CHAPTER 7

PSEUDORANDOM NUMBER GENERATION AND STREAM CIPHERS
7.1 Principles of Pseudorandom Number Generation

The Use of Random Numbers
TRNGs, PRNGs, and PRFs
PRNG Requirements
Algorithm Design

7.2 Pseudorandom Number Generators

Linear Congruential Generators
Blum Blum Shub Generator

7.3 Pseudorandom Number Generation Using a Block Cipher

PRNG Using Block Cipher Modes of Operation
ANSI X9.17 PRNG
7.4 Stream Ciphers

7.5 RC4
   - Initialization of S
   - Stream Generation
   - Strength of RC4

7.6 True Random Number Generators
   - Entropy Sources
   - Skew
KEY POINTS

- A capability with application to a number of cryptographic functions is random or pseudorandom number generation. The principle requirement for this capability is that the generated number stream be unpredictable.
- A stream cipher is a symmetric encryption algorithm in which ciphertext output is produced bit-by-bit or byte-by-byte from a stream of plaintext input. The most widely used such cipher is RC4.
7.1 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 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.
• **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 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**.
Source of true randomness

Conversion to binary

Random bit stream

Deterministic algorithm

Pseudorandom bit stream

PRNG = pseudorandom number generator

Context-specific values

Deterministic algorithm

Pseudorandom value

PRF = pseudorandom function

Figure 7.1 Random and Pseudorandom Number Generators
Figure 7.1 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 7.4. Also, see Figure 3.1a.

- **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 16 and 17.
PRNG Requirements

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.
Figure 7.2 Generation of Seed Input to PRNG
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. The most important example of the latter is **RC4**, described in Section 7.5.
• **Algorithms based on existing cryptographic algorithms:** Cryptographic algorithms have the effect of randomizing input. 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. Three broad categories of cryptographic algorithms are commonly used to create PRNGs:
  
  — **Symmetric block ciphers:** This approach is discussed in Section 7.3.
  
  — **Asymmetric ciphers:** The number theoretic concepts used for an asymmetric cipher can also be adapted for a PRNG; this approach is examined in Chapter 10.
  
  — **Hash functions and message authentication codes:** This approach is examined in Chapter 12.
### 7.2 PSEUDORANDOM NUMBER GENERATORS

#### 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:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$m$</td>
<td>the modulus</td>
<td>$m &gt; 0$</td>
</tr>
<tr>
<td>$a$</td>
<td>the multiplier</td>
<td>$0 &lt; a &lt; m$</td>
</tr>
<tr>
<td>$c$</td>
<td>the increment</td>
<td>$0 \leq c &lt; m$</td>
</tr>
<tr>
<td>$X_0$</td>
<td>the starting value, or seed</td>
<td>$0 \leq X_0 &lt; m$</td>
</tr>
</tbody>
</table>

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

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

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

These equations can be solved for $a$, $c$, and $m$. 
Blum Blum Shub Generator

A popular approach to generating secure pseudorandom numbers is known as the **Blum, Blum, Shub (BBS)** generator, 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,

$$p \equiv q \equiv 3 \pmod{4}$$
This notation, explained more fully in Chapter 4, simply means that 
\((p \text{ mod } 4) = (q \text{ mod } 4) = 3\). 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:

\[
\begin{align*}
X_0 &= s^2 \text{ mod } n \\
\text{for } i &= 1 \text{ to } \infty \\
X_i &= (X_{i-1})^2 \text{ mod } n \\
B_i &= X_i \text{ mod } 2
\end{align*}
\]

Thus, the least significant bit is taken at each iteration. Table 7.1, shows an example 
of BBS operation. Here, \(n = 192649 = 383 \times 503\), and the seed \(s = 101355\).
## Table 7.1  Example Operation of BBS Generator

<table>
<thead>
<tr>
<th>$i$</th>
<th>$X_i$</th>
<th>$B_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>20749</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>143135</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>177671</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>97048</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>89992</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>174051</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>80649</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>45663</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>69442</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>186894</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>177046</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$i$</th>
<th>$X_i$</th>
<th>$B_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>137922</td>
<td>0</td>
</tr>
<tr>
<td>12</td>
<td>123175</td>
<td>1</td>
</tr>
<tr>
<td>13</td>
<td>8630</td>
<td>0</td>
</tr>
<tr>
<td>14</td>
<td>114386</td>
<td>0</td>
</tr>
<tr>
<td>15</td>
<td>14863</td>
<td>1</td>
</tr>
<tr>
<td>16</td>
<td>133015</td>
<td>1</td>
</tr>
<tr>
<td>17</td>
<td>106065</td>
<td>1</td>
</tr>
<tr>
<td>18</td>
<td>45870</td>
<td>0</td>
</tr>
<tr>
<td>19</td>
<td>137171</td>
<td>1</td>
</tr>
<tr>
<td>20</td>
<td>48060</td>
<td>0</td>
</tr>
</tbody>
</table>
7.3 PSEUDORANDOM NUMBER GENERATION USING A BLOCK CIPHER

PRNG Using Block Cipher Modes of Operation

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

Figure 7.3 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 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).
Figure 7.3  PRNG Mechanisms Based on Block Ciphers

(a) CTR mode
(b) OFB mode
The CTR algorithm for PRNG can be summarized as follows.

```plaintext
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.

```plaintext
while (len (temp) < requested_number_of_bits) do
  V = E(Key, V)
  temp = temp || V
```
### Table 7.2 Example Results for PRNG Using OFB

<table>
<thead>
<tr>
<th>Output Block</th>
<th>Fraction of One Bits</th>
<th>Fraction of Bits that Match with Preceding Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>1786f4c7ff6e291dbfdd90ec3453176</td>
<td>0.57</td>
<td>—</td>
</tr>
<tr>
<td>5e17b22b14677a4d66890f87565eae64</td>
<td>0.51</td>
<td>0.52</td>
</tr>
<tr>
<td>fd18284ac82251dfb3aa62c326cd46cc</td>
<td>0.47</td>
<td>0.54</td>
</tr>
<tr>
<td>c8e545198a758ef5dd86b41946389bd5</td>
<td>0.50</td>
<td>0.44</td>
</tr>
<tr>
<td>fe7bae0e23019542962e2c52d215a2e3</td>
<td>0.47</td>
<td>0.48</td>
</tr>
<tr>
<td>14fdf5ec99469598ae0379472803accd</td>
<td>0.49</td>
<td>0.52</td>
</tr>
<tr>
<td>6aecca972e5a3ef17bd1a1b775fc8b929</td>
<td>0.57</td>
<td>0.48</td>
</tr>
<tr>
<td>f7e97badf359d128f00d9b4ae323db64</td>
<td>0.55</td>
<td>0.45</td>
</tr>
<tr>
<td>Output Block</td>
<td>Fraction of One Bits</td>
<td>Fraction of Bits that Match with Preceding Block</td>
</tr>
<tr>
<td>----------------------------------</td>
<td>----------------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>1786f4c7ff6e291dbdfdd90ec3453176</td>
<td>0.57</td>
<td>—</td>
</tr>
<tr>
<td>60809669a3e092a01b463472fdcae420</td>
<td>0.41</td>
<td>0.41</td>
</tr>
<tr>
<td>d4e6e170b46b0573eedf88ee39bff33d</td>
<td>0.59</td>
<td>0.45</td>
</tr>
<tr>
<td>5f8fcfc5deca18ea246785d7fadc76f8</td>
<td>0.59</td>
<td>0.52</td>
</tr>
<tr>
<td>90e63ed27bb07868c753545bdd57ee28</td>
<td>0.53</td>
<td>0.52</td>
</tr>
<tr>
<td>0125856fdf4a17f747c7833695c52235</td>
<td>0.50</td>
<td>0.47</td>
</tr>
<tr>
<td>f4be2d179b0f2548fd748c8fc7c81990</td>
<td>0.51</td>
<td>0.48</td>
</tr>
<tr>
<td>1151fc48f90eebac658a3911515c3c66</td>
<td>0.47</td>
<td>0.45</td>
</tr>
</tbody>
</table>
ANSI X9.17 PRNG

One of the strongest (cryptographically speaking) PRNGs is specified in ANSI X9.17. A number of applications employ this technique, including financial security applications and PGP (the latter described in Chapter 18).

Figure 7.4  ANSI X9.17 Pseudorandom Number Generator
\( DT_i \) Date/time value at the beginning of \( i \)th generation stage
\( V_i \) Seed value at the beginning of \( i \)th generation stage
\( R_i \) Pseudorandom number produced by the \( i \)th generation stage
\( K_1, K_2 \) DES keys used for each stage

Then

\[
R_i = \text{EDE}([K_1, K_2], [V_i \oplus \text{EDE}([K_1, K_2], DT_i)])
\]
\[
V_{i+1} = \text{EDE}([K_1, K_2], [R_i \oplus \text{EDE}([K_1, K_2], DT_i)])
\]

where \( \text{EDE}([K_1, K_2], X) \) refers to the sequence encrypt-decrypt-encrypt using two-key triple DES to encrypt \( X \).
A typical stream cipher encrypts plaintext one byte at a time, although a stream cipher may be designed to operate on one bit at a time or on units larger than a byte at a time. Figure 7.5 is a representative diagram of stream cipher structure. In this structure, a key is input to a pseudorandom bit generator that produces a stream of 8-bit numbers that are apparently random. The output of the generator, called a 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

\[
\begin{align*}
11001100 & \text{ plaintext} \\
\oplus & 01101100 \text{ key stream} \\
10100000 & \text{ ciphertext}
\end{align*}
\]
Decryption requires the use of the same pseudorandom sequence:

\[
\begin{align*}
10100000 & \quad \text{ciphertext} \\
\oplus 01101100 & \quad \text{key stream} \\
11001100 & \quad \text{plaintext}
\end{align*}
\]

The stream cipher is similar to the one-time pad discussed in Chapter 2. The difference is that a one-time pad uses a genuine random number stream, whereas a stream cipher uses a pseudorandom number stream.
Figure 7.5  Stream Cipher Diagram
Table 7.4  Speed Comparisons of Symmetric Ciphers on a Pentium II

<table>
<thead>
<tr>
<th>Cipher</th>
<th>Key Length</th>
<th>Speed (Mbps)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DES</td>
<td>56</td>
<td>9</td>
</tr>
<tr>
<td>3DES</td>
<td>168</td>
<td>3</td>
</tr>
<tr>
<td>RC2</td>
<td>Variable</td>
<td>0.9</td>
</tr>
<tr>
<td>RC4</td>
<td>Variable</td>
<td>45</td>
</tr>
</tbody>
</table>
7.5 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 Secure Sockets Layer/Transport Layer Security (SSL/TLS) standards that have been defined for communication between Web browsers and servers. It is also used in the Wired Equivalent Privacy (WEP) protocol and the newer WiFi Protected Access (WPA) protocol that are part of the IEEE 802.11 wireless LAN standard. 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.
7.5 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 Secure Sockets Layer/Transport Layer Security (SSL/TLS) standards that have been defined for communication between Web browsers and servers. It is also used in the Wired Equivalent Privacy (WEP) protocol and the newer WiFi Protected Access (WPA) protocol that are part of the IEEE 802.11 wireless LAN standard. 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.
Initialization of S

To begin, the entries of S are set equal to the values from 0 through 255 in ascending order; that is, \( S[0] = 0, S[1] = 1, \ldots, S[255] = 255 \). A temporary vector, \( T \), is also created. If the length of the key \( K \) is 256 bytes, then \( T \) 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

```c
/* 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]:

```c
/* 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]:

```c
/* 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 7.6 illustrates the RC4 logic.
Use $T[j]$ to initiate $S[j]$

Use $S[j]$ only! ($S[j]$ will change.)

Figure 7.6 RC4
Strength of RC4

A number of papers have been published analyzing methods of attacking RC4 (e.g., [KNUD98], [MIST98], [FLUH00], [MANT01]). None of these approaches is practical against RC4 with a reasonable key length, such as 128 bits. A more serious problem is reported in [FLUH01]. The authors demonstrate that the WEP protocol, intended to provide confidentiality on 802.11 wireless LAN networks, is vulnerable to a particular attack approach. In essence, the problem is not with RC4 itself but the way in which keys are generated for use as input to RC4. This particular problem does not appear to be relevant to other applications using RC4 and can be remedied in WEP by changing the way in which keys are generated. This problem points out the difficulty in designing a secure system that involves both cryptographic functions and protocols that make use of them.