Statistical Properties of Pseudo Random Sequences and Experiments with PHP and Debian OpenSSL

Yongge Wang[1] and Tony Nicol[2]

[1] UNC Charlotte, USA
[2] University of Liverpool, UK
yongge.wang@uncc.edu, tonynicol@inbox.com

Abstract. NIST SP800-22 (2010) proposed the state of the art statistical testing techniques for testing the quality of (pseudo) random generators. However, it is easy to construct natural functions that are considered as GOOD pseudorandom generators by the NIST SP800-22 test suite though the output of these functions is easily distinguishable from the uniform distribution. This paper proposes solutions to address this challenge by using statistical distance based testing techniques. We carried out both NIST tests and LIL based tests on the following pseudorandom generators by generating more than 200TB of data in total: (1) the standard C linear congruential generator, (2) Mersenne Twister pseudorandom generator, (3) PHP random generators (including Mersenne Twister and Linear Congruential based), and (4) Debian Linux (CVE-2008-0166) pseudorandom generator with OpenSSL 0.9.8c-1. As a first important result, our experiments show that, PHP pseudorandom generator implementation (both linear congruential generators and Mersenne Twister generators) outputs completely insecure bits if the output is not further processed. As a second result, we illustrate the advantages of our LIL based testing over NIST testing. It is known that Debian Linux (CVE-2008-0166) pseudorandom generator based on OpenSSL 0.9.8c-1 is flawed and the output sequences are predictable. Our LIL tests on these sequences discovered the flaws in Debian Linux implementation. However, NIST SP800-22 test suite is not able to detect this flaw using the NIST recommended parameters. It is concluded that NIST SP800-22 test suite is not sufficient and distance based LIL test techniques be included in statistical testing practice. It is also recommended that all pseudorandom generator implementations be comprehensively tested using state-of-the-art statistically robust testing tools.

Keywords: pseudorandom generators, statistical testing, OpenSSL, the law of the iterated logarithm.

1 Introduction

The weakness in pseudorandom generators could be used to mount a variety of attacks on Internet security. Heninger et al [6] surveyed millions of TLS and SSH servers and found out that 0.75% of TLS certificates share keys due to poor implementation of pseudorandom generators. Furthermore, they were able to recover RSA private keys for 0.50% of TLS hosts and 0.03% of SSH hosts because their public keys shared non-trivial common factors (due to poor implementation of pseudorandom generators), and

M. Kutyłowski and J. Vaidya (Eds.): ESORICS 2014, Part I, LNCS 8712, pp. 454–471, 2014.

DSA private keys for 1.03% of SSH hosts because of insufficient signature randomness (again, due to poor implementation of pseudorandom generators). It is reported in the Debian Security Advisory DSA-1571-1 [3] that the random number generator in Debian's OpenSSL release CVE-2008-0166 is predictable. The weakness in Debian pseudorandom generator affected the security of OpenSSH, Apache (mod_sl), the onion router (TOR), OpenVPN, and other applications (see, e.g., [1]). These examples show that it is important to improve the quality of pseudorandom generators by designing systematic testing techniques to discover these weak implementations in the early stage of system development.

Statistical tests are commonly used as a first step in determining whether or not a generator produces high quality random bits. For example, NIST SP800-22 Revision 1A [12] proposed the state of art statistical testing techniques for determining whether a random or pseudorandom generator is suitable for a particular cryptographic application. In a statistical test of [12], a significance level $\alpha \in [0.001, 0.01]$ is chosen for each test. For each input sequence, a P-value is calculated and the input string is accepted as pseudorandom if P-value $\geq \alpha$. A pseudorandom generator is considered good if, with probability α, the sequences produced by the generator fail the test. For an in-depth analysis, NIST SP800-22 recommends additional statistical procedures such as the examination of P-value distributions (e.g., using χ^2-test). In section 3, we will show that NIST SP800-22 test suite has inherent limitations with straightforward Type II errors. Furthermore, our extensive experiments (based on over 200TB of random bits generated) show that NIST SP800-22 techniques could not detect the weakness in the above mentioned pseudorandom generators.

In order to address the challenges faced by NIST SP800-22, this paper designs a "behavioristic" testing approach which is based on statistical distances. Based on this approach, the details of LIL testing techniques are developed. As an example, we carried out LIL testing on the flawed Debian Linux (CVE-2008-0166) pseudorandom generator based on OpenSSL 0.9.8c-1 and on the standard C linear congruential generator. As we expected, both of these pseudorandom generators failed the LIL testing since we know that the sequences produced by these two generators are strongly predictable. However, as we have mentined earlier, our experiments show that the sequences produced by these two generators pass the NIST SP800-22 test suite using the recommended parameters. In other words, NIST SP800-22 test suite with the recommended parameters has no capability in detecting these known deviations from randomness. Furthermore, it is shown that for several pseudorandom generators (e.g., the linear congruential generator), the LIL test results on output strings start off fine but deteriorate as the string length increases beyond that which NIST can handle since NIST testing tool package has an integer overflow issue.

The paper is organized as follows. Section 2 introduces notations. Section 3 points out the limitation of NIST SP800-22 testing tools. Section 4 discusses the law of iterated logarithm (LIL). Section 5 reviews the normal approximation to binomial distributions. Section 6 introduces statistical distance based LIL tests. Section 7 reports experimental results, and Section 8 contains general discussions on OpenSSL random generators.

2 Notations and Pseudorandom Generators

In this paper, N and R^+ denotes the set of natural numbers (starting from 0) and the set of non-negative real numbers, respectively. $\Sigma = \{0,1\}$ is the binary alphabet, Σ^* is the set of (finite) binary strings, Σ^n is the set of binary strings of length n, and Σ^∞ is the set of infinite binary sequences. The length of a string x is denoted by $|x|$. For strings $x, y \in \Sigma^*$, xy is the concatenation of x and y, $x \sqsubseteq y$ denotes that x is an initial segment of y. For a sequence $x \in \Sigma^* \cup \Sigma^\infty$ and a natural number $n \geq 0$, $x\restriction n = x[0..n-1]$ denotes the initial segment of length n of x ($x\restriction n = x[0..n-1] = x$ if $|x| \leq n$) while $x[n]$ denotes the nth bit of x, i.e., $x[0..n-1] = x[0]\ldots x[n-1]$.

The concept of "effective similarity" (see, e.g., Wang [14]) is defined as follows: Let $X = \{X_n\}_{n\in N}$ and $Y = \{Y_n\}_{n\in N}$ be two probability ensembles such that each of X_n and Y_n is a distribution over Σ^n. We say that X and Y are computationally (or statistically) indistinguishable if for every feasible algorithm A (or every algorithm A), the total variation difference between X_n and Y_n is a negligible function in n.

Let $l : N \to N$ with $l(n) \geq n$ for all $n \in N$ and G be a polynomial-time computable algorithm such that $|G(x)| = l(|x|)$ for all $x \in \Sigma^*$. Then G is a polynomial-time pseudorandom generator if the ensembles $\{G(U_n)\}_{n\in N}$ and $\{U_{l(n)}\}_{n\in N}$ are computationally indistinguishable.

3 Limitations of NIST SP800-22

In this section, we show that NIST SP800-22 test suite has inherent limitations with straightforward Type II errors. Our first example is based on the following observation: for a function F that mainly outputs "random strings" but, with probability α, outputs biased strings (e.g., strings consisting mainly of 0's), F will be considered as a "good" pseudorandom generator by NIST SP800-22 test though the output of F could be distinguished from the uniform distribution (thus, F is not a pseudorandom generator by definition). Let $\mathrm{RAND}_{c,n}$ be the sets of Kolmogorov c-random binary strings of length n, where $c \geq 1$. That is, for a universal Turing machine M, let

$$\mathrm{RAND}_{c,n} = \{x \in \{0,1\}^n : \text{ if } M(y) = x \text{ then } |y| \geq |x| - c\}. \tag{1}$$

Let α be a given significance level of NIST SP800-22 test and $\mathcal{R}_{2n} = \mathcal{R}_1^n \cup \mathcal{R}_2^n$ where

$$\mathcal{R}_1^n \subset \mathrm{RAND}_{2,2n} \text{ and } |\mathcal{R}_1^n| = 2^n(1-\alpha)$$
$$\mathcal{R}_2^n \subset \{0^n x : x \in \{0,1\}^n\} \text{ and } |\mathcal{R}_2^n| = 2^n\alpha.$$

Furthermore, let $f_n : \{0,1\}^n \to \mathcal{R}_{2n}$ be an ensemble of random functions (not necessarily computable) such that $f(x)$ is chosen uniformly at random from $\mathcal{R}_{2n}$. Then for each n-bit string x, with probability $1-\alpha$, $f_n(x)$ is Kolmogorov 2-random and with probability α, $f_n(x) \in \mathcal{R}_2^n$. Since all Kolmogorov 2-random strings are guaranteed to pass NIST SP800-22 test at significance level α (otherwise, they are not Kolmogorov 2-random by definition) and all strings in $\mathcal{R}_2^n$ fail NIST SP800-22 test at significance level α for large enough n, the function ensemble $\{f_n\}_{n\in N}$ is considered as a "good" pseudorandom generator by NIST SP800-22 test suite. On the other hand, a similar proof as in Wang [14]

can be used to show that that $\mathcal{R}_{2n}$ could be efficiently distinguished from the uniform distribution with a non-negligible probability. Thus $\{f_n\}_{n\in N}$ is not a cryptographically secure pseudorandom generator.

The above example shows the limitation of testing approaches specified in NIST SP800-22. The limitation is mainly due to the fact that NIST SP800-22 does not fully realize the differences between the two common approaches to pseudorandomness definitions as observed and analyzed in Wang [14]. In other words, the definition of pseudorandom generators is based on the indistinguishability concepts though techniques in NIST SP800-22 mainly concentrate on the performance of individual strings. In this paper, we propose testing techniques that are based on statistical distances such as root-mean-square deviation or Hellinger distance. The statistical distance based approach is more accurate in deviation detection and avoids above type II errors in NIST SP800-22. Our approach is illustrated using the LIL test design.

4 Stochastic Properties of Long Pseudorandom Sequences

The law of the iterated logarithm (LIL) describes the fluctuation scales of a random walk. For a nonempty string $x \in \Sigma^*$, let

$$S(x) = \sum_{i=0}^{|x|-1} x[i] \quad \text{and} \quad S^*(x) = \frac{2 \cdot S(x) - |x|}{\sqrt{|x|}}$$

where $S(x)$ denotes the *number* of 1s in x and $S^*(x)$ denotes the *reduced number* of 1s in x. $S^*(x)$ amounts to measuring the deviations of $S(x)$ from $\frac{|x|}{2}$ in units of $\frac{1}{2}\sqrt{|x|}$.

The law of large numbers states that, for a pseudo random sequence ξ, the limit of $\frac{S(\xi\restriction n)}{n}$ is $\frac{1}{2}$, which corresponds to the frequency (Monobit) test in NIST SP800-22 [12]. But it states nothing about the reduced deviation $S^*(\xi\restriction n)$. It is intuitively clear that, for a pseudorandom sequence ξ, $S^*(\xi\restriction n)$ will sooner or later take on arbitrary large values (though slowly). The law of the iterated logarithm (LIL), which was first discovered by Khinchin [7], gives an optimal upper bound $\sqrt{2\ln\ln n}$ for the fluctuations of $S^*(\xi\restriction n)$. It is shown in [13] that p-random sequences satisfy common statistical laws such as the law of the iterated logarithm. Thus it is reasonable to expect that pseudorandom sequences produced by pseudorandom generators satisfy these laws also.

5 Normal Approximations to S_{lil}

In this section, we provide several results on normal approximations to the function $S_{lil}(\cdot)$ that will be used in following sections. The DeMoivre-Laplace theorem is a normal approximation to the binomial distribution, which states that the number of "successes" in n independent coin flips with head probability $1/2$ is approximately a normal distribution with mean $n/2$ and standard deviation $\sqrt{n}/2$. We first review a few classical results on the normal approximation to the binomial distribution.

Definition 1. *The normal density function with mean μ and variance σ is defined as*

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}; \tag{2}$$

For $\mu = 0$ and $\sigma = 1$, we have the standard normal density function $\varphi(x)$ and the standard normal distribution function $\Phi(x)$:

$$\varphi(x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}} \quad \text{and} \quad \Phi(x) = \int_{-\infty}^{x} \varphi(y)dy \tag{3}$$

The following DeMoivre-Laplace limit theorem is derived from the approximation theorem on page 181 of [4].

Theorem 1. *For fixed x_1, x_2, we have*

$$\lim_{n\to\infty} Prob\left[x_1 \le S^*(\xi\restriction n) \le x_2\right] = \Phi(x_2) - \Phi(x_1). \tag{4}$$

The growth speed for the above approximation is bounded by $\max\{k^2/n^2, k^4/n^3\}$ *where* $k = S(\xi\restriction n) - \frac{n}{2}$.

In this paper, we only consider tests for $n \ge 2^{26}$ and $x_2 \le 1$. That is, $S^*(\xi\restriction n) \le \sqrt{2\ln\ln n}$. Thus $k = S(\xi\restriction n) - \frac{n}{2} \simeq \frac{\sqrt{n}}{2} S^*(\xi\restriction n) \le \sqrt{2n\ln\ln n}/2$. Hence, we have $\max\left\{\frac{k^2}{n^2}, \frac{k^4}{n^3}\right\} = \frac{k^2}{n^2} = \frac{(1-\alpha)^2 \ln\ln n}{2n} < 2^{-22}$

By Theorem 1, the approximation probability calculation errors in this paper will be less than 2^{-22} which is negligible. Unless stated otherwise, we will not mention the approximation errors in the remainder of this paper.

6 Snapshot LIL Tests and Random Generator Evaluation

The distribution S_{lil} defines a probability measure on the real line R. Let $\mathcal{R} \subset \Sigma^n$ be a set of m sequences with a standard probability definition on it. That is, for each $x_0 \in \mathcal{R}$, let $Prob[x = x_0] = \frac{1}{m}$. Then each set $\mathcal{R} \subset \Sigma^n$ induces a probability measure $\mu_n^{\mathcal{R}}$ on R by letting

$$\mu_n^{\mathcal{R}}(I) = Prob\left[S_{lil}(x) \in I, x \in \mathcal{R}\right]$$

for each Lebesgue measurable set I on R. For $U = \Sigma^n$, we use μ_n^U to denote the corresponding probability measure induced by the uniform distribution. By definition, if $\mathcal{R}_n$ is the collection of all length n sequences generated by a pseudorandom generator, then the difference between μ_n^U and $\mu_n^{\mathcal{R}_n}$ is negligible.

By Theorem 1, for a uniformly chosen ξ, the distribution of $S^*(\xi\restriction n)$ could be approximated by a normal distribution of mean 0 and variance 1, with error bounded by $\frac{1}{n}$ (see [4]). In other words, the measure μ_n^U can be calculated as

$$\mu_n^U((-\infty, x]) \simeq \Phi(x\sqrt{2\ln\ln n}) = \sqrt{2\ln\ln n}\int_{-\infty}^{x} \phi(y\sqrt{2\ln\ln n})dy. \tag{5}$$

Figure 1 shows the distributions of μ_n^U for $n = 2^{26}, \cdots, 2^{34}$. For the reason of convenience, in the remaining part of this paper, we will use $\mathcal{B}$ as the discrete partition of the real line R defined by

$$\{(\infty, 1), [1, \infty)\} \cup \{[0.05x - 1, 0.05x - 0.95) : 0 \le x \le 39\}.$$

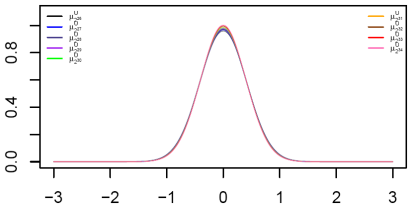

Fig. 1. Density functions for distributions μ_n^U with $n = 2^{26}, \cdots, 2^{34}$

Table 1. The distribution μ_n^U induced by S_{lil} for $n = 2^{26}, \cdots, 2^{34}$ (due to symmetry, only distribution on the positive part of real line R is given)

	2^{26}	2^{27}	2^{28}	2^{29}	2^{30}	2^{31}	2^{32}	2^{33}	2^{34}
$[0.00, 0.05)$	.047854	.048164	.048460	.048745	.049018	.049281	.049534	.049778	.050013
$[0.05, 0.10)$	.047168	.047464	.047748	.048020	.048281	.048532	.048773	.049006	.049230
$[0.10, 0.15)$	.045825	.046096	.046354	0.04660	.046839	.047067	.047287	.047498	.047701
$[0.15, 0.20)$	.043882	.044116	.044340	.044553	.044758	.044953	.045141	.045322	.045496
$[0.20, 0.25)$	.041419	.041609	.041789	.041961	.042125	.042282	.042432	.042575	.042713
$[0.25, 0.30)$	.038534	.038674	.038807	.038932	.039051	.039164	.039272	.039375	.039473
$[0.30, 0.35)$	.035336	.035424	.035507	.035584	.035657	.035725	0.03579	.035850	.035907
$[0.35, 0.40)$	.031939	.031976	.032010	.032041	.032068	.032093	.032115	.032135	.032153
$[0.40, 0.45)$	.028454	.028445	.028434	.028421	.028407	.028392	.028375	.028358	.028340
$[0.45, 0.50)$	.024986	.024936	.024886	.024835	.024785	.024735	.024686	.024637	.024588
$[0.50, 0.55)$	.021627	.021542	.021460	.021379	.021300	.021222	.021146	.021072	.020999
$[0.55, 0.60)$	.018450	.018340	.018234	.018130	.018029	.017931	.017836	.017743	.017653
$[0.60, 0.65)$	.015515	.015388	.015265	.015146	.015032	.014921	.014813	.014709	.014608
$[0.65, 0.70)$	.012859	.012723	.012591	.012465	.012344	.012227	.012114	.012004	.011899
$[0.70, 0.75)$	.010506	.010367	.010234	.010106	.009984	.009867	.009754	.009645	.009541
$[0.75, 0.80)$	.008460	.008324	.008195	.008072	.007954	.007841	.007733	.007629	.007530
$[0.80, 0.85)$	.006714	.006587	.006466	.006351	.006241	.006137	.006037	.005941	.005850
$[0.85, 0.90)$	.005253	.005137	.005027	.004923	.004824	.004730	.004640	.004555	.004474
$[0.90, 0.95)$	.004050	.003948	.003851	.003759	.003672	.003590	.003512	.003438	.003368
$[0.95, 1.00)$	.003079	.002990	.002906	.002828	.002754	.002684	.002617	.002555	.002495
$[1.00, \infty)$	.008090	.007750	.007437	.007147	.006877	.006627	.006393	.006175	.005970

With this partition, Table 1 lists values $\mu_n^U(I)$ on $\mathcal{B}$ with $n = 2^{26}, \cdots, 2^{34}$. Since $\mu_n^U(I)$ is symmetric, it is sufficient to list the distribution in the positive side of the real line.

In order to evaluate a pseudorandom generator G, first choose a list of testing points $n_0, \cdots, n_t$ (e.g., $n_0 = 2^{26+t}$). Secondly use G to generate a set $\mathcal{R} \subseteq \Sigma^{n_t}$ of m sequences. Lastly compare the distances between the two probability measures $\mu_n^{\mathcal{R}}$ and μ_n^U for $n = n_0, \cdots, n_t$.

A generator G is considered "good", if for sufficiently large m (e.g., $m \geq 10,000$), the distances between $\mu_n^{\mathcal{R}}$ and μ_n^U are negligible (or smaller than a given threshold). There are various definitions of statistical distances for probability measures. In our analysis, we will consider the total variation distance [2]

$$d(\mu_n^{\mathcal{R}}, \mu_n^U) = \sup_{A \subseteq \mathcal{B}} \left| \mu_n^{\mathcal{R}}(A) - \mu_n^U(A) \right| \tag{6}$$

Hellinger distance [5]

$$H(\mu_n^{\mathcal{R}} \| \mu_n^U) = \frac{1}{\sqrt{2}} \sqrt{\sum_{A \in \mathcal{B}} \left(\sqrt{\mu_n^{\mathcal{R}}(A)} - \sqrt{\mu_n^U(A)} \right)^2} \tag{7}$$

and the root-mean-square deviation

$$\mathrm{RMSD}(\mu_n^{\mathcal{R}}, \mu_n^U) = \sqrt{\frac{\sum_{A \in \mathcal{B}} \left(\mu_n^{\mathcal{R}}(A) - \mu_n^U(A) \right)^2}{|\mathcal{B}|}}. \tag{8}$$

7 Experimental Results

As an example to illustrate the importance of LIL tests, we carried out both NIST SP800-22 tests [9] and LIL tests on the following commonly used pseudorandom bit generators: The standard C linear congruential generator, Mersenne Twister generators, PHP web server random bit generators (both MT and LCG), and Debian (CVE-2008-0166) random bit generator with OpenSSL 0.9.8c-1. Among these generators, linear congruential generators and Debian Linux (CVE-2008-0166) pseudorandom generators are not cryptographically strong. Thus they should fail a good statistical test. As we expected, both of these generators failed LIL tests. However, neither of these generators failed NIST SP800-22 tests which shows the limitation of NIST SP800-22 test suite.

It should be noted that NIST SP800-22 test suite [9] checks the first 1,215,752,192 bits (≃145MB) of a given sequence since the software uses 4-byte `int` data type for integer variables only. For NIST SP800-22 tests, we used the parameter $\alpha = 0.01$ for all experiments. For each pseudorandom generator, we generated $10,000 \times 2$GB sequences. The results, analysis, and comparisons are presented in the following sections.

7.1 The Standard C Linear Congruential Generator

A linear congruential generator (LCG) is defined by the recurrence relation

$$X_{n+1} = aX_n + c \mod m$$

where X_n is the sequence of pseudorandom values, m is the modulus, and $a, c < m$. For any initial seeding value X_0, the generated pseudorandom sequence is $\xi = X_0 X_1 \cdots$ where X_i is the binary representation of the integer X_i.

Linear congruential generators (LCG) have been included in various programming languages. For example, C and C++ functions drand48(), jrand48(), mrand48(), and rand48() produce uniformly distributed random numbers on Borland C/C++ rand() function returns the 16 to 30 bits of

$$X_{n+1} = \texttt{0x343FD} \cdot X_n + \texttt{0x269EC3} \mod 2^{32}.$$

LCG is also implemented in Microsoft Visual Studio, Java.Util.Random class, Borland Delphi, and PHP. In our experiments, we tested the standard linear congruential generator used in Microsoft Visual Studio.

In our experiments, we generated $10,000 \times 2$GB sequences by calling Microsoft Visual Studio stdlib function rand() which uses the standard C linear congruential generator. Each sequence is generated with a 4-byte seed from www.random.org [11]. For the $10,000 \times 2$GB sequences, we used a total of $10,000 \times 4$-byte seeds from www.random.org. The rand() function returns a 15-bit integer in the range $[0, 0x7FFF]$ each time. Since LCG outputs tend to be correlated in the lower bits, we shift the returned 15 bits right by 7 positions. In other words, for each rand() call, we only use the most significant 8 bits. This is a common approach that most programmers will do to offset low bit correlation and missing most significant bits (MSB).

Since linear congruential generator is predictable and not cryptographically strong, we expected that these 10,000 sequences should fail both NIST SP800-22 tests and LIL tests. To our surprise, the collection of 10,000 sequences passed NIST SP800-22 [9] testing with the recommended parameters. Specifically, for the randomly selected 10 sequences, all except one of the 150 non-overlapping template tests passed the NIST test (pass ratio = 0.965). In other words, these sequences are considered as random by NIST SP800-22 testing standards. On the other hand, these sequences failed LIL tests as described in the following.

Table 2. Total variation and Hellinger distances for Standard C LCG

n	2^{26}	2^{27}	2^{28}	2^{29}	2^{30}	2^{31}	2^{32}	2^{33}	2^{34}
d	.061	.097	.113	.156	.176	.261	.324	.499	.900
H	.064	.088	.126	.167	.185	.284	.387	.529	.828
RMSD	.004	.006	.008	.010	.011	.017	.021	.031	.011

Based on snapshot LIL tests at points $2^{26}, \cdots, 2^{34}$, the corresponding total variation distance $d(\mu_n^{cLCG}, \mu_n^U)$, Hellinger distance $H(\mu_n^{cLCG} \| \mu_n^U)$, and the root-mean-square deviation $\text{RMSD}(\mu_n^{cLCG}, \mu_n^U)$ at sample size 1000 are calculated and shown in Table 2. It is observed that at the sample size 1000, the average distance between μ_n^{cLCG} and μ_n^U is larger than 0.10 and the root-mean-square deviation is large than 0.01. It is clear that this sequence collection is far away from the true random source.

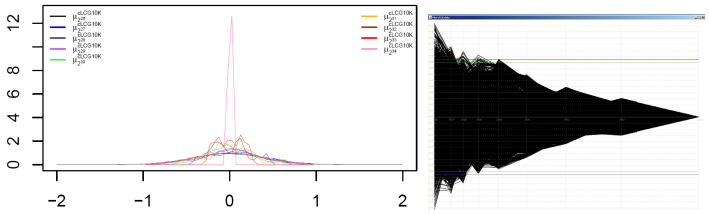

Fig. 2. Density functions for distributions μ_n^{cLCG} with $n = 2^{26}, \cdots, 2^{34}$ (first) and LIL curves for the standard C LCG (second) for 10,000×2GB strings

The first picture in Figure 2 shows that the distributions of μ_n^{cLCG} for $n = 2^{26}, \cdots, 2^{34}$ are far away from the expected distribution in Figure 1. Furthermore, the second picture in 2 shows the LIL-test result curves for the 10,000 sequences. For a good random bit generator, the LIL curves should be distributed within the y-axis interval $[-1, 1]$

through the entire x-axis according to the normal distribution. For example, a good curve should look like the third picture in the following Figure 3. However, LIL curves for the standard C LCG generated sequences in the second picture of Figure 2 start reasonably well but deteriorate as the string length increases.

7.2 Mersenne Twister Generators

Mersenne Twister (MT) is a pseudorandom generator designed by Matsumoto and Nishimura [8] and it is included in numerous software packages such as R, Python, PHP, Maple, ANSI/ISO C++, SPSS, SAS, and many others. The commonly used Mersenne Twister MT19937 is based on the Mersenne prime $2^{19937} - 1$ and has a long period of $2^{19937} - 1$. The Mersenne Twister is sensitive to the seed value. For example, too many zeros in the seed can lead to the production of many zeros in the output and if the seed contains correlations then the output may also display correlations.

In order to describe the pseudorandom bit generation process MT19937, we first describe the tempering transform function $t(x)$ on 32-bit strings. For $x \in \Sigma^{32}$, $t(x)$ is defined by

$$\begin{aligned} y_1 &:= x \oplus (x >> 11) \\ y_2 &:= y_1 \oplus ((y_1 << 7) \text{ AND } 0\text{x}9\text{D}2\text{C}5680) \\ y_3 &:= y_2 \oplus ((y_2 << 15) \text{ AND } 0\text{xEFC60000}) \\ t(x) &:= y_3 \oplus (y_3 >> 18) \end{aligned}$$

Let $x_0, x_2, \cdots, x_{623} \in \Sigma^{32}$ be $32 \times 624 = 19968$ bits seeding values for the MT19937 pseudorandom generator. Then the MT19937 output is the sequence $t(x_{624})t(x_{625})t(x_{626})$ $\cdots$ where for $k = 0, 1, 2, 3, \cdots$, we have $x_{624+k} = x_{397+k} \oplus (x_k[0]x_{k+1}[1..31])A$ and A is the 32×32 matrix

$$A = \begin{pmatrix} 0 & I_{31} \\ a_{31} & (a_{30}, \cdots, a_0) \end{pmatrix}$$

with $a_{31}a_{30} \cdots a_0 = 0\text{x}9908\text{B}0\text{DF}$. For a 32 bit string x, xA is interpreted as multiplying the 32 bit vector x by matrix A from the right hand side.

Using the source code provided in Matsumoto and Nishimura [8], we generated $10,000 \times 2$GB sequences. The collection of these sequences passed NIST SP800-22 [9] test with the recommended parameters. The following discussion shows that these sequences have very good performance in LIL testing also. Thus we can consider these sequences passed the LIL test.

Based on snapshot LIL tests at points $2^{26}, \cdots, 2^{34}$, the corresponding total variation distance $d(\mu_n^{MT19937}, \mu_n^U)$, Hellinger distance $H(\mu_n^{MT19937} || \mu_n^U)$, and the root-mean-square deviation $\text{RMSD}(\mu_n^{MT19937}, \mu_n^U)$ at sample size 1,000 (resp. 10,000) are calculated and shown in Table 3. In Table 3, the subscript 1 is for sample size 1,000 and the subscript 2 is for sample size 10,000.

Figure 3 shows the distributions of $\mu_n^{MT19937}$ for $n = 2^{26}, \cdots, 2^{34}$ where the curves are plotted on top of the expected distribution in Figure 1. Furthermore, the third picture in Figure 3 shows the LIL-test result curves for the 10,000 sequences. The plot in the third picture of Figure 3 is close to what we are expecting for a random source.

Table 3. Total variation and Hellinger distances for MT19937

n	2^{26}	2^{27}	2^{28}	2^{29}	2^{30}	2^{31}	2^{32}	2^{33}	2^{34}
d_1	.057	.068	.084	.068	.063	.075	.073	.079	.094
H_1	.056	.077	.072	.069	.065	.083	.074	.080	.081
RMSD_1	.004	.004	.005	.004	.004	.005	.005	.005	.006
d_2	.023	.025	.026	.021	.020	.025	.026	.027	.020
H_2	.022	.022	.024	.021	.021	.026	.024	.023	.020
RMSD_2	.001	.002	.002	.001	.001	.002	.002	.002	.001

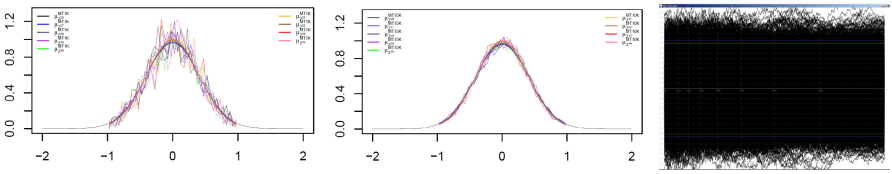

Fig. 3. Density functions for distributions $\mu_n^{MT19937}$ at $n = 2^{26}, \cdots, 2^{34}$ with 1000 (first) and 10,000 (second) strings and LIL plot for Mersenne Twister MT19937 with 10,000×2GB strings (third)

7.3 PHP Web Server Random Bit Generators

PHP is a server side processing language and its random number generator is very important for guaranteeing Web server security. In the experiments, we installed an Apache web server together with PHP v5.3.5. By default, PHP supports rand(), which is a linear congruential random bit generator, and m_rand() which is a Mersenne Twister random bit generator. The random bit outputs from these two generators are tested in the experiments. By modifying php.ini script in PHP 5.3, one may also use the OpenSSL pseudorandom generator via the openssl_random_pseudo_bytes() function call.

PHP Mersenne Twister. In Section 7.2, we showed that the output of the correctly implemented Mersenne Twister pseudorandom generators has very good performance and passes both the NIST and LIL testing. However, if the Mersenne Twister in PHP implementation is not properly post-processed, it generates completely non-random outputs. This is illustrated by our experiments on the PHP Mersenne Twister implementation.

Since the PHP server script is slow in generating a large amount of pseudorandom bits, we only generated 6×2GB random bit strings from hte PHP Mersenne Twister m_rand() function call. It is estimated to take 2 years for our computer to generate $10,000 \times 2$GB random bit strings since each 2GB sequence takes 90 minutes to generate.

As discussed earlier, it is expected that LIL values stay within the interval $[-1, 1]$. However, LIL curves for the 6 PHP MT generated sequences display a range from 0 to -2000. This indicates that these sequences are overwhelmed with zeros which get worse as the sequence gets longer.

By checking the rand.c code in PHP 5.3.27, it seems that programmers are prepared to make arbitrary changes with arbitrary post-processing. In particular, for the PHP Mersenne Twister, it will output an integer in the range $[0, 0x7FFFFFFF]$ each time while the source code in Matsumoto and Nishimura [8] that we used in Section 7.2

outputs an integer in the range $[0, 0xFFFFFFFF]$ each time. This difference is not realized by some PHP implementers as illustrated in the following comments of PHP rand.c. Thus it is important to use the LIL test to detect these weak implementations.

```
/* Melo: hmms.. randomMT() returns 32 random bits...
 * Yet, the previous php_rand only returns 31 at most.
 * So I put a right shift to loose the lsb. It *seems*
 * better than clearing the msb.
 * Update:
 * I talked with Cokus via email and it won't ruin
 * the algorithm */
```

The experiments show that all of 6 PHP Mersenne Twister generated sequences fail NIST SP800-22 tests, illustrating the effect of users not accommodating the limitations of the PHP 31 bit implementation.

PHP Linear Congruential Generator. Since it is slow to generate a large amount of random bits using PHP script, we only generated 6×2GB sequences using the PHP rand() function call (similarly, it is estimated to take 2 years for our computer to generate $10,000 \times 2$GB random bits). All of the sequences have similar LIL curves as shown in the first picture of Figure 4. The second picture in Figure 4 shows that the distributions of μ_n^{phpLCG} at $n = 2^{26}, \cdots, 2^{34}$ are far away from the expected distribution in Figure 1. One may also compare the second picture in Figure 4 against the density distributions by the standard C linear congruential generator in Figure 2. In summary, the PHP implementation of the linear congruential generator comprehensively failed NIST and LIL tests.

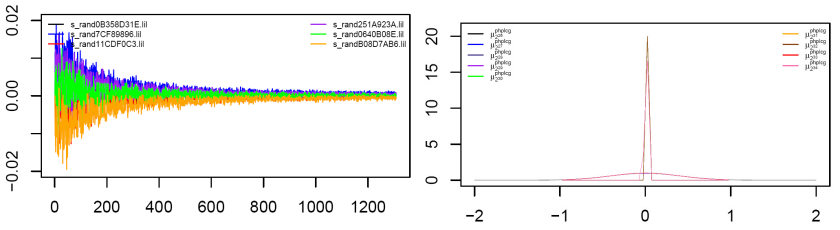

Fig. 4. LIL curves for PHP LCG generated sequences (first) and density functions for distributions μ_n^{phpLCG} (second) of 6×2GB PHP LCG sequences with $n = 2^{26}, \cdots, 2^{34}$

7.4 Flawed Debian's OpenSSL package

It is reported in Debian Security Advisory DSA-1571-1 [3] that the random number generator in Debian Linux (CVE-2008-0166) pseudorandom generator based on OpenSSL 0.9.8c-1 is predictable since the following line of code in `md_rand.c` has been removed by one of its implementors.

```
MD_Update(&m, buf, j); /*purify complains*/
```

Note that the code `MD_Update(&m,buf,j)` is responsible for adding the entropy into the state that is passed to the random bit generation process from the main seeding function. By commenting out this line of codes, the generator will have small number of states which will be predictable.

We generated $10,000 \times 2$GB sequences using this version of the flawed Debian OpenSSL with multi-threads (the single thread results are much worse). The snapshot LIL test result for this flawed Debian OpenSSL implementation is shown in Figure 5, where the first picture is for the sample size of 1,000 and the second picture is for the sample size of 10,000. In particular, Figure 5 shows the distributions of μ_n^{Debian} for $n = 2^{26}, \cdots, 2^{34}$ where the curves are plotted on top of the expected distribution in Figure 1. As a comparison, we carried out snapshot LIL test on the standard OpenSSL pseudorandom generator [10]. We generated $10,000 \times 2$GB sequences using the standard implementation of OpenSSL (with single thread). The snapshot LIL test result for this standard OpenSSL implementation is shown in Figure 6, where the first picture is for the sample size of 1,000 and the second picture is for the sample size of 10,000. In particular, Figure 6 shows the distributions of $\mu_n^{OpenSSL}$ for $n = 2^{26}, \cdots, 2^{34}$ where the curves are plotted on top of the expected distribution in Figure 1.

The results in Figures 5 and 6 indicate that the flawed Debian pseudorandom generator has a very large statistical distance from the uniform distribution while the standard OpenSSL pseudorandom generator has a smaller statistical distance from the uniform distribution. In other words, statistical distance based LIL tests could be used to detect such kinds of implementation weakness conveniently.

While the Debian Linux implementation of openSSL pseudorandom generator fails the LIL test obviously, the experiments show that the collection of the 10,000 sequences passed the NIST SP800-22 testing with the recommended parameters.

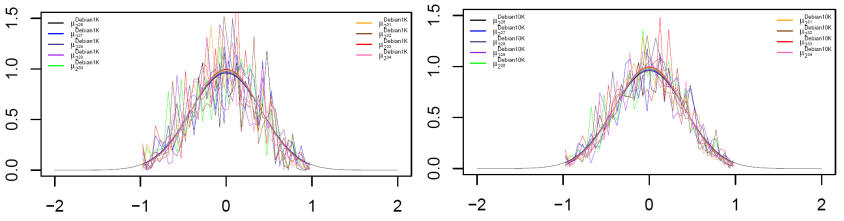

Fig. 5. Density functions for distributions μ_n^{Debian} with $n = 2^{26}, \cdots, 2^{34}$

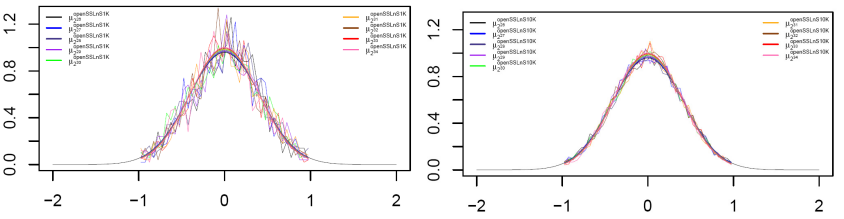

Fig. 6. Density functions for distributions $\mu_n^{OpenSSL}$ with $n = 2^{26}, \cdots, 2^{34}$

7.5 Summary of Experiments

As a summary, Table 4 lists the results of both NIST SP800-22 testing and LIL testing on commonly used pseudorandom generators. In the table, we listed the expected testing results for MT19937 as "pass" since MT19937 was designed to be k-distributed to 32-bit accuracy for every $1 \leq k \leq 623$. In other words, the output of MT19937 is uniformly distributed and should pass all statistical tests even though the output is not cryptographically strong. The results in Table 4 show that the LIL testing techniques always produce expected results while NIST SP800-22 test suite does not.

Table 4. NIST SP800-22 and LIL testing results

Generator	NIST SP800-22	LIL	expected result
Standard C LCG	pass	fail	fail
MT19937	pass	pass	pass
PHP LCG	fail	fail	fail
PHP MT19937	fail	fail	fail
flawed Debian openSSL	pass	fail	fail
standard openSSL	pass	pass	pass

8 General Discussion on OpenSSL Random Generators

It is noted in [1] that the serious flaws in Debian OpenSSL had not been noticed for more than 2 years. A key contributor to this problem was the lack of documentation and poor source code commenting of OpenSSL making it very difficult for a maintainer to understand the consequences of a change to the code. This section provides an analysis of the OpenSSL default RNG. We hope this kind of documentation will help the community to improve the quality of OpenSSL implementations.

Figure 7 illustrates the architecture of the OpenSSL RNG. It consists of a 1023 byte circular array named `state` which is the entropy pool from which random numbers are created. `state` and some other global variables are accessible from all threads. Crypto locks protect the global data from thread contention except for the update of `state` as this improves performance. Locked access, direct access to data from threads, and the mapping of global to local variables (e.g., `state_num` to `st_num, md` to `local_md`) are illustrated in Figure 7.

`state` is the entropy pool that is a declared array of of 1023+ `MD_DIGEST_SIZE` bytes. However the RNG algorithm only uses `state[0..1022]` in a circular manner. There are two index markers `state_num` and `state_index` on `state` which mark the region of `state` to be accessed during reads or updates. `md` is the global message digest produced by the chosen one-way hash function which defaults to SHA1 making `MD_DIGEST_LENGTH = 20`. `md` is used and updated by each thread as it seeds the RNG.

Each thread maintains a count of the number of message digest blocks used during seeding. This counter is copied to the global `md_count` enabling other threads to read it as another entropy source. The global variable `entropy` records the entropy level of the entropy pool. This value is checked when generating random numbers to ensure they are based on sufficient entropy. `initialized` is a global flag to indicating seed status. If not initialized, entropy collection and seeding functions are called.

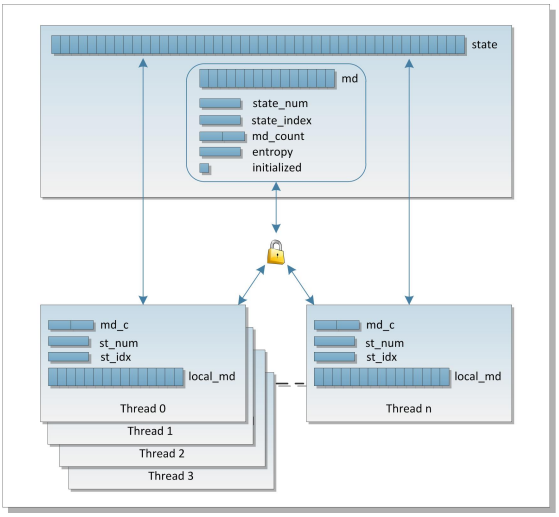

Fig. 7. High Level view of OpenSSL RNG

8.1 OpenSSL Entropy Collection

Entropy data is required to seed the RNG. OpenSSL caters for a number of entropy sources ranging from its default source through to third party random bit generators. This section discusses the OpenSSL library-supplied entropy collection process. Once entropy data is collected, it is passed to `ssleay_rand_add` or `ssleay_rand_seed` to be added into the RNG's entropy pool.

`RAND_poll` is the key entropy collection function. Default entropy data sources for Windows installations are illustrated in Figure 8. A check is made to determine the operating system and if Windows 32 bit, ADVAPI32.DLL, KERNEL32.DLL and NETAPI32.DLL are loaded. These libraries include Windows crypto, OS, and network functions. Following is an overview of the default entropy collection process.

1. Collect network data using `netstatget(NULL, L"LanmanWorkstation", 0, 0, &outbuf)`. By using `LanmanWorkstation`, it returns a `STAT_WORKSTATION_0` structure in `outbuf` containing 45 fields of data including: time of stat collection, number of bytes received and sent on LAN, number of bytes read from and written to disk etc. Each field is estimated as 1 byte of entropy. `netstatget` is also called with `LanmanServer` to obtain another 17 bytes of entropy in `STAT_SERVER_0`.
2. Collect random data from the cryptographic service provided by ADVAPI32. Use the default cryptographic service provider in `hProvider` to call `CryptGenRandom` and obtain 64 bytes of random data in `buff`. the `RAND_add` function is passed 0 as the entropy estimate despite this data coming from an SHA-based crypto RNG so presumably the OpenSSL programmer does not trust this source. An attempt is made to access the processor's on-chip RNG and if successful 64 bytes of random data are passed to `RAND_add` with a 100% entropy value.

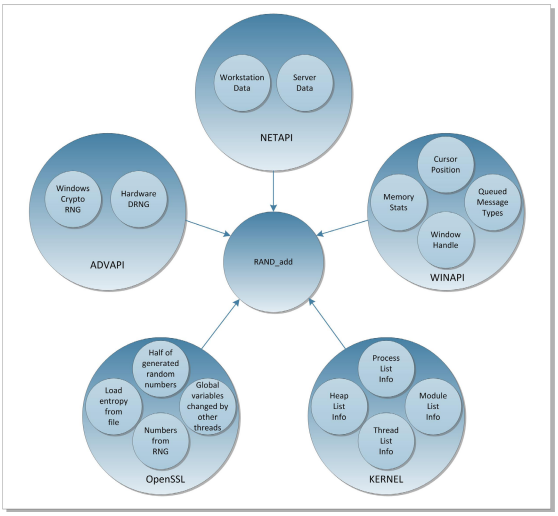

Fig. 8. OpenSSL entropy sources on Windows

3. Get entropy data from Windows message queue, 4-byte foreground window handle, and 2-byte cursor position. However, dynamically tracing these operations identified an OpenSSL coding error discussed in Section 8.2.
4. Get kernel-based entropy data by taking a snapshot of the heap status then walking the heap collecting entropy from each entry. Similarly walk the process list, thread list and module list. The depth that each of the four lists is traversed is determined as follows: the heap-walk continues while there is another entry and either the `good` flag is false OR a timeout has not expired AND the number or iterations has not exceeded a max count. This ensures loop termination in a reasonable time. However, setting the `good` flag is suspicious as it is set if random data is retrieved from the Microsoft crypto library or from the hardware DRNG. This is odd as zero was assigned as the entropy value for the crypto library numbers and data from the DRNG may be unavailable yet the `good` flag is still set which limits the amount of kernel data collected.
5. Add the state of global physical and virtual memory. The current process ID is also added to ensure that each thread has something different than the others.

8.2 Potential Bugs in OpenSSL Entropy Collection

```
418.        CURSORINFO ci;
419.        ci.cbSize = sizeof(CURSORINFO);
420.        if (cursor(&ci))
421.            RAND_add(&ci, ci.cbSize, 2);
```

In above OpenSSL code, a static trace implies that all 20 bytes of `CURSOR_INFO` are added into the entropy pool as `ci.cbsize` is set to the size of the `CURSORINFO` structure. The programmer has decided that this data is worth an entropy value of 2 which

is passed to RAND_add. However, a dynamic code trace shows that ci.cbsize is set to zero after the call to cursor(&ci), where cursor is defined as:

```
395.        cursor = (GETCURSORINFO) GetProcAddress(user, "GetCursorInfo");
```

user is the DLL module handle containing function GetCursorInfo. GetCursorInfo that returns true on success and ci.cbsize is initialized to sizeof (CURSORINFO) before the call. However, MSDN does not promise to maintain the fields in this structure on return yet the OpenSSL code relies on it. Our experiments show the ci.cbsize is zero yet is attributed an entropy value of 2.

RAND_add calls ssleay_rand_add. The local variables in ssleay_rand_add are shown in the following.

```
static int ssleay_rand_add(const void *buf, int num, double add)
    {
    int i,j,k,st_idx;
    long md_c[2];
    unsigned char local_md[MD_DIGEST_LENGTH];
    EVP_MD_CTX m;
    int do_not_lock;
```

According to the code, the ssleay_rand_add function increments the global entropy value by 2 if there is not enough current entropy. However, in the Windows environment, the ci.cbsize is always 0 yet it has 2 bytes of entropy added and if timing causes this to happen multiple times due to other threads also incrementing the entropy counter, there could potentially be a situation where there is substantially less entropy than that reported. Specifically, once the entropy threshold of 32 is reached, entropy is no longer updated.

8.3 Seeding the RNG

To seed the RNG, RAND_add is called and the collected entropy data, its length and an entropy estimate are passed in as function parameters. For flexibility, this function is a wrapper for the actual entropy addition function to enable alternatives to be chosen by RAND_get_rand_method so the function binding is dynamic through a pointer to meth->add. RAND_get_rand_method returns the addresses of the preferred functions. For example, it checks for an external device and if not found it returns the address of the default functions in a structure of type RAND_METHOD which holds pointers to the functions. Of the five functions now available, RAND_add() calls meth->add() which in this case (default) points to the physical function ssleay_rand_add. Studying ssleay_rand_add reveals that the entropy data passed to it is hashed directly into the RNG's state.

```
static void ssleay_rand_add(const void *buf,int num,double add)
```

A byte buffer buf of length num containing data, ideally from a good entropy source, is passed to this function to be mixed into the RNG. add is the entropy value of the data in buff estimated by the programmer. For system generated entropy, the value is not calculated but presumably estimated by the OpenSSL developers. RAND_add is available to the caller to add more or better entropy if required. In a summary, Figure 9 describes the seeding flowchart for OpenSSL random number generators.

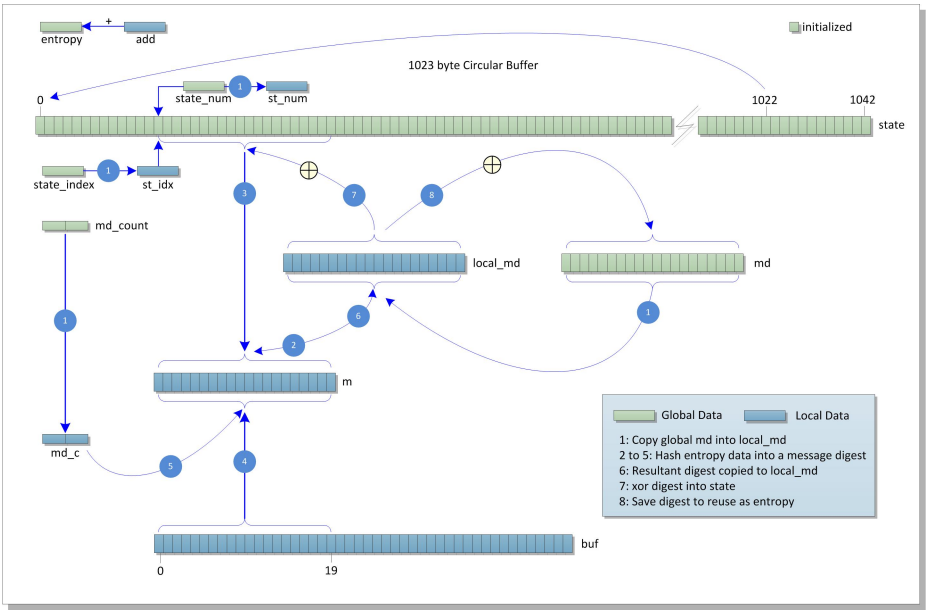

Fig. 9. Seeding the OpenSSL Random Number Generator

OpenSSL provides a second function `ssleay_rand_seed` to seed the RNG, but this simply calls `ssleay_rand_add`, providing the buffer size as the entropy value, i.e., it assumes 100% entropy.

8.4 OpenSSL Documentation Error

If a user requests secure random numbers but the entropy is inadequate, an error message is generated pointing them to: http://www.openssl.org/support/faq.html. The FAQ under "Why do I get a 'PRNG not seeded' error message?" states: "As of version 0.9.5, the OpenSSL functions that need randomness report an error if the random number generator has not been seeded with at least 128 bits of randomness". Yet in the code, entropy is defined in `rand_lcl.h` as 32 (bytes) which is 256 bits.

9 Conclusion

This paper proposed statistical distance based LIL testing techniques. This technique has been used to identify flaws in several commonly used pseudorandom generator implementations that have not been detected by NIST SP800-22 testing tools. It is concluded that the LIL testing technique is an important tool and should be used for statistical testing. We also provided a detailed documentation on OpenSSl random generators and described several potential attacks.

References

1. Ahmad, D.: Two years of broken crypto: debian's dress rehearsal for a global pki compromise. IEEE Security & Privacy 6(5), 70–73 (2008)
2. Clarkson, J.A., Adams, C.R.: On definitions of bounded variation for functions of two variables. Tran. AMS 35(4), 824–854 (1933)
3. Debian. Debian security advisory dsa-1571-1, http://www.debian.org/security/2008/dsa-1571
4. Feller, W.: Introduction to probability theory and its applications, vol. I. John Wiley & Sons, Inc., New York (1968)
5. Hellinger, E.: Neue begründung der theorie quadratischer formen von unendlichvielen veränderlichen. J. für die reine und angewandte Mathematik 136, 210–271 (1909)
6. Heninger, N., Durumeric, Z., Wustrow, E., Halderman, J.A.: Mining your ps and qs: Detection of widespread weak keys in network devices. In: Proc. 21st USENIX Security Symposium, vol. 2 (2012)
7. Khinchin, A.: Über einen satz der wahrscheinlichkeitsrechnung. Fund. Math. 6, 9–20 (1924)
8. Matsumoto, M., Nishimura, T.: Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. ACM TOMACS 8(1), 3–30 (1998)
9. NIST. Test suite (2010), http://csrc.nist.gov/groups/ST/toolkit/rng/
10. OpenSSL. Openssl implementation from http://www.openssl.com/
11. RANDOM.ORG. Random.org, http://www.random.org/
12. Rukhin, A., Soto, J., Nechvatal, J., Smid, M., Barker, E., Leigh, S., Levenson, M., Vangel, M., Banks, D., Heckert, A., Dray, J., Vo, S.: A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications. NIST SP 800-22 (2010)
13. Wang, Y.: Resource bounded randomness and computational complexity. Theoret. Comput. Sci. 237, 33–55 (2000)
14. Wang, Y.: A comparison of two approaches to pseudorandomness. Theoretical computer science 276(1), 449–459 (2002)