b. For each case of problem (a), determine the number $x = \text{NUM}_{26}(X)$
c. Determine the byte form $[\boldsymbol{x}]$ for each number x computed in problem (b).
d. What is the smallest power of the radix (26) that is greater than each of the numerical strings determined in (b)?
e. Is it related to the length of the numeral string in each case, in problem (a)? If so, what is this relationship?

7.16 Refer to algorithms FF1 and FF2.
a. For step 1, for each algorithm, $u \leftarrow \lfloor n/2 \rfloor$ and $v \leftarrow \lceil n - u \rceil$. Show that for any three integers x, y, and n:
if $x = \lceil n/2 \rceil$ and $y = \lceil n - x \rceil$, then:
i. Either $x = n/2$, or $x = (n - 1)/2$.
ii. Either $y = n/2$, or $y = (n + 1)/2$.
iii. $x \leq y$. (Under what condition is $x = y$?)
b. What is the significance of result in the previous sub-problem (iii), in terms of the lengths u and v of the left and right half-strings, respectively?

7.17 In step 3 of Algorithm FF1, what do b and d represent? What is the unit of measurement (bits, bytes, digits, characters) of each of these quantities?

7.18 In the inputs to algorithms FF1, FF2, and FF3, why are the specified radix ranges important? For example, why should $radix \in [0..2^8]$ for Algorithm FF2, or $radix \in [2..2^{16}]$ in the case of Algorithm FF3?

Programming Problems

7.1 Create software that can encrypt and decrypt in cipher block chaining mode using one of the following ciphers: affine modulo 256, Hill modulo 256, S-DES, DES.

Test data for S-DES using a binary initialization vector of 1010 1010. A binary plaintext of 0000 0001 0010 0011 encrypted with a binary key of 01111 11101 should give a binary plaintext of 1111 0100 0000 1011. Decryption should work correspondingly.

7.2 Create software that can encrypt and decrypt in 4-bit cipher feedback mode using one of the following ciphers: additive modulo 256, affine modulo 256, S-DES;

or

8-bit cipher feedback mode using one of the following ciphers: 2×2 Hill modulo 256. Test data for S-DES using a binary initialization vector of 1010 1011. A binary plaintext of 0001 0010 0011 0100 encrypted with a binary key of 01111 11101 should give a binary plaintext of 1110 1100 1111 1010. Decryption should work correspondingly.

7.3 Create software that can encrypt and decrypt in counter mode using one of the following ciphers: affine modulo 256, Hill modulo 256, S-DES.

Test data for S-DES using a counter starting at 0000 0000. A binary plaintext of 0000 0001 0000 0010 0000 0100 encrypted with a binary key of 01111 11101 should give a binary plaintext of 0011 1000 0100 1111 0011 0010. Decryption should work correspondingly.

7.4 Implement a differential cryptanalysis attack on 3-round S-DES.

CHAPTER 8

Random Bit Generation and Stream Ciphers

LEARNING OBJECTIVES

After studying this chapter, you should be able to:

- Explain the concepts of randomness and unpredictability with respect to random numbers.
- Understand the differences among true random number generators, pseudorandom number generators, and pseudorandom functions.
- Present an overview of requirements for pseudorandom number generators.
- Explain how a block cipher can be used to construct a pseudorandom number generator.
- Present an overview of stream ciphers and RC4.
- Explain the significance of skew.

An important cryptographic function is the generation of random bit streams. Random bits streams are used in a wide variety of contexts, including key generation and encryption. In essence, there are two fundamentally different strategies for generating random bits or random numbers. One strategy, which until recently dominated in cryptographic applications, computes bits deterministically using an algorithm. This class of random bit generators is known as pseudorandom number generators (PRNGs) or deterministic random bit generators (DRBGs). The other strategy is to produce bits non-deterministically using some physical source that produces some sort of random output. This latter class of random bit generators is known as true random number generators (TRNGs) or non-deterministic random bit generators (NRBGs).

The chapter begins with an analysis of the basic principles of PRNGs. Next, we look at some common PRNGs, including PRNGs based on the use of a symmetric block cipher. The chapter then moves on to the topic of symmetric stream ciphers, which are based on the use of a PRNG.

The remainder of the chapter is devoted to TRNGs. We look first at the basic principles and structure of TRNGs, and then examine a specific product, the Intel Digital Random Number Generator.

Throughout this chapter, reference is made to four important NIST documents:

- SP 800-90A (*Recommendation for Random Number Generation Using Deterministic Random Bit Generators,* June 2015): Specifies mechanisms for the generation of random bits using deterministic methods.
- SP 800-90B (*Recommendation for the Entropy Sources Used for Random Bit Generation,* January 2018): Covers design principles and requirements for entropy sources (ES), the devices from which we get unpredictable randomness and NRNGs.
- SP 800-90C (*Recommendation for Random Bit Generator (RBG) Constructions,* April 2016): Discusses how to combine the entropy sources in 90B with the DRNG's from 90A to provide large quantities of unpredictable bits for cryptographic applications.

- SP 800-22 (*A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications,* April 2010) discusses the selection and testing of NRBGs and DRBGs.

These specifications have heavily influenced the implementation of random bit generators in industry both in the U.S. and worldwide.

8.1 PRINCIPLES OF PSEUDORANDOM NUMBER GENERATION

Random numbers play an important role in the use of encryption for various network security applications. In this section, we provide a brief overview of the use of random numbers in cryptography and network security and then focus on the principles of pseudorandom number generation.

The Use of Random Numbers

A number of network security algorithms and protocols based on cryptography make use of random binary numbers. For example,

- Key distribution and reciprocal (mutual) authentication schemes, such as those discussed in Chapters 14 and 15. In such schemes, two communicating parties cooperate by exchanging messages to distribute keys and/or authenticate each other. In many cases, nonces are used for handshaking to prevent replay attacks. The use of random numbers for the nonces frustrates an opponent's efforts to determine or guess the nonce, in order to repeat an obsolete transaction.
- Session key generation. We will see a number of protocols in this book where a secret key for symmetric encryption is generated for use for a particular transaction (or session) and is valid for a short period of time. This key is generally called a session key.
- Generation of keys for the RSA public-key encryption algorithm (described in Chapter 9).
- Generation of a bit stream for symmetric stream encryption (described in this chapter).

These applications give rise to two distinct and not necessarily compatible requirements for a sequence of random numbers: randomness and **unpredictability**.

RANDOMNESS Traditionally, the concern in the generation of a sequence of allegedly random numbers has been that the sequence of numbers be random in some well-defined statistical sense. The following two criteria are used to validate that a sequence of numbers is random:

- **Uniform distribution:** The distribution of bits in the sequence should be uniform; that is, the frequency of occurrence of ones and zeros should be approximately equal.
- **Independence:** No one subsequence in the sequence can be inferred from the others.

Although there are well-defined tests for determining that a sequence of bits matches a particular distribution, such as the uniform distribution, there is no such test to "prove" independence. Rather, a number of tests can be applied to demonstrate if a sequence does not exhibit independence. The general strategy is to apply a number of such tests until the confidence that independence exists is sufficiently strong. That is, if each of a number of tests fails to show that a sequence of bits is not independent, then we can have a high level of confidence that the sequence is in fact independent.

In the context of our discussion, the use of a sequence of numbers that appear statistically random often occurs in the design of algorithms related to cryptography. For example, a fundamental requirement of the RSA public-key encryption scheme discussed in Chapter 9 is the ability to generate prime numbers. In general, it is difficult to determine if a given large number N is prime. A brute-force approach would be to divide N by every odd integer less than $\sqrt{N}$. If N is on the order, say, of 10^{150}, which is a not uncommon occurrence in public-key cryptography, such a brute-force approach is beyond the reach of human analysts and their computers. However, a number of effective algorithms exist that test the primality of a number by using a sequence of randomly chosen integers as input to relatively simple computations. If the sequence is sufficiently long (but far, far less than $\sqrt{10^{150}}$), the primality of a number can be determined with near certainty. This type of approach, known as randomization, crops up frequently in the design of algorithms. In essence, if a problem is too hard or time-consuming to solve exactly, a simpler, shorter approach based on randomization is used to provide an answer with any desired level of confidence.

Unpredictability In applications such as reciprocal authentication, session key generation, and stream ciphers, the requirement is not just that the sequence of numbers be statistically random but that the successive members of the sequence are unpredictable. With "true" random sequences, each number is statistically independent of other numbers in the sequence and therefore unpredictable. Although true random numbers are used in some applications, they have their limitations, such as inefficiency, as is discussed shortly. Thus, it is more common to implement algorithms that generate sequences of numbers that appear to be random but are in fact not random. In this latter case, care must be taken that an opponent not be able to predict future elements of the sequence on the basis of earlier elements.

TRNGs, PRNGs, and PRFs

Cryptographic applications typically make use of algorithmic techniques for random number generation. These algorithms are deterministic and therefore produce sequences of numbers that are not statistically random. However, if the algorithm is good, the resulting sequences will pass many tests of randomness. Such numbers are referred to as pseudorandom numbers.

You may be somewhat uneasy about the concept of using numbers generated by a deterministic algorithm as if they were random numbers. Despite what might be called philosophical objections to such a practice, it generally works. That is, under most circumstances, pseudorandom numbers will perform as well as if they were random for a given use. The phrase "as well as" is unfortunately subjective, but the

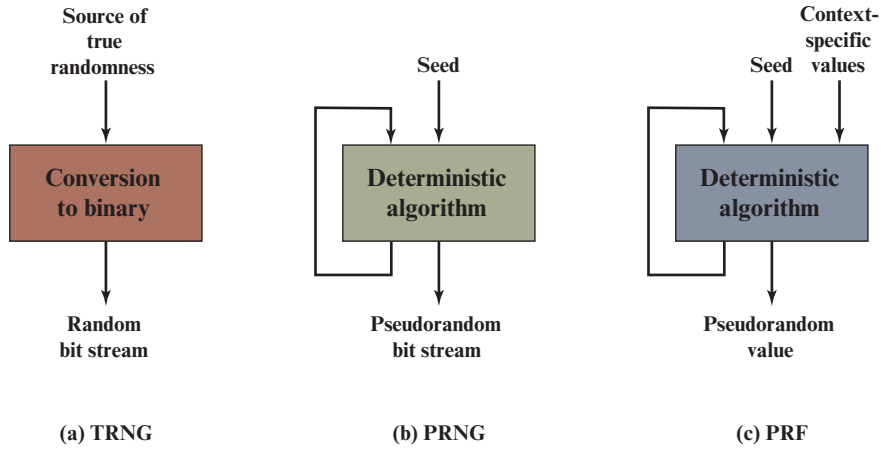


Figure 8.1 Random and Pseudorandom Number Generators

use of pseudorandom numbers is widely accepted. The same principle applies in statistical applications, in which a statistician takes a sample of a population and assumes that the results will be approximately the same as if the whole population were measured.

Figure 8.1 contrasts a **true random number generator (TRNG)** with two forms of pseudorandom number generators. A TRNG takes as input a source that is effectively random; the source is often referred to as an **entropy source**. We discuss such sources in Section 8.6. In essence, the entropy source is drawn from the physical environment of the computer and could include things such as keystroke timing patterns, disk electrical activity, mouse movements, and instantaneous values of the system clock. The source, or combination of sources, serve as input to an algorithm that produces random binary output. The TRNG may simply involve conversion of an analog source to a binary output. The TRNG may involve additional processing to overcome any bias in the source; this is discussed in Section 8.6.

In contrast, a PRNG takes as input a fixed value, called the **seed**, and produces a sequence of output bits using a deterministic algorithm. Quite often, the seed is generated by a TRNG. Typically, as shown, there is some feedback path by which some of the results of the algorithm are fed back as input as additional output bits are produced. The important thing to note is that the output bit stream is determined solely by the input value or values, so that an adversary who knows the algorithm and the seed can reproduce the entire bit stream.

Figure 8.1 shows two different forms of PRNGs, based on application.

- **Pseudorandom number generator:** An algorithm that is used to produce an open-ended sequence of bits is referred to as a **PRNG**. A common application for an open-ended sequence of bits is as input to a symmetric stream cipher, as discussed in Section 8.4. Also, see Figure 4.1a.

- **Pseudorandom function (PRF):** A PRF is used to produce a pseudorandom string of bits of some fixed length. Examples are symmetric encryption keys and nonces. Typically, the PRF takes as input a seed plus some context specific values, such as a user ID or an application ID. A number of examples of PRFs will be seen throughout this book, notably in Chapters 19 and 20.

Other than the number of bits produced, there is no difference between a PRNG and a PRF. The same algorithms can be used in both applications. Both require a seed and both must exhibit randomness and unpredictability. Further, a PRNG application may also employ context-specific input. In what follows, we make no distinction between these two applications.

PRNG Requirements

When a PRNG or PRF is used for a cryptographic application, then the basic requirement is that an adversary who does not know the seed is unable to determine the pseudorandom string. For example, if the pseudorandom bit stream is used in a stream cipher, then knowledge of the pseudorandom bit stream would enable the adversary to recover the plaintext from the ciphertext. Similarly, we wish to protect the output value of a PRF. In this latter case, consider the following scenario. A 128-bit seed, together with some context-specific values, are used to generate a 128-bit secret key that is subsequently used for symmetric encryption. Under normal circumstances, a 128-bit key is safe from a brute-force attack. However, if the PRF does not generate effectively random 128-bit output values, it may be possible for an adversary to narrow the possibilities and successfully use a brute force attack.

This general requirement for secrecy of the output of a PRNG or PRF leads to specific requirements in the areas of randomness, unpredictability, and the characteristics of the seed. We now look at these in turn.

Randomness In terms of randomness, the requirement for a PRNG is that the generated bit stream appear random even though it is deterministic. There is no single test that can determine if a PRNG generates numbers that have the characteristic of randomness. The best that can be done is to apply a sequence of tests to the PRNG. If the PRNG exhibits randomness on the basis of multiple tests, then it can be assumed to satisfy the randomness requirement. NIST SP 800-22 specifies that the tests should seek to establish the following three characteristics.

- **Uniformity:** At any point in the generation of a sequence of random or pseudorandom bits, the occurrence of a zero or one is equally likely, that is, the probability of each is exactly 1/2. The expected number of zeros (or ones) is $n/2$, where n = the sequence length.
- **Scalability:** Any test applicable to a sequence can also be applied to subsequences extracted at random. If a sequence is random, then any such extracted subsequence should also be random. Hence, any extracted subsequence should pass any test for randomness.
- **Consistency:** The behavior of a generator must be consistent across starting values (seeds). It is inadequate to test a PRNG based on the output from a single seed or a TRNG on the basis of an output produced from a single physical output.

SP 800-22 lists 15 separate tests of randomness. An understanding of these tests requires a basic knowledge of statistical analysis, so we don't attempt a technical description here. Instead, to give some flavor for the tests, we list three of the tests and the purpose of each test, as follows.

- **Frequency test:** This is the most basic test and must be included in any test suite. The purpose of this test is to determine whether the number of ones and zeros in a sequence is approximately the same as would be expected for a truly random sequence.
- **Runs test:** The focus of this test is the total number of runs in the sequence, where a run is an uninterrupted sequence of identical bits bounded before and after with a bit of the opposite value. The purpose of the runs test is to determine whether the number of runs of ones and zeros of various lengths is as expected for a random sequence.
- **Maurer's universal statistical test:** The focus of this test is the number of bits between matching patterns (a measure that is related to the length of a compressed sequence). The purpose of the test is to detect whether or not the sequence can be significantly compressed without loss of information. A significantly compressible sequence is considered to be non-random.

Unpredictability A stream of pseudorandom numbers should exhibit two forms of unpredictability:

- **Forward unpredictability:** If the seed is unknown, the next output bit in the sequence should be unpredictable in spite of any knowledge of previous bits in the sequence.
- **Backward unpredictability:** It should also not be feasible to determine the seed from knowledge of any generated values. No correlation between a seed and any value generated from that seed should be evident; each element of the sequence should appear to be the outcome of an independent random event whose probability is 1/2.

The same set of tests for randomness also provide a test of unpredictability. If the generated bit stream appears random, then it is not possible to predict some bit or bit sequence from knowledge of any previous bits. Similarly, if the bit sequence appears random, then there is no feasible way to deduce the seed based on the bit sequence. That is, a random sequence will have no correlation with a fixed value (the seed).

Seed Requirements For cryptographic applications, the seed that serves as input to the PRNG must be secure. Because the PRNG is a deterministic algorithm, if the adversary can deduce the seed, then the output can also be determined. Therefore, the seed must be unpredictable. In fact, the seed itself must be a random or pseudorandom number.

Typically, the seed is generated by a TRNG, as shown in Figure 8.2. This is the scheme recommended by SP 800-90A. The reader may wonder, if a TRNG is available, why it is necessary to use a PRNG. If the application is a stream cipher, then a TRNG is not practical. The sender would need to generate a keystream of

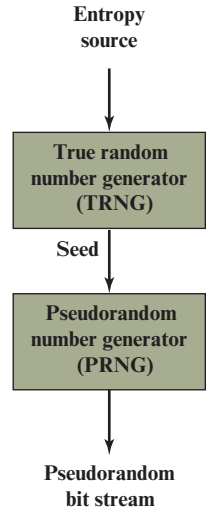


Figure 8.2 Generation of Seed Input to PRNG

bits as long as the plaintext and then transmit the keystream and the ciphertext securely to the receiver. If a PRNG is used, the sender need only find a way to deliver the stream cipher key, which is typically 128 or 256 bits, to the receiver in a secure fashion.

Even in the case of a PRF application, in which only a limited number of bits is generated, it is generally desirable to use a TRNG to provide the seed to the PRF and use the PRF output rather than use the TRNG directly. As is explained in Section 8.6, a TRNG may produce a binary string with some bias. The PRF would have the effect of conditioning the output of the TRNG so as to eliminate that bias.

Finally, the mechanism used to generate true random numbers may not be able to generate bits at a rate sufficient to keep up with the application requiring the random bits.

Algorithm Design

Cryptographic PRNGs have been the subject of much research over the years, and a wide variety of algorithms have been developed. These fall roughly into two categories.

- **Purpose-built algorithms:** These are algorithms designed specifically and solely for the purpose of generating pseudorandom bit streams. Some of these algorithms are used for a variety of PRNG applications; several of these are described in the next section. Others are designed specifically for use in a stream cipher. This topic is examined later in this chapter.
- **Algorithms based on existing cryptographic algorithms:** Cryptographic algorithms have the effect of randomizing input data. Indeed, this is a requirement of such algorithms. For example, if a symmetric block cipher produced

ciphertext that had certain regular patterns in it, it would aid in the process of cryptanalysis. Thus, cryptographic algorithms can serve as the core of PRNGs. SP 800-90A recommends three categories of such algorithms:

–**Symmetric block ciphers:** This approach is discussed in Section 8.3.

–**Hash functions and message authentication codes:** These approaches are examined in Chapter 12.

Any of these approaches can yield a cryptographically strong PRNG. A purpose-built algorithm may be provided by an operating system for general use. For applications that already use certain cryptographic algorithms for encryption or authentication, it makes sense to reuse the same code for the PRNG. Thus, all of these approaches are in common use.

8.2 PSEUDORANDOM NUMBER GENERATORS

In this section, we look at two types of algorithms for PRNGs.

Linear Congruential Generators

A widely used technique for pseudorandom number generation is an algorithm first proposed by Lehmer [LEHM51], which is known as the linear congruential method. The algorithm is parameterized with four numbers, as follows:

m	the modulus	$m > 0$
a	the multiplier	$0 < a < m$
c	the increment	$0 \leq c < m$
X_0	the starting value, or seed	$0 \leq X_0 < m$

The sequence of random numbers $\{X_n\}$ is obtained via the following iterative equation:

$$X_{n+1} = (aX_n + c) \bmod m$$

If m, a, c, and X_0 are integers, then this technique will produce a sequence of integers with each integer in the range $0 \leq X_n < m$.

The selection of values for a, c, and m is critical in developing a good random number generator. For example, consider $a = c = 1$. The sequence produced is obviously not satisfactory. Now consider the values $a = 7, c = 0, m = 32$, and $X_0 = 1$. This generates the sequence {7, 17, 23, 1, 7, etc.}, which is also clearly unsatisfactory. Of the 32 possible values, only four are used; thus, the sequence is said to have a period of 4. If, instead, we change the value of a to 5, then the sequence is {5, 25, 29, 17, 21, 9, 13, 1, 5, etc. }, which increases the period to 8.

We would like m to be very large, so that there is the potential for producing a long series of distinct random numbers. A common criterion is that m be nearly equal to the maximum representable nonnegative integer for a given computer. Thus, a value of m near to or equal to 2^{31} is typically chosen.

[PARK88] proposes three tests to be used in evaluating a random number generator:

T_1: The function should be a full-period generating function. That is, the function should generate all the numbers from 0 through $m - 1$ before repeating.
T_2: The generated sequence should appear random.
T_3: The function should implement efficiently with 32-bit arithmetic.

With appropriate values of *a, c*, and *m*, these three tests can be passed. With respect to T_1, it can be shown that if *m* is prime and $c = 0$, then for certain values of *a* the period of the generating function is $m - 1$, with only the value 0 missing. For 32-bit arithmetic, a convenient prime value of *m* is $2^{31} - 1$. Thus, the generating function becomes

$$X_{n+1} = (aX_n) \bmod (2^{31} - 1)$$

Of the more than 2 billion possible choices for *a*, only a handful of multipliers pass all three tests. One such value is $a = 7^5 = 16807$, which was originally selected for use in the IBM 360 family of computers [LEWI69]. This generator is widely used and has been subjected to a more thorough testing than any other PRNG. It is frequently recommended for statistical and simulation work (e.g., [JAIN91]).

The strength of the linear congruential algorithm is that if the multiplier and modulus are properly chosen, the resulting sequence of numbers will be statistically indistinguishable from a sequence drawn at random (but without replacement) from the set $1, 2, \ldots, m - 1$. But there is nothing random at all about the algorithm, apart from the choice of the initial value X_0. Once that value is chosen, the remaining numbers in the sequence follow deterministically. This has implications for cryptanalysis.

If an opponent knows that the linear congruential algorithm is being used and if the parameters are known (e.g., $a = 7^5, c = 0, m = 2^{31} - 1$), then once a single number is discovered, all subsequent numbers are known. Even if the opponent knows only that a linear congruential algorithm is being used, knowledge of a small part of the sequence is sufficient to determine the parameters of the algorithm. Suppose that the opponent is able to determine values for X_0, X_1, X_2, and X_3. Then

$$X_1 = (aX_0 + c) \bmod m$$
$$X_2 = (aX_1 + c) \bmod m$$
$$X_3 = (aX_2 + c) \bmod m$$

These equations can be solved for *a, c*, and *m*.

Thus, although it is nice to be able to use a good PRNG, it is desirable to make the actual sequence used nonreproducible, so that knowledge of part of the sequence on the part of an opponent is insufficient to determine future elements of the sequence. This goal can be achieved in a number of ways. For example, [BRIG79] suggests using an internal system clock to modify the random number stream. One way to use the clock would be to restart the sequence after every *N* numbers using the current clock value (mod *m*) as the new seed. Another way would be simply to add the current clock value to each random number (mod *m*).

Blum Blum Shub Generator

A popular approach to generating secure pseudorandom numbers is known as the Blum Blum Shub (BBS) generator (see Figure 8.3), named for its developers [BLUM86]. It has perhaps the strongest public proof of its cryptographic strength of any purpose-built algorithm. The procedure is as follows. First, choose two large prime numbers, p and q, that both have a remainder of 3 when divided by 4. That is,

$$p \equiv q \equiv 3 (\text{mod } 4)$$

This notation, explained more fully in Chapter 2, simply means that $(p \bmod 4) = (q \bmod 4) = 3$. For example, the prime numbers 7 and 11 satisfy $7 \equiv 11 \equiv 3 (\text{mod } 4)$. Let $n = p \times q$. Next, choose a random number s, such that s is relatively prime to n; this is equivalent to saying that neither p nor q is a factor of s. Then the BBS generator produces a sequence of bits B_i according to the following algorithm:

$$\begin{aligned} X_0 &= s^2 \bmod n \\ \textbf{for } i &= 1 \textbf{ to } \infty \\ X_i &= (X_{i-1})^2 \bmod n \\ B_i &= X_i \bmod 2 \end{aligned}$$

Thus, the least significant bit is taken at each iteration. Table 8.1 shows an example of BBS operation. Here, $n = 192649 = 383 \times 503$, and the seed $s = 101355$.

The BBS is referred to as a cryptographically secure pseudorandom bit generator (CSPRBG). A CSPRBG is defined as one that passes the *next-bit test*, which, in turn, is defined as follows [MENE97]: A pseudorandom bit generator is said to pass the next-bit test if there is not a polynomial-time algorithm[1] that, on input of the first k bits of an output sequence, can predict the $(k + 1)$st bit with probability significantly greater than 1/2. In other words, given the first k bits of the

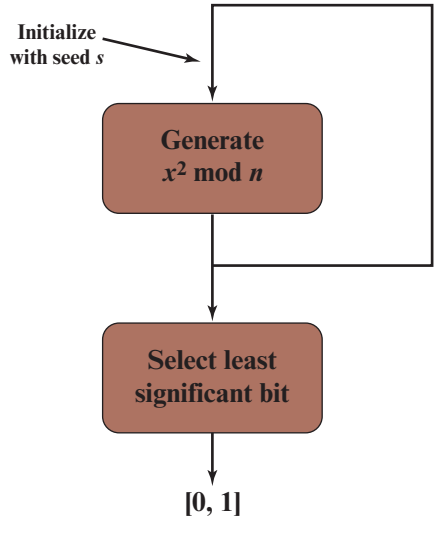


Figure 8.3 Blum Blum Shub Block Diagram

[1]A polynomial-time algorithm of order k is one whose running time is bounded by a polynomial of order k.

Table 8.1 Example Operation of BBS Generator

i	X_i	B_i
0	20749	
1	143135	1
2	177671	1
3	97048	0
4	89992	0
5	174051	1
6	80649	1
7	45663	1
8	69442	0
9	186894	0
10	177046	0
11	137922	0
12	123175	1
13	8630	0
14	114386	0
15	14863	1
16	133015	1
17	106065	1
18	45870	0
19	137171	1
20	48060	0

sequence, there is not a practical algorithm that can even allow you to state that the next bit will be 1 (or 0) with probability greater than 1/2. For all practical purposes, the sequence is unpredictable. The security of BBS is based on the difficulty of factoring n. That is, given n, we need to determine its two prime factors p and q.

8.3 PSEUDORANDOM NUMBER GENERATION USING A BLOCK CIPHER

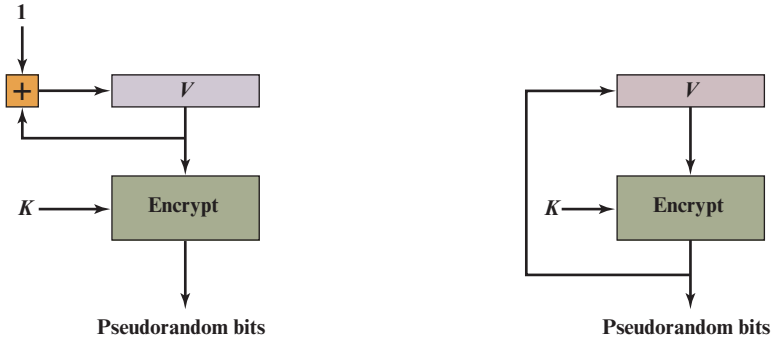
A popular approach to PRNG construction is to use a symmetric block cipher as the heart of the PRNG mechanism. For any block of plaintext, a symmetric block cipher produces an output block that is apparently random. That is, there are no patterns or regularities in the ciphertext that provide information that can be used to deduce the plaintext. Thus, a symmetric block cipher is a good candidate for building a pseudorandom number generator.

If an established, standardized block cipher is used, such as DES or AES, then the security characteristics of the PRNG can be established. Further, many applications already make use of DES or AES, so the inclusion of the block cipher as part of the PRNG algorithm is straightforward.

PRNG Using Block Cipher Modes of Operation

Two approaches that use a block cipher to build a PNRG have gained widespread acceptance: the CTR mode and the OFB mode. The CTR mode is recommended in NIST SP 800-90A, in the ANSI standard X9.82 (*Random Number Generation*), and in RFC 4086 (*Randomness Requirements for Security,* June 2005). The OFB mode is recommended in X9.82 and RFC 4086.

Figure 8.4 illustrates the two methods. In each case, the seed consists of two parts: the encryption key value and a value V that will be updated after each block of pseudorandom numbers is generated. Thus, for AES-128, the seed consists of a 128-bit key and a 128-bit V value. In the CTR case, the value of V is incremented



(a) CTR mode (b) OFB mode

Figure 8.4 PRNG Mechanisms Based on Block Ciphers

by 1 after each encryption. In the case of OFB, the value of V is updated to equal the value of the preceding PRNG block. In both cases, pseudorandom bits are produced one block at a time (e.g., for AES, PRNG bits are generated 128 bits at a time).

The CTR algorithm for PRNG, called CTR_DRBG, can be summarized as follows.

```
while (len (temp) < requested_number_of_bits) do
      V = (V + 1) mod 2^128
      output_block = E(Key, V)
      temp = temp || output_block
```

The OFB algorithm can be summarized as follows.

```
while (len (temp) < requested_number_of_bits) do
      V = E(Key, V)
      temp = temp || V
```

To get some idea of the performance of these two PRNGs, consider the following short experiment. A random bit sequence of 256 bits was obtained from random.org, which uses three radios tuned between stations to pick up atmospheric noise. These 256 bits form the seed, allocated as

Key:	**cfb0ef3108d49cc4562d5810b0a9af60**
V:	**4c89af496176b728ed1e2ea8ba27f5a4**

The total number of one bits in the 256-bit seed is 124, or a fraction of 0.48, which is reassuringly close to the ideal of 0.5.

For the OFB PRNG, Table 8.2 shows the first eight output blocks (1024 bits) with two rough measures of security. The second column shows the fraction of one bits in each 128-bit block. This corresponds to one of the NIST tests. The results indicate that the output is split roughly equally between zero and one bits. The third column shows the fraction of bits that match between adjacent blocks. If this number

Table 8.2 Example Results for PRNG Using OFB

Output Block	Fraction of One Bits	Fraction of Bits that Match with Preceding Block
1786f4c7ff6e291dbdfdd90ec3453176	0.57	—
5e17b22b14677a4d66890f87565eae64	0.51	0.52
fd18284ac82251dfb3aa62c326cd46cc	0.47	0.54
c8e545198a758ef5dd86b41946389bd5	0.50	0.44
fe7bae0e23019542962e2c52d215a2e3	0.47	0.48
14fdf5ec99469598ae0379472803accd	0.49	0.52
6aeca972e5a3ef17bd1a1b775fc8b929	0.57	0.48
f7e97badf359d128f00d9b4ae323db64	0.55	0.45

Table 8.3 Example Results for PRNG Using CTR

Output Block	Fraction of One Bits	Fraction of Bits that Match with Preceding Block
1786f4c7ff6e291dbdfdd90ec3453176	0.57	—
60809669a3e092a01b463472fdcae420	0.41	0.41
d4e6e170b46b0573eedf88ee39bff33d	0.59	0.45
5f8fcfc5deca18ea246785d7fadc76f8	0.59	0.52
90e63ed27bb07868c753545bdd57ee28	0.53	0.52
0125856fdf4a17f747c7833695c52235	0.50	0.47
f4be2d179b0f2548fd748c8fc7c81990	0.51	0.48
1151fc48f90eebac658a3911515c3c66	0.47	0.45

differs substantially from 0.5, that suggests a correlation between blocks, which could be a security weakness. The results suggest no correlation.

Table 8.3 shows the results using the same key and V values for CTR mode. Again, the results are favorable.

NIST CTR_DRBG

We now look more closely at the details of the PRNG defined in NIST SP 800-90A based on the CTR mode of operation. The PRNG is referred to as CTRDRBG (counter mode–deterministic random bit generator). CTR_DRBG is widely implemented and is part of the hardware random number generator implemented on all recent Intel processor chips (discussed in Section 8.6).

The DRBG assumes that an entropy source is available to provide random bits. Typically, the entropy source will be a TRNG based on some physical source. Other sources are possible if they meet the required entropy measure of the application. Entropy is an information theoretic concept that measures unpredictability, or randomness; see Appendix B for details. The encryption algorithm used in the DRBG may be 3DES with three keys or AES with a key size of 128, 192, or 256 bits.

Table 8.4 CTR_DRBG Parameters

	3DES	AES-128	AES-192	AES-256
outlen	64	128	128	128
keylen	168	128	192	256
seedlen	232	256	320	384
reseed_interval	$\leq 2^{32}$	$\leq 2^{48}$	$\leq 2^{48}$	$\leq 2^{48}$

Four parameters are associated with the algorithm:

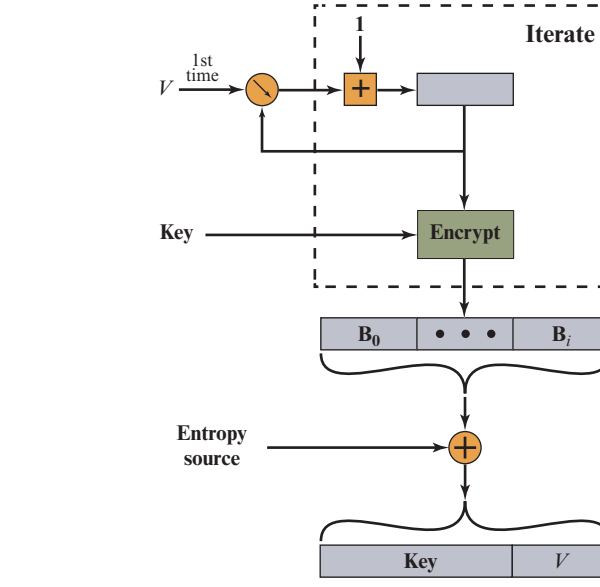
- **Output block length** (*outlen*): Length of the output block of the encryption algorithm.
- **Key length** (*keylen*): Length of the encryption key.
- **Seed length** (*seedlen*): The seed is a string of bits that is used as input to a DRBG mechanism. The seed will determine a portion of the internal state of the DRBG, and its entropy must be sufficient to support the security strength of the DRBG. $seedlen = outlen + keylen$.
- **Reseed interval** (*reseed_interval*): Length of the encryption key. It is the maximum number of output blocks generated before updating the algorithm with a new seed.

Table 8.4 lists the values specified in SP 800-90A for these parameters.

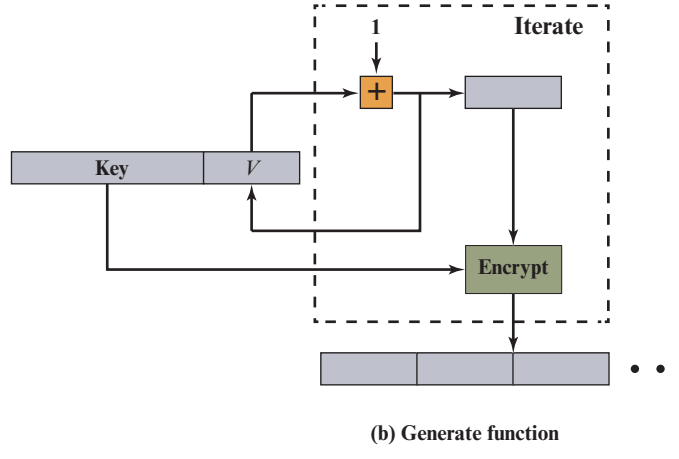
INITIALIZE Figure 8.5 shows the two principal functions that comprise CTR_DRBG. We first consider how CTR_DRBG is initialized, using the initialize and update function (Figure 8.5a). Recall that the CTR block cipher mode requires both an encryption key K and an initial counter value, referred to in SP 800-90A as the counter V. The combination of K and V is referred to as the *seed*. To start the DRGB operation, initial values for K and V are needed, and can be chosen arbitrarily. As an example, the Intel Digital Random Number Generator, discussed in Section 8.6, uses the values $K = 0$ and $V = 0$. These values are used as parameters for the CTR mode of operation to produce at least *seedlen* bits. In addition, exactly *seedlen* bits must be supplied from what is referred to as an *entropy source*. Typically, the entropy source would be some form of TRNG.

With these inputs, the CTR mode of encryption is iterated to produce a sequence of output blocks, with V incremented by 1 after each encryption. The process continues until at least *seedlen* bits have been generated. The leftmost *seedlen* bits of output are then XORed with the *seedlen* entropy bits to produce a new seed. In turn, the leftmost *keylen* bits of the seed form the new key and the rightmost *outlen* bits of the seed form the new counter value V.

GENERATE Once values of Key and V are obtained, the DRBG enters the generate phase and is able to generate pseudorandom bits, one output block at a time (Figure 8.5b). The encryption function is iterated to generate the number of pseudorandom bits desired. Each iteration uses the same encryption key. The counter value V is incremented by 1 for each iteration.



(a) Initialize and update function



(b) Generate function

Figure 8.5 CTR_DRBG Functions

UPDATE To enhance security, the number of bits generated by any PRNG should be limited. CTR_DRGB uses the parameter *reseed_interval* to set that limit. During the generate phase, a reseed counter is initialized to 1 and then incremented with each iteration (each production of an output block). When the reseed counter reaches *reseed_interval*, the update function is invoked (Figure 8.5a). The update function is the same as the initialize function. In the update case, the Key and *V* values last used by the generate function serve as the input parameters to the update function. The update function takes *seedlen* new bits from an entropy source and produces a new seed (Key, *V*). The generate function can then resume production of pseudorandom

bits. Note that the result of the update function is to change both the Key and V values used by the generate function.

8.4 STREAM CIPHERS

Stream ciphers can be viewed a pseudorandom equivalent of a one-time pad. The one-time pad uses a long random key, of length equal to the plaintext message. A stream cipher uses a short secret key and a pseudorandomly generated stream of bits, computationally indistinguishable from a stream of random digits. Traditionally, block ciphers have been more widely used, in a greater range of applications. This is primarily due to the ability of block ciphers to easily be used in a variety of ways using different modes of operation. In addition, block ciphers can be used as stream ciphers via modes of operation such as Counter, OFB, and CBC.

In recent years, there has been a resurgence of interest in the use of stream ciphers [BIRY04]. Stream ciphers are useful when there is a need to encrypt large amounts of fast streaming data. And stream ciphers are well suited to use in devices with very limited memory and processing power, called **constrained devices**. Examples include small wireless sensors as part of an Internet of Things (IoT) and radio frequency identification (RFID) tags.

Figure 8.6 shows the structure of a typical stream cipher. There are three internal elements. There is a secret **state** s_i (i.e., memory) that evolves with time during encryption and decryption; the initial state is designated as s_0. A **state transition function** f, at each bit generation time, computes a new state value from the old state value. An **output function** g produces the stream of bits used for encryption and decryption, known as the **keystream** z_i. A secret key K provides input to the stream cipher, and is used to initialize the state. K may also serve as an input parameter to f. Some stream ciphers also include an initialization vector IV that is

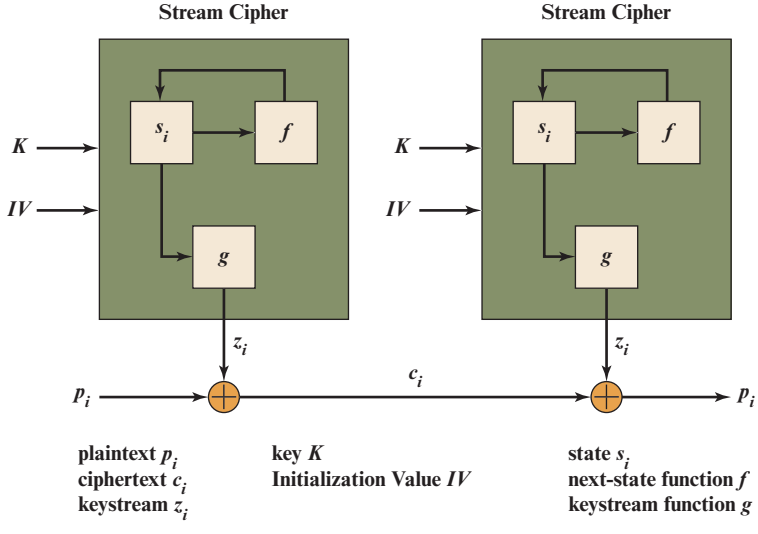


Figure 8.6 Generic Structure of a Typical Stream Cipher

used, along with K, to initialize the state. As is the case for block ciphers, the IV for a stream cipher need not be secret. However, it should be unpredictable and unique.

The keystream is combined one byte at a time with the plaintext stream using the bitwise exclusive-OR (XOR) operation. For example, if the next byte generated by the generator is 01101100 and the next plaintext byte is 11001100, then the resulting ciphertext byte is

```
  11001100 plaintext
⊕ 01101100 key stream
  10100000 ciphertext
```

Decryption requires the use of the same pseudorandom sequence:

```
  10100000 ciphertext
⊕ 01101100 key stream
  11001100 plaintext
```

The stream cipher is similar to the one-time pad discussed in Chapter 3. The difference is that a one-time pad uses a genuine random number stream, whereas a stream cipher uses a pseudorandom number stream.

[KUMA97] lists the following important design considerations for a stream cipher.

1. The encryption sequence should have a large period. A pseudorandom number generator uses a function that produces a deterministic stream of bits that eventually repeats. The longer the period of repeat the more difficult it will be to do cryptanalysis. This is essentially the same consideration that was discussed with reference to the Vigenère cipher, namely that the longer the keyword the more difficult the cryptanalysis.
2. The keystream should approximate the properties of a true random number stream as close as possible. For example, there should be an approximately equal number of 1s and 0s. If the keystream is treated as a stream of bytes, then all of the 256 possible byte values should appear approximately equally often. The more random-appearing the keystream is, the more randomized the ciphertext is, making cryptanalysis more difficult.
3. Note from Figure 8.6 that the output of the pseudorandom number generator is conditioned on the value of the input key. To guard against brute-force attacks, the key needs to be sufficiently long. The same considerations that apply to block ciphers are valid here. Thus, with current technology, a key length of at least 128 bits is desirable.

With a properly designed pseudorandom number generator, a stream cipher can be as secure as a block cipher of comparable key length. A potential advantage of a stream cipher is that stream ciphers that do not use block ciphers as a building block are typically faster and use far less code than do block ciphers. The example in this chapter, RC4, can be implemented in just a few lines of code. In recent years, this advantage has diminished with the introduction of AES, which is quite efficient in software. Furthermore, hardware acceleration techniques are now available for AES. For example, the Intel AES Instruction Set has machine instructions for one round of encryption and decryption and key generation. Using the hardware instructions results in speedups of about an order of magnitude compared to pure software implementations [XU10].

One advantage of a block cipher is that you can reuse keys. In contrast, if two plaintexts are encrypted with the same key using a stream cipher, then cryptanalysis is often quite simple [DAWS96]. If the two ciphertext streams are XORed together, the result is the XOR of the original plaintexts. If the plaintexts are text strings, credit card numbers, or other byte streams with known properties, then cryptanalysis may be successful.

For applications that require encryption/decryption of a stream of data, such as over a data communications channel or a browser/Web link, a stream cipher might be the better alternative. For applications that deal with blocks of data, such as file transfer, email, and database, block ciphers may be more appropriate. However, either type of cipher can be used in virtually any application.

A stream cipher can be constructed with any cryptographically strong PRNG, such as the ones discussed in Sections 8.2 and 8.3. In the next section, we look at a stream cipher that uses a PRNG designed specifically for the stream cipher.

8.5 RC4

RC4 is a stream cipher designed in 1987 by Ron Rivest for RSA Security. It is a variable key size stream cipher with byte-oriented operations. The algorithm is based on the use of a random permutation. Analysis shows that the period of the cipher is overwhelmingly likely to be greater than 10^{100} [ROBS95a]. Eight to sixteen machine operations are required per output byte, and the cipher can be expected to run very quickly in software. RC4 is used in the WiFi Protected Access (WPA) protocol that are part of the IEEE 802.11 wireless LAN standard. It is optional for use in Secure Shell (SSH) and Kerberos. RC4 was kept as a trade secret by RSA Security. In September 1994, the RC4 algorithm was anonymously posted on the Internet on the Cypherpunks anonymous remailers list.

The RC4 algorithm is remarkably simple and quite easy to explain. A variable-length key of from 1 to 256 bytes (8 to 2048 bits) is used to initialize a 256-byte state vector S, with elements S[0],S[1], . . . ,S[255]. At all times, S contains a permutation of all 8-bit numbers from 0 through 255. For encryption and decryption, a byte k is generated from S by selecting one of the 255 entries in a systematic fashion. As each value of k is generated, the entries in S are once again permuted.

Initialization of S

To begin, the entries of S are set equal to the values from 0 through 255 in ascending order; that is, S[0] = 0, S[1] = 1, . . . , S[255] = 255. A temporary vector, T, is also created. If the length of the key K is 256 bytes, then K is transferred to T. Otherwise, for a key of length *keylen* bytes, the first *keylen* elements of T are copied from K, and then K is repeated as many times as necessary to fill out T. These preliminary operations can be summarized as

```
/* Initialization */
for i = 0 to 255 do
 S[i] = i;
 T[i] = K[i mod keylen];
```

Next we use T to produce the initial permutation of S. This involves starting with S[0] and going through to S[255], and for each S[i], swapping S[i] with another byte in S according to a scheme dictated by T[i]:

```
/* Initial Permutation of S */
j = 0;
for i = 0 to 255 do
    j = (j + S[i] + T[i]) mod 256;
    Swap (S[i], S[j]);
```

Because the only operation on S is a swap, the only effect is a permutation. S still contains all the numbers from 0 through 255.

Stream Generation

Once the S vector is initialized, the input key is no longer used. Stream generation involves cycling through all the elements of S[i], and for each S[i], swapping S[i] with another byte in S according to a scheme dictated by the current configuration of S. After S[255] is reached, the process continues, starting over again at S[0]:

```
/* Stream Generation */
i, j = 0;
while (true)
  i = (i + 1) mod 256;
  j = (j + S[i]) mod 256;
  Swap (S[i], S[j]);
  t = (S[i] + S[j]) mod 256;
  k = S[t];
```

To encrypt, XOR the value k with the next byte of plaintext. To decrypt, XOR the value *k* with the next byte of ciphertext.

Figure 8.7 illustrates the RC4 logic.

Strength of RC4

More recently, [PAUL07] revealed a more fundamental vulnerability in the RC4 key scheduling algorithm that reduces the amount of effort to discover the key. Recent cryptanalysis results [ALFA13] exploit biases in the RC4 keystream to recover repeatedly encrypted plaintexts. As a result of the discovered weaknesses, particularly those reported in [ALFA13], the IETF issued RFC 7465 prohibiting the use of RC4 in TLS (*Prohibiting RC4 Cipher Suites*, February 2015). In its latest TLS guidelines, NIST also prohibited the use of RC4 for government use (SP 800-52, *Guidelines for the Selection, Configuration, and Use of Transport Layer Security (TLS) Implementations*, September 2013).

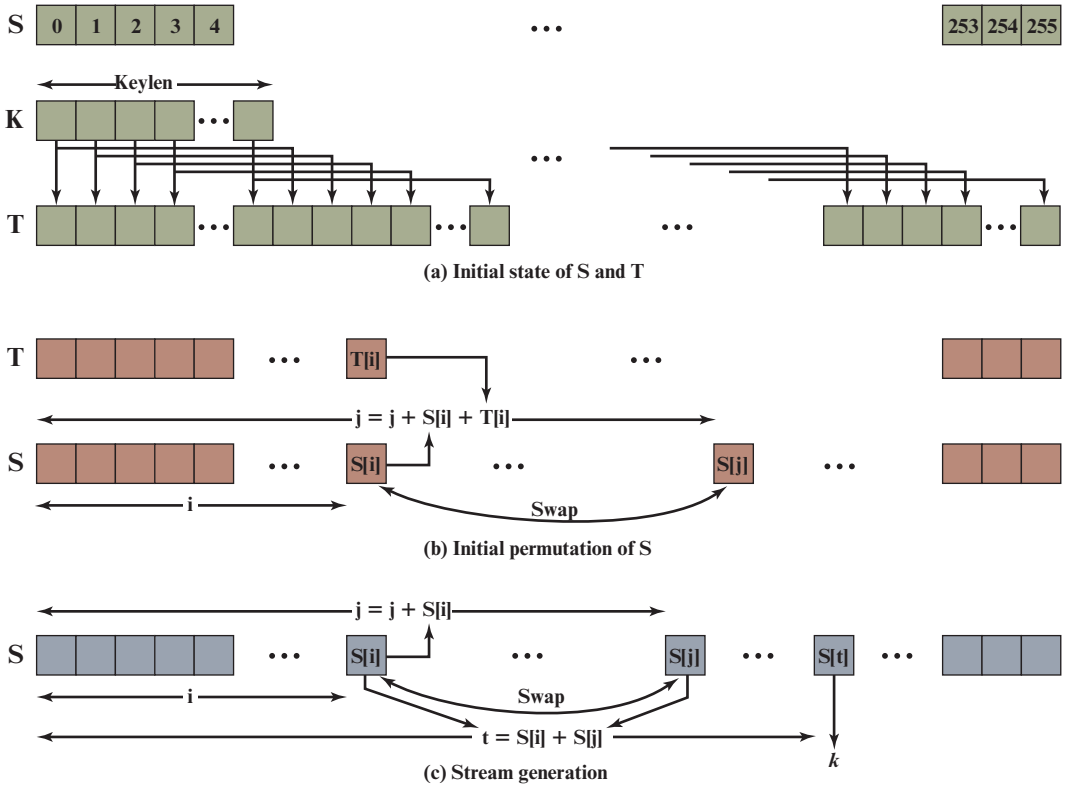


Figure 8.7 RC4

8.6 STREAM CIPHERS USING FEEDBACK SHIFT REGISTERS

With the increasing use of highly constrained devices, such as those used in the IoT, there has been increasing interest in developing new stream ciphers that take up minimal memory, are highly efficient, and have minimal power consumption requirements. Most of the recently developed stream ciphers are based on the use of feedback shift registers (FSRs). Feedback shift registers exhibit the desired performance behavior, are well-suited to compact hardware implementation, and there are well-developed theoretical results on the statistical properties of the bit sequences they produce.

An FSR consists of a sequence of 1-bit memory cells. Each cell has an output line, which indicates the value currently stored, and an input line. At discrete time instants, known as clock times, the value in each storage device is replaced by the value indicated by its input line. The effect is as follows: The rightmost (least significant) bit is shifted out as the output bit for this clock cycle. The other bits are shifted one bit position to the right. The new leftmost (most significant) bit is calculated as a function of the other bits in the FSR.

This section introduces the two types of feedback shift registers: linear feedback shift registers (LFSRs) and nonlinear feedback shift registers. We then examine a contemporary example: the Grain stream cipher.

Linear Feedback Shift Registers

In general, a function f is linear if $f(x + y) = f(x) + f(y)$, and $af(x) = f(ax)$. For the specific case of an FSR, an FSR is linear if the feedback function only involves modulo-2 (logical exclusive-OR) addition of bits in the register.

The circuit is implemented as follows:

1. The LFSR contains n bits.
2. There are from 1 to $(n - 1)$ XOR gates.
3. The presence or absence of a gate corresponds to the presence or absence of a term in the characteristic polynomial (explained subsequently), $P(X)$, excluding the X^n term.

Two equivalent ways of characterizing the LFSR are used. We can think of the generator as implementing a sum of XOR terms:

$$B_n = A_1 B_{n-1}\, A_2 B_{n-2}\, A_3 B_{n-3} \bullet\bullet\bullet A_n B_0 = \sum_{i=1}^{n} A_i B_{n-i} \qquad \textbf{(8.1)}$$

Figure 8.8 illustrates this equation. At each clock signal, the B_i values are calculated and shifted right. Thus, the calculated value of B_n becomes the value in the B_{n-1} cell, and so on down to B_0, which is shifted out as the output bit. An actual implementation would not have the multiply circuits; instead, for $A_i = 0$, the corresponding XOR circuit is eliminated. Figure 8.9a is an example of a 4-bit LFSR that implements the equation:

$$B_4 = B_0 \oplus B_1 \qquad \textbf{(8.2)}$$

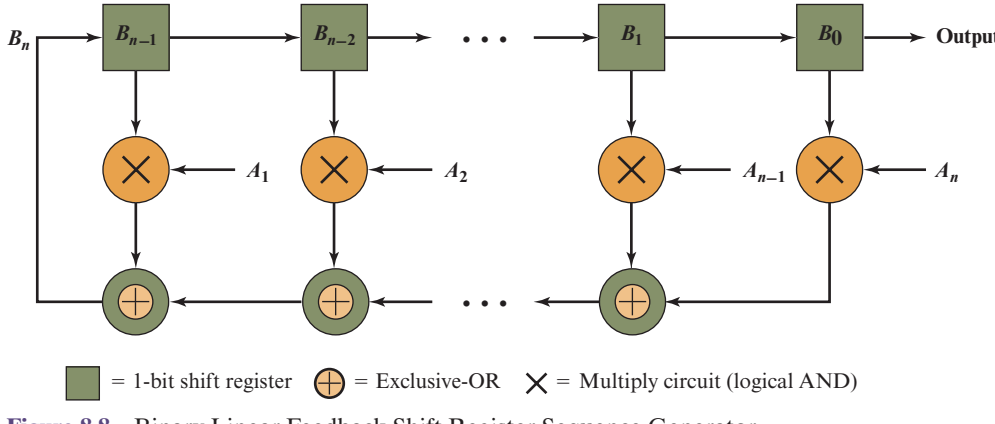
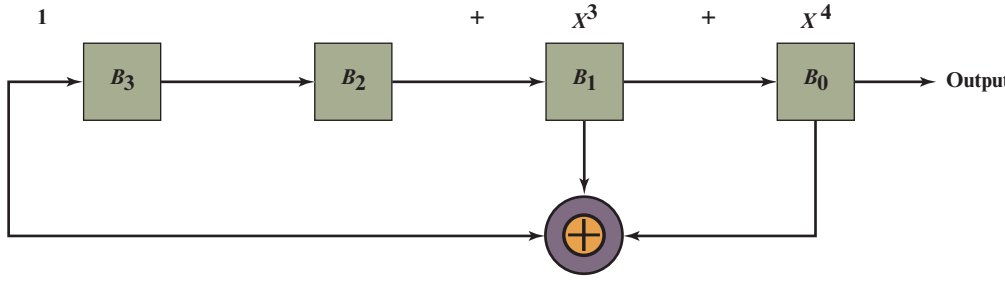


Figure 8.8 Binary Linear Feedback Shift Register Sequence Generator



(a) Shift-register implementation

State	B_3	B_2	B_1	B_0	B_0 Å B_1	output
Initial = 0	1	0	0	0	0	0
1	0	1	0	0	0	0
2	0	0	1	0	1	0
3	1	0	0	1	1	1
4	1	1	0	0	0	0
5	0	1	1	0	1	0
6	1	0	1	1	0	1
7	0	1	0	1	1	1
8	1	0	1	0	1	0
9	1	1	0	1	1	1
10	1	1	1	0	1	0
11	1	1	1	1	0	1
12	0	1	1	1	0	1
13	0	0	1	1	0	1
14	0	0	0	1	1	1
15 = 0	1	0	0	0	0	0

(b) Example with initial state of 1000

Figure 8.9 4-Bit Linear Feedback Shift Register

The shift register technique has several important advantages. The sequences generated by an LFSR can be nearly random with long periods. In addition, LFSRs are easy to implement in hardware and can run at high speeds.

It can be shown that the output of an n-bit LFSR is periodic with maximum period $N = 2^n - 1$. The all-zeros sequence occurs only if either the initial contents of the LFSR are all zero or the coefficients in Equation (8.1) are all zero (no feedback). A feedback configuration can always be found that gives a period of N; the resulting sequences are called **maximal-length sequences**, or **m-sequences**.

Figure 8.9b shows the generation of an m-sequence for the LFSR of Figure 8.9a. The LFSR implements Equation (8.2) with an initial state of 1000 ($B_3 = 1, B_2 = 0, B_1 = 0, B_0 = 0$). Figure 8.9b shows the step-by-step operation as the LFSR is clocked one bit at a time. Each row of the table shows the values currently stored in the four shift register elements. In addition, the row shows the value that appears at the output of the exclusive-OR circuit. Finally, the row shows the value of the output bit, which is just B_0. Note that the output repeats after 15 bits. That is, the period of the sequence, or the length of the m-sequence, is $15 = 2^4 - 1$. This same periodic m-sequence is generated regardless of the initial state of the LFSR (except for 0000), as shown in Figure 8.9. With each different initial state, the m-sequence begins at a different point in its cycle, but it is the same sequence.

For any given size of LFSR, a number of different unique m-sequences can be generated by using different values for the A_i in Equation (8.1).

An equivalent definition of an LFSR configuration is a **characteristic polynomial**. The characteristic polynomial $P(X)$ that corresponds to Equation (8.1) has the form:

$$P(X) = 1 + A_1X + A_2X^2 + \bullet\bullet\bullet + A_{n-1}X^{n-1} + A_nX^n = 1 + \sum_{i=1}^{n} A_i X^i \quad \textbf{(8.3)}$$

One useful attribute of the characteristic polynomial is that it can be used to find the sequence generated by the corresponding LFSR, by taking the reciprocal of the polynomial. For example, for the 3-bit LSFR with $P(X) = 1 + X + X^3$,

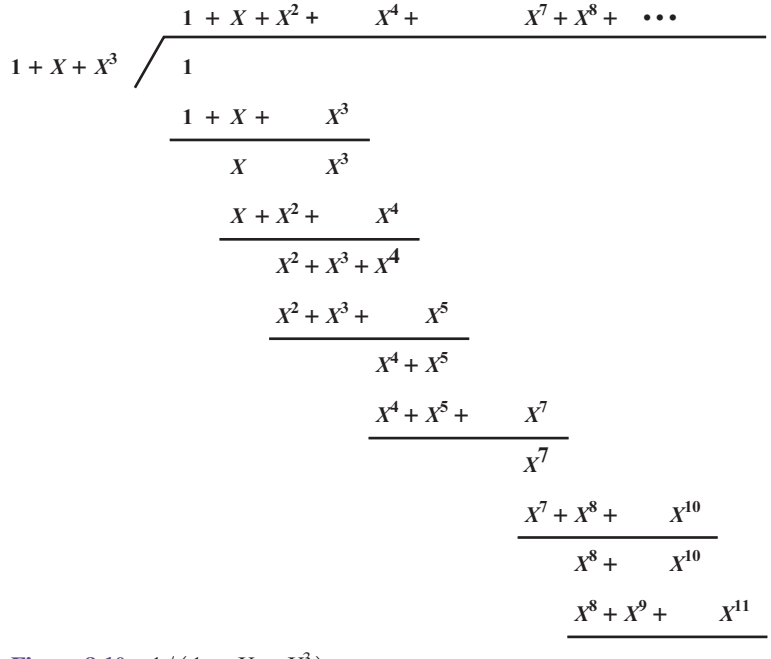


Figure 8.10 $1/(1 + X + X^3)$

we perform the division $1/(1 + X + X^3)$. Figure 8.10 depicts the long division. The result is:

$$1 + X + X^2 + (0 \times X^3) + X^4 + (0 \times X^5) + (0 \times X^6)$$

after which the pattern repeats. This means that the shift register output is 1110100.

Because the period of this sequence is $7 = 2^3 - 1$, this is an m-sequence. Notice that we are doing division somewhat differently from the normal method. This is because the subtractions are done modulo 2, or using the XOR function, and in this system, subtraction produces the same result as addition.

A characteristic polynomial produces an m-sequence if and only if it is a primitive polynomial.[2] Thus, $P(X) = 1 + X + X^3$ is a primitive polynomial. Similarly, the polynomial corresponding to Figure 8.9a is $P(X) = 1 + X + X^4$, which is a primitive polynomial.

Alternatively, some sources in the literature define a **generating polynomial** as follows:

$$G(X) = X^n P\left(\frac{1}{X}\right) = X^n + \sum_{i=1}^{n} A_i X^{n-i}$$

There is no practical difference; both $P(X)$ and $G(X)$ generate the same output bit sequence.

Although a LFSR defined by a primitive polynomial produces a good pseudorandom number bit stream, a single LFSR by itself is not suitable as a stream cipher. The stream cipher would simply consist of taking the XOR of successive bits of plaintext with successive bits generated by the LFSR. If an n-bit LFSR is used as a stream cipher, then the initial contents of the register constitute the key. It can be shown that if the feedback function is known (i.e., the values of the A_i are known) and if an adversary can determine n consecutive bits of the stream, then the adversary can determine the entire stream. This is due to the linearity of the feedback function. Further, if the feedback function is not known, then $2n$ bits of the output stream suffice to determine the entire stream.

One way to develop an LFSR-based stream cipher is to use multiple LFSRs, perhaps of different lengths, that are combined in some fashion. Another way is to incorporate a nonlinear feedback shift register (NFSR).

Nonlinear Feedback Shift Registers

The term *linear*, in the context of LFSR, means that the coefficients A_i in Equations 8.1 and 8.3 are constants; in particular these are Boolean constants (0 or 1). For an NFSR, the coefficients may be variables. An example is Figure 8.11, which can be expressed as:

$$B_5 = B_4 \oplus B_3 B_2$$

[2]Primitive polynomials are defined in Chapter 5.

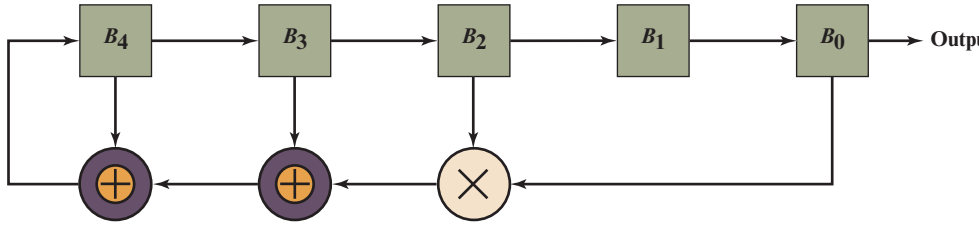


Figure 8.11 A Nonlinear Feedback Shift Register

or, equivalently:

$$P(X) = 1 + X + X^2X^4$$

As with LFSRs, an NFSR is not by itself suitable as a stream cipher. There is no theory to analyze them. However, it may be combined with an NFSR to produce a stream cipher of known maximum period and high security.

Grain-128a

Grain is a family of hardware-efficient stream ciphers. Grain was accepted as part of the eSTREAM effort to approve a number of new stream ciphers (described in Chapter 23). The eSTREAM specification, called Grain v1, defines two stream ciphers, one with an 80-bit key and a 64-bit initialization vector (IV), and one with a 128-bit key and 80-bit IV. Grain has since been revised and expanded to include authentication, referred to as Grain-128a [AGRE11, HELL06]. The eSTREAM final report [BABB08] states that Grain has pushed the state of the art in terms of compact implementation.

Grain-128a consists of two shift registers, one with linear feedback and the second with nonlinear feedback, and a filter function. The registers are coupled by very lightweight, but judiciously chosen Boolean functions. The LFSR guarantees a minimum period for the keystream, and it also provides balancedness in the output. The NFSR, together with a nonlinear filter, introduces nonlinearity to the cipher. The input to the NFSR is masked with the output of the LFSR so that the state of the NFSR is balanced.

Output for Encryption Figure 8.12a shows the structure of Grain-128a for producing a stream of output bits to be used for encrypting a stream of plaintext by a simple bitwise XOR operation. Grain-128a uses a convention of numbering the bits in the registers increasing from left to right and doing a left shift, with the leftmost bit as output. The LFSR at iteration i is defined as follows:

$$s_{i+128} = s_i \oplus s_{i+7} \oplus s_{i+38} \oplus s_{i+70} \oplus s_{i+81} \oplus s_{i+96}$$

The equivalent generator function is:

$$f(x) = 1 + x^{32} + x^{47} + x^{58} + x^{90} + x^{121} + x^{128}$$

The NFSR is defines as follows:

$$\begin{aligned} b_{i+128} &= s_i \oplus b_i \oplus b_{i+26} \oplus b_{i+56} \oplus b_{i+91} \oplus b_{i+96} \\ &\oplus b_{i+3}b_{i+67} \oplus b_{i+11}b_{i+13} \oplus b_{i+17}b_{i+18} \\ &\oplus b_{i+27}b_{i+59} \oplus b_{i+40}b_{i+48} \oplus b_{i+61}b_{i+65} \\ &\oplus b_{i+68}b_{i+84} \oplus b_{i+88}b_{i+92}b_{i+93}b_{i+95} \\ &\oplus b_{i+22}b_{i+24}b_{i+25} \oplus b_{i+70}b_{i+78}b_{i+82} \end{aligned}$$

The equivalent generator function, which is a primitive polynomial, is:

$$\begin{aligned} g1(x) &= 1 + x^{32} + x^{37} + x^{72} + x^{102} + x^{128} \\ g2(x) &= x^{44}x^{60} + x^{61}x^{125} + x^{63}x^{67} + x^{69}x^{101} + x^{80}x^{88} + x^{110}x^{111} \\ &\quad + x^{115}x^{117} + x^{46}x^{50}x^{58} + x^{103}x^{104}x^{106} + x^{33}x^{35}x^{36}x^{40} \\ g(x) &= g1(x) + g2(x) \end{aligned}$$

Thus, the NFSR output has both linear and nonlinear components. Note that the generator function for the NFSR does not feed directly back into the register but is XORed with the LFSR output s_i, which masks the input to the NFSR.

The actual generation of an output bit from the grain structure proceeds in several stages. The filter function h takes 9 variables from the two shift registers. It is designed to be balanced, highly nonlinear, and produce secure output. It is defined as:

$$h = b_{i+12}s_{i+8} \oplus s_{i+13}s_{i+20} \oplus s_{i+95}s_{i+42} \oplus s_{i+60}s_{i+79}s_{i+94}$$

Next, a pre-output function masks h with 1 bit of the LFSR and 7 bits of the NFSR, using the following simple linear function:

$$y_i = h \oplus s_{i+93} \oplus \sum_{j \in A} b_{i+j}$$

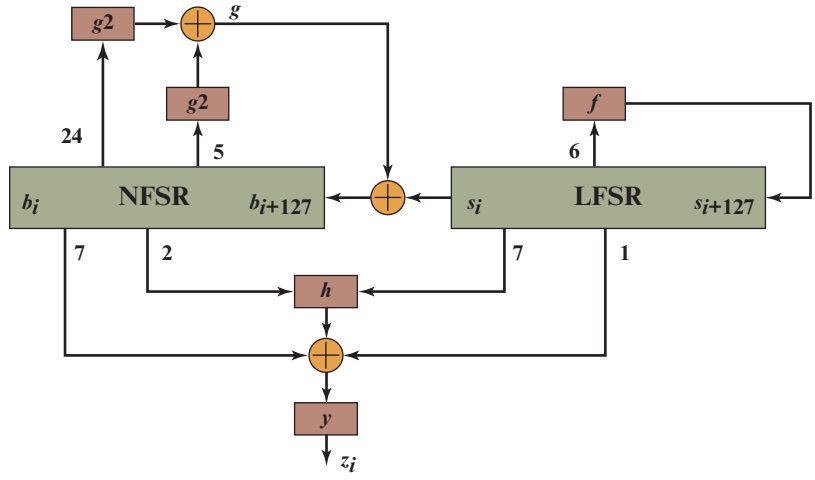
where $A = \{2, 15, 36, 45, 64, 73, 89\}$. The output function is defined as

$$z_i = y_{64+2i}$$

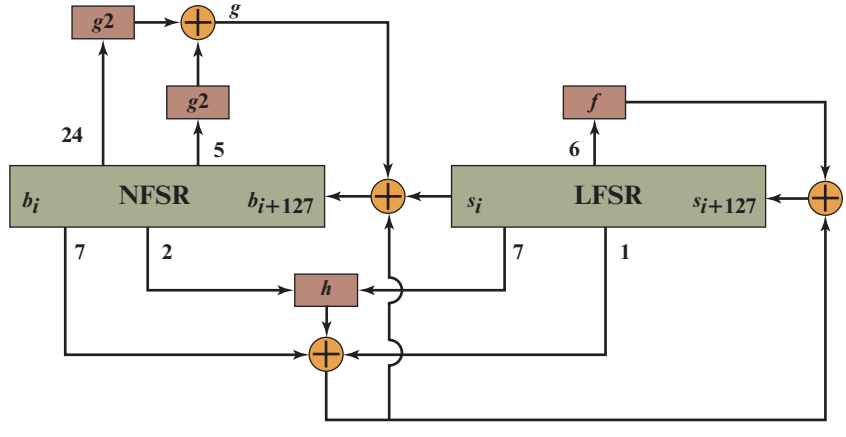
That is, the output consists of every second bit after skipping the first 64 bits. These 64 initial bits and the other half may be used for authentication, as described subsequently.

Because the LFSR is 128 bits and uses a primitive polynomial, the period is guaranteed to be at least $2^{128} - 1$. Because of the NFSR and the fact that the input to this is masked with the output of the LFSR, the exact period will depend on the key and the IV used. The input to the NFSR is masked with the output of the LFSR in order to make sure that the NFSR state is balanced.

KEY AND IV INITIALIZATION Grain-128a is initialized by placing the 128-bit key in the 128-bit NFSR. The 128-bit LFSR is initialized using the 96 bits IV_i of the IV as follows:



(a) Output generator



(b) Key initialization

Figure 8.12 Grain-128a Stream Cipher

$$s_i = \begin{cases} IV_i & 0 \le i \le 95 \\ 1 & 96 \le i \le 126 \\ 0 & i = 127 \end{cases}$$

Then, the two registers, comprising 256 bits, are clocked 256 times without producing any keystream. Instead, the pre-output function is fed back and XORed with the input to both the NFSR and LFSR (Figure 8.12b). This operation fully replaces the IV and key with the initial state of the registers. This process effectively scrambles the contents of the shift registers before the keystream is generated.

Encryption Encryption is now easily defined. Assume a message of length L defined by the bits $m_0, \ldots, m_{L-1}$. Then the ciphertext bits c_i are calculated as:

$$c_i = z_i \oplus m_i$$

And the message is recovered from the ciphertext as follows:

$$m_i = z_i \oplus c_i$$

Authentication Optionally, Grain-128a generates a 32-bit authentication tag. For this purpose, there is a 32-bit register called the accumulator, with the bits at time i denoted by $a_i^0, \ldots, a_i^{31}$. There is also a 32-bit shift register, with the bits at time i denoted by $r_i, \ldots, r_{i+31}$. The accumulator is initialized with the first 32 bits of y_i and the register is initialized with the second sequence of 32 bits of y_i. Recall that these 64 bits were excluded in forming z_i. At each time i, the shift register is updated by assigning $r_{i+32} = y_{64+2i+1}$, and then shifting left 1 bit. Thus, the bits not used in encryption are used for authentication. At each time i, all of the bits of the accumulator are updated as $a_{i+1}^j = a_i^j \oplus m_i r_{i+j}$ for $0 \leq j \leq 31$ and $0 \leq i \leq L$. The final content of the accumulator, $a_{L+1}^0, \ldots, a_{L+1}^{31}$, is the authentication tag.

8.7 TRUE RANDOM NUMBER GENERATORS

Entropy Sources

A true random number generator (TRNG) uses a nondeterministic source to produce randomness. Most operate by measuring unpredictable natural processes, such as pulse detectors of ionizing radiation events, gas discharge tubes, and leaky capacitors. Intel has developed a commercially available chip that samples thermal noise by sampling the output of a coupled pair of inverters. LavaRnd is an open source project for creating truly random numbers using inexpensive cameras, open source code, and inexpensive hardware. The system uses a saturated CCD in a light-tight can as a chaotic source to produce the seed. Software processes the result into truly random numbers in a variety of formats.

RFC 4086 lists the following possible sources of randomness that, with care, easily can be used on a computer to generate true random sequences.

- **Sound/video input:** Many computers are built with inputs that digitize some real-world analog source, such as sound from a microphone or video input from a camera. The "input" from a sound digitizer with no source plugged in or from a camera with the lens cap on is essentially thermal noise. If the system has enough gain to detect anything, such input can provide reasonably high quality random bits.
- **Disk drives:** Disk drives have small random fluctuations in their rotational speed due to chaotic air turbulence [JAKO98]. The addition of low-level disk seek-time instrumentation produces a series of measurements that contain this randomness. Such data is usually highly correlated, so significant processing is needed. Nevertheless, experimentation a decade ago showed that, with such

Table 8.5 Comparison of PRNGs and TRNGs

	Pseudorandom Number Generators	True Random Number Generators
Efficiency	Very efficient	Generally inefficient
Determinism	Deterministic	Nondeterministic
Periodicity	Periodic	Aperiodic

processing, even slow disk drives on the slower computers of that day could easily produce 100 bits a minute or more of excellent random data.

There is also an online service (random.org), which can deliver random sequences securely over the Internet.

Comparison of PRNGs and TRNGs

Table 8.5 summarizes the principal differences between PRNGs and TRNGs. PRNGs are efficient, meaning they can produce many numbers in a short time, and deterministic, meaning that a given sequence of numbers can be reproduced at a later date if the starting point in the sequence is known. Efficiency is a nice characteristic if your application needs many numbers, and determinism is handy if you need to replay the same sequence of numbers again at a later stage. PRNGs are typically also periodic, which means that the sequence will eventually repeat itself. While periodicity is hardly ever a desirable characteristic, modern PRNGs have a period that is so long that it can be ignored for most practical purposes.

TRNGs are generally rather inefficient compared to PRNGs, taking considerably longer time to produce numbers. This presents a difficulty in many applications. For example, cryptography system in banking or national security might need to generate millions of random bits per second. TRNGs are also nondeterministic, meaning that a given sequence of numbers cannot be reproduced, although the same sequence may of course occur several times by chance. TRNGs have no period.

Conditioning[3]

A TRNG may produce an output that is biased in some way, such as having more ones than zeros or vice versa. More generally, NIST SP 800-90B defines a random process as **biased** with respect to an assumed discrete set of potential outcomes (i.e., possible output values) if some of those outcomes have a greater probability of occurring than do others. For example, a physical source such as electronic noise may contain a superposition of regular structures, such as waves or other periodic phenomena, which may appear to be random, yet are determined to be non-random using statistical tests.

In addition to bias, another concept used by SP 800-98B is that of **entropy rate**. SP 800-90B defines entropy rate as the rate at which a digitized noise source (or entropy source) provides entropy; it is computed as the assessed amount of entropy provided by a bit string output from the source, divided by the total number of bits in the bit string

[3] The reader unfamiliar with the concepts of entropy and min-entropy should read Appendix B before proceeding.

(yielding assessed bits of entropy per output bit). This will be a value between 0 (no entropy) and 1 (full entropy). Entropy rate is a measure of the randomness or unpredictability of a bit string. Another way of expressing it is that the entropy rate is k/n for a random source of length n bits and min-entropy k. Min-entropy is a measure of the number of random bits and is explained in Appendix B. In essence, a block of bits or a bit stream that is unbiased, and in which each bit and each group of bits is independent of all other bits and groups of bits will have an entropy rate of 1.

For hardware sources of random bits, the recommended approach is to assume that there may be bias and/or an entropy rate of less than 1 and to apply techniques to further "randomize" the bits. Various methods of modifying a bit stream for this purpose have been developed. These are referred to as **conditioning algorithms** or **deskewing algorithms**.

Typically, conditioning is done by using a cryptographic algorithm to "scramble" the random bits so as to eliminate bias and increase entropy. The two most common approaches are the use of a hash function or a symmetric block cipher.

Hash Function As we describe in Chapter 11, a hash function produces an n-bit output from an input of arbitrary length. A simple way to use a hash function for conditioning is as follows. Blocks of m input bits, with $m \geq n$, are passed through the hash function and the n output bits are used as random bits. To generate a stream of random bits, successive input blocks pass through the hash function to produce successive hashed output blocks.

Operating systems typically provide a built-in mechanism for generating random numbers. For example, Linux uses four entropy sources: mouse and keyboard activity, disk I/O operations, and specific interrupts. Bits are generated from these four sources and combined in a pooled buffer. When random bits are needed, the appropriate number of bits are read from the buffer and passed through the SHA-1 hash function [GUTT06].

A more complex approach is the hash derivation function specified in SP800-90A. Hash_df can be defined as follows:

Parameters:

input_string: The string to be hashed.

outlen: Output length.

no_of_bits_to_return: The number of bits to be returned by Hash_df. The maximum length (*max_number_of_bits*) is implementation dependent, but shall be less than or equal to (255 × *outlen*). *no_of_bits_to_return* is represented as a 32-bit integer.

requested_bits: The result of performing the Hash_df.

Hash_df Process:

1. *temp* = the Null string
2. $len = \left\lceil \frac{no\_of\_bits\_to\_return}{outlen} \right\rceil$

3. *counter* = 0x01 Comment: An 8-bit binary value representing the integer "1."
4. For *i* = 1 to *len* do Comment: In 4.1, *no_of_bits_to_return* is used as a 32-bit string.
   4.1. *temp* = *temp* || **Hash** (*counter* || *no_of_bits_to_return* || *input_string*).
   4.2. *counter* = *counter* + 1.
5. *requested_bits* = **leftmost** (*temp, no_of_bits_to_return*).
6. Return (**SUCCESS**, *requested_bits*).

This algorithm takes an input block of bits of arbitrary length and returns the requested number of bits, which may be up to 255 times as long as the hash output length.

The reader may be uneasy that the output consists of hashed blocks in which the input to the hash function for each block is the same input string and differs only by the value of the counter. However, cryptographically strong hash functions, such as the SHA family, provide excellent diffusion (as defined in Chapter 4) so that change in the counter value results in dramatically different outputs.

Block Cipher Instead of a hash function, a block cipher such as AES can be used to scramble the TRNG bits. Using AES, a simple approach would be to take 128-bit blocks of TRNG bits and encrypt each block with AES and some arbitrary key. SP 800-90B outlines an approach similar to the hash_df function described previously. The Intel implementation discussed subsequently provides an example of using AES for conditioning.

Health Testing

Figure 8.13 provides a general model for a nondeterministic random bit generator. A hardware noise source produces a true random output. This is digitized to produce true, or nondeterministic, source of bits. This bit source then passes through a conditioning module to mitigate bias and maximize entropy.

Figure 8.13 also shows a health-testing module, which is used on the outputs of both the digitizer and conditioner. In essence, health testing is used to validate that the noise source is working as expected and that the conditioning module is produced output with the desired characteristics. Both forms of health testing are recommended by SP 800-90B.

Health Tests on the Noise Source The nature of the health testing of the noise source depends strongly on the technology used to produce noise. In general, we can assume that the digitized output of the noise source will exhibit some bias. Thus, the traditional statistical tests, such as those defined in SP 800-22 and discussed in Section 8.1, are not useful for monitoring the noise source, because the noise source is likely to always fail. Rather, the tests on the noise source need to be tailored to the expected statistical behavior of the correctly operating noise source. The goal is not to determine if the source is unbiased, which it isn't, but if it is operating as expected.

SP 800-90B specifies that continuous tests be done on digitized samples obtained from the noise source (point A in Figure 8.13). The purpose is to test for variability. More specifically, the purpose is to determine if the noise source is

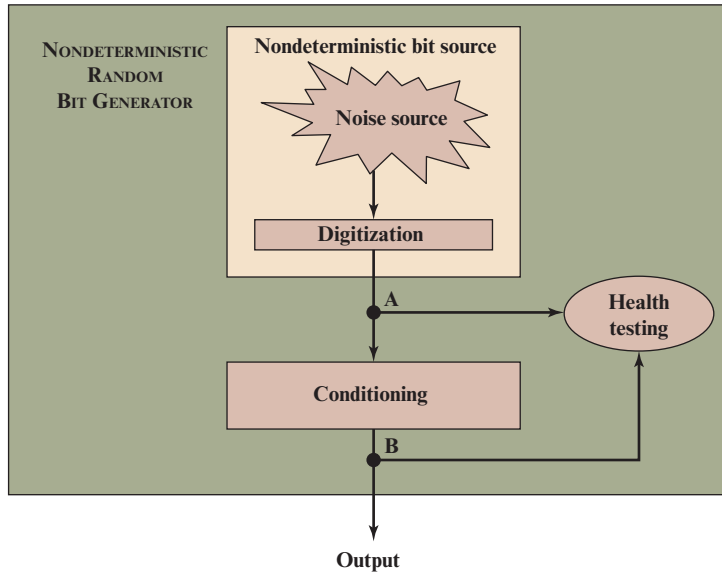


Figure 8.13 NRBG Model

producing at the expected entropy rate. SP 800-90B mandates the use of two tests: the Repetition Count Test and the Adaptive Proportion Test.

The **Repetition Count Test** is designed to quickly detect a catastrophic failure that causes the noise source to become "stuck" on a single output value for a long time. For this test, it is assumed that a given noise source is assessed to have a given min-entropy value of H. The entropy is expressed as the amount of entropy per sample, where a sample could be a single bit or some block of bits of length n. With an assessed value of H, it is straightforward to calculate the probability that a sequence of C consecutive samples will yield identical sample values. For example, a noise source with one bit of min-entropy per sample has no more than a 1/2 probability of repeating some sample value twice in a row, no more than 1/4 probability of repeating some sample value three times in a row, and in general, no more than $(1/2)^{C-1}$ probability of repeating some sample value C times in a row. To generalize, for a noise source with H bits of min-entropy per sample, we have:

$$\Pr[C \text{ identical samples in a row}] \leq (2^{-H})^{(C-1)}$$

The Repetition Count Test involves looking for consecutive identical samples. If the count reaches some cutoff value C, then an error condition is raised. To determine the value of C used in the test, the test must be configured with a parameter W, which is the acceptable false-positive probability associated with an alarm triggered by C repeated sample values. To avoid false positives, W should be set at some very small number greater than 0. Given W, we can now determine the value of C. Specifically, we want C to be the smallest number that satisfies the equation $W \leq (2^{-H})^{(C-1)}$. Reworking terms, this gives us a value of:

$$C = \left\lceil 1 + \frac{-\log(W)}{H} \right\rceil$$

For example, for $W = 2^{-30}$, an entropy source with $H = 7.3$ bits per sample would have a cutoff value C of $\left\lceil 1 + \frac{30}{7.3} \right\rceil = 6$.

The Repetition Count Test starts by recording a sample value and then counting the number of repetitions of the same value. If the counter reaches the cutoff value C, an error is reported. If a sample value is encountered that differs from the preceding sample, then the counter is reset to 1 and the algorithm starts over.

The **Adaptive Proportion Test** is designed to detect a large loss of entropy, such as might occur as a result of some physical failure or environmental change affecting the noise source. The test continuously measures the local frequency of occurrence of some sample value in a sequence of noise source samples to determine if the sample occurs too frequently.

The test starts by recording a sample value and then observes N successive sample values. If the initial sample value is observed at least C times, then an error condition is reported. SP 800-90B recommends that a probability of a false positive of $W = 2^{-30}$ be used for the test and provides guidance on the selection of values for N and C.

Health Tests on the Conditioning Function SP 800-90B specifies that health tests should also be applied to the output of the conditioning component (point B in Figure 8.13), but does not indicate which tests to use. The purpose of the health tests on the conditioning component is to assure that the output behaves as a true random bit stream. Thus, it is reasonable to use the tests for randomness defined in SP 800-22, and described in Section 8.1.

Intel Digital Random Number Generator

As was mentioned, TRNGs have traditionally been used only for key generation and other applications where only a small number of random bits were required. This is because TRNGs have generally been inefficient, with a low bit rate of random bit production.

The first commercially available TRNG that achieves bit production rates comparable with that of PRNGs is the Intel digital random number generator (DRNG) [TAYL11, MECH14], offered on new multicore chips since May 2012.[4]

Two notable aspects of the DRNG:

1. It is implemented entirely in hardware. This provides greater security than a facility that includes a software component. A hardware-only implementation should also be able to achieve greater computation speed than a software module.

[4]It is unfortunate that Intel chose the acronym DRNG for an NRBG. It confuses with DRBG, which is a pseudorandom number bit generator.

2. The entire DRNG is on the same multicore chip as the processors. This eliminates the I/O delays found in other hardware random number generators.

DRNG Hardware Architecture Figure 8.14 shows the overall structure of the DRNG. The first stage of the DRNG generates random numbers from thermal noise. The heart of the stage consists of two inverters (NOT gates), with the output of each inverter connected to the input of the other. Such an arrangement has two stable states, with one inverter having an output of logical 1 and the other having an output of logical 0. The circuit is then configured so that both inverters are forced to have the same indeterminate state (both inputs and both outputs at logical 1) by clock pulses. Random thermal noise within the inverters soon jostles the two inverters into a mutually stable state. Additional circuitry is intended to compensate for any biases or correlations. This stage is capable, with current hardware, of generating random bits at a rate of 4 Gbps.

The output of the first stage is generated 512 bits at a time. To assure that the bit stream does not have **skew** or bias, a conditioner randomizes its input using a cryptographic function. In this case, the function is referred to as CBC-MAC or

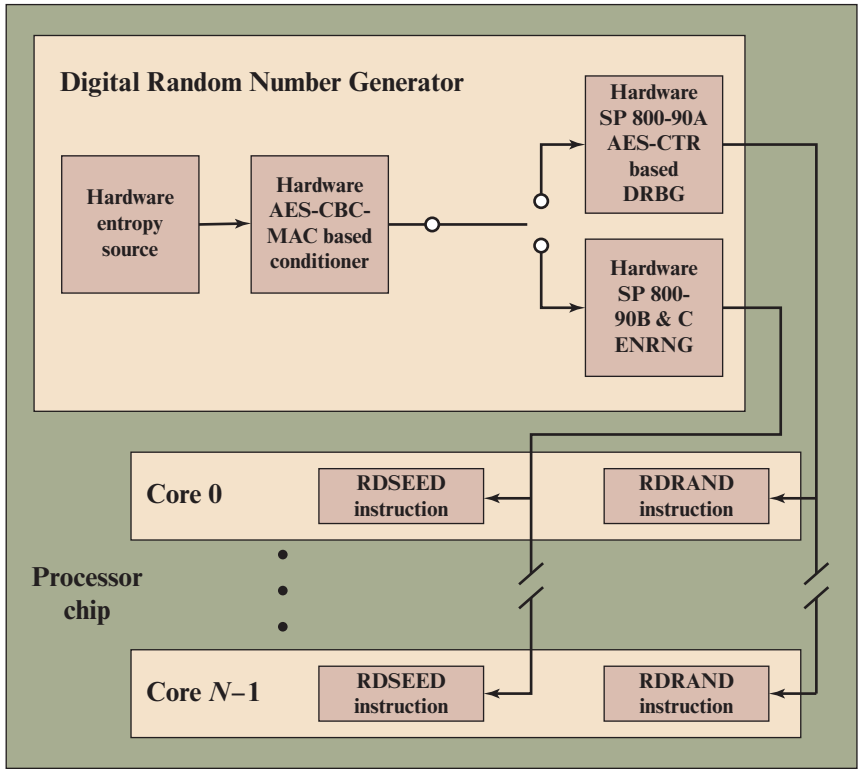


Figure 8.14 Intel Processor Chip with Random Number Generator

CMAC, as specified in NIST SP 800-38B. In essence, CMAC encrypts its input using the cipher block chaining (CBC) mode (Figure 8.4) and outputs the final block. We examine CMAC in detail in Chapter 12. The output of this stage is generated 256 bits at a time and is intended to exhibit true randomness with no skew or bias.

While the hardware's circuitry generates random numbers from thermal noise much more quickly than its predecessors, it is still not fast enough for some of today's computing requirements. To enable the DRNG to generate random numbers as quickly as a software DRBG, and also maintain the high quality of the random numbers, a third stage is added. This stage uses the 256-bit random numbers to seed a cryptographically secure DRBG that creates 128-bit numbers. From one 256-bit seed, the DRBG can output many pseudorandom numbers, exceeding the 3-Gbps rate of the entropy source. An upper bound of 511 128-bit samples can be generated per seed. The algorithm used for this stage is CTR_DRBG, described in Section 8.3.

The output of the PRNG stage is available to each of the cores on the chip via the RDRAND instruction. RDRAND retrieves a 16-, 32-, or 64-bit random value and makes it available in a software-accessible register.

Preliminary data from a pre-production sample on a system with a third generation Intel® Core™ family processor produced the following performance [INTE12]: up to 70 million RDRAND invocations per second, and a random data production rate of over 4 Gbps.

The output of the conditioner is also made available to another module, known as an enhanced nondeterministic random number generator (ENRNG) that provides random numbers that can be used as seeds for various cryptographic algorithms. The ENRNG is compliant with specifications in SP 800-90B and 900-90C. The output of the ENRNG stage is available to each of the cores on the chip via the RDSEED instruction. RDSEED retrieves a hardware-generated random seed value from the ENRNG and stores it in the destination register given as an argument to the instruction.

DRNG Logical Structure Figure 8.15 provides a simplified view of the logical flow of the Intel DRBG. As was described, the heart of the hardware entropy source is a pair of inverters that feed each other. Two transistors, driven by the same clock, force the inputs and outputs of both inverters to the logical 1 state. Because this is an unstable state, thermal noise will cause the configuration to settle randomly into a stable state with either Node A at logical 1 and Node B at logical 0, or the reverse. Thus the module generates random bits at the clock rate.

The output of the entropy source is collected 512 bits at a time and used to feed to two CBC hardware implementations using AES encryption. Each implementation takes two blocks of 128 bits of "plaintext" and encrypts using the CBC mode. The output of the second encryption is retained. For both CBC modules, an all-zeros key is used initially. Subsequently, the output of the PRNG stage is fed back to become the key for the conditioner stage.

The output of the conditioner stage consists of 256 bits. This block is provided as input to the update function of the DRGB stage. The update function is initialized

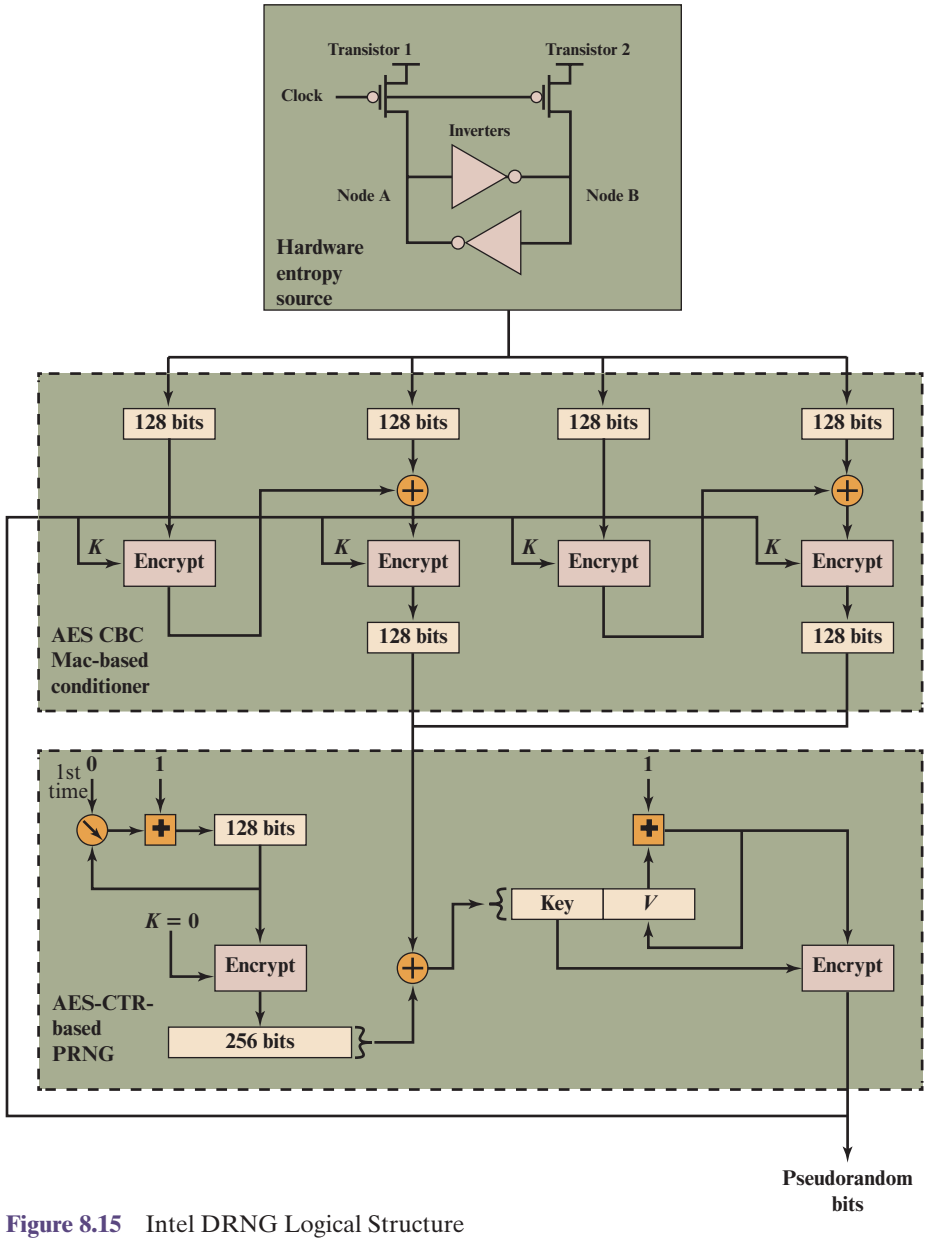


Figure 8.15 Intel DRNG Logical Structure

with the all-zeros key and the counter value 0. The function is iterated twice to produce a 256-block, which is then XORed with the input from the conditioner stage. The results are used as the 128-bit key and the 128-bit seed for the generate function. The generate function produces pseudorandom bits in 128-bit blocks.

8.8 KEY TERMS, REVIEW QUESTIONS, AND PROBLEMS

Key Terms

backward unpredictability	pseudorandom function (PRF)	stream cipher
deskewing algorithm	pseudorandom number generator (PRNG)	true random number generator (TRNG)
entropy source	seed	unpredictability
forward unpredictability	skew	
keystream		

Review Questions

8.1 List two criteria to validate the randomness of a sequence of numbers.

8.2 What is ANSI X9.17 PRNG?

8.3 What is the recommended key length for a stream cipher to guard against brute force attacks?

8.4 What is the difference between a one-time pad and a stream cipher?

8.5 The 802.11 standard protocol used RC4 for its agility and simplicity for encryption and decryption. There are a few simple steps in RC4 including the initialization of S to a number from 0 to 255, followed by permutation. Explain the stream generation, and main benefits and drawbacks of RC4.

8.6 List a few applications of stream ciphers and block ciphers.

Problems

8.1 If we take the linear congruential algorithm with an additive component of 0,

$$X_{n+1} = (aX_n) \bmod m$$

Then it can be shown that if m is prime and if a given value of a produces the maximum period of $m - 1$, then a^k will also produce the maximum period, provided that k is less than m and that k and $m - 1$ are relatively prime. Demonstrate this by using $X_0 = 1$ and $m = 31$ and producing the sequences for $a^k = 3, 3^2, 3^3$, and 3^4.

8.2 **a.** What is the maximum period obtainable from the following generator?

$$X_{n+1} = (aX_n) \bmod 2^4$$

b. What should be the value of a?

c. What restrictions are required on the seed?

8.3 You may wonder why the modulus $m = 2^{31} - 1$ was chosen for the linear congruential method instead of simply 2^{31}, because this latter number can be represented with no additional bits and the mod operation should be easier to perform. In general, the modulus $2^k - 1$ is preferable to 2^k. Why is this so?

8.4 With the linear congruential algorithm, a choice of parameters that provides a full period does not necessarily provide a good randomization. For example, consider the following two generators:

$$X_{n+1} = (11X_n) \bmod 13$$

$$X_{n+1} = (2X_n) \bmod 13$$

Write out the two sequences to show that both are full periods. Which one appears more random to you?

8.5 In any use of pseudorandom numbers, whether for encryption, simulation, or statistical design, it is dangerous to trust blindly the random number generator that happens to be available in your computer's system library. [PARK88] found that many contemporary textbooks and programming packages make use of flawed algorithms for pseudorandom number generation. This exercise will enable you to test your system.

The test is based on a theorem attributed to Ernesto Cesaro (see [KNUT98] for a proof), which states the following: Given two randomly chosen integers, x and y, the probability that $\gcd(x, y) = 1$ is $6/\pi^2$. Use this theorem in a program to determine statistically the value of π. The main program should call three subprograms: the random number generator from the system library to generate the random integers; a subprogram to calculate the greatest common divisor of two integers using Euclid's Algorithm; and a subprogram that calculates square roots. If these latter two programs are not available, you will have to write them as well. The main program should loop through a large number of random numbers to give an estimate of the aforementioned probability. From this, it is a simple matter to solve for your estimate of π.

If the result is close to 3.14, congratulations! If not, then the result is probably low, usually a value of around 2.7. Why would such an inferior result be obtained?

8.6 What RC4 key value will leave S unchanged during initialization? That is, after the initial permutation of S, the entries of S will be equal to the values from 0 through 255 in ascending order.

8.7 RC4 has a secret internal state which is a permutation of all the possible values of the vector S and the two indices i and j.

a. Using a straightforward scheme to store the internal state, how many bits are used?

b. Suppose we think of it from the point of view of how much information is represented by the state. In that case, we need to determine how may different states there are, then take the log to base 2 to find out how many bits of information this represents. Using this approach, how many bits would be needed to represent the state?

8.8 Alice and Bob agree to communicate privately via email using a scheme based on RC4, but they want to avoid using a new secret key for each transmission. Alice and Bob privately agree on a 128-bit key k. To encrypt a message m, consisting of a string of bits, the following procedure is used.

1. Choose a random 64-bit value v
2. Generate the ciphertext $c = \text{RC4}(v \| k) \oplus m$
3. Send the bit string $(v \| c)$

a. Suppose Alice uses this procedure to send a message m to Bob. Describe how Bob can recover the message m from $(v \| c)$ using k.

b. If an adversary observes several values $(v_1 \| c_1), (v_2 \| c_2), \ldots$ transmitted between Alice and Bob, how can he/she determine when the same key stream has been used to encrypt two messages?

c. Approximately how many messages can Alice expect to send before the same key stream will be used twice? Use the result from the birthday paradox described in Appendix E.

d. What does this imply about the lifetime of the key k (i.e., the number of messages that can be encrypted using k)?

8.9 Show that the polynomial $P(X) = 1 + X + X^4$ is a primitive generator polynomial for the circuit of Figure 8.9a by calculating $1/P(X)$ and showing that the coefficients of the resulting polynomial repeat the output pattern in Figure 8.9b.

8.10 This problem demonstrates that different LFSRs can be used to generate an m-sequence.

a. Assume an initial state of 10000 in the LFSR of Figure 8.16a. In a manner similar to Figure 8.9b, show the generation of an m-sequence.

b. Now assume the configuration of Figure 8.16b, with the same initial state, and repeat part (a). Show that this configuration also produces an m-sequence, but that it is a different sequence from that produced by the first LFSR.

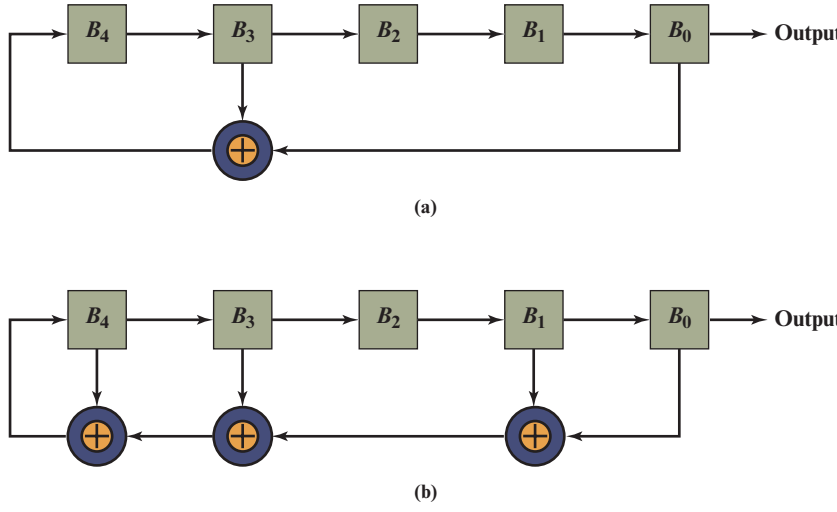


Figure 8.16 Two Different Configurations of LFSRs of Length 5

8.11 Suppose you have a true random bit generator where each bit in the generated stream has the same probability of being a 0 or 1 as any other bit in the stream and that the bits are not correlated; that is the bits are generated from identical independent distribution. However, the bit stream is biased. The probability of a 1 is $0.5 + \partial$ and the probability of a 0 is $0.5 - \partial$, where $0 < \partial < 0.5$. A simple conditioning algorithm is as follows: Examine the bit stream as a sequence of nonoverlapping pairs. Discard all 00 and 11 pairs. Replace each 01 pair with 0 and each 10 pair with 1.

a. What is the probability of occurrence of each pair in the original sequence?
b. What is the probability of occurrence of 0 and 1 in the modified sequence?
c. What is the expected number of input bits to produce x output bits?
d. Suppose that the algorithm uses overlapping successive bit pairs instead of nonoverlapping successive bit pairs. That is, the first output bit is based on input bits 1 and 2, the second output bit is based on input bits 2 and 3, and so on. What can you say about the output bit stream?

8.12 Another approach to conditioning is to consider the bit stream as a sequence of nonoverlapping groups of n bits each and output the parity of each group. That is, if a group contains an odd number of ones, the output is 1; otherwise the output is 0.

a. Express this operation in terms of a basic Boolean function.
b. Assume, as in the preceding problem, that the probability of a 1 is $0.5 + \partial$. If each group consists of 2 bits, what is the probability of an output of 1?
c. If each group consists of 4 bits, what is the probability of an output of 1?
d. Generalize the result to find the probability of an output of 1 for input groups of n bits.

8.13 It is important to note that the Repetition Count Test described in Section 8.6 is not a very powerful health test. It is able to detect only catastrophic failures of an entropy source. For example, a noise source evaluated at 8 bits of min-entropy per sample has a cutoff value of 5 repetitions to ensure a false-positive rate of approximately once per four billion samples generated. If that noise source somehow failed to the point that it was providing only 6 bits of min-entropy per sample, how many samples would be expected to be needed before the Repetition Count Test would notice the problem?