12

Pseudo-random Number Generation

12.1 Introduction and Examples

There are many situations in cryptography where it is important to be able to generate random numbers, bit-strings, etc. For example, cryptographic keys are to be generated at random from a specified keyspace, and many protocols require random numbers to be generated during their execution. Generating random numbers by means of coin tosses or other physical processes is time-consuming and expensive, so in practice it is common to use a *pseudo-random bit generator* (or PRBG). A PRBG starts with a short random bit-string (a "seed") and expands it into a much longer "random-looking" bit-string. Thus a PRBG reduces the amount of random bits that are required in an application.

More formally, we have the following definition.

DEFINITION 12.1 *Let k, ℓ be positive integers such that $\ell \geq k + 1$ (where ℓ is a specified polynomial function of k). A (k, ℓ)-**pseudo-random bit generator** (more briefly, a (k, ℓ)-PRBG) is a function $f : (\mathbb{Z}_2)^k \rightarrow (\mathbb{Z}_2)^\ell$ that can be computed in polynomial time (as a function of k). The input $s_0 \in (\mathbb{Z}_2)^k$ is called the **seed**, and the output $f(s_0) \in (\mathbb{Z}_2)^\ell$ is called a **pseudo-random** bit-string.*

The function f is deterministic, so the bit-string $f(s_0)$ is dependent only on the seed. Our goal is that the pseudo-random bit-string $f(s_0)$ should "look like" truly random bits, given that the seed is chosen at random. Giving a precise definition is quite difficult, but we will try to give an intuitive description of the concept later in this chapter.

One motivating example for studying this type PRBG is as follows. Recall the concept of perfect secrecy that we studied in Chapter 2. One realization of perfect secrecy is the **One-time Pad**, where the plaintext and the key are both bit-strings of a specified length, and the ciphertext is constructed by taking the bitwise exclusive-or of the plaintext and the key. The practical difficulty of the **One-time Pad** is that the key, which must be randomly generated and communicated over a

FIGURE 12.1
Linear Congruential Generator

Let $M \geq 2$ be an integer, and let $1 \leq a, b \leq M-1$. Define $k = \lceil \log_2 M \rceil$ and let $k+1 \leq \ell \leq M-1$.

For a seed s_0, where $0 \leq s_0 \leq M-1$, define

$$s_i = (as_{i-1} + b) \bmod M$$

for $1 \leq i \leq \ell$, and then define

$$f(s_0) = (z_1, z_2, \ldots, z_\ell),$$

where

$$z_i = s_i \bmod 2,$$

$1 \leq i \leq \ell$. Then f is a (k, ℓ)-**Linear Congruential Generator**.

secure channel, must be as long as the plaintext in order to ensure perfect secrecy. PRBGs provide a possible way of alleviating this problem. Suppose Alice and Bob agree on a PRBG and communicate a seed over the secure channel. Alice and Bob can then both compute the same string of pseudo-random bits, which will be used as a **One-time Pad**. Thus the seed functions as a key, and the PBRG can be thought of as a keystream generator for a stream cipher.

We now present some well-known PRBGs to motivate and illustrate some of the concepts we will be studying. First, we observe that a linear feedback shift register, as described in Section 1.1.7, can be thought of as a PRBG. Given a k-bit seed, an LFSR of degree k can be used to produce as many as $2^k - k - 1$ further bits before repeating. The PRBG obtained from an LFSR is very insecure: we already observed in Section 1.2.5 that knowledge of any $2k$ consecutive bits suffice to allow the seed to be determined, and hence the entire sequence can be reconstructed by an opponent. (Although we have not yet defined security of a PRBG, it should be clear that the existence of an attack of this type means that the generator is insecure!)

Another well-known (but insecure) PRBG, called the **Linear Congruential Generator**, is presented in Figure 12.1. Here is a very small example to illustrate.

Example 12.1
We can obtain a (5, 10)-PRBG by taking $M = 31$, $a = 3$ and $b = 5$ in the **Linear Congruential Generator**. If we consider the mapping $s \mapsto 3s + 5 \bmod 31$, then

TABLE 12.1
Bit-strings produced by the linear congruential generator

seed	sequence
0	1010001101
1	0100110101
2	1101010001
3	0001101001
4	1100101101
5	0100011010
6	1000110010
7	0101000110
8	1001101010
9	1010011010
10	0110010110
11	1010100011
12	0011001011
13	1111111111
14	0011010011
15	1010100011
16	0110100110
17	1001011010
18	0101101010
19	0101000110
20	1000110100
21	0100011001
22	1101001101
23	0001100101
24	1101010001
25	0010110101
26	1010001100
27	0110101000
28	1011010100
29	0011010100
30	0110101000

$13 \mapsto 13$, and the other 30 residues are permuted in a cycle of length 30, namely 0, 5, 20, 3, 14, 16, 22, 9, 1, 8, 29, 30, 2, 11, 7, 26, 21, 6, 23, 12, 10, 4, 17, 25, 18, 28, 27, 24, 15, 19. If the seed is anything other than 13, then the seed specifies a starting point in this cycle, and the next 10 elements, reduced modulo 2, form the pseudo-random sequence.

The 31 possible pseudo-random bit-strings produced by this generator are illustrated in Table 12.1. □

We can use some concepts developed in earlier chapters to consrtruct PRBGs.

FIGURE 12.2
RSA Generator

Let p, q to be two $(k/2)$-bit primes, and define $n = pq$. Let b be chosen such that $\gcd(b, \phi(n)) = 1$. As always, n and b are public while p and q are secret.

A seed s_0 is any element of $\mathbb{Z}_n^*$, so s_0 has k bits. For $i \geq 1$, define

$$s_{i+1} = {s_i}^b \bmod n,$$

and then define

$$f(s_0) = (z_1, z_2, \ldots, z_\ell),$$

where

$$z_i = s_i \bmod 2,$$

$1 \leq i \leq \ell$. Then f is a (k, ℓ)-**RSA Generator**.

For example, the output feedback mode of DES, as described in Section 3.4.1, can be thought of as a PRBG; moreover, it appears to be computationally secure.

Another approach in constructing very fast PRBGs is to combine LFSRs in some way that the output looks less linear. One such method, due to Coppersmith, Krawczyk and Mansour, is called the **Shrinking Generator**. Suppose we have two LFSRs, one of degree k_1 and one of k_2. We will require a total of $k_1 + k_2$ bits as our seed, in order to initialize both LFSRs. The first LFSR will produce a sequence of bits, say $a_1, a_2, \ldots$, and the second produces a sequence of bits $b_1, b_2, \ldots$. Then we define a sequence of pseudo-random bits $z_1, z_2, \ldots$ by the rule

$$z_i = a_{i_k},$$

where i_k is the position of the kth 1 in the sequence $b_1, b_2, \ldots$. These pseudo-random bits comprise a subsequence of the bits produced by the first LFSR. This method of pseudo-random bit generation is very fast and is resistent to various known attacks, but there does not seem to be any way to prove that it is secure.

In the rest of this chapter, we will investigate PRBGs that can be proved to be secure given some plausible computational assumption. There are PRBGs based on the fundamental problems of factoring (as it relates to the **RSA** public-key cryptosystem) and the **Discrete Logarithm** problem. A PRBG based on the **RSA** encryption function is shown in Figure 12.2, and a PRBG based on the **Discrete Logarithm** problem is discussed in the exercises.

We now give an example of the **RSA Generator**.

TABLE 12.2
Bits produced by RSA generator

i	s_i	z_i
0	75634	
1	31483	1
2	31238	0
3	51968	0
4	39796	0
5	28716	0
6	14089	1
7	5923	1
8	44891	1
9	62284	0
10	11889	1
11	43467	1
12	71215	1
13	10401	1
14	77444	0
15	56794	0
16	78147	1
17	72137	1
18	89592	0
19	29022	0
20	13356	0

Example 12.2
Suppose $n = 91261 = 263 \times 347$, $b = 1547$, and $s_0 = 75364$. The first 20 bits produced by the **RSA Generator** are computed as shown in Table 12.2. Hence the bit-string resulting from this seed is

$$10000111011110011000.$$

□

12.2 Indistinguishable Probability Distributions

There are two main objectives of a pseudo-random number generator: it should be fast (i.e., computable in polynomial time as a function of k) and it should be secure. Of course, these two requirements are often conflicting. The PRBGs based on linear congruences or linear feedback shift registers are indeed very fast. These PRBGs are quite useful in simulations, but they are very insecure for cryptographic applications.

Let us now try to make precise the idea of a PRBG being "secure." Intuitively, a string of k^m bits produced by a PRBG should look "random." That is, it should be impossible in an amount of time that is polynomial in k (equivalently, polynomial in ℓ) to distinguish a string of ℓ pseudo-random bits produced by a PRBG from a string of ℓ truly random bits.

This motivates the idea of distinguishability of probability distributions. Here is a definition of this concept.

DEFINITION 12.2 *Suppose p_0 and p_1 are two probability distributions on the set $(\mathbb{Z}_2)^\ell$ of bit-strings of length ℓ. Let $\mathbf{A} : (\mathbb{Z}_2)^\ell \to \{0,1\}$ be a probabilistic algorithm that runs in polynomial time (as a function of ℓ). Let $\epsilon > 0$. For $j = 0, 1$, define*

$$E_{\mathbf{A}}(p_j) = \sum_{(z_1,\ldots,z_\ell)\in(\mathbb{Z}_2)^\ell} p_j(z_1,\ldots,z_\ell) \times p(\mathbf{A}(z_1,\ldots,z_\ell) = 1 | (z_1,\ldots,z_\ell)).$$

*We say that $\mathbf{A}$ is an **ϵ-distinguisher** of p_0 and p_1 provided that*

$$|E_{\mathbf{A}}(p_0) - E_{\mathbf{A}}(p_1)| \geq \epsilon,$$

*and we say that p_0 and p_1 are **ϵ-distinguishable** if there exists an ϵ-distinguisher of p_0 and p_1.*

REMARK If $\mathbf{A}$ is a deterministic algorithm, then the conditional probabilities

$$p(\mathbf{A}(z_1,\ldots,z_\ell) = 1 | (z_1,\ldots,z_\ell))$$

always have the value 0 or 1. ∎

The intuition behind this definition is as follows. The algorithm $\mathbf{A}$ tries to decide if a bit-string $(z_1,\ldots,z_\ell)$ of length ℓ is more likely to have arisen from probability distribution p_1 or from probability distribution p_0. This algorithm may use random numbers if desired, i.e., it can be probabilistic. The output $\mathbf{A}(z_1,\ldots,z_\ell)$ represents the algorithm's guess as to which of these two probability distributions is more likely to have produced $(z_1,\ldots,z_\ell)$. The quantity $E_{\mathbf{A}}(p_j)$ represents the average (i.e., expected) value of the output of $\mathbf{A}$ over the probability distribution p_j, for $j = 0, 1$. This is computed by summing over all possible sequences $(z_1,\ldots,z_\ell)$ the product of the probability of the ℓ-tuple $(z_1,\ldots,z_\ell)$ and the probability that $\mathbf{A}$ answers "1" when given $(z_1,\ldots,z_\ell)$ as input. $\mathbf{A}$ is an ϵ-distinguisher provided that the values of these two expectations are at least ϵ apart.

The relevance to PRBGs is as follows. Consider the sequence of ℓ bits produced by the PRBG. There are 2^ℓ possible sequences, and if the bits were chosen independently at random, each of these 2^ℓ sequences would occur with equal probability $1/2^\ell$. Thus a truly random sequence corresponds to an equiprobable

distribution on the set of all bit-strings of length ℓ. Suppose we denote this probability distribution by p_0.

Now, consider sequences produced by the PRBG. Suppose a k-bit seed is chosen at random, and then the PRBG is used to obtain a bit-string of length ℓ. Then we obtain a probability distribution on the set of all bit-strings of length ℓ, which we denote by p_1. (For the purposes of illustration, suppose we make the simplifying assumption that no two seeds give rise to the same sequence of bits. Then, of the 2^ℓ possible sequences, 2^k sequences each occur with probability $1/2^k$, and the remaining $2^\ell - 2^k$ sequences never occur. So, in this case, the probability distribution p_1 is very non-uniform.)

Even though the two probability distributions p_0 and p_1 may be quite different, it is still conceivable that they might be ϵ-distinguishable only for small values of ϵ. This is our objective in constructing PRBGs.

Example 12.3
Suppose that a PRBG only produces sequences in which exactly $\ell/2$ bits have the value 0 and $\ell/2$ bits have the value 1. Define the function **A** by

$$\mathbf{A}(z_1, \ldots, z_\ell) = \begin{cases} 1 & \text{if } (z_1, \ldots, z_\ell) \text{ has } \ell/2 \text{ bits equal to } 0 \\ 0 & \text{otherwise.} \end{cases}$$

In this case, the algorithm **A** is deterministic. It is not hard to see that

$$E_{\mathbf{A}}(p_0) = \frac{\binom{\ell}{\ell/2}}{2^\ell}$$

and

$$E_{\mathbf{A}}(p_1) = 1.$$

It can be shown that

$$\lim_{\ell \to \infty} \frac{\binom{\ell}{\ell/2}}{2^\ell} = 0.$$

Hence, for any fixed value of $\epsilon < 1$, p_0 and p_1 are ϵ-distinguishable if ℓ is sufficiently large. □

12.2.1 Next Bit Predictors

Another useful concept in studying PRBGs is that of a next bit predictor, which works as follows. Let f be a (k, ℓ)-PRBG. Suppose we have a probabilistic algorithm $\mathbf{B}_i$, which takes as input the first $i-1$ bits produced by f (given an unknown seed), say $z_1, \ldots, z_{i-1}$, and attempts to predict the next bit z_i. The value i can be any value such that $0 \leq i \leq \ell - 1$. We say that $\mathbf{B}_i$ is an *ϵ-next bit predictor*

if $\mathbf{B}_i$ can predict the ith bit of a pseudo-random sequence with probability at least $1/2 + \epsilon$, where $\epsilon > 0$.

We can give a more precise formulation of this concept in terms of probability distributions, as follows. We have already defined the probability distribution p_1 on $(\mathbb{Z}_2)^\ell$ induced by the PRBG f. We can also look at the probability distributions induced by f on any of the ℓ pseudo-random output bits (or indeed on any subset of these ℓ output bits). So, for $1 \leq i \leq \ell$, we will can think of the ith pseudo-random output bit as a random variable that we will denote by $\mathbf{z}_i$.

In view of these definitions, we have the following characterization of a next bit predictor.

THEOREM 12.1

Let f be a (k, ℓ)-PRBG. Then the probabilistic algorithm $\mathbf{B}_i$ is an ϵ-next bit predictor for f if and only if

$$\sum_{(z_1,\ldots,z_{i-1})\in(\mathbb{Z}_2)^{i-1}} p_1(z_1,\ldots,z_{i-1}) \times p(\mathbf{z}_i = \mathbf{B}_i|(z_1,\ldots,z_{i-1})) \geq \frac{1}{2} + \epsilon.$$

PROOF The probability of correctly predicting the ith bit is computed by summing over all possible $(i-1)$-tuples $(z_1,\ldots,z_{i-1})$ the product of the probability that the $(i-1)$-tuple $(z_1,\ldots,z_{i-1})$ is produced by the PRBG and the probability that the ith bit is predicted correctly given the $(i-1)$-tuple $(z_1,\ldots,z_{i-1})$. ∎

The reason for the expression $1/2 + \epsilon$ in this definition is that any predicting algorithm can predict the next bit of a random sequence with probability $1/2$. If a sequence is not random, then it may be possible to predict the next bit with higher probability. (Note that it is unnecessary to consider algorithms that predict the next bit with probability less than $1/2$, because in this case an algorithm that replaces every prediction z by $1 - z$ will predict the next bit with probability greater than $1/2$.)

We illustrate these ideas by producing a next-bit predictor for the **Linear Congruential Generator** of Example 12.1.

Example 12.3 ***(Cont.)***

For any i such that $1 \leq i \leq 9$, Define $\mathbf{B}_i(z) = 1 - z$. That is, $\mathbf{B}_i$ predicts that a 0 is most likely to be followed by a 1, and vice versa. It is not hard to compute from Table 12.1 that each of these predictors $\mathbf{B}_i$ is a $\frac{9}{62}$-next bit predictor (i.e., they predict the next bit correctly with probability $20/31$). □

We can use a next bit predictor to construct a distinguishing algorithm $\mathbf{A}$, as shown in Figure 12.3. The input to algorithm $\mathbf{A}$ is a sequence of bits, $z_1,\ldots,z_\ell$, and $\mathbf{A}$ calls the algorithm $\mathbf{B}_i$ as a subroutine.

FIGURE 12.3
Constructing a distinguisher from a next bit predictor

Input: an ℓ-tuple $(z_1, \ldots, z_\ell)$
1. compute $z := \mathbf{B}_i(z_1, \ldots, z_{i-1})$
2. **if** $z = z_i$ **then**
   $\mathbf{A}(z_1, \ldots, z_\ell) = 1$
   else
   $\mathbf{A}(z_1, \ldots, z_\ell) = 0$.

THEOREM 12.2
Suppose $\mathbf{B}_i$ is an ϵ-next bit predictor for the (k, ℓ)-PRBG f. Let p_1 be the probability distribution induced on $(\mathbb{Z}_2)^\ell$ by f, and let p_0 be the uniform probability distribution on $(\mathbb{Z}_2)^\ell$. Then $\mathbf{A}$, as described in Figure 12.3, is an ϵ-distinguisher of p_1 and p_0.

PROOF First, observe that

$$\mathbf{A}_i(z_1, \ldots, z_\ell) = 1 \Leftrightarrow \mathbf{B}_i(z_1, \ldots, z_{i-1}) = z_i.$$

Also, the output of $\mathbf{A}$ is independent of the values of $z_{i+1}, \ldots, z_\ell$. Thus we can compute as follows:

$$\begin{aligned}
E_{\mathbf{A}}(p_1) &= \sum_{(z_1,\ldots,z_\ell)\in(\mathbb{Z}_2)^\ell} p_1(z_1, \ldots, z_\ell) \times p(\mathbf{A} = 1|(z_1, \ldots, z_\ell)) \\
&= \sum_{(z_1,\ldots,z_i)\in(\mathbb{Z}_2)^i} p_1(z_1, \ldots, z_i) \times p(\mathbf{A} = 1|(z_1, \ldots, z_i)) \\
&= \sum_{(z_1,\ldots,z_i)\in(\mathbb{Z}_2)^i} p_1(z_1, \ldots, z_i) \times p(\mathbf{B}_i = z_i|(z_1, \ldots, z_i)) \\
&= \sum_{(z_1,\ldots,z_{i-1})\in(\mathbb{Z}_2)^{i-1}} p_1(z_1, \ldots, z_{i-1}) \times p(\mathbf{z}_i = \mathbf{B}_i|(z_1, \ldots, z_{i-1})) \\
&\geq \frac{1}{2} + \epsilon.
\end{aligned}$$

On the other hand, any predictor $\mathbf{B}_i$ will predict the ith bit of a truly random sequence with probability $1/2$. Then, it is not difficult to see that $E_{\mathbf{A}}(p_0) = 1/2$. Hence,

$$|E_{\mathbf{A}}(p_0) - E_{\mathbf{A}}(p_1)| \geq \epsilon,$$

as desired. ∎

One of the main results in the theory of pseudo-random bit generators, due to Yao, is that a next bit predictor is a *universal* test. That is, a PRBG is "secure" if and only if there does not exist an ϵ-next bit predictor except for very small values of ϵ. Theorem 12.2 proves the implication in one direction. To prove the converse, we need to show how the existence of a distinguisher implies the existence of a next bit predictor. This is done in Theorem 12.3.

THEOREM 12.3
Suppose **A**, *is an ϵ-distinguisher of p_1 and p_0, where p_1 is the probability distribution induced on $(\mathbb{Z}_2)^\ell$ by the (k, ℓ)-PRBG f, and p_0 is the uniform probability distribution on $(\mathbb{Z}_2)^\ell$. Then for some i, $1 \leq i \leq \ell - 1$, there exists an ϵ/ℓ-next bit predictor $\mathbf{B}_i$ for f.*

PROOF For $0 \leq i \leq \ell$, define q_i to be a probability distribution on $(\mathbb{Z}_2)^\ell$ where the first i bits are generated using f, and the remaining $\ell - i$ bits are generated at random. Thus $q_0 = p_0$ and $q_\ell = p_1$. For $0 \leq i \leq \ell$, let $Q_i = E_{\mathbf{A}}(q_i)$. We are given that

$$|E_{\mathbf{A}}(q_0) - E_{\mathbf{A}}(q_\ell)| \geq \epsilon.$$

By the triangle inequality, we have that

$$|E_{\mathbf{A}}(q_0) - E_{\mathbf{A}}(q_\ell)| \leq \sum_{i=1}^{\ell} |E_{\mathbf{A}}(q_{i-1}) - E_{\mathbf{A}}(q_i)|.$$

Hence, it follows that there is at least one value i, $1 \leq i \leq \ell$, such that

$$|E_{\mathbf{A}}(q_{i-1}) - E_{\mathbf{A}}(q_i)| \geq \frac{\epsilon}{\ell}.$$

Without loss of generality, we will assume that

$$E_{\mathbf{A}}(q_{i-1}) - E_{\mathbf{A}}(q_i) \geq \frac{\epsilon}{\ell}.$$

We are going to construct an ith bit predictor (for this specified value of i). The predicting algorithm is probabilistic in nature and is presented in Figure 12.4. Here is the idea behind this construction. The predicting algorithm in fact produces an ℓ-tuple according to the probability distribution q_{i-1}, given that $z_1, \ldots, z_{\ell-1}$ are generated by the PRBG. If **A** answers "0," then it thinks that the ℓ-tuple was most likely generated according to the probability distribution q_i. Now q_{i-1} and q_i differ only in that the ith bit is generated at random in q_{i-1}, whereas it is generated according to the PRBG in q_i. Hence, when **A** answers "0," it thinks that the ith bit, z_i, is what would be produced by the PRBG. Hence, in this case we take z_i as our prediction of the ith bit. If **A** answers "1," it thinks that z_i is random, so we take $1 - z_i$ as our prediction of the ith bit.

FIGURE 12.4
Constructing a next bit predictor from a distinguisher

Input: an $(i-1)$-tuple $(z_1, \ldots, z_{i-1})$
1. choose $(z_i, \ldots, z_\ell) \in (\mathbb{Z}_2)^{\ell-i+1}$ at random
2. compute $z := \mathbf{A}(z_1, \ldots, z_\ell)$
3. define $\mathbf{B}_i(z_1, \ldots, z_{i-1}) = (z + z_i) \bmod 2$.

We need to compute the probability that the ith bit is predicted correctly. Observe that if $\mathbf{A}$ answers "0," then the prediction is correct with probability

$$p_1(z_i|(z_1, \ldots, z_{i-1})),$$

where p_1 is the probability distribution induced by the PRBG. If $\mathbf{A}$ answers "1," then the prediction is correct with probability

$$1 - p_1(z_i|(z_1, \ldots, z_{i-1})).$$

For brevity, we denote $\mathbf{z} = (z_1, \ldots, z_\ell)$. In our computation, we will make use of the fact that

$$q_{i-1}(\mathbf{z}) \times p_1(z_i|(z_1, \ldots, z_{i-1})) = \frac{q_i(\mathbf{z})}{2}.$$

This can be proved easily as follows:

$$\begin{aligned}
&q_{i-1}(z_1, \ldots, z_\ell) \times p_1(z_i|(z_1, \ldots, z_{i-1})) \\
&= q_{i-1}(z_1, \ldots, z_{i-1}) \times \frac{1}{2^{\ell-i+1}} \times p_1(z_i|(z_1, \ldots, z_{i-1})) \\
&= q_{i-1}(z_1, \ldots, z_i) \times \frac{1}{2^{\ell-i+1}} \\
&= \frac{q_{i-1}(z_1, \ldots, z_\ell)}{2}.
\end{aligned}$$

Now we can perform our main computation:

$$\begin{aligned}
&p(\mathbf{z}_i = \mathbf{B}_i(z_1, \ldots, z_{i-1})) \\
&= \sum_{\mathbf{z} \in (\mathbb{Z}_2)^\ell} q_{i-1}(\mathbf{z})[p(\mathbf{A} = 0|\mathbf{z}) \times p_1(z_i|(z_1, \ldots, z_{i-1})) \\
&\quad + p(\mathbf{A} = 1|\mathbf{z}) \times (1 - p_1(z_i|(z_1, \ldots, z_{i-1}))] \\
&= \sum_{\mathbf{z} \in (\mathbb{Z}_2)^\ell} \frac{q_i(\mathbf{z})}{2} \times p(\mathbf{A} = 0|\mathbf{z}) + \sum_{\mathbf{z} \in (\mathbb{Z}_2)^\ell} q_{i-1}(\mathbf{z}) \times p(\mathbf{A} = 1|\mathbf{z})
\end{aligned}$$

$$
\begin{aligned}
&- \sum_{\mathbf{z} \in (\mathbb{Z}_2)^\ell} \frac{q_i(\mathbf{z})}{2} \times p(\mathbf{A} = 1|\mathbf{z}) \\
&= \frac{1 - \epsilon_{\mathbf{A}}(q_i)}{2} + \epsilon_{\mathbf{A}}(q_{i-1}) - \frac{\epsilon_{\mathbf{A}}(q_i)}{2} \\
&= \frac{1}{2} + E_{\mathbf{A}}(q_{i-1}) - E_{\mathbf{A}}(q_i) \\
&\geq \frac{1}{2} + \frac{\epsilon}{\ell},
\end{aligned}
$$

which was what we wanted to prove. ∎

12.3 The Blum-Blum-Shub Generator

In this section we describe one of the most popular PRBGs, due to Blum, Blum, and Shub. First, we review some results on Jacobi symbols from Section 4.5 and other number-theoretic facts from other parts of Chapter 4.

Suppose p and q are two distinct primes, and let $n = pq$. Recall that the Jacobi symbol

$$
\left(\frac{x}{n}\right) = \begin{cases} 0 & \text{if } \gcd(x, n) > 1 \\ 1 & \text{if } \left(\frac{x}{p}\right) = \left(\frac{x}{q}\right) = 1 \text{ or if } \left(\frac{x}{p}\right) = \left(\frac{x}{q}\right) = -1 \\ -1 & \text{if one of } \left(\frac{x}{p}\right) \text{ and } \left(\frac{x}{q}\right) \text{ is 1 and the other is } -1. \end{cases}
$$

Denote the quadratic residues modulo n by $\mathrm{QR}(n)$. That is,

$$
\mathrm{QR}(n) = \{x^2 \bmod n : x \in \mathbb{Z}_n^*\}.
$$

Recall that x is a quadratic residue modulo n if and only if

$$
\left(\frac{x}{p}\right) = \left(\frac{x}{q}\right) = 1.
$$

Define

$$
\tilde{\mathrm{QR}}(n) = \left\{x \in \mathbb{Z}_n^* \backslash \mathrm{QR}(n) : \left(\frac{x}{n}\right) = 1\right\}.
$$

Thus

$$
\tilde{\mathrm{QR}}(n) = \left\{x \in \mathbb{Z}_n^* : \left(\frac{x}{p}\right) = \left(\frac{x}{q}\right) = -1\right\}.
$$

An element $x \in \tilde{\mathrm{QR}}(n)$ is called a *pseudo-square* modulo n.

FIGURE 12.5
Quadratic Residues

Problem Instance A positive integer n that is the product of two unknown primes p and q, and an integer $x \in \mathbb{Z}_n^*$ such that $\left(\frac{x}{n}\right) = 1$.
Question Is x a quadratic residue modulo n?

The **Blum-Blum-Shub Generator**, as well as some other cryptographic systems, is based on the **Quadratic Residues** problem defined in Figure 12.5. (In Chapter 4, we defined the **Quadratic Residues** problem modulo a prime and showed that it is easy to solve; here we have a composite modulus.) Observe that the **Quadratic Residues** problem requires us to distinguish quadratic residues modulo n from pseudo-squares modulo n. This can be no more difficult than factoring n. For if the factorization $n = pq$ can be computed, then it is a simple matter to compute $\left(\frac{x}{p}\right)$, say. Given that $\left(\frac{x}{n}\right) = 1$, it follows that x is a quadratic residue if and only if $\left(\frac{x}{p}\right) = 1$.

There does not appear to be any way to solve the **Quadratic Residues** problem efficiently if the factorization of n is not known. So this problem appears to be intractible if it is infeasible to factor n.

The **Blum-Blum-Shub Generator** is presented in Figure 12.6 The generator works quite simply. Given a seed $s_0 \in \mathrm{QR}(n)$, we compute the sequence $s_1, s_2, \ldots, s_\ell$ by successive squaring modulo n, and then reduce each s_i modulo 2 to obtain z_i. It follows that

$$z_i = \left({s_0}^{2^i} \bmod n\right) \bmod 2,$$

$1 \leq i \leq \ell$.

We now give an example of the **BBS Generator**.

Example 12.4
Suppose $n = 192649 = 383 \times 503$ and $s_0 = 101355^2 \bmod n = 20749$. The first 20 bits produced by the **BBS Generator** are computed as shown in Table 12.3. Hence the bit-string resulting from this seed is

$$11001110000100111010.$$

□

Here is a feature of the **BBS Generator** that is useful when we look at its security. Since $n = pq$ where $p \equiv q \equiv 3 \bmod 4$, it follows that for any quadratic

FIGURE 12.6
Blum-Blum-Shub Generator

Let p, q to be two $(k/2)$-bit primes such that $p \equiv q \equiv 3 \bmod 4$, and define $n = pq$. Let $\mathrm{QR}(n)$ denote the set of quadratic residues modulo n.

A seed s_0 is any element of $\mathrm{QR}(n)$. For $i \geq 1$, define

$$s_{i+1} = {s_i}^2 \bmod n,$$

and then define

$$f(s_0) = (z_1, z_2, \ldots, z_\ell),$$

where

$$z_i = s_i \bmod 2,$$

$1 \leq i \leq \ell$. Then f is a (k, ℓ)-PRBG, called the **Blum-Blum-Shub Generator**, which we abbreviate to **BBS Generator**.

TABLE 12.3
Bits produced by BBS generator

i	s_i	z_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

residue x, there is a unique square root of x that is also a quadratic residue. This square root is called the *principal* square root of x. It follows the mapping $x \mapsto x^2 \bmod n$ used to define the **BBS Generator** is a permutation on $\mathrm{QR}(n)$, the set of quadratic residues modulo n.

12.3.1 Security of the BBS Generator

In this section, we look at the security of the **BBS Generator** in detail. We begin by supposing that the pseudo-random bits produced by the **BBS Generator** are ϵ-distinguishable from ℓ random bits and then see where that leads us. Throughout this section, $n = pq$, where p and q are primes such that $p \equiv q \equiv 3 \bmod 4$, and the factorization $n = pq$ is unknown.

We have already discussed the idea of a next bit predictor. In this section we consider a similar concept that we call a *previous bit predictor*. A previous bit predictor for a (k, ℓ)-**BBS Generator** will take as input ℓ pseudo-random bits produced by the generator (as determined by an unknown random seed $s_0 \in \mathrm{QR}(n)$), and attempt to predict the value $z_0 = s_0 \bmod 2$. A previous bit predictor can be a probabilistic algorithm, and we say that a previous bit predictor $\mathbf{B}_0$ is an ϵ-previous bit predictor if its probability of correctly guessing z_0 is at least $1/2+\epsilon$, where this probability is computed over all possible seeds s_0.

We state the following theorem, which is similar to Theorem 12.3, without proof.

THEOREM 12.4

Suppose **A**, *is an ϵ-distinguisher of p_1 and p_0, where p_1 is the probability distribution induced on $(\mathbb{Z}_2)^\ell$ by the (k, ℓ)-***BBS Generator***, f, and p_0 is the uniform probability distribution on $(\mathbb{Z}_2)^\ell$. Then there exists an (ϵ/ℓ)-previous bit predictor $\mathbf{B}_0$ for f.*

We now show how to use an (ϵ/ℓ)-previous bit predictor, $\mathbf{B}_0$, to construct a probabilistic algorithm that distinguishes quadratic residues modulo n from pseudo-squares modulo n with probability $1/2 + \epsilon$. This algorithm **A**, presented in Figure 12.7, uses $\mathbf{B}_0$ as a subroutine, or oracle.

THEOREM 12.5

*Suppose $\mathbf{B}_0$ is an ϵ-previous bit predictor for the (k, ℓ)-***BBS Generator** f. Then the algorithm* **A**, *as described in Figure 12.7, determines quadratic residuosity correctly with probability at least $1/2+\epsilon$, where this probability is computed over all possible inputs $x \in \mathrm{QR}(n) \cup \tilde{\mathrm{QR}}(n)$.*

PROOF Since $n = pq$ and $p \equiv q \equiv 3 \bmod 4$, it follows that $\left(\frac{-1}{n}\right) = 1$, so $-1 \in \tilde{\mathrm{QR}}(n)$. Hence, if $\left(\frac{x}{n}\right) = 1$, then the principal square root of $s_0 = x^2$ is x if

FIGURE 12.7
Constructing a quadratic residue distinguisher from a previous bit predictor

Input: $x \in \mathbb{Z}_n{}^*$ such that $\left(\frac{x}{n}\right) = 1$
1. compute $s_0 = x^2 \bmod n$
2. use the **BBS Generator** to compute $z_1, \ldots, z_\ell$ from seed s_0
3. compute $z = \mathbf{B}_0(z_1, \ldots, z_\ell)$
4. **if** $(s_0 \bmod 2) = (z \bmod 2)$ **then**
   answer "$x \in \mathrm{QR}(n)$"
   else
   answer "$x \in \tilde{\mathrm{QR}}(n)$."

$x \in \mathrm{QR}(n)$; and $-x$ if $x \in \tilde{\mathrm{QR}}(n)$. But

$$(-x \bmod n) \bmod 2 \neq (x \bmod n) \bmod 2,$$

so it follows that algorithm **A** gives the correct answer if and only if $\mathbf{B}_0$ correctly predicts z_0. The result then follows immediately. ∎

Theorem 12.5 shows how we can distinguish pseudo-squares from quadratic residues with probability at least $1/2 + \epsilon$. We now show that this leads to a Monte Carlo algorithm that gives the correct answer with probability at least $1/2 + \epsilon$. In other words, for any $x \in \mathrm{QR}(n) \cup \tilde{\mathrm{QR}}(n)$, the Monte Carlo algorithm gives the correct answer with probabilty at least $1/2 + \epsilon$. Note that this algorithm is an *unbiased* algorithm (it may give an incorrect answer for any input) in contrast to the Monte Carlo algorithms that we studied in Section 4.5 which were all biased algorithms.

The Monte Carlo algorithm $\mathbf{A}_1$ is presented in Figure 12.8. It calls the previous algorithm **A** as a subroutine.

THEOREM 12.6
Suppose that algorithm **A** *determines quadratic residuosity correctly with probability at least* $1/2 + \epsilon$. *Then the algorithm* $\mathbf{A}_1$, *as described in Figure 12.8, is a Monte Carlo algorithm for* **Quadratic Residues** *with error probability at most* $1/2 - \epsilon$.

PROOF For any given input $x \in \mathrm{QR}(n) \cup \tilde{\mathrm{QR}}(n)$, the effect of step 3 in algorithm $\mathbf{A}_1$ is to produce an element x' that is a random element of $\mathrm{QR}(n) \cup \tilde{\mathrm{QR}}(n)$ whose status as a quadratic residue is known. ∎

FIGURE 12.8
A Monte Carlo algorithm for Quadratic Residues

Input: $x \in \mathbb{Z}_n{}^*$ such that $\left(\frac{x}{n}\right) = 1$

1. choose $r \in \mathbb{Z}_n^*$ at random
2. with probability $1/2$, compute

$$x' = r^2 x \bmod n,$$

otherwise compute

$$x' = -r^2 x \bmod n.$$

3. call $\mathbf{A}(x')$, obtaining an answer "QR" or "$\tilde{\text{QR}}$"
4. **if**

$$\mathbf{A}(x') = \text{QR and } x' = r^2 x \bmod n$$

or

$$\mathbf{A}(x') = \tilde{\text{QR}} \text{ and } x' = -r^2 x \bmod n$$

then

answer "$x \in$ QR"

else

answer "$x \in \tilde{\text{QR}}$."

The last step is to show that any (unbiased) Monte Carlo algorithm that has error probability at most $1/2 - \epsilon$ can be used to construct an unbiased Monte Carlo algorithm with error probability at most δ, for any $\delta > 0$. In other words, we can make the probability of correctness arbitrarily close to 1. The idea is to run the given Monte Carlo algorithm $2m + 1$ times, for some integer m, and take the "majority vote" as the answer. By computing the error probability of this algorithm, we can also see how m depends on δ. This dependence is stated in the following theorem.

THEOREM 12.7

Suppose $\mathbf{A}_1$ *is an unbiased Monte Carlo algorithm with error probability at most* $1/2 - \epsilon$. *Suppose we run* $\mathbf{A}_1$ $n = 2m + 1$ *times on a given instance* I, *and we take the most frequent answer. Then the error probability of the resulting algorithm is*

at most

$$\frac{(1-4\epsilon^2)^m}{2}.$$

PROOF The probability of obtaining exactly i correct answers in the n trials is at most

$$\binom{n}{i}\left(\frac{1}{2}+\epsilon\right)^i\left(\frac{1}{2}-\epsilon\right)^{n-i}.$$

The probability that the most frequent answer is incorrect is equal to the probability that the number of correct answers in the n trials is at most m. Hence, we compute as follows

$$\begin{aligned}
p(\text{error}) &\le \sum_{i=0}^{m}\binom{n}{i}\left(\frac{1}{2}+\epsilon\right)^i\left(\frac{1}{2}-\epsilon\right)^{2m+1-i} \\
&= \left(\frac{1}{2}+\epsilon\right)^m\left(\frac{1}{2}-\epsilon\right)^{m+1}\sum_{i=0}^{m}\binom{n}{i}\left(\frac{1/2-\epsilon}{1/2+\epsilon}\right)^{m-i} \\
&\le \left(\frac{1}{2}+\epsilon\right)^m\left(\frac{1}{2}-\epsilon\right)^{m+1}\sum_{i=0}^{m}\binom{n}{i} \\
&= \left(\frac{1}{2}+\epsilon\right)^m\left(\frac{1}{2}-\epsilon\right)^{m+1}2^{2m} \\
&= \left(\frac{1}{4}-\epsilon^2\right)^m\left(\frac{1}{2}-\epsilon\right)2^{2m} \\
&= (1-4\epsilon^2)^m\left(\frac{1}{2}-\epsilon\right) \\
&\le \frac{(1-4\epsilon^2)^m}{2},
\end{aligned}$$

as required. ∎

Suppose we want to lower the probability of error to some value δ, where $0 < \delta < 1/2 - \epsilon$. We need to choose m so that

$$\frac{(1-4\epsilon^2)^m}{2} \le \delta.$$

Hence, it suffices to take

$$m = \left\lceil \frac{1+\log_2 \delta}{\log_2(1-4\epsilon^2)} \right\rceil.$$

Then, if algorithm A is run $2m + 1$ times, the majority vote yields the correct answer with probability at least $1 - \delta$. It is not hard to show that this value of m is at most $c/(\delta\epsilon^2)$ for some constant c. Hence, the number of times that the algorithm must be run is polynomial in $1/\delta$ and $1/\epsilon$.

Example 12.5
Suppose we start with a Monte Carlo algorithm that returns the correct answer with probability at least .55, so $\epsilon = .05$. If we desire a Monte Carlo algorithm in which the probability of error is at most .05, then it suffices to take $m = 230$ and $n = 471$. □

Let us combine all the reductions we have done. We have the following sequence of implications:

(k, ℓ)-**BBS Generator** can be ϵ-distinguished from ℓ random bits
$\Downarrow$
(ϵ/ℓ)-previous bit predictor for (k, ℓ)-**BBS Generator**
$\Downarrow$
distinguishing algorithm for **Quadratic Residues** that is correct with probability at least $1/2 + \epsilon/\ell$
$\Downarrow$
unbiased Monte Carlo algorithm for **Quadratic Residues** having error probability at most $1/2 - \epsilon/\ell$
$\Downarrow$
unbiased Monte Carlo algorithm for **Quadratic Residues** having error probability at most δ, for any $\delta > 0$

Since it is widely believed that there is no polynomial-time Monte Carlo algorithm for **Quadratic Residues** with small error probability, we have some evidence that the **BBS Generator** is secure.

We close this section by mentioning a way of improving the efficiency of the **BBS Generator**. The sequence of pseudo-random bits is constructed by taking the least significant bit of each s_i, where $s_i = s_0^{2^i} \bmod n$. Suppose instead that we extract the m least significant bits from each s_i. This will improve the efficiency of the LFSR by a factor of m, but we need to ask if the LFSR will remain secure. It has been shown that this approach will remain secure provided that $m \leq \log_2 \log_2 n$. So we can extract about $\log_2 \log_2 n$ pseudo-random bits per modular squaring. In a realistic implementation of the **BBS Generator**, $n \approx 10^{160}$, so we can extract nine bits per squaring.

12.4 Probabilistic Encryption

Probabilistic encryption is an idea of Goldwasser and Micali. One motivation is as follows. Suppose we have a public-key cryptosystem, and we wish to encrypt a single bit, i.e., $x = 0$ or 1. Since anyone can compute $e_K(0)$ and $e_K(1)$, it is a simple matter for an opponent to determine if a ciphertext y is an encryption of 0 or an encryption of 1. More generally, an opponent can always determine if the plaintext has a specified value by encrypting a hypothesized plaintext, hoping to match a given ciphertext.

The goal of probabilistic encryption is that "no information" about the plaintext should be computable from the ciphertext (in polynomial time). This objective can be realized by a public-key cryptosystem in which encryption is probabilistic rather than deterministic. Since there are "many" possible encryptions of each plaintext, it is not feasible to text whether a given ciphertext is an encryption of a particular plaintext.

Here is a formal mathematical definition of this concept.

DEFINITION 12.3 *A **probabilistic public-key cryptosystem** is defined to be a six-tuple $(\mathcal{P}, \mathcal{C}, \mathcal{K}, \mathcal{E}, \mathcal{D}, \mathcal{R})$, where $\mathcal{P}$ is the set of **plaintexts**, $\mathcal{C}$ is the set of **ciphertexts**, $\mathcal{K}$ is the **keyspace**, and for each **key** $K \in \mathcal{K}$, $e_K \in \mathcal{E}$ is a public **encryption rule** and $d_K \in \mathcal{D}$ is a secret **decryption rule**. The following properties should be satisfied:*

1. *Each $e_K : \mathcal{P} \times \mathcal{R} \to \mathcal{C}$ and $d_K : \mathcal{C} \to \mathcal{P}$ are functions such that*

$$d_K(e_K(b, r)) = b$$

   for every plaintext $b \in \mathcal{P}$ and every $r \in \mathcal{R}$. (In particular, this implies that $e_K(x, r) \neq e_K(x', r,)$ if $x \neq x'$.)

2. *Let ϵ be a specified **security parameter**. For any fixed $K \in \mathcal{K}$ and for any $x \in \mathcal{P}$, define a probability distribution $p_{K,x}$ on $\mathcal{C}$, where $p_{K,x}(y)$ denotes the probability that y is the ciphertext given that K is the key and x is the plaintext (this probability is computed over all $r \in \mathcal{R}$). Suppose $x, x' \in \mathcal{P}$, $x \neq x'$, and $K \in \mathcal{K}$. Then the probability distributions $p_{K,x}$ and $p_{K,x'}$ are not ϵ-distinguishable.*

Here is how the system works. To encrypt a plaintext x, choose a random $r \in \mathcal{R}$ and compute $y = e_K(x, r)$. Any such value $y = e_K(x, r)$ can be decrypted to x. Property 2 is stating that the probability distribution of all encryptions of x cannot be distinguished from the probability distribution of all encryptions of x' if $x' \neq x$. Informally, an encryption of x "looks like" an encryption of x'. The security parameter ϵ should be small: in practice we would want to have $\epsilon = c/|\mathcal{R}|$ for some small $c > 0$.

FIGURE 12.9
Goldwasser-Micali Probabilistic Public-key Cryptosystem

Let $n = pq$, where p and q are primes, and let $m \in \tilde{\text{QR}}(n)$. The integers n and m are public; the factorization $n = pq$ is secret. Let $\mathcal{P} = \{0, 1\}$, $\mathcal{C} = \mathcal{R} = \mathbb{Z}_n{}^*$, and define

$$\mathcal{K} = \{(n, p, q, m) : n = pq, p, q \text{ prime}, m \in \tilde{\text{QR}}(n)\}.$$

For $K = (n, p, q, m)$, define

$$e_K(x, r) = m^x r^2 \bmod n$$

and

$$d_K(y) = \begin{cases} 0 & \text{if } y \in \text{QR}(n) \\ 1 & \text{if } y \notin \text{QR}(n), \end{cases}$$

where $x = 0$ or 1 and $r, y \in \mathbb{Z}_n{}^*$.

We now present the **Goldwasser-Micali Probabilistic Public-key Cryptosystem** in Figure 12.9. This system encrypts one bit at a time. A 0 bit is encrypted to a random quadratic residue modulo n; a 1 bit is encrypted to a random pseudo-square modulo n. When Bob recieves an element $y \in \text{QR}(n) \cup \tilde{\text{QR}}(n)$, he can use his knowledge of the factorization of n to determine whether $y \in \text{QR}(n)$ or whether $y \in \tilde{\text{QR}}(n)$. He does this by computing

$$\left(\frac{y}{n}\right) = (-1)^{(p-1)/2} \bmod p;$$

then

$$y \in \text{QR}(n) \Leftrightarrow \left(\frac{y}{n}\right) = 1.$$

A more efficient probabilistic public-key cryptosystem was given by Blum and Goldwasser. The **Blum-Goldwasser Probabilistic Public-key Cryptosystem** is presented in Figure 12.10. The basic idea is as follows. A random seed s_0 generates a sequence of ℓ psuedorandom bits $z_1, \ldots, z_\ell$ using the **BBS Generator**. The z_i's are used as a keystream, i.e., they are exclusive-ored with the ℓ plaintext bits to form the ciphertext. As well, the $(\ell + 1)$st element $s_{\ell+1} = s_0{}^{2^{\ell+1}} \bmod n$ is transmitted as part of the ciphertext.

When Bob receives the ciphertext, he can compute s_0 from $s_{\ell+1}$, then reconstruct the keystream, and finally exclusive-or the keystream with the ℓ ciphertext bits to obtain the plaintext. We should explain how Bob derives s_0 from $s_{\ell+1}$. Recall that each s_{i-1} is the principal square root of s_i. Now, $n = pq$ with

FIGURE 12.10
Blum-Goldwasser Probabilistic Public-key Cryptosystem

Let $n = pq$, where p and q are primes, $p \equiv q \equiv 3 \bmod 4$. The integer n is public; the factorization $n = pq$ is secret. Let $\mathcal{P} = (\mathbb{Z}_2)^\ell$, $\mathcal{C} = (\mathbb{Z}_2)^\ell \times {\mathbb{Z}_n}^*$ and $\mathcal{R} = {\mathbb{Z}_n}^*$. Define

$$\mathcal{K} = \{(n, p, q) : n = pq, p, q \text{ prime}\}.$$

For $K = (n, p, q)$ and $x \in (\mathbb{Z}_2)^\ell$ and $r \in {\mathbb{Z}_n}^*$, encrypt x as follows:

1. Compute $z_1, \ldots, z_\ell$ from seed $s_0 = r$ using the **BBS Generator**.
2. Compute $s_{\ell+1} = {s_0}^{2^{\ell+1}} \bmod n$.
3. Compute $y_i = (x_i + z_i) \bmod 2$ for $1 \leq i \leq \ell$.
4. Define $y = (y_1, \ldots, y_\ell, s_{\ell+1})$.

To decrypt y, Bob performs the following steps:

1. Compute $a_1 = ((p+1)/4)^{\ell+1} \bmod (p-1)$.
2. Compute $a_2 = ((q+1)/4)^{\ell+1} \bmod (q-1)$.
3. Compute $b_1 = {s_{\ell+1}}^{a_1} \bmod p$.
4. Compute $b_2 = {s_{\ell+1}}^{a_2} \bmod q$.
5. Use the Chinese remainder theorem to find s_0 such that

$$s_0 \equiv b_1 \bmod p$$

and

$$s_0 \equiv b_2 \bmod q.$$

6. Compute $z_1, \ldots, z_\ell$ from seed $s_0 = r$ using the **BBS Generator**.
7. Compute $x_i = (y_i + z_i) \bmod 2$ for $1 \leq i \leq \ell$.
8. The plaintext $x = (x_1, \ldots, x_\ell)$.

$p \equiv q \equiv 3 \bmod 4$, so the square roots of any quadratic residue x modulo p are $\pm x^{(p+1)/4}$. Using properties of Jacobi symbols, we have that

$$\left(\frac{x^{(p+1)/4}}{p}\right) = \left(\frac{1}{p}\right)^{(p+1)/4} = 1.$$

It follows that $x^{(p+1)/4}$ is the principal square root of x modulo p. Similarly, $x^{(q+1)/4}$ is the principal square root of x modulo q. Then, using the the Chinese remainder theorem, we can find the principal square root of x modulo n.

More generally, $x^{((p+1)/4)^{\ell+1}}$ will be the principal $2^{\ell+1}$st root of x modulo p and $x^{((p+1)/4)^{\ell+1}}$ will be the principal $2^{\ell+1}$st root of x modulo q. Since $\mathbb{Z}_p^*$ has order $p-1$, we can reduce the exponent $((p+1)/4)^{\ell+1}$ modulo $p-1$ in the computation $x^{((p+1)/4)^{\ell+1}} \bmod p$. Similarly, we can reduce the exponent $((q+1)/4)^{\ell+1}$ modulo $q-1$. In Figure 12.10, having obtained the principal $2^{\ell+1}$st roots of $s_{\ell+1}$ modulo p and modulo q (steps 1-4 of the decryption process), the Chinese remainder theorem is used to compute the principal $2^{\ell+1}$st roots of $s_{\ell+1}$ modulo n.

Here is an example to illustrate.

Example 12.6

Suppose $n = 192649$, as in Example 12.4. Suppose further that Alice chooses $r = 20749$ and wants to encrypt the 20-bit plaintext string

$$x = 11010011010011101101.$$

She will first compute the keystream

$$z = 11001110000100111010,$$

exactly as in Example 12.4, and then exclusive-or it with the plaintext, to obtain the ciphertext

$$y = 00011101010111010111$$

which she transmits to Bob. She also computes

$$s_{21} = {s_{20}}^2 \bmod n = 94739$$

and sends it to Bob.

Of course Bob knows the factorization $n = 383 \times 503$, so $(p+1)/4 = 96$ and $(q+1)/4 = 126$. He begins by computing

$$\begin{aligned} a_1 &= ((p+1)/4)^{\ell+1} \bmod (p-1) \\ &= 96^{21} \bmod 382 \\ &= 266 \end{aligned}$$

and

$$\begin{aligned} a_2 &= ((q+1)/4)^{\ell+1} \bmod (q-1) \\ &= 126^{21} \bmod 502 \\ &= 486. \end{aligned}$$

Next, he calculates

$$\begin{aligned} b_1 &= {s_{21}}^{a_1} \bmod p \\ &= 94739^{266} \bmod 383 \\ &= 67 \end{aligned}$$

and

$$\begin{aligned} b_2 &= {s_{21}}^{a_2} \bmod q \\ &= 94739^{486} \bmod 503 \\ &= 126. \end{aligned}$$

Now Bob proceeds to solve the system of congruences

$$\begin{aligned} r &\equiv 67 \pmod{383} \\ r &\equiv 126 \pmod{503} \end{aligned}$$

to obtain Alice's seed $r = 20749$. Then he constructs Alice's keystream from r. Finally, he exclusive-ors the keystream with the ciphertext to get the plaintext. □

12.5 Notes and References

A lengthy treatment of PRBGs can be found in the book by Kranakis [KR86]. See also the survey paper by Lagarias [LA90].

The **Shrinking Generator** is due to Coppersmith, Krawczyk, and Mansour [CKM94]; another practical method of constructing PBRGs using LFSRs has been given by Gunther [GU88]. For methods of breaking the **Linear Congruential Generator**, see Boyar [BO89].

The basic theory of secure PRBGs is due to Yao [YA82], who proved the universality of the next bit test. Further basic results can be found in Blum and Micali [BM84]. The **BBS Generator** is described in [BBS86]. The security of the **Quadratic Residues** problem is studied by Goldwasser and Micali [GM84],

FIGURE 12.11
Discrete Logarithm Generator

Let p be a k-bit prime, and let α be a primitive element modulo p.

A seed x_0 is any element of $\mathbb{Z}_p^*$. For $i \geq 1$, define

$$x_{i+1} = \alpha^{x_i} \bmod p,$$

and then define

$$f(x_0) = (z_1, z_2, \ldots, z_\ell),$$

where

$$z_i = \begin{cases} 1 & \text{if } x_i > p/2 \\ 0 & \text{if } x_i < p/2. \end{cases}$$

Then f is called a (k, ℓ)-**Discrete Logarithm Generator**.

on which we based much of Section 12.3.1. We have, however, used the approach of Brassard and Bratley [BB88A, Section 8.6] to reduce the error probability of an unbiased Monte Carlo algorithm.

The **RSA Generator** is studied in Alexi, Chor, Goldreich, and Schnorr [ACGS88]. PRBGs based on the **Discrete Logarithm** problem are treated in Blum and Micali [BM84], Long and Wigderson [LW88], and Håstad, Schrift, and Shamir [HSS93]. A sufficient condition for the secure extraction of multiple bits per iteration of a PRBG was proved by Vazirani and Vazirani [VV84].

The concept of probabilistic encryption is due to Goldwasser and Micali [GM84]; the **Blum-Goldwasser Cryptosystem** is presented in [BG85].

Exercises

12.1 Consider the **Linear Congruential Generator** defined by $s_i = (as_{i-1} + b) \bmod M$. Suppose that $M = qa + 1$ where a is odd and q is even, and suppose that $b = 1$. Show that the next bit predictor $\mathbf{B}_i(z) = 1 - z$ for the i bit is an ϵ-next bit predictor, where

$$\frac{1}{2} + \epsilon = \frac{q(a+1)}{2M}.$$

12.2 Suppose we have an **RSA Generator** with $n = 36863, b = 229$ and seed $s_0 = 25$. Compute the first 100 bits produced by this generator.

12.3 A PRBG based on the **Discrete Logarithm** problem is given in Figure 12.11. Suppose $p = 21383$, the primitive element $\alpha = 5$ and the seed $s_0 = 15886$. Compute the first 100 bits produced by this generator.

12.4 Suppose that Bob has knowledge of the factorization $n = pq$ in the **BBS Generator**.
(a) Show how Bob can use this knowledge to compute any s_i from s_0 with $2k$ multiplications modulo $\phi(n)$ and $2k$ multiplications modulo n, where n

TABLE 12.4
Blum-Goldwasser Ciphertext

```
E1866663F17FDBD1DC8C8FD2EEBC36AD7F53795DBA3C9CE22D
C9A9C7E2A56455501399CA6B98AED22C346A529A09C1936C61
ECDE10B43D226EC683A669929F2FFB912BFA96A8302188C083
46119E4F61AD8D0829BD1CDE1E37DBA9BCE65F40C0BCE48A80
0B3D087D76ECD1805C65D9DB730B8D0943266D942CF04D7D4D
76BFA891FA21BE76F767F1D5DCC7E3F1D86E39A9348B3
```

has k bits in its binary representation. (If i is large compared to k, then this approach represents a substantial improvement over the i multiplications required to sequentially compute $s_0, \ldots, s_i$.)

(b) Use this method to compute s_{10000} if $n = 59701 = 227 \times 263$ and $s_0 = 17995$.

12.5 We proved that, in order to reduce the error probability of an unbiased Monte Carlo algorithm from $1/2 - \epsilon$ to δ, where $\delta + \epsilon < 1/2$, it suffices to run the algorithm m times, where

$$m = \left\lceil \frac{1 + \log_2 \delta}{\log_2(1 - 4\epsilon^2)} \right\rceil .$$

Prove that this value of m is $1/(\delta\epsilon^2))$.

12.6 Suppose Bob receives some ciphertext which was encrypted with the **Blum-Goldwasser Probabilistic Public-key Cryptosystem**. The original plaintext consisted of English text. Each alphabetic character was converted to a bitstring of length five in the obvious way: $A \leftrightarrow 00000, B \leftrightarrow 00001, \ldots, Z \leftrightarrow 11001$. The plaintext consisted of 236 alphabetic characters, so a bitstring of length 1180 resulted. This bitstring was then encrypted. The resulting ciphertext bitstring was then converted to a hexadecimal representation, to save space. The final string of 295 hexadecimal characters is presented in Table 12.4. Also, $s_{1181} = 20291$ is part of the ciphertext, and $n = 29893$ is Bob's public key. Bob's secret factorization of n is $n = pq$, where $p = 167$ and $q = 179$.

Your task is to decrypt the given ciphertext and restore the original English plaintext, which was taken from "Under the Hammer," by John Mortimer, Penguin Books, 1994.