Chapter 6

Stream Ciphers

Contents in Brief

6.1 Introduction

Stream ciphers are an important class of encryption algorithms. They encrypt individual characters (usually binary digits) of a plaintext message one at a time, using an encryption transformation which varies with time. By contrast, *block ciphers* (Chapter 7) tend to simultaneously encrypt groups of characters of a plaintext message using a fixed encryption transformation. Stream ciphers are generally faster than block ciphers in hardware, and have less complex hardware circuitry. They are also more appropriate, and in some cases mandatory (e.g., in some telecommunications applications), when buffering is limited or when characters must be individually processed as they are received. Because they have limited or no error propagation, stream ciphers may also be advantageous in situations where transmission errors are highly probable.

There is a vast body of theoretical knowledge on stream ciphers, and various design principles for stream ciphers have been proposed and extensively analyzed. However, there are relatively few fully-specified stream cipher algorithms in the open literature. This unfortunate state of affairs can partially be explained by the fact that most stream ciphers used in practice tend to be proprietary and confidential. By contrast, numerous concrete block cipher proposals have been published, some of which have been standardized or placed in the public domain. Nevertheless, because of their significant advantages, stream ciphers are widely used today, and one can expect increasingly more concrete proposals in the coming years.

Chapter outline

The remainder of §6.1 introduces basic concepts relevant to stream ciphers. Feedback shift registers, in particular linear feedback shift registers (LFSRs), are the basic building block in most stream ciphers that have been proposed; they are studied in §6.2. Three general techniques for utilizing LFSRs in the construction of stream ciphers are presented in §6.3: using a nonlinear combining function on the outputs of several LFSRs (§6.3.1), using a nonlinear filtering function on the contents of a single LFSR (§6.3.2), and using the output of one (or more) LFSRs to control the clock of one (or more) other LFSRs (§6.3.3). Two concrete proposals for clock-controlled generators, the alternating step generator and the shrinking generator are presented in §6.3.3. §6.4 presents a stream cipher not based on LFSRs, namely SEAL. §6.5 concludes with references and further chapter notes.

6.1.1 Classification

Stream ciphers can be either symmetric-key or public-key. The focus of this chapter is symmetric-key stream ciphers; the Blum-Goldwasser probabilistic public-key encryption scheme (§8.7.2) is an example of a public-key stream cipher.

6.1 Note (*block vs. stream ciphers*) Block ciphers process plaintext in relatively large blocks (e.g., $n \geq 64$ bits). The same function is used to encrypt successive blocks; thus (pure) block ciphers are *memoryless*. In contrast, stream ciphers process plaintext in blocks as small as a single bit, and the encryption function may vary as plaintext is processed; thus stream ciphers are said to have memory. They are sometimes called *state ciphers* since encryption depends on not only the key and plaintext, but also on the current state. This distinction between block and stream ciphers is not definitive (see Remark 7.25); adding a small amount of memory to a block cipher (as in the CBC mode) results in a stream cipher with large blocks.

(i) The one-time pad

Recall (Definition 1.39) that a *Vernam cipher* over the binary alphabet is defined by

$$c_i = m_i \oplus k_i \ \text{ for } i = 1, 2, 3 \ldots,$$

where $m_1, m_2, m_3, \ldots$ are the plaintext digits, $k_1, k_2, k_3, \ldots$ (the *keystream*) are the key digits, $c_1, c_2, c_3, \ldots$ are the ciphertext digits, and $\oplus$ is the XOR function (bitwise addition modulo 2). Decryption is defined by $m_i = c_i \oplus k_i$. If the keystream digits are generated independently and randomly, the Vernam cipher is called a *one-time pad*, and is unconditionally secure (§1.13.3(i)) against a ciphertext-only attack. More precisely, if M, C, and K are random variables respectively denoting the plaintext, ciphertext, and secret key, and if $H()$ denotes the entropy function (Definition 2.39), then $H(M|C) = H(M)$. Equivalently, $I(M;C) = 0$ (see Definition 2.45): the ciphertext contributes no information about the plaintext.

Shannon proved that a necessary condition for a symmetric-key encryption scheme to be unconditionally secure is that $H(K) \geq H(M)$. That is, the uncertainty of the secret key must be at least as great as the uncertainty of the plaintext. If the key has bitlength k, and the key bits are chosen randomly and independently, then $H(K) = k$, and Shannon's necessary condition for unconditional security becomes $k \geq H(M)$. The one-time pad is unconditionally secure regardless of the statistical distribution of the plaintext, and is optimal in the sense that its key is the smallest possible among all symmetric-key encryption schemes having this property.

An obvious drawback of the one-time pad is that the key should be as long as the plaintext, which increases the difficulty of key distribution and key management. This motivates the design of stream ciphers where the keystream is *pseudorandomly* generated from a smaller secret key, with the intent that the keystream appears random to a computationally bounded adversary. Such stream ciphers do not offer unconditional security (since $H(K) \ll H(M)$), but the hope is that they are computationally secure (§1.13.3(iv)).

Stream ciphers are commonly classified as being *synchronous* or *self-synchronizing*.

(ii) Synchronous stream ciphers

6.2 Definition A *synchronous* stream cipher is one in which the keystream is generated independently of the plaintext message and of the ciphertext.

The encryption process of a synchronous stream cipher can be described by the equations

$$\begin{aligned} \sigma_{i+1} &= f(\sigma_i, k), \\ z_i &= g(\sigma_i, k), \\ c_i &= h(z_i, m_i), \end{aligned}$$

where σ_0 is the *initial state* and may be determined from the key k, f is the *next-state function*, g is the function which produces the *keystream* z_i, and h is the *output function* which combines the keystream and plaintext m_i to produce ciphertext c_i. The encryption and decryption processes are depicted in Figure 6.1. The OFB mode of a block cipher (see §7.2.2(iv)) is an example of a synchronous stream cipher.

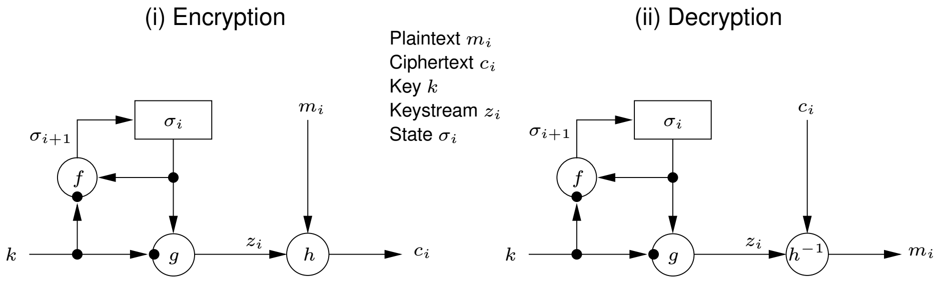

Figure 6.1: *General model of a synchronous stream cipher.*

6.3 Note (*properties of synchronous stream ciphers*)

(i) *synchronization requirements.* In a synchronous stream cipher, both the sender and receiver must be *synchronized* – using the same key and operating at the same position (state) within that key – to allow for proper decryption. If synchronization is lost due to ciphertext digits being inserted or deleted during transmission, then decryption fails and can only be restored through additional techniques for re-synchronization. Techniques for re-synchronization include re-initialization, placing special markers at regular intervals in the ciphertext, or, if the plaintext contains enough redundancy, trying all possible keystream offsets.

(ii) *no error propagation.* A ciphertext digit that is modified (but not deleted) during transmission does not affect the decryption of other ciphertext digits.

(iii) *active attacks.* As a consequence of property (i), the insertion, deletion, or replay of ciphertext digits by an active adversary causes immediate loss of synchronization, and hence might possibly be detected by the decryptor. As a consequence of property (ii), an active adversary might possibly be able to make changes to selected ciphertext digits, and know exactly what affect these changes have on the plaintext. This illustrates that additional mechanisms must be employed in order to provide data origin authentication and data integrity guarantees (see §9.5.4).

Most of the stream ciphers that have been proposed to date in the literature are additive stream ciphers, which are defined below.

6.4 Definition A *binary additive stream cipher* is a synchronous stream cipher in which the keystream, plaintext, and ciphertext digits are binary digits, and the output function h is the XOR function.

Binary additive stream ciphers are depicted in Figure 6.2. Referring to Figure 6.2, the *keystream generator* is composed of the next-state function f and the function g (see Figure 6.1), and is also known as the *running key generator*.

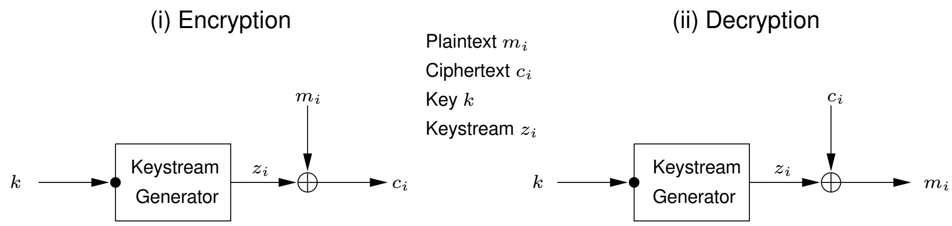

Figure 6.2: *General model of a binary additive stream cipher.*

(iii) Self-synchronizing stream ciphers

6.5 Definition A *self-synchronizing* or *asynchronous* stream cipher is one in which the keystream is generated as a function of the key and a fixed number of previous ciphertext digits.

The encryption function of a self-synchronizing stream cipher can be described by the equations

$$\begin{aligned}\sigma_i &= (c_{i-t}, c_{i-t+1}, \dots, c_{i-1}),\\ z_i &= g(\sigma_i, k),\\ c_i &= h(z_i, m_i),\end{aligned}$$

where $\sigma_0 = (c_{-t}, c_{-t+1}, \dots, c_{-1})$ is the (non-secret) *initial state*, k is the *key*, g is the function which produces the *keystream* z_i, and h is the *output function* which combines the keystream and plaintext m_i to produce ciphertext c_i. The encryption and decryption processes are depicted in Figure 6.3. The most common presently-used self-synchronizing stream ciphers are based on block ciphers in 1-bit cipher feedback mode (see §7.2.2(iii)).

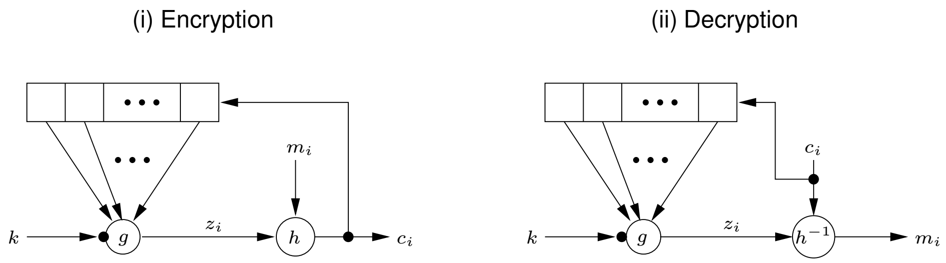

Figure 6.3: *General model of a self-synchronizing stream cipher.*

6.6 Note (*properties of self-synchronizing stream ciphers*)

(i) *self-synchronization.* Self-synchronization is possible if ciphertext digits are deleted or inserted, because the decryption mapping depends only on a fixed number of preceding ciphertext characters. Such ciphers are capable of re-establishing proper decryption automatically after loss of synchronization, with only a fixed number of plaintext characters unrecoverable.

(ii) *limited error propagation.* Suppose that the state of a self-synchronization stream cipher depends on t previous ciphertext digits. If a single ciphertext digit is modified (or even deleted or inserted) during transmission, then decryption of up to t subsequent ciphertext digits may be incorrect, after which correct decryption resumes.

(iii) *active attacks.* Property (ii) implies that any modification of ciphertext digits by an active adversary causes several other ciphertext digits to be decrypted incorrectly, thereby improving (compared to synchronous stream ciphers) the likelihood of being detected by the decryptor. As a consequence of property (i), it is more difficult (than for synchronous stream ciphers) to detect insertion, deletion, or replay of ciphertext digits by an active adversary. This illustrates that additional mechanisms must be employed in order to provide data origin authentication and data integrity guarantees (see §9.5.4).

(iv) *diffusion of plaintext statistics.* Since each plaintext digit influences the entire following ciphertext, the statistical properties of the plaintext are dispersed through the ciphertext. Hence, self-synchronizing stream ciphers may be more resistant than synchronous stream ciphers against attacks based on plaintext redundancy.

6.2 Feedback shift registers

Feedback shift registers, in particular linear feedback shift registers, are the basic components of many keystream generators. §6.2.1 introduces linear feedback shift registers. The linear complexity of binary sequences is studied in §6.2.2, while the Berlekamp-Massey algorithm for computing it is presented in §6.2.3. Finally, nonlinear feedback shift registers are discussed in §6.2.4.

6.2.1 Linear feedback shift registers

Linear feedback shift registers (LFSRs) are used in many of the keystream generators that have been proposed in the literature. There are several reasons for this:

1. LFSRs are well-suited to hardware implementation;
2. they can produce sequences of large period (Fact 6.12);
3. they can produce sequences with good statistical properties (Fact 6.14); and
4. because of their structure, they can be readily analyzed using algebraic techniques.

6.7 Definition A *linear feedback shift register* (LFSR) of length L consists of L *stages* (or *delay elements*) numbered $0, 1, \dots, L-1$, each capable of storing one bit and having one input and one output; and a clock which controls the movement of data. During each unit of time the following operations are performed:

(i) the content of stage 0 is output and forms part of the *output sequence*;

(ii) the content of stage i is moved to stage $i-1$ for each i, $1 \le i \le L-1$; and
(iii) the new content of stage $L-1$ is the *feedback bit* s_j which is calculated by adding together modulo 2 the previous contents of a fixed subset of stages $0, 1, \ldots, L-1$.

Figure 6.4 depicts an LFSR. Referring to the figure, each c_i is either 0 or 1; the closed semi-circles are AND gates; and the feedback bit s_j is the modulo 2 sum of the contents of those stages i, $0 \le i \le L-1$, for which $c_{L-i} = 1$.

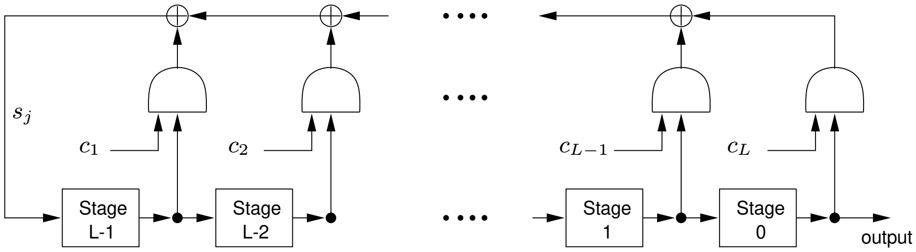

Figure 6.4: *A linear feedback shift register (LFSR) of length L.*

6.8 Definition The LFSR of Figure 6.4 is denoted $\langle L, C(D)\rangle$, where $C(D) = 1 + c_1D + c_2D^2 + \cdots + c_LD^L \in \mathbb{Z}_2[D]$ is the *connection polynomial*. The LFSR is said to be *non-singular* if the degree of $C(D)$ is L (that is, $c_L = 1$). If the initial content of stage i is $s_i \in \{0,1\}$ for each i, $0 \le i \le L-1$, then $[s_{L-1}, \ldots, s_1, s_0]$ is called the *initial state* of the LFSR.

6.9 Fact If the initial state of the LFSR in Figure 6.4 is $[s_{L-1}, \ldots, s_1, s_0]$, then the output sequence $s = s_0, s_1, s_2, \ldots$ is uniquely determined by the following recursion:

$$s_j = (c_1s_{j-1} + c_2s_{j-2} + \cdots + c_Ls_{j-L}) \bmod 2 \quad \text{for } j \ge L.$$

6.10 Example (*output sequence of an LFSR*) Consider the LFSR $\langle 4, 1 + D + D^4\rangle$ depicted in Figure 6.5. If the initial state of the LFSR is $[0,0,0,0]$, the output sequence is the zero sequence. The following tables show the contents of the stages D_3, D_2, D_1, D_0 at the end of each unit of time t when the initial state is $[0,1,1,0]$.

t	D_3	D_2	D_1	D_0
0	0	1	1	0
1	0	0	1	1
2	1	0	0	1
3	0	1	0	0
4	0	0	1	0
5	0	0	0	1
6	1	0	0	0
7	1	1	0	0

t	D_3	D_2	D_1	D_0
8	1	1	1	0
9	1	1	1	1
10	0	1	1	1
11	1	0	1	1
12	0	1	0	1
13	1	0	1	0
14	1	1	0	1
15	0	1	1	0

The output sequence is $s = 0,1,1,0,0,1,0,0,0,1,1,1,1,0,1,\ldots$, and is periodic with period 15 (see Definition 5.25). □

The significance of an LFSR being non-singular is explained by Fact 6.11.

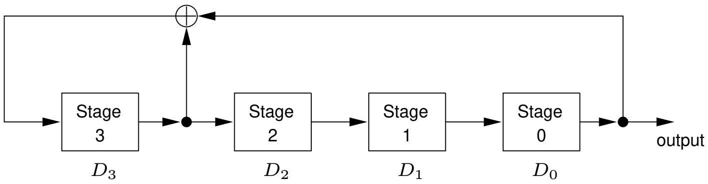

Figure 6.5: *The LFSR $\langle 4, 1 + D + D^4\rangle$ of Example 6.10.*

6.11 Fact Every output sequence (i.e., for all possible initial states) of an LFSR $\langle L, C(D)\rangle$ is periodic if and only if the connection polynomial $C(D)$ has degree L.

If an LFSR $\langle L, C(D)\rangle$ is *singular* (i.e., $C(D)$ has degree less than L), then not all output sequences are periodic. However, the output sequences are *ultimately periodic*; that is, the sequences obtained by ignoring a certain finite number of terms at the beginning are periodic. For the remainder of this chapter, it will be assumed that all LFSRs are non-singular. Fact 6.12 determines the periods of the output sequences of some special types of non-singular LFSRs.

6.12 Fact (*periods of LFSR output sequences*) Let $C(D) \in \mathbb{Z}_2[D]$ be a connection polynomial of degree L.

(i) If $C(D)$ is irreducible over $\mathbb{Z}_2$ (see Definition 2.190), then each of the $2^L - 1$ non-zero initial states of the non-singular LFSR $\langle L, C(D)\rangle$ produces an output sequence with period equal to the least positive integer N such that $C(D)$ divides $1 + D^N$ in $\mathbb{Z}_2[D]$. (Note: it is always the case that this N is a divisor of $2^L - 1$.)

(ii) If $C(D)$ is a primitive polynomial (see Definition 2.228), then each of the $2^L - 1$ non-zero initial states of the non-singular LFSR $\langle L, C(D)\rangle$ produces an output sequence with maximum possible period $2^L - 1$.

A method for generating primitive polynomials over $\mathbb{Z}_2$ uniformly at random is given in Algorithm 4.78. Table 4.8 lists a primitive polynomial of degree m over $\mathbb{Z}_2$ for each m, $1 \le m \le 229$. Fact 6.12(ii) motivates the following definition.

6.13 Definition If $C(D) \in \mathbb{Z}_2[D]$ is a primitive polynomial of degree L, then $\langle L, C(D)\rangle$ is called a *maximum-length* LFSR. The output of a maximum-length LFSR with non-zero initial state is called an *m-sequence*.

Fact 6.14 demonstrates that the output sequences of maximum-length LFSRs have good statistical properties.

6.14 Fact (*statistical properties of m-sequences*) Let s be an m-sequence that is generated by a maximum-length LFSR of length L.

(i) Let k be an integer, $1 \le k \le L$, and let $\overline{s}$ be any subsequence of s of length $2^L + k - 2$. Then each non-zero sequence of length k appears exactly 2^{L-k} times as a subsequence of $\overline{s}$. Furthermore, the zero sequence of length k appears exactly $2^{L-k} - 1$ times as a subsequence of $\overline{s}$. In other words, the distribution of patterns having fixed length of at most L is almost uniform.

(ii) s satisfies Golomb's randomness postulates (§5.4.3). That is, every m-sequence is also a pn-sequence (see Definition 5.29).

6.15 Example (*m-sequence*) Since $C(D) = 1 + D + D^4$ is a primitive polynomial over $\mathbb{Z}_2$, the LFSR $\langle 4, 1+D+D^4\rangle$ is a maximum-length LFSR. Hence, the output sequence of this LFSR is an m-sequence of maximum possible period $N = 2^4-1 = 15$ (cf. Example 6.10). Example 5.30 verifies that this output sequence satisfies Golomb's randomness properties. □

6.2.2 Linear complexity

This subsection summarizes selected results about the linear complexity of sequences. All sequences are assumed to be binary sequences. Notation: s denotes an infinite sequence whose terms are $s_0, s_1, s_2, \dots$; s^n denotes a finite sequence of length n whose terms are $s_0, s_1, \dots, s_{n-1}$ (see Definition 5.24).

6.16 Definition An LFSR is said to *generate* a sequence s if there is some initial state for which the output sequence of the LFSR is s. Similarly, an LFSR is said to *generate* a finite sequence s^n if there is some initial state for which the output sequence of the LFSR has s^n as its first n terms.

6.17 Definition The *linear complexity* of an infinite binary sequence s, denoted $L(s)$, is defined as follows:

(i) if s is the zero sequence $s = 0, 0, 0, \dots$, then $L(s) = 0$;
(ii) if no LFSR generates s, then $L(s) = \infty$;
(iii) otherwise, $L(s)$ is the length of the shortest LFSR that generates s.

6.18 Definition The *linear complexity* of a finite binary sequence s^n, denoted $L(s^n)$, is the length of the shortest LFSR that generates a sequence having s^n as its first n terms.

Facts 6.19 – 6.22 summarize some basic results about linear complexity.

6.19 Fact (*properties of linear complexity*) Let s and t be binary sequences.

(i) For any $n \geq 1$, the linear complexity of the subsequence s^n satisfies $0 \leq L(s^n) \leq n$.
(ii) $L(s^n) = 0$ if and only if s^n is the zero sequence of length n.
(iii) $L(s^n) = n$ if and only if $s^n = 0, 0, 0, \dots, 0, 1$.
(iv) If s is periodic with period N, then $L(s) \leq N$.
(v) $L(s \oplus t) \leq L(s) + L(t)$, where $s \oplus t$ denotes the bitwise XOR of s and t.

6.20 Fact If the polynomial $C(D) \in \mathbb{Z}_2[D]$ is irreducible over $\mathbb{Z}_2$ and has degree L, then each of the 2^L-1 non-zero initial states of the non-singular LFSR $\langle L, C(D)\rangle$ produces an output sequence with linear complexity L.

6.21 Fact (*expectation and variance of the linear complexity of a random sequence*) Let s^n be chosen uniformly at random from the set of all binary sequences of length n, and let $L(s^n)$ be the linear complexity of s^n. Let $B(n)$ denote the parity function: $B(n) = 0$ if n is even; $B(n) = 1$ if n is odd.

(i) The expected linear complexity of s^n is

$$E(L(s^n)) = \frac{n}{2} + \frac{4 + B(n)}{18} - \frac{1}{2^n}\left(\frac{n}{3} + \frac{2}{9}\right).$$

Hence, for moderately large n, $E(L(s^n)) \approx \frac{n}{2} + \frac{2}{9}$ if n is even, and $E(L(s^n)) \approx \frac{n}{2} + \frac{5}{18}$ if n is odd.

(ii) The variance of the linear complexity of s^n is $\mathrm{Var}(L(s^n)) =$

$$\frac{86}{81} - \frac{1}{2^n}\left(\frac{14 - B(n)}{27}n + \frac{82 - 2B(n)}{81}\right) - \frac{1}{2^{2n}}\left(\frac{1}{9}n^2 + \frac{4}{27}n + \frac{4}{81}\right).$$

Hence, $\mathrm{Var}(L(s^n)) \approx \frac{86}{81}$ for moderately large n.

6.22 Fact (*expectation of the linear complexity of a random periodic sequence*) Let s^n be chosen uniformly at random from the set of all binary sequences of length n, where $n = 2^t$ for some fixed $t \geq 1$, and let s be the n-periodic infinite sequence obtained by repeating the sequence s^n. Then the expected linear complexity of s is $E(L(s^n)) = n - 1 + 2^{-n}$.

The linear complexity profile of a binary sequence is introduced next.

6.23 Definition Let $s = s_0, s_1, \ldots$ be a binary sequence, and let L_N denote the linear complexity of the subsequence $s^N = s_0, s_1, \ldots, s_{N-1}$, $N \geq 0$. The sequence $L_1, L_2, \ldots$ is called the *linear complexity profile of* s. Similarly, if $s^n = s_0, s_1, \ldots, s_{n-1}$ is a finite binary sequence, the sequence $L_1, L_2, \ldots, L_n$ is called the *linear complexity profile of* s^n.

The linear complexity profile of a sequence can be computed using the Berlekamp-Massey algorithm (Algorithm 6.30); see also Note 6.31. The following properties of the linear complexity profile can be deduced from Fact 6.29.

6.24 Fact (*properties of linear complexity profile*) Let $L_1, L_2, \ldots$ be the linear complexity profile of a sequence $s = s_0, s_1, \ldots$.

(i) If $j > i$, then $L_j \geq L_i$.

(ii) $L_{N+1} > L_N$ is possible only if $L_N \leq N/2$.

(iii) If $L_{N+1} > L_N$, then $L_{N+1} + L_N = N + 1$.

The linear complexity profile of a sequence s can be graphed by plotting the points (N, L_N), $N \geq 1$, in the $N \times L$ plane and joining successive points by a horizontal line followed by a vertical line, if necessary (see Figure 6.6). Fact 6.24 can then be interpreted as saying that the graph of a linear complexity profile is non-decreasing. Moreover, a (vertical) jump in the graph can only occur from below the line $L = N/2$; if a jump occurs, then it is symmetric about this line. Fact 6.25 shows that the expected linear complexity of a random sequence should closely follow the line $L = N/2$.

6.25 Fact (*expected linear complexity profile of a random sequence*) Let $s = s_0, s_1, \ldots$ be a random sequence, and let L_N be the linear complexity of the subsequence $s^N = s_0, s_1, \ldots, s_{N-1}$ for each $N \geq 1$. For any fixed index $N \geq 1$, the expected smallest j for which $L_{N+j} > L_N$ is 2 if $L_N \leq N/2$, or $2 + 2L_N - N$ if $L_N > N/2$. Moreover, the expected increase in linear complexity is 2 if $L_N \geq N/2$, or $N - 2L_N + 2$ if $L_N < N/2$.

6.26 Example (*linear complexity profile*) Consider the 20-periodic sequence s with cycle

$$s^{20} = 1,0,0,1,0,0,1,1,1,1,0,0,0,1,0,0,1,1,1,0.$$

The linear complexity profile of s is $1,1,1,3,3,3,3,5,5,5,6,6,6,8,8,8,9,9,10,10,11,$ $11,11,11,14,14,14,14,15,15,15,17,17,17,18,18,19,19,19,19,\ldots$. Figure 6.6 shows the graph of the linear complexity profile of s. □

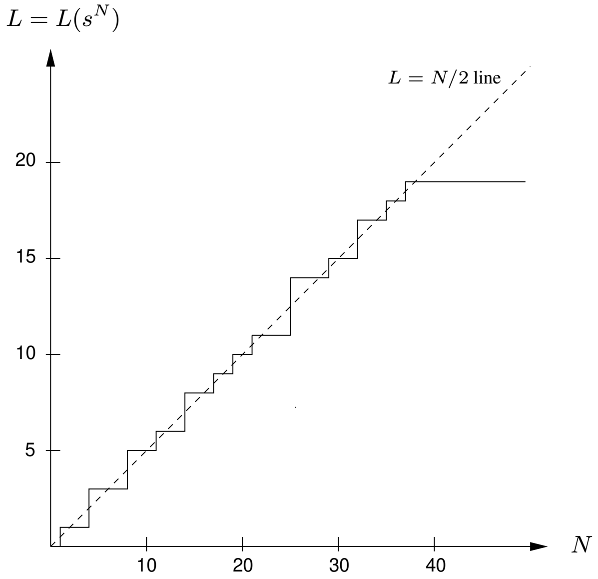

Figure 6.6: *Linear complexity profile of the* 20*-periodic sequence of Example 6.26.*

As is the case with all statistical tests for randomness (cf. §5.4), the condition that a sequence s have a linear complexity profile that closely resembles that of a random sequence is *necessary* but not *sufficient* for s to be considered random. This point is illustrated in the following example.

6.27 Example (*limitations of the linear complexity profile*) The linear complexity profile of the sequence s defined as

$$s_i = \begin{cases} 1, & \text{if } i = 2^j - 1 \text{ for some } j \geq 0, \\ 0, & \text{otherwise,} \end{cases}$$

follows the line $L = N/2$ as closely as possible. That is, $L(s^N) = \lfloor (N+1)/2 \rfloor$ for all $N \geq 1$. However, the sequence s is clearly non-random. □

6.2.3 Berlekamp-Massey algorithm

The Berlekamp-Massey algorithm (Algorithm 6.30) is an efficient algorithm for determining the linear complexity of a finite binary sequence s^n of length n (see Definition 6.18). The algorithm takes n iterations, with the Nth iteration computing the linear complexity of the subsequence s^N consisting of the first N terms of s^n. The theoretical basis for the algorithm is Fact 6.29.

6.28 Definition Consider the finite binary sequence $s^{N+1} = s_0, s_1, \ldots, s_{N-1}, s_N$. For $C(D) = 1 + c_1 D + \cdots + c_L D^L$, let $\langle L, C(D) \rangle$ be an LFSR that generates the subsequence $s^N = s_0, s_1, \ldots, s_{N-1}$. The *next discrepancy* d_N is the difference between s_N and the $(N+1)^{\text{st}}$ term generated by the LFSR: $d_N = (s_N + \sum_{i=1}^{L} c_i s_{N-i}) \bmod 2$.

6.29 Fact Let $s^N = s_0, s_1, \ldots, s_{N-1}$ be a finite binary sequence of linear complexity $L = L(s^N)$, and let $\langle L, C(D) \rangle$ be an LFSR which generates s^N.

(i) The LFSR $\langle L, C(D)\rangle$ also generates $s^{N+1} = s_0, s_1, \ldots, s_{N-1}, s_N$ if and only if the next discrepancy d_N is equal to 0.
(ii) If $d_N = 0$, then $L(s^{N+1}) = L$.
(iii) Suppose $d_N = 1$. Let m the largest integer $< N$ such that $L(s^m) < L(s^N)$, and let $\langle L(s^m), B(D)\rangle$ be an LFSR of length $L(s^m)$ which generates s^m. Then $\langle L', C'(D)\rangle$ is an LFSR of smallest length which generates s^{N+1}, where

$$L' = \begin{cases} L, & \text{if } L > N/2, \\ N+1-L, & \text{if } L \leq N/2, \end{cases}$$

and $C'(D) = C(D) + B(D) \cdot D^{N-m}$.

6.30 Algorithm Berlekamp-Massey algorithm

INPUT: a binary sequence $s^n = s_0, s_1, s_2, \ldots, s_{n-1}$ of length n.
OUTPUT: the linear complexity $L(s^n)$ of s^n, $0 \leq L(s^n) \leq n$.

1. *Initialization.* $C(D) \leftarrow 1,\ L \leftarrow 0,\ m \leftarrow -1,\ B(D) \leftarrow 1,\ N \leftarrow 0$.
2. While $(N < n)$ do the following:
   - 2.1 *Compute the next discrepancy d.* $d \leftarrow (s_N + \sum_{i=1}^{L} c_i s_{N-i}) \bmod 2$.
   - 2.2 If $d = 1$ then do the following:
     $T(D) \leftarrow C(D),\ C(D) \leftarrow C(D) + B(D) \cdot D^{N-m}$.
     If $L \leq N/2$ then $L \leftarrow N+1-L,\ m \leftarrow N,\ B(D) \leftarrow T(D)$.
   - 2.3 $N \leftarrow N + 1$.
3. Return(L).

6.31 Note (*intermediate results in Berlekamp-Massey algorithm*) At the end of each iteration of step 2, $\langle L, C(D)\rangle$ is an LFSR of smallest length which generates s^N. Hence, Algorithm 6.30 can also be used to compute the linear complexity profile (Definition 6.23) of a finite sequence.

6.32 Fact The running time of the Berlekamp-Massey algorithm (Algorithm 6.30) for determining the linear complexity of a binary sequence of bitlength n is $O(n^2)$ bit operations.

6.33 Example (*Berlekamp-Massey algorithm*) Table 6.1 shows the steps of Algorithm 6.30 for computing the linear complexity of the binary sequence $s^n = 0,0,1,1,0,1,1,1,0$ of length $n = 9$. This sequence is found to have linear complexity 5, and an LFSR which generates it is $\langle 5, 1 + D^3 + D^5\rangle$. □

6.34 Fact Let s^n be a finite binary sequence of length n, and let the linear complexity of s^n be L. Then there is a unique LFSR of length L which generates s^n if and only if $L \leq \frac{n}{2}$.

An important consequence of Fact 6.34 and Fact 6.24(iii) is the following.

6.35 Fact Let s be an (infinite) binary sequence of linear complexity L, and let t be a (finite) subsequence of s of length at least $2L$. Then the Berlekamp-Massey algorithm (with step 3 modified to return both L and $C(D)$) on input t determines an LFSR of length L which generates s.

s_N	d	$T(D)$	$C(D)$	L	m	$B(D)$	N
$-$	$-$	$-$	1	0	-1	1	0
0	0	$-$	1	0	-1	1	1
0	0	$-$	1	0	-1	1	2
1	1	1	$1+D^3$	3	2	1	3
1	1	$1+D^3$	$1+D+D^3$	3	2	1	4
0	1	$1+D+D^3$	$1+D+D^2+D^3$	3	2	1	5
1	1	$1+D+D^2+D^3$	$1+D+D^2$	3	2	1	6
1	0	$1+D+D^2+D^3$	$1+D+D^2$	3	2	1	7
1	1	$1+D+D^2$	$1+D+D^2+D^5$	5	7	$1+D+D^2$	8
0	1	$1+D+D^2+D^5$	$1+D^3+D^5$	5	7	$1+D+D^2$	9

Table 6.1: *Steps of the Berlekamp-Massey algorithm of Example 6.33.*

6.2.4 Nonlinear feedback shift registers

This subsection summarizes selected results about nonlinear feedback shift registers. A function with n binary inputs and one binary output is called a *Boolean function* of n variables; there are 2^{2^n} different Boolean functions of n variables.

6.36 Definition A (general) *feedback shift register* (FSR) of length L consists of L *stages* (or *delay elements*) numbered $0, 1, \ldots, L-1$, each capable of storing one bit and having one input and one output, and a clock which controls the movement of data. During each unit of time the following operations are performed:

(i) the content of stage 0 is output and forms part of the *output sequence*;
(ii) the content of stage i is moved to stage $i-1$ for each i, $1 \le i \le L-1$; and
(iii) the new content of stage $L-1$ is the *feedback bit* $s_j = f(s_{j-1}, s_{j-2}, \ldots, s_{j-L})$, where the *feedback function* f is a Boolean function and s_{j-i} is the previous content of stage $L-i$, $1 \le i \le L$.

If the initial content of stage i is $s_i \in \{0,1\}$ for each $0 \le i \le L-1$, then $[s_{L-1}, \ldots, s_1, s_0]$ is called the *initial state* of the FSR.

Figure 6.7 depicts an FSR. Note that if the feedback function f is a linear function, then the FSR is an LFSR (Definition 6.7). Otherwise, the FSR is called a *nonlinear* FSR.

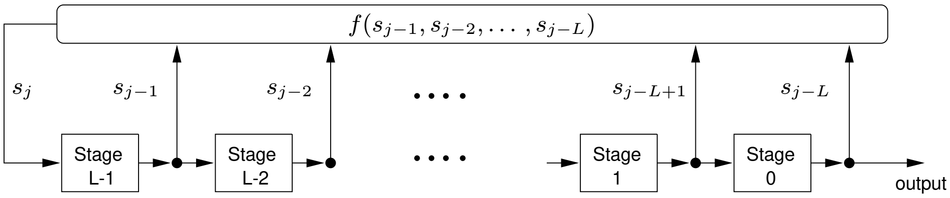

Figure 6.7: *A feedback shift register (FSR) of length L.*

6.37 Fact If the initial state of the FSR in Figure 6.7 is $[s_{L-1}, \ldots, s_1, s_0]$, then the output sequence $s = s_0, s_1, s_2, \ldots$ is uniquely determined by the following recursion:

$$s_j = f(s_{j-1}, s_{j-2}, \ldots, s_{j-L}) \quad \text{for } j \ge L.$$

6.38 Definition An FSR is said to be *non-singular* if and only if every output sequence of the FSR (i.e., for all possible initial states) is periodic.

6.39 Fact An FSR with feedback function $f(s_{j-1}, s_{j-2}, \ldots, s_{j-L})$ is non-singular if and only if f is of the form $f = s_{j-L} \oplus g(s_{j-1}, s_{j-2}, \ldots, s_{j-L+1})$ for some Boolean function g.

The period of the output sequence of a non-singular FSR of length L is at most 2^L.

6.40 Definition If the period of the output sequence (for any initial state) of a non-singular FSR of length L is 2^L, then the FSR is called a *de Bruijn FSR*, and the output sequence is called a *de Bruijn sequence*.

6.41 Example (*de Bruijn sequence*) Consider the FSR of length 3 with nonlinear feedback function $f(x_1, x_2, x_3) = 1 \oplus x_2 \oplus x_3 \oplus x_1 x_2$. The following tables show the contents of the 3 stages of the FSR at the end of each unit of time t when the initial state is $[0, 0, 0]$.

t	Stage 2	Stage 1	Stage 0
0	0	0	0
1	1	0	0
2	1	1	0
3	1	1	1

t	Stage 2	Stage 1	Stage 0
4	0	1	1
5	1	0	1
6	0	1	0
7	0	0	1

The output sequence is the de Bruijn sequence with cycle $0, 0, 0, 1, 1, 1, 0, 1$. □

Fact 6.42 demonstrates that the output sequence of de Bruijn FSRs have good statistical properties (compare with Fact 6.14(i)).

6.42 Fact (*statistical properties of de Bruijn sequences*) Let s be a de Bruijn sequence that is generated by a de Bruijn FSR of length L. Let k be an integer, $1 \leq k \leq L$, and let $\overline{s}$ be any subsequence of s of length $2^L + k - 1$. Then each sequence of length k appears exactly 2^{L-k} times as a subsequence of $\overline{s}$. In other words, the distribution of patterns having fixed length of at most L is uniform.

6.43 Note (*converting a maximum-length LFSR to a de Bruijn FSR*) Let R_1 be a maximum-length LFSR of length L with (linear) feedback function $f(s_{j-1}, s_{j-2}, \ldots, s_{j-L})$. Then the FSR R_2 with feedback function $g(s_{j-1}, s_{j-2}, \ldots, s_{j-L}) = f \oplus \overline{s}_{j-1}\overline{s}_{j-2} \cdots \overline{s}_{j-L+1}$ is a de Bruijn FSR. Here, $\overline{s}_i$ denotes the complement of s_i. The output sequence of R_2 is obtained from that of R_1 by simply adding a 0 to the end of each subsequence of $L - 1$ 0's occurring in the output sequence of R_1.

6.3 Stream ciphers based on LFSRs

As mentioned in the beginning of §6.2.1, linear feedback shift registers are widely used in keystream generators because they are well-suited for hardware implementation, produce sequences having large periods and good statistical properties, and are readily analyzed using algebraic techniques. Unfortunately, the output sequences of LFSRs are also easily predictable, as the following argument shows. Suppose that the output sequence s of an LFSR has linear complexity L. The connection polynomial $C(D)$ of an LFSR of length L which generates s can be efficiently determined using the Berlekamp-Massey algorithm

(Algorithm 6.30) from any (short) subsequence t of s having length at least $n = 2L$ (cf. Fact 6.35). Having determined $C(D)$, the LFSR $\langle L, C(D)\rangle$ can then be initialized with any substring of t having length L, and used to generate the remainder of the sequence s. An adversary may obtain the required subsequence t of s by mounting a known or chosen-plaintext attack (§1.13.1) on the stream cipher: if the adversary knows the plaintext subsequence $m_1, m_2, \ldots, m_n$ corresponding to a ciphertext sequence $c_1, c_2, \ldots, c_n$, the corresponding keystream bits are obtained as $m_i \oplus c_i$, $1 \leq i \leq n$.

6.44 Note (*use of LFSRs in keystream generators*) Since a well-designed system should be secure against known-plaintext attacks, an LFSR should never be used by itself as a keystream generator. Nevertheless, LFSRs are desirable because of their very low implementation costs. Three general methodologies for destroying the linearity properties of LFSRs are discussed in this section:

(i) using a nonlinear combining function on the outputs of several LFSRs (§6.3.1);
(ii) using a nonlinear filtering function on the contents of a single LFSR (§6.3.2); and
(iii) using the output of one (or more) LFSRs to control the clock of one (or more) other LFSRs (§6.3.3).

Desirable properties of LFSR-based keystream generators

For essentially all possible secret keys, the output sequence of an LFSR-based keystream generator should have the following properties:

1. large period;
2. large linear complexity; and
3. good statistical properties (e.g., as described in Fact 6.14).

It is emphasized that these properties are only *necessary* conditions for a keystream generator to be considered cryptographically secure. Since mathematical proofs of security of such generators are not known, such generators can only be deemed *computationally secure* (§1.13.3(iv)) after having withstood sufficient public scrutiny.

6.45 Note (*connection polynomial*) Since a desirable property of a keystream generator is that its output sequences have large periods, component LFSRs should always be chosen to be maximum-length LFSRs, i.e., the LFSRs should be of the form $\langle L, C(D)\rangle$ where $C(D) \in \mathbb{Z}_2[D]$ is a primitive polynomial of degree L (see Definition 6.13 and Fact 6.12(ii)).

6.46 Note (*known vs. secret connection polynomial*) The LFSRs in an LFSR-based keystream generator may have *known* or *secret* connection polynomials. For known connections, the secret key generally consists of the initial contents of the component LFSRs. For secret connections, the secret key for the keystream generator generally consists of both the initial contents and the connections.

For LFSRs of length L with secret connections, the connection polynomials should be selected uniformly at random from the set of all primitive polynomials of degree L over $\mathbb{Z}_2$. Secret connections are generally recommended over known connections as the former are more resistant to certain attacks which use precomputation for analyzing the particular connection, and because the former are more amenable to statistical analysis. Secret connection LFSRs have the drawback of requiring extra circuitry to implement in hardware. However, because of the extra security possible with secret connections, this cost may sometimes be compensated for by choosing shorter LFSRs.

6.47 Note (*sparse vs. dense connection polynomial*) For implementation purposes, it is advantageous to choose an LFSR that is *sparse*; i.e., only a few of the coefficients of the connection polynomial are non-zero. Then only a small number of connections must be made between the stages of the LFSR in order to compute the feedback bit. For example, the connection polynomial might be chosen to be a primitive trinomial (cf. Table 4.8). However, in some LFSR-based keystream generators, special attacks can be mounted if sparse connection polynomials are used. Hence, it is generally recommended not to use sparse connection polynomials in LFSR-based keystream generators.

6.3.1 Nonlinear combination generators

One general technique for destroying the linearity inherent in LFSRs is to use several LFSRs in parallel. The keystream is generated as a nonlinear function f of the outputs of the component LFSRs; this construction is illustrated in Figure 6.8. Such keystream generators are called *nonlinear combination generators*, and f is called the *combining function*. The remainder of this subsection demonstrates that the function f must satisfy several criteria in order to withstand certain particular cryptographic attacks.

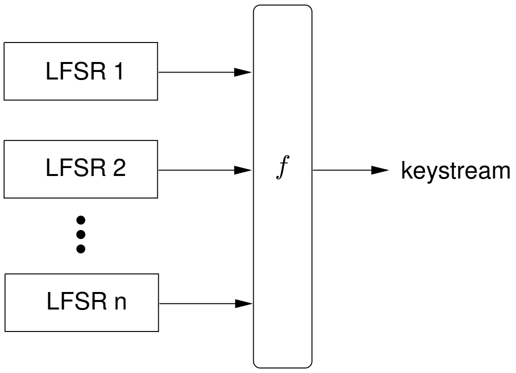

Figure 6.8: *A nonlinear combination generator. f is a nonlinear combining function.*

6.48 Definition A product of m distinct variables is called an m^{th} *order product* of the variables. Every Boolean function $f(x_1, x_2, \dots, x_n)$ can be written as a modulo 2 sum of distinct m^{th} order products of its variables, $0 \le m \le n$; this expression is called the *algebraic normal form* of f. The *nonlinear order* of f is the maximum of the order of the terms appearing in its algebraic normal form.

For example, the Boolean function $f(x_1, x_2, x_3, x_4, x_5) = 1 \oplus x_2 \oplus x_3 \oplus x_4x_5 \oplus x_1x_3x_4x_5$ has nonlinear order 4. Note that the maximum possible nonlinear order of a Boolean function in n variables is n. Fact 6.49 demonstrates that the output sequence of a nonlinear combination generator has high linear complexity, provided that a combining function f of high nonlinear order is employed.

6.49 Fact Suppose that n maximum-length LFSRs, whose lengths $L_1, L_2, \dots, L_n$ are pairwise distinct and greater than 2, are combined by a nonlinear function $f(x_1, x_2, \dots, x_n)$ (as in Figure 6.8) which is expressed in algebraic normal form. Then the linear complexity of the keystream is $f(L_1, L_2, \dots, L_n)$. (The expression $f(L_1, L_2, \dots, L_n)$ is evaluated over the integers rather than over $\mathbb{Z}_2$.)

6.50 Example (*Geffe generator*) The Geffe generator, as depicted in Figure 6.9, is defined by three maximum-length LFSRs whose lengths L_1, L_2, L_3 are pairwise relatively prime, with nonlinear combining function

$$f(x_1, x_2, x_3) \;=\; x_1x_2 \oplus (1 + x_2)x_3 \;=\; x_1x_2 \oplus x_2x_3 \oplus x_3.$$

The keystream generated has period $(2^{L_1} - 1) \cdot (2^{L_2} - 1) \cdot (2^{L_3} - 1)$ and linear complexity $L = L_1L_2 + L_2L_3 + L_3$.

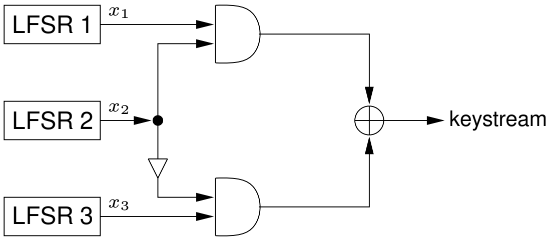

Figure 6.9: *The Geffe generator.*

The Geffe generator is cryptographically weak because information about the states of LFSR 1 and LFSR 3 leaks into the output sequence. To see this, let $x_1(t), x_2(t), x_3(t), z(t)$ denote the t^{th} output bits of LFSRs 1, 2, 3 and the keystream, respectively. Then the *correlation probability* of the sequence $x_1(t)$ to the output sequence $z(t)$ is

$$\begin{aligned} P(z(t) = x_1(t)) &= P(x_2(t) = 1) + P(x_2(t) = 0) \cdot P(x_3(t) = x_1(t)) \\ &= \frac{1}{2} + \frac{1}{2} \cdot \frac{1}{2} \;=\; \frac{3}{4}. \end{aligned}$$

Similarly, $P(z(t) = x_3(t)) = \frac{3}{4}$. For this reason, despite having high period and moderately high linear complexity, the Geffe generator succumbs to correlation attacks, as described in Note 6.51. □

6.51 Note (*correlation attacks*) Suppose that n maximum-length LFSRs $R_1, R_2, \ldots, R_n$ of lengths $L_1, L_2, \ldots, L_n$ are employed in a nonlinear combination generator. If the connection polynomials of the LFSRs and the combining function f are public knowledge, then the number of different keys of the generator is $\prod_{i=1}^{n}(2^{L_i} - 1)$. (A key consists of the initial states of the LFSRs.) Suppose that there is a correlation between the keystream and the output sequence of R_1, with correlation probability $p > \frac{1}{2}$. If a sufficiently long segment of the keystream is known (e.g., as is possible under a known-plaintext attack on a binary additive stream cipher), the initial state of R_1 can be deduced by counting the number of coincidences between the keystream and all possible shifts of the output sequence of R_1, until this number agrees with the correlation probability p. Under these conditions, finding the initial state of R_1 will take at most $2^{L_1} - 1$ trials. In the case where there is a correlation between the keystream and the output sequences of each of $R_1, R_2, \ldots, R_n$, the (secret) initial state of each LFSR can be determined independently in a total of about $\sum_{i=1}^{n}(2^{L_i} - 1)$ trials; this number is far smaller than the total number of different keys. In a similar manner, correlations between the output sequences of particular subsets of the LFSRs and the keystream can be exploited.

In view of Note 6.51, the combining function f should be carefully selected so that there is no statistical dependence between any small subset of the n LFSR sequences and the keystream. This condition can be satisfied if f is chosen to be m^{th}-order correlation immune.

6.52 Definition Let $X_1, X_2, \ldots, X_n$ be independent binary variables, each taking on the values 0 or 1 with probability $\frac{1}{2}$. A Boolean function $f(x_1, x_2, \ldots, x_n)$ is m^{th}-*order correlation immune* if for each subset of m random variables $X_{i_1}, X_{i_2}, \ldots, X_{i_m}$ with $1 \leq i_1 < i_2 < \cdots < i_m \leq n$, the random variable $Z = f(X_1, X_2, \ldots, X_n)$ is statistically independent of the random vector $(X_{i_1}, X_{i_2}, \ldots, X_{i_m})$; equivalently, $I(Z; X_{i_1}, X_{i_2}, \ldots, X_{i_m}) = 0$ (see Definition 2.45).

For example, the function $f(x_1, x_2, \ldots, x_n) = x_1 \oplus x_2 \oplus \cdots \oplus x_n$ is $(n-1)^{\text{th}}$-order correlation immune. In light of Fact 6.49, the following shows that there is a tradeoff between achieving high linear complexity and high correlation immunity with a combining function.

6.53 Fact If a Boolean function $f(x_1, x_2, \ldots, x_n)$ is m^{th}-order correlation immune, where $1 \leq m < n$, then the nonlinear order of f is at most $n - m$. Moreover, if f is *balanced* (i.e., exactly half of the output values of f are 0) then the nonlinear order of f is at most $n-m-1$ for $1 \leq m \leq n-2$.

The tradeoff between high linear complexity and high correlation immunity can be avoided by permitting *memory* in the nonlinear combination function f. This point is illustrated by the summation generator.

6.54 Example (*summation generator*) The combining function in the summation generator is based on the fact that integer addition, when viewed over $\mathbb{Z}_2$, is a nonlinear function with memory whose correlation immunity is maximum. To see this in the case $n = 2$, let $a = a_{m-1}2^{m-1} + \cdots + a_1 2 + a_0$ and $b = b_{m-1}2^{m-1} + \cdots + b_1 2 + b_0$ be the binary representations of integers a and b. Then the bits of $z = a + b$ are given by the recursive formula:

$$\begin{aligned} z_j &= f_1(a_j, b_j, c_{j-1}) = a_j \oplus b_j \oplus c_{j-1} \quad 0 \leq j \leq m, \\ c_j &= f_2(a_j, b_j, c_{j-1}) = a_j b_j \oplus (a_j \oplus b_j) c_{j-1}, \quad 0 \leq j \leq m-1, \end{aligned}$$

where c_j is the carry bit, and $c_{-1} = a_m = b_m = 0$. Note that f_1 is 2^{nd}-order correlation immune, while f_2 is a *memoryless* nonlinear function. The carry bit c_{j-1} carries all the nonlinear influence of less significant bits of a and b (namely, $a_{j-1}, \ldots, a_1, a_0$ and $b_{j-1}, \ldots, b_1, b_0$).

The summation generator, as depicted in Figure 6.10, is defined by n maximum-length LFSRs whose lengths $L_1, L_2, \ldots, L_n$ are pairwise relatively prime. The secret key con-

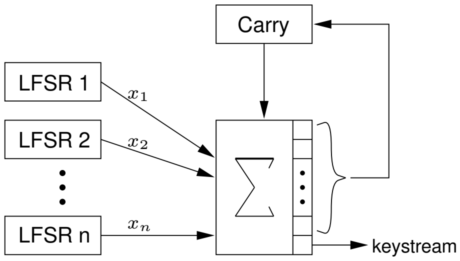

Figure 6.10: *The summation generator.*

sists of the initial states of the LFSRs, and an initial (integer) carry C_0. The keystream is generated as follows. At time j ($j \geq 1$), the LFSRs are stepped producing output bits $x_1, x_2, \dots, x_n$, and the *integer* sum $S_j = \sum_{i=1}^{n} x_i + C_{j-1}$ is computed. The keystream bit is $S_j \bmod 2$ (the least significant bit of S_j), while the new carry is computed as $C_j = \lfloor S_j/2 \rfloor$ (the remaining bits of S_j). The period of the keystream is $\prod_{i=1}^{n}(2^{L_i} - 1)$, while its linear complexity is close to this number.

Even though the summation generator has high period, linear complexity, and correlation immunity, it is vulnerable to certain correlation attacks and a known-plaintext attack based on its 2-adic span (see page 218). □

6.3.2 Nonlinear filter generators

Another general technique for destroying the linearity inherent in LFSRs is to generate the keystream as some nonlinear function of the stages of a single LFSR; this construction is illustrated in Figure 6.11. Such keystream generators are called *nonlinear filter generators*, and f is called the *filtering function*.

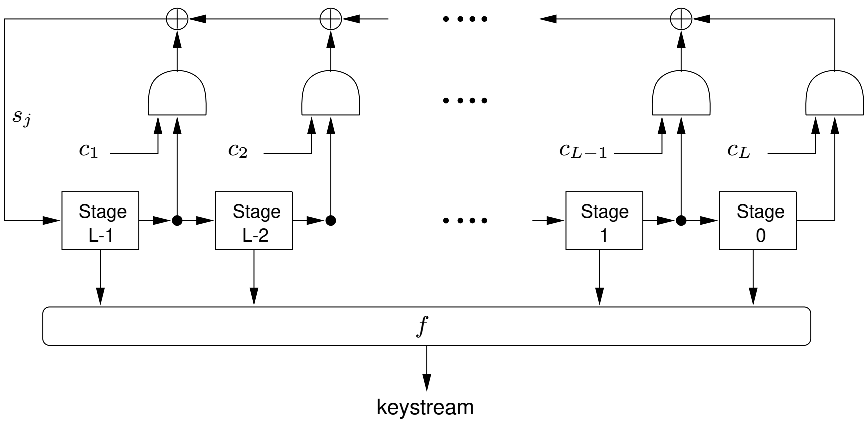

Figure 6.11: *A nonlinear filter generator. f is a nonlinear Boolean filtering function.*

Fact 6.55 describes the linear complexity of the output sequence of a nonlinear filter generator.

6.55 Fact Suppose that a nonlinear filter generator is constructed using a maximum-length LFSR of length L and a filtering function f of nonlinear order m (as in Figure 6.11).

(i) (*Key's bound*) The linear complexity of the keystream is at most $L_m = \sum_{i=1}^{m} \binom{L}{i}$.

(ii) For a fixed maximum-length LFSR of prime length L, the fraction of Boolean functions f of nonlinear order m which produce sequences of maximum linear complexity L_m is

$$P_m \approx \exp(-L_m/(L \cdot 2^L)) > e^{-1/L}.$$

Therefore, for large L, most of the generators produce sequences whose linear complexity meets the upper bound in (i).

The nonlinear function f selected for a filter generator should include many terms of each order up to the nonlinear order of f.

6.56 Example (*knapsack generator*) The knapsack keystream generator is defined by a maximum-length LFSR $\langle L, C(D)\rangle$ and a modulus $Q = 2^L$. The secret key consists of L knapsack integer weights $a_1, a_2, \ldots, a_L$ each of bitlength L, and the initial state of the LFSR. Recall that the subset sum problem (§3.10) is to determine a subset of the knapsack weights which add up to a given integer s, provided that such a subset exists; this problem is **NP**-hard (Fact 3.91). The keystream is generated as follows: at time j, the LFSR is stepped and the knapsack sum $S_j = \sum_{i=1}^{L} x_i a_i \bmod Q$ is computed, where $[x_L, \ldots, x_2, x_1]$ is the state of the LFSR at time j. Finally, selected bits of S_j (after S_j is converted to its binary representation) are extracted to form part of the keystream (the $\lceil \lg L \rceil$ least significant bits of S_j should be discarded). The linear complexity of the keystream is then virtually certain to be $L(2^L - 1)$.

Since the state of an LFSR is a binary vector, the function which maps the LFSR state to the knapsack sum S_j is indeed nonlinear. Explicitly, let the function f be defined by $f(x) = \sum_{i=1}^{L} x_i a_i \bmod Q$, where $x = [x_L, \ldots, x_2, x_1]$ is a state. If x and y are two states then, in general, $f(x \oplus y) \neq f(x) + f(y)$. □

6.3.3 Clock-controlled generators

In nonlinear combination generators and nonlinear filter generators, the component LFSRs are clocked regularly; i.e., the movement of data in all the LFSRs is controlled by the same clock. The main idea behind a *clock-controlled generator* is to introduce nonlinearity into LFSR-based keystream generators by having the output of one LFSR control the *clocking* (i.e., stepping) of a second LFSR. Since the second LFSR is clocked in an irregular manner, the hope is that attacks based on the regular motion of LFSRs can be foiled. Two clock-controlled generators are described in this subsection: (i) the alternating step generator and (ii) the shrinking generator.

(i) The alternating step generator

The alternating step generator uses an LFSR R_1 to control the stepping of two LFSRs, R_2 and R_3. The keystream produced is the XOR of the output sequences of R_2 and R_3.

6.57 Algorithm Alternating step generator

SUMMARY: a control LFSR R_1 is used to selectively step two other LFSRs, R_2 and R_3.
OUTPUT: a sequence which is the bitwise XOR of the output sequences of R_2 and R_3.
The following steps are repeated until a keystream of desired length is produced.

1. Register R_1 is clocked.
2. If the output of R_1 is 1 then:
   R_2 is clocked; R_3 is not clocked but its previous output bit is repeated.
   (For the first clock cycle, the "previous output bit" of R_3 is taken to be 0.)
3. If the output of R_1 is 0 then:
   R_3 is clocked; R_2 is not clocked but its previous output bit is repeated.
   (For the first clock cycle, the "previous output bit" of R_2 is taken to be 0.)
4. The output bits of R_2 and R_3 are XORed; the resulting bit is part of the keystream.

More formally, let the output sequences of LFSRs R_1, R_2, and R_3 be $a_0, a_1, a_2, \ldots$, $b_0, b_1, b_2, \ldots$, and $c_0, c_1, c_2 \ldots$, respectively. Define $b_{-1} = c_{-1} = 0$. Then the keystream produced by the alternating step generator is $x_0, x_1, x_2, \ldots$, where $x_j = b_{t(j)} \oplus c_{j-t(j)-1}$

and $t(j) = (\sum_{i=0}^{j} a_i) - 1$ for all $j \geq 0$. The alternating step generator is depicted in Figure 6.12.

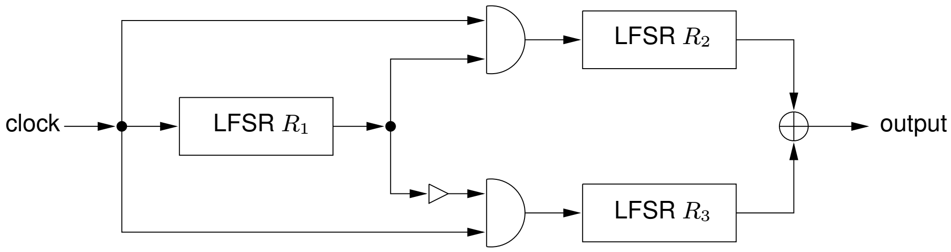

Figure 6.12: *The alternating step generator.*

6.58 Example (*alternating step generator with artificially small parameters*) Consider an alternating step generator with component LFSRs $R_1 = \langle 3, 1 + D^2 + D^3 \rangle$, $R_2 = \langle 4, 1 + D^3 + D^4 \rangle$, and $R_3 = \langle 5, 1 + D + D^3 + D^4 + D^5 \rangle$. Suppose that the initial states of R_1, R_2, and R_3 are $[0, 0, 1]$, $[1, 0, 1, 1]$, and $[0, 1, 0, 0, 1]$, respectively. The output sequence of R_1 is the 7-periodic sequence with cycle

$$a^7 = 1, 0, 0, 1, 0, 1, 1.$$

The output sequence of R_2 is the 15-periodic sequence with cycle

$$b^{15} = 1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 0, 0, 1, 0, 0.$$

The output sequence of R_3 is the 31-periodic sequence with cycle

$$c^{31} = 1, 0, 0, 1, 0, 1, 0, 1, 1, 0, 0, 0, 0, 1, 1, 1, 0, 0, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 0, 0, 0.$$

The keystream generated is

$$x = 1, 0, 1, 1, 1, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 1, 1, 1, 1, 0, 1, 1, 0, 0, 0, 1, 1, 1, 0, \ldots. \quad \square$$

Fact 6.59 establishes, under the assumption that R_1 produces a de Bruijn sequence (see Definition 6.40), that the output sequence of an alternating step generator satisfies the basic requirements of high period, high linear complexity, and good statistical properties.

6.59 Fact (*properties of the alternating step generator*) Suppose that R_1 produces a de Bruijn sequence of period 2^{L_1}. Furthermore, suppose that R_2 and R_3 are maximum-length LFSRs of lengths L_2 and L_3, respectively, such that $\gcd(L_2, L_3) = 1$. Let x be the output sequence of the alternating step generator formed by R_1, R_2, and R_3.

(i) The sequence x has period $2^{L_1} \cdot (2^{L_2} - 1) \cdot (2^{L_3} - 1)$.

(ii) The linear complexity $L(x)$ of x satisfies

$$(L_2 + L_3) \cdot 2^{L_1 - 1} < L(x) \leq (L_2 + L_3) \cdot 2^{L_1}.$$

(iii) The distribution of patterns in x is almost uniform. More precisely, let P be any binary string of length t bits, where $t \leq \min(L_2, L_3)$. If $x(t)$ denotes any t consecutive bits in x, then the probability that $x(t) = P$ is $\left(\frac{1}{2}\right)^t + O(1/2^{L_2 - t}) + O(1/2^{L_3 - t})$.

Since a de Bruijn sequence can be obtained from the output sequence s of a maximum-length LFSR (of length L) by simply adding a 0 to the end of each subsequence of $L-1$ 0's occurring in s (see Note 6.43), it is reasonable to expect that the assertions of high period, high linear complexity, and good statistical properties in Fact 6.59 also hold when R_1 is a maximum-length LFSR. Note, however, that this has not yet been proven.

6.60 Note (*security of the alternating step generator*) The LFSRs R_1, R_2, R_3 should be chosen to be maximum-length LFSRs whose lengths L_1, L_2, L_3 are pairwise relatively prime: $\gcd(L_1, L_2) = 1$, $\gcd(L_2, L_3) = 1$, $\gcd(L_1, L_3) = 1$. Moreover, the lengths should be about the same. If $L_1 \approx l$, $L_2 \approx l$, and $L_3 \approx l$, the best known attack on the alternating step generator is a divide-and-conquer attack on the control register R_1 which takes approximately 2^l steps. Thus, if $l \approx 128$, the generator is secure against all presently known attacks.

(ii) The shrinking generator

The shrinking generator is a relatively new keystream generator, having been proposed in 1993. Nevertheless, due to its simplicity and provable properties, it is a promising candidate for high-speed encryption applications. In the shrinking generator, a control LFSR R_1 is used to select a portion of the output sequence of a second LFSR R_2. The keystream produced is, therefore, a *shrunken* version (also known as an *irregularly decimated subsequence*) of the output sequence of R_2, as specified in Algorithm 6.61 and depicted in Figure 6.13.

6.61 Algorithm Shrinking generator

SUMMARY: a control LFSR R_1 is used to control the output of a second LFSR R_2.
The following steps are repeated until a keystream of desired length is produced.

1. Registers R_1 and R_2 are clocked.
2. If the output of R_1 is 1, the output bit of R_2 forms part of the keystream.
3. If the output of R_1 is 0, the output bit of R_2 is discarded.

More formally, let the output sequences of LFSRs R_1 and R_2 be $a_0, a_1, a_2, \ldots$ and $b_0, b_1, b_2, \ldots$, respectively. Then the keystream produced by the shrinking generator is $x_0, x_1, x_2, \ldots$, where $x_j = b_{i_j}$, and, for each $j \geq 0$, i_j is the position of the j^{th} 1 in the sequence $a_0, a_1, a_2, \ldots$.

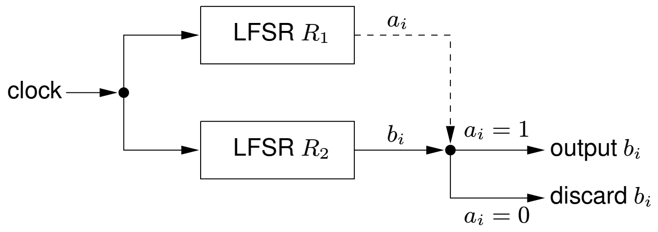

Figure 6.13: *The shrinking generator.*

6.62 Example (*shrinking generator with artificially small parameters*) Consider a shrinking generator with component LFSRs $R_1 = \langle 3, 1 + D + D^3 \rangle$ and $R_2 = \langle 5, 1 + D^3 + D^5 \rangle$. Suppose that the initial states of R_1 and R_2 are $[1, 0, 0]$ and $[0, 0, 1, 0, 1]$, respectively. The output sequence of R_1 is the 7-periodic sequence with cycle

$$a^7 = 0, 0, 1, 1, 1, 0, 1,$$

while the output sequence of R_2 is the 31-periodic sequence with cycle

$$b^{31} = 1,0,1,0,0,0,0,1,0,0,1,0,1,1,0,0,1,1,1,1,1,0,0,0,1,1,0,1,1,1,0.$$

The keystream generated is

$$x = 1,0,0,0,0,1,0,1,1,1,1,1,0,1,1,1,0,\ldots. \qquad \square$$

Fact 6.63 establishes that the output sequence of a shrinking generator satisfies the basic requirements of high period, high linear complexity, and good statistical properties.

6.63 Fact (*properties of the shrinking generator*) Let R_1 and R_2 be maximum-length LFSRs of lengths L_1 and L_2, respectively, and let x be an output sequence of the shrinking generator formed by R_1 and R_2.

(i) If $\gcd(L_1, L_2) = 1$, then x has period $(2^{L_2} - 1) \cdot 2^{L_1 - 1}$.

(ii) The linear complexity $L(x)$ of x satisfies

$$L_2 \cdot 2^{L_1-2} < L(x) \le L_2 \cdot 2^{L_1-1}.$$

(iii) Suppose that the connection polynomials for R_1 and R_2 are chosen uniformly at random from the set of all primitive polynomials of degrees L_1 and L_2 over $\mathbb{Z}_2$. Then the distribution of patterns in x is almost uniform. More precisely, if P is any binary string of length t bits and $x(t)$ denotes any t consecutive bits in x, then the probability that $x(t) = P$ is $(\frac{1}{2})^t + O(t/2^{L_2})$.

6.64 Note (*security of the shrinking generator*) Suppose that the component LFSRs R_1 and R_2 of the shrinking generator have lengths L_1 and L_2, respectively. If the connection polynomials for R_1 and R_2 are known (but not the initial contents of R_1 and R_2), the best attack known for recovering the secret key takes $O(2^{L_1} \cdot L_2^3)$ steps. On the other hand, if secret (and variable) connection polynomials are used, the best attack known takes $O(2^{2L_1} \cdot L_1 \cdot L_2)$ steps. There is also an attack through the linear complexity of the shrinking generator which takes $O(2^{L_1} \cdot L_2^2)$ steps (regardless of whether the connections are known or secret), but this attack requires $2^{L_1} \cdot L_2$ consecutive bits from the output sequence and is, therefore, infeasible for moderately large L_1 and L_2. For maximum security, R_1 and R_2 should be maximum-length LFSRs, and their lengths should satisfy $\gcd(L_1, L_2) = 1$. Moreover, secret connections should be used. Subject to these constraints, if $L_1 \approx l$ and $L_2 \approx l$, the shrinking generator has a security level approximately equal to 2^{2l}. Thus, if $L_1 \approx 64$ and $L_2 \approx 64$, the generator appears to be secure against all presently known attacks.

6.4 Other stream ciphers

While the LFSR-based stream ciphers discussed in §6.3 are well-suited to hardware implementation, they are not especially amenable to software implementation. This has led to several recent proposals for stream ciphers designed particularly for fast software implementation. Most of these proposals are either proprietary, or are relatively new and have not received sufficient scrutiny from the cryptographic community; for this reason, they are not presented in this section, and instead only mentioned in the chapter notes on page 222.

Two promising stream ciphers specifically designed for fast software implementation are SEAL and RC4. SEAL is presented in §6.4.1. RC4 is used in commercial products, and has a variable key-size, but it remains proprietary and is not presented here. Two other widely used stream ciphers not based on LFSRs are the Output Feedback (OFB; see §7.2.2(iv)) and Cipher Feedback (CFB; see §7.2.2(iii)) modes of block ciphers. Another class of keystream generators not based on LFSRs are those whose security relies on the intractability of an underlying number-theoretic problem; these generators are much slower than those based on LFSRs and are discussed in §5.5.

6.4.1 SEAL

SEAL (Software-optimized Encryption Algorithm) is a binary additive stream cipher (see Definition 6.4) that was proposed in 1993. Since it is relatively new, it has not yet received much scrutiny from the cryptographic community. However, it is presented here because it is one of the few stream ciphers that was specifically designed for efficient software implementation and, in particular, for 32-bit processors.

SEAL is a length-increasing pseudorandom function which maps a 32-bit *sequence number* n to an L-bit keystream under control of a 160-bit secret key a. In the preprocessing stage (step 1 of Algorithm 6.68), the key is stretched into larger tables using the table-generation function G_a specified in Algorithm 6.67; this function is based on the Secure Hash Algorithm SHA-1 (Algorithm 9.53). Subsequent to this preprocessing, keystream generation requires about 5 machine instructions per byte, and is an order of magnitude faster than DES (Algorithm 7.82).

The following notation is used in SEAL for 32-bit quantities A, B, C, D, X_i, and Y_j:

- $\overline{A}$: bitwise complement of A
- $A \wedge B$, $A \vee B$, $A \oplus B$: bitwise AND, inclusive-OR, exclusive-OR
- "$A \hookleftarrow s$": 32-bit result of rotating A left through s positions
- "$A \hookrightarrow s$": 32-bit result of rotating A right through s positions
- $A + B$: mod 2^{32} sum of the unsigned integers A and B
- $f(B,C,D) \stackrel{\text{def}}{=} (B \wedge C) \vee (\overline{B} \wedge D)$; $g(B,C,D) \stackrel{\text{def}}{=} (B \wedge C) \vee (B \wedge D) \vee (C \wedge D)$; $h(B,C,D) \stackrel{\text{def}}{=} B \oplus C \oplus D$
- $A \| B$: concatenation of A and B
- $(X_1, \dots, X_j) \leftarrow (Y_1, \dots, Y_j)$: simultaneous assignments $(X_i \leftarrow Y_i)$, where $(Y_1, \dots, Y_j)$ is evaluated prior to any assignments.

6.65 Note (*SEAL 1.0 vs. SEAL 2.0*) The table-generation function (Algorithm 6.67) for the first version of SEAL (SEAL 1.0) was based on the Secure Hash Algorithm (SHA). SEAL 2.0 differs from SEAL 1.0 in that the table-generation function for the former is based on the modified Secure Hash Algorithm SHA-1 (Algorithm 9.53).

6.66 Note (*tables*) The table generation (step 1 of Algorithm 6.68) uses the compression function of SHA-1 to expand the secret key a into larger tables T, S, and R. These tables can be precomputed, but only after the secret key a has been established. Tables T and S are 2K bytes and 1K byte in size, respectively. The size of table R depends on the desired bitlength L of the keystream — each 1K byte of keystream requires 16 bytes of R.

6.67 Algorithm Table-generation function for SEAL 2.0

$G_a(i)$
INPUT: a 160-bit string a and an integer i, $0 \le i < 2^{32}$.
OUTPUT: a 160-bit string, denoted $G_a(i)$.

1. *Definition of constants*. Define four 32-bit constants (in hex): $y_1 = $ 0x5a827999, $y_2 = $ 0x6ed9eba1, $y_3 = $ 0x8f1bbcdc, $y_4 = $ 0xca62c1d6.
2. *Table-generation function.*
   (*initialize* 80 *32-bit words* $X_0, X_1, \dots, X_{79}$)
   Set $X_0 \leftarrow i$. For j from 1 to 15 do: $X_j \leftarrow$ 0x00000000.
   For j from 16 to 79 do: $X_j \leftarrow ((X_{j-3} \oplus X_{j-8} \oplus X_{j-14} \oplus X_{j-16}) \hookleftarrow 1)$.
   (*initialize working variables*)
   Break up the 160-bit string a into five 32-bit words: $a = H_0H_1H_2H_3H_4$.
   $(A, B, C, D, E) \leftarrow (H_0, H_1, H_2, H_3, H_4)$.
   (*execute four rounds of 20 steps, then update; t is a temporary variable*)
   (*Round 1*) For j from 0 to 19 do the following:
   $t \leftarrow ((A \hookleftarrow 5) + f(B, C, D) + E + X_j + y_1)$,
   $(A, B, C, D, E) \leftarrow (t, A, B \hookleftarrow 30, C, D)$.
   (*Round 2*) For j from 20 to 39 do the following:
   $t \leftarrow ((A \hookleftarrow 5) + h(B, C, D) + E + X_j + y_2)$,
   $(A, B, C, D, E) \leftarrow (t, A, B \hookleftarrow 30, C, D)$.
   (*Round 3*) For j from 40 to 59 do the following:
   $t \leftarrow ((A \hookleftarrow 5) + g(B, C, D) + E + X_j + y_3)$,
   $(A, B, C, D, E) \leftarrow (t, A, B \hookleftarrow 30, C, D)$.
   (*Round 4*) For j from 60 to 79 do the following:
   $t \leftarrow ((A \hookleftarrow 5) + h(B, C, D) + E + X_j + y_4)$,
   $(A, B, C, D, E) \leftarrow (t, A, B \hookleftarrow 30, C, D)$.
   (*update chaining values*)
   $(H_0, H_1, H_2, H_3, H_4) \leftarrow (H_0 + A, H_1 + B, H_2 + C, H_3 + D, H_4 + E)$.
   (*completion*) The value of $G_a(i)$ is the 160-bit string $H_0 \| H_1 \| H_2 \| H_3 \| H_4$.

6.68 Algorithm Keystream generator for SEAL 2.0

SEAL(a,n)
INPUT: a 160-bit string a (the secret key), a (non-secret) integer n, $0 \le n < 2^{32}$ (the sequence number), and the desired bitlength L of the keystream.
OUTPUT: keystream y of bitlength L', where L' is the least multiple of 128 which is $\ge L$.

1. *Table generation*. Generate the tables T, S, and R, whose entries are 32-bit words. The function F used below is defined by $F_a(i) = H^i_{i \bmod 5}$, where $H^i_0H^i_1H^i_2H^i_3H^i_4 = G_a(\lfloor i/5 \rfloor)$, and where the function G_a is defined in Algorithm 6.67.
   1.1 For i from 0 to 511 do the following: $T[i] \leftarrow F_a(i)$.
   1.2 For j from 0 to 255 do the following: $S[j] \leftarrow F_a(\text{0x00001000} + j)$.
   1.3 For k from 0 to $4 \cdot \lceil (L-1)/8192 \rceil - 1$ do: $R[k] \leftarrow F_a(\text{0x00002000} + k)$.
2. *Initialization procedure.* The following is a description of the subroutine `Initialize`$(n, l, A, B, C, D, n_1, n_2, n_3, n_4)$ which takes as input a 32-bit word n and an integer l, and outputs eight 32-bit words A, B, C, D, n_1, n_2, n_3, and n_4. This subroutine is used in step 4.
   $A \leftarrow n \oplus R[4l]$, $B \leftarrow (n \hookrightarrow 8) \oplus R[4l+1]$, $C \leftarrow (n \hookrightarrow 16) \oplus R[4l+2]$,
   $D \leftarrow (n \hookrightarrow 24) \oplus R[4l+3]$.

For j from 1 to 2 do the following:

$P \leftarrow A \wedge 0\text{x}000007\text{fc},\ B \leftarrow B + T[P/4],\ A \leftarrow (A \hookrightarrow 9),$
$P \leftarrow B \wedge 0\text{x}000007\text{fc},\ C \leftarrow C + T[P/4],\ B \leftarrow (B \hookrightarrow 9),$
$P \leftarrow C \wedge 0\text{x}000007\text{fc},\ D \leftarrow D + T[P/4],\ C \leftarrow (C \hookrightarrow 9),$
$P \leftarrow D \wedge 0\text{x}000007\text{fc},\ A \leftarrow A + T[P/4],\ D \leftarrow (D \hookrightarrow 9).$

$(n_1, n_2, n_3, n_4) \leftarrow (D, B, A, C).$
$P \leftarrow A \wedge 0\text{x}000007\text{fc},\ B \leftarrow B + T[P/4],\ A \leftarrow (A \hookrightarrow 9).$
$P \leftarrow B \wedge 0\text{x}000007\text{fc},\ C \leftarrow C + T[P/4],\ B \leftarrow (B \hookrightarrow 9).$
$P \leftarrow C \wedge 0\text{x}000007\text{fc},\ D \leftarrow D + T[P/4],\ C \leftarrow (C \hookrightarrow 9).$
$P \leftarrow D \wedge 0\text{x}000007\text{fc},\ A \leftarrow A + T[P/4],\ D \leftarrow (D \hookrightarrow 9).$

3. Initialize y to be the empty string, and $l \leftarrow 0$.
4. Repeat the following:
   4.1 Execute the procedure `Initialize`$(n, l, A, B, C, D, n_1, n_2, n_3, n_4)$.
   4.2 For i from 1 to 64 do the following:
   $P \leftarrow A \wedge 0\text{x}000007\text{fc},\ B \leftarrow B + T[P/4],\ A \leftarrow (A \hookrightarrow 9),\ B \leftarrow B \oplus A,$
   $Q \leftarrow B \wedge 0\text{x}000007\text{fc},\ C \leftarrow C \oplus T[Q/4],\ B \leftarrow (B \hookrightarrow 9),\ C \leftarrow C + B,$
   $P \leftarrow (P + C) \wedge 0\text{x}000007\text{fc},\ D \leftarrow D + T[P/4],\ C \leftarrow (C \hookrightarrow 9),\ D \leftarrow D \oplus C,$
   $Q \leftarrow (Q + D) \wedge 0\text{x}000007\text{fc},\ A \leftarrow A \oplus T[Q/4],\ D \leftarrow (D \hookrightarrow 9),\ A \leftarrow A + D,$
   $P \leftarrow (P + A) \wedge 0\text{x}000007\text{fc},\ B \leftarrow B \oplus T[P/4],\ A \leftarrow (A \hookrightarrow 9),$
   $Q \leftarrow (Q + B) \wedge 0\text{x}000007\text{fc},\ C \leftarrow C + T[Q/4],\ B \leftarrow (B \hookrightarrow 9),$
   $P \leftarrow (P + C) \wedge 0\text{x}000007\text{fc},\ D \leftarrow D \oplus T[P/4],\ C \leftarrow (C \hookrightarrow 9),$
   $Q \leftarrow (Q + D) \wedge 0\text{x}000007\text{fc},\ A \leftarrow A + T[Q/4],\ D \leftarrow (D \hookrightarrow 9),$
   $y \leftarrow y \,\|\, (B + S[4i-4]) \,\|\, (C \oplus S[4i-3]) \,\|\, (D + S[4i-2]) \,\|\, (A \oplus S[4i-1]).$
   If y is $\geq L$ bits in length then return(y) and stop.
   If i is odd, set $(A, C) \leftarrow (A + n_1, C + n_2)$. Otherwise, $(A, C) \leftarrow (A + n_3, C + n_4)$.
   4.3 Set $l \leftarrow l + 1$.

6.69 Note (*choice of parameter L*) In most applications of SEAL 2.0 it is expected that $L \leq 2^{19}$; larger values of L are permissible, but come at the expense of a larger table R. A preferred method for generating a longer keystream without requiring a larger table R is to compute the concatenation of the keystreams SEAL(a,0), SEAL(a,1), SEAL(a,2),.... Since the sequence number is $n < 2^{32}$, a keystream of length up to 2^{51} bits can be obtained in this manner with $L = 2^{19}$.

6.70 Example (*test vectors for SEAL 2.0*) Suppose the key a is the 160-bit (hexadecimal) string

```
67452301 efcdab89 98badcfe 10325476 c3d2e1f0,
```

$n =$ 0x013577af, and $L = 32768$ bits. Table R consists of words $R[0], R[1], \ldots, R[15]$:

```
5021758d ce577c11 fa5bd5dd 366d1b93 182cff72 ac06d7c6
2683ead8 fabe3573 82a10c96 48c483bd ca92285c 71fe84c0
bd76b700 6fdcc20c 8dada151 4506dd64
```

The table T consists of words $T[0], T[1], \ldots, R[511]$:

```
92b404e5 56588ced 6c1acd4e bf053f68 09f73a93 cd5f176a
b863f14e 2b014a2f 4407e646 38665610 222d2f91 4d941a21
........ ........ ........ ........ ........ ........
3af3a4bf 021e4080 2a677d95 405c7db0 338e4b1e 19ccf158
```

The table S consists of words $S[0], S[1], \ldots, S[255]$:

```
907c1e3d ce71ef0a 48f559ef 2b7ab8bc 4557f4b8 033e9b05
4fde0efa 1a845f94 38512c3b d4b44591 53765dce 469efa02
........ ........ ........ ........ ........ ........
bd7dea87 fd036d87 53aa3013 ec60e282 1eaef8f9 0b5a0949
```

The output y of Algorithm 6.68 consists of 1024 words $y[0], y[1], \ldots, y[1023]$:

```
37a00595 9b84c49c a4be1e05 0673530f 0ac8389d c5878ec8
da6666d0 6da71328 1419bdf2 d258bebb b6a42a4d 8a311a72
........ ........ ........ ........ ........ ........
547dfde9 668d50b5 ba9e2567 413403c5 43120b5a ecf9d062
```

The XOR of the 1024 words of y is 0x098045fc. □

6.5 Notes and further references

§6.1

Although now dated, Rueppel [1075] provides a solid introduction to the analysis and design of stream ciphers. For an updated and more comprehensive survey, see Rueppel [1081]. Another recommended survey is that of Robshaw [1063].

The concept of unconditional security was introduced in the seminal paper by Shannon [1120]. Maurer [819] surveys the role of information theory in cryptography and, in particular, secrecy, authentication, and secret sharing schemes. Maurer [811] devised a *randomized stream cipher* that is unconditionally secure "with high probability". More precisely, an adversary is unable to obtain any information whatsoever about the plaintext with probability arbitrarily close to 1, unless the adversary can perform an infeasible computation. The cipher utilizes a publicly-accessible source of random bits whose length is much greater than that of all the plaintext to be encrypted, and can conceivably be made practical. Maurer's cipher is based on the impractical *Rip van Winkle cipher* of Massey and Ingermarsson [789], which is described by Rueppel [1081].

One technique for solving the re-synchronization problem with synchronous stream ciphers is to have the receiver send a resynchronization request to the sender, whereby a new internal state is computed as a (public) function of the original internal state (or key) and some public information (such as the time at the moment of the request). Daemen, Govaerts, and Vandewalle [291] showed that this approach can result in a total loss of security for some published stream cipher proposals. Proctor [1011] considered the trade-off between the security and error propagation problems that arise by varying the number of feedback ciphertext digits. Maurer [808] presented various design approaches for self-synchronizing stream ciphers that are potentially superior to designs based on block ciphers, both with respect to encryption speed and security.

§6.2

An excellent introduction to the theory of both linear and nonlinear shift registers is the book by Golomb [498]; see also Selmer [1107], Chapters 5 and 6 of Beker and Piper [84], and Chapter 8 of Lidl and Niederreiter [764]. A lucid treatment of m-sequences can be found in Chapter 10 of McEliece [830]. While the discussion in this chapter has been restricted to sequences and feedback shift registers over the binary field $\mathbb{Z}_2$, many of the results presented can be generalized to sequences and feedback shift registers over any finite field $\mathbb{F}_q$.

The results on the expected linear complexity and linear complexity profile of random sequences (Facts 6.21, 6.22, 6.24, and 6.25) are from Chapter 4 of Rueppel [1075]; they also appear in Rueppel [1077]. Dai and Yang [294] extended Fact 6.22 and obtained bounds for the expected linear complexity of an n-periodic sequence for each possible value of n. The bounds imply that the expected linear complexity of a random periodic sequence is close to the period of the sequence. The linear complexity profile of the sequence defined in Example 6.27 was established by Dai [293]. For further theoretical analysis of the linear complexity profile, consult the work of Niederreiter [927, 928, 929, 930].

Facts 6.29 and 6.34 are due to Massey [784]. The Berlekamp-Massey algorithm (Algorithm 6.30) is due to Massey [784], and is based on an earlier algorithm of Berlekamp [118] for decoding BCH codes. While the algorithm in §6.2.3 is only described for binary sequences, it can be generalized to find the linear complexity of sequences over any field. Further discussion and refinements of the Berlekamp-Massey algorithm are given by Blahut [144]. There are numerous other algorithms for computing the linear complexity of a sequence. For example, Games and Chan [439] and Robshaw [1062] present efficient algorithms for determining the linear complexity of binary sequences of period 2^n; these algorithms have limited practical use since they require an entire cycle of the sequence.

Jansen and Boekee [632] defined the *maximum order complexity* of a sequence to be the length of the shortest (not necessarily linear) feedback shift register (FSR) that can generate the sequence. The expected maximum order complexity of a random binary sequence of length n is approximately $2\lg n$. An efficient linear-time algorithm for computing this complexity measure was also presented; see also Jansen and Boekee [631].

Another complexity measure, the *Ziv-Lempel complexity measure*, was proposed by Ziv and Lempel [1273]. This measure quantifies the rate at which new patterns appear in a sequence. Mund [912] used a heuristic argument to derive the expected Ziv-Lempel complexity of a random binary sequence of a given length. For a detailed study of the relative strengths and weaknesses of the linear, maximum order, and Ziv-Lempel complexity measures, see Erdmann [372].

Kolmogorov [704] and Chaitin [236] introduced the notion of so-called *Turing-Kolmogorov -Chaitin complexity*, which measures the minimum size of the input to a fixed universal Turing machine which can generate a given sequence; see also Martin-Löf [783]. While this complexity measure is of theoretical interest, there is no algorithm known for computing it and, hence, it has no apparent practical significance. Beth and Dai [124] have shown that the Turing-Kolmogorov-Chaitin complexity is approximately twice the linear complexity for most sequences of sufficient length.

Fact 6.39 is due to Golomb and Welch, and appears in the book of Golomb [498, p.115]. Lai [725] showed that Fact 6.39 is only true for the binary case, and established necessary and sufficient conditions for an FSR over a general finite field to be nonsingular.

Klapper and Goresky [677] introduced a new type of feedback register called a *feedback with carry shift register* (FCSR), which is equipped with auxiliary memory for storing the (integer) carry. An FCSR is similar to an LFSR (see Figure 6.4), except that the contents of the tapped stages of the shift register are added *as integers* to the current content of the memory to form a sum S. The least significant bit of S (i.e., $S \bmod 2$) is then fed back into the first (leftmost) stage of the shift register, while the remaining higher order bits (i.e., $\lfloor S/2 \rfloor$) are retained as the new value of the memory. If the FCSR has L stages, then the space required for the auxiliary memory is at most $\lg L$ bits. FCSRs can be conveniently analyzed using the algebra over the 2-adic numbers just as the algebra over finite fields is used to analyze LFSRs.

Any periodic binary sequence can be generated by a FCSR. The 2-*adic span* of a periodic sequence is the number of stages and memory bits in the smallest FCSR that generates the sequence. Let s be a periodic sequence having a 2-adic span of T; note that T is no more than the period of s. Klapper and Goresky [678] presented an efficient algorithm for finding an FCSR of length T which generates s, given $2T + 2\lceil \lg T \rceil + 4$ of the initial bits of s. A comprehensive treatment of FCSRs and the 2-adic span is given by Klapper and Goresky [676].

§6.3

Notes 6.46 and 6.47 on the selection of connection polynomials were essentially first pointed out by Meier and Staffelbach [834] and Chepyzhov and Smeets [256] in relation to fast correlation attacks on regularly clocked LFSRs. Similar observations were made by Coppersmith, Krawczyk, and Mansour [279] in connection with the shrinking generator. More generally, to withstand sophisticated correlation attacks (e.g., see Meier and Staffelbach [834]), the connection polynomials should not have low-weight polynomial multiples whose degrees are not sufficiently large.

Klapper [675] provides examples of binary sequences having high linear complexity, but whose linear complexity is low when considered as sequences (whose elements happen to be only 0 or 1) over a larger finite field. This demonstrates that high linear complexity (over $\mathbb{Z}_2$) by itself is inadequate for security. Fact 6.49 was proven by Rueppel and Staffelbach [1085].

The Geffe generator (Example 6.50) was proposed by Geffe [446]. The *Pless generator* (Arrangement D of [978]) was another early proposal for a nonlinear combination generator, and uses four J-K flip-flops to combine the output of eight LFSRs. This generator also succumbs to a divide-and-conquer attack, as was demonstrated by Rubin [1074].

The *linear syndrome attack* of Zeng, Yang, and Rao [1265] is a known-plaintext attack on keystream generators, and is based on earlier work of Zeng and Huang [1263]. It is effective when the known keystream B can be written in the form $B = A \oplus X$, where A is the output sequence of an LFSR with known connection polynomial, and the sequence X is unknown but sparse in the sense that it contains more 0's than 1's. If the connection polynomials of the Geffe generator are all known to an adversary, and are primitive trinomials of degrees not exceeding n, then the initial states of the three component LFSRs (i.e., the secret key) can be efficiently recovered from a known keystream segment of length $37n$ bits.

The correlation attack (Note 6.51) on nonlinear combination generators was first developed by Siegenthaler [1133], and estimates were given for the length of the observed keystream required for the attack to succeed with high probability. The importance of correlation immunity to nonlinear combining functions was pointed out by Siegenthaler [1132], who showed the tradeoff between high correlation immunity and high nonlinear order (Fact 6.53). Meier and Staffelbach [834] presented two new so-called *fast correlation attacks* which are more efficient than Siegenthaler's attack in the case where the component LFSRs have sparse feedback polynomials, or if they have low-weight polynomial multiples (e.g., each having fewer than 10 non-zero terms) of not too large a degree. Further extensions and refinements of correlation attacks can be found in the papers of Mihaljević and Golić [874], Chepyzhov and Smeets [256], Golić and Mihaljević [491], Mihaljević and J. Golić [875], Mihaljević [873], Clark, Golić, and Dawson [262], and Penzhorn and Kühn [967]. A comprehensive survey of correlation attacks on LFSR-based stream ciphers is the paper by Golić [486]; the cases where the combining function is memoryless or with memory, as well as when the LFSRs are clocked regularly or irregularly, are all considered.

The summation generator (Example 6.54) was proposed by Rueppel [1075, 1076]. Meier and Staffelbach [837] presented correlation attacks on combination generators having memory, cracked the summation generator having only two component LFSRs, and as a result recommended using several LFSRs of moderate lengths rather than just a few long LFSRs in the summation generator. As an example, if a summation generator employs two LFSRs each having length approximately 200, and if 50 000 keystream bits are known, then Meier and Staffelbach's attack is expected to take less than 700 trials, where the dominant step in each trial involves solving a 400×400 system of binary linear equations. Dawson [312] presented another known-plaintext attack on summation generators having two component LFSRs, which requires fewer known keystream bits than Meier and Staffelbach's attack. Dawson's attack is only faster than that of Meier and Staffelbach in the case where both LFSRs are relatively short. Recently, Klapper and Goresky [678] showed that the summation generator has comparatively low 2-adic span (see page 218). More precisely, if a and b are two sequences of 2-adic span $\lambda_2(a)$ and $\lambda_2(b)$, respectively, and if s is the result of combining them with the summation generator, then the 2-adic span of s is at most $\lambda_2(a) + \lambda_2(b) + 2\lceil \lg(\lambda_2(a))\rceil + 2\lceil \lg(\lambda_2(b))\rceil + 6$. For example, if m-sequences of period $2^L - 1$ for $L = 7, 11, 13, 15, 16, 17$ are combined with the summation generator, then the resulting sequence has linear complexity nearly 2^{79}, but the 2-adic span is less than 2^{18}. Hence, the summation generator is vulnerable to a known-plaintext attack when the component LFSRs are all relatively short.

The probability distribution of the carry for addition of n random integers was analyzed by Staffelbach and Meier [1167]. It was proven that the carry is balanced for even n and biased for odd n. For $n = 3$ the carry is strongly biased, however, the bias converges to 0 as n tends to ∞. Golić [485] pointed out the importance of the correlation between linear functions of the output and input in general combiners with memory, and introduced the so-called *linear sequential circuit approximation method* for finding such functions that produce correlated sequences. Golić [488] used this as a basis for developing a *linear cryptanalysis* technique for stream ciphers, and in the same paper proposed a stream cipher called GOAL, incorporating principles of modified truncated linear congruential generators (see page 187), self-clock-control, and randomly generated combiners with memory.

Fact 6.55(i) is due to Key [670], while Fact 6.55(ii) was proven by Rueppel [1075]. Massey and Serconek [794] gave an alternate proof of Key's bound that is based on the Discrete Fourier Transform. Siegenthaler [1134] described a correlation attack on nonlinear filter generators. Forré [418] has applied fast correlation attacks to such generators. Anderson [29] demonstrated other correlations which may be useful in improving the success of correlation attacks. An attack called the *inversion attack*, proposed by Golić [490], may be more effective than Anderson's attack. Golić also provides a list of design criteria for nonlinear filter generators. Ding [349] introduced the notion of differential cryptanalysis for nonlinear filter generators where the LFSR is replaced by a simple counter having arbitrary period.

The *linear consistency attack* of Zeng, Yang, and Rao [1264] is a known-plaintext attack on keystream generators which can discover key redundancies in various generators. It is effective in situations where it is possible to single out a certain portion k_1 of the secret key k, and form a linear system of equations $Ax = b$ where the matrix A is determined by k_1, and b is determined from the known keystream. The system of equations should have the property that it is consistent (and with high probability has a unique solution) if k_1 is the true value of the subkey, while it is inconsistent with high probability otherwise. In these circumstances, one can mount an exhaustive search for k_1, and subsequently mount a separate attack for the remaining bits of k. If the bitlengths of k_1 and k are l_1 and l, respectively, the attack demonstrates that the security level of the generator is $2^{l_1} + 2^{l-l_1}$, rather than 2^l.

The *multiplexer generator* was proposed by Jennings [637]. Two maximum-length LFSRs having lengths L_1, L_2 that are relatively prime are employed. Let h be a positive integer satisfying $h \leq \min(L_1, \lg L_2)$. After each clock cycle, the contents of a fixed subset of h stages of the first LFSR are selected, and converted to an integer t in the interval $[0, L_2 - 1]$ using a $1 - 1$ mapping θ. Finally, the content of stage t of the second LFSR is output as part of the keystream. Assuming that the connection polynomials of the LFSRs are known, the linear consistency attack provides a known-plaintext attack on the multiplexer generator requiring a known keystream sequence of length $N \geq L_1 + L_2 2^h$ and 2^{L_1+h} linear consistency tests. This demonstrates that the choice of the mapping θ and the second LFSR do not contribute significantly to the security of the generator.

The linear consistency attack has also been considered by Zeng, Yang, and Rao [1264] for the *multispeed inner-product generator* of Massey and Rueppel [793]. In this generator, two LFSRs of lengths L_1 and L_2 are clocked at different rates, and their contents combined at the lower clock rate by taking the inner-product of the $\min(L_1, L_2)$ stages of the two LFSRs. The paper by Zeng et al. [1266] is a readable survey describing the effectiveness of the linear consistency and linear syndrome attacks in cryptanalyzing stream ciphers.

The knapsack generator (Example 6.56) was proposed by Rueppel and Massey [1084] and extensively analyzed by Rueppel [1075], however, no concrete suggestions on selecting appropriate parameters (the length L of the LFSR and the knapsack weights) for the generator were given. No weaknesses of the knapsack generator have been reported in the literature.

The idea of using the output of a register to control the stepping of another register was used in several rotor machines during the second world war, for example, the German Lorenz SZ40 cipher. A description of this cipher, and also an extensive survey of clock-controlled shift registers, is provided by Gollmann and Chambers [496].

The alternating step generator (Algorithm 6.57) was proposed in 1987 by Günther [528], who also proved Fact 6.59 and described the divide-and-conquer attack mentioned in Note 6.60. The alternating step generator is based on the *stop-and-go* generator of Beth and Piper [126]. In the stop-and-go generator, a control register R_1 is used to control the stepping of another register R_2 as follows. If the output of R_1 is 1, then R_2 is clocked; if the output of R_1 is 0, then R_2 is not clocked, however, its previous output is repeated. The output of R_2 is then XORed with the output sequence of a third register R_3 which is clocked at the same rate as R_1. Beth and Piper showed how a judicious choice of registers R_1, R_2, and R_3 can guarantee that the output sequence has high linear complexity and period, and good statistical properties. Unfortunately, the generator succumbs to the linear syndrome attack of Zeng, Yang, and Rao [1265] (see also page 218): if the connection polynomials of R_1 and R_2 are primitive trinomials of degree not exceeding n, and known to the adversary, then the initial states of the three component LFSRs (i.e., the secret key) can be efficiently recovered from a known-plaintext segment of length $37n$ bits.

Another variant of the stop-and-go generator is the *step-1/step-2* generator due to Gollmann and Chambers [496]. This generator uses two maximum-length registers R_1 and R_2 of the same length. Register R_1 is used to control the stepping of R_2 as follows. If the output of R_1 is 0, then R_2 is clocked once; if the output of R_1 is 1, then R_2 is clocked twice before producing the next output bit. Živković [1274] proposed an *embedding correlation attack* on R_2 whose complexity of $O(2^{L_2})$, where L_2 is the length of R_2.

A *cyclic register* of length L is an LFSR with feedback polynomial $C(D) = 1 + D^L$. Gollmann [494] proposed *cascading* n cyclic registers of the same prime length p by arranging them serially in such a way that all except the first register are clock-controlled by their predecessors; the Gollmann *p-cycle cascade* can be viewed as an extension of the stop-and-go generator (page 220). The first register is clocked regularly, and its output bit is the input bit to the second register. In general, if the input bit to the i^{th} register (for $i \geq 2$) at time t is a_t, then the i^{th} register is clocked if $a_t = 1$; if $a_t = 0$, the register is not clocked but its previous output bit is repeated. The output bit of the i^{th} register is then XORed with a_t, and the result becomes the input bit to the $(i+1)^{\text{st}}$ register. The output of the last register is the output of the p-cycle cascade. The initial (secret) stage of a component cyclic register should not be the all-0's vector or the all-1's vector. Gollmann proved that the period of the output sequence is p^n. Moreover, if p is a prime such that 2 is a generator of $\mathbb{Z}_p^*$, then the output sequence has linear complexity p^n. This suggests very strongly using long cascades (i.e., n large) of shorter registers rather than short cascades of longer registers. A variant of the Gollmann cascade, called an *m-sequence cascade*, has the cyclic registers replaced by maximum-length LFSRs of the same length L. Chambers [237] showed that the output sequence of such an m-sequence cascade has period $(2^L-1)^n$ and linear complexity at least $L(2^L-1)^{n-1}$. Park, Lee, and Goh [964] extended earlier work of Menicocci [845] and reported breaking 9-stage m-sequence cascades where each LFSR has length 100; they also suggested that 10-stage m-sequence cascades may be insecure. Chambers and Gollmann [239] studied an attack on p-cycle and m-sequence cascades called *lock-in*, which results in a reduction in the effective key space of the cascades.

The shrinking generator (Algorithm 6.61) was proposed in 1993 by Coppersmith, Krawczyk, and Mansour [279], who also proved Fact 6.63 and described the attacks mentioned in Note 6.64. The irregular output rate of the shrinking generator can be overcome by using a short buffer for the output; the influence of such a buffer is analyzed by Kessler and Krawczyk [669]. Krawczyk [716] mentions some techniques for improving software implementations. A throughput of 2.5 Mbits/sec is reported for a C language implementation on a 33MHz IBM workstation, when the two shift registers each have lengths in the range 61–64 bits and secret connections are employed. The security of the shrinking generator is studied further by Golić [487].

A key generator related to the shrinking generator is the *self-shrinking generator* (SSG) of Meier and Staffelbach [838]. The self-shrinking generator uses only one maximum-length LFSR R. The output sequence of R is partitioned into pairs of bits. The SSG outputs a 0 if a pair is 10, and outputs a 1 if a pair is 11; 01 and 00 pairs are discarded. Meier and Staffelbach proved that the self-shrinking generator can be implemented as a shrinking generator. Moreover, the shrinking generator can be implemented as a self-shrinking generator (whose component LFSR is not maximum-length). More precisely, if the component LFSRs of a shrinking generator have connection polynomials $C_1(D)$ and $C_2(D)$, its output sequence can be produced by a self-shrinking generator with connection polynomial $C(D) = C_1(D)^2 \cdot C_2(D)^2$. Meier and Staffelbach also proved that if the length of R is L, then the period and linear complexity of the output sequence of the SSG are at least $2^{\lfloor L/2 \rfloor}$ and $2^{\lfloor L/2 \rfloor - 1}$, respectively. Moreover, they provided strong evidence that this period and linear complexity is in fact about 2^{L-1}. Assuming a randomly chosen, but known, connection polynomial, the best attack presented by Meier and Staffelbach on the SSG takes $2^{0.79L}$ steps. More recently, Mihaljević [871] presented a significantly faster probabilistic attack on the SSG. For example, if $L = 100$, then the new attack takes 2^{57} steps and requires a portion of the output sequence of length 4.9×10^8. The attack does not have an impact on the security of the shrinking generator.

A recent survey of techniques for attacking clock-controlled generators is given by Gollmann [495]. For some newer attack techniques, see Mihaljević [872], Golić and O'Connor [492], and Golić [489]. Chambers [238] proposed a clock-controlled cascade composed of LFSRs each of length 32. Each 32-bit portion of the output sequence of a component LFSR is passed through an invertible scrambler box (*S-box*), and the resulting 32-bit sequence is used to control the clock of the next LFSR. Baum and Blackburn [77] generalized the notion of a clock-controlled shift register to that of a register based on a finite group.

§6.4

SEAL (Algorithm 6.68) was designed and patented by Coppersmith and Rogaway [281]. Rogaway and Coppersmith [1066] report an encryption speed of 7.2 Mbytes/sec for an assembly language implementation on a 50 MHz 486 processor with $L = 4096$ bits, assuming precomputed tables (cf. Note 6.66).

Although the stream cipher RC4 remains proprietary, alleged descriptions have been published which are output compatible with certified implementations of RC4; for example, see Schneier [1094]. Blöcher and Dichtl [156] proposed a fast software stream cipher called *FISH* (Fibonacci Shrinking generator), which is based on the shrinking generator principle applied to the lagged Fibonacci generator (also known as the additive generator) of Knuth [692, p.27]. Anderson [28] subsequently presented a known-plaintext attack on FISH which requires a few thousand 32-bit words of known plaintext and a work factor of about 2^{40} computations. Anderson also proposed a fast software stream cipher called *PIKE* based on the Fibonacci generator and the stream cipher A5; a description of A5 is given by Anderson [28].

Wolfram [1251, 1252] proposed a stream cipher based on one-dimensional cellular automata with nonlinear feedback. Meier and Staffelbach [835] presented a known-plaintext attack on this cipher which demonstrated that key lengths of 127 bits suggested by Wolfram [1252] are insecure; Meier and Staffelbach recommend key sizes of about 1000 bits.

Klapper and Goresky [679] presented constructions for FCSRs (see page 217) whose output sequences have nearly maximal period, are balanced, and are nearly de Bruijn sequences in the sense that for any fixed non-negative integer t, the number of occurrences of any two t-bit sequences as subsequences of a period differs by at most 2. Such FCSRs are good candidates for usage in the construction of secure stream ciphers, just as maximum-length LFSRs were used in §6.3. Goresky and Klapper [518] introduced a generalization of FCSRs called d-FCSRs, based on *ramified* extensions of the 2-adic numbers (d is the ramification).