Chapter 5

Pseudorandom Bits and Sequences

Contents in Brief

5.1	**Introduction**	**169**
5.2	**Random bit generation**	**171**
5.3	**Pseudorandom bit generation**	**173**
5.4	**Statistical tests**	**175**
5.5	**Cryptographically secure pseudorandom bit generation**	**185**
5.6	**Notes and further references**	**187**

5.1 Introduction

The security of many cryptographic systems depends upon the generation of unpredictable quantities. Examples include the keystream in the one-time pad (§1.5.4), the secret key in the DES encryption algorithm (§7.4.2), the primes p, q in the RSA encryption (§8.2) and digital signature (§11.3.1) schemes, the private key a in the DSA (§11.5.1), and the challenges used in challenge-response identification systems (§10.3). In all these cases, the quantities generated must be of sufficient size and be "random" in the sense that the probability of any particular value being selected must be sufficiently small to preclude an adversary from gaining advantage through optimizing a search strategy based on such probability. For example, the key space for DES has size 2^{56}. If a secret key k were selected using a true random generator, an adversary would on average have to try 2^{55} possible keys before guessing the correct key k. If, on the other hand, a key k were selected by first choosing a 16-bit random secret s, and then expanding it into a 56-bit key k using a complicated but publicly known function f, the adversary would on average only need to try 2^{15} possible keys (obtained by running every possible value for s through the function f).

This chapter considers techniques for the generation of random and pseudorandom bits and numbers. Related techniques for pseudorandom bit generation that are generally discussed in the literature in the context of stream ciphers, including linear and nonlinear feedback shift registers (Chapter 6) and the output feedback mode (OFB) of block ciphers (Chapter 7), are addressed elsewhere in this book.

Chapter outline

The remainder of §5.1 introduces basic concepts relevant to random and pseudorandom bit generation. §5.2 considers techniques for random bit generation, while §5.3 considers some techniques for pseudorandom bit generation. §5.4 describes statistical tests designed to measure the quality of a random bit generator. Cryptographically secure pseudorandom bit generators are the topic of §5.5. §5.6 concludes with references and further chapter notes.

5.1.1 Background and Classification

5.1 Definition A *random bit generator* is a device or algorithm which outputs a sequence of statistically independent and unbiased binary digits.

5.2 Remark (*random bits vs. random numbers*) A random bit generator can be used to generate (uniformly distributed) random numbers. For example, a random integer in the interval $[0, n]$ can be obtained by generating a random bit sequence of length $\lfloor \lg n \rfloor + 1$, and converting it to an integer; if the resulting integer exceeds n, one option is to discard it and generate a new random bit sequence.

§5.2 outlines some physical sources of random bits that are used in practice. Ideally, secrets required in cryptographic algorithms and protocols should be generated with a (true) random bit generator. However, the generation of random bits is an inefficient procedure in most practical environments. Moreover, it may be impractical to securely store and transmit a large number of random bits if these are required in applications such as the one-time pad (§6.1.1). In such situations, the problem can be ameliorated by substituting a random bit generator with a pseudorandom bit generator.

5.3 Definition A *pseudorandom bit generator* (PRBG) is a deterministic[1] algorithm which, given a truly random binary sequence of length k, outputs a binary sequence of length $l \gg k$ which "appears" to be random. The input to the PRBG is called the *seed*, while the output of the PRBG is called a *pseudorandom bit sequence*.

The output of a PRBG is *not* random; in fact, the number of possible output sequences is at most a small fraction, namely $2^k/2^l$, of all possible binary sequences of length l. The intent is to take a small truly random sequence and expand it to a sequence of much larger length, in such a way that an adversary cannot efficiently distinguish between output sequences of the PRBG and truly random sequences of length l. §5.3 discusses ad-hoc techniques for pseudorandom bit generation. In order to gain confidence that such generators are secure, they should be subjected to a variety of statistical tests designed to detect the specific characteristics expected of random sequences. A collection of such tests is given in §5.4. As the following example demonstrates, passing these statistical tests is a *necessary* but not *sufficient* condition for a generator to be secure.

5.4 Example (*linear congruential generators*) A *linear congruential generator* produces a pseudorandom sequence of numbers $x_1, x_2, x_3, \ldots$ according to the linear recurrence

$$x_n \ = \ ax_{n-1} + b \bmod m, \quad n \geq 1;$$

integers a, b, and m are *parameters* which characterize the generator, while x_0 is the (secret) *seed*. While such generators are commonly used for simulation purposes and probabilistic algorithms, and pass the statistical tests of §5.4, they are *predictable* and hence entirely insecure for cryptographic purposes: given a partial output sequence, the remainder of the sequence can be reconstructed even if the parameters a, b, and m are unknown. □

[1] *Deterministic* here means that given the same initial seed, the generator will always produce the same output sequence.

A minimum security requirement for a pseudorandom bit generator is that the length k of the random seed should be sufficiently large so that a search over 2^k elements (the total number of possible seeds) is infeasible for the adversary. Two general requirements are that the output sequences of a PRBG should be statistically indistinguishable from truly random sequences, and the output bits should be unpredictable to an adversary with limited computational resources; these requirements are captured in Definitions 5.5 and 5.6.

5.5 Definition A pseudorandom bit generator is said to pass all *polynomial-time*[2] *statistical tests* if no polynomial-time algorithm can correctly distinguish between an output sequence of the generator and a truly random sequence of the same length with probability significantly greater that $\frac{1}{2}$.

5.6 Definition A pseudorandom bit generator is said to pass the *next-bit test* if there is no polynomial-time algorithm which, on input of the first l bits of an output sequence s, can predict the $(l+1)^{\text{st}}$ bit of s with probability significantly greater than $\frac{1}{2}$.

Although Definition 5.5 appears to impose a more stringent security requirement on pseudorandom bit generators than Definition 5.6 does, the next result asserts that they are, in fact, equivalent.

5.7 Fact (*universality of the next-bit test*) A pseudorandom bit generator passes the next-bit test if and only if it passes all polynomial-time statistical tests.

5.8 Definition A PRBG that passes the next-bit test (possibly under some plausible but unproved mathematical assumption such as the intractability of factoring integers) is called a *cryptographically secure pseudorandom bit generator* (CSPRBG).

5.9 Remark (*asymptotic nature of Definitions 5.5, 5.6, and 5.8*) Each of the three definitions above are given in complexity-theoretic terms and are asymptotic in nature because the notion of "polynomial-time" is meaningful for asymptotically large inputs only; the resulting notions of security are relative in the same sense. To be more precise in Definitions 5.5, 5.6, 5.8, and Fact 5.7, a pseudorandom bit generator is actually a *family* of such PRBGs. Thus the theoretical security results for a family of PRBGs are only an indirect indication about the security of individual members.

Two cryptographically secure pseudorandom bit generators are presented in §5.5.

5.2 Random bit generation

A (true) random bit generator requires a naturally occurring source of randomness. Designing a hardware device or software program to exploit this randomness and produce a bit sequence that is free of biases and correlations is a difficult task. Additionally, for most cryptographic applications, the generator must not be subject to observation or manipulation by an adversary. This section surveys some potential sources of random bits.

Random bit generators based on natural sources of randomness are subject to influence by external factors, and also to malfunction. It is imperative that such devices be tested periodically, for example by using the statistical tests of §5.4.

[2]The running time of the test is bounded by a polynomial in the length l of the output sequence.

(i) Hardware-based generators

Hardware-based random bit generators exploit the randomness which occurs in some physical phenomena. Such physical processes may produce bits that are biased or correlated, in which case they should be subjected to de-skewing techniques mentioned in (iii) below. Examples of such physical phenomena include:

1. elapsed time between emission of particles during radioactive decay;
2. thermal noise from a semiconductor diode or resistor;
3. the frequency instability of a free running oscillator;
4. the amount a metal insulator semiconductor capacitor is charged during a fixed period of time;
5. air turbulence within a sealed disk drive which causes random fluctuations in disk drive sector read latency times; and
6. sound from a microphone or video input from a camera.

Generators based on the first two phenomena would, in general, have to be built externally to the device using the random bits, and hence may be subject to observation or manipulation by an adversary. Generators based on oscillators and capacitors can be built on VLSI devices; they can be enclosed in tamper-resistant hardware, and hence shielded from active adversaries.

(ii) Software-based generators

Designing a random bit generator in software is even more difficult than doing so in hardware. Processes upon which software random bit generators may be based include:

1. the system clock;
2. elapsed time between keystrokes or mouse movement;
3. content of input/output buffers;
4. user input; and
5. operating system values such as system load and network statistics.

The behavior of such processes can vary considerably depending on various factors, such as the computer platform. It may also be difficult to prevent an adversary from observing or manipulating these processes. For instance, if the adversary has a rough idea of when a random sequence was generated, she can guess the content of the system clock at that time with a high degree of accuracy. A well-designed software random bit generator should utilize as many good sources of randomness as are available. Using many sources guards against the possibility of a few of the sources failing, or being observed or manipulated by an adversary. Each source should be sampled, and the sampled sequences should be combined using a complex *mixing function*; one recommended technique for accomplishing this is to apply a cryptographic hash function such as SHA-1 (Algorithm 9.53) or MD5 (Algorithm 9.51) to a concatenation of the sampled sequences. The purpose of the mixing function is to distill the (true) random bits from the sampled sequences.

(iii) De-skewing

A natural source of random bits may be defective in that the output bits may be *biased* (the probability of the source emitting a 1 is not equal to $\frac{1}{2}$) or *correlated* (the probability of the source emitting a 1 depends on previous bits emitted). There are various techniques for generating truly random bit sequences from the output bits of such a defective generator; such techniques are called *de-skewing techniques*.

5.10 Example (*removing biases in output bits*) Suppose that a generator produces biased but uncorrelated bits. Suppose that the probability of a 1 is p, and the probability of a 0 is $1-p$, where p is unknown but fixed, $0 < p < 1$. If the output sequence of such a generator is grouped into pairs of bits, with a 10 pair transformed to a 1, a 01 pair transformed to a 0, and 00 and 11 pairs discarded, then the resulting sequence is both unbiased and uncorrelated. □

A practical (although not provable) de-skewing technique is to pass sequences whose bits are biased or correlated through a cryptographic hash function such as SHA-1 or MD5.

5.3 Pseudorandom bit generation

A one-way function f (Definition 1.12) can be utilized to generate pseudorandom bit sequences (Definition 5.3) by first selecting a random seed s, and then applying the function to the sequence of values $s, s+1, s+2, \dots$; the output sequence is $f(s), f(s+1), f(s+2), \dots$. Depending on the properties of the one-way function used, it may be necessary to only keep a few bits of the output values $f(s+i)$ in order to remove possible correlations between successive values. Examples of suitable one-way functions f include a cryptographic hash function such as SHA-1 (Algorithm 9.53), or a block cipher such as DES (§7.4) with secret key k.

Although such ad-hoc methods have not been proven to be cryptographically secure, they appear sufficient for most applications. Two such methods for pseudorandom bit and number generation which have been standardized are presented in §5.3.1 and §5.3.2. Techniques for the cryptographically secure generation of pseudorandom bits are given in §5.5.

5.3.1 ANSI X9.17 generator

Algorithm 5.11 is a U.S. Federal Information Processing Standard (FIPS) approved method from the ANSI X9.17 standard for the purpose of pseudorandomly generating keys and initialization vectors for use with DES. E_k denotes DES E-D-E two-key triple-encryption (Definition 7.32) under a key k; the key k should be reserved exclusively for use in this algorithm.

5.11 Algorithm ANSI X9.17 pseudorandom bit generator

INPUT: a random (and secret) 64-bit seed s, integer m, and DES E-D-E encryption key k.
OUTPUT: m pseudorandom 64-bit strings $x_1, x_2, \dots, x_m$.

1. Compute the intermediate value $I = E_k(D)$, where D is a 64-bit representation of the date/time to as fine a resolution as is available.
2. For i from 1 to m do the following:
   2.1 $x_i \leftarrow E_k(I \oplus s)$.
   2.2 $s \leftarrow E_k(x_i \oplus I)$.
3. Return$(x_1, x_2, \dots, x_m)$.

Each output bitstring x_i may be used as an initialization vector (IV) for one of the DES modes of operation (§7.2.2). To obtain a DES key from x_i, every eighth bit of x_i should be reset to odd parity (cf. §7.4.2).

5.3.2 FIPS 186 generator

The algorithms presented in this subsection are FIPS-approved methods for pseudorandomly generating the secret parameters for the DSA (§11.5.1). Algorithm 5.12 generates DSA private keys a, while Algorithm 5.14 generates the per-message secrets k to be used in signing messages. Both algorithms use a secret seed s which should be randomly generated, and utilize a one-way function constructed by using either SHA-1 (Algorithm 9.53) or DES (Algorithm 7.82), respectively described in Algorithms 5.15 and 5.16.

5.12 Algorithm FIPS 186 pseudorandom number generator for DSA private keys

INPUT: an integer m and a 160-bit prime number q.
OUTPUT: m pseudorandom numbers $a_1, a_2, \ldots, a_m$ in the interval $[0, q-1]$ which may be used as DSA private keys.

1. If Algorithm 5.15 is to be used in step 4.3 then select an arbitrary integer b, $160 \leq b \leq 512$; if Algorithm 5.16 is to be used then set $b \leftarrow 160$.
2. Generate a random (and secret) b-bit seed s.
3. Define the 160-bit string $t =$ `67452301 efcdab89 98badcfe 10325476 c3d2e1f0` (in hexadecimal).
4. For i from 1 to m do the following:
   - 4.1 (optional user input) Either select a b-bit string y_i, or set $y_i \leftarrow 0$.
   - 4.2 $z_i \leftarrow (s + y_i) \bmod 2^b$.
   - 4.3 $a_i \leftarrow G(t, z_i) \bmod q$. ($G$ is either that defined in Algorithm 5.15 or 5.16.)
   - 4.4 $s \leftarrow (1 + s + a_i) \bmod 2^b$.
5. Return$(a_1, a_2, \ldots, a_m)$.

5.13 Note (*optional user input*) Algorithm 5.12 permits a user to augment the seed s with random or pseudorandom strings derived from alternate sources. The user may desire to do this if she does not trust the quality or integrity of the random bit generator which may be built into a cryptographic module implementing the algorithm.

5.14 Algorithm FIPS 186 pseudorandom number generator for DSA per-message secrets

INPUT: an integer m and a 160-bit prime number q.
OUTPUT: m pseudorandom numbers $k_1, k_2, \ldots, k_m$ in the interval $[0, q-1]$ which may be used as the per-message secret numbers k in the DSA.

1. If Algorithm 5.15 is to be used in step 4.1 then select an integer b, $160 \leq b \leq 512$; if Algorithm 5.16 is to be used then set $b \leftarrow 160$.
2. Generate a random (and secret) b-bit seed s.
3. Define the 160-bit string $t =$ `efcdab89 98badcfe 10325476 c3d2e1f0 67452301` (in hexadecimal).
4. For i from 1 to m do the following:
   - 4.1 $k_i \leftarrow G(t, s) \bmod q$. ($G$ is either that defined in Algorithm 5.15 or 5.16.)
   - 4.2 $s \leftarrow (1 + s + k_i) \bmod 2^b$.
5. Return$(k_1, k_2, \ldots, k_m)$.

5.15 Algorithm FIPS 186 one-way function using SHA-1

INPUT: a 160-bit string t and a b-bit string c, $160 \le b \le 512$.
OUTPUT: a 160-bit string denoted $G(t,c)$.

1. Break up t into five 32-bit blocks: $t = H_1\|H_2\|H_3\|H_4\|H_5$.
2. Pad c with 0's to obtain a 512-bit message block: $X \leftarrow c\|0^{512-b}$.
3. Divide X into 16 32-bit words: $x_0x_1 \ldots x_{15}$, and set $m \leftarrow 1$.
4. Execute step 4 of SHA-1 (Algorithm 9.53). (This alters the H_i's.)
5. The output is the concatenation: $G(t,c) = H_1\|H_2\|H_3\|H_4\|H_5$.

5.16 Algorithm FIPS 186 one-way function using DES

INPUT: two 160-bit strings t and c.
OUTPUT: a 160-bit string denoted $G(t,c)$.

1. Break up t into five 32-bit blocks: $t = t_0\|t_1\|t_2\|t_3\|t_4$.
2. Break up c into five 32-bit blocks: $c = c_0\|c_1\|c_2\|c_3\|c_4$.
3. For i from 0 to 4 do the following: $x_i \leftarrow t_i \oplus c_i$.
4. For i from 0 to 4 do the following:
   - 4.1 $b_1 \leftarrow c_{(i+4) \bmod 5}$, $b_2 \leftarrow c_{(i+3) \bmod 5}$.
   - 4.2 $a_1 \leftarrow x_i$, $a_2 \leftarrow x_{(i+1) \bmod 5} \oplus x_{(i+4) \bmod 5}$.
   - 4.3 $A \leftarrow a_1\|a_2$, $B \leftarrow b_1'\|b_2$, where b_1' denotes the 24 least significant bits of b_1.
   - 4.4 Use DES with key B to encrypt A: $y_i \leftarrow \mathrm{DES}_B(A)$.
   - 4.5 Break up y_i into two 32-bit blocks: $y_i = L_i\|R_i$.
5. For i from 0 to 4 do the following: $z_i \leftarrow L_i \oplus R_{(i+2) \bmod 5} \oplus L_{(i+3) \bmod 5}$.
6. The output is the concatenation: $G(t,c) = z_0\|z_1\|z_2\|z_3\|z_4$.

5.4 Statistical tests

This section presents some tests designed to measure the quality of a generator purported to be a random bit generator (Definition 5.1). While it is impossible to give a mathematical proof that a generator is indeed a random bit generator, the tests described here help detect certain kinds of weaknesses the generator may have. This is accomplished by taking a sample output sequence of the generator and subjecting it to various statistical tests. Each statistical test determines whether the sequence possesses a certain attribute that a truly random sequence would be likely to exhibit; the conclusion of each test is not definite, but rather *probabilistic*. An example of such an attribute is that the sequence should have roughly the same number of 0's as 1's. If the sequence is deemed to have failed any one of the statistical tests, the generator may be *rejected* as being non-random; alternatively, the generator may be subjected to further testing. On the other hand, if the sequence passes all of the statistical tests, the generator is *accepted* as being random. More precisely, the term "accepted" should be replaced by "not rejected", since passing the tests merely provides probabilistic evidence that the generator produces sequences which have certain characteristics of random sequences.

§5.4.1 and §5.4.2 provide some relevant background in statistics. §5.4.3 establishes some notation and lists Golomb's randomness postulates. Specific statistical tests for randomness are described in §5.4.4 and §5.4.5.

5.4.1 The normal and chi-square distributions

The normal and χ^2 distributions are widely used in statistical applications.

5.17 Definition If the result X of an experiment can be any real number, then X is said to be a *continuous* random variable.

5.18 Definition A *probability density function* of a continuous random variable X is a function $f(x)$ which can be integrated and satisfies:

(i) $f(x) \geq 0$ for all $x \in \mathbb{R}$;
(ii) $\int_{-\infty}^{\infty} f(x)\,dx = 1$; and
(iii) for all $a, b \in \mathbb{R}$, $P(a < X \leq b) = \int_a^b f(x)\,dx$.

(i) The normal distribution

The normal distribution arises in practice when a large number of independent random variables having the same mean and variance are summed.

5.19 Definition A (continuous) random variable X has a *normal distribution* with *mean* μ and *variance* σ^2 if its probability density function is defined by

$$f(x) \;=\; \frac{1}{\sigma\sqrt{2\pi}} \exp\left\{\frac{-(x-\mu)^2}{2\sigma^2}\right\}, \quad -\infty < x < \infty.$$

Notation: X is said to be $N(\mu, \sigma^2)$. If X is $N(0,1)$, then X is said to have a *standard normal distribution*.

A graph of the $N(0,1)$ distribution is given in Figure 5.1. The graph is symmetric

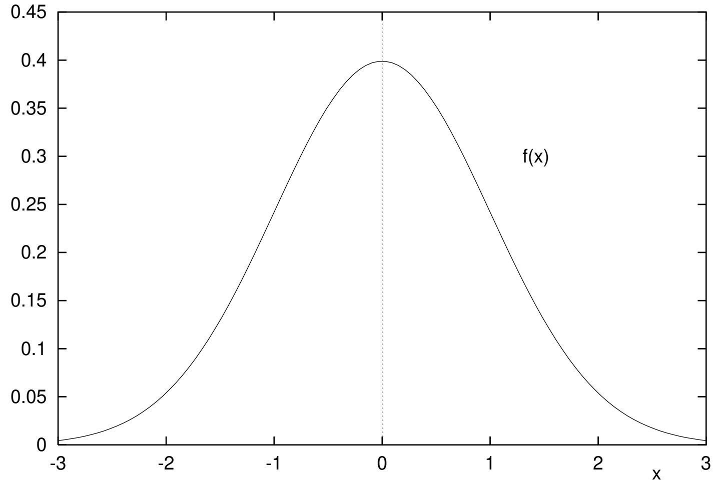

Figure 5.1: *The normal distribution* $N(0,1)$.

about the vertical axis, and hence $P(X > x) = P(X < -x)$ for any x. Table 5.1 gives some percentiles for the standard normal distribution. For example, the entry ($\alpha = 0.05$, $x = 1.6449$) means that if X is $N(0,1)$, then X exceeds 1.6449 about 5% of the time.

Fact 5.20 can be used to reduce questions about a normal distribution to questions about the standard normal distribution.

α	0.1	0.05	0.025	0.01	0.005	0.0025	0.001	0.0005
x	1.2816	1.6449	1.9600	2.3263	2.5758	2.8070	3.0902	3.2905

Table 5.1: *Selected percentiles of the standard normal distribution. If X is a random variable having a standard normal distribution, then $P(X > x) = \alpha$.*

5.20 Fact If the random variable X is $N(\mu, \sigma^2)$, then the random variable $Z = (X - \mu)/\sigma$ is $N(0, 1)$.

(ii) The χ^2 distribution

The χ^2 distribution can be used to compare the *goodness-of-fit* of the observed frequencies of events to their expected frequencies under a hypothesized distribution. The χ^2 distribution with v degrees of freedom arises in practice when the squares of v independent random variables having standard normal distributions are summed.

5.21 Definition Let $v \geq 1$ be an integer. A (continuous) random variable X has a χ^2 *(chi-square) distribution* with v *degrees of freedom* if its probability density function is defined by

$$f(x) = \begin{cases} \dfrac{1}{\Gamma(v/2)2^{v/2}} x^{(v/2)-1} e^{-x/2}, & 0 \leq x < \infty, \\ 0, & x < 0, \end{cases}$$

where Γ is the gamma function.[3] The *mean* and *variance* of this distribution are $\mu = v$, and $\sigma^2 = 2v$.

A graph of the χ^2 distribution with $v = 7$ degrees of freedom is given in Figure 5.2. Table 5.2 gives some percentiles of the χ^2 distribution for various degrees of freedom. For

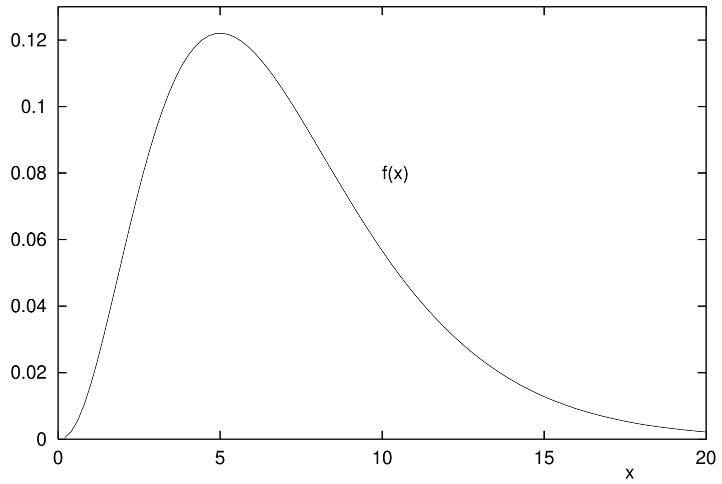

Figure 5.2: *The χ^2 (chi-square) distribution with $v = 7$ degrees of freedom.*

example, the entry in row $v = 5$ and column $\alpha = 0.05$ is $x = 11.0705$; this means that if X has a χ^2 distribution with 5 degrees of freedom, then X exceeds 11.0705 about 5% of the time.

[3] The *gamma function* is defined by $\Gamma(t) = \int_0^\infty x^{t-1} e^{-x} dx$, for $t > 0$.

	α					
v	0.100	0.050	0.025	0.010	0.005	0.001
1	2.7055	3.8415	5.0239	6.6349	7.8794	10.8276
2	4.6052	5.9915	7.3778	9.2103	10.5966	13.8155
3	6.2514	7.8147	9.3484	11.3449	12.8382	16.2662
4	7.7794	9.4877	11.1433	13.2767	14.8603	18.4668
5	9.2364	11.0705	12.8325	15.0863	16.7496	20.5150
6	10.6446	12.5916	14.4494	16.8119	18.5476	22.4577
7	12.0170	14.0671	16.0128	18.4753	20.2777	24.3219
8	13.3616	15.5073	17.5345	20.0902	21.9550	26.1245
9	14.6837	16.9190	19.0228	21.6660	23.5894	27.8772
10	15.9872	18.3070	20.4832	23.2093	25.1882	29.5883
11	17.2750	19.6751	21.9200	24.7250	26.7568	31.2641
12	18.5493	21.0261	23.3367	26.2170	28.2995	32.9095
13	19.8119	22.3620	24.7356	27.6882	29.8195	34.5282
14	21.0641	23.6848	26.1189	29.1412	31.3193	36.1233
15	22.3071	24.9958	27.4884	30.5779	32.8013	37.6973
16	23.5418	26.2962	28.8454	31.9999	34.2672	39.2524
17	24.7690	27.5871	30.1910	33.4087	35.7185	40.7902
18	25.9894	28.8693	31.5264	34.8053	37.1565	42.3124
19	27.2036	30.1435	32.8523	36.1909	38.5823	43.8202
20	28.4120	31.4104	34.1696	37.5662	39.9968	45.3147
21	29.6151	32.6706	35.4789	38.9322	41.4011	46.7970
22	30.8133	33.9244	36.7807	40.2894	42.7957	48.2679
23	32.0069	35.1725	38.0756	41.6384	44.1813	49.7282
24	33.1962	36.4150	39.3641	42.9798	45.5585	51.1786
25	34.3816	37.6525	40.6465	44.3141	46.9279	52.6197
26	35.5632	38.8851	41.9232	45.6417	48.2899	54.0520
27	36.7412	40.1133	43.1945	46.9629	49.6449	55.4760
28	37.9159	41.3371	44.4608	48.2782	50.9934	56.8923
29	39.0875	42.5570	45.7223	49.5879	52.3356	58.3012
30	40.2560	43.7730	46.9792	50.8922	53.6720	59.7031
31	41.4217	44.9853	48.2319	52.1914	55.0027	61.0983
63	77.7454	82.5287	86.8296	92.0100	95.6493	103.4424
127	147.8048	154.3015	160.0858	166.9874	171.7961	181.9930
255	284.3359	293.2478	301.1250	310.4574	316.9194	330.5197
511	552.3739	564.6961	575.5298	588.2978	597.0978	615.5149
1023	1081.3794	1098.5208	1113.5334	1131.1587	1143.2653	1168.4972

Table 5.2: *Selected percentiles of the χ^2 (chi-square) distribution. A (v, α)-entry of x in the table has the following meaning: if X is a random variable having a χ^2 distribution with v degrees of freedom, then $P(X > x) = \alpha$.*

Fact 5.22 relates the normal distribution to the χ^2 distribution.

5.22 Fact If the random variable X is $N(\mu, \sigma^2)$, $\sigma^2 > 0$, then the random variable $Z = (X - \mu)^2/\sigma^2$ has a χ^2 distribution with 1 degree of freedom. In particular, if X is $N(0,1)$, then $Z = X^2$ has a χ^2 distribution with 1 degree of freedom.

5.4.2 Hypothesis testing

A *statistical hypothesis*, denoted H_0, is an assertion about a distribution of one or more random variables. A *test* of a statistical hypothesis is a procedure, based upon observed values of the random variables, that leads to the acceptance or rejection of the hypothesis H_0. The test only provides a measure of the strength of the evidence provided by the data against the hypothesis; hence, the conclusion of the test is not definite, but rather probabilistic.

5.23 Definition The *significance level* α of the test of a statistical hypothesis H_0 is the probability of rejecting H_0 when it is true.

In this section, H_0 will be the hypothesis that a given binary sequence was produced by a random bit generator. If the significance level α of a test of H_0 is too high, then the test may reject sequences that were, in fact, produced by a random bit generator (such an error is called a *Type I error*). On the other hand, if the significance level of a test of H_0 is too low, then there is the danger that the test may accept sequences even though they were not produced by a random bit generator (such an error is called a *Type II error*).[4] It is, therefore, important that the test be carefully designed to have a significance level that is appropriate for the purpose at hand; a significance level α between 0.001 and 0.05 might be employed in practice.

A statistical test is implemented by specifying a *statistic* on the random sample.[5] Statistics are generally chosen so that they can be efficiently computed, and so that they (approximately) follow an $N(0,1)$ or a χ^2 distribution (see §5.4.1). The value of the statistic for the sample output sequence is computed and compared with the value expected for a random sequence as described below.

1. Suppose that a statistic X for a random sequence follows a χ^2 distribution with v degrees of freedom, and suppose that the statistic can be expected to take on larger values for nonrandom sequences. To achieve a significance level of α, a *threshold* value x_α is chosen (using Table 5.2) so that $P(X > x_\alpha) = \alpha$. If the value X_s of the statistic for the sample output sequence satisfies $X_s > x_\alpha$, then the sequence *fails* the test; otherwise, it *passes* the test. Such a test is called a *one-sided* test. For example, if $v = 5$ and $\alpha = 0.025$, then $x_\alpha = 12.8325$, and one expects a random sequence to fail the test only 2.5% of the time.
2. Suppose that a statistic X for a random sequence follows an $N(0,1)$ distribution, and suppose that the statistic can be expected to take on both larger and smaller values for nonrandom sequences. To achieve a significance level of α, a *threshold* value x_α is chosen (using Table 5.1) so that $P(X > x_\alpha) = P(X < -x_\alpha) = \alpha/2$. If the value

[4] Actually, the probability β of a Type II error may be completely independent of α. If the generator is not a random bit generator, the probability β depends on the nature of the defects of the generator, and is usually difficult to determine in practice. For this reason, assuming that the probability of a Type II error is proportional to α is a useful intuitive guide when selecting an appropriate significance level for a test.

[5] A *statistic* is a function of the elements of a random sample; for example, the number of 0's in a binary sequence is a statistic.

X_s of the statistic for the sample output sequence satisfies $X_s > x_\alpha$ or $X_s < -x_\alpha$, then the sequence *fails* the test; otherwise, it *passes* the test. Such a test is called a *two-sided* test. For example, if $\alpha = 0.05$, then $x_\alpha = 1.96$, and one expects a random sequence to fail the test only 5% of the time.

5.4.3 Golomb's randomness postulates

Golomb's randomness postulates (Definition 5.28) are presented here for historical reasons – they were one of the first attempts to establish some *necessary* conditions for a periodic pseudorandom sequence to look random. It is emphasized that these conditions are far from being *sufficient* for such sequences to be considered random. Unless otherwise stated, all sequences are binary sequences.

5.24 Definition Let $s = s_0, s_1, s_2, \ldots$ be an infinite sequence. The subsequence consisting of the first n terms of s is denoted by $s^n = s_0, s_1, \ldots, s_{n-1}$.

5.25 Definition The sequence $s = s_0, s_1, s_2, \ldots$ is said to be N-*periodic* if $s_i = s_{i+N}$ for all $i \geq 0$. The sequence s is *periodic* if it is N-periodic for some positive integer N. The *period* of a periodic sequence s is the smallest positive integer N for which s is N-periodic. If s is a periodic sequence of period N, then the *cycle* of s is the subsequence s^N.

5.26 Definition Let s be a sequence. A *run* of s is a subsequence of s consisting of consecutive 0's or consecutive 1's which is neither preceded nor succeeded by the same symbol. A run of 0's is called a *gap*, while a run of 1's is called a *block*.

5.27 Definition Let $s = s_0, s_1, s_2, \ldots$ be a periodic sequence of period N. The *autocorrelation function* of s is the integer-valued function $C(t)$ defined as

$$C(t) = \frac{1}{N}\sum_{i=0}^{N-1}(2s_i - 1)\cdot(2s_{i+t} - 1), \quad \text{for } 0 \leq t \leq N-1.$$

The autocorrelation function $C(t)$ measures the amount of similarity between the sequence s and a shift of s by t positions. If s is a random periodic sequence of period N, then $|N \cdot C(t)|$ can be expected to be quite small for all values of t, $0 < t < N$.

5.28 Definition Let s be a periodic sequence of period N. *Golomb's randomness postulates* are the following.

R1: In the cycle s^N of s, the number of 1's differs from the number of 0's by at most 1.

R2: In the cycle s^N, at least half the runs have length 1, at least one-fourth have length 2, at least one-eighth have length 3, etc., as long as the number of runs so indicated exceeds 1. Moreover, for each of these lengths, there are (almost) equally many gaps and blocks.[6]

R3: The autocorrelation function $C(t)$ is two-valued. That is for some integer K,

$$N \cdot C(t) = \sum_{i=0}^{N-1}(2s_i - 1)\cdot(2s_{i+t} - 1) = \begin{cases} N, & \text{if } t = 0, \\ K, & \text{if } 1 \leq t \leq N-1. \end{cases}$$

[6] Postulate R2 implies postulate R1.

5.29 Definition A binary sequence which satisfies Golomb's randomness postulates is called a *pseudo-noise sequence* or a *pn-sequence*.

Pseudo-noise sequences arise in practice as output sequences of maximum-length linear feedback shift registers (cf. Fact 6.14).

5.30 Example (*pn-sequence*) Consider the periodic sequence s of period $N = 15$ with cycle

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

The following shows that the sequence s satisfies Golomb's randomness postulates.

R1: The number of 0's in s^{15} is 7, while the number of 1's is 8.

R2: s^{15} has 8 runs. There are 4 runs of length 1 (2 gaps and 2 blocks), 2 runs of length 2 (1 gap and 1 block), 1 run of length 3 (1 gap), and 1 run of length 4 (1 block).

R3: The autocorrelation function $C(t)$ takes on two values: $C(0) = 1$ and $C(t) = \frac{-1}{15}$ for $1 \leq t \leq 14$.

Hence, s is a pn-sequence. □

5.4.4 Five basic tests

Let $s = s_0, s_1, s_2, \ldots, s_{n-1}$ be a binary sequence of length n. This subsection presents five statistical tests that are commonly used for determining whether the binary sequence s possesses some specific characteristics that a truly random sequence would be likely to exhibit. It is emphasized again that the outcome of each test is not definite, but rather probabilistic. If a sequence passes all five tests, there is no guarantee that it was indeed produced by a random bit generator (cf. Example 5.4).

(i) Frequency test (monobit test)

The purpose of this test is to determine whether the number of 0's and 1's in s are approximately the same, as would be expected for a random sequence. Let n_0, n_1 denote the number of 0's and 1's in s, respectively. The statistic used is

$$X_1 = \frac{(n_0 - n_1)^2}{n} \tag{5.1}$$

which approximately follows a χ^2 distribution with 1 degree of freedom if $n \geq 10$. [7]

(ii) Serial test (two-bit test)

The purpose of this test is to determine whether the number of occurrences of 00, 01, 10, and 11 as subsequences of s are approximately the same, as would be expected for a random sequence. Let n_0, n_1 denote the number of 0's and 1's in s, respectively, and let n_{00}, n_{01}, n_{10}, n_{11} denote the number of occurrences of 00, 01, 10, 11 in s, respectively. Note that $n_{00} + n_{01} + n_{10} + n_{11} = (n - 1)$ since the subsequences are allowed to overlap. The statistic used is

$$X_2 = \frac{4}{n-1}\left(n_{00}^2 + n_{01}^2 + n_{10}^2 + n_{11}^2\right) - \frac{2}{n}\left(n_0^2 + n_1^2\right) + 1 \tag{5.2}$$

which approximately follows a χ^2 distribution with 2 degrees of freedom if $n \geq 21$.

[7] In practice, it is recommended that the length n of the sample output sequence be much larger (for example, $n \gg 10000$) than the minimum specified for each test in this subsection.

(iii) Poker test

Let m be a positive integer such that $\lfloor \frac{n}{m} \rfloor \geq 5 \cdot (2^m)$, and let $k = \lfloor \frac{n}{m} \rfloor$. Divide the sequence s into k non-overlapping parts each of length m, and let n_i be the number of occurrences of the i^{th} type of sequence of length m, $1 \leq i \leq 2^m$. The poker test determines whether the sequences of length m each appear approximately the same number of times in s, as would be expected for a random sequence. The statistic used is

$$X_3 \;=\; \frac{2^m}{k}\left(\sum_{i=1}^{2^m} n_i^2\right) - k \tag{5.3}$$

which approximately follows a χ^2 distribution with $2^m - 1$ degrees of freedom. Note that the poker test is a generalization of the frequency test: setting $m = 1$ in the poker test yields the frequency test.

(iv) Runs test

The purpose of the runs test is to determine whether the number of runs (of either zeros or ones; see Definition 5.26) of various lengths in the sequence s is as expected for a random sequence. The expected number of gaps (or blocks) of length i in a random sequence of length n is $e_i = (n-i+3)/2^{i+2}$. Let k be equal to the largest integer i for which $e_i \geq 5$. Let B_i, G_i be the number of blocks and gaps, respectively, of length i in s for each i, $1 \leq i \leq k$. The statistic used is

$$X_4 \;=\; \sum_{i=1}^{k} \frac{(B_i - e_i)^2}{e_i} + \sum_{i=1}^{k} \frac{(G_i - e_i)^2}{e_i} \tag{5.4}$$

which approximately follows a χ^2 distribution with $2k - 2$ degrees of freedom.

(v) Autocorrelation test

The purpose of this test is to check for correlations between the sequence s and (non-cyclic) shifted versions of it. Let d be a fixed integer, $1 \leq d \leq \lfloor n/2 \rfloor$. The number of bits in s not equal to their d-shifts is $A(d) = \sum_{i=0}^{n-d-1} s_i \oplus s_{i+d}$, where $\oplus$ denotes the XOR operator. The statistic used is

$$X_5 \;=\; 2\left(A(d) - \frac{n-d}{2}\right)/\sqrt{n-d} \tag{5.5}$$

which approximately follows an $N(0,1)$ distribution if $n - d \geq 10$. Since small values of $A(d)$ are as unexpected as large values of $A(d)$, a two-sided test should be used.

5.31 Example (*basic statistical tests*) Consider the (non-random) sequence s of length $n = 160$ obtained by replicating the following sequence four times:

$$11100\ 01100\ 01000\ 10100\ 11101\ 11100\ 10010\ 01001.$$

(i) (*frequency test*) $n_0 = 84$, $n_1 = 76$, and the value of the statistic X_1 is 0.4.

(ii) (*serial test*) $n_{00} = 44$, $n_{01} = 40$, $n_{10} = 40$, $n_{11} = 35$, and the value of the statistic X_2 is 0.6252.

(iii) (*poker test*) Here $m = 3$ and $k = 53$. The blocks 000, 001, 010, 011, 100, 101, 110, 111 appear 5, 10, 6, 4, 12, 3, 6, and 7 times, respectively, and the value of the statistic X_3 is 9.6415.

(iv) (*runs test*) Here $e_1 = 20.25$, $e_2 = 10.0625$, $e_3 = 5$, and $k = 3$. There are 25, 4, 5 blocks of lengths 1, 2, 3, respectively, and 8, 20, 12 gaps of lengths 1, 2, 3, respectively. The value of the statistic X_4 is 31.7913.

(v) (*autocorrelation test*) If $d = 8$, then $A(8) = 100$. The value of the statistic X_5 is 3.8933.

For a significance level of $\alpha = 0.05$, the threshold values for X_1, X_2, X_3, X_4, and X_5 are 3.8415, 5.9915, 14.0671, 9.4877, and 1.96, respectively (see Tables 5.1 and 5.2). Hence, the given sequence s passes the frequency, serial, and poker tests, but fails the runs and autocorrelation tests. □

5.32 Note (*FIPS 140-1 statistical tests for randomness*) FIPS 140-1 specifies four statistical tests for randomness. Instead of making the user select appropriate significance levels for these tests, explicit bounds are provided that the computed value of a statistic must satisfy. A single bitstring s of length 20000 bits, output from a generator, is subjected to each of the following tests. If any of the tests fail, then the generator fails the test.

(i) *monobit test*. The number n_1 of 1's in s should satisfy $9654 < n_1 < 10346$.

(ii) *poker test*. The statistic X_3 defined by equation (5.3) is computed for $m = 4$. The poker test is passed if $1.03 < X_3 < 57.4$.

(iii) *runs test*. The number B_i and G_i of blocks and gaps, respectively, of length i in s are counted for each i, $1 \le i \le 6$. (For the purpose of this test, runs of length greater than 6 are considered to be of length 6.) The runs test is passed if the 12 counts B_i, G_i, $1 \le i \le 6$, are each within the corresponding interval specified by the following table.

Length of run	Required interval
1	$2267 - 2733$
2	$1079 - 1421$
3	$502 - 748$
4	$223 - 402$
5	$90 - 223$
6	$90 - 223$

(iv) *long run test*. The long run test is passed if there are no runs of length 34 or more.

For high security applications, FIPS 140-1 mandates that the four tests be performed each time the random bit generator is powered up. FIPS 140-1 allows these tests to be substituted by alternative tests which provide equivalent or superior randomness checking.

5.4.5 Maurer's universal statistical test

The basic idea behind Maurer's universal statistical test is that it should not be possible to significantly compress (without loss of information) the output sequence of a random bit generator. Thus, if a sample output sequence s of a bit generator can be significantly compressed, the generator should be rejected as being defective. Instead of actually compressing the sequence s, the universal statistical test computes a quantity that is related to the length of the compressed sequence.

The *universality* of Maurer's universal statistical test arises because it is able to detect any one of a very general class of possible defects a bit generator might have. This class includes the five defects that are detectable by the basic tests of §5.4.4. A drawback of the universal statistical test over the five basic tests is that it requires a much longer sample output sequence in order to be effective. Provided that the required output sequence can be efficiently generated, this drawback is not a practical concern since the universal statistical test itself is very efficient.

Algorithm 5.33 computes the statistic X_u for a sample output sequence $s = s_0, s_1, \dots, s_{n-1}$ to be used in the universal statistical test. The parameter L is first chosen from the

L	μ	σ_1^2
1	0.7326495	0.690
2	1.5374383	1.338
3	2.4016068	1.901
4	3.3112247	2.358
5	4.2534266	2.705
6	5.2177052	2.954
7	6.1962507	3.125
8	7.1836656	3.238

L	μ	σ_1^2
9	8.1764248	3.311
10	9.1723243	3.356
11	10.170032	3.384
12	11.168765	3.401
13	12.168070	3.410
14	13.167693	3.416
15	14.167488	3.419
16	15.167379	3.421

Table 5.3: *Mean μ and variance σ^2 of the statistic X_u for random sequences, with parameters L, K as $Q \to \infty$. The variance of X_u is $\sigma^2 = c(L,K)^2 \cdot \sigma_1^2/K$, where $c(L,K) \approx 0.7 - (0.8/L) + (1.6 + (12.8/L)) \cdot K^{-4/L}$ for $K \geq 2^L$.*

interval $[6, 16]$. The sequence s is then partitioned into non-overlapping L-bit blocks, with any leftover bits discarded; the total number of blocks is $Q+K$, where Q and K are defined below. For each i, $1 \leq i \leq Q+K$, let b_i be the integer whose binary representation is the i^{th} block. The blocks are scanned in order. A table T is maintained so that at each stage $T[j]$ is the position of the last occurrence of the block corresponding to integer j, $0 \leq j \leq 2^L - 1$. The first Q blocks of s are used to initialize table T; Q should be chosen to be at least $10 \cdot 2^L$ in order to have a high likelihood that each of the 2^L L-bit blocks occurs at least once in the first Q blocks. The remaining K blocks are used to define the statistic X_u as follows. For each i, $Q+1 \leq i \leq Q+K$, let $A_i = i - T[b_i]$; A_i is the number of positions since the last occurrence of block b_i. Then

$$X_u = \frac{1}{K} \sum_{i=Q+1}^{Q+K} \lg A_i. \tag{5.6}$$

K should be at least $1000 \cdot 2^L$ (and, hence, the sample sequence s should be at least $(1010 \cdot 2^L \cdot L)$ bits in length). Table 5.3 lists the mean μ and variance σ^2 of X_u for random sequences for some sample choices of L as $Q \to \infty$.

5.33 Algorithm Computing the statistic X_u for Maurer's universal statistical test

INPUT: a binary sequence $s = s_0, s_1, \ldots, s_{n-1}$ of length n, and parameters L, Q, K.
OUTPUT: the value of the statistic X_u for the sequence s.

1. *Zero the table T.* For j from 0 to $2^L - 1$ do the following: $T[j] \leftarrow 0$.
2. *Initialize the table T.* For i from 1 to Q do the following: $T[b_i] \leftarrow i$.
3. sum$\leftarrow$0.
4. For i from $Q+1$ to $Q+K$ do the following:
   - 4.1 sum$\leftarrow$sum $+ \lg(i - T[b_i])$.
   - 4.2 $T[b_i] \leftarrow i$.
5. $X_u \leftarrow$sum$/K$.
6. Return(X_u).

Maurer's universal statistical test uses the computed value of X_u for the sample output sequence s in the manner prescribed by Fact 5.34. To test the sequence s, a two-sided test should be used with a significance level α between 0.001 and 0.01 (see §5.4.2).

5.34 Fact Let X_u be the statistic defined in (5.6) having mean μ and variance σ^2 as given in Table 5.3. Then, for random sequences, the statistic $Z_u = (X_u - \mu)/\sigma$ approximately follows an $N(0,1)$ distribution.

5.5 Cryptographically secure pseudorandom bit generation

Two cryptographically secure pseudorandom bit generators (CSPRBG – see Definition 5.8) are presented in this section. The security of each generator relies on the presumed intractability of an underlying number-theoretic problem. The modular multiplications that these generators use make them relatively slow compared to the (ad-hoc) pseudorandom bit generators of §5.3. Nevertheless they may be useful in some circumstances, for example, generating pseudorandom bits on hardware devices which already have the circuitry for performing modular multiplications. Efficient techniques for implementing modular multiplication are presented in §14.3.

5.5.1 RSA pseudorandom bit generator

The RSA pseudorandom bit generator is a CSPRBG under the assumption that the RSA problem is intractable (§3.3; see also §3.9.2).

5.35 Algorithm RSA pseudorandom bit generator

SUMMARY: a pseudorandom bit sequence $z_1, z_2, \ldots, z_l$ of length l is generated.

1. *Setup*. Generate two secret RSA-like primes p and q (cf. Note 8.8), and compute $n = pq$ and $\phi = (p-1)(q-1)$. Select a random integer e, $1 < e < \phi$, such that $\gcd(e, \phi) = 1$.
2. Select a random integer x_0 (the *seed*) in the interval $[1, n-1]$.
3. For i from 1 to l do the following:
   3.1 $x_i \leftarrow x_{i-1}^e \bmod n$.
   3.2 $z_i \leftarrow$ the least significant bit of x_i.
4. The output sequence is $z_1, z_2, \ldots, z_l$.

5.36 Note (*efficiency of the RSA PRBG*) If $e = 3$ is chosen (cf. Note 8.9(ii)), then generating each pseudorandom bit z_i requires one modular multiplication and one modular squaring. The efficiency of the generator can be improved by extracting the j least significant bits of x_i in step 3.2, where $j = c \lg \lg n$ and c is a constant. Provided that n is sufficiently large, this modified generator is also cryptographically secure (cf. Fact 3.87). For a modulus n of a fixed bitlength (e.g., 1024 bits), an explicit range of values of c for which the resulting generator remains cryptographically secure (cf. Remark 5.9) under the intractability assumption of the RSA problem has not been determined.

The following modification improves the efficiency of the RSA PRBG.

5.37 Algorithm Micali-Schnorr pseudorandom bit generator

SUMMARY: a pseudorandom bit sequence is generated.

1. *Setup*. Generate two secret RSA-like primes p and q (cf. Note 8.8), and compute $n = pq$ and $\phi = (p-1)(q-1)$. Let $N = \lfloor \lg n \rfloor + 1$ (the bitlength of n). Select an integer e, $1 < e < \phi$, such that $\gcd(e, \phi) = 1$ and $80e \leq N$. Let $k = \lfloor N(1 - \frac{2}{e}) \rfloor$ and $r = N - k$.
2. Select a random sequence x_0 (the *seed*) of bitlength r.
3. *Generate a pseudorandom sequence of length* $k \cdot l$. For i from 1 to l do the following:
   3.1 $y_i \leftarrow x_{i-1}^e \bmod n$.
   3.2 $x_i \leftarrow$ the r most significant bits of y_i.
   3.3 $z_i \leftarrow$ the k least significant bits of y_i.
4. The output sequence is $z_1 \parallel z_2 \parallel \cdots \parallel z_l$, where $\parallel$ denotes concatenation.

5.38 Note (*efficiency of the Micali-Schnorr PRBG*) Algorithm 5.37 is more efficient than the RSA PRBG since $\lfloor N(1 - \frac{2}{e}) \rfloor$ bits are generated per exponentiation by e. For example, if $e = 3$ and $N = 1024$, then $k = 341$ bits are generated per exponentiation. Moreover, each exponentiation requires only one modular squaring of an $r = 683$-bit number, and one modular multiplication.

5.39 Note (*security of the Micali-Schnorr PRBG*) Algorithm 5.37 is cryptographically secure under the assumption that the following is true: the distribution $x^e \bmod n$ for random r-bit sequences x is indistinguishable by all polynomial-time statistical tests from the uniform distribution of integers in the interval $[0, n-1]$. This assumption is stronger than requiring that the RSA problem be intractable.

5.5.2 Blum-Blum-Shub pseudorandom bit generator

The Blum-Blum-Shub pseudorandom bit generator (also known as the $x^2 \bmod n$ generator or the *BBS* generator) is a CSPRBG under the assumption that integer factorization is intractable (§3.2). It forms the basis for the Blum-Goldwasser probabilistic public-key encryption scheme (Algorithm 8.56).

5.40 Algorithm Blum-Blum-Shub pseudorandom bit generator

SUMMARY: a pseudorandom bit sequence $z_1, z_2, \ldots, z_l$ of length l is generated.

1. *Setup*. Generate two large secret random (and distinct) primes p and q (cf. Note 8.8), each congruent to 3 modulo 4, and compute $n = pq$.
2. Select a random integer s (the *seed*) in the interval $[1, n-1]$ such that $\gcd(s, n) = 1$, and compute $x_0 \leftarrow s^2 \bmod n$.
3. For i from 1 to l do the following:
   3.1 $x_i \leftarrow x_{i-1}^2 \bmod n$.
   3.2 $z_i \leftarrow$ the least significant bit of x_i.
4. The output sequence is $z_1, z_2, \ldots, z_l$.

5.41 Note (*efficiency of the Blum-Blum-Shub PRBG*) Generating each pseudorandom bit z_i requires one modular squaring. The efficiency of the generator can be improved by extracting the j least significant bits of x_i in step 3.2, where $j = c \lg\lg n$ and c is a constant. Provided that n is sufficiently large, this modified generator is also cryptographically secure. For a modulus n of a fixed bitlength (eg. 1024 bits), an explicit range of values of c for which the resulting generator is cryptographically secure (cf. Remark 5.9) under the intractability assumption of the integer factorization problem has not been determined.

5.6 Notes and further references

§5.1

Chapter 3 of Knuth [692] is the definitive reference for the classic (non-cryptographic) generation of pseudorandom numbers. Knuth [692, pp.142-166] contains an extensive discussion of what it means for a sequence to be random. Lagarias [724] gives a survey of theoretical results on pseudorandom number generators. Luby [774] provides a comprehensive and rigorous overview of pseudorandom generators.

For a study of linear congruential generators (Example 5.4), see Knuth [692, pp.9-25]. Plumstead/Boyar [979, 980] showed how to predict the output of a linear congruential generator given only a few elements of the output sequence, and when the parameters a, b, and m of the generator are unknown. Boyar [180] extended her method and showed that linear *multivariate* congruential generators (having recurrence equation $x_n = a_1x_{n-1} + a_2x_{n-2} + \cdots + a_lx_{n-l} + b \bmod m$), and *quadratic* congruential generators (having recurrence equation $x_n = ax_{n-1}^2 + bx_{n-1} + c \bmod m$) are cryptographically insecure. Finally, Krawczyk [713] generalized these results and showed how the output of any *multivariate polynomial* congruential generator can be efficiently predicted. A *truncated* linear congruential generator is one where a fraction of the least significant bits of the x_i are discarded. Frieze et al. [427] showed that these generators can be efficiently predicted if the generator parameters a, b, and m are known. Stern [1173] extended this method to the case where only m is known. Boyar [179] presented an efficient algorithm for predicting linear congruential generators when $O(\log\log m)$ bits are discarded, and when the parameters a, b, and m are unknown. No efficient prediction algorithms are known for truncated multivariate polynomial congruential generators. For a summary of cryptanalytic attacks on congruential generators, see Brickell and Odlyzko [209, pp.523-526].

For a formal definition of a statistical test (Definition 5.5), see Yao [1258]. Fact 5.7 on the universality of the next-bit test is due to Yao [1258]. For a proof of Yao's result, see Kranakis [710] and §12.2 of Stinson [1178]. A proof of a generalization of Yao's result is given by Goldreich, Goldwasser, and Micali [468]. The notion of a cryptographically secure pseudorandom bit generator (Definition 5.8) was introduced by Blum and Micali [166]. Blum and Micali also gave a formal description of the next-bit test (Definition 5.6), and presented the first cryptographically secure pseudorandom bit generator whose security is based on the discrete logarithm problem (see page 189). Universal tests were presented by Schrift and Shamir [1103] for verifying the assumed properties of a pseudorandom generator whose output sequences are not necessarily uniformly distributed.

The first provably secure pseudorandom *number* generator was proposed by Shamir [1112]. Shamir proved that predicting the next number of an output sequence of this generator is equivalent to inverting the RSA function. However, even though the numbers as a whole may be unpredictable, certain parts of the number (for example, its least significant bit) may be biased or predictable. Hence, Shamir's generator is not cryptographically secure in the sense of Definition 5.8.

§5.2

Agnew [17] proposed a VLSI implementation of a random bit generator consisting of two identical metal insulator semiconductor capacitors close to each other. The cells are charged over the same period of time, and then a 1 or 0 is assigned depending on which cell has a greater charge. Fairfield, Mortenson, and Coulthart [382] described an LSI random bit generator based on the frequency instability of a free running oscillator. Davis, Ihaka, and Fenstermacher [309] used the unpredictability of air turbulence occurring in a sealed disk drive as a random bit generator. The bits are extracted by measuring the variations in the time to access disk blocks. Fast Fourier Transform (FFT) techniques are then used to remove possible biases and correlations. A sample implementation generated 100 random bits per minute. For further guidance on hardware and software-based techniques for generating random bits, see RFC 1750 [1043].

The de-skewing technique of Example 5.10 is due to von Neumann [1223]. Elias [370] generalized von Neumann's technique to a more efficient scheme (one where fewer bits are discarded). Fast Fourier Transform techniques for removing biases and correlations are described by Brillinger [213]. For further ways of removing correlations, see Blum [161], Santha and Vazirani [1091], Vazirani [1217], and Chor and Goldreich [258].

§5.3

The idea of using a one-way function f for generating pseudorandom bit sequences is due to Shamir [1112]. Shamir illustrated why it is difficult to prove that such ad-hoc generators are cryptographically secure without imposing some further assumptions on f. Algorithm 5.11 is from Appendix C of the ANSI X9.17 standard [37]; it is one of the approved methods for pseudorandom bit generation listed in FIPS 186 [406]. Meyer and Matyas [859, pp.316-317] describe another DES-based pseudorandom bit generator whose output is intended for use as data-encrypting keys. The four algorithms of §5.3.2 for generating DSA parameters are from FIPS 186.

§5.4

Standard references on statistics include Hogg and Tanis [559] and Wackerly, Mendenhall, and Scheaffer [1226]. Tables 5.1 and 5.2 were generated using the Maple symbolic algebra system [240]. Golomb's randomness postulates (§5.4.3) were proposed by Golomb [498].

The five statistical tests for local randomness outlined in §5.4.4 are from Beker and Piper [84]. The serial test (§5.4.4(ii)) is due to Good [508]. It was generalized to subsequences of length greater than 2 by Marsaglia [782] who called it the *overlapping m-tuple test*, and later by Kimberley [674] who called it the *generalized serial test*. The underlying distribution theories of the serial test and the runs test (§5.4.4(iv)) were analyzed by Good [507] and Mood [897], respectively. Gustafson [531] considered alternative statistics for the runs test and the autocorrelation test (§5.4.4(v)).

There are numerous other statistical tests of local randomness. Many of these tests, including the gap test, coupon collector's test, permutation test, run test, maximum-of-t test, collision test, serial test, correlation test, and spectral test are described by Knuth [692]. The poker test as formulated by Knuth [692, p.62] is quite different from that of §5.4.4(iii). In the former, a sample sequence is divided into m-bit blocks, each of which is further subdivided into l-bit sub-blocks (for some divisor l of m). The number of m-bit blocks having r distinct l-bit sub-blocks ($1 \leq r \leq m/l$) is counted and compared to the corresponding expected numbers for random sequences. Erdmann [372] gives a detailed exposition of many of these tests, and applies them to sample output sequences of six pseudorandom bit generators. Gustafson et al. [533] describe a computer package which implements various statistical tests for assessing the strength of a pseudorandom bit generator. Gustafson, Dawson, and Golić [532] proposed a new *repetition test* which measures the number of repetitions of l-bit blocks. The test requires a count of the number of patterns repeated, but does not require the frequency of each pattern. For this reason, it is feasible to apply this test for larger values of l (e.g. $l = 64$) than would be permissible by the poker test or Maurer's universal statistical test (Algorithm 5.33). Two spectral tests have been developed, one based on the discrete Fourier transform by Gait [437], and one based on the Walsh transform by Yuen [1260]. For extensions of these spectral tests, see Erdmann [372] and Feldman [389].

FIPS 140-1 [401] specifies security requirements for the design and implementation of cryptographic modules, including random and pseudorandom bit generators, for protecting (U.S. government) unclassified information.

The universal statistical test (Algorithm 5.33) is due to Maurer [813] and was motivated by source coding algorithms of Elias [371] and Willems [1245]. The class of defects that the test is able to detect consists of those that can be modeled by an ergodic stationary source with limited memory; Maurer argues that this class includes the possible defects that could occur in a practical implementation of a random bit generator. Table 5.3 is due to Maurer [813], who provides derivations of formulae for the mean and variance of the statistic X_u.

§5.5

Blum and Micali [166] presented the following general construction for CSPRBGs. Let D be a finite set, and let $f\colon D \to D$ be a permutation that can be efficiently computed. Let $B\colon D \to \{0,1\}$ be a Boolean predicate with the property that $B(x)$ is hard to compute given only $x \in D$, however, $B(x)$ can be efficiently computed given $y = f^{-1}(x)$. The output sequence $z_1, z_2, \ldots, z_l$ corresponding to a seed $x_0 \in D$ is obtained by computing $x_i = f(x_{i-1})$, $z_i = B(x_i)$, for $1 \leq i \leq l$. This generator can be shown to pass the next-bit test (Definition 5.6). Blum and Micali [166] proposed the first concrete instance of a CSPRBG, called the *Blum-Micali generator*. Using the notation introduced above, their method can be described as follows. Let p be a large prime, and α a generator of $\mathbb{Z}_p^*$. Define $D = \mathbb{Z}_p^* = \{1, 2, \ldots, p-1\}$. The function $f : D \to D$ is defined by $f(x) = \alpha^x \bmod p$. The function $B : D \to \{0,1\}$ is defined by $B(x) = 1$ if $0 \leq \log_\alpha x \leq (p-1)/2$, and $B(x) = 0$ if $\log_\alpha x > (p-1)/2$. Assuming the intractability of the discrete logarithm problem in $\mathbb{Z}_p^*$ (§3.6; see also §3.9.1), the Blum-Micali generator was proven to satisfy the next-bit test. Long and Wigderson [772] improved the efficiency of the Blum-Micali generator by simultaneously extracting $O(\lg \lg p)$ bits (cf. §3.9.1) from each x_i. Kaliski [650, 651] modified the Blum-Micali generator so that the security depends on the discrete logarithm problem in the group of points on an elliptic curve defined over a finite field.

The RSA pseudorandom bit generator (Algorithm 5.35) and the improvement mentioned in Note 5.36 are due to Alexi et al. [23]. The Micali-Schnorr improvement of the RSA PRBG (Algorithm 5.37) is due to Micali and Schnorr [867], who also described a method that transforms any CSPRBG into one that can be accelerated by parallel evaluation. The method of parallelization is *perfect*: m parallel processors speed the generation of pseudorandom bits by a factor of m.

Algorithm 5.40 is due to Blum, Blum, and Shub [160], who showed that their pseudorandom bit generator is cryptographically secure assuming the intractability of the quadratic residuosity problem (§3.4). Vazirani and Vazirani [1218] established a stronger result regarding the security of this generator by proving it cryptographically secure under the weaker assumption that integer factorization is intractable. The improvement mentioned in

Note 5.41 is due to Vazirani and Vazirani. Alexi et al. [23] proved analogous results for the *modified-Rabin generator*, which differs as follows from the Blum-Blum-Shub generator: in step 3.1 of Algorithm 5.40, let $\overline{x} = x_{i-1}^2 \bmod n$; if $\overline{x} < n/2$, then $x_i = \overline{x}$; otherwise, $x_i = n - \overline{x}$.

Impagliazzo and Naor [569] devised efficient constructions for a CSPRBG and for a universal one-way hash function which are provably as secure as the subset sum problem. Fischer and Stern [411] presented a simple and efficient CSPRBG which is provably as secure as the *syndrome decoding problem*.

Yao [1258] showed how to obtain a CSPRBG using any one-way permutation. Levin [761] generalized this result and showed how to obtain a CSPRBG using any one-way function. For further refinements, see Goldreich, Krawczyk, and Luby [470], Impagliazzo, Levin, and Luby [568], and Håstad [545].

A *random function* $f\colon \{0,1\}^n \to \{0,1\}^n$ is a function which assigns independent and random values $f(x) \in \{0,1\}^n$ to all arguments $x \in \{0,1\}^n$. Goldreich, Goldwasser, and Micali [468] introduced a computational complexity measure of the randomness of functions. They defined a function to be *poly-random* if no polynomial-time algorithm can distinguish between values of the function and true random strings, even when the algorithm is permitted to select the arguments to the function. Goldreich, Goldwasser, and Micali presented an algorithm for constructing poly-random functions assuming the existence of one-way functions. This theory was applied by Goldreich, Goldwasser, and Micali [467] to develop provably secure protocols for the (essentially) storageless distribution of secret identification numbers, message authentication with timestamping, dynamic hashing, and identify friend or foe systems. Luby and Rackoff [776] showed how poly-random permutations can be efficiently constructed from poly-random functions. This result was used, together with some of the design principles of DES, to show how any CSPRBG can be used to construct a symmetric-key block cipher which is provably secure against chosen-plaintext attack. A simplified and generalized treatment of Luby and Rackoff's construction was given by Maurer [816].

Schnorr [1096] used Luby and Rackoff's poly-random permutation generator to construct a pseudorandom bit generator that was claimed to pass all statistical tests depending only on a small fraction of the output sequence, even when infinite computational resources are available. Rueppel [1079] showed that this claim is erroneous, and demonstrated that the generator can be distinguished from a truly random bit generator using only a small number of output bits. Maurer and Massey [821] extended Schnorr's work, and proved the existence of pseudorandom bit generators that pass all statistical tests depending only on a small fraction of the output sequence, even when infinite computational resources are available. The security of the generators does not rely on any unproved hypothesis, but rather on the assumption that the adversary can access only a limited number of bits of the generated sequence. This work is primarily of theoretical interest since no such polynomial-time generators are known.