Perbandingan Algoritma Linear Congruential Generators, BlumBlumShub, dan MersenneTwister untuk Membangkitkan Bilangan Acak Semu

Andresta Ramadhan
13503030
Program studi teknik informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung
if13030@students.if.itb.ac.id

Abstraksi
Random number atau bilangan acak adalah sebuah bilangan yang dihasilkan dari sebuah proses, yang keluarannya tidak dapat diprediksi dan secara berurutan tidak bisa dihasilkan bilangan yang sama. Proses yang menghasilkan random number disebut *random number generator*. Walaupun kelihatannya cukup sederhana, dari definisinya, tetapi pada kenyataanya cukup sulit untuk menghasilkan bilangan yang benar-benar acak. *Testing randomness* bertujuan untuk menentukan apakah uratan beberapa bilangan dihasilkan oleh *random generator* atau bukan.
Pada makalah ini akan dibahas mengenai bagaimana memproduksi bilangan random menggunakan perangkat lunak, menghasilkan *psedo random number*, untuk beberapa keperluan seperti games,kriptografi,statistik, serta penjelasan beberapa algoritma untuk menghasilkan random number.

kata kunci : *random number*, *pseudo random number*, *random generator*

Pendahuluan
Bilangan acak (random number) merupakan hal yang sangat penting dalam kriptografi. Kita memerlukan sebuah bilangan untuk melakukan enkripsi pada email, untuk menandatangani dokumen secara digital, pembayaran elektronik dan lainnya. Pada enkripsi kunci simetris dan asimetris bilangan acak digunakan untuk pembangkitan parameter kunci pada algoritma kunci-publik dan pembangkitan *initialization vector* (*IV*) pada algoritma kunci-simetri.

Sayangnya bilangan yang benar-benar acak sangatlah sulit untuk dihasilkan, terutama pada oleh komputer yang memang didesain deterministic. Hal ini yang menyebabkan bilangan acak yang dihasilkan oleh komputer disebut bilangan acak semu (pseudo random number).

Sebelum mengetahui apa itu bilangan acak semu, penting bagi kita untuk melihat perbedaan pengertian antara bilangan acak semu dalam konteks pemrograman biasa, seperti untuk permainan (game), simulasi, statistik dan sebagainya, dengan bilangan acak dalam konteks kriptografi.

Dalam konteks kriptografi, sebuah bilangan acak adalah bilangan yang tidak dapat diprediksi nilainya sebelum bilangan tersebut dihasilkan atau dibangkitkan. Secara umum, jika bilangan acak yang akan dihasilkan antara [0..n-1], bilangan tersebut tidak dapat diprediksi kemungkinan kemunculannya lebih dari 1/n.

Pembangkit Bilangan Acak Semu
Pseudorandom Number Generator (PNRG) atau dalam bahasa indonesia Pembangkit bilangan acak semu adalah sebuah algoritma yang membangkitkan sebuah deret bilangan yang tidak benar-benar acak. Keluaran dari pembangkit bilangan acak semu hanya mendekati beberapa dari sifat-sifat yang dimiliki bilangan acak.Walaupun bilangan yang benar-benar acak hanya dapat dibangkitkan oleh perangkat keras pembangkit bilangan acak, bukannya oleh perangkat lunak komputer, akan tetapi bilangan acak semu banyak digunakan dalam beberapa

seperti untuk simulasi dalam ilmu fisika, matematik, biologi dan sebagainya, dan juga merupakan hal yang sangat penting dalam dunia kriptografi. Beberapa algoritma enkripsi baik yang simetris maupun nirsimetris memerlukan bilangan acak sebagai parameter masukannya seperti parameter kunci pada algoritma kunci-publik dan pembangkitan initialization vector (IV) pada algoritma kunci-simetri. Walaupu terlihat sederhana untuk mendapatkan bilangan acak, tetapi diperlukan analisis matematika yang teliti untuk membangkitkan bilangan seacak mungkin. Robert R. Coveyoudari Oak Ridge National Laboratory menulis sebuah artikel berjudull "The generation of random numbers is too important to be left to chance" dan John von Neumann menulis sebuah ide yang lebih menarik "Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin"

Berikut ini beberapa pembangkit bilangan acak semu :

1. linear congruential generators (LCG)
2. lagged Fibonacci generators
3. linear feedback shift registers
4. generalised feedback shift registers
5. Blum Blum Shub
6. Mersenne twister

Untuk linear congruential generators (LCG) Blum Blum Shub dan Mersenne twister akan dibahas lebih jauh pada bab selanjutnya.

Karena semua PRNG berjalan diatas sebuah komputer yang deterministik maka keluaran yang dihasilkannya akan memiliki sifat yang tidak dimiliki bilangan random sesungguhnya yaitu periode. Hal ini berarti pada putaran tertentu setelah dijalankan akan dihasilkan deret yang berulang, lilihat pada LCG. Tentu saja jika sebuah pembangkit bilangan acak memiliki memory yang terbatas (karena dijalankan diatas komputer yang memiliki memori terbatas) setelah beberapa waktu pembangkit tersebut akan kembali pada state semula dan hal ini menyebabkan pengulangan deret yang dihasilkan sebelumnya. Pembangkit yang tidak memiliki periode (non-periodic generator) dapat saja dirancang pada sebuah komputer yang deterministik, tetapi dibutuhkan memori yang tidak terbatas selama program pembangkit bilangan tersebut dijalankan. Pada perangkat yang sekarang ada hal ini tidak mungkin dilakukan. Sebuah PNRG dapat dimulai dari state (keadaan) tertentu dengan parameter masukan yang dinamakan random seed (umpan acak), tetapi bagaimanapun acaknya umpan PRNG akan selalu menghasilkan deret yang sama.

Konsekuensi yang dihasilkan dari deterministik komputer pada prakteknya dapat saja dihindari. Panjang dari maksimum periode dibuat sepanjang mungkin sehingga tidak ada komputer yang dapat mencapai satu periode dalam waktu yang diharapkan. Jika satu periode tidak dicapai maka pengulangan deret bilangan acak tidak terjadi. Penggunaan cara seperti ini tidak cukup baik untuk beberapa aplikasi yang membutuhkan waktu komputasi yang cepat, karena semakin panjang suatu periode akan membutuhkan sumberdaya dari komputer yang besar juga.

Pada kenyataannya, keluaran dari beberapa PRNG menghasilkan bilangan-bilangan yang gagal dalam uji pendeteksian pola secara statistik. Dibawah ini beberapa masalahnya :
periode yang dihasilkan lebihh pendek (tidak penuh satu periode) dari yang diharapkan untuk beberapa nilai umpan. dalam hal ini umpan dikatakan lemah bagi PNRG.
distribusi yang buruk dari deret bilangan acak yang dihasilkan
Adanya korelasi antar deret bilangan acak yang dihasilkan.
Beberapa bit lebih acak dibandingkan bit-bit lainnya

Aplikasi pembangkit bilangan acak RANDU yang digunakan selama beberapa dekade pada komputer mainframe adalah contoh masalah deterministik komputer dalam menghasilkan bilangan acak. Selain itu, fungsi RAND yang dimiliki sistem operasi Windows hanya memiliki 15 bit 'state' dan bit tersebut akan diulang setelah membangkitkan 32768 byte.

[Pendekatan awal PRNG]
Awal PRNG pada komputer, diusulkan oleh John von Neumann pada tahun 1946, dikenal sebagai middle-square method. Metode ini sangat sederhana, algoritmanya adalah sebagai berikut:
pilih bilangan sembarang
kuadratkan bilangan tersebut
ambil beberapa digit ditengahdari hasil kuadrat tersebut
Bilangan yang diambil merupakan bilangan acak yang dihasilkan dari metode ini, bilangan tersebut juga merupakan umpan untuk iterasi menghasilkan bilangan acak selanjutnya.
Sebagai contohnya dipilih bilangan 1234, kemudian dikuadratkan menjadi 1522756 atau

dapat ditulis 01522756 dalam format 8 digit karena bilangan yang dipilih pertama adalah 4 digit. 5227 merupakan bilangan yang dihasilkan pada iterasi pertama sebagai bilangan acak. Iterasi selanjutnya menghasilkan 3215.

Masalah dari metode "middle-square" adalah semua deret-deretnya akan dengan cepat mengulang dirinya sendiri.

PNRG yang cocok untuk aplikasi-aplikasi kriptografi disebut cryptographically secure PRNG (PNRG) atau dalam bahasa indonesia Pembangkit Bilangan Acak yang Aman secara Kriptografi. Perbedaan yang mendasar antara PNRG dengan CSPNRG adalah bahwa CSPNRG harus tampak random dari berbagai jenis pemeriksaan atau uji kerandoman sedangkan PNRG hanya harus terlihat random pada uji statistik standar. Bagaimanapun ada beberapa pembangkit bilangan acak yang didesain kriteria tersebut yang walaupun tidak cukup kuat untuk secara kriptografi

Dibawah ini beberapa kelompok dari CSPNRG :
Stream ciphers atau block ciphers yang berjalan pada mode counter atau output feedback
Perancangan algoritma khusus dengan keamaanan yang terjamin.
PNRG yang dirancang agar aman secara kriptografi
Kombinasi beberapa PNRG dengantujuan menghilangkan ketidak acakan suatu bilangan dalam deret

CSPNRG akan dibahas lebih jelas pada bab selanjutnya.

German Institute for Security in Information Technology menetapkan 4 kriteria kualitas untuk pembangkit bilangan acak semu yang deterministik
untuk keperluan aplikasi kriptografi, hanya pembangkit bilangan acak yang memenuhi kriteria sampai K-4 yang dapat diterima.

Pembangikit Bilangan Acak Semu yang Aman Secara Kriptografi

Seperti telah dijelaskan pada bab sebelunya, cryptographically secure pseudo-random number generator (CSPRNG) merupakan pseudo-random number generator (PRNG) dengan properti-properti yang membuat pembangkit bilangan acak tersebut aman secara kriptografi, dalam hal ini cocok digunakan pada aplikasi kriptografi.

Kualitas dari keacakan suatu bilangan yang dibutuhkan beberapa aplikasi kriptografi berbeda-beda. Sebagai contohnya untuk menciptakan sebuah nonce pada beberapa protokol yang dibutuhkan hanyalah keunikannya. Di sisi lain, pembangkitan sebuah kunci membutuhkan kualitas yang lebih tinggi

Syarat-syarat sebuah PNRG pasti dipenuhi oleh CSPNRG, tetapi tidak demikian sebaliknya. ada 2 syarat utama yang harus dipenuhi oleh CSPNRG. Pertama, CSPNRG memiliki properti statistik yang bagus artinya lulus uji keacakan secara statistik. Kedua, CSPNRG tidak dapat ditembus dengan berbagai macan serangan, bahkan jika sebagian dari initial state atau pun running state diketahui oleh penyerang.

Kebanyakan PNRG tidak dapat digunakan sebagai CSPNRG dan akan gagal pada dua properti yang disebutkan diatas. Walaupun PNRG tersebut mengeluarkan deret bilangan yang terlihat acak dan dapat lulus dalam uji keacakan statistik, tapi tidak tahan terhadap reverese engineering (rekayasa balik). Selain itu pada kebanyakan PNRG ketika state tertentu didapatkan atau diketahui maka bilangan acak selanjutnya yang akan dihasilkan dapat diprediksi nilainya.

CSPNRG secara eksplisit dirancang agar tahan terhadap serangan-serangan seperti itu.

Perancangan CSPNRG dibagi menjadi tiga kelas yaitu berdasarkan block cipher, berdasarkan permasalahan matematis, dan perancangan untuk tujuan tertentu
1. Perancangan berdasarkan block cipher
2. Perancangan berdasarkan teori bilangan
3. Perancangan tujuan tertentu

Standar-standar untuk CSPNRG diantaranya:
1. FIPS 186-2
2. NIST SP 800-90: Hash_DRBG, HMAC_DRBG, CTR_DRBG and Dual_EC_DRBG.
3. ANSI X9.17-1985 Appendix C
4. ANSI X9.31-1998 Appendix A.2.4
5. ANSI X9.62-1998 Annex A.4, obsoleted by ANSI X9.62-2005, Annex D (HMAC_DRBG)

Linear Congruential Generator

Linear Congruential Generator atau jika diubah kedalam bahasa Indonesia menjadi Pembangkit Bilangan Acak Kongruen-Lanjar merupakan pembangkit bilangan acak yang sederhana, mudah dimengerti teorinya, dan juga mudah untuk diimplementasikan.

LCG didefinisikan dalam relasi berulang berikut :

$$X_n = (AX_{n-1} + B) \bmod M$$

dimana
Xn = bilangan acak ke-n dari deretnya
Xn-1 = bilangan acak sebelumnya
A = faktor pengali
B = *increment*
M=modulus
X0 adalah kunci pembangkit atau disebut juga umpan (*seed*)

Periode LCG paling besar adalah M bahkan pada kebanyakan kasus periodenya kurang dari M. Maksudnya adalah deret bilangan acak yang dihasilkan tidak lebih banyak dari modulonya. Perhatikan contoh berikut.

Misalkan
untuk $X1_n$ A = 5 , B = 13, M = 23 dan $X_0 = 0$
untuk $X2_n$ A = 4 , B = 12, M = 23 dan $X_0 = 0$

$$X1_n = (5X1_{n-1} + 13) \bmod 23$$

$$X2_n = (4X1_{n-1} + 12) \bmod 23$$

n	X1n	X2n
0	0	0
1	13	12
2	9	14
3	12	22
4	4	8
5	10	21
6	17	4
7	6	5
8	20	9
9	21	2
10	3	20
11	5	0
12	15	12
13	19	14
14	16	22
15	1	8
16	18	21
17	11	4
18	22	5
19	8	9
20	7	2
21	2	20
22	0	0
23	13	12
24	9	14
25	12	22

Tabel1

Dari tabel 1 diatas, terlihat bahwa deret bilangan acak yang dihasilkan berulang setelah n tertentu. Untuk $X1_n$ deret berulang pada n yang ke 23, hal ini sama dengan nilai Mnya yang berarti periode untuk $X1_n$ adalah sebesar M atau dengan kata lain $X1_n$ mempunyai periode penuh. Sedangkan pada $X2_n$ kita dapat melihat bahwa periodenya kurang dari M, berulang pada n = 11. Pemilihan nilai A sebagai faktor pengali dan B sebagai *increment* mempengaruhi deret bilangan acak yang dihasilkan oleh algoritma ini.

LCG akan memiliki periode penuh jika memenuhi syarat sebagai berikut :

1. B relatif prima terhadap M
2. A-1 dapat dibagi dengan semua faktor prima dari M
3. A-1 kelipatan 4 jika M kelipatan 4
4. nilai M lebih besar dari max(A,B,X_0)
5. A > 0 dan B > 0

Walaupun cepat dalam pemrosesan tapi LCG tidak dipakai sebagai pembangkit bilangan acak untuk kriptografi karena hasil yang dikeluarkan sangat tergantung terhadap nilai A , B , dan M. Selain itu deret yang dihasilkan dapat dengan mudah ditebak nilainya.

Blum Blum Shub

blum blum shub adalah sebuah pembangkit bilangan acak semu (pseudo random genertor) yang aman secara kriptografi, diusulkan tahun 1986 oleh Lenore Blum, Manuel Blum dan Michael Shub.

Secara sederhana algoritmanya adalah sebagai berikut :

1. Pilih dua buah bilangan prima rahasia, p dan q, yang masing-masing kongruen dengan 3 modulo 4.
2. Kalikan keduanya menjadi n = pq. Bilangan m ini disebut bilangan bulat Blum
3. Pilih bilangan bulat acak lain, s, sebagai umpan sedemikian sehingga:
   (i) $2 \leq s < n$
   (ii) s dan n relatif prima
   kemudian hitung x0 = s2 mod n
4. Barisan bit acak dihasilkan dengan melakukan iterasi berikut sepanjang yang diinginkan:
   (i) Hitung xi = xi – 1 2 mod n
   (ii) zi = bit LSB (Least Significant Bit) dari xi

Barisan bit acak yang dihasilkan adalah adalah z1, z2, z3, …

Kode sumber Blum Blum Shub dapat dilihat pada tabel dibawah ini:

```
/*
 * File : blumblumshub.java
 */

import java.util.Random;
import java.security.SecureRandom;
import java.math.BigInteger;

public class BlumBlumShub {
        // pre-compute a few values
        private static final BigInteger two = BigInteger.valueOf(2L);
        private static final BigInteger three = BigInteger.valueOf(3L);
        private static final BigInteger four = BigInteger.valueOf(4L);

        private BigInteger n;
        private BigInteger state;

        /**
         * Generate appropriate prime number for use in Blum-Blum-Shub.
         *
         * This generates the appropriate primes (p = 3 mod 4) needed to compute the
         * "n-value" for Blum-Blum-Shub.
         *
         * @param bits Number of bits in prime
         * @param rand A source of randomness
         */
        private static BigInteger getPrime(int bits, Random rand) {
                BigInteger p;
                while (true) {
                        p = new BigInteger(bits, 100, rand);
                        if (p.mod(four).equals(three))
                                break;
                }
                return p;
        }

        /**
         * This generates the "n value" -- the multiplication of two equally sized
         * random prime numbers -- for use in the Blum-Blum-Shub algorithm.
         *
         * @param bits
         *          The number of bits of security
         * @param rand
         *              A random instance to aid in generating primes
         * @return A BigInteger, the <i>n</i>.
         */
        public static BigInteger generateN(int bits, Random rand) {
                BigInteger p = getPrime(bits/2, rand);
                BigInteger q = getPrime(bits/2, rand);

                // make sure p != q (almost always true, but just in case, check)
                while (p.equals(q)) {
                        q = getPrime(bits, rand);
                }
                return p.multiply(q);
        }

        /**
          * Constructor, specifing bits for <i>n</i>
         *
         * @param bits number of bits
         */
        public BlumBlumShub(int bits) {
                this(bits, new Random());
        }

        /**
         * Constructor, generates prime and seed
```

```
	 *
	 * @param bits
	 * @param rand
	 */
	public BlumBlumShub(int bits, Random rand) {
		this(generateN(bits, rand));
	}

	/**
	 * A constructor to specify the "n-value" to the Blum-Blum-Shub algorithm.
	 * The inital seed is computed using Java's internal "true" random number
	 * generator.
	 *
	 * @param n
	 *          The n-value.
	 */
	public BlumBlumShub(BigInteger n) {
		this(n, SecureRandom.getSeed(n.bitLength() / 8));
	}

	/**
	 * A constructor to specify both the n-value and the seed to the
	 * Blum-Blum-Shub algorithm.
	 *
	 * @param n
	 *          The n-value using a BigInteger
	 * @param seed
	 *            The seed value using a byte[] array.
	 */
	public BlumBlumShub(BigInteger n, byte[] seed) {
		this.n = n;
		setSeed(seed);
	}

	/**
	 * Sets or resets the seed value and internal state
	 *
	 * @param seedBytes
	 *          The new seed.
	 */
	public void setSeed(byte[] seedBytes) {
		// ADD: use hardwired default for n
		BigInteger seed = new BigInteger(1, seedBytes);
		state = seed.mod(n);
	}

	/**
	 * Returns up to numBit random bits
	 *
	 * @return int
	 */
	public int next(int numBits) {
		// TODO: find out how many LSB one can extract per cycle.
		//  it is more than one.
		int result = 0;
		for (int i = numBits; i != 0; --i) {
			state = state.modPow(two, n);
			result = (result << 1) | (state.testBit(0) == true ? 1 : 0);
		}
		return result;
	}

	/**
	 * main program to test blumblumshub
	 */
	public void main(String[] args) {
		// First use the internal, stock "true" random number
		// generator to get a "true random seed"
		SecureRandom r = new SecureRandom();
		System.out.println("Generating stock random seed");
		r.nextInt(); // need to do something for SR to be triggered.

		// Use this seed to generate a n-value for Blum-Blum-Shub
		// This value can be re-used if desired.
		System.out.println("Generating N");
		int bitsize = 512;
		BigInteger nval = BlumBlumShub.generateN(bitsize, r);

		// now get a seed
		byte[] seed = new byte[bitsize/8];
		r.nextBytes(seed);

		// now create an instance of BlumBlumShub
		BlumBlumShub bbs = new BlumBlumShub(nval, seed);
```

```
                // and do something
                System.out.println("Generating
10 bytes");
                for (int i = 0; i < 10; ++i) {

        System.out.println(bbs.next(8));
                }

                // OR
                // do everything almost
automatically
                BlumBlumShub bbs2 = new
BlumBlumShub(bitsize /*,+ optional random
instance */);

                // reuse a nval (it's ok to do this)
                BlumBlumShub bbs3 = new
BlumBlumShub(nval);
        }
}
```

Mersenne Twister

Mersenne twister adalah pembangkit bilangan acak semu yang dibuat pada tahun 1997 oleh Makoto Matsumoto dan Takuji Nishimura

Kode sumber Mersenne Twister dapat dilihat pada tabel dibawah ini:

```
public class MersenneTwister {
        /**
         * N, Internal array size
         */
        private static final int N = 624;

        /**
         * M
         */
        private static final int M = 397;

        /**
         * State vector
         */
        private final int[] mt = new int[N];

        /**
         * Internal counter of position in state.
         * mti == N+1 means we haven't been
initialized yet,.
         */
        private int mti = N+1;

        /**
         * Constants
         */
        private static final int mag01[] = {0x0,
0x9908b0df};

        /**
         * Constructor LSB of the current ime
         */
        public MersenneTwister() {
                setSeed((int)
System.currentTimeMillis());
        }

        /**
         * Constructor using a given seed.
         */
        public MersenneTwister(final int seed) {
                setSeed(seed);
        }

        /**
         * Constructor using an array.
         */
        public MersenneTwister(final int[] array)
{
                setSeed(array);
        }

        /**
         * Initalize the pseudo random number
generator with 32-bits.
         */
        public void setSeed(final int seed) {
                mt[0] = seed;
                for (mti = 1; mti < N; mti++) {
                        mt[mti] = (1812433253
* (mt[mti - 1] ^ (mt[mti - 1] >>> 30)) + mti);
                }
        }

        /**
         * Direct seeding using an array.
         *
         */
        public void setSeed(final int[] array) {
                setSeed(19650218);
                int i = 1;
                int j = 0;
                int k = (N > array.length ? N :
array.length);

                for (; k != 0; k--) {
                        mt[i] = (mt[i] ^ ((mt[i -
1] ^ (mt[i - 1] >>> 30)) * 1664525))
                                        +
array[j] + j;
```

```
                        i++;
                        j++;
                        if (i >= N) {
                                mt[0] = mt[N -
1];
                                i = 1;
                        }
                        if (j >= array.length)
                                j = 0;
                }
                for (k = N - 1; k != 0; k--) {
                        mt[i] = (mt[i] ^ ((mt[i -
1] ^ (mt[i - 1] >>> 30)) * 1566083941))
                                        - i;
                        i++;
                        if (i >= N) {
                                mt[0] = mt[N -
1];
                                i = 1;
                        }
                }
                mt[0] = 0x80000000; // MSB is
1; assuring non-zero initial array
        }

        public int next(final int bits) {
                int y;
                if (mti >= N) {
                        int kk;
                        for (kk = 0; kk < N - M;
kk++) {
                                y = (mt[kk] &
0x80000000) | (mt[kk + 1] & 0x7fffffff);
                                mt[kk] =
mt[kk + M] ^ (y >>> 1) ^ mag01[y & 0x1];
                        }
                        for (; kk < N - 1; kk++)
{
                                y = (mt[kk] &
0x80000000) | (mt[kk + 1] & 0x7fffffff);
                                mt[kk] =
mt[kk + (M - N)] ^ (y >>> 1) ^ mag01[y & 0x1];
                        }
                        y = (mt[N - 1] &
0x80000000) | (mt[0] & 0x7fffffff);
                        mt[N - 1] = mt[M - 1] ^
(y >>> 1) ^ mag01[y & 0x1];

                        mti = 0;
                }

                y = mt[mti++];
                y ^= y >>> 11;
                y ^= (y << 7) & 0x9d2c5680;
                y ^= (y << 15) & 0xefc60000;
                y ^= (y >>> 18);

                return y >>> (32 - bits);
        }
}
```

Kesimpulan

Dari beberapa algoritma untuk membangkitkan bilangan acak semu, tidak ada yang benar-benar dapat menghasilkan bilangan acak secara sempurna dalam arti benar-benar acak dan tanpa ada perulangan selama pembankit yang digunakan adalah komputer yang memiliki sifat deterministik. Bilanga yang benar-benar acak hanya dapat dihasilkan oleh perangkat keras (hardware)

Referensi

E. Knuth. *The Art of Computer Programming*, Volume 2: *Seminumerical Algorithms*, Third Edition. Addison-Wesley, 1997. ISBN 0-201-89684-2. Section 3.2.1: The Linear Congruential Method, pp.10–26.

A thorough discussion: Stephen K. Park and Keith W. Miller, *Random Number Generators: Good Ones Are Hard To Find*, Communications of the ACM, 31(10):1192-1201, 1988

Michael Luby, *Pseudorandomness and Cryptographic Applications*, Princeton Univ Press, 1996. A definitive source of techniques for provably random sequences.

John von Neumann, "Various techniques used in connection with random digits," in A.S. Householder, G.E. Forsythe, and H.H. Germond, eds., *Monte Carlo Method*, National Bureau of Standards Applied Mathematics Series, 12 (Washington, D.C.: U.S. Government Printing Office, 1951)

Donald Knuth. *The Art of Computer Programming*, Volume 2: *Seminumerical Algorithms*, Third Edition. Addison-Wesley, 1997. ISBN 0-201-89684-2. Chapter 3, pp.1–193. Extensive coverage of statistical tests for non-randomness.

Peterson. Ivars. *The Jungles of Randomness: A Mathematical Safari.* Wiley, NY, 1998. (pp. 178) ISBN 0-471-16449-6

"Various techniques used in connection with random digits", Applied Mathematics Series, no. 12, 36-38 (1951).