Available online at www.sciencedirect.com

ScienceDirect

journal homepage: www.elsevier.com/locate/cose

Computers & Security

On statistical distance based testing of pseudo random sequences and experiments with PHP and Debian OpenSSL☆

Yongge Wang [a,*], *Tony Nicol* [b]

[a] *Dept. SIS, UNC Charlotte, Charlotte, NC 28223, USA*
[b] *University of Liverpool, UK*

ARTICLE INFO

Article history:
Received 10 July 2014
Received in revised form
10 April 2015
Accepted 21 May 2015
Available online 2 June 2015

Keywords:
Statistical testing
Pseudorandom generators
The law of the iterated logarithm
Web security
NIST randomness testing
Statistical distance
OpenSSL
PHP random generator

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 commonly deployed pseudorandom generators such as the standard C linear congruential generator, Mersenne Twister pseudorandom generator, and Debian Linux (CVE-2008-0166) pseudorandom generator with OpenSSL 0.9.8c-1. Based on experimental results, 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.

© 2015 Elsevier Ltd. All rights reserved.

1. Introduction

The weakness in pseudorandom generators could be used to mount a variety of attacks on Internet security. It is reported in the Debian Security Advisory DSA-1571-1 [5] 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., (Ahmad, 2008)). These examples show that it is important to improve the quality of pseudorandom

☆ An extended abstract of this paper appeared in ESORICS 2014.

* *Corresponding author.*
E-mail addresses: yongge.wang@uncc.edu (Y. Wang), tonynicol@inbox.com (T. Nicol).
URL: http://www.sis.uncc.edu/~yonwang/

http://dx.doi.org/10.1016/j.cose.2015.05.005
0167-4048/© 2015 Elsevier Ltd. All rights reserved.

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 (Rukhin et al., 2010) 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 Rukhin et al. (2010), 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 200 TB 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 mentioned 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 distance based LIL testing experimental results, and Section 8 reports some experimental results on LIL curves of commonly used random generators. Section 9 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 $n \geq 0$, $x \upharpoonright n = x[0..n-1]$ denotes the initial segment of length n of x ($x \upharpoonright n = x[0..n-1] = x$ 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" by Goldwasser and Micali (Goldwasser and Micali, 1984) and Yao (Yao, 1982) 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 difference $d_A(n) = |Prob[A(X_n) = 1] - Prob[A(Y_n) = 1]|$ 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 $\text{RAND}_{c,n}$ 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

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

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

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

Furthermore, let $f_n : \{0,1\}^n \to \mathscr{R}_{2n}$ be an ensemble of random functions (not necessarily computable) such that $f(x)$ is chosen uniformly at random from $\mathscr{R}_{2n}$. Then for each n-bit string x, with probability $1-\alpha$, $f_n(x)$ is Kolmogorov 2-random and with probability α, $f_n(x) \in \mathscr{R}_2^n$.

It should be noted that, for each given significance level $\alpha \leq 0.1$ and each test T from the 15 NIST SP800-22 tests, the following condition is satisfied:

$$|\{x \in \{0,1\}^n : x \text{ fails test } T \text{ at significance level } \alpha\}| \leq \frac{2^n}{c_T}$$

for a constant $c_T \gg 10$. Thus one can construct a Turing machine $M_{\alpha,T}$ with the following properties: for each string x of length n that fails the test T at the significance level α, there exists another string y such that $M_{\alpha,T}(y) = x$ and $|y| \leq n-3$. In

other words, we can compress x for at least 3 bits. It follows that all Kolmogorov 2-random strings are guaranteed to pass NIST SP800-22 test at significance level $\alpha \le 0.1$ and all strings in $\mathscr{R}_2^n$ 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 (Wang, 2002, 1997, 1996a, 1996b; Calude et al., 2001) can be used to show that $\mathscr{R}_{2n}$ 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.

As another example, let $\{f'_n\}_{n\in N}$ be a pseudorandom generator with $f'_n : \{0,1\}^n \to \{0,1\}^{l(n)}$ where $l(n) > n$. Assume that $\{f'_n\}_{n\in N}$ is a good pseudorandom generator by NIST SP800-22 in-depth statistical analysis of the P-value distributions (e.g., using χ^2-test). Define a new pseudorandom generators $\{f_n\}_{n\in N}$ as follows:

$$f_n(x) = \begin{cases} f'_n(x) & \text{if } f'_n(x) \text{ contains more 0's than 1's} \\ f'_n(x) \oplus 1^{l(n)} & \text{otherwise} \end{cases} \quad (2)$$

Then the following Theorem 3.1 shows that $\{f_n\}_{n\in N}$ is also a good pseudorandom generator by NIST SP800-22 in-depth statistical analysis of the P-value distributions (e.g., using χ^2-test). However, the output of $\{f_n\}_{n\in N}$ is trivially distinguishable from the uniform distribution.

Theorem 3.1. *Let $f'_n : \{0,1\}^n \to \{0,1\}^{l(n)}$ and $\{f'_n\}_{n\in N}$ be defined by the equation (2). Then for each $x\in\{0,1\}^n$, the P-values for $f(x)$ and $f'(x)$ are the same for all of the 15 NIST SP800-22 testing.*

Proof. The theorem can be proved for all of the 15 NIST SP800-22 testing using the symmetric properties (when 0s and 1s are flipped) of the corresponding probability distributions. In the following, we prove the theorem for the monobit testing of NIST SP800-22. For NIST SP800-22 monobit testing, the P-value on a binary string y is defined by

$$P-\text{value}(y) = erfc\left(\frac{\left|\sum_{i=0}^{|y|-1} 2y[i] - |y|\right|}{\sqrt{2|y|}}\right) \quad (3)$$

where

$$erfc(z) = \frac{2}{\sqrt{\pi}}\int_z^{\infty} e^{-u^2}du$$

Thus it is straightforward that P-value($f(x)$)=P-value($f'(x)$) for all $x\in\{0,1\}^n$.

The above two examples 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 (Wang, 2002). 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 $\frac{S(\xi\upharpoonright n)}{n}$ is ½, which corresponds to the frequency (Monobit) test in NIST SP800-22 (Rukhin et al., 2010). But it states nothing about the reduced deviation $S^*(\xi\upharpoonright n)$. It is intuitively clear that, for a pseudorandom sequence ξ, $S^*(\xi\upharpoonright n)$ will sooner or later take on arbitrary large values (though slowly). The law of the iterated logarithm (LIL), which was first discovered by Khinchin (Khinchin, 1924), gives an optimal upper bound $\sqrt{2\ln\ln n}$ for the fluctuations of $S^*(\xi\upharpoonright n)$. Based on this fact by Khinchin (Khinchin, 1924), we will use the following function in this paper

$$S_{lil}(x) = \frac{S^*(x)}{\sqrt{2\ln\ln|x|}} = \frac{2\cdot S(x) - |x|}{\sqrt{2|x|\ln\ln|x|}} \quad (4)$$

It is shown in Wang (2000) 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 5.1. *The normal density function with mean μ and variance σ is defined as*

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}}e^{-\frac{(x-\mu)^2}{2\sigma^2}}; \quad (5)$$

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

$$\varphi(x) = \frac{1}{\sqrt{2\pi}}e^{-\frac{x^2}{2}} \quad \text{and} \quad \Phi(x) = \int_{-\infty}^{x}\varphi(y)dy \quad (6)$$

The following DeMoivre-Laplace limit theorem is derived from the approximation theorem on page 181 of Feller (1968).

Theorem 5.2. *For fixed* x_1, x_2, *we have*

$$\lim_{n\to\infty} Prob\left[x_1 \le S^*(\xi\upharpoonright n) \le x_2\right] = \Phi(x_2) - \Phi(x_1). \quad (7)$$

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

$$\left|Prob\left[x_1 \le S^*(\xi\upharpoonright n) \le x_2\right] - (\Phi(x_2) - \Phi(x_1))\right| \le \max\left\{\frac{k^2}{n^2}, \frac{k^4}{n^3}\right\}$$

for all $n>0$.

In this paper, we only consider tests for $n \ge 2^{26}$ and $x_2 \le \sqrt{2\ln\ln n}$. Thus

$$k = S(\xi\upharpoonright n) - \frac{n}{2} \simeq \frac{\sqrt{n}}{2} S^*(\xi\upharpoonright n) \le \frac{\sqrt{2n\ln\ln n}}{2}.$$

Hence, we have $\max\{k^2/n^2, k^4/n^3\} = k^2/n^2 = (1-\alpha)^2 \ln\ln n / 2n < 2^{-22}$

By Theorem 5.2, 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 induced by the function S_{lil} defines a probability measure on the real line R. Let $\mathscr{R} \subset \Sigma^n$ be a set of m sequences with a standard probability definition on it. That is, for each $x_0 \in \mathscr{R}$, let $Prob[x=x_0] = 1/m$. Then each set $\mathscr{R} \subset \Sigma^n$ induces a probability measure $\mu_n^{\mathscr{R}}$ on R by letting

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

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 $\mathscr{R}_n$ is the collection of all length n sequences generated by a pseudorandom generator, then the difference between μ_n^U and $\mu_n^{\mathscr{R}_n}$ is negligible.

By Theorem 5.2, for a uniformly chosen ξ (that is, each bit of ξ is chosen uniformly at random from {0,1}), the distribution induced by the function $S^*(\xi\upharpoonright n)$ could be approximated by a normal distribution of mean 0 and variance 1, with error bounded by $1/n$ (see (Feller, 1968)). In other words, the measure μ_n^U can be calculated as

$$\mu_n^U((-\infty, x]) \simeq \Phi(x\sqrt{2\ln\ln n}) = \sqrt{2\ln\ln n}\int_{-\infty}^{x} \phi(y\sqrt{2\ln\ln n})dy. \quad (8)$$

Based on Equation (8), Fig. 1 shows the distributions of μ_n^U for $n=2^{26}, \ldots, 2^{34}$. For the reason of convenience, in the remaining part of this paper, we will use $\mathscr{B}$ as the discrete partition of the real line R defined by

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

With this partition, Table 1 lists values $\mu_n^U(I)$ on $\mathscr{B}$ with $n=2^{26}, \ldots, 2^{34}$. Since $\mu_n^U(I)$ is symmetric, it is sufficient to list the

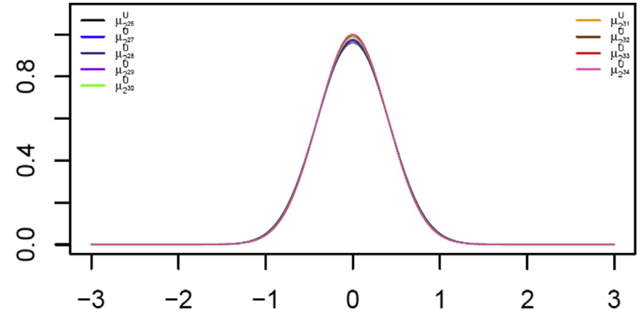

Fig. 1 – Density functions for distributions μ_n^U with $n=2^{26}, \ldots, 2^{34}$

distribution in the positive side of the real line. As an example, the cell at row [0.00,0.05) and column 2^{26} contains the value .047854, this means that $\mu_{2^{26}}^U([0.00, 0.05)) = .047854$.

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

A generator G is considered "good", if for sufficiently large m (e.g., $m \ge 10{,}000$), the distances between $\mu_n^{\mathscr{R}}$ 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 (Clarkson and Adams, 1933)

$$d(\mu_n^{\mathscr{R}}, \mu_n^U) = \sup_{A \subseteq \mathscr{B}} \left|\mu_n^{\mathscr{R}}(A) - \mu_n^U(A)\right| \quad (9)$$

Hellinger distance (Hellinger, 1909)

$$H(\mu_n^{\mathscr{R}} \| \mu_n^U) = \frac{1}{\sqrt{2}}\sqrt{\sum_{A \in \mathscr{B}}\left(\sqrt{\mu_n^{\mathscr{R}}(A)} - \sqrt{\mu_n^U(A)}\right)^2} \quad (10)$$

and the root-mean-square deviation

$$RMSD(\mu_n^{\mathscr{R}}, \mu_n^U) = \sqrt{\frac{\sum_{A \in \mathscr{B}} \left(\mu_n^{\mathscr{R}}(A) - \mu_n^U(A)\right)^2}{|\mathscr{B}|}}. \quad (11)$$

7. Distance based LIL experimental results

As an example to illustrate the importance of LIL tests, we carried out both NIST SP800-22 tests (NIST, 2010) 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.

Table 1 – The distribution μ_n^U induced by S_{lil} for $n=2^{26}, \ldots, 2^{34}$

	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,∞)	.008090	.007750	.007437	.007147	.006877	.006627	.006393	.006175	.005970

It should be noted that NIST SP800-22 test suite (NIST, 2010) checks the first 1,215,752,192 bits (≃145 MB) 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 × 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 $\xi = X_0X_1\cdots$ 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 16–30 bits of

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

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 × 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 (RANDOM.ORG). For the 10,000 × 2 GB sequences, we used a total of 10,000×4-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 (NIST, 2010) 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.

Based on snapshot LIL tests at points $2^{26}, \ldots, 2^{34}$, 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 $\mathrm{RMSD}(\mu_n^{cLCG}, \mu_n^U)$ 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 larger than 0.01. It is clear that this sequence collection is far away from the true random source.

Fig. 2 shows that the distributions of μ_n^{cLCG} for $n = 2^{26}, \ldots, 2^{34}$ are far away from the expected distribution in Fig. 1. Furthermore, Fig. 3 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 picture in Fig. 5. However, LIL curves for the standard C LCG generated sequences in Fig. 3 start reasonably well but deteriorate as the string length increases.

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

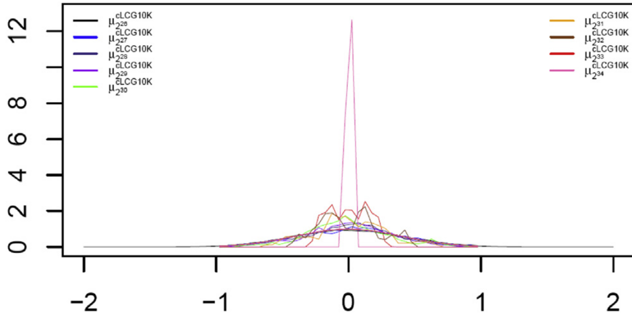

Fig. 2 – Density functions for distributions μ_n^{cLCG} with $n = 2^{26}, \ldots, 2^{34}$ for 10,000× 2 GB strings.

7.2. *Mersenne Twister generators*

Mersenne Twister (MT) is a pseudorandom generator designed by Matsumoto and Nishimura (Matsumoto and Nishimura, 1998) 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

$$\begin{aligned} y_1 &:= x \oplus (x >> 11) \\ y_2 &:= y_1 \oplus ((y_1 << 7) \text{ AND } 0x9D2C5680) \\ y_3 &:= y_2 \oplus ((y_2 << 15) \text{ AND } 0xEFC60000) \\ t(x) &:= y_3 \oplus (y_3 >> 18) \end{aligned}$$

Let $x_0, x_2, \ldots, x_{623} \in \Sigma^{32}$ be seeding values of $32 \times 624 = 19968$ bits 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 $a_{31}a_{30} \ldots a_0 = 0$ x9908B0DF. 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 (Matsumoto and Nishimura, 1998), we generated 10,000 × 2 GB sequences. The collection of these sequences passed NIST SP800-22 (NIST, 2010) 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 $2^{26}, \ldots, 2^{34}$, 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

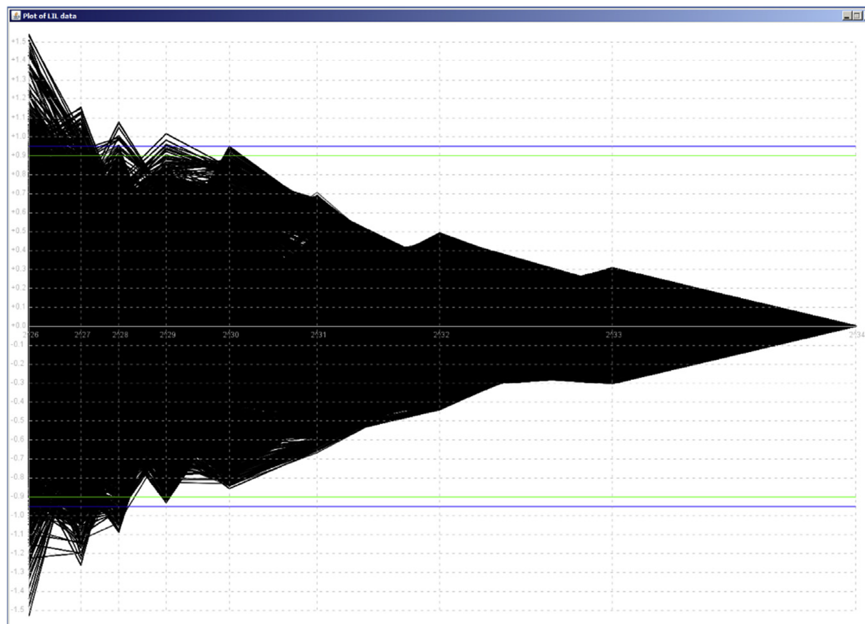

Fig. 3 – LIL curves for the standard C LCG for 10,000× 2 GB strings.

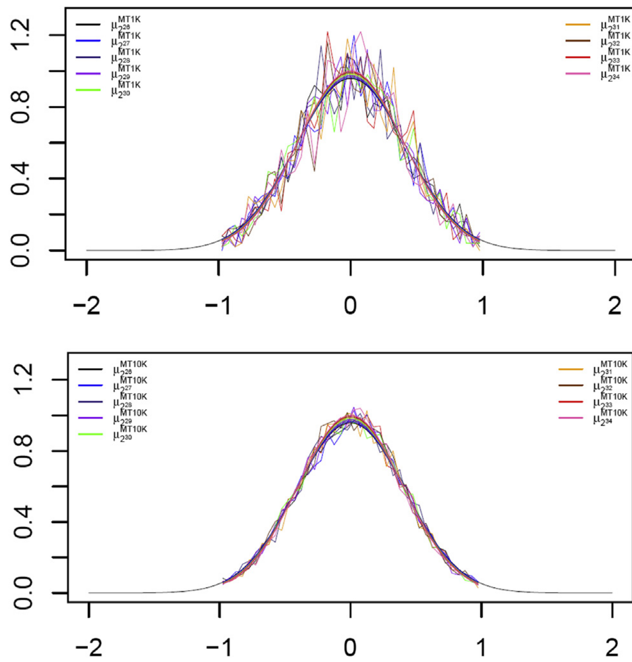

Fig. 4 – Density functions for distributions $\mu_n^{MT19937}$ at $n = 2^{26}, ..., 2^{34}$ with 1000 (first) and 10,000 (second) strings.

RMSD($\mu_n^{MT19937}, \mu_n^U$) at sample size 1000 (resp. 10,000) are calculated and shown in Table 3. In Table 3, the subscript 1 is for sample size 1000 and the subscript 2 is for sample size 10,000.

Fig. 4 shows the distributions of $\mu_n^{MT19937}$ for $n = 2^{26}, ..., 2^{34}$ where the curves are plotted on top of the expected distribution in Fig. 1. Furthermore, Fig. 5 shows the LIL-test result curves for the 10,000 sequences. The plot in Fig. 5 is close to what we are expecting for a random source.

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.

7.3.1. 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 6×2 GB random bit strings from the PHP Mersenne Twister m_rand() function call. It is estimated to take 2 years for our computer to generate 10,000 × 2 GB random bit strings since each 2 GB sequence takes 90 min 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

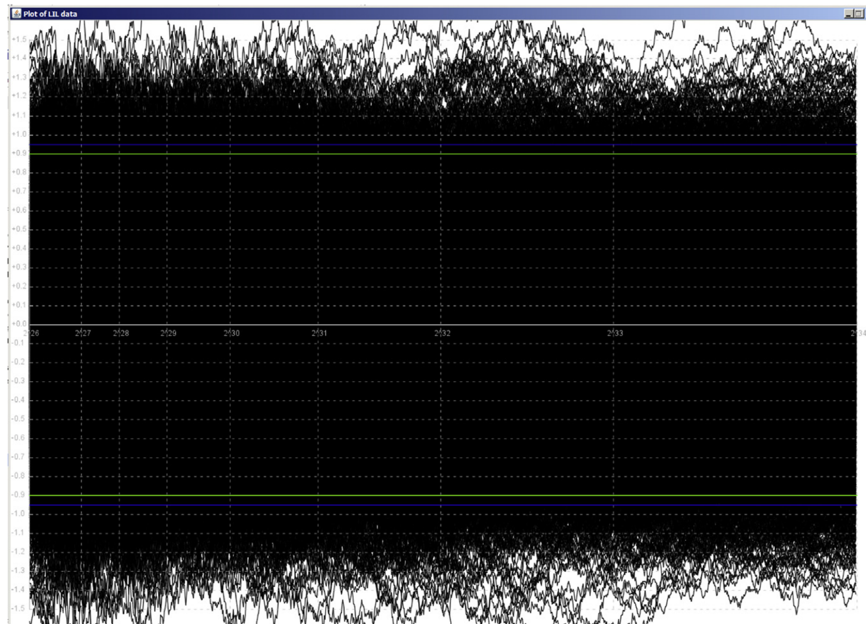

Fig. 5 – LIL curve for Mersenne Twister MT19937 with 10,000× 2 GB strings.

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

Twister, it will output an integer in the range [0,0x7FFFFFFF] each time while the source code in Matsumoto and Nishimura (Matsumoto and Nishimura, 1998) 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.

7.3.2. *PHP linear congruential generator*

Since it is slow to generate a large amount of random bits using PHP script, we only generated 6×2 GB sequences using the PHP rand() function call (similarly, it is estimated to take 2 years for our computer to generate 10,000 × 2 GB random bits). All of the sequences have similar LIL curves as shown in the first picture of Fig. 6. The second picture in Fig. 6 shows that the distributions of μ_n^{phpLCG} at $n=2^{26}, \ldots, 2^{34}$ are far away from the expected distribution in Fig. 1. One may also compare the second picture in Fig. 6 against the density distributions by the standard C linear congruential generator in Fig. 3. In summary, the PHP implementation of the linear congruential generator comprehensively failed NIST and LIL tests.

7.4. *Flawed Debian's OpenSSL package*

It is reported in Debian Security Advisory DSA-1571-1 (Debian) 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 × 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 Fig. 7, where the first picture is for the sample size of 1000 and the second picture is for the sample size of 10,000. In particular, Fig. 7 shows the distributions of μ_n^{Debian} for $n=2^{26}, \ldots, 2^{34}$ where the curves are plotted on top of the expected distribution in Fig. 1.As a comparison, we carried out snapshot LIL test on the standard OpenSSL pseudorandom generator (OpenSSL). We generated 10,000 × 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 Fig. 8, where the first picture is for the sample size of 1000 and the second picture is for the sample size of 10,000. In particular, Fig. 8 shows the distributions of $\mu_n^{OpenSSL}$ for $n=2^{26}, \ldots, 2^{34}$ where the curves are plotted on top of the expected distribution in Fig. 1.

The results in Figs. 7 and 8 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

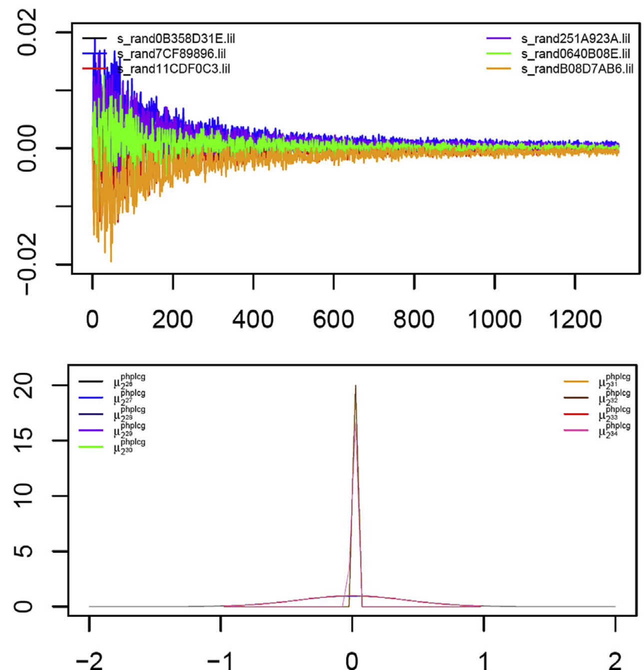

Fig. 6 – LIL curves for PHP LCG generated sequences (first) and density functions for distributions μ_n^{phpLCG} (second) of 6×2 GB PHP LCG sequences with $n = 2^{26},\cdots,2^{34}$

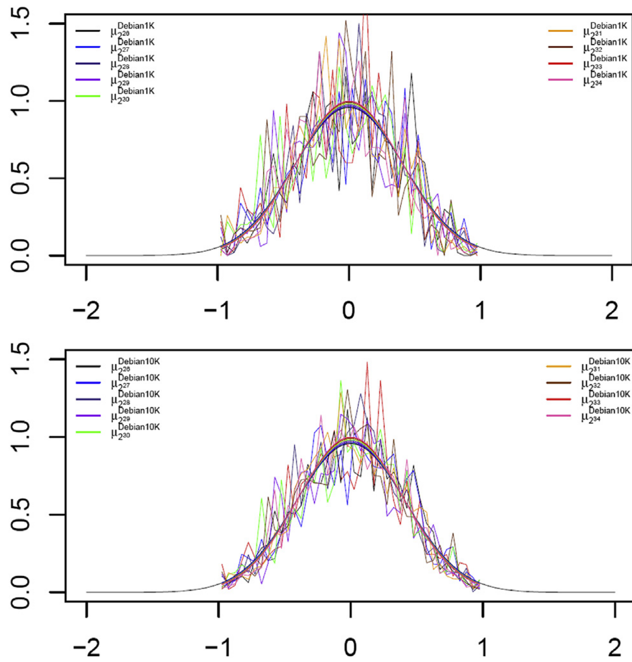

Fig. 7 – Density functions for distributions μ_n^{Debian} with $n=2^{26}, \ldots, 2^{34}$

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.

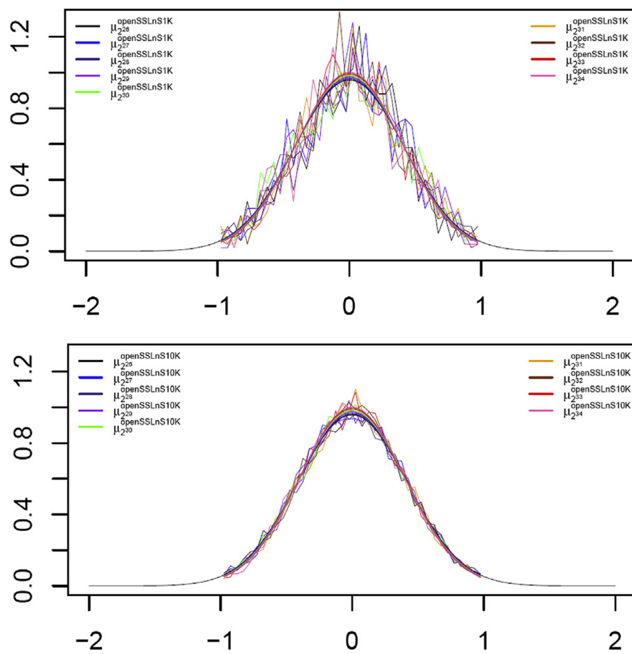

Fig. 8 – Density functions for distributions $\mu_n^{OpenSSL}$ with $n=2^{26}, \ldots, 2^{34}$

7.5. *Distance convergence and discussion*

It is important to determine the required number of strings for each distance-based statistical testing and to investigate the distributions of the statistics in distance based statistical tests since they are very different from NIST SP800-22 recommended testing. For example, the same α in one test could indicate a very different level of sensitivity when running another test.

Though it is important to investigate the relationsihp between LIL testing parameter selection criteria and the P-value α in NIST SP800-22 testing suite, this paper will focus on the threshold parameter selection criteria for LIL distance based testing. Indeed, we would recommend that all of the 15 testing techniques in NIST SP800-22 should be converted to distance based testing approach. Thus new parameters (instead of the current P-value α) for each of the 15 testing in NIST SP800-22 should be determined.

One question that could be raised is whether the specific parameter selection for the LIL distance based tests could be the reason why some random generators (such as the flawed Debian Linux OpenSSL) failed LIL test but passed NIST SP800-22 tests. The answer to this question is no. A weak random generator could pass NIST SP800-22 tests for various reasons. For the specific case of the flawed Debian Linux OpenSSL randomness generator, it passed the NIST SP800-22 tests due to the limitation that we point out in Theorem 3.1 of Section 3.

The experimental results in previous sections show that statistical distances obtained from Mersenne Twister generators could be used as general threshold distances for LIL testing. In this section, we try to get the approximate threshold distances for LIL statistical testing. Fig. 9 shows total variation, Hellinger, and root-mean-square deviation arithmetic mean distances for Mersenne Twister, standard C linear congruential generator, flawed Debian OpenSSL generator with single thread, flawed Debian OpenSSL generator with multiple threads, standard OpenSSL generator without external seed, standard OpenSSL generator with external seed, Microsoft crypto pseudorandom generator, and NIST DRBG-SHA256 generator. Graphs in the Figure are generated at the sample sizes: 1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000, 9000, and 10000. In these graphs, we use a unit of 1 in x-axis 1 to denote 1000 sample size. From these graphs, it is clear that standard C linear congruential generator has the worst distances from a true random source, Debian OpenSSL generator with single thread has the second worst distances, and Debian OpenSSL generator with multiple threads has the third worst distances. Other distance trends coincide well with the distance trend for Mersenne Twister generator. The collection of arithmetic mean distances that are used to plot Fig. 9 are shown in Table 4. Table 4 lists total variation, Hellinger, and root-mean-square deviation arithmetic mean distances for Mersenne Twister, standard C linear congruential generator, Debian OpenSSL generator with single thread, Debian OpenSSL generator with multiple threads, OpenSSL generator without external seed, OpenSSL generator with external seed, Microsoft crypto pseudorandom generator, and NIST DRBG-SHA256 generator. Distances are presented for sample sizes: 1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000, 9000, and 10000.

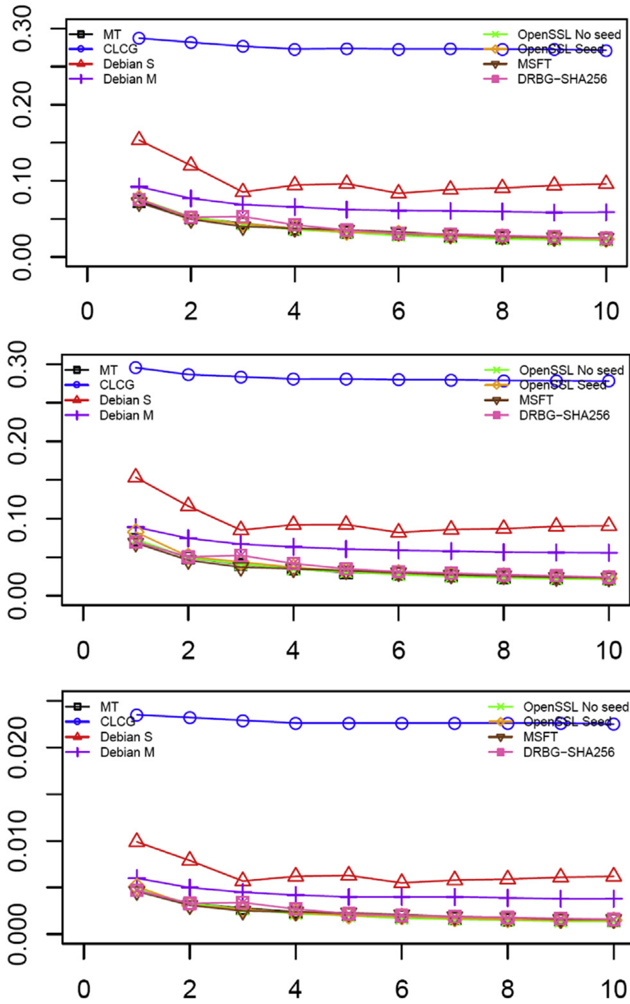

Fig. 9 – Total variation (first picture), Hellinger (second picture), and root-mean-square deviation (third picture) arithmetic mean distances.

For any given sample size, it is feasible to use Mersenne Twister generator to randomly generate a collection of random sequences of the given sample size and then obtain the corresponding threshold distance for LIL testing at this sample size. However, it is also possible to approximate the threshold distance using power functions. We combined a total of 55,000 × 2GB (=110 TB) of random sequences generated using Mersenne Twister generator and calculated the total variation, Hellinger, and root-mean-square deviation arithmetic mean distances at sample sizes 5 K, 10 K, 15 K, 20 K, 25 K, 30 K, 35 K, 40 K, 45 K, 50 K, and 55 K as shown in Table 5.

Using arithmetic mean distances from Table 5, we can obtain the approximate power functions for threshold distances as follows

$$\begin{aligned} y &= 4.6983x^{-0.57} && \text{for total variation distances} \\ y &= 3.3809x^{-0.541} && \text{for Hellinger distances} \\ y &= 0.3404x^{-0.583} && \text{for root} - \text{mean} - \text{square deviation distances} \end{aligned}$$

where x represents sample size of sequences to be generated and y represents the threshold distance for a LIL test to pass at the sample size x. As an example, Fig. 10 shows the curves for total variation distance and the trend line $y = 4.6983x^{-0.57}$ for 55,000 × 2GB generated sequences.

7.6. *Summary of distance based experiments*

As a summary, Table 6 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 $1 \le k \le 623$. 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 6 show that the LIL testing techniques always produce expected results while NIST SP800-22 test suite does not.

8. LIL curves for commonly used pseudo random generators

Section 7 presents experimental results on distance based LIL testing where density function curves are used to demonstrate the testing results. Several LIL curves are also presented in Figs. 3, 5 and 6 to illustrate the testing results. Since the function S_{lil} is the heart for distance based LIL testing and it is related to the distribution of Wiener process (the Brownian motions), it is useful to interpret LIL curves as Brownian motions. Specifically, a good pseudo random generator should generate LIL curves as in Fig. 5 instead of Figs. 3 and 6. In this section, we present some experiments on the LIL curves for commonly used pseudo random generators. It should be noted that the results in this section are not analyzed using the distance based LIL testing distribution. We present the experimental results for LIL curves to motivate the potential future testing design using LIL curves.

8.1. *Java SHA1PRNG API based sequences*

The SHA1PRNG API in Java generates sequences SHA1′(s,0) SHA1′(s,1)…, where s is an optional seeding string of arbitrary length, the counter i is 64 bits long, and SHA1′(s,i) is the first 64 bits of SHA1(s,i). When no seed is provided, Java provides random seeds itself. In our experiments, we generated one hundred of sequences without seeds and another one hundred of sequences with 32 bytes random seeds. For each sequence generation, the "random.nextBytes()" method of SecureRandom Class is called 2^{26} times and a 20-byte output is requested for each call. This produces sequences of 1.34 GB long. The LIL test is then run on these sequences and we observed similar trend curves for all sequences. Specifically, we observed that $S_{lil}(\xi[0..n-1]) \le 0.5$ for $n > 5$MB on average and then $-0.75 \le S_{lil}(\xi[0\ldots n-1]) \le 0.3$ for $n > 6$MB on average. We also observed that the value S_{lil} is smaller than 0 for majority parts of each sequence. This means that these sequences generally contain more zeros than ones. We suspected that the smaller (or negative) values of S_{lil} were caused by sparse values in the counter since it is 64 bits long and we only feed values between 0 to 2^{26} to it. In order to verify our conjecture, we generated one hundred of sequences SHA1′(x_1)SHA1′(x_2)…

Table 4 – Total variation, Hellinger, and root-mean-square deviation arithmetic mean distances.

Generator		1 K	2 K	3 K	4 K	5 K	6 K	7 K	8 K	9 K	10 K
MT	$\overline{d}$	.0730	.0519	.0445	.0372	.0326	.0299	.0278	.0253	.0246	.0237
	$\overline{H}$	.0729	.0491	.0407	.0353	.03007	.0288	.0259	.0237	.0233	.0224
	$\overline{RMSD}$	.0047	.0033	.0028	.0024	.0021	.0019	.0017	.0016	.0016	.0015
C LCG	$\overline{d}$	.2873	.2819	.2767	.2731	.2738	.2731	.2733	.2728	.2725	.2714
	$\overline{H}$	.2955	.2865	.2837	.2807	.2808	.2799	.2797	.2789	.2786	.2781
	$\overline{RMSD}$	.0235	.0232	.0229	.0226	.0226	.0226	.0226	.0226	.0226	.0225
Debian S-thread	$\overline{d}$	.1535	.1203	.0854	.0944	.0963	.0837	.0887	.0907	.0941	.0961
	$\overline{H}$	.1533	.1161	.0852	.0920	.0922	.0821	.0859	.0870	.0899	.0907
	$\overline{RMSD}$	.0099	.0079	.0057	.0062	.0063	.0055	.0058	.0059	.0061	.0062
Debian M-thread	$\overline{d}$	.0924	.0770	.0689	.0658	.0624	.0611	.0608	.0596	.0585	.0588
	$\overline{H}$	.0891	.0745	.0670	.0633	.0607	.0590	.0579	.0566	.0560	.0557
	$\overline{RMSD}$	.0060	.0050	.0045	.0042	.0040	.0040	.0040	.0039	.0038	.0038
OpenSSL No Seed	$\overline{d}$	.0765	.0529	.0433	.0359	.0326	.0282	.0257	.0237	.0228	.0220
	$\overline{H}$	.0729	.0494	.0413	.0346	.0310	.0273	.0250	.0232	.0221	.0212
	$\overline{RMSD}$	.0048	.0033	.0027	.0022	.0020	.0017	.0016	.0015	.0014	.0014
OpenSSL with Seed	$\overline{d}$	.0769	.0499	.0439	.0376	.0328	.0310	.0280	.0267	.0246	.0236
	$\overline{H}$	.0826	.0510	.0438	.0369	.0322	.0297	.0272	.0259	.0239	.0228
	$\overline{RMSD}$	.0051	.0031	.0027	.0023	.0020	.0019	.0017	.0016	.0015	.0015
MSFT	$\overline{d}$	.0716	.0495	.0401	.0378	.0358	.0331	.0293	.0276	.0258	.0252
	$\overline{H}$	.0685	.0461	.0371	.0351	.0327	.0301	.0272	.0261	.0239	.0233
	$\overline{RMSD}$	.0046	.0031	.0025	.0024	.0023	.0021	.0019	.0018	.0016	.0016
DRBG-SHA256	$\overline{d}$	.0754	.0522	.0533	.0423	.0355	.0312	.0302	.0283	.0268	.0248
	$\overline{H}$	.0698	.0503	.0524	.0418	.0350	.0313	.0294	.0274	.0258	.0240
	$\overline{RMSD}$	.0047	.0033	.0034	.0027	.0022	.0020	.0019	.0018	.0017	.0016

using SHA1PRNG API without seeds, where $x_1x_2x_3\ldots$ are pseudorandom sequences generated by AES128 with different keys and each x_i is 64 bits long. We then run the LIL test and observed that S_{lil} values for these sequences take larger values though they still fail the LIL test.

Fig. 11 shows three typical LIL test results. The black line is for a sequence with seed s = SHA256(0xD2029649D2029649). The blue line (in web version) is for a sequence without a seed. The red line (in web version) is for a sequence without a seed but with counters replaced by random strings from $x_1x_2x_3\ldots = \mathrm{AES}(k,0)\mathrm{AES}(k,1)\ldots$, where

k = 0x0E14533E1F056F7C7E192B3F4C4D7E6F.

To reduce the size of the figure, we use the scale $10000n^2$ for the x-axis. In other words, Fig. 11 shows the values $S_{lil}(\xi[0\ldots 10000n^2-1])$ for $1 \le n \le 1037$. The readers may ask: does $S_{lil}(\xi[0\ldots m-1])$ reach either 1 or −1 for $10000n^2 < m < 10000(n+1)^2$? For each sequence, we generated the curve using the scale $8n$ also (that is, we take values at the end of each byte of the sequence) and the result is similar to the scale $10000n^2$. The reason is that the values of S_{lil} change very slowly when n is large.

Fig. 12 shows another result on six sequences generated by SHA1PRNG API using the same scale $10000n^2$ for the x-axis. The first three lines (black, blue, red (in web version)) are for sequences that are generated by SHA1PRNG API without seeds. The fourth line (purple (in web version)) is for a sequence that is generated by SHA1PRNG API without a seed but with a decreasing counter. That is, it is for the sequence $\mathrm{SHA1}'(2^{26})\mathrm{SHA1}'(2^{26}-1)\ldots$ The fifth and sixth lines (green and orange (in web version)) are for sequences that are generated by SHA1PRNG API with 64 bytes and 70 bytes of random seeds respectively.

8.2. *NIST SP800-90A based sequences*

NIST SP800.90A specifies three kinds of DRBG generators: hash function based, block cipher based, and ECC based. For DRBG generators, the maximum number of calls between reseeding is 2^{48} for hash function and AES based generators (the number is 2^{32} for T-DES and ECC-DRBG generators). For sequences that we have generated, no reseeding is needed according to this rule.

8.2.1. *Block cipher based DRBG*

For block cipher based generators, we generated 100 sequences in the format of $\mathrm{AES128}(k,V)\mathrm{AES128}(k,V+1)\ldots$ and $\mathrm{DES}(k,V)\mathrm{DES}(k,V+1)\ldots$ where k is the random key and V is derived from random seeds. The value of V is revised after the primitive is called 2^{12} times according to (Barker and Kelsey,

Table 5 – Total variation, Hellinger, and root-mean-square deviation arithmetic mean distances for 55,000 ×2 GB sequences.

	5 K	10 K	15 K	20 K	25 K	30 K	35 K	40 K	45 K	50 K	55 K
$\overline{d}$	.0370	.0252	.0200	.0160	.0147	.0128	.0118	.0112	.0107	.0100	.0097
$\overline{H}$	.0343	.0233	.0186	.0157	.0142	.0125	.0118	.0110	.0104	.0097	.0095
$\overline{RMSD}$	.0024	.0016	.0013	.0010	.0009	.0008	.0008	.0007	.0007	.0006	.0006

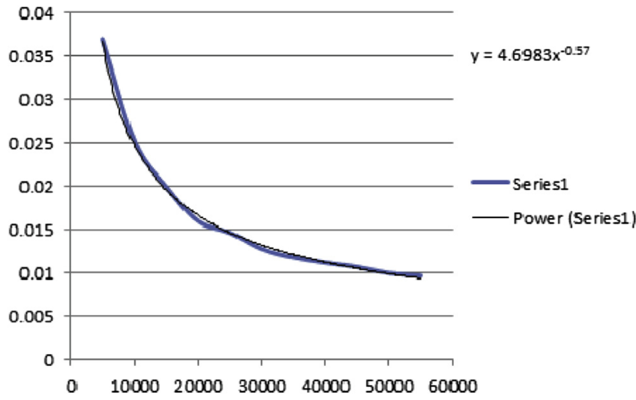

Fig. 10 – Total variation distance trend line.

2012). Each sequence is 2.15 GB long. The LIL test is run on these sequences and we observed that $S_{lil}(\xi[0..n-1]) \in [-0.95, 0.95]$ on average. It is interesting to mention that the values of S_{lil} fluctuate evenly in the interval $[-0.9, 0.9]$ for these block cipher based sequences. This is different from the results for hash function based sequences for which the values S_{lil} are more biased with smooth fluctuation (cf. the results in Section 8.1 and the results later in this section). When n increases, the value S_{lil} for all block cipher based sequences tends not to go above 0.99.

As an example, Fig. 13 shows the LIL test results on three sequences generated by AES128 and DES. Similarly, we use the scale $10000n^2$ for the x-axis. $E_1(\cdot)$ and $E_2(\cdot)$ denote AES128 and DES encryption functions respectively. These lines are for sequences $E_1(k,ctr_0)E_1(k,ctr_0+1)\ldots$, $E_2(k_1,0)E_2(k_1,1)\ldots$, and $E_2(k_2,ctr_0)E_2(k_2,ctr_0+1)\ldots$ respectively, where k, k_1, k_2 are random keys and ctr_0 is a random value of 8 bytes.

Dual ECC-DRBG NIST SP800-90A recommends a dual EC-DBRG where the underlying elliptic curve is defined by $y^2 = x^3 - 3x + b \pmod{p}$. SP800-90A recommends three curves for the random bits generation: P-256, P-384, and P-521. The initialization parameter includes two points P and Q on the curve. The random bits are generated from stages and the random generator has its internal state. For simplicity, we use a number $s \in F_q$ to denote the internal state of the generator. When the state is s_i, the generator first calculates a point $R_i = s_i Q$ on the elliptic curve and outputs as random bits the least significant 240 bits (respectively, 368 bits and 504 bits) of the x-coordinate of R_i for P-256 (respectively, P-384 and P-521). After outputting the random bits, the internal state of the

Table 6 – 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

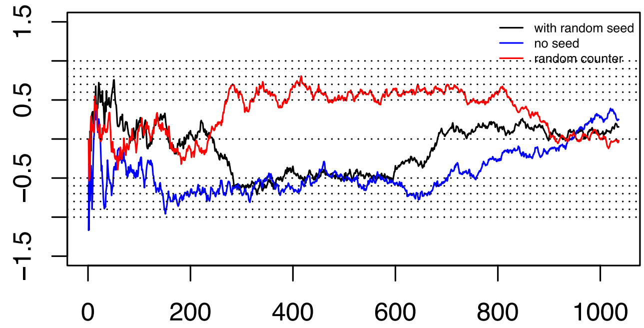

Fig. 11 – Typical results for Java SHA1PRNG API.

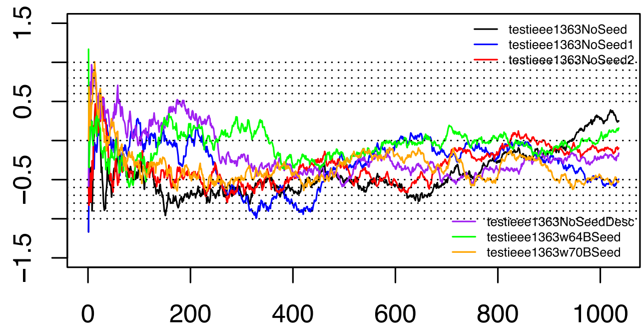

Fig. 12 – More results for Java SHA1PRNG API.

generator is updated to $s_{i+1} = x(s_i P)$ where $x(s_i P)$ denotes the x-coordinate of the elliptic curve point $s_i P$. In our experiments, we generated 16 random sequences lilDataecc0nistP256, ⋯, lilDataecc15nistP256 using the curve P-256 with the initial states s_0 = SHA(0), …, s_0=SHA(15) respectively. For each sequence generation, we make 2^{22} calls to elliptic exponentiation primitives and each call outputs 240 bits. Thus each sequence is 120 MB long. Since it takes 26 h for the DELL Optiplex 755 computer (with Bouncy Castle ECC Library for Java Netbeans) to generate one sequence, we have not tried to generate longer sequences. Fig. 14 shows the test results for these 16 random sequences.

Hash function based DRBG For hash function based DRBG in NIST SP800.90A, a hash function G is used to generate sequences $G(V)G(V+1)G(V+2)\ldots$ where V is a seedlen-bit counter that is derived from the secret seeds, seedlen is 440 for SHA1,

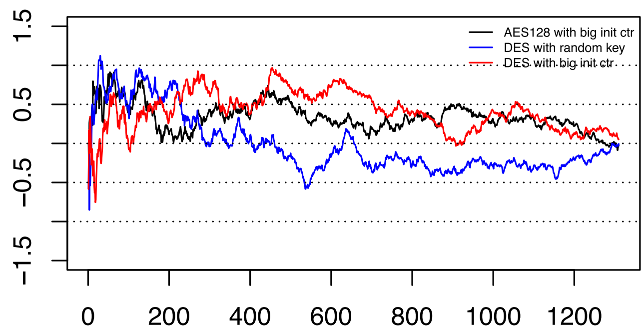

Fig. 13 – Results for AES and DES based generators.

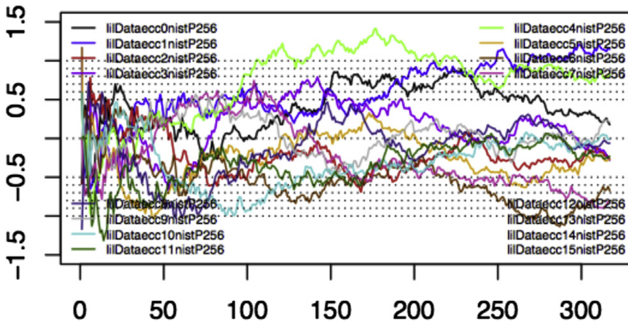

Fig. 14 – Results for ECC-DRBG

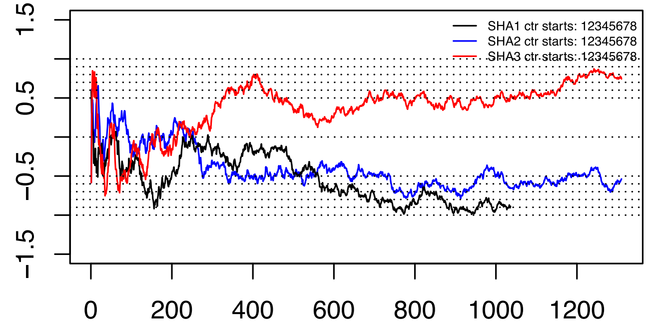

Fig. 16 – Results for non-NIST SHA1/SHA2/SHA3-DRBG.

and the value of V is revised after at most 2^{19} bits are output. We generated several hundreds of sequences with randomly chosen seeds for SHA1 and SHA256 based DRBG. As an example, Fig. 15 shows the test results on six typical sequences generated by SHA1-based DRBG with the scale $10000n^2$ for the x-axis.

8.3. Other hash function based generators

In order to analyze the randomness properties of hash functions from different angles, we also generated hash function based pseudorandom sequences without following the procedures in NIST SP800.90A. We used different sizes of seed values (e.g., 4 bytes–100 bytes) and different counter styles (e.g., a counter begins from 0 instead of V or use decremental counters). The results show that S_{lil} curves for SHA1-based and SHA256-based sequences are similar. But they are different from S_{lil} curves for Keccak256-based sequences. Specifically, if we use 4 bytes of seeds and 8 bytes of counters that start from 0, then for large enough n, $S_{lil}(\xi[0..n-1]) \le 0$ for SHA1/SHA256 based sequences and $S_{lil}(\xi[0..n-1]) \ge 0$ for Keccak256 based sequences. These results seem to reveal the non-random property of SHA1/SHA2/Keccak functions and show that Keccak (SHA3) may not have better stochastic properties than SHA1/SHA2.

Fig. 16 shows the test results on three typical sequences generated by SHA1, SHA256, and Keccak256 using the scale $1000n^2$ for the x-axis. These sequences are generated with empty seeds and with a 32-bit counter starting at 12345678. The hash functions are called 2^{26} times. Thus the SHA1 based sequence is 1.34 GB and the SHA256/Keccak256 based sequences are 2.15 GB. It is observed that, for SHA1 and SHA256 based sequences, we have $S_{lil}(\xi[0..n-1]) \le 0$ when n is sufficiently large and, for sequences generated using Keccak256, we have $S_{lil}(\xi[0..n-1]) \ge 0$ when n is sufficiently large.

Fig. 17 shows another LIL test result on nine sequences based on the SHA1 hash function using the scale $10000n^2$ for the x-axis. Each sequence is 1.34 GB long (2^{26} calls to the hash function). In the following, we use G_1 to represent the SHA1 hash function and all random integers and random seeds are taken from a sequence generated by AES128 in counter mode. The line SHA1NoSeedDesc is for the sequence $G_1(ctr_0)\ldots G_1(0)$ with a 4-byte decreasing counter that starts at $ctr_0 = 2^{26}-1$. Line SHA1NoSeedLargeCtr is for the sequence $G_1(ctr_0)$ $G_1(ctr_0+1)\ldots G_1(ctr_0+2^{26}-1)$ where ctr_0 is a random 4-byte integer. The line SHA1w4BR10B0 is for the sequence $G_1(s,v_0,0)\ldots$ where s is a 4-byte random seed, v_0 is 10 bytes of 0, and ctr is a 4-byte counter starting at 0. The line SHA1w4BR is for the sequence $G_1(s,0)G_1(s,1)\ldots$ where s is a 4-byte random seed and ctr is a 4-byte counter starting at 0. The line SHA1w10BR is for the sequence $G_1(s,0)G_1(s,1)\ldots$ where s is a 10-byte random seed and ctr is a 4-byte counter starting at 0. The line SHA1w70BR is for the sequence $G_1(s,0)\ldots$ where s is a 70-byte random seed and ctr is a 4-byte counter starting at 0. The line SHA1w100BR is for $G_1(s,ctr_0)G_1(s,ctr_0+1)\ldots$ where s is a 100-byte random seed and ctr_0 is a 4-byte random integer. The line SHA1wRS64CTR is for the sequence $G_1(s,0)G_1(s,1)\ldots$ with a 8-byte counter and a 8-byte random seed s. The line SHA14B4B is for $G_1(s,ctr_0)G_1(s,ctr_0+1)\ldots$ where s is a 4-byte random seed and ctr_0 is a 4-byte random integer.

Fig. 18 shows some LIL test results on 12 sequences based on SHA256 and Keccak256 hash functions using the scale

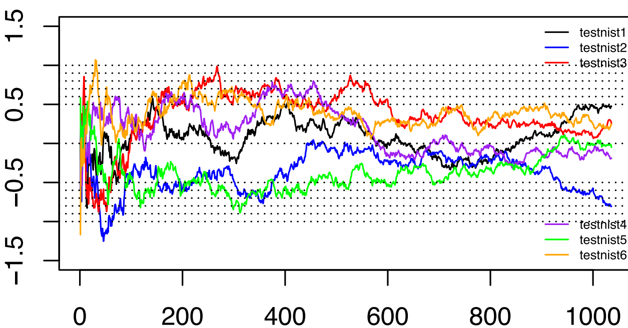

Fig. 15 – Results for NIST SHA1-DRBG.

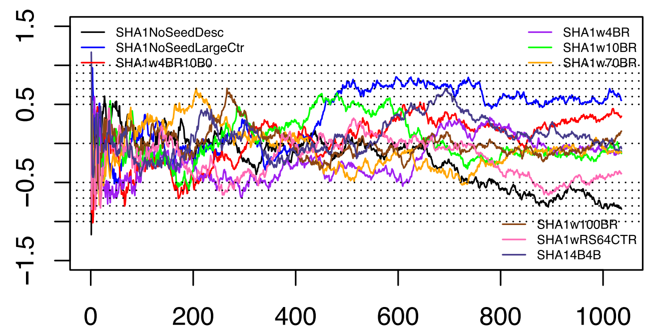

Fig. 17 – Results for some sequences based on SHA1.

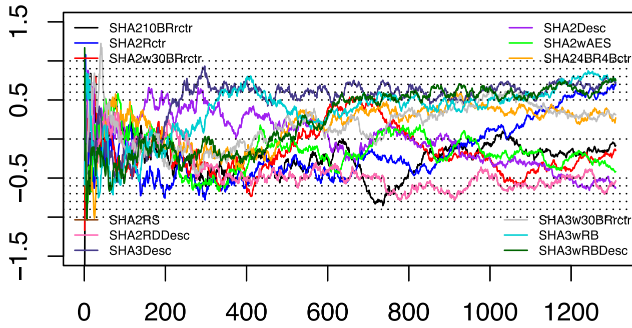

Fig. 18 – Results for some sequences based on SHA2/SHA3.

$10000n^2$ for the x-axis. Each sequence is 2.15 GB long (2^{26} calls to the hash function). In the following, we use G_2 and G_3 to denote SHA256 and Keccak256 respectively. All random integers and random seeds are taken from random positions of a pseudorandom sequence generated by AES128 in counter mode with a random key. The line SHA210BRrctr is for the sequence $G_2(s,ctr_0)G_2(s,ctr_0+1)\ldots$ where ctr_0 is a 4-byte random integer and s is a 210-byte random seed. The line SHA2Rctr is for the sequence $G_2(ctr_0)G_2(ctr_0+1)\ldots$ where ctr_0 is a 4-byte random integer. Line SHA2w30BRrctr is for the sequence $G_2(s,ctr_0)G_2(s,ctr_0+1)\ldots$ where s is a 30-byte random seed and ctr_0 is a 4-byte random integer. The line SHA2Desc is for the sequence $G_2(2^{26}-1)G_2(2^{26}-2)\ldots G_2(0)$ where the counter is a 4-byte integer. The line SHA2wAES is for the sequence $G_2(x_0)G_2(x_1)\cdots G_2(x_{2^{26}-1})$ where $x_0x_1\ldots$ is a pseudorandom sequence and x_i is 8 bytes. The line SHA24BR4Bctr is for the sequence $G_2(s,ctr_0)G_2(s,ctr_0+1)\ldots$ where ctr_0 is a 4-byte random integer and s is a 24-byte random seed. The line SHA2RS is for the sequence $G_2(s,0)G_2(s,1)\ldots$ where s is a 500-byte random seed the counter is 4 bytes. Line SHA2RDDesc is for the sequence $G_2(s,2^{26}-1)G_2(s,2^{26}-2)\ldots G_1(s,0)$ where s is a 100-byte random seed and the counter is 4 bytes. The line SHA3Desc is for the sequence $G_3(2^{26}-1)G_3(2^{26}-2)\ldots G_3(0)$ where ctr_0 is a 4-byte random integer. The line SHA3w30BRrctr is for the sequence $G_3(s,ctr_0)G_3(s,ctr_0+1)\ldots$ where s is a 30-byte random seed and ctr_0 is a 4-byte random integer. The line SHA3wRB is for $G_3(s,0)G_3(s,1)\ldots$ where s is a 4-byte random seed and the counter is 4 bytes. The line SHA3wRBDesc is for the sequence $G_3(s,2^{26}-1)G_3(s,2^{26}-2)\ldots G_3(s,0)$ where s is a 8-byte random seed and the counter is 4 bytes.

8.4. *Fortuna PRNG*

Fortuna pseudorandom number generator (Schneier and Ferguson (Ferguson and Schneier, 2003)) uses block ciphers such as AES in counter mode and the key is changed each time after at most 1 MB of data is generated. We uses AES-128 as the underlying block cipher to instantiate Fortuna PRNG. In total, we generated 100 sequences: fortunaAES0, ..., fortunaAES99. Each of these sequence is 1 GB long. Specifically, for the generation of sequence fortunaAESi, we run AES-128 in counter mode and use keys SHA1($i\|j$) for $0\leq j\leq 2^{10}$. Each of the AES key SHA1($i\|j$) is used to encrypt 2^{16} consecutive counters. Fig. 19 shows the LIL-test results for the sequences fortunaAES0, ..., lilDatafortunaAES15. In Fig. 19, most of the curves lie strictly

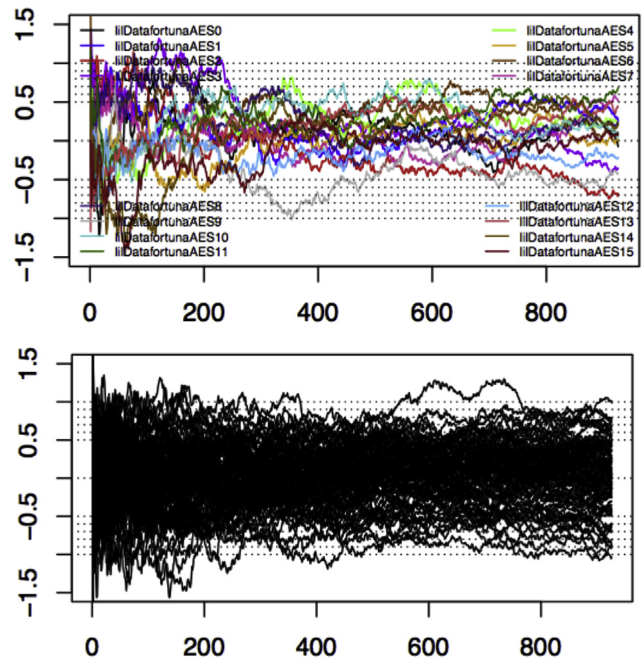

Fig. 19 – Results for Fortuna-PRNG.

within a proper sub-interval of [−1,1] though one line reaches −1. Thus we may or may not claim that Fortuna-PRNG-AES passes the LIL test. More testing is needed to confirm whether Fortuna PRNG passes the LIL-test.

9. General discussion on OpenSSL random generators

It is noted in Ahmad (2008) 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.

Fig. 20 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.

`state` is the entropy pool that is a declared array 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.

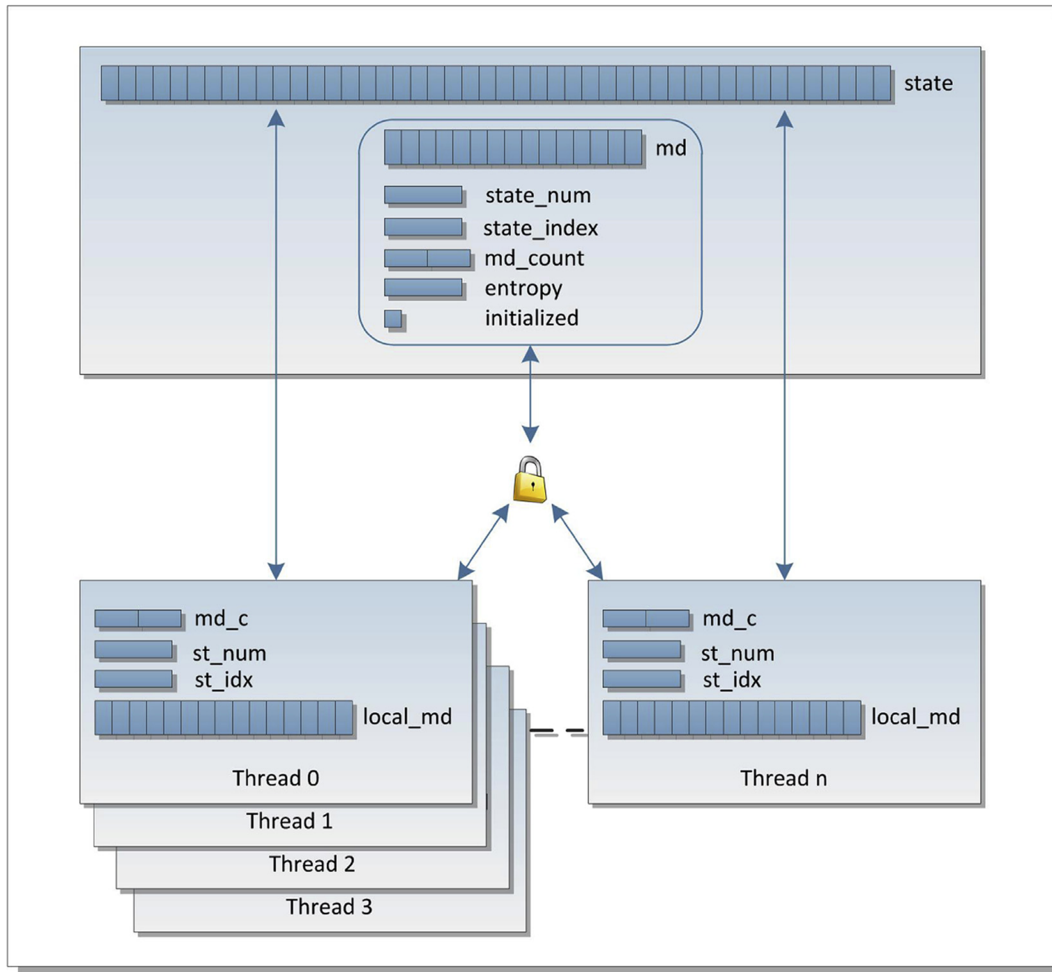

Fig. 20 – High Level view of OpenSSL RNG.

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 indicate seed status. If not initialized, entropy collection and seeding functions are called.

9.1. *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 Fig. 21. 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 `netstatget(NULL, L"LanmanWorkstation", 0, 0, &outbuf)`. By using `LanmanWorkstation`, `netstatget` 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 cryptographic service provided by ADVAPI32. Use the 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.

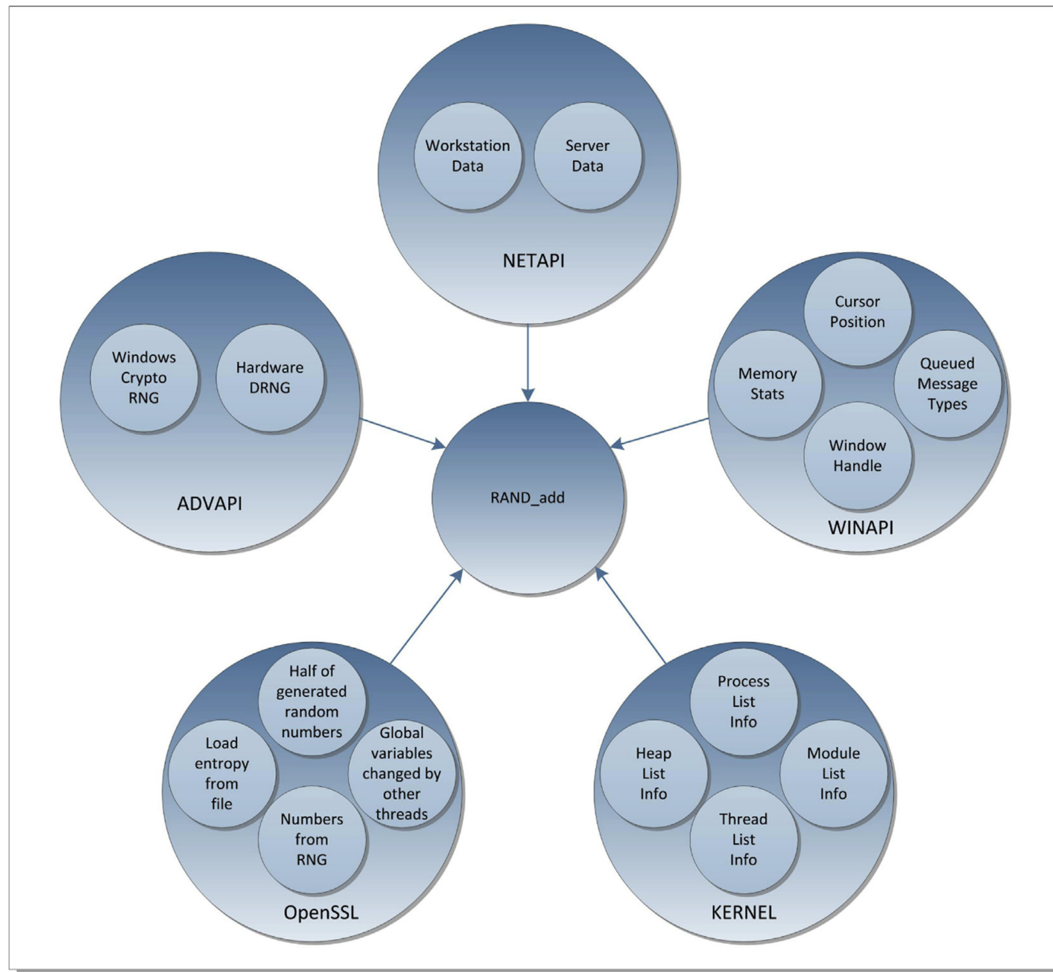

Fig. 21 – OpenSSL entropy sources on Windows.

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 9.2.
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.

9.2. *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 a DLL module handle containing function `GetCursorInfo`, which 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.

9.3. *Optional additional entropy sources*

In OpenSSL RNG, additional entropy is not gathered by the default RAND_poll function. Thus it requires programmers to explicitly call the collection functions. It should be also noted that the Unix version of RAND_poll is quite different to the Windows version as it reflects the differences in architecture. In UNIX, the version of RAND_poll is selected by a conditional compilation switch.

In OpenSSL RNG, additional entropy can be added by calling RAND_load_file function in randfile.c. This mixes file content into the RNG using RAND_add. randfile.c also contains RAND_write_file which calls RAND_bytes to collect random data from state to overwrite the seed file providing fresh entropy for when it is read next. Care needs to be taken with these functions since RAND_write_file does not take a size parameter to specify the number of bytes to write and is fixed at 1024 by a declaration #define RAND_DATA 1024. So if a 1 MB file of random data was used as a seed data, only 1 KB of it will be overwritten by RAND_write_file with seed data so on successive reads of the seed file, 99.9% of the seed data is the same every time. For Windows-based systems, three more entropy gathering functions are available in rand_win.c: RAND_event, readtimer and readscreen. RAND_event collects entropy from key presses and mouse movement, readtimer collects data from the high speed counters and readscreen adds the hash values from each line of screen data. In each instance, the collected data is added into the entropy pool using RAND_add.

9.4. *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 available functions, RAND_add() calls meth->add() which in this case 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, Fig. 22 describes the seeding flowchart for OpenSSL random number generators.

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.

9.5. *Summary of OpenSSL RNG*

In a summary, OpenSSL generates pseudorandom bits using the following process:

1. Set crypto lock to protect global data updates.
2. Copy global variables into local variables to avoid conflict from other threads. The global message digest md is an entropy source if multithreading as other threads update it so copy it to local_md. However, in a single threaded application this value will not change.
3. Add num entropy bytes in buf to the global state_index (wrapping if needed) so another thread will access state from there. This thread starts from st_idx.
4. Initialize the m in EVP_MD_CTX structure with pointers to crypto functions and data blocks for hash operations. The key structure within m is the digest substructure as this contains pointers to the main hash functions and the message digest.
5. Partially hash local_md into m by calling MD_update.
6. Adjust global state_num to be greater than or equal to state_index to mark the region of state to be modified by the entropy data in buf.
7. In chunks of MD_DIGEST_LENGTH do
   - Compute the hash output m as

$$SHA1(local_md \| 20\,bytes\,of\,state \| 20\,bytes\,of\,buf \| md_c[0] \| md_c[1])$$

The resultant digest in m is copied into local_md ready to be mixed into state.

- Increment the local md counter md_c[1]
- Add MD_DIGEST_LENGTH to buf to point to the next bytes of entropy.
- XOR the 20 byte hash in local_md into state from state [st_idx] incrementing st_idx on each write. Note: no lock is applied to state during this update to improve performance. However, the data is XORed into state as

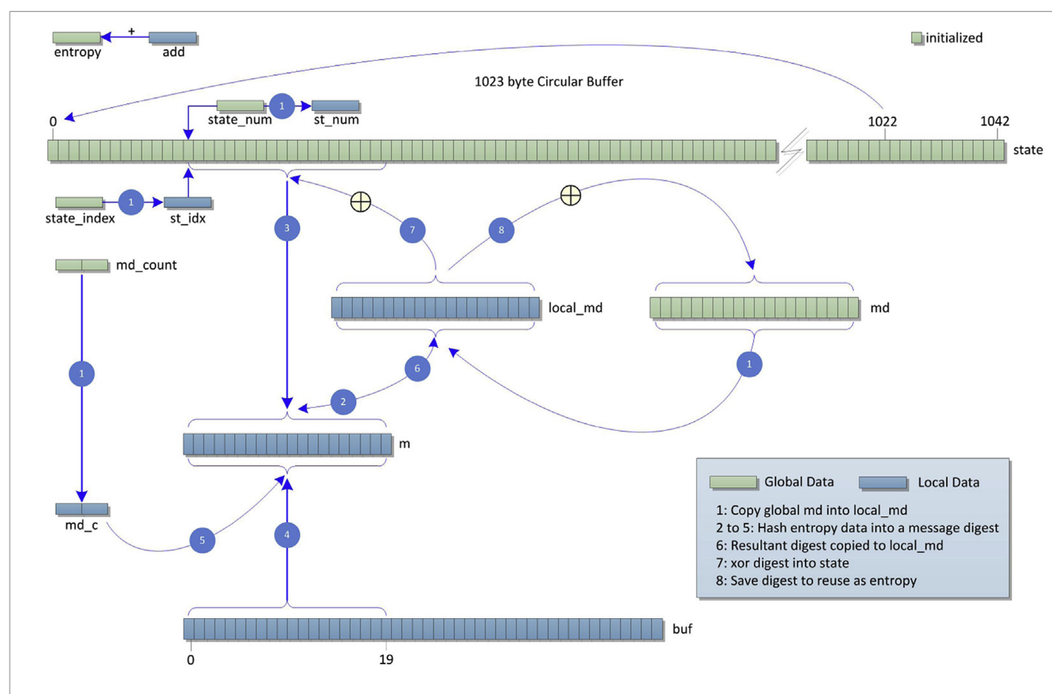

Fig. 22 – Seeding the OpenSSL random number generator.

simply copying it in would overwrite any data added by other threads.

- Continue until num bytes of buf have been hashed into the pool.

8. Securely clean-up and release RAM: EVP_MD_CTX_cleanup(&m).
9. Set crypto lock to enable exclusive update to global data.
10. XOR local_md into the global md. This value will be used as entropy when read by the current or another thread.
11. If there is less than required 256 bits of entropy then increase it by entropy estimate add passed into ssleay_rand_add.
12. Release the lock.

9.6. *Get Random Numbers*

There are two simple top level functions to request random numbers from OpenSSL. The first is to use ssleay_rand_nopseudo_bytes where the required andom numbers do not need to be cryptographically strong. This is achieved by calling ssleay_rand_bytes with the pseudo flag set to 0. Thus ssleay_rand_bytes returns all random numbers in buf whether secure or not. The second approach is to use ssleay_rand_bytes with the pseudo flag set to 1 to generate secure random bits (the details are shown in Fig. 23). The difference between the two approaches is that for the non-secure request, the entropy value is ignored while for the secure request, if entropy is too low then an error is flagged but the number extraction is exactly the same in both cases.

In reference to Fig. 23, ssleay_rand_bytes starts by creating a message digest context using EVP_MD_CTX_init to create a structure m with the appropriate digest functions and structures for hashing. A crypto lock is applied to provide exclusive access to global variables. If the RNG has not been initialized, RAND_poll is called to seed the RNG with system generated entropy. The flag ok is set if the entropy level has reached ENTROPY_NEEDED which is defined as 32. If not ok, the global variable entropy is reduced by the number of random numbers requested. Entropy added to the RNG within this function is distributed throughout the state structure; this is referred to in OpenSSL as “stirring the pool”. This involves using a dummy seed: #define DUMMY_SEED “…” These 20 characters are hashed into the RNG state using ssleay_rand_add:

```
ssleay_rand_add(DUMMY_SEED, MD_DIGEST_LENGTH, 0.0);
```

The hash is continually applied in groups of 20 across the whole of state to distribute new entropy. If there is enough entropy (ok flag is set), the pool will not be stirred again as the stirred_pool flag is set and is a static variable so will be “remembered” next time the function is called. It is probably declared volatile to avoid the compiler optimizing it into a register thus hiding it from other threads: static volatile int stirred_pool = 0; The function makes a copy of global variables: state_index and state_num to mark the 20 element section of state to work on. Message digest counters and a previously created message digest will be used as further entropy sources during the extraction of random numbers. Once these variables have been copied, the crypto lock is removed. The random number extraction iterative process then begins until all the requested numbers have been copied into the provided buffer buf. The following is a step-

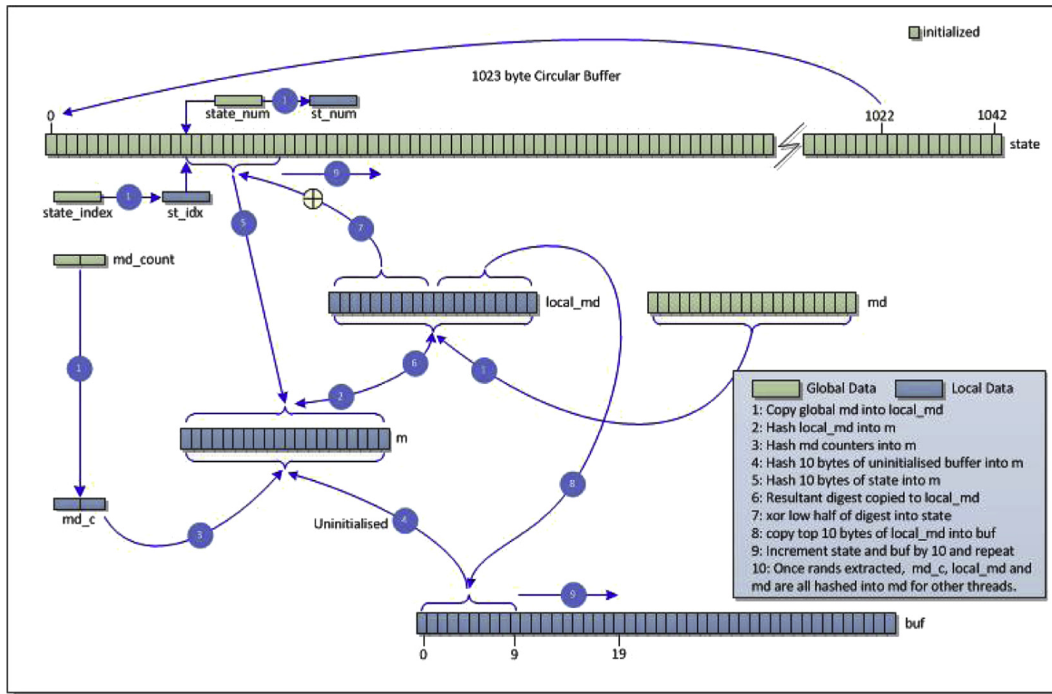

Fig. 23 – Get random numbers.

by-step high level explanation of random bit extraction. first set crypto lock and copy global variables into local variables. The global variable `md` holds the previously generated message digest, possibly from a different thread, so is used as an entropy source. The `md` counters are another small entropy source as these have been incremented, possibly by other threads as well as the current thread.

1. For `num` bytes in `buf` in chunks of 20 do (get random)
   (a) The `local_md` (copy of global `md`) is partially hashed into m.
   (b) Partially hash local md counter `md_c` (copy of the global counters) into `m`.
   (c) Partially hash 10 bytes of the caller's uninitialized buffer `buf` into `m`. Note: this is the second statement removed by the Debian maintainer but is harmless. This statement can be conditionally compiled-out using the `PURIFY` switch enabling users to utilizes tools such as valgrind or purify which forbid the use of uninitialized data.
   (d) Partially hash 10 bytes of data from `state[st_idx]` to `state[st_idx+9]` (circularly) into `m`.
   (e) Finalize the hash value in `m` and copy it to `local_md`.

$$\text{local_md} = \text{SHA1}(\text{md}\|\text{md_c}\|10 \text{ bytes of buf}\|10 \text{ bytes of state})$$

`local_md` now holds 20 bytes of random data.

   (f) XOR the lower 10 bytes of `local_md` into `state [st_idx...st_idx+9]` as new entropy. The RNG is continually supplied with new self-generated seed data.
   (g) Take the top 10 bytes of `local_md` and copy into `buf` as 10 bytes of random data to be returned to the caller.
   (h) Increment `st_idx` and `buf` pointer to repeat the process until all num random bytes have generated and copied into the `buf`.

2. End of the iterative process of creating random numbers for the user.

Once the `buffer` contains the requested numbers, `MD_Update` is called several times to create a new hash for the global `md`:

$$\text{md} = \text{SHA1}(\text{md_c}[0]\|\text{md_c}[1]\|\text{local_md}\|\text{md})$$

where `md` is used as entropy by other threads, or by the current thread if there is only one, next time the function is called. If non-crypto random bits were requested, the function returns 0. If crypto random bits were requested and enough entropy was used during generation then 1 is returned. If crypto random bits were requested but not enough entropy was used, 0 is returned and an error `RAND_R_PRNG_NOT_SEEDED` is generated.

9.7. *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.

9.8. *Several single pont failures for OpenSSL RNG*

In this section, we briefly describe several single points of failure for OpenSSL random number generator. The attacks don't attempt to crash the application as this would be quickly discovered; they aim to force the RNG to generate weak random numbers in anticipation of the attack going unnoticed to then enable the hacker to compromise security operations dependent on those random numbers.

The first single point of failure for OpenSSL RNG is the function `MD_Final`. Entropy is added to the RNG state using sequential calls to `MD_Update` but the final hash is not created until `MD_Final` is called. If `MD_Final` is not called then the hash value is not added into the state. There are three places where `MD_Final` could be attacked. One of these is chosen to demonstrate the effect of an attack of this type. The source code was modified to determine the effect. By removing the `MD_Final` call at the point in the `file md_rand.c`, the RNG is severely compromised causing its period to be reduced to `MD_DIGEST_LENGTH/2` which in this case is 10 bytes. In our experiments, we were able to carry out this attack on the binary distributions of the codes. We searched the executable for a string of machine code bytes identifying the call to `MD_Final`, the call was effectively removed by replacing the 5 byte call instruction with `XCHG EAX, EAX` which is op code `0x90` (a no operation (NOP)).

The second single point of failure is made by forcing `_EVP_DigestFinal` to return before doing anything of value. This is so damaging because it is the only function responsible for returning the hash value so will negate all calls to `MD_Final`. This is effected by simply searching string "8B 45 08 8B 08 83 79 08 40" and overwriting a single byte in the executable with the op code C3 (ret). After this modification, the OpenSSL RNG output is a string of 0's. Care needs to be taken with these attacks to avoid misaligning the stack and crashing the executable. Finding the precise attack points will be a little more involved generally but they will be fixed and identifiable so can be found whether within static or dynamic libraries.

For the third single point of failure, we use `RAND_poll` function. This function is wholly responsible for collecting system generated entropy or entropy EGD etc. We can attack a single global flag `initialized` by setting its value to 1. Then `RAND_poll` can be bypassed completely leaving no entropy at all unless external seeding functions are called. However, OpenSSL does detect that the entropy value is below 32 so returns an error which if trapped can alert the user. But if a second global variable `entropy` is also attacked, then the RNG will, unless extra seeding functions are called, generate the same numbers every time.

For the fourth example, we attack the `ssleay_rand_bytes` function. There is a check for `buf` length of zero and if detected it returns 1 which means function returned successfully. The simulated attack overrides the number of bytes requested setting `num = 0` to prove the function returns without error. As this function does not return the number of bytes read, the caller assumes the returned `buffer` contains all the bytes they asked for but in fact, the `buffer` contents are unchanged. If the `buffer` is uninitialized, its contents may even look random but it will be the same every time. By analyzing the machine code, the attack point is identified: We remove the jump instruction by overwriting with NOP instruction `0x90` so the function will always return with 1 having done nothing. If the caller has asked for 300 numbers, they will use 300 bytes from their buffer believing them to be random. This was emulated in a test program then the attack was carried out on the OpenSSL.exe by searching for a specific string of bytes then overwriting two bytes with `0x90`.

10. 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 distance based LIL testing technique is an important tool and should be used for statistical testing. Though distance based LIL testing is important, it only covers the law of the iterated logarithm. In the NIST SP800-22 testing suite, test techniques for 15 statistical laws are designed. For the short term, we recommend that all pseudorandom generators be tested using both NIST SP800-22 testing tools and our distance based LIL testing. For the long term, we recommend that new testing tools be designed using distance based techniques. That is, for each of the 15 statistical laws in NIST SP800-22 testing suite, a corresponding distance based test should be designed. In this paper, we also provided a detailed documentation on OpenSSl random generators and described several potential attacks.

Acknowledgements

Our thanks to the reviewers for their comments and suggestions.

REFERENCES

Ahmad D. Two years of broken crypto: debian's dress rehearsal for a global pki compromise. Secur Priv IEEE 2008;6(5):70–3.
Barker E, Kelsey J. NIST SP 800-90A: Recommendation for random number generation using deterministic random bit generators. NIST; 2012.
Calude C, Hertling P, Khoussainov B, Wang Y. Recursively enumerable reals and chaitin's Ω numbers. Theor Comput Sci 2001;255:125–49.
Clarkson JA, Adams CR. On definitions of bounded variation for functions of two variables. Tran AMS 1933;35(4):824–54.
Debian. Debian security advisory dsa-1571-1. available at: http://www.debian.org/security/2008/dsa-1571.
Feller W. Introduction to probability theory and its applications, vol. I. New York: John Wiley & Sons, Inc.; 1968.
Ferguson Niels, Schneier Bruce. Practical cryptography, vol. 141. New York: Wiley; 2003.
Goldwasser S, Micali S. Probabilistic encryption. J Comput Sys Sci 1984;28(2):270–99.
Hellinger E. Neue begründung der theorie quadratischer formen von unendlichvielen veränderlichen. J. für die reine Angew Math 1909;136:210–71.

Khinchin A. Über einen satz der wahrscheinlichkeitsrechnung. Fund Math 1924;6:9–20.
Matsumoto M, Nishimura T. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. ACM TOMACS 1998;8(1):3–30.
NIST. Test suite. 2010. http://csrc.nist.gov/groups/ST/toolkit/rng/.
OpenSSL. Openssl implementation from http://www.openssl.com/.
RANDOM.ORG. Random.org http://www.random.org/.
Rukhin A, Soto J, Nechvatal J, Smid M, Barker E, Leigh S, et al. A statistical test suite for random and pseudorandom number generators for cryptographic applications. NIST SP 800–822. 2010.
Wang Yongge. Randomness, stochasticity and approximations. In: RANDOM; 1997. p. 213–25.
Wang Y. Catetory, measure, and polynomial time approximations. SIAM J Comput 1999a;28:394–408.
Wang Y. A separation of two randomness concepts. Inf Proccess Lett 1999b;69:115–8.
Wang Yongge. Resource bounded randomness and computational complexity. Theor Comput Sci 2000;237:33–55.
Wang Yongge. A comparison of two approaches to pseudorandomness. Theor Comput Sci 2002;276(1):449–59.
Yao AC. Theory and applications of trapdoor functions. In: Proc. 23rd IEEE FOCS; 1982. p. 80–91.

Yongge Wang received his PhD degree from the University of Heidelberg of Germany. Since then, Dr. Wang has worked in the industry for a few years until he joined UNC Charlotte in 2002. In particular, Dr. Wang has worked in Certicom (now a division of RIM) as a cryptographic mathematician specializing in efficient cryptographic techniques for wireless communications. Dr. Wang has been actively participated in and contributed to standard bodies such as IETF, W3C XML Security protocols, IEEE 1363 standardization groups for cryptographic techniques, ANSI X9 group for the financial services industry standards, and ANSI T11 groups for SAN network security standards. Dr. Wang is the inventor of Secure Remote Password authentication protocol SRP5 which is an IEEE 1363.2 standard and the inventor of identity based key agreement protocol WANG-KE which is an IEEE 1363.3 standard. Dr. Wang has also worked with Cisco researchers and American Gas Association researchers to design security protocols for the SCADA industry. Dr. Wang has published extensively on research topics including computational complexity, algorithmic information theory, randomness and pseudorandomness, critical infrastructure protection, perfectly secure message transmission, cryptography and secure authenticated communications, and statistical testing.

Tony Nicol is a Master student at University of Liverpool, UK.