HKU CSIS Tech Report TR-2003-07

# Development of Cryptographic Random Number Generators*

W.W. Tsang, C.W. Tso, Lucas Hui, K.P. Chow, K.H. Pun, C.F. Chong, H.W. Chan

Department of Computer Science and Information Systems
The University of Hong Kong

August 2003

**Abstract**
The key generation module is the most secret component of a cryptosystem. Keys are generated using random number generators (RNGs). In an implementation of a cryptosystem, we may consider developing the RNG component separately by in-house engineers instead of by the main contractor. This document describes an easy-to-follow procedure for the development of highly secure cryptographic RNGs. A checklist for auditing a secure cryptographic RNG is also included.

## 1. Introduction

The use of random number generators (RNGs) in generating cryptographic keys is common. The most important criteria for such application are: (i) that the keys are chosen from a very large set, and (ii) uncertainty. The first criterion requires that the RNG must have a very long period and the RNG passes all known tests of randomness. The second criterion requires that there is no feasible way to predict that certain numbers are more likely to be generated than others. There are two additional concerns in practice: (i) to ensure the correctness of software, and (ii) to maintain high unpredictability of keys even when a host computer has been seized or intruded. An overview of cryptographic RNGs can be found in [Menezes97]. Many practical problems in the design and analysis of cryptographic RNGs have been addressed in [Gutmann98].

The key generation module is the most secret component of a cryptosystem. For security, an architect of a cryptosystem may want to develop his own RNGs instead of using other people's programs or designs. This paper provides a practical guide for the design and testing of cryptographic RNGs. Section 2 describes various kinds of RNGs and their characteristics. We recommend combining outputs of at least four generators of different kinds, including an unpredictable one. Section 3 describes the most well known sets of statistical tests of randomness. These tests check whether the outputs of an RNG are uniformly distributed and independent. We suggest that at least two component RNGs in the combined generator shall pass all these tests. Section 4 describes how to use the collision test to ensure that the seed space of an RNG is not trivially small. If the seed space of an RNG is too small, an attacker will be able to regenerate a key by initializing the RNG with all possible seeds one by one exhaustively. Section 5 suggests practical measures that protect the key generation module against system attacks. It also includes ways that minimize the leakage of secret when the computer that runs the module has been seized. Section 6 provides a checklist for auditing a secure RNG. Redundancy that safeguards the security has been built-in.

*Research supported by Innovation and Technology Support Programme, HKSAR, Grant ITS/277/00

## 2. Random number Generation

There are two main categories of RNGs in computers—deterministic RNGs and unpredictable RNGs. The former compute next random number from current states of the RNG whereas the latter collect uncertainties from special devices or computer hardware. Many deterministic RNGs are backed with strong theories and they produced statistically satisfactory random numbers. However, the numbers generated by some of them can be derived from preceding outputs [Plumstead82]. On the other hand, the output of an unpredictable RNG cannot be derived with certainty. These RNGs usually require special devices and are susceptible to hardware failures. Some of them are slow comparing with computers and their raw outputs may fail in some statistical tests.

To obtain the advantages of both groups, we suggest combining outputs of RNGs of different categories. Two bit sequences can be combined using modulo addition, subtraction or bitwise exclusive-or. The combination makes the numbers more independent, have longer period and are harder to predict. Marsgalia has proved that the result of combining two independent sequences of random numbers are more evenly distributed than any one of the original [Marsaglia84]. Moreover, the period of the sequence combined from two independent sequences is the least common multiple of the two originals. A secure RNG for cryptographic key generation shall consist of at least one unpredictable RNG and three deterministic RNGs of different families. The period of the combined RNG shall be larger than $2^{\boldsymbol{n}}$, where $n$ is the number of bits in a cryptographic key. Moreover, the amount of entropy gathered from an unpredictable RNG in the generation of one key shall exceed $n$ bits.

The following is a brief overview on some candidates for the component RNGs. Note that anyone of them will not be a secure RNG by itself.

### 2.1 Linear congruential generator (LCG)

LCG is the most well studied and popular deterministic generator. It was proposed by Lehmer in 1949 [Lehmer49] and is among the fastest RNGs. The general formula for computing the next number from the current one is $X_{n+1} = a X_n + c \bmod m$ . A thorough review of the underlying theory is found in [Knuth98]. If $c$ is not equal to zero, and $(a, m)$ are properly chosen as suggested in page 184 in Knuth's book, the period of the generator is $m$, independent of what the initial value is. The finding of a good multiplier, $a$, requires conducting the spectral test. An implementation of this test is available at *Center for Information Security and Cryptography, Department of Computer Science and Information Systems, The University of Hong Kong* <http://www.csis.hku.hk/~cisc>. It is well-known that LCG fails in many statistical tests.

### 2.2 Lagged Fibonacci generators

The Lagged Fibonacci generator was originally suggested by Mitchell and Moore in 1958 [unpublished]. The formula is $X_n = X_{n-p+q} + X_{n-p} \bmod m$. In a 32-bit computer, $m$ is usually set to $2^{32}$ for efficiency. The indices $p$ and $q$ are chosen such that $x^p + x^q + 1$ is a primitive trinomial. The period of such a generator is $2^{31}(2^p - 1)$. This generator is fast and easy-to-implement. The most prominent feature is that a very long period can be achieved by choosing a large $p$. One derivative suggested by Marsaglia is to replace the addition in the formula with multiplication, i.e., $X_n = X_{n-p+q} \times X_{n-p} \bmod m$ [Marsaglia84]. The period is $2^{30}(2^p - 1)$. This new generator scrambles the bits more thoroughly. A drawback is that only 31 bits are generated in each round (the least significant bit is dropped) and special attention is needed in the initialization. Another common derivative is called *generalized feedback shift register* generator (GFSR) suggested by Lewis and Payne [Lewis73] . The formula is $X_n = X_{n-p+q} \oplus X_{n-p}$ and the period is $2^p - 1$. This generator does not mix bits in different positions in a word. Special care is needed in the initialization of these generators.

## 2.3 Mersenne Twister

Mersenne Twister (MT) is an extension of GFSR generator suggested by Makoto Matsumoto and Takuji Nishimura [Matsumoto98]. The general formula is $x_{k+n} = x_{k+m} \oplus ((x_k{}^u \mid x_{k+1}{}^l)\mathbf{A})$, $1 \le m \le n$. Let $w$ be the number of bits in a word and $0 \le r \le w$-1. $x_k{}^u$ is the upper $w$-$r$ bits of $x_k$ where $x_{k+1}{}^l$ is the lower $r$ bits of $x_{k+1}$ . The operator, | , concatenates the two operands to form a word. **A** is a $w \times w$ binary matrix. When a vector of bits multiplies with **A**, say $x$**A**, the effect is equivalent to (i) first shifting $x$ to the right 1 bit position, (ii) if the rightmost bit of the original $x$ is 1, exclusive-or the shifting result with the last row of **A**.

If the parameters ($w, n, m, r$) are chosen according to the guidelines set in [Matsumoto98], the period of this generator is $2^{nw-r} - 1$. Moreover, by returning $z = x_{k+n}\mathbf{T}$, where **T** is a binary matrix that tampers the bits in $x_{k+n}$, the resulting sequence has a very desirable theoretical feature—evenly distributed in high dimension with high accuracy. A version of MT where ($w, n, m, r$) = (32, 624, 397, 31) has been implemented in C and posted in *Mersenne Twister Home Page* <http://www.math.keio.ac.jp/~matumoto/emt.html>. It runs fast and has a period of $2^{19937} - 1$. This generator passes all known statistical tests of randomness.

## 2.4 Blum-Blum-Shub generators (BBS)

The formula for the BBS generator is $X_{n+1} = X_n{}^2 \bmod m$, where $m$=$pq$, $|p| = |q|$ (i.e. both have equal lengths), p and $q$ are distinct primes of the form 4$x$+3 [Blum86]. With the assumption that the quadratic residuacity problem is intractable, the BBS generator is practically unpredictable. Unlike other deterministic generators, the period of a BBS generator depends on the seed and can only be worked out using an algorithm. Moreover, the BBS generator is much slower than other deterministic generators.

## 2.5 DSA generators

Two deterministic generators are suggested in the standard for digital signature algorithm (DSA) [FIPS-186]. A DSA generator starts with a random seed and computes the next value using a hash function, either SHA-1 or DES. As they are well-studied and are approved by the standard institute, these generators are considered very secure and commonly used in cryptosystems. Comparing with other deterministic generators used in simulations, these generators are much slower.

## 2.6 Unpredictable generators

An unpredictable generator gathers entropy (randomness) from physical phenomena such as

- radioactive decay
- thermal noise
- transistor noise
- clock
- states of volatile memory/hardware
- keyboard/mouse movement timings

These generators do not require a seed. The numbers generated are unpredictable and irreproducible. Many are implemented in special devices attached to a host computer and are subject to hardware failure. In general, their speed is slower than the deterministic ones. Outputs of some unpredictable generators are found to be non-uniformly distributed. This problem can be rectified by combining their outputs with those of a good deterministic generator.

### 2.6.1 Intel Random Number Generator [Jun99, Intel810]

The Intel RNG resides in the 82802 Firmware Hub (FWH), which is a core component of the Intel 810 chipset. The generator uses thermal noise from resistors as the source of randomness. The noise is amplified and used to modulate a low frequency clock. This clock is then used as a reference to sample a high frequency clock. The drift between the two clocks induces randomness in the sample values.

### 2.6.2 HAVEGE [www.irisa.fr/caps/projects/hipsor/HAVEGE.html]

Hardware volatile entropy gathering and expansion (HAVEGE) is a software generator that gathers entropy from the timing differences in carrying out machine level instructions. The differences are caused by the states of instruction and data caches, branch predictions, translation lookaside buffer (TLB) and the unpredictable time delay affected by interrupts. The states of the cache and TLB determine whether there will be a page-missed delay and the duration of the delay. The entropy gathered is further mixed with output of a deterministic generator. The ultimate throughput is more than 100 Mbits per second. The program of this generator is machine dependent. The platforms currently supported include Sun workstations and PCs.

### 2.6.3 Linux RNG [Linux RNG]

The Linux RNG is implemented in the kernel module. It maintains an entropy pool that consists of timings of inter-key presses, mouse interrupts, block request interrupts, etc. Random numbers are generated by taking the SHA function on the pool contents. This generator is very slow.

### 2.6.1 ComScire QNG Model J1000KU [www.comscire.com]

The device gathers entropy from thermal noise and amplifier noise. The company that markets the device claims that the generator produces one Mbits per second that passes all tests of randomness.

## 3. Tests of Randomness

Statistical tests are commonly used to reject poor RNGs. The underlying hypothesis is that the numbers being tested are uniformly distributed and independent. A test uses the numbers in an experiment and checks whether the statistics collected are within typical ranges. For example, the collision test simulates throwing balls randomly into urns [Christiansen75, Knuth98]. A collision occurs when a ball falls into an urn that is already occupied. The test rejects an RNG if too many or too few collisions occur.

### 3.1 Diehard Battery [Marsaglia95]

Diehard is the most widely distributed and used software package for checking RNGs. It includes the following tests [Marsaglia02].

- Birthday Spacings
- GCD
- Gorilla
- Overlapping Permutations
- Binary Rank $n \times n$
- Binary Rank 6×8
- Monkey Tests OPSO, OQSO, DNA
- Count the 1's
- Count the 1's specific
- Parking Lot
- Minimum Distance
- Random Spheres
- The Squeeze
- Overlapping Sums
- Runs Up and Down
- The Craps

The tests are designed for checking 32-bit RNGs. We have made minor modifications of the code so that they are also applicable to RNGs of 24- and 31-bits. The tests are then applied to the 57 deterministic RNGs in the GSL-GNU Library <http://www.gnu.org/software/gsl/gsl.html>. If a test returns a p-value of greater than 0.01 and less than 0.99 for an RNG, we put a 'P' (means "Passed") in the respective entry. Otherwise we put an 'F' (means "Failed"). The results are shown in Table 3.1.

| RNG ID | RNG | No. of Bits | Birthday Spacing | GCD | Gorilla | Over-lapping | Binary Rank nxn | Binary Rank 6x8 | OPSO | OQSO | DNA | Count the 1's |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | borosh13 | 32 | F | F | F | P | F | F | F | F | F | F |
| 2 | Cmrg | 31 | P | P | P | P | P | P | P | P | P | P |
| 3 | Coveyou | 32 | F | F | F | P | F | F | F | F | F | F |
| 4 | fishman18 | 31 | F | F | F | P | F | F | F | F | F | F |
| 5 | fishman20 | 31 | F | F | F | P | F | F | F | F | F | F |
| 6 | fishman2x | 31 | P | P | P | P | P | P | P | P | P | P |
| 7 | gfsr4 | 32 | P | P | P | P | P | P | P | P | P | P |
| 8 | Knuthran | 30 | F | P | P | P | P | P | P | P | P | P |
| 9 | knuthran2 | 31 | F | F | F | P | P | F | F | F | F | F |
| 10 | lecuyer21 | 31 | F | F | F | P | P | P | P | P | P | P |
| 11 | Minstd | 31 | F | F | F | P | P | P | P | P | P | P |
| 12 | Mrg | 31 | P | P | P | P | P | P | P | P | P | P |
| 13 | mt19937 | 32 | P | P | P | P | P | P | P | P | P | P |
| 14 | R250 | 32 | F | F | F | P | P | F | F | F | F | F |
| 15 | ran0 | 31 | F | F | F | P | P | P | P | P | P | P |
| 16 | ran1 | 31 | F | P | P | P | P | P | P | P | P | P |
| 17 | ran2 | 31 | P | P | P | P | P | P | P | P | P | P |
| 18 | ran3 |  | F | F | P | P | P | P | P | P | P | P |
| 19 | Rand | 31 | F | F | F | P | P | F | F | F | F | F |
| 20 | rand48 | 32 | P | P | F | P | P | P | F | F | F | P |
| 21 | random128-bsd | 31 | F | P | P | P | P | P | P | P | P | P |
| 22 | random128-glibc2 | 31 | F | P | P | P | P | P | P | P | P | P |
| 23 | random128-libc5 | 31 | F | P | P | P | P | P | P | P | P | P |
| 24 | random256-bsd | 31 | F | P | P | P | P | P | P | P | F | P |
| 25 | random256-glibc2 | 31 | F | P | P | P | P | P | P | P | P | P |
| 26 | random256-libc5 | 31 | F | P | P | P | P | P | P | P | P | P |
| 27 | random32-bsd | 31 | F | F | F | P | P | F | F | F | F | F |
| 28 | random32-glibc2 | 31 | F | F | F | P | P | P | F | F | F | F |
| 29 | random32-libc5 | 31 | F | F | F | P | P | P | F | F | F | F |
| 30 | random64-bsd | 31 | F | P | F | P | P | P | P | F | F | P |
| 31 | random64-glibc2 | 31 | F | P | F | P | P | P | P | F | F | P |
| 32 | random64-libc5 | 31 | F | P | F | P | P | P | P | F | F | P |
| 33 | random8-bsd | 31 | F | F | F | P | P | F | F | F | F | F |
| 34 | random8-glibc2 | 31 | F | F | F | P | P | F | F | F | F | F |
| 35 | random8-libc5 | 31 | F | F | F | P | P | F | F | F | F | F |
| 36 | random-bsd | 31 | F | P | P | P | P | P | P | P | P | P |
| 37 | random-glibc2 | 31 | F | P | P | P | P | P | P | P | P | P |
| 38 | random-libc5 | 31 | F | P | P | P | P | P | P | P | P | P |
| 39 | Randu | 31 | F | F | F | F | F | F | F | F | F | F |
| 40 | Ranf | 32 | P | P | F | P | P | P | F | F | F | P |
| 41 | Ranlux | 24 | F | P | P | P | P | P | P | P | P | P |
| 42 | ranlux389 | 24 | P | P | P | P | P | P | P | P | P | P |
| 43 | ranlxd1 | 32 | P | P | P | P | P | P | P | P | P | P |
| 44 | ranlxd2 | 32 | P | P | P | P | P | P | P | P | P | P |
| 45 | ranlxs0 | 24 | F | P | P | P | P | P | P | P | P | P |
| 46 | ranlxs1 | 24 | P | P | P | P | P | P | P | P | P | P |
| 47 | ranlxs2 | 24 | P | P | P | P | P | P | P | P | P | P |
| 48 | Ranmar | 24 | P | P | P | P | P | P | P | P | P | P |
| 49 | Slatec | 22 | F | F | F | F | P | F | F | F | F | F |
| 50 | Taus | 32 | P | P | P | P | P | P | P | P | P | P |
| 51 | Transputer | 32 | F | F | F | P | F | F | F | F | F | F |
| 52 | Tt800 | 32 | P | P | P | P | P | P | P | P | P | P |
| 53 | Uni | 15 | F | P | F | P | P | P | P | P | P | P |
| 54 | Uni32 | 31 | F | P | F | P | P | P | P | P | P | P |
| 55 | Vax | 32 | F | F | F | P | P | F | F | F | F | F |
| 56 | waterman14 | 32 | F | F | F | P | F | F | F | F | F | F |
| 57 | Zuf | 24 | F | P | P | P | P | P | P | P | P | P |

Table 3.1a  Results of testing the GSL-GNU RNGs with the Diehard Battery.

| RNG # | RNG | No. of Bits | Count the 1's Specific | Parking Lot | Minimum Distance | Random Spheres | Squeeze | Over-lapping Sums | Runs | Craps |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | borosh13 | 32 | F | P | P | P | P | P | P | P |
| 2 | Cmrg | 31 | P | P | P | P | P | P | P | P |
| 3 | Coveyou | 32 | F | P | P | P | P | P | P | P |
| 4 | fishman18 | 31 | F | P | F | P | P | P | P | P |
| 5 | fishman20 | 31 | F | P | F | P | P | F | P | P |
| 6 | fishman2x | 31 | P | P | P | P | P | P | P | P |
| 7 | gfsr4 | 32 | P | P | P | P | P | P | P | P |
| 8 | Knuthran | 30 | P | P | P | P | P | P | P | P |
| 9 | knuthran2 | 31 | F | P | P | P | P | P | P | P |
| 10 | lecuyer21 | 31 | P | P | P | P | P | P | P | P |
| 11 | Minstd | 31 | P | P | P | P | P | P | P | P |
| 12 | Mrg | 31 | P | P | P | P | P | P | P | P |
| 13 | mt19937 | 32 | P | P | P | P | P | P | P | P |
| 14 | R250 | 32 | F | P | P | P | P | P | P | P |
| 15 | ran0 | 31 | P | P | P | P | P | F | P | P |
| 16 | ran1 | 31 | P | P | P | P | P | P | P | P |
| 17 | ran2 | 31 | P | P | P | P | P | P | P | P |
| 18 | ran3 |  | P | P | P | P | F | P | P | P |
| 19 | Rand | 31 | F | P | P | P | P | P | P | P |
| 20 | rand48 | 32 | P | P | P | P | P | P | P | P |
| 21 | random128-bsd | 31 | P | P | P | P | F | P | P | P |
| 22 | random128-glibc2 | 31 | P | P | P | P | F | F | P | P |
| 23 | random128-libc5 | 31 | P | P | P | P | F | P | P | P |
| 24 | random256-bsd | 31 | P | P | P | P | P | P | P | P |
| 25 | random256-glibc2 | 31 | P | P | P | P | P | P | P | P |
| 26 | random256-libc5 | 31 | P | P | P | P | P | P | P | P |
| 27 | random32-bsd | 31 | F | P | P | P | F | F | P | F |
| 28 | random32-glibc2 | 31 | F | P | P | P | F | P | P | F |
| 29 | random32-libc5 | 31 | F | P | P | P | F | P | P | P |
| 30 | random64-bsd | 31 | P | P | P | P | F | P | P | P |
| 31 | random64-glibc2 | 31 | P | P | P | P | F | P | P | P |
| 32 | random64-libc5 | 31 | P | P | P | P | F | P | P | P |
| 33 | random8-bsd | 31 | F | P | P | P | P | P | P | P |
| 34 | random8-glibc2 | 31 | F | P | P | P | P | P | P | P |
| 35 | random8-libc5 | 31 | F | P | P | P | P | P | P | P |
| 36 | random-bsd | 31 | P | P | P | P | F | P | P | P |
| 37 | random-glibc2 | 31 | P | P | P | P | F | F | P | P |
| 38 | random-libc5 | 31 | P | P | P | P | F | P | P | P |
| 39 | Randu | 31 | F | P | F | P | P | P | P | F |
| 40 | Ranf | 32 | P | P | P | P | P | P | P | P |
| 41 | Ranlux | 24 | P | P | P | P | P | P | P | P |
| 42 | ranlux389 | 24 | P | P | P | P | P | P | P | P |
| 43 | ranlxd1 | 32 | P | P | P | P | P | P | P | P |
| 44 | ranlxd2 | 32 | P | P | P | P | P | P | P | P |
| 45 | ranlxs0 | 24 | P | P | P | P | P | P | P | P |
| 46 | ranlxs1 | 24 | P | P | P | P | P | P | P | P |
| 47 | ranlxs2 | 24 | P | P | P | P | P | P | P | P |
| 48 | Ranmar | 24 | P | P | P | P | P | P | P | P |
| 49 | Slatec | 22 | F | P | P | P | F | F | P | P |
| 50 | Taus | 32 | P | P | P | P | P | P | P | P |
| 51 | Transputer | 32 | F | P | P | P | P | P | P | P |
| 52 | Tt800 | 32 | P | P | P | P | P | P | P | P |
| 53 | Uni | 15 | P | P | P | P | P | P | P | P |
| 54 | Uni32 | 31 | P | P | P | P | F | P | P | P |
| 55 | Vax | 32 | F | P | P | P | P | F | P | P |
| 56 | waterman14 | 32 | F | P | P | P | P | P | P | P |
| 57 | Zuf | 24 | P | P | P | P | P | P | P | P |

Table 3.1b  Results of testing the GSL-GNU RNGs with the Diehard Battery.

## 3.2 Knuth's collection

The most well-known collection of tests for random number generators is the one compiled by Knuth that comprises 11 tests [Knuth98]. The following briefly describes each test and includes the parameters used in our implementation.

### 3.2.1 Frequency Test

Sample $n$ random number in $[0, d]$, count the number of times that each value occurs and conducts a Chi-square goodness-of-fit test. In our implementation, $d = 2^{24}$ and $n = 5d$.

### 3.2.2 Serial Test

Sample $n$ pairs of random numbers in $[0, d]$, For each pair of possible value, $(q, r)$, $0 \le q, r \le d$, count the number of times that $(q, r)$ occurs. Conduct a Chi-square goodness-of-fit test. In our program, $d = 2^{12}$ and $n = 5d^2$.

### 3.2.3 Gap Test

Let $\alpha$ and $\beta$ be two real numbers with $0 \le \alpha \le \beta \le 1$. $u$'s are uniform random number in [0,1). A gap of length $r$ is a consecutive subsequence, $u_j$, $u_{j+1}$, …, $u_{j+r}$ in which $u_{j+r}$ lies between $\alpha$ and $\beta$ but not the others. $r$ follows a geometric distribution. Conducts a Chi-square goodness-of-fit test on the samples of $r$. In our program, $\alpha = 0$ and $\beta = 1/2^{20}$. $5\times2^{20}$ samples of $r$'s are tested.

### 3.2.4 Partition Test (Poker Test)

The classical poker test considers five cards chosen randomly and observes which of the following seven patterns is matched:

| | |
|---|---|
| All different: | abcde |
| One pair: | aabcd |
| Two pairs: | aabbc |
| Three of a kind: | aaabc |
| Full house: | aaabb |
| Four of a kind: | aaaab |
| Five of a kind: | aaaaa |

A simpler version is implemented. The card value is a random number in [0, 511]. A hand consists of 5 cards. Only five categories are considered:

5 values = all different
4 values = one pair;
3 values = two pairs or three of a kind
2 values = full house, or four of a kind
1 value = five of a kind

A Chi-square test is conducted on approximately 45 million samples of hands.

### 3.2.5 Coupon Collector's Test

Consider that an RNG produces random number in the range of $[0, d)$. This test observes how many numbers are needed to obtain a complete set of integers from 0 to $d - 1$. In our program, $d = 256$ and 100,000 sets are sampled.

### 3.2.6 Permutation Test

A group of $t$ real numbers can have $t!$ possible relative orderings. The probability that an ordering occurs is $1/t!$ . In our program, $t = 10$ and $5t!$ groups are sampled.

### 3.2.7 Run Test

There are many versions of run test. The exact version described in Knuth's book is implemented. The length of a sequence, $n$, is chosen to be 10000.

### 3.2.8 Maximum-of-*t* Test

This test checks the distribution of the maximum of $t$ uniform random numbers in [0,1). The distribution function is $F(x) = x^t$, $0 \le x \le 1$. In our program, $t = 40$ and two million samples are taken. The Kolmogorov-Smirnov test is then used to check the goodness-of-fit.

### 3.2.9 Collision Test

This test simulates throwing balls randomly into urns. A collision occurs when a ball falls into an urn that is already occupied. The test counts the number of collisions. It rejects an RNG if too many or too few collisions occur. In our program, $2^{20}$ balls are thrown to $2^{20}$ urns.

### 3.2.10 Birthday Spacings Test

Consider choosing $m$ birthdays randomly from a year of $n$ days. Sort the birthdays. The spacings (intervals) between consecutive birthdays asymptotically follows the Poisson distribution with the parameter $\lambda = m^3/(4n)$. In our program, $n = 2^{32}$ and $m = 4096$.

### 3.2.11 Serial Correlation Test

Consider $n$ independent uniform random numbers, $u_0, u_1, \ldots, u_{n-1}$. The serial correlation $C$ is defined as

$$C = \frac{n(u_0u_1 + u_1u_2 + \cdots + u_{n-2}u_{n-1} + u_{n-1}u_0) - (u_0 + u_1 + \cdots + u_{n-1})^2}{n(u_0^2 + u_1^2 + \cdots + u_{n-1}^2) - (u_0 + u_1 + \cdots + u_{n-1})^2}$$

$C$'s value lies between –1 and 1with mean $\mu_n$ and variance $\sigma_n^2$ defined below

$$\mu_n = \frac{-1}{n-1}\ , \quad \sigma_n^2 = \frac{n^2}{(n-1)^2(n-2)}\ , \quad n > 2.$$

In our program, $n = 65536$. 1024 C's are generated and tested with the Kolmogorov-Smirnov test.

We have applied these tests on the 57 RNGs in the GSL-GNU Library. The test results are shown in Table 3.2.

| | RNG | No. of bits | Freq uency | Serial | Gap | Partition | Coupon | Permu tation | Run | Maximum -t | Collision | Birthday | Serial Corr |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | borosh13 | 32 | F | F | F | F | F | F | P | P | F | F | P |
| 2 | cmrg | 31 | P | P | P | P | P | P | P | P | P | P | P |
| 3 | coveyou | 32 | F | F | F | F | F | P | P | P | F | F | P |
| 4 | fishman18 | 31 | F | F | F | F | F | F | P | P | F | F | P |
| 5 | fishman20 | 31 | F | F | F | F | F | F | P | P | F | F | P |
| 6 | fishman2x | 31 | P | P | P | P | P | P | P | P | P | P | P |
| 7 | gfsr4 | 32 | P | P | P | P | P | P | P | P | P | P | P |
| 8 | knuthran | 30 | P | P | P | P | P | P | P | P | P | F | P |
| 9 | knuthran2 | 31 | F | F | F | F | F | P | P | P | F | F | P |
| 10 | lecuyer21 | 31 | F | F | F | P | P | F | P | P | F | F | P |
| 11 | minstd | 31 | F | F | F | P | P | F | P | P | F | F | P |
| 12 | mrg | 31 | P | P | P | P | P | P | P | P | P | P | P |
| 13 | mt19937 | 32 | P | P | P | P | P | P | P | P | P | P | P |
| 14 | R250 | 32 | P | F | P | P | P | P | P | P | P | F | P |
| 15 | ran0 | 31 | F | F | F | P | P | F | P | P | F | F | P |
| 16 | ran1 | 31 | F | P | P | P | P | P | P | P | P | F | P |
| 17 | ran2 | 31 | P | P | P | P | P | P | P | P | P | P | P |
| 18 | ran3 | | P | P | P | P | P | P | P | P | P | F | P |
| 19 | rand | 31 | F | F | F | F | F | F | P | P | F | F | P |
| 20 | rand48 | 32 | P | F | P | P | F | P | P | P | F | P | P |
| 21 | random128-bsd | 31 | P | P | P | P | P | P | P | F | P | F | P |
| 22 | random128-glibc2 | 31 | P | P | P | P | P | P | P | F | P | F | P |
| 23 | random128-libc5 | 31 | P | P | P | P | P | P | P | F | P | F | P |
| 24 | random256-bsd | 31 | P | P | P | P | P | P | P | P | P | F | P |
| 25 | random256-glibc2 | 31 | P | P | P | P | P | P | P | P | P | F | P |
| 26 | random256-libc5 | 31 | P | P | P | P | P | P | P | P | P | F | P |
| 27 | random32-bsd | 31 | F | F | P | F | F | F | P | F | F | F | P |
| 28 | random32-glibc2 | 31 | F | F | P | F | F | F | P | F | F | F | P |
| 29 | random32-libc5 | 31 | F | F | P | F | F | F | P | F | F | F | P |
| 30 | random64-bsd | 31 | F | F | P | F | F | P | P | F | F | F | P |
| 31 | random64-glibc2 | 31 | P | F | P | F | F | P | P | F | F | F | P |
| 32 | random64-libc5 | 31 | P | F | P | F | F | P | P | F | F | F | P |
| 33 | random8-bsd | 31 | F | F | F | F | F | F | P | P | F | F | P |
| 34 | random8-glibc2 | 31 | F | F | F | F | F | F | P | P | F | F | P |
| 35 | random8-libc5 | 31 | F | F | F | F | F | F | P | P | F | F | P |
| 36 | random-bsd | 31 | P | P | P | P | P | P | P | F | P | F | P |
| 37 | random-glibc2 | 31 | P | P | P | P | P | P | P | F | P | F | P |
| 38 | random-libc5 | 31 | P | P | P | P | P | P | P | F | P | F | P |
| 39 | randu | 31 | F | F | F | F | F | F | P | F | F | F | P |
| 40 | ranf | 32 | P | F | P | F | F | P | P | P | F | P | P |
| 41 | ranlux | 24 | P | P | P | P | P | P | P | P | P | F | P |
| 42 | ranlux389 | 24 | P | P | P | P | P | P | P | P | P | P | P |
| 43 | ranlxd1 | 32 | P | P | P | P | P | P | P | P | P | P | P |
| 44 | ranlxd2 | 32 | P | P | P | P | P | P | P | P | P | P | P |
| 45 | ranlxs0 | 24 | P | P | P | P | P | P | P | P | P | F | P |
| 46 | ranlxs1 | 24 | P | P | P | P | P | P | P | P | P | P | P |
| 47 | ranlxs2 | 24 | P | P | P | P | P | P | P | P | P | P | P |
| 48 | ranmar | 24 | P | P | P | P | P | P | P | P | P | P | P |
| 49 | slatec | 22 | F | F | F | F | F | F | P | P | F | F | F |
| 50 | taus | 32 | P | P | P | P | P | P | P | P | P | P | P |
| 51 | transputer | 32 | F | F | F | F | F | F | P | P | F | F | P |
| 52 | tt800 | 32 | P | P | P | P | P | P | P | P | P | P | P |
| 53 | uni | 15 | P | P | P | P | P | P | P | F | F | F | P |
| 54 | uni32 | 31 | P | P | P | P | P | P | P | F | F | F | P |
| 55 | vax | 32 | F | F | P | F | F | P | P | P | F | F | P |
| 56 | waterman14 | 32 | F | F | P | F | F | P | P | P | F | F | P |
| 57 | zuf | 24 | P | P | P | P | P | P | P | P | P | F | P |

Table 3.2 Results of testing the GSL-GNU RNGs with the tests in Knuth's collection.

## 3.3 NIST Test Suite

The National Institute of Standards and Technology (NIST) has suggested 16 statistical tests for checking RNGs in [Rukhin01]. These tests are:

- Frequency (Monobit) Test
- Frequency Test within a Block
- Runs Test
- Tests for the longest Run of Ones in a Block
- Binary Matrix Rank Test
- Discrete Fourier Transform (Spectral) Test
- Non-overlapping Template Matching Test
- Overlapping Template Matching Test
- Maurer's "Universal Statistical" Test
- Lampel-Ziv Compression Test
- Linear Complexity Test
- Serial Test
- Approximate Entropy Test
- Cumulative Sums (Cusum) Test
- Random Excursions Test
- Random Excursions Variant Test

We have downloaded the software posted in the official Website: *Random number generation and testing* <http://csrc.nist.gov/rng/>. We found that four of these tests, the Linear Complexity, Non-overlapping, Random Excursions and Random Excursions Variant tests, did not run properly. These four tests are re-coded according to the specifications. Then we applied all the tests to the RNGs in the GSL-GNU Library. The results are shown in Table 3.3.

| RNG # | RNG | No. of bits | Freq-uency (Monobit) | Block Freq-uency | Cumu -lative Sums | Runs | Long Runs | Rank | Fourier Trans-form | Over-lapping Template | Maurer's Universal |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | borosh13 | 32 | P | F | P | F | F | F | F | F | F |
| 2 | Cmrg | 31 | P | P | P | P | P | P | P | P | P |
| 3 | Coveyou | 32 | P | F | P | F | F | F | F | F | F |
| 4 | fishman18 | 31 | P | F | P | F | F | P | F | F | F |
| 5 | fishman20 | 31 | F | F | F | F | F | P | F | F | F |
| 6 | fishman2x | 31 | P | P | P | P | P | P | P | P | P |
| 7 | gfsr4 | 32 | P | P | P | P | P | P | P | P | P |
| 8 | Knuthran | 30 | P | P | P | P | P | P | P | P | P |
| 9 | knuthran2 | 31 | P | F | P | F | F | P | F | F | F |
| 10 | lecuyer21 | 31 | P | P | P | P | P | P | P | P | P |
| 11 | minstd | 31 | P | P | P | P | P | P | P | P | P |
| 12 | mrg | 31 | P | P | P | P | P | P | P | P | P |
| 13 | mt19937 | 32 | P | P | P | P | P | P | P | P | P |
| 14 | r250 | 32 | F | F | F | F | P | P | F | F | P |
| 15 | ran0 | 31 | P | P | P | P | P | P | P | P | P |
| 16 | ran1 | 31 | P | P | P | P | P | P | P | P | P |
| 17 | ran2 | 31 | P | P | P | P | P | P | P | P | P |
| 18 | ran3 |  | P | P | P | P | P | P | P | P | P |
| 19 | rand | 31 | P | F | P | P | P | P | F | P | P |
| 20 | rand48 | 32 | P | P | P | P | P | P | P | P | P |
| 21 | random128-bsd | 31 | P | P | P | P | P | P | P | P | P |
| 22 | random128-glibc2 | 31 | P | P | P | P | P | P | P | P | P |
| 23 | random128-libc5 | 31 | P | P | P | P | P | P | P | P | P |
| 24 | random256-bsd | 31 | P | P | P | P | P | P | P | P | P |
| 25 | random256-glibc2 | 31 | P | P | P | P | P | P | P | P | P |
| 26 | random256-libc5 | 31 | P | P | P | P | P | P | P | P | P |
| 27 | random32-bsd | 31 | F | P | F | F | P | P | P | P | P |
| 28 | random32-glibc2 | 31 | F | P | F | F | P | P | P | P | P |
| 29 | random32-libc5 | 31 | F | P | F | F | P | P | P | P | P |
| 30 | random64-bsd | 31 | P | P | P | P | P | P | P | P | P |
| 31 | random64-glibc2 | 31 | P | P | P | P | P | P | P | P | P |
| 32 | random64-libc5 | 31 | P | P | P | P | P | P | P | P | P |
| 33 | random8-bsd | 31 | P | P | P | P | P | P | F | P | P |
| 34 | random8-glibc2 | 31 | P | P | P | P | P | P | F | P | P |
| 35 | random8-libc5 | 31 | P | P | P | P | P | P | F | P | P |
| 36 | random-bsd | 31 | P | P | P | P | P | P | P | P | P |
| 37 | random-glibc2 | 31 | P | P | P | P | P | P | P | P | P |
| 38 | random-libc5 | 31 | P | P | P | P | P | P | P | P | P |
| 39 | randu | 31 | F | F | F | F | F | P | F | F | F |
| 40 | ranf | 32 | P | P | P | P | P | P | P | P | P |
| 41 | ranlux | 24 | P | P | P | P | P | P | P | P | P |
| 42 | ranlux389 | 24 | P | P | P | P | P | P | P | P | P |
| 43 | ranlxd1 | 32 | P | P | P | P | P | P | P | P | P |
| 44 | ranlxd2 | 32 | P | P | P | P | P | P | P | P | P |
| 45 | ranlxs0 | 24 | P | P | P | P | P | P | P | P | P |
| 46 | ranlxs1 | 24 | P | P | P | P | P | P | P | P | P |
| 47 | ranlxs2 | 24 | P | P | P | P | P | P | P | P | P |
| 48 | ranmar | 24 | P | P | P | P | P | P | P | P | P |
| 49 | slatec | 22 | P | F | P | P | P | P | F | P | F |
| 50 | taus | 32 | P | P | P | P | P | P | P | P | P |
| 51 | transputer | 32 | P | F | P | F | F | F | F | F | F |
| 52 | tt800 | 32 | P | P | P | P | P | P | P | P | P |
| 53 | uni | 15 | P | P | P | P | P | P | P | P | P |
| 54 | uni32 | 31 | P | P | P | P | P | P | P | P | P |
| 55 | vax | 32 | P | P | P | P | P | P | F | P | P |
| 56 | waterman14 | 32 | P | P | P | F | F | F | F | F | F |
| 57 | zuf | 24 | P | P | P | P | P | P | P | P | P |

Table 3.3a  Results of testing the GSL-GNU RNGs with the NIST tests.

| RNG # | RNG | No. of bits | Appro-ximate Entropy | Serial | Lempel-Ziv | Linear Comp-lexity | Non-over-lapping | Random Excur-sions | Random Excur-sions Variant |
|---|---|---|---|---|---|---|---|---|---|
| 1 | borosh13 | 32 | F | P | F | P | F | F | P |
| 2 | Cmrg | 31 | P | P | P | P | P | P | P |
| 3 | Coveyou | 32 | F | P | F | P | F | F | P |
| 4 | fishman18 | 31 | F | F | F | P | F | F | P |
| 5 | fishman20 | 31 | F | F | F | P | F | F | F |
| 6 | fishman2x | 31 | P | P | P | P | P | P | P |
| 7 | gfsr4 | 32 | P | P | P | P | P | P | P |
| 8 | Knuthran | 30 | P | P | P | P | P | P | P |
| 9 | knuthran2 | 31 | F | F | F | P | F | P | P |
| 10 | lecuyer21 | 31 | P | P | P | P | P | P | P |
| 11 | minstd | 31 | P | P | P | P | P | P | P |
| 12 | mrg | 31 | P | P | P | P | P | P | P |
| 13 | mt19937 | 32 | P | P | P | P | P | P | P |
| 14 | r250 | 32 | F | P | F | P | F | P | P |
| 15 | ran0 | 31 | P | P | P | P | P | P | P |
| 16 | ran1 | 31 | P | P | P | P | P | P | P |
| 17 | ran2 | 31 | P | P | P | P | P | P | P |
| 18 | ran3 |  | P | P | P | P | P | P | P |
| 19 | rand | 31 | P | P | P | P | F | P | P |
| 20 | rand48 | 32 | P | P | P | P | P | P | P |
| 21 | random128-bsd | 31 | P | P | P | P | P | P | P |
| 22 | random128-glibc2 | 31 | P | P | P | P | P | P | P |
| 23 | random128-libc5 | 31 | P | P | P | P | P | P | P |
| 24 | random256-bsd | 31 | P | P | P | P | P | P | P |
| 25 | random256-glibc2 | 31 | P | P | P | P | P | P | P |
| 26 | random256-libc5 | 31 | P | P | P | P | P | P | P |
| 27 | random32-bsd | 31 | F | F | P | P | P | P | P |
| 28 | random32-glibc2 | 31 | F | F | P | P | P | P | P |
| 29 | random32-libc5 | 31 | F | F | P | P | P | P | P |
| 30 | random64-bsd | 31 | P | P | P | P | P | P | P |
| 31 | random64-glibc2 | 31 | P | P | P | P | P | P | P |
| 32 | random64-libc5 | 31 | P | P | P | P | P | P | P |
| 33 | random8-bsd | 31 | P | P | P | P | F | P | P |
| 34 | random8-glibc2 | 31 | P | P | P | P | F | P | P |
| 35 | random8-libc5 | 31 | P | P | P | P | F | P | P |
| 36 | random-bsd | 31 | P | P | P | P | P | P | P |
| 37 | random-glibc2 | 31 | P | P | P | P | P | P | P |
| 38 | random-libc5 | 31 | P | P | P | P | P | P | P |
| 39 | randu | 31 | F | F | F | P | F | F | F |
| 40 | ranf | 32 | P | P | P | P | P | P | P |
| 41 | ranlux | 24 | P | P | P | P | P | P | P |
| 42 | ranlux389 | 24 | P | P | P | P | P | P | P |
| 43 | ranlxd1 | 32 | P | P | P | P | P | P | P |
| 44 | ranlxd2 | 32 | P | P | P | P | P | P | P |
| 45 | ranlxs0 | 24 | P | P | P | P | P | P | P |
| 46 | ranlxs1 | 24 | P | P | P | P | P | P | P |
| 47 | ranlxs2 | 24 | P | P | P | P | P | P | P |
| 48 | ranmar | 24 | P | P | P | P | P | P | P |
| 49 | slatec | 22 | F | F | P | P | F | P | P |
| 50 | taus | 32 | P | P | P | P | P | P | P |
| 51 | transputer | 32 | F | F | F | P | F | F | P |
| 52 | tt800 | 32 | P | P | P | P | P | P | P |
| 53 | uni | 15 | P | P | P | P | P | P | P |
| 54 | uni32 | 31 | P | P | P | P | P | P | P |
| 55 | vax | 32 | P | P | P | P | F | P | P |
| 56 | waterman14 | 32 | F | F | F | P | F | F | P |
| 57 | zuf | 24 | P | P | P | P | P | P | P |

Table 3.3b Results of testing the GSL-GNU RNGs with the NIST tests.

4. Initialization of RNGs

A deterministic RNG requires an initial state to start operation. The initial state may be computed from a seed supplied by a human operator, or an ever-changing data source, e.g., time. It has been repeatedly reported that loopholes are created due to mishandling of seeds [Gutmann98, Marsaglia01]. The following are considerations and suggestions on initializing RNGs.

(i) The amount of entropy in a seed for initializing all the RNGs in a cryptosystem must be larger than the bit length of a key.
(ii) The format of a seed provided by a human operator must be checked.
(iii) For each RNG, a value is computed from the seed. The computation is designed so that the change of any bit in the seed gives different results. This resulting value is then combined with the default state of the RNG to form an initial state.
(iv) Certain RNGs have restrictions on the states, e.g., bits cannot be all zeros. Make sure that the restrictions are satisfied in the initial states.

To assure that an RNG is properly initialized with a seed, we can conduct a modified collision test [Tsang00]. In the test, seeds are sampled. Each seed is used to initialize the RNG and one random number is generated. These random numbers are then examined with the collision test. If the seed space is too small or the seeds are distributed unevenly, the number of collisions record will be higher than expected.

## 5. Measures against system attacks

Nowadays, computers that are connected to the Internet are constantly being scanned by intrusion software. From time to time, we heard news that computers are tampered by intruders. In this section, we suggest ways that reduce the chance of system attacks, either via the Internet or by insiders. We also describe measures that prevent the regeneration of the keys previously generated when a cryptosystem has been seized.

### 5.1 Isolation of the key generation module

The key generation module is the most secret component of a cryptosystem. The computer that runs the module shall be a standalone machine locked in a concealed room. This computer shall be dedicated entirely to run the module and has one and only one link that connects itself to the main computer running the other components. The protocol of this link shall be different from those commonly used in the Internet or a local area network. If the main computer is connected to the Internet, it must have vigilant firewall and intrusion-detection software installed, including the ones that detect the notorious buffer-overflow attacks, *Immunix adaptive system survivability* <http://www.cse.ogi.edu/DISC/projects/immunix>.

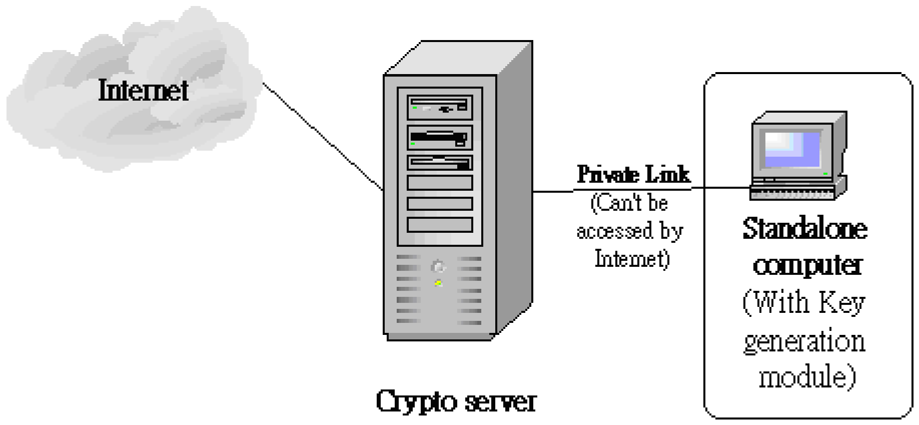

Figure 5.1 Isolation of the computer that runs the key generation module

### 5.2 Checking the integrity of the code and configuration files

The highest prize to an intruder is being able to alter a system without being noticed and he can predict the keys generated in the future. To avoid such pitfall, the code and configuration files shall be checked regularly for any illegal alterations. This can be done using hash functions, e.g., MD5 or SHA. A thorough discussion on using hash functions for checking against contamination of critical information can be found in *Check against Contamination of critical Information* <http://www.csis.hku.hk/cisc/projects/va/contam_index.html>

### 5.3 Power-up tests

The power-up tests are efficient statistical tests that examine the outputs of RNGs when a cryptographic RNG model is initialized. It may detect illegal alteration of code, initializing RNGs with improper seeds, or other problems in the module that deteriorates the quality of the output, e.g., the failure of a hardware RNG. Four tests are proposed in the Federal Information Processing Standards Publication (FIPS PUB140-2) in <http://csrc.ncsl.nist.gov/publications/fips/>. They are

- Monobit Test,
- Poker Test,
- Run Test,
- Long Runs Test.

Our study shows that these tests are efficient but far from effective (See Table 5.3). There are many poor RNGs pass these tests but fail in either Diehard or Knuth's collections. In the future, we will suggest replacing these tests with efficient versions of the gorilla test and the birthday spacing test [Marsaglia02]. These two tests are the most stringent that we have encountered in our pursuit in RNG research over 20 years.

## 5.4 Accumulating randomness regularly

If only deterministic RNGs are used, we suggest changing the seeds regularly, in particular, before and after the generation of a key. A new seed may be formed by combining the old one with some random data obtained from volatile caches or clock readings. The new amount of entropy accumulated in the seeds during the generation of one key shall exceed the number of bits in the key. With such a preventive measure, an attacker will not be able to work out the old or the new keys from the current states of the memory.

## 5.5 Hiding critical information

An effective way to prevent an attacker to work out the states of an RNG module is to hind the critical data in the memory. A scheme that scrambles data in RAM has been developed by our team. Details can be found in *Hide critical information in RAM* <http://www.csis.hku.hk/cisc/projects/va/ram_index.html>

| RNG # | RNG | Bits | Frequency (Monobit) | Poker | Runs | Long Runs |
|---|---|---|---|---|---|---|
| 1 | borosh13 | 32 | P | F | P | P |
| 2 | cmrg | 31 | P | P | P | P |
| 3 | coveyou | 32 | P | F | P | P |
| 4 | fishman18 | 31 | P | F | F | P |
| 5 | fishman20 | 31 | F | F | F | P |
| 6 | fishman2x | 31 | P | P | P | P |
| 7 | gfsr4 | 32 | P | P | P | P |
| 8 | knuthran | 30 | P | P | P | P |
| 9 | knuthran2 | 31 | P | F | F | P |
| 10 | lecuyer21 | 31 | P | P | P | P |
| 11 | minstd | 31 | P | P | P | P |
| 12 | mrg | 31 | P | P | P | P |
| 13 | mt19937 | 32 | P | P | P | P |
| 14 | r250 | 32 | P | F | P | P |
| 15 | ran0 | 31 | P | P | P | P |
| 16 | ran1 | 31 | P | P | P | P |
| 17 | ran2 | 31 | P | P | P | P |
| 18 | ran3 |  | P | P | P | P |
| 19 | rand | 31 | P | F | P | P |
| 20 | rand48 | 32 | P | P | P | P |
| 21 | random128-bsd | 31 | P | P | P | P |
| 22 | random128-glibc2 | 31 | P | P | P | P |
| 23 | random128-libc5 | 31 | P | P | P | P |
| 24 | random256-bsd | 31 | P | P | P | P |
| 25 | random256-glibc2 | 31 | P | P | P | P |
| 26 | random256-libc5 | 31 | P | P | P | P |
| 27 | random32-bsd | 31 | P | P | P | P |
| 28 | random32-glibc2 | 31 | P | P | P | P |
| 29 | random32-libc5 | 31 | P | P | P | P |
| 30 | random64-bsd | 31 | P | P | P | P |
| 31 | random64-glibc2 | 31 | P | P | P | P |
| 32 | random64-libc5 | 31 | P | P | P | P |
| 33 | random8-bsd | 31 | P | F | P | P |
| 34 | random8-glibc2 | 31 | P | F | P | P |
| 35 | random8-libc5 | 31 | P | F | P | P |
| 36 | random-bsd | 31 | P | P | P | P |
| 37 | random-glibc2 | 31 | P | P | P | P |
| 38 | random-libc5 | 31 | P | P | P | P |
| 39 | randu | 31 | F | F | F | P |
| 40 | ranf | 32 | P | P | P | P |
| 41 | ranlux | 24 | P | P | P | P |
| 42 | ranlux389 | 24 | P | P | P | P |
| 43 | ranlxd1 | 32 | P | P | P | P |
| 44 | ranlxd2 | 32 | P | P | P | P |
| 45 | ranlxs0 | 24 | P | P | P | P |
| 46 | ranlxs1 | 24 | P | P | P | P |
| 47 | ranlxs2 | 24 | P | P | P | P |
| 48 | ranmar | 24 | P | P | P | P |
| 49 | slatec | 22 | P | F | P | P |
| 50 | taus | 32 | P | P | P | P |
| 51 | transputer | 32 | P | F | F | P |
| 52 | tt800 | 32 | P | P | P | P |
| 53 | uni | 15 | P | P | P | P |
| 54 | uni32 | 31 | P | P | P | P |
| 55 | vax | 32 | P | P | P | P |
| 56 | waterman14 | 32 | P | F | P | P |
| 57 | zuf | 24 | P | P | P | P |

Table 5.3 Results of testing the GSL-GNU RNGs with the power-up tests in the Federal Information Processing Standards Publication (FIPS PUB140-2)

6. A check list for auditing a cryptographic random number generator (RNG) module
   a. Check the documentations on the following
      i. computing platform: computer, external connection, OS and compiler,
      ii. references (brand, model, websites, papers) of unpredictable RNGs,
      iii. formulations, parameters, periods, and references of deterministic RNGs,
      iv. how outputs of component RNGs are combined,
      v. format of power-up seeds provided by human operators,
      vi. method that initializes deterministic RNGs using the power-up seed,
      vii. how unpredictable data are collected and accumulated during execution,
      viii. power-up tests of randomness,
      ix. integrity checking on the executable code and the configuration file.

   b. The RNG module shall be accommodated in a stand-alone computer that is connected to the host computer via a dedicated connection. Security measures that detect both physical intrusion and electronic intrusion shall be taken, including a mechanism that detects intrusions using the stack-overflow technique.

   c. The RNG module shall consist of at least one unpredictable RNG and three deterministic RNGs of different kinds. When the module is invoked, it calls each component RNG. The values returned are then combined to form a random number.

   d. With high probability, the method that combines values returned from component RNGs shall give different results when any bit in its input is flipped. Verify this condition by altering a typical input bit by bit. Repeat 10000 times.

   e. Let $n$ be the number of bits in a cryptographic key generated using the RNG module. The least common multiple of the periods of the deterministic RNGs shall be larger than $2^n$.

   f. With high probability, the amount of entropy gathered from an unpredictable RNG in the generation of one cryptographic key shall exceed $n$ bits.

   g. The size of the seed space shall be larger than $2^n$.

   h. Review the code of the programs according to the documents.

   i. Verify the correctness of the implementation of each deterministic RNG using a mathematical package, e.g., *Maple*. Starting from same initial state, the numbers generated by both programs shall be exactly the same after 1, 10, 100, 1000, …, 1000000 rounds.

   j. Test the randomness of each component RNG using the following sets of tests. (The unpredictable RNG and at least two software RNGs shall pass all tests.)
      *i. Diehard*
      ii. Knuth's collection
      iii. Statistical Test Suite for RNGs for Cryptographic applications*, NIST Special Publication 800-22*

   k. The random numbers returned from the RNG module shall pass all the above tests.

l. For each deterministic RNG, use the collision test to check the randomness of the states initialized by random seeds.

m. Verify that a power-up seed provided by an operator is actually used to initialize each deterministic RNG. Check whether the RNG being tested produces different numbers when a bit of a typical seed is flipped. Repeat 10000 times.

n. Verify that the program actually collects and accumulates unpredictable data during execution, e.g., from the system clock. The data collected shall be used to change the states of some deterministic RNGs.

o. Check the effectiveness of the power-up tests using two defected deterministic RNGs that marginally fail in the tests.

p. Verify that the power-up tests are actually applied to each individual RNG when the module is initialized. If a hardware RNG is not in place or not functioning, the power-up tests shall show failure results.

q. The module shall regularly check, at least once per day, whether a hardware RNG is in place and functioning.

r. Test the signature scheme that verifies the integrity of the executable code and the configuration file.

s. Verify that the integrity checking is performed in the initialization of the module.

t. Check that the execution time of the module is acceptable.

## 7. References

Blum L., Blum, M, and Shub, M., 1986, *A simple unpredictable pseudo-random number generator*," *SIAM J. Comput.,* 15(2). pp. 364-83.

Center for Information Security and Cryptography, Department of Computer Science and Information Systems, The University of Hong Kong
http://www.csis.hku.hk/~cisc

Check against Contamination of critical Information, *Vulnerability Analysis Tools for Cryptographic Keys, Center for Information Security and Cryptography, Department of Computer Science and Information Systems, The University of Hong Kong*
http://www.csis.hku.hk/cisc/projects/va/contam_index.html

Comscire
*www.comscire.com*

Federal Information Processing Standards, Information Technology Laboratory, Computer Security Resource Center (CSRC), *National Institute of Standards and Technology*
http://csrc.ncsl.nist.gov/publications/fips/

FIPS PUB 140-2, *Security Requirements for Cryptographic Modules*
http://csrc.nist.gov/cryptval/140-2.htm

FIPS-186, Digital signature standard (DSS)
http://www.itl.nist.gov/fipspubs/fip186.htm

Gutmann, P., 1998, Software generation of practically strong random numbers, *Proceedings of the 7th USENIX Security symposium*, San Antonio, Texas, January 26-29 (an updated version of the paper is posted in http://www.cryptoapps.com/~peter/06_random.pdf ).

GSL-GNU Library
http://www.gnu.org/software/gsl/gsl.html

HAVAGE
*www.irisa.fr/caps/projects/hipsor/HAVEGE.html*

Hide critical information in RAM, *Vulnerability Analysis Tools for Cryptographic Keys, Center for Information Security and Cryptography, Department of Computer Science and Information Systems, The University of Hong Kong*
http://www.csis.hku.hk/cisc/projects/va/ram_index.html

Immunix adaptive system survivability
http://www.cse.ogi.edu/DISC/projects/immunix

Intel® 810 Chipset Design Guide, June 1999.

Jun, B.and Kocher, P, 1999, The Intel® Random Number Generator, Crytography research, Inc. white paper prepared for intel corporation.

Knuth, D. E., 1998, *The Art of Computer Programming*, Vol. 2, 3rd ed., Addison-Wesley.

Lehmer, D.H., 1949, Mathematical methods in large-scale computing units, *Proc. 2nd Sympos. on Large-Scale Digital Calculating Machinery, Cambridge, MA,* pages 141-146, Cambridge, MA, Harvard University Press.

Lewis, T.G. and Payne, W.H., 1973, Generalized Feedback Shift Register Pseudorandom Number Algorithm, *Journal of the ACM*, v.20, p.456-468.

Linux RNG, extensive comments in Linux source code '/usr/src/linux/drivers/char/random.c'

Marsaglia, G., 1984, A current View of Random Number Generators, Keynote Address, *Computer Science and Statistics: 16th Symposium on the Interface*, Atlanta.

Marsaglia, G., 1995, Diehard battery of tests of randomness, *The Marsaglia random number CDROM,* Department of Statistics, Florida State University.

Marsaglia, G, 2001, Letter to the editor: Problems with the use of computers for selecting jury panels, *Jurimetrics*, Vol. 41, No. 4.

Marsaglia, G. and Tsang W. W., 2002. Some difficult-to-pass tests of randomness, *Journal of Statistical Software*, (available at http://www.jstatsoft.org/ ), Vol. 7, Issue 3, Pages 1-8, January.

Matsumoto, M., and Nishimura, T., 1998, Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator, *ACM Trans. Model. Comput. Simul. 8*, No. 1, 3-30.

Menezes, A, van Oorschot P.C. and Vanstone, S.A. 1997, *Handbook of Applied Cryptography*, CRC Press.

Mersenne Twister Home Page
http://www.math.keio.ac.jp/~matumoto/emt.html

Random number generation and testing, *National Institute of Standards and Technology*
http://csrc.nist.gov/rng/

Plumstead, J. 1982, Inferring a sequence generated by a linear congruence, *Proceedings of the 23rd IEEE Symposium. on Foundations of Computer Science*, Chicago.

Rukhin, A., 2001, A statistical test suite for random and pseudorandom number generators for cryptographic applications, *NIST Special Publication* 800-22.

W.W. Tsang, L.C.K. Hui, K.P. Chow and C.F. Chong, 2000, Tuning the collision test for stringency, *Technical Report: TR-2000-05*, Department of Computer Science and Information Systems, The University of Hong Kong.