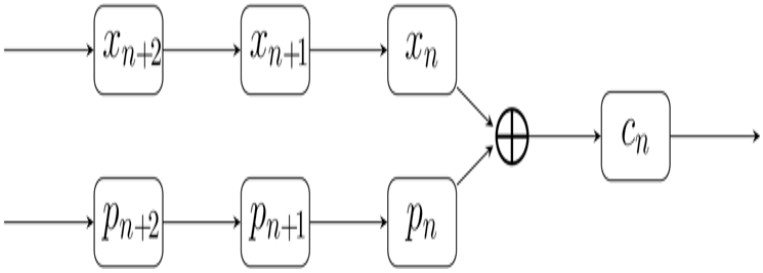
Chapter 5 Stream Ciphers

The one-time pad provides a strong form of secrecy, but since key transmission is difficult, it is desirable to devise substitutes that are easier to use. Stream ciphers are one way of achieving this goal. As in the one-time pad, the plaintext is written as a string of bits. Then a binary keystream is generated and XORed with the plaintext to produce the ciphertext.

Figure 5.1 Stream cipher encryption



p_n = plaintext bit, x_n = key bit, c_n = ciphertext bit.

Figure 5.1 Full Alternative Text

For the system to be secure, the keystream needs to approximate a random sequence, so we need a good source of random-looking bits. In Section 5.1, we discuss pseudorandom number generators. In Sections 5.2 and 5.3, we describe two commonly used stream ciphers and the pseudorandom number generators that they use. Although they have security weaknesses, they give an idea of methods that can be used.

In the next chapter, we discuss block ciphers and various modes of operations. Some of the most secure stream ciphers are actually good block ciphers used, for

example, in OFB or CTR mode. See Subsections 6.3.4 and 6.3.5.

There is one problem that is common to all stream ciphers that are obtained by XORing pseudorandom numbers with plaintext and is one of the reasons that authentication and message integrity checks are added to protect communications. Suppose Eve knows where the word “good” occurs in a plaintext that has been encrypted with a stream cipher. If she intercepts the ciphertext, she can XOR the bits for good $\oplus$ evil at the appropriate place in the ciphertext before continuing the transmission of the ciphertext. When the ciphertext is decrypted, “good” will be changed to “evil.” This type of attack was one of the weaknesses of the WEP system, which is discussed in Section 14.3.

5.1 Pseudorandom Bit Generation

The one-time pad and many other cryptographic applications require sequences of random bits. Before we can use a cryptographic algorithm, such as DES (Chapter 7) or AES (Chapter 8), it is necessary to generate a sequence of random bits to use as the key.

One way to generate random bits is to use natural randomness that occurs in nature. For example, the thermal noise from a semiconductor resistor is known to be a good source of randomness. However, just as flipping coins to produce random bits would not be practical for cryptographic applications, most natural conditions are not practical due to the inherent slowness in sampling the process and the difficulty of ensuring that an adversary does not observe the process. We would therefore like a method for generating randomness that can be done in software. Most computers have a method for generating random numbers that is readily available to the user. For example, the standard C library contains a function *rand()* that generates pseudorandom numbers between 0 and 65535. This pseudorandom function takes a **seed** as input and produces an output bitstream.

The *rand()* function and many other pseudorandom number generators are based on linear congruential generators. A **linear congruential generator** produces a sequence of numbers $x_1, x_2, \cdots$, where

$$x_n = ax_{n-1} + b \pmod{m}.$$

The number x_0 is the initial seed, while the numbers a, b, and m are parameters that govern the relationship. The

use of pseudorandom number generators based on linear congruential generators is suitable for experimental purposes, but is highly discouraged for cryptographic purposes. This is because they are predictable (even if the parameters a, b, and m are not known), in the sense that an eavesdropper can use knowledge of some bits to predict future bits with fairly high probability. In fact, it has been shown that any polynomial congruential generator is cryptographically insecure.

In cryptographic applications, we need a source of bits that is nonpredictable. We now discuss two ways to create such nonpredictable bits.

The first method uses one-way functions. These are functions $f(x)$ that are easy to compute but for which, given y, it is computationally infeasible to solve $y = f(x)$ for x. Suppose that we have such a one-way function f and a random seed s. Define $x_j = f(s + j)$ for $j = 1, 2, 3, \ldots$. If we let b_j be the least significant bit of x_j, then the sequence $b_0, b_1, \cdots$ will often be a pseudorandom sequence of bits (but see Exercise 14). This method of random bit generation is often used, and has proven to be very practical. Two popular choices for the one-way function are DES (Chapter 7) and SHA, the Secure Hash Algorithm (Chapter 11). As an example, the cryptographic pseudorandom number generator in the OpenSSL toolkit (used for secure communications over the Internet) is based on SHA.

Another method for generating random bits is to use an intractable problem from number theory. One of the most popular cryptographically secure pseudorandom number generators is the **Blum-Blum-Shub (BBS) pseudorandom bit generator**, also known as the quadratic residue generator. In this scheme, one first generates two large primes p and q that are both congruent to 3 mod 4. We set $n = pq$ and choose a random integer x that is relatively prime to n. To

initialize the BBS generator, set the initial seed to $x_0 \equiv x^2 \pmod{n}$. The BBS generator produces a sequence of random bits $b_1, b_2, \cdots$ by

1. $x_j \equiv x_{j-1}^2 \pmod{n}$
2. b_j is the least significant bit of x_j.

Example

Let

$$p = 24672462467892469787 \text{ and } q = 396736894567834589803,$$

$$n = 978847614085311079416885521741371578196 1.$$

Take $x = 873245647888478349013$. The initial seed is

$$\begin{aligned} x_0 &\equiv x^2 \pmod{n} \\ &\equiv 884529871047878009708991774601012286317 2. \end{aligned}$$

The values for $x_1, x_2, \cdots x_8$ are

$$\begin{aligned} x_1 &\equiv 711889428113132952274596245549812382240 8 \\ x_2 &\equiv 314517460888889316415138015206070451822 7 \\ x_3 &\equiv 489800778230715623327223318557489943035 5 \\ x_4 &\equiv 393545781893511292234709354618967231038 9 \\ x_5 &\equiv 675099511510097048901761303198740246040 \\ x_6 &\equiv 428991482877174013354619065826651517132 6 \\ x_7 &\equiv 443106671145437826089038638559381752166 8 \\ x_8 &\equiv 733687612419504639741423533367500537243 6. \end{aligned}$$

Taking the least significant bit of each of these, which is easily done by checking whether the number is odd or even, produces the sequence
$b_1, \cdots, b_8 = 0, 1, 1, 1, 0, 0, 0, 0.$

The Blum-Blum-Shub generator is very likely unpredictable. See [Blum-Blum-Shub]. A problem with BBS is that it can be slow to calculate. One way to improve its speed is to extract the k least significant bits

of x_j. As long as $k \leq \log_2 \log_2 n$, , this seems to be cryptographically secure.

5.2 Linear Feedback Shift Register Sequences

Note: **In this section, all congruences are mod 2**.

In many situations involving encryption, there is a trade-off between speed and security. If one wants a very high level of security, speed is often sacrificed, and vice versa. For example, in cable television, many bits of data are being transmitted, so speed of encryption is important. On the other hand, security is not usually as important since there is rarely an economic advantage to mounting an expensive attack on the system.

In this section, we describe a method that could be used when speed is more important than security. However, the real use is as one building block in more complex systems.

The sequence

$$0100001001011001111100011011101010000100101100111 1$$

can be described by giving the initial values

$$x_1 \equiv 0,\ x_2 \equiv 1,\ x_3 \equiv 0,\ x_4 \equiv 0,\ x_5 \equiv 0$$

and the linear recurrence relation

$$x_{n+5} \equiv x_n + x_{n+2} \pmod 2.$$

This sequence repeats after 31 terms.

For another example, see Example 18 in the Computer Appendices.

More generally, consider a linear recurrence relation of **length** m:

$$x_{n+m} \equiv c_0 x_n + c_1 x_{n+1} + \cdots + c_{m-1} x_{n+m-1} \pmod 2,$$

where the coefficients $c_0, c_1, \ldots$ are integers. If we specify the **initial values**

$$x_1, x_2, \ldots, x_m,$$

then all subsequent values of x_n can be computed using the recurrence. The resulting sequence of 0s and 1s can be used as the key for encryption. Namely, write the plaintext as a sequence of 0s and 1s, then add an appropriate number of bits of the key sequence to the plaintext mod 2, bit by bit. For example, if the plaintext is 1011001110001111 and the key sequence is the example given previously, we have

$$\begin{array}{rl} \text{(plaintext)} & 1011001110001111 \\ \text{(key)} \oplus & 0100001001011001 \\ \text{(ciphertext)} & 1111000111010110 \end{array}$$

Decryption is accomplished by adding the key sequence to the ciphertext in exactly the same way.

One advantage of this method is that a key with large period can be generated using very little information. The long period gives an improvement over the Vigenère method, where a short period allowed us to find the key. In the above example, specifying the initial vector $\{0, 1, 0, 0, 0\}$ and the coefficients $\{1, 0, 1, 0, 0\}$ yielded a sequence of period 31, so 10 bits were used to produce 31 bits. It can be shown that the recurrence

$$x_{n+31} \equiv x_n + x_{n+3}$$

and any nonzero initial vector will produce a sequence with period $2^{31} - 1 = 2147483647$. Therefore, 62 bits produce more than two billion bits of key. This is a great advantage over a one-time pad, where the full two billion bits must be sent in advance.

This method can be implemented very easily in hardware using what is known as a **linear feedback shift register** (LFSR) and is very fast. In Figure 5.2 we depict

an example of a linear feedback shift register in a simple case. More complicated recurrences are implemented using more registers and more XORs.

Figure 5.2 A Linear Feedback Shift Register Satisfying

$$x_{n+3} = x_{n+1} + x_n.$$

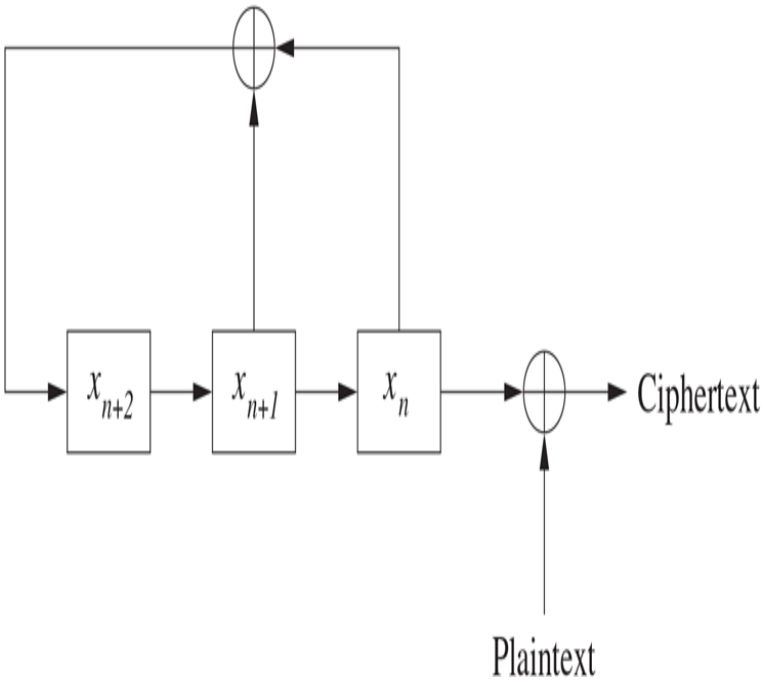


Figure 5.2 Full Alternative Text

For each increment of a counter, the bit in each box is shifted to other boxes as indicated, with $\oplus$ denoting the addition mod 2 of the incoming bits. The output, which is the bit x_n, is added to the next bit of plaintext to produce the ciphertext. The diagram in Figure 5.2 represents the recurrence $x_{n+3} \equiv x_{n+1} + x_n$. Once the initial values $x_1,\ x_2,\ x_3$ are specified, the machine produces the subsequent bits very efficiently.

Unfortunately, the preceding encryption method succumbs easily to a known plaintext attack. More precisely, if we know only a few consecutive bits of plaintext, along with the corresponding bits of ciphertext, we can determine the recurrence relation and therefore compute all subsequent bits of the key. By subtracting (or adding; it's all the same mod 2) the plaintext from the ciphertext mod 2, we obtain the bits of the key. Therefore, for the rest of this discussion, we will ignore the ciphertext and plaintext and assume we have discovered a portion of the key sequence. Our goal is to use this portion of the key to deduce the coefficients of the recurrence and consequently compute the rest of the key.

For example, suppose we know the initial segment 011010111100 of the sequence $011010111100010011010111\ldots$, which has period 15, and suppose we know it is generated by a linear recurrence. How do we determine the coefficients of the recurrence? We do not necessarily know even the length, so we start with length 2 (length 1 would produce a constant sequence). Suppose the recurrence is $x_{n+2} = c_0 x_n + c_1 x_{n+1}$. Let $n = 1$ and $n = 2$ and use the known values $x_1 = 0$, $x_2 = 1$, $x_3 = 1$, $x_4 = 0$. We obtain the equations

$$\begin{aligned} 1 &\equiv c_0 \cdot 0 + c_1 \cdot 1 \qquad (n = 1) \\ 0 &\equiv c_0 \cdot 1 + c_1 \cdot 1 \qquad (n = 2). \end{aligned}$$

In matrix form, this is

$$\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} c_0 \\ c_1 \end{pmatrix} \equiv \begin{pmatrix} 1 \\ 0 \end{pmatrix}.$$

The solution is $c_0 = 1$, $c_1 = 1$, so we guess that the recurrence is $x_{n+2} \equiv x_n + x_{n+1}$. Unfortunately, this is not correct since $x_6 \not\equiv x_4 + x_5$. Therefore, we try length 3. The resulting matrix equation is

$$\begin{matrix} 0 & 1 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \end{matrix} \begin{matrix} c_0 \\ c_1 \\ c_2 \end{matrix} \equiv \begin{matrix} 0 \\ 1 \\ 0 \end{matrix} .$$

The determinant of the matrix is 0 mod 2; in fact, the equation has no solution. We can see this because every column in the matrix sums to 0 mod 2, while the vector on the right does not.

Now consider length 4. The matrix equation is

$$\begin{matrix} 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{matrix} \begin{matrix} c_0 \\ c_1 \\ c_2 \\ c_3 \end{matrix} \equiv \begin{matrix} 1 \\ 0 \\ 1 \\ 1 \end{matrix} .$$

The solution is $c_0 = 1$, $c_1 = 1$, $c_2 = 0$, $c_3 = 0$. The resulting recurrence is now conjectured to be

$$x_{n+4} \equiv x_n + x_{n+1}.$$

A quick calculation shows that this generates the remaining bits of the piece of key that we already know, so it is our best guess for the recurrence that generates the key sequence.

What happens if we try length 5? The matrix equation is

$$\begin{matrix} 0 & 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 \end{matrix} \begin{matrix} c_0 \\ c_1 \\ c_2 \\ c_3 \\ c_4 \end{matrix} \equiv \begin{matrix} 0 \\ 1 \\ 1 \\ 1 \\ 1 \end{matrix} .$$

The determinant of the matrix is 0 mod 2. Why? Notice that the last row is the sum of the first and second rows. This is a consequence of the recurrence relation: $x_5 \equiv x_1 + x_2$, $x_6 \equiv x_2 + x_3$, etc. As in linear algebra with real or complex numbers, if one row of a matrix is a linear combination of other rows, then the determinant is 0.

Similarly, if we look at the 6×6 matrix, we see that the 5th row is the sum of the first and second rows, and the

6th row is the sum of the second and third rows, so the determinant is 0 mod 2. In general, when the size of the matrix is larger than the length of the recurrence relation, the relation forces one row to be a linear combination of other rows, hence the determinant is 0 mod 2.

The general situation is as follows. To test for a recurrence of length m, we assume we know $x_1, x_2, \ldots, x_{2m}$. The matrix equation is

$$\begin{matrix} x_1 & x_2 & \cdots & x_m & c_0 \\ x_2 & x_3 & \cdots & x_{m+1} & c_1 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ x_m & x_{m+1} & \cdots & x_{2m-1} & c_{m-1} \end{matrix} \equiv \begin{matrix} x_{m+1} \\ x_{m+2} \\ \vdots \\ x_{2m} \end{matrix} .$$

We show later that the matrix is invertible mod 2 if and only if there is no linear recurrence of length less than m that is satisfied by $x_1, x_2, \ldots, x_{2m-1}$.

A strategy for finding the coefficients of the recurrence is now clear. Suppose we know the first 100 bits of the key. For $m = 2, 3, 4, \ldots$, form the $m \times m$ matrix as before and compute its determinant. If several consecutive values of m yield 0 determinants, stop. The last m to yield a nonzero (i.e., 1 mod 2) determinant is probably the length of the recurrence. Solve the matrix equation to get the coefficients $c_0, \ldots, c_{m-1}$. It can then be checked whether the sequence that this recurrence generates matches the sequence of known bits of the key. If not, try larger values of m.

Suppose we don't know the first 100 bits, but rather some other 100 consecutive bits of the key. The same procedure applies, using these bits as the starting point. In fact, once we find the recurrence, we can also work backwards to find the bits preceding the starting point.

Here is an example. Suppose we have the following sequence of 100 bits:

1001100100111000110001010001111011001111101010100101101101011000011011100101011110000000100010010000.

The first 20 determinants, starting with $m = 1$, are

$$1, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0.$$

A reasonable guess is that $m = 8$ gives the last nonzero determinant. When we solve the matrix equation for the coefficients we get

$$\{c_0, c_1, \ldots, c_7\} = \{1, 1, 0, 0, 1, 0, 0, 0\},$$

so we guess that the recurrence is

$$x_{n+8} \equiv x_n + x_{n+1} + x_{n+4}.$$

This recurrence generates all 100 terms of the original sequence, so we have the correct answer, at least based on the knowledge that we have.

Suppose that the 100 bits were in the middle of some sequence, and we want to know the preceding bits. For example, suppose the sequence starts with x_{17}, so $x_{17} = 1$, $x_{18} = 0$, $x_{19} = 0, \ldots$. Write the recurrence as

$$x_n \equiv x_{n+1} + x_{n+4} + x_{n+8}$$

(it might appear that we made some sign errors, but recall that we are working mod 2, so $-x_n \equiv x_n$ and $-x_{n+8} \equiv x_{n+8}$). Letting $n = 16$ yields

$$\begin{aligned} x_{16} &\equiv x_{17} + x_{20} + x_{24} \\ &\equiv 1 + 0 + 1 \equiv 0. \end{aligned}$$

Continuing in this way, we successively determine $x_{15}, x_{14}, \ldots, x_1$.

For more examples, see Examples 19 and 20 in the Computer Appendices.

We now prove the result we promised.

Proposition

Let $x_1, x_2, x_3, \ldots$ be a sequence of bits produced by a linear recurrence mod 2. For each $n \geq 1$, let

$$M_n = \begin{matrix} x_1 & x_2 & \cdots & x_n \\ x_2 & x_3 & \cdots & x_{n+1} \\ \vdots & \vdots & \ddots & \vdots \\ x_n & x_{n+1} & \cdots & x_{2n-1} \end{matrix}.$$

Let N be the length of the shortest recurrence that generates the sequence $x_1, x_2, x_3, \ldots$. Then $\det(M_N) \equiv 1 \pmod 2$ and $\det(M_n) \equiv 0 \pmod 2$ for all $n > N$.

Proof. We first make a few remarks on the length of recurrences. A sequence could satisfy a length 3 relation such as $x_{n+3} \equiv x_{n+2}$. It would clearly then also satisfy shorter relations such as $x_{n+1} = x_n$ (at least for $n \geq 2$). However, there are less obvious ways that a sequence could satisfy a recurrence of length less than expected. For example, consider the relation $x_{n+4} \equiv x_{n+3} + x_{n+1} + x_n$. Suppose the initial values of the sequence are $1, 1, 0, 1$. The recurrence allows us to compute subsequent terms: $1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1 \ldots$. It is easy to see that the sequence satisfies $x_{n+2} \equiv x_{n+1} + x_n$.

If there is a recurrence of length N and if $n > N$, then one row of the matrix M_n is congruent mod 2 to a linear combination of other rows. For example, if the recurrence is $x_{n+3} = x_{n+2} + x_n$, then the fourth row is the sum of the first and third rows. Therefore, $\det(M_n) \equiv 0 \pmod 2$ for all $n > N$.

Now suppose $\det(M_N) \equiv 0 \pmod 2$. Then there is a nonzero row vector $b = (b_0, \ldots, b_{N-1})$ such that $bM_N \equiv 0$. We'll show that this gives a recurrence relation for the sequence $x_1, x_2, x_3, \ldots$ and that the length of this relation is less than N. This contradicts the

assumption that N is smallest. This contradiction implies that $\det\,(M_N) \equiv 1 \pmod 2$.

Let the recurrence of length N be

$$x_{n+N} \equiv c_0 x_n + \cdots + c_{N-1} x_{n+N-1}.$$

For each $i \geq 0$, let

$$M^{(i)} = \begin{matrix} x_{i+1} & x_{i+2} & \cdots & x_{i+N} \\ x_{i+2} & x_{i+3} & \cdots & x_{i+N+1} \\ \vdots & \vdots & \ddots & \vdots \\ x_{i+N} & x_{i+N+1} & \cdots & x_{i+2N-1} \end{matrix}.$$

Then $M^{(0)} = M_N$. The recurrence relation implies that

$$M^{(i)} \begin{matrix} c_0 \\ c_1 \\ \vdots \\ c_{N-1} \end{matrix} \equiv \begin{matrix} x_{i+N+1} \\ x_{i+N+2} \\ \vdots \\ x_{i+2N} \end{matrix},$$

which is the last column of $M^{(i+1)}$.

By the choice of b, we have $bM^{(0)} = bM_N = 0$. Suppose that we know that $bM^{(i)} = 0$ for some i. Then

$$b \begin{matrix} x_{i+N+1} \\ x_{i+N+2} \\ \vdots \\ x_{i+2N} \end{matrix} \equiv bM^{(i)} \begin{matrix} c_0 \\ c_1 \\ \vdots \\ c_{m-1} \end{matrix} \equiv 0.$$

Therefore, b annihilates the last column of $M^{(i+1)}$. Since the remaining columns of $M^{(i+1)}$ are columns of $M^{(i)}$, we find that $bM^{(i+1)} \equiv 0$. By induction, we obtain $bM^{(i)} \equiv 0$ for all $i \geq 0$.

Let $n \geq 1$. The first column of $M^{(n-1)}$ yields

$$b_0 x_n + b_1 x_{n+1} + \cdots + b_{N-1} x_{n+N-1} \equiv 0.$$

Since b is not the zero vector, $b_j \neq 0$ for at least one j. Let m be the largest j such that $b_j \neq 0$, which means that $b_m = 1$. We are working mod 2, so

$b_m x_{n+m} \equiv -x_{n+m}$. Therefore, we can rearrange the relation to obtain

$$x_{n+m} \equiv b_0 x_n + b_1 x_{n+1} + \cdots + b_{m-1} x_{n+m-1}.$$

This is a recurrence of length m. Since $m \leq N-1$, and N is assumed to be the shortest possible length, we have a contradiction. Therefore, the assumption that $\det(M_N) \equiv 0$ must be false, so $\det(M_N) \equiv 1$. This completes the proof.

Finally, we make a few comments about the period of a sequence. Suppose the length of the recurrence is m. Any m consecutive terms of the sequence determine all future elements, and, by reversing the recurrence, all previous values, too. Clearly, if we have m consecutive 0s, then all future values are 0. Also, all previous values are 0. Therefore, we exclude this case from consideration. There are $2^m - 1$ strings of 0s and 1s of length m in which at least one term is nonzero. Therefore, as soon as there are more than $2^m - 1$ terms, some string of length m must occur twice, so the sequence repeats. The period of the sequence is at most $2^m - 1$.

Associated to a recurrence $x_{n+m} \equiv c_0 x_n + c_1 x_{n+1} + \cdots + c_{m-1} x_{n+m-1} \pmod 2$, there is a polynomial

$$f(T) = T^m - c_{m-1} T^{m-1} - \cdots - c_0.$$

If $f(T)$ is irreducible mod 2 (this means that it is not congruent to the product of two lower-degree polynomials), then it can be shown that the period divides $2^m - 1$. An interesting case is when $2^m - 1$ is prime (these are called Mersenne primes). If the period isn't 1, that is, if the sequence is not constant, then the period in this special case must be maximal, namely $2^m - 1$ (see Section 3.11). The example where the period is $2^{31} - 1$ is of this type.

Linear feedback shift register sequences have been studied extensively. For example, see [Golomb] or [van der Lubbe].

One way of thwarting the above attack is to use nonlinear recurrences, for example,

$$x_{n+3} \equiv x_{n+2}x_n + x_{n+1}.$$

Moreover, a look-up table that takes inputs x_n, x_{n+1}, x_{n+2} and outputs a bit x_{n+3} could be used, or several LFSRs could be combined nonlinearly and some of these LFSRs could have irregular clocking. Generally, these systems are somewhat harder to break. However, we shall not discuss them here.

5.3 RC4

RC4 is a stream cipher that was developed by Rivest and has been widely used because of its speed and simplicity. The algorithm was originally secret, but it was leaked to the Internet in 1994 and has since been extensively analyzed. In particular, certain statistical biases were found in the keystream it generated, especially in the initial bits. Therefore, often a version called RC4-drop[n] is used, in which the first n bits are dropped before starting the keystream. However, this version is still not recommended for situations requiring high security.

To start the generation of the keystream for RC4, the user chooses a key, which is a binary string between 40 and 256 bits long. This is put into the Key Scheduling Algorithm. It starts with an array S consisting of the numbers from 0 to 255, regarded as 8-bit bytes, and outputs a permutation of these entries, as follows:

Algorithm 1 RC4 Key Scheduling Algorithm

- 1: **for** i from 0 to 255 **do**
- 2: $S[i] := i$
- 3: $j := 0$
- 4: **for** i from 0 to 255 **do**
- 5: $j := (j + S[i] + \text{key}[i \bmod \text{keylength}]) \bmod 256$
- 6: $\text{swap}(S[i], S[j])$

This algorithm starts by initializing the entries in S as $S[i] = i$ for i running from 0 through 255. Suppose the

user-supplied key is $10100000\cdots$ (let's say the key length is 40).

The algorithm starts with $j = 0$ and $i = 0$. The value of j is updated to
$j + S[i] + \text{key}[i \bmod 40] = 0 + 0 + 1 = 1$. Then $S[i] = S[0] = 0$ and $S[j] = S[1] = 1$ are swapped, so now $S[0] = 1$ and $S[1] = 0$.

We now move to $i = 1$. The value $j = 1$ is updated to $1 + 0 + \text{key}[1] = 1$, so $S[1]$ is swapped with itself, which means it is not changed here.

We now move to $i = 2$. The value $j = 1$ is updated to $1 + 2 + \text{key}[2] = 4$, so $S[2] = 2$ is swapped with $S[4] = 4$, yielding $S[2] = 4$ and $S[4] = 2$.

We now move to $i = 3$. The value $j = 4$ is updated to $4 + 3 + 0 = 7$, so $S[3]$ and $S[7]$ are swapped, yielding $S[3] = 7$ and $S[7] = 3$.

Let's look at one more value of i, namely, $i = 4$. The value $j = 7$ is updated to $7 + 2 + 0 = 9$ (recall that $S[4]$ became 2 earlier), and we obtain $S[4] = 9$ and $S[9] = 2$.

This process continues through $i = 255$ and yields an array S of length 256 consisting of a permutation of the numbers from 0 through 255.

The array S is entered into the Pseudorandom Generation Algorithm.

Algorithm 2 RC4 Pseudorandom Generation Algorithm (PRGA)

- $i := 0$
- $j := 0$
- **while** GeneratingOutput: **do**
    - $i := (i + 1) \bmod 256$
    - $j := (j + S[i]) \bmod 256$
    - $\text{swap}(S[i],\ S[j])$
    - output $S[(S[i] + S[j]) \bmod 256]$

This algorithm runs as long as needed and each round outputs a number between 0 and 255, regarded as an 8-bit byte. This byte is XORed with the corresponding byte of the plaintext to yield the ciphertext.

Weaknesses. Generally, the keystream that is output by a stream cipher should be difficult to distinguish from a randomly generated bitstream. For example, the R Game (see Section 4.5) could be played, and the probability of winning should be negligibly larger than 1/2. For RC4, there are certain observable biases. The second byte in the output should be 0 with probability 1/256. However, Mantin and Shamir [Mantin-Shamir] showed that this byte is 0 with twice that probability. Moreover, they found that the probability that the first two bytes are simultaneously 0 is $3/256^2$ instead of the expected $1/256^2$.

Biases have also been found in the state S that is output by the Key Scheduling Algorithm. For example, the probability that $S[0] = 1$ is about 37% larger than the expected probability of $1/256$, while the probability that $S[0] = 255$ is 26% less than expected.

Although any key length from 40 to 255 bits can be chosen, the use of small key sizes is not recommended because the algorithm can succumb to a brute force attack.

5.4 Exercises

1. A sequence generated by a length 3 recurrence starts 001110. Find the next four elements of the sequence.

2. The LFSR sequence $10011101\cdots$ is generated by a recurrence relation of length 3:
$x_{n+3} \equiv c_0 x_n + c_1 x_{n+1} + c_2 x_{n+2} \pmod 2$. Find the coefficients c_0, c_1, c_2.

3. The LFSR sequence $100100011110\cdots$ is generated by a recurrence relation of length 4:
$x_{n+4} \equiv c_0 x_n + c_1 x_{n+1} + c_2 x_{n+2} + c_3 x_{n+3} \pmod 2$. Find the coefficients c_0, c_1, c_2, c_3.

4. The LFSR sequence $10111001\cdots$ is generated by a recurrence of length 3: $x_{n+3} \equiv c_0 x_n + c_1 x_{n+1} + c_2 x_{n+2} \pmod 2$. Find the coefficients c_0, c_1, and c_2.

5. Suppose we build an LFSR machine that works mod 3 instead of mod 2. It uses a recurrence of length 2 of the form

$$x_{n+2} \equiv c_0 x_n + c_1 x_{n+1} \pmod 3$$

to generate the sequence 1, 1, 0, 2, 2, 0, 1, 1. Set up and solve the matrix equation to find the coefficients c_0 and c_1.

6. The sequence
$x_1 = 0, x_2 = 1, x_3 = 1, x_4 = 1, x_5 = 0, x_6 = 0, x_7 = 1, x_8 = 0, x_9 = 1, \cdots$
is generated by a recurrence relation

$$x_{n+3} \equiv c_0 x_n + c_1 x_{n+1} + c_2 x_{n+2} \pmod 2.$$

Determine x_{10}, x_{11}, x_{12}.

7. Consider the sequence starting $k_1 = 1$, $k_2 = 0$, $k_3 = 1$ and defined by the length 3 recurrence
$k_{n+3} \equiv k_n + k_{n+1} + k_{n+2} \pmod 2$. This sequence can also be given by a length 2 recurrence. Determine this length 2 recurrence by setting up and solving the appropriate matrix equations.

8. Suppose we build an LFSR-type machine that works mod 2. It uses a recurrence of length 2 of the form

$$x_{n+2} \equiv c_0 x_n + c_1 x_{n+1} + 1 \pmod 2$$

to generate the sequence 1,1,0,0,1,1,0,0. Find c_0 and c_1.

9. Suppose you modify the LFSR method to work mod 5 and you use a (not quite linear) recurrence relation

$$x_{n+2} \equiv c_0 x_n + c_1 x_{n+1} + 2 \pmod 5,$$
$$x_1 = 0,\ x_2 = 1,\ x_3 = 1,\ x_4 = 0.$$

Find the coefficients c_0 and c_1.

10. 1. Suppose you make a modified LFSR machine using the recurrence relation $x_{n+2} \equiv A + Bx_n + Cx_{n+1} \pmod 2$, where A, B, C are constants. Suppose the output starts 0, 1, 1, 0, 0, 1, 1, 0, 0. Find the constants A, B, C.

    2. Show that the sequence does not satisfy any linear recurrence of the form $x_{n+2} \equiv c_0 x_n + c_1 x_{n+1} \pmod 2$.

11. Bob has a great idea to generate pseudorandom bytes. He takes the decimal expansion of π, which we assume is random, and chooses three consecutive digits in this expansion, starting at a randomly chosen point. He then regards what he gets as a three-digit integer, which he writes as a 10-digit number in binary. Finally, he chooses the last eight binary digits to get a byte. For example, if the digits of π that he chooses are 159, he changes this to 0010011111. This yields the byte 10011111.

    1. Show that this pseudorandom number generator produces some bytes more often than others.

    2. Suppose Bob modifies his algorithm so that if his three-digit decimal integer is greater than or equal to 512, then he discards it and tries again. Show that this produces random output (assuming the same is true for π).

12. Suppose you have a Geiger counter and a radioactive source. If the Geiger counter counts an even number of radioactive particles in a second, you write 0. If it records an odd number of particles, you write 1. After a period of time, you have a binary sequence. It is reasonable to expect that the probability p_n that n particles are counted in a second satisfies a Poisson distribution

$$p_n = e^{-\lambda}\frac{\lambda^n}{n!} \text{ for } n \geq 0,$$

where λ is a parameter (in fact, λ is the average number of particles per second).

    1. Show that if $0 < \lambda < 1$ then $p_0 > p_1 > p_2 \cdots$.

    2. Show that if $\lambda < 1$ then the binary sequence you obtain is expected to have more 0s than 1s.

    3. More generally, show that, whenever $\lambda \geq 0$,

$$\text{Prob}(n \text{ is even}) = e^{-\lambda}\cosh(\lambda)$$
$$\text{Prob}(n \text{ is odd}) = e^{-\lambda}\sinh(\lambda).$$

4. Show that for every $\lambda \geq 0$,

$$\text{Prob}(n \text{ is even}) > \text{Prob}(n \text{ is odd}).$$

This problem shows that, although a Geiger counter might be a good source of randomness, the naive method of using it to obtain a pseudorandom sequence is biased.

13.

1. Suppose that during the PRGA of RC4, there occur values of $i = i_0$ and $j = j_0$ such that $j_0 = i_0 + 1$ and $S[i_0 + 1] = 1$. The next values of i, j in the algorithm are i_1, j_1, with $i_1 = i + 1$ and $j_1 = j + 1$. Show that $S[i_1 + 1] = 1$, so this property continues for all future i, j.

2. The values of i, j before i_0, j_0 are $i^- = i_0 - 1$ and $j^- = j_0 - 1$. Show that $S[i^-] = 1$, so if this property occurs, then it occurred for all previous values of i, j.

3. The starting values of i and j are $i = 0$ and $j = 0$. Use this to show that there are never values of i, j such that $j = i + 1$ and $S[i] = 1$.

14. Let $f(x)$ be a one-way function. In Section 5.1, it was pointed out that usually the least significant bits of $f(s + j)$ for $j = 1, 2, 3, \ldots$ (s is a seed) can be used to give a pseudorandom sequence of bits. Show how to append some bits to $f(x)$ to obtain a new one-way function for which the sequence of least significant bits is not pseudorandom.

5.5 Computer Problems

1. The following sequence was generated by a linear feedback shift register. Determine the recurrence that generated it.

```
1, 0, 1, 0, 0, 1, 1, 0, 1, 1, 0, 0, 0, 1,
0, 0, 1, 0, 0,
0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1, 0, 1, 1,
1, 1, 1, 1, 0,
0, 1, 0, 1, 0, 1, 0, 0, 0, 1, 1, 0, 0, 1,
1, 1, 1, 0, 1,
1, 1, 0, 1, 0, 1, 1, 0, 1, 0, 0, 1, 1, 0,
1, 1, 0, 0, 0,
1, 0, 0, 1, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0,
0, 0, 1, 0, 1,
1, 1, 1, 1, 1
```

   (It is stored in the downloadable computer files (`bit.ly/2JbcS6p`) under the name *L101*.)

2. The following are the first 100 terms of an LFSR output. Find the coefficients of the recurrence.

```
1, 0, 0, 1, 1, 0, 0, 1, 0, 0, 1, 1, 1, 0,
0, 0, 1, 1, 0,
0, 0, 1, 0, 1, 0, 0, 0, 1, 1, 1, 1, 0, 1,
1, 0, 0, 1, 1,
1, 1, 1, 0, 1, 0, 1, 0, 1, 0, 0, 1, 0, 1,
1, 0, 1, 1, 0,
1, 0, 1, 1, 0, 0, 0, 0, 1, 1, 0, 1, 1, 1,
0, 0, 1, 0, 1,
0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0,
0, 0, 1, 0, 0,
1, 0, 0, 0, 0
```

   (It is stored in the downloadable computer files (`bit.ly/2JbcS6p`) under the name *L100*.)

3. The following ciphertext was obtained by XORing an LFSR output with the plaintext.

```
0, 1, 1, 0, 0, 0, 1, 0, 1, 0, 1, 1, 1, 0,
```

```
0, 1, 1, 1, 0,
1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 1, 0, 0,
0, 1, 0, 1, 0,
1, 1, 1, 0, 0, 1, 1, 1, 0, 1, 0, 1
```

Suppose you know the plaintext starts

```
1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0,
0
```

Find the plaintext. (The ciphertext is stored in the downloadable computer files (`bit.ly/2JbcS6p`) under the name *L011*.)