = Q_j(0) = 0$ and hence $\gcd(P_j, Q_j) \neq 1$, contradicting Eq. (2.27).

This proves that the sequence that satisfies the recurrence given in Theorem 2.19 has a perfect linear complexity profile. \square

We remark that even a sequence with a perfect linear complexity profile can be highly regular. For example, consider the sequence $(a_i)_{i \in \mathbb{N}}$ with $a_0 = 1$, $a_{2^j} = 1$ for $j \in \mathbb{N}$ and $a_j = 0$ for $j \in \mathbb{N} \setminus \{1, 2^k \mid k \in \mathbb{N}\}$ given by Dai in [71]. This sequence is obviously a very weak key stream, but as one can check by Theorem 2.19, it has an optimal linear complexity profile.

2.6 Implementation of LFSRs

This book is about the mathematics of stream ciphers, but in cryptography mathematics is not everything. We have mentioned in the introduction that LFSRs are popular because they can be implemented very efficiently. This section should justify this claim.

Fig. 2.7 The Fibonacci implementation of an LFSR

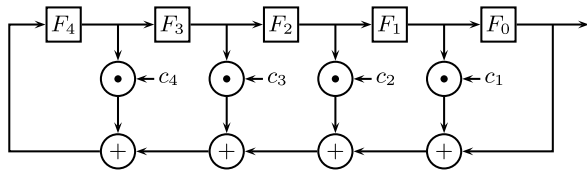
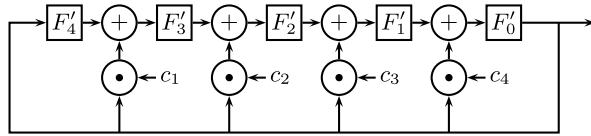


Fig. 2.8 The Galois implementation of an LFSR



2.6.1 Hardware Realization of LFSRs

For implementing LFSRs in hardware there are two basic strategies. Either we convert Fig. 2.1 directly into the hardware, which then looks like Fig. 2.7. If the feedback polynomial is fixed we can save the AND-gates (multiplication).

This way to implement an LFSR is called the *Fibonacci implementation* or sometimes the *simple shift register generator* (SSRG).

The alternative implementation (see Fig. 2.8) is called the *Galois implementation* (alternative names: *multiple-return shift register generator* (MRSRG) or *modular shift register generator* (MSRG)).

The advantage of the Galois implementation is that every signal must pass through at most one XOR-gate. By contrast, with a dense feedback polynomial the feedback signal in the Fibonacci implementation must pass through approximately $n/2$ XOR-gates.

As indicated in the figure, one has to reverse the feedback coefficients in the Galois implementation. The internal states of the Fibonacci implementation and the Galois implementation are connected by Theorem 2.20.

Theorem 2.20 *The Galois implementation generates the same sequence as the Fibonacci implementation if it is initialized with $F'_i = \sum_{j=0}^i F_{i-j}c_{n-j}$, ($0 \leq i \leq n$) where F_0, \dots, F_{n-1} is the initial state of the Fibonacci implementation of the LFSR.*

Proof We have $F'_0 = F_0$, so the next output bit is the same in both implementations.

We must prove that the next state $\hat{F}_{n-1}, \dots, \hat{F}_0$ of the Fibonacci implementation and the next state $\hat{F}'_{n-1}, \dots, \hat{F}'_0$ of the Galois implementation again satisfy $\hat{F}'_i = \sum_{j=0}^i \hat{F}_{i-j}c_{n-j}$, ($0 \leq i \leq n$).

For $i \leq n - 2$ we get

$$\begin{aligned}
 \hat{F}'_i &= F'_{i+1} + c_{n-1-i} F'_0 \\
 &= \sum_{j=0}^{i+1} F_{i+1-j} c_{n-j} + c_{n-1-i} F'_0 \\
 &= \sum_{j=0}^i F_{i+1-j} c_{n-j} \\
 &= \sum_{j=0}^i \hat{F}_{i-j} c_{n-j}.
 \end{aligned}$$

For $i = n - 1$ we have

$$\begin{aligned}
 \hat{F}'_{n-1} &= F'_0 = F_0 \\
 &= \sum_{i=0}^{n-1} c_i F_i + \sum_{i=1}^{n-1} c_i F_i \\
 &= \hat{F}_{n-1} + \sum_{i=1}^{n-1} c_i \hat{F}_{i-1}.
 \end{aligned}$$

So both implementations give the same result. □

2.6.2 Software Realization of LFSRs

Now we look at software implementation of LFSRs. All modern processors have instructions that help to achieve fast implementations. The main problem with software implementations is that we lose the advantage of low power consumption that we have with specialized hardware. Also, block ciphers with implementations based on table look-up become a good alternative.

2.6.2.1 Bit-Oriented Implementation

We first describe a bit-oriented implementation. The advantage is that we have a direct simulation of the LFSR in the software. For optimal performance it would be better to use a byte-oriented implementation.

We use an array of words $w_0, \dots, w_{\hat{n}}$ which represent the internal state of the shift register. We can easily implement the shift operation on this bit field by calling the shift and rotation instructions of our processor (see Algorithm 2.4

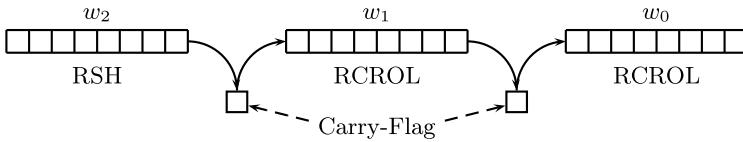


Fig. 2.9 Right shift over several words

and Fig. 2.9). Unfortunately, even a language like C does not provide direct access to the rotation operations. So we have to use hand-written assembler code. A portable C-implementation of the shift operation is a bit slower. Good code can be found in [255], which contains many tips about implementing cryptographic functions.

Algorithm 2.4 Right shift over several words

```

1: RSH  $w_{\hat{n}-1}$  {Right shift by 1}
2: for  $k \leftarrow \hat{n} - 1$  to  $\hat{n} - 1$  do
3:   RCROL  $w_k$  {Right roll by 1, use the carry flag}
4: end for

```

If the feedback polynomial has only a few non-zero coefficients we can compute the feedback value by $f = x[f_1] + \dots + x[f_k]$. However, if the feedback polynomial has many non-zero coefficients, this method is too slow. A better technique is to store the feedback polynomial in the bit field f . We compute $x \& f$ and count the number of non-zero bits in $x \& f$ modulo 2. The operation of counting the numbers of set bits in a bit field is known as *sideway addition* or *population count* (see Sect. 12.1.2). If our computer supports this operation we should use it, otherwise we should use Algorithm 2.5 which directly computes the number of non-zero bits in $x \& f$ modulo 2 (see also Sect. 5.2 in [276]).

Algorithm 2.5 Sideway addition mod 2 (32 bit version)

Ensure: $y = x_0 + \dots + x_{31} \pmod 2$

```

1:  $y \leftarrow x \oplus (x \gg 1)$ 
2:  $y \leftarrow y \oplus (y \gg 2)$ 
3:  $y \leftarrow a(y \& a) \pmod{2^{32}}$  {with  $a = (11111111)_{16}$ }
4:  $y \leftarrow (y \gg 28) \& 1$ 

```

Theorem 2.21 *Given the input $x = (x_0, \dots, x_{n-1})_2$, Algorithm 2.5 computes $x_0 + x_1 + \dots + x_n \pmod 2$.*

Proof After the first line we have $y_0 + y_2 + \dots + y_{62} = x_0 + \dots + x_{63}$ and after the second line we have

$$y_0 + y_4 + \dots + y_{60} = x_0 + \dots + x_{63}.$$

Since $y \& a$ has at most 8 non-zero bits, we can be sure that the multiplication does not overflow, i.e.

$$a(y \& a) = \left(\sum_{i=0}^7 y_{4j}, \sum_{i=0}^6 y_{4j}, \dots, y_0 + y_4, y_0 \right)_4.$$

In the final step we extract the bit $\sum_{i=0}^7 y_{4j} \bmod 2$. □

If we work with 64-bit words, we must add an extra shift $y \leftarrow y \oplus (y \gg 4)$ after line two and use the mask $a = (11111111)_{256}$ instead of $a = (11111111)_{16}$.

Some processors (such as the IA32 family) have a slow multiplication routine (≈ 10 clock cycles, while the shift and XOR takes only 1 clock cycle). In this case Algorithm 2.6, which avoids multiplication, may be faster.

Algorithm 2.6 Sideway addition mod 2 (without multiplication)

Require: $x = (x_{n-1} \dots x_0)$ is n -bit word

Ensure: $y = SADD(x) \bmod 2$

```

1:  $y \leftarrow x$ 
2: for  $k \leftarrow 0$  to  $\lfloor (\log_2(n-1)) \rfloor$  do
3:    $y \leftarrow y \oplus (y \gg 2^k)$ 
4: end for
5:  $y \leftarrow y \& 1 \{y \leftarrow y \bmod 2\}$ 

```

2.6.2.2 Word-Oriented Implementation

The bitwise generation of an LFSR sequence is attractive for simulating hardware realizations. However, on most computers it will be faster to generate the sequence word-wise. Let s be the word size of our computer, i.e. $s = 8$ on a small embedded processor (such as Intel's MCS-51 series) or $s = 32$ or $s = 64$ on a Desktop machine. We assume that s is a power of 2 and that $s = 2^d$.

For simplicity we assume that the length n of the feedback shift register is divisible by the word size s . Let $n = s\hat{n}$.

Let $c_{j,k}$ be the coefficients of the generator matrix associated with the LFSR (see Eq. (2.17)). Define

$$f_i(a_0 + \dots + 2^7 a_7) = \bigoplus_{k=0}^7 \left(\sum_{j=0}^{s-1} 2^j a_k c_{8i+k,n+j} \right). \quad (2.32)$$

Let $x = (x_{n-1}, \dots, x_0)_2$ be the internal state of the LFSR. Then the next word $x' = (x_{n+s}, \dots, x_n)_2$ is

$$\begin{aligned} x' &= \left(\bigoplus_{i=0}^{n-1} x_i c_{i,n+s}, \dots, \bigoplus_{i=0}^{n-1} x_i c_{i,n} \right)_2 \\ &= \bigoplus_{i=0}^{n-1} \sum_{j=0}^{s-1} 2^j x_i c_{i,j}. \end{aligned} \quad (2.33)$$

(This is just the definition of the generator matrix.)

Now write $x = (\hat{x}_{\hat{n}-1}, \dots, \hat{x})$ and regroup the sum in Eq. (2.33), yielding:

$$x' = \bigoplus_{i=0}^{\hat{n}-1} f_i(\hat{x}_i). \quad (2.34)$$

Equation (2.34) gives us a table look-up method to compute the word of the LFSR sequence. We just have to pre-compute the functions f_i and evaluate Eq. (2.33). Algorithm 2.7 shows this in pseudo-code.

Algorithm 2.7 LFSR byte-oriented implementation (table look-ups)

```

1: output  $w_0$ 
2:  $w \leftarrow f_0(w_0)$ 
3: for  $k \leftarrow 1$  to  $\hat{n} - 1$  do
4:    $w \leftarrow w \oplus f_i$ 
5:    $w_{i-1} \leftarrow w_i$ 
6: end for
7:  $w_{\hat{n}-1} \leftarrow w$ 

```

Algorithm 2.7 uses huge look-up tables. This may be a problem in embedded devices. In this case we can use the following algorithm that is based on the idea that we can use byte operations to compute several sideways additions simultaneously and which needs no look-up table.

The core of our program is Algorithm 2.8, which takes 2^k words w_0, \dots, w_{2^k-1} of 2^k bits each and computes the word $y = (y_{2^k-1} \dots y_0)$ with $y_i = \text{SADD}(w_i) \bmod 2$. (SADD denotes the sideways addition.)

Theorem 2.22 *The result $y = \text{PSADD}(d, 0)$ of Algorithm 2.8 satisfies $y_i = \text{SADD}(w_i) \bmod 2$.*

Proof We prove by induction on d' that $y^{(d',k)} = \text{PSADD}(d', k)$ satisfies

$$\bigoplus_{j=0}^{2^{d-d'}} y_{i+j2^d}^{(d',k)} = \text{SADD}(w_{k+i}) \bmod 2 \quad (2.35)$$

Algorithm 2.8 Parallel sideways addition mod 2**Ensure:** $PSADD(d, 0)$ returns the word $(SADD(w_{2^s-1}) \bmod 2, \dots, SADD(w_0) \bmod 2)_2$

```

1: if  $d' = 0$  then
2:   return  $w_i$ 
3: else
4:    $y \leftarrow PSADD(d' - 1, k)$ 
5:    $y \leftarrow (y \ggg 2^{d'-1} \oplus y)$ 
6:    $y' \leftarrow PSADD(d' - 1, k + 2^{d-1})$ 
7:    $y' \leftarrow (y' \lll 2^{d'-1} \oplus y')$ 
8:   return  $(y \& \mu_{d'-1}) \mid (y' \& \overline{\mu_{d'-1}})$ 
9: end if

```

for $i \in \{0, \dots, 2^{d'} - 1\}$.

For $d' = 0$ we have simply $y^{(0,k)} = w_k$ and Eq. (2.35) is trivial.

If $d' > 0$ we call $PSADD(d' - 1, k)$ in line 4 and the return value satisfies Eq. (2.35). The shift and XOR operation in step 4 give us a word $y = y^{(d'-1,k)}$ which satisfies

$$\bigoplus_{j=0}^{2^{d-d'}-1} y_{i+j2^{d'-1}}^{(d'-1,k)} = SADD(w_{k+i}) \bmod 2$$

for $i \in \{0, \dots, 2^{d'-1} - 1\}$. The shift and XOR operation in line 5 computes the sums $y_{i+(2j)2^{d'-1}}^{(d'-1,k)} \oplus y_{i+(2j+1)2^{d'-1}}^{(d'-1,k)}$. Thus after line 5 the word y satisfies

$$\bigoplus_{j=0}^{2^{d-d'}-1} y_{i+j2^{d'}} = SADD(w_{k+i}) \bmod 2$$

for $i \in \{0, \dots, 2^{d'-1} - 1\}$.

Similarly we process the word y' in the lines 6 and 7 and we have

$$\bigoplus_{j=0}^{2^{d-d'}-1} y'_{i+2^{d'-1}+j2^{d'}} = SADD(w_{(k+2^{d-1})+i}) \bmod 2$$

for $i \in \{0, \dots, 2^{d'-1} - 1\}$. (That is, we use a left shift in line 7 instead of a right shift, resulting in the $+2^{d-1}$ term in the index of y' .)

Finally we use in line 8 the mask μ_{d-1} (see Sect. 12.1.1) to select the right bits from words w and w' . \square

A problem is how to find the right input words w_i for Algorithm 2.8. One possibility is to pre-compute feedback polynomials f_0, \dots, f_{s-1} in Algorithm 2.7. This

Algorithm 2.9 LFSR update with parallel sideways addition mod 2**Require:** x is the internal state of the LFSR**Ensure:** y is the next s bits in the LFSR sequence

```

1: for  $i \leftarrow 0$  to  $n - 1$  do
2:    $w_i \leftarrow$  byte-wise XOR of  $(x \gg i) \& f$ 
3: end for
4: Compute  $y$  with  $y_i = \text{SADD}(w_i) \bmod 2$ 
5:  $s \leftarrow 0, z \leftarrow y$ 
6: for  $i \leftarrow 1$  to  $n - 1$  do
7:    $s \leftarrow s \oplus f[n - i + 1] = 1$ 
8:    $y \leftarrow y \oplus (z \gg i)$ 
9: end for

```

Table 2.2 Speed of different LFSR implementations (128 bit LFSR)

Bitwise generation	74.7 Mbits/sec
byte-wise generation (table look-up)	666 Mbits/sec
byte-wise generation (PSADD)	84.4 Mbits/sec

Table 2.3 Speed of an LFSR with feedback polynomial $z^{127} + z + 1$

Generic bitwise generation	74.7 Mbits/sec
trinomial bitwise generation	132 Mbits/sec
Algorithm 2.10	15300 Mbits/sec

needs ns bits in a look-up table, but with a few extra operations we can avoid storing the extra polynomials f_1, \dots, f_{s-1} .

We can use Algorithm 2.8 to compute the next 2^k bits in the LFSR sequence as follows. The internal state of our LFSR is stored in the bit field x .

The idea of Algorithm 2.9 is that $\text{SADD}((X \gg i) \& f) \bmod 2$ is almost x_{n+i} . The only thing that is missing is $x_n f_{n-i+1} \oplus \dots \oplus x_{n+i-1} f_{n-1}$. The loop in lines 6–9 computes this correction term.

All of the above implementations were designed for arbitrary feedback polynomials $f(z) = z^n - \sum_{j=0}^n c_j z^j$. However, if we choose a feedback polynomial with few coefficients, with the additional property that $f(z) - z^n$ has a low degree, we can obtain a very fast implementation. This is especially true if we use a trinomial $f(z) = z^n + z^k + 1$ as a feedback polynomial.

Algorithm 2.10 describes how we can compute the next $n - k$ bits of the LFSR sequence. The internal state of the LFSR is denoted by x .

Such a special algorithm is of course much faster than the generic algorithms. However, feedback polynomials of low weight do not only help to speed up the implementation of an LFSR, there are also special attacks against stream ciphers based on these LFSRs (see Sect. 4.3). One should keep this in mind when designing an LFSR-based stream cipher. Most often, the extra speed up of an LFSR with a sparse feedback polynomial is not worth the risk of a special attack.

Algorithm 2.10 Generating an LFSR sequence with the feedback polynomial $z^n + z^k + 1$

```

1:  $y \leftarrow ((x \gg k) \oplus x) \& (2^k - 1)$ 
2: output  $y$ 
3:  $x \leftarrow (x \gg k) | (y \ll k)$ 

```

We close this section with some timings for the implementation of our algorithms. All programs run on a single core of a 32-bit Intel Centrino Duo with 1.66 Gz.

Stream Ciphers

Klein, A.

2013, XIX, 399 p. 71 illus., Softcover

ISBN: 978-1-4471-5078-7