Analysis of Statistical Properties of Inherent Randomness

Peter O. Okech
Faculty of Information Technology
Strathmore University, Nairobi Kenya
pokech@strathmore.edu

Nicholas Mc Guire
Distributed & Embedded Systems Lab
School of Information and Science and Engineering
Lanzhou University, P.R.China
mcquire@lzu.edu.cn

Abstract

For contemporary computing systems with performance enhancing hardware components, the execution time of a given instruction or sequence of instructions shows a marked variance. The temporal indeterminism at instruction level can be attributed to the internal complexity of superscalar CPUs, as well as the effect of systems software. There is a need to develop metrics to measure this inherent randomness associated with the complexity of the hardware/software system. In this paper, we present an analysis of the statistical properties of inherent randomness. We consider this as the first step in the process of deriving a suitable metric. Our approach involves generating a random binary sequence based on the temporal non-deterministic execution of threads. We then perform a number of statistical tests, to evaluate quality of the bit stream produced. It is our belief that these results can form a basis for an acceptable metric of system level randomness and could further provide a set of quantitative measures that can be used to compare the complexity of hardware/software systems like GNU/Linux on contemporary superscalar/multi-core systems.

1 Introduction

A major focus in real time systems research is that of ensuring temporal determinism in relation to task execution. The approach taken by Real Time Operating Systems (RTOS) designers in improving determinism and predictability of their products is to minimize latency and jitter, while enforcing sequential predictability by excessive locking regimes. This approach has led to improved response time of RTOS, but has not eliminated system jitters. Some parts of the system jitter are associated with the instruction history rather than a functional code-path and must be viewed as inherent in complex software systems running on non-deterministic hardware.

In the field of timing analysis for schedulability of task in real time systems, one approach is that of the estimation of the worst case execution time (WCET) of a task. These figures are then combined for different tasks, in order to calculate the feasibility of meeting the deadlines of each of the task. To correctly calculate the WCET one must take into account the execution time of each instruction in the worst case execution path. It is now accepted that there are variation in the execution time of a program which occurs because the hardware it executes on varies in the amount of time required to perform the same set of instructions. These hard to predict variations [1] are mainly caused by the complex mechanisms employed by modern processors to increase their performance such as pipelining, caching, out-of-order execution of instructions and branch prediction. These performance enhancing techniques introduce local non-determinism in the timing behavior of individual instructions [2].

The notion of inherent randomness of modern computing hardware platform was introduced in [3]. Inherent randomness is the measure of how non-deterministic the execution of functionally deterministic instructions are on contemporary processors with performance enhancing features. Our aims are to measure this randomness, and develop an appropriate quantitative measure of the indeterminism of modern computing platforms. This is a challenging task, since we cannot directly quantify the level of randomness. In this respect, we have developed a multi-threaded random number generator (RNG) which harvest the entropy of the inherent randomness of modern hardware/software system.

In this paper, we present the concept behind the RNG based on inherent randomness of modern computing platform and the results of statistical test done on the binary sequence produced by the generator. The motivation for analyzing the statistical properties of inherent randomness is to investigate those attributes of the random sequence which could form the basis of a suitable metric for modern computing hardware/software platforms, and to allow quantitative comparison of complexity of such systems.

The rest of the paper is organized as follows. In section 2, we survey related works. Section 3 details some public domain test suites for empirical testing of random number generators. We describe in section 4 the main idea on which our RNG is based. Section 5, the methods section describes our choices made during the design and testing of the RNG. The results and our analysis is presented in section 6. The paper's conclusion is given in the last section.

2 Related Work

Our works is based on the variability of instruction execution time. This phenomena has been investigated in literature. An earlier investigation related to scheduling jitter of RTLinux and its relation to low-level CPU functional units - memory cache, BTB, TLB - issues was presented at the Real Time Linux Workshop in Lile 2005 [4]. The main conclusion of this work was that the jitter observed in the real-time Linux extensions was not primarily related to the interrupt processing, as was commonly claimed on mailing lists, but actually related to low-level hardware execution time variance and thus in principle not much influenced by the RTOS. Further the distribution of jitter indicated that the cause was actually primarily random processes occurring rather than well defined delays introduced by disabling of interrupts - the later would have resulted in visible "spikes" in the jitter rather than a smooth distribution. The Jitter distribution is shown in figure 1.

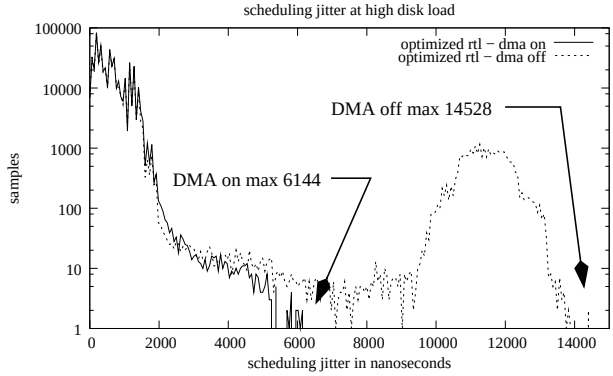

FIGURE 1: *Jitter Distribution*

Random number generators as usually classified as either true (or physical) random number generator (TRNG) or pseudo random number generators (PRNG). The fundamental difference between the two types is that a PRNG use a deterministic algorithm to generate random number sequences while a TRNG is an application that samples a source of entropy, traditionally a physical process. That is, given the same initial value (seed) the sequence produced by a PRNG repeats itself, a property that a TRNG lacks. Based on this perspective, software systems typically regarded as deterministic systems, are not considered as possible sources of true randomness. Davis and Niphadkar [5] describe how to use the non-determinism of the scheduling of concurrent thread to generate a sequence of random bits. They describe this approach as neither a TRNG or a PRNG, though we believe it should be classified as a true random number generator, since it does not use a deterministic algorithm to generate its sequence neither does it require seeding. This is similar to our work in which we extract randomness from the temporal variability of instruction execution in a modern superscalar CPU.

3 Statistical Evaluation

An ideal random number generator is characterized by the property that the random numbers it generates are independent and uniformly distributed on a finite range. These number can be either an integer or a real numbers. It is possible to view an integer or a real number in the range (0,1] as a sequence of binary bits through its binary expansion. Some generators directly produce binary digits, and

for such generators, the values from the set {0,1} that constitute the random sequence are expected to be independent and to have a long run average value of 0.5.

In order to evaluate a random number generator, the generator (or its output) is usually submitted to statistical tests that aims at revealing any statistical deficiencies by looking at empirical evidence against the null hypothesis. The null hypothesis, representing the perfect behavior of a truly random generator of random bits, is that the random bits are independent and identically distributed over the two element set {0,1} and there is zero covariance across all bit of the random number.

The output of most statistical tests is a p-value, which is a measure of the strength of the evidence provided by the data against the hypothesis. The significance level α of the test of a statistical hypothesis is the probability of rejecting the null hypothesis when it is true. A sequence is indicated as a pass/success by a given test if the p-value $\geq \alpha$, thus the null hypothesis is accepted, otherwise the hypothesis is rejected and the result is called a failure. The sequence that passes a test would be considered to be random with a confidence $1 - \alpha$.

It is generally accepted that passing a number of statistical tests does not prove that the output of an RNG is random sequence, but only improves the confidence in the RNG for a given application scenario.

The classical testing of a sequence of random numbers based on mathematical statistic is presented in Donald Knuth's seminal work [6]. These tests reveal the possible deviations of the distribution of the numbers from a uniform distribution. Over the years, others have implemented these traditional tests, as well as developed new more stringent tests. Among these are publicly available tests suites such as the ENT [8] test program from Formilab, DIEHARD [8], the NIST [9] and TestU01 [10]. We will briefly discuss these tests next.

ENT is a Pseudorandom Number Sequence Test Program developed by John Walker in 1998. It implements five standard tests for randomness, namely entropy, chi-square test, Arithmetic mean, Monte Carlo estimation of PI and Serial Correlation. These are straight forward mathematical metrics and identify major departure from randomness. The simplicity of the ENT test utility make it attractive for quick evaluation of the randomness of bit sequences.

The DIEHARD battery of tests was developed in 1996 by Prof. Georges Marsaglia from the Florida State University for testing randomness of sequences of numbers. It was supposed to give a better way of analysis in comparison to original Federal Information Processing Standard Publication FIPS 140 (FIPS) [11] statistical tests. It's other objective was to develop more stringent tests which go beyond Knuths classical methods in order to meet the new challenges posed by sophisticated applications of random numbers. The DIEHARD test is still widely regarded as a very comprehensive collection of tests for detecting non-randomness, though several authors have reported that it is no longer maintained.

The National Institute of Standards and Technology (NIST) developed a Statistical Test Suite in 2001 as a statistical package to test the randomness of arbitrary long binary sequences produced by either hardware or software based random number generators especially for cryptographic applications. The test suite was developed in response to a perceived need for a credible and comprehensive set of tests for binary random number generators. NIST is now considered as a de facto standard testing of RNGs for cryptographic applications.

The TestU01 suite is an ANSI C library of generic random number generators of various types. There is also a well defined collection of statistical tests which can be used to verify and test a generator. The tests can be applied to either a generator integrated into the library as a new generator or run on a file created from an external implementation. For testing pseudo-random number generators, the batteries SmallCrush, Crush and BigCrush are recommended. The tests in these batteries are used to test both the structure and the output of the PRNG. For binary sequence stored in a file, the batteries Rabbit, Alphabit, and BlockAlphabit can be used. We chose to perform tests on the binary files generated from our RNG using Alphabit and Rabbit which apply a variety of tests and produce 17 and 39 different statistics respectively. The tests in each of the batteries are shown in table 1 below.

Test	Alphabit		Rabbit	
	No	parameters	No	parameters
Serial Over	4	2, 4, 8, 16 bits	1	2
Hamming	3	6, 32 bits	7	16, 32, 64, 128
Random Walk	2	64, 320 bits	3	128, 1024, 10016
Close Pair			2	2, 4
Linear Complexity			1	
Lempel-Ziv Compression			1	
Auto Corellation			2	lad 1, 2
Runs test			1	
Frequency test within a Block			1	
Appearance Spacing			1	
Discrete Fourier Transform			2	
Periods in a String			1	
Binary Matrix rank			3	32x32, 320x320, 1024x1024

TABLE 1: *TestU01 tests for binary sequence*

The description of each of the tests are given in the TestU01 manuals.

4 Accessing the Randomness Source

In our earlier paper [3], we demonstrated that a modern CPU exhibits randomness that is not triggered by asynchronous events such as interrupts. We noted that this randomness scales with system complexity (both hardware and software). If the assumption of inherent randomness of the computing platform holds, then the main task would be to decide on which technique to use in capturing this source of entropy for the generation of a binary stream. There are potentially many approaches to harness bits from a randomness source. One of the objectives of this work was to devise a simple method to harvests the inherent randomness. The application that does this should ideally run in user space. Unlike the applications that sample physical processes, we set to develop a program that extracts the system entropy without the need for post-processing.

The approach we used to harvest the systems entropy involved writing a multi-threaded application, in which several threads share an unprotected memory variable. If the threads do not make make use of an explicit mechanism to prevent access of the variable from being simultaneous, then a data race can occur. A race condition is in general the result of nondeterministic ordering of a set of events [12]. The most common symptom of a race condition is unpredictable values of variables that are shared between multiple threads. To detect if the occurrence of a race has corrupted data, the value of the variable after an execution can be compared against a theoretically correct execution.

Our thesis is that the non deterministic execution a code sequence can impact the occurrence of a race condition. To illustrate this, consider two threads running on a uniprocessor system with preemption and a fixed scheduling quanta. Let us suppose that both the threads are updating a shared variable, e.g. incrementing the variable by 1. If the execution times of both threads are constant with no variations, it is possible that race conditions would occur after some duration with a noticeable pattern.

On the other hand, if the execution times of the two threads varies by infinitely small amount and the variance is uniformly distributed, then the pattern of race occurrence will vanish for a sufficiently long run of execution. The variations can be attributed to the previously discussed jitter inherent in the system. Similarly for systems with multiple execution cores, if the two threads are scheduled on different cores, any infinitely small jitter in the thread execution time can potential affect the ordering of the store operation.

Conceptually, by allowing a data race to occur, we can indicate this outcome by say the bit 1, and the non-occurrence of a race by the binary digit 0 during an execution - thus we can emit a random bit stream. The simplified view of the code is shown below.

```
/*
 * This is the TRNG!
 */
void * Thread(void *v) {
   unsigned long n;
   for (n = 1; n <= N; ++n) {
       ++count;
   }
```

```
   return NULL;
}

void generate_sequence(int *samples){
   int i,n;
   pthread_t t[NUM_THREADS];

   for(i=0;i<*samples;i++){
      count=0;

      for (n = 0; n < NUM_THREADS; ++n){
         if(pthread_create(t+n, ...)){
            perror("pthread_create");
            exit(-1);
      }  }
      for (n = 0; n < NUM_THREADS; ++n){
         if(pthread_join(t[n],NULL)){
            perror("pthread_join");
            exit(-1);
      }  }

      /* check the occurence of a race */
      if(count != N*NUM_THREADS){
                  /* emit 1 */

      } else {
                  /* emit 0 */

      }
   }
   return;
}
```

In the actual implementation, we make use of the value stored in the shared variable (count in the above code) as a source of the bit stream. If there is a race i.e.

```
count != N*NUM_THREADS
```

we take the least significant byte of the variable and add it to the pool of random numbers. This pool is then written to a binary file.

5 Approach

In this section, we describe the strategies used in relation to two main tasks, namely harvesting the entropy of the systems and performing the statistical tests. We first describe the design options available for the random number generator.

Several parameter values can be varied in to control the quality of the RNG output. The main parameters of the RNG are:

Number of Threads Determines the number of concurrent access to the shared variable.

Number of Runs The number executions for which the number of occurrence of the conditions is recorded. Statistically, this is the samples drawn from an infinite population.

Number of iterations The number of loops through which a thread iterates when updating the shared variable.

File Size The size of the binary file, expressed in bits that the RNG produces.

Number of Samples The number of files to output in an execution of the RNG.

The first decision to make is the number of the threads updating the shared variable. The simplest is the case where we have only 2 threads. We have generated the samples used for the test using 4 threads, to take advantage of the Quad processor available to us and determine the effect of CPU pining.

As stated in section 4. above, we use the occurrence of a race condition to generate the random bit sequence. For a given execution of the the threads, we want to know how many instances have resulted in a race condition. To determine this, we need a reasonable large number of runs. We have arbitrarily chosen the value of 1000, though any large value would be appropriate. In statistical terms, this corresponds to how many observations are made for which the events race or no race will be recorded.

The threads in our RNG compete to update the value of an unprotected shared variable. If the number of updates is low, the probability of a race condition is also low. This value is represented by the variable N, the number of loops that the threads iterates through when updating the variable. We investigated the effect of different values of N on the number of races for an idle system. For a small value of N only a few races occur in 1000 runs, but when the value of N is increased, the number of races also increases, but the probability never reaches 100%. This distribution of races for against the number of loops is depicted in figure 2.

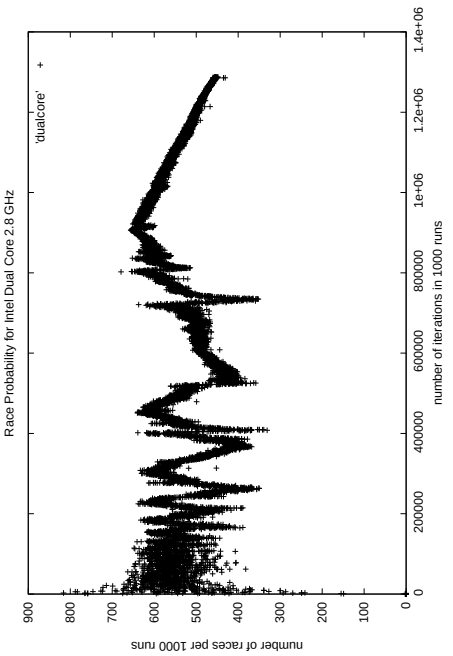

FIGURE 2: *Races Probability*

In the design of the RNG, one fundamental question is this - What is the optimal value of N? We posit that it is dependent on 1) the architecture and 2) the system load. By studying the plot of N against the number of races, we can determine the appropriate value for a given hardware architecture. To handle changes in the load of the system, its is not possible to manually manipulate the value of N. To deal with this challenge we developed an adaptive piece of code, which is still require fine tuning, that would increase or decrease the value with the following strategies:

- ensure that the number of races is close to 60% of the number of runs
- ensure that changes are within given range, to avoid generating biased bit streams

We have also allowed for the specification of the size of the binary file to be generated. With this option, it was possible to create small files, and test them in order to see the effect of other parameters. Our strategy for generating very large files (e.g. 1GB to enable validation for cryptographic applications) is to generate within a loop 10 samples of 1MB each which can later be combined. For this paper, we have performed tests on 1 MB files generated using this approach.

To perform preliminary statistical tests, we used the the ENT test suite. The ENT has a small set of simple tests, which are easy to interpret. We then used the TestU01 suite to test the statistical properties of the binary sequence generated by our RNG. We chose to use TestU01 since it is the most comprehensive of the publicly available and it encompasses most of the test in the other suites such as DIEHARD and the NIST suites. Specifically, we used the test batteries Alphabit and Rabbit which are collections of tests designed to verify the randomness properties of sequences stored in a binary file.

The experimental platform for used in our work consisted of Intel processors with varied complexity. For the results presented in this paper, the RNG was tested and executed on the following platforms, all based on the Debian distribution (lenny) with the Linux kernel 2.6.26 series:

- 3.0 GHz Intel Xeon Quad - sample 1
- 2.8 GHz Intel Pentium Dual Core - sample 2

6 Results and Analysis

Our first examination of the data set produced by our RNG was to confirm if indeed the occurrence of the race condition is a stochastic process. Based on the Central Limit Theorem that states that for a large sample many random events converge to a Gaussian distribution, we expect the plot of frequencies of the number of races in a large sample of the run of our RNG to exhibit an approximate bell curve. The frequency distribution of races over a sample of 100000 runs with 4 threads on the 2.8 MHz Pentium Dual Core machine is given is given in Figure 3. The data was fitted to the Gaussian distribution with parameters a=170.86, b=604.932 and c=0.0306713 based on the function $g(x) = ae^{-\frac{(x-b)^2}{2c^2}}$.

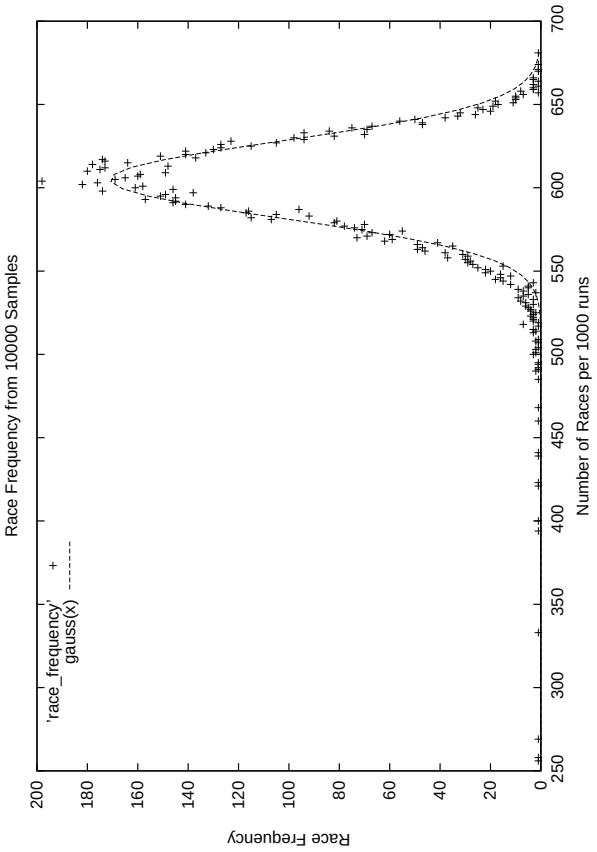

FIGURE 3: *Race Frequency fitted to a Gaussian Distribution*

We performed tests from the ENT test suite on 1 MB files from the two system, termed as sample 1 and sample 2. For comparison purposes, we include test on the Linux /dev/random from the Xeon Quad machine. The runtime option used during the test, -b tells the test to treat the input file as a bit stream. The results is presented in the table 2.

	Sample1	Sample2	dev/random
Test			
Entropy	1.000000	1.000000	1.000000
Chi square	1.909247 (16.70%)	2.112779 (14.61%)	0.507247 (47.63%)
Arithmetic mean	0.499761	0.499749	0.499888
Monte Carlo value for Pi	3.144624 (0.10%)	3.144636 (0.10%)	3.142313 (0.02%)
Serial correlation	0.000066	-0.000484	0.000059

TABLE 2: *Results of the ENT Tests*

The value for entropy, which represents the information density of the file's content is not appropriate in this case, since the input files are viewed as a bit stream. The Chi square distribution values for our RNG are higher that those of /dev/random. The percentage values which indicates how frequently a truly random sequence would exceed the value calculated are in respectable range, and far much better than the degree to which the dev/random sequence is suspected of being non-random.

The values for Arithmetic Mean are better for /dev/random than the output of our RNG. The estimated value of Pi using the Monte Carlo method and the Serial Correlation of the bytes in the file are close to the expectation. In our analysis, the values based on this five basic tests show that the high quality of the entropy source, especially for a complex processor such as the Quad CPU.

The same data set used in the ENT test were subjected to the TestU01 battery suites Rabbit and Alphabit. A description of the tests in these batteries can be found in TestU01 documentation. The results of the tests are given in the Table 2.

Battery	Statistics	Failures	
		Sample 1	Sample 2
Alphabit	17	0	0
Rabbit	39	0	1

TABLE 3: *Results of TestU01*

All the tests in the Alphabit battery were passed by the two samples. The output from the Dual core system failed the Hamming Correlation test for a block size of 128 bit, returning a p-value of 0.00099.

The test using ENT and TestU01 show that for the 1 MB binary sequence from the two architectures, the Xeon Quad Processor and the Pentium Dual Core Processor produce high quality random sequences.

7 Conclusions

In this paper, we have presented the results of the statistical tests of the random bit pattern extracted from the inherent non-determinism of contemporary computing platform. First, we developed a simple random number generator based on the execution of multiple threads. By allowing threads to race on an unprotected variable we have produced quality random binary sequences.

We then subjected the output of the random

number generator a number of statistical tests. The results from the ENT test suite show that the statistical values of Arithmetic means and Monte Carlo simulation of PI are all in the reasonable ranges of random numbers. So are the values of Chi square and Serial correlation. The bit streams also passed the tests in the Rabbit and Alphabit batteries of the TestU01 statistical test suite, providing evidence that the RNG that we developed to harness the inherent randomness of instruction execution in the underlying computational architecture generate quality random numbers.

The results presented in this paper are based on an idle systems, with only the basic services running. We are currently working on the parameters that would filter out the effects of system loads. We are also investigating an approach that would automatically change system parameters to allow the RNG to produce non-biased bit streams even in the presence of high system load.

In the future, we plan to conduct further test on hardware of varying complexity, from simple pre-2006 processors to the newer generation Intel processors. We hope to extend the analysis to other architectural families.

Our proposal to use the statistical properties of random sequence as attributes of a system randomness metric needs deeper investigation. As opposed to being a single value of merit, we expect that the metric of inherent system randomness to be composed of several statistics. Thus we need to further investigate which of the statistics produced by the test suites as suitable attributes of the envisioned metric.

References

[1] *Measuring Performance in Real-Time Linux*, Fredrick M. Proctor, In the Proceedings of the 3rd Real-Time Linux Workshop, Milan, 2001.

[2] *Design for time-predictability*, Lothar Thiele and Reinhard Wilhelm, In: Proceedings of the Dagstuhl Perspectives Workshop on Design of Systems with Predictable Behavior, 2004.

[3] *Analysis of inherent randomness of the Linux kernel*, Nicholas Mc Guire, Peter Okech, Georg Schiesser, In Proceedings of the 11th Real-Time Linux Workshop, Dresden, 2009.

[4] *Benchmarking - Cache issues*, Nicholas Mc Guire and Qingguo Zhou, In Proceedings of the 11th Real-Time Linux Workshop, Lille, 2006.

[5] *LibMT-PRNG: A Multitheaded Pseudo Random Number Generator*, Mathew Davis and Sameer Niphadkar, Dr Dobb's Journal, April 2009.

[6] *The Art of Computer Programming, Volume 2: Seminumerical Algorithms (2nd Edition)*, Donald E. Knuth, Addison-Wesley, 1981.

[7] *ENT: A Pseudorandom Number Sequence Test Program*, http://www.fourmilab.ch/random/

[8] *Diehard Battery of Tests of Randomness*, http://www.stat.fsu.edu/pub/diehard/

[9] *NIST, Special Publication 800-22, A statistical Test Suite for Random and Pseudo-random Number Generators for Cryptographic Applications*, Available at: http://csrc.nist.gov/groups/ST/toolkit/rng/

[10] *TestU01: A C Library for Empirical Testing of Random Number Generators*, Pierre Lecuyer and Richard Simard, ACM Transactions on Mathematical Software, 2007.

[11] *Federal Information Processing Standards Publication 140-1, Security Requirements for Cryptographic Modules*, U.S. Department of Commerce/NIST, 1994, Springfield, VA: NTIS

[12] *Modern Operating Systems, 3rd Edition*, Andrew S. Tanenbaum, Prentice Hall, 2007.