On Stochastic Security of Java Crypto and NIST DRBG Pseudorandom Sequences

Yongge Wang
Dept. SIS, UNC Charlotte
Charlotte, NC 28223, USA
Email: yongge.wang@uncc.edu

***Abstract*—Cryptographic primitives such as secure hash functions (e.g., SHA1, SHA2, and SHA3) and symmetric key block ciphers (e.g., AES and TDES) have been commonly used to design pseudorandom generators with counter modes (e.g., in Java Crypto Library and in NIST SP800-90A standards). It is assumed that if these primitives are secure then the pseudorandom generators based on these primitives are also secure. However, no systematic research and analysis have been done to support this assumption. Based on complexity theoretic results for pseudorandom sequences, this paper analyzes stochastic properties of long sequences produced by hash function based pseudorandom generators DRBG from NIST SP800-90A and SHA1PRNG from Java Crypto Library. Our results show that none of these sequences satisfy the law of the iterated logarithm (LIL) which holds for polynomial time pseudorandom sequences. Our results also show that if the seeds and counters for pseudorandom generators are not appropriately chosen, then the generated sequences have strongly biased values for LIL-tests and could be distinguished from uniformly chosen sequences with a high probability. Based on these results, appropriate seeding and counter methods are proposed for pseudorandom generator designs. The results in this paper reveal some "non-random" behavior of SHA1, SHA2, and of the recently announced SHA3.**

I. Introduction

Pseudorandom generators and pseudorandom sequences play important roles in modern cryptography. For example, the weakness in pseudorandom generators was employed to attack the SSL protocol [12]. A string is said to be cryptographically pseudorandom if no efficient observer can distinguish it from a uniformly chosen string of the same length. Secure cryptographic hash functions such as SHA1, SHA2, and SHA3 [21], [5] are often used to generate pseudorandom sequences with fixed length outputs (e.g., 160 bits or 256 bits). In practice it is also important to generate long pseudorandom sequences. For example, in the security proof of cryptographic protocols using the "random oracle model" paradigm (see, e.g., [2], [9], [19]), the participants and the adversaries may make polynomial number of queries to the random oracle and the total output from the random oracle could be several gigabytes (GBs) long. In order for the security proof to work, it is assumed that the random oracle output is computationally indistinguishable from a string that is chosen with the uniform probability. In practice, the random oracle is normally instantiated using secure hash functions.

Though security of hash functions such as SHA1, SHA2, and SHA3 has been extensively studied from the one-wayness and collision resistant aspects, there has been limited research on the quality of long pseudorandom sequences generated by cryptographic hash functions. Recently, the authors of [4] used the indifferentiability concept from [19] to analyze the security of sponge function based pseudorandom generators [4], [1] by assuming that the underlying primitives are random permutations (or random functions). However, there is no existing feasible approach to verify whether a given primitive is a random permutation (or a random function). Even if a hash function (e.g., SHA1) performs like a random function based on existing statistical tests (e.g., NIST SP800-22 Revision 1A [22]), when it is called many times for a long sequence generation, the resulting long sequence may not satisfy the properties of pseudorandomness and could be distinguished from a uniformly chosen sequence. The experimental results in this paper show that sequences produced by pseudorandom generators SHA1PRNG (in Java) and hash function based DRBG (from NIST SP800-90A) could be distinguished from uniformly chosen sequences with a high probability (e.g., at least 2%).

In complexity theoretic research, random sequences are considered as Brownian motions that are described by the Wiener process. One of the important laws for the Wiener process is the law of the iterated logarithm (LIL) which says that, for a pseudorandom sequence ξ, the value $S_{lil}(\xi[0..n-1])$ (this value is defined in Theorem 5.1) should stay in $[-1,1]$ and reach both ends infinitely often when n increases. It is known that polynomial time pseudorandom sequences follow LIL. However, our experimental results show that, for sequences generated by SHA1PRNG in Java and hash function based DRBG in NIST SP800-90A, $S_{lil}(\xi[0..n-1])$ stays within a proper sub-interval of $[-1,1]$ and does not reach either bound when n is large.

We also observed that if seeds and counters for hash function based pseudorandom generators are chosen in such a way that the last message block for the hash function (e.g., SHA1, SHA2, and SHA3) consists mainly of padded 0-bits, then S_{lil} for SHA1 and SHA2-generated sequences take negative or small positive values when n is large while S_{lil} for SHA3-generated sequences take positive values when n is large. These experimental results show that the newly designed SHA3 may not provide better stochastic security compared to SHA1 and SHA2. For hash function based DRBG in NIST SP800-90A, the seeding information to the pseudorandom generator is converted to a seedlen-bit counter, where seedlen is 440 for SHA1/SHA256 and is 888 for SHA384/SHA512. Thus the input to each hash function call in DRBG contains one message block after the hash function internal padding. Our experiments show that sequences generated by NIST SP800-90A DRBG have relatively flat S_{lil} value distribution when n becomes large. Thus they could be distinguished with a high probability from uniformly chosen sequences. In order to avoid these deficiencies, we propose to use dynamic changing inputs to each of the hash function call so that the last message block has significant changes in the 0-1 distribution and the second from the last message block (which contributes to the state of the hash function operation) changes for each hash function call also. The dynamic inputs could be generated using linear feedback shift registers (LFSRs) or another pseudorandom generator. Our experiments show that the proposed new generators produce better S_{lil}-value distributions.

Canetti, Goldreich and Halevi [9] showed that there exist "artificially designed" cryptosystems for which the random oracle cannot be realized by a family of hash functions and Maurer, Renner, and Holenstein [19] generalized this result by showing that a random oracle contains substantially more entropy than a finite random string. Though these results show the limitation of the random oracle methodology, it is still widely used to prove the security of practical protocols such as RSA-OAEP ([3], [7], [24]). Let $\mathcal{HR}$ be the family of sequences with $-0.99 \leq S_{lil}(\xi[0..n-1]) < 0.99$ for $n = 3 \times 2^{24}$ (= 6MB). We show that any subset of $\mathcal{HR}$ could be efficiently distinguished from a set of uniformly chosen sequences with a probability of 2%. Our experiments show that sequences generated by NIST SP800-90A DRBG and Java SHA1PRNG belong to $\mathcal{HR}$. Thus random oracles in practical protocols should not be instantiated using these pseudorandom generators.

The paper is organized as follows. Section II introduces notations. Section III reformulates pseudorandom generator concepts in terms of martingales. Section IV compares cryptographic pseudorandom sequences and complexity theoretic pseudorandom sequences. Section V discusses the law of iterated logarithms (LIL). Section VI proposes two LIL tests and justifies these tests. Section VII reports experimental results and Section VIII proposes an improved pseudorandom generator design.

II. Notations

Classical random sequences were first introduced as a type of disordered sequences, called "Kollektivs", by von Mises [26] as a foundation for probability theory. The two features characterizing a Kollektiv are: the existence of limiting relative frequencies within the sequence and the invariance of these limits under the operation of an "admissible place selection". Here an admissible place selection is a procedure for selecting a subsequence of a given sequence ξ in such a way that the decision to select a term $\xi[n]$ does not depend on the value of $\xi[n]$. Ville [25] showed that von Mises' approach is not satisfactory by proving that: for each countable set of "admissible place selection" rules, there exists a "Kollektiv" which does not satisfy the law of the iterated logarithm (LIL). Later, Martin-Löf [18] developed the notion of random sequences based on the notion of typicalness. A sequence is typical if it is not in any *constructive* null sets. Schnorr [23] and Lutz [17] introduced p-randomness concepts by defining the *constructive* null sets as polynomial time computable measure 0 sets. The law of the iterated logarithm (LIL) plays a central role in the study of the Wiener process and Wang [27] showed that LIL holds for p-random sequences.

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 $|x|$. λ is the empty string. 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 $n \geq 0$, $x[0..n]$ denotes the initial segment of length $n+1$ of x ($x[0..n] = x$ if $|x| \leq n+1$) while $x[n]$ denotes the nth bit of x, i.e., $x[0..n] = x[0] \ldots x[n]$. For each string w, $\mathbf{C}_w = \{w\xi \ : \xi \in \Sigma^\infty\}$ is called the basic open set defined by w. For a set $\mathbf{C}$ of infinite sequences, $Prob[\mathbf{C}]$ denotes the probability that $\xi \in \mathbf{C}$ when ξ is chosen by a uniform random experiment. Martingales are used to describe betting strategies in probability theory.

Definition 2.1: (Ville [25]) A martingale is a function $F : \Sigma^* \to R^+$ such that, for all $x \in \Sigma^*$,

$$F(x) = \frac{F(x1) + F(x0)}{2}.$$

We say that a martingale F *succeeds* on a sequence $\xi \in \Sigma^\infty$ if $\limsup_n F(\xi[0..n-1]) = \infty$.

Lemma 2.2: (Ville [25]) Let F be a martingale and $F_k = \{x \in \Sigma^* \ : \ F(x) > k\}$. Then $Prob[F_k \cdot \Sigma^\infty] \leq F(\lambda)k^{-1}$.

Based on Lemma 2.2, Ville [25] showed that a set of infinite sequences has probability 0 (or Lebesgue measure 0) if and only if there is a martingale which succeeds on all sequences in the set. For each basic open set $\mathbf{C}_{x_0}$, define a martingale F^{x_0} by

$$F^{x_0}(x) = \begin{cases} 2^{|x|-|x_0|} & x \sqsubseteq x_0 \\ 1 & x_0 \sqsubseteq x \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

Then $F^{x_0}(\lambda) = 1/2^{|x_0|} = Prob[\mathbf{C}_{x_0}]$ and, for all $x \in x_0 \cdot \Sigma^*$, $F^{x_0}(x) = 1$. Throughout the paper, we will use the martingale F^U for the uniform distribution by letting $F^U(x) = 1$ for all $x \in \Sigma^*$. That is, $F^U(x) == F^\lambda(x)$ or all $x \in \Sigma^*$.

A martingale ensemble $\{F_n\}_{n \in N}$ is a sequence of martingales with the following properties:

1) For each $n \in N$, F_n is a martingale with $F_n(\lambda) = 1$.
2) For $|x| > n$, $F_n(x) = F_n(x[0..n-1])$.

In other words, for a martingale ensemble $\{F_n\}_{n \in N}$, each F_n defines a probability distribution over Σ^n.

III. Pseudorandom generators

The concept of "effective similarity" by Goldwasser and Micali [13] and Yao [28] 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. This concept can be rephrased in terms of martingales. First, we note that each probability ensemble $X = \{X_n\}_{n\in N}$ can be represented as a martingale ensemble $\{F_n\}_{n\in N}$ with the following properties:

1) $F_n(\lambda) = 1$ and $F_n(x) = 2^n \cdot Prob[X_n = x]$ for all $x \in \Sigma^n$. In other words, $F_n(x) = Prob[X_n|x]$ for all $x \in \Sigma^n$.
2) For $|x| > n$, $F_n(x) = F_n(x[0..n-1])$.

Definition 3.1: Let $\{F_n\}_{n\in N}$ and $\{\bar{F}_n\}_{n\in N}$ be two martingale ensembles. $\{F_n\}_{n\in N}$ and $\{\bar{F}_n\}_{n\in N}$ are computationally (respectively, statistically) indistinguishable if for any polynomial time computable set $D \in \Sigma^*$ (respectively, any set $D \in \Sigma^*$) and any polynomial p, the inequality (2) holds for almost all n.

$$\frac{1}{2^n} \cdot \left| \sum_{x\in D\cap\Sigma^n} F_n(x) - \sum_{x\in D\cap\Sigma^n} \bar{F}_n(x) \right| \leq \frac{1}{p(n)} \qquad (2)$$

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 $x \in \Sigma^*$. The martingale ensemble $\left\{F^G_{l(n)}\right\}_{n\in N}$ is defined by letting $F^G_{l(n)}(x) = \sum_{x_0\in\Sigma^n} F^{n,G(x_0)}(x)$ where

$$F^{n,G(x_0)}(x) = \begin{cases} 2^{|x|-n} & x \sqsubseteq G(x_0) \\ 0 & \text{otherwise} \end{cases}$$

for $|x| \leq l(n)$, and $F^{n,G(x_0)}(x) = F^{n,G(x_0)}(x[0..l(n)-1])$ for $|x| > l(n)$. It should be noted that $F^{n,G(x_0)}$ is different from the martingale $F^{G(x_0)}$ for the basic open set $\mathbf{C}_{G(x_0)}$.

Let $\{F^U_n\}_{n\in N}$ be the martingale ensemble for the uniform distribution with $F^U_n = F^U$ for all n. Then the pseudorandom generator concept [6], [28] could be rephrased in terms of martingales as follows.

Definition 3.2: Let $l : N \to N$ with $l(n) > n$ for all $n \in N$. A pseudorandom generator is a polynomial-time algorithm G with the following properties:

1) $|G(x)| = l(|x|)$ for all $x \in \Sigma^*$.
2) The martingale ensembles $\left\{F^G_{l(n)}\right\}_{n\in N}$ and $\left\{F^U_{l(n)}\right\}_{n\in N}$ are computationally indistinguishable.

IV. Long Pseudorandom sequences

In cryptography, long pseudorandom sequences are often generated using a cryptographic hash function or a block cipher. For example, in SecureRandom class of Java Cryptography Architecture [14], [15], the API SHA1PRNG generates a long pseudorandom sequence using the SHA1 hash function. NIST SP 800-90A [1] recommends three categories of random bit generators: hash function based, block cipher based, and elliptic curve based.

To evaluate the quality of pseudorandom sequences, NIST SP800-22 Revision 1A [22] (software package available at [20]) proposed a statistical test suite for pseudorandom number generators that includes 15 tests: frequency (monobit), number of 1-runs and 0-runs, longest-1-runs, binary matrix rank, discrete Fourier transform, template matching, Maurer's "universal statistical" test, linear complexity, serial test, the approximate entropy, the cumulative sums (cusums), the random excursions, and the random excursions variants. A sequence passes a test if the calculated P-value α for the sequence is large than a pre-selected threshold value from $[0.001, 0.01]$,

Computational complexity based pseudorandom sequences have been studied extensively in the literature. For example, p-random sequences are defined by taking each polynomial time computable martingale as a statistical test.

Definition 4.1: (Schnorr [23], Lutz [17], and Wang [27]) An infinite sequence $\xi \in \Sigma^\infty$ is p-random (polynomial time random) if for any polynomial time computable martingale F, F does not succeed on ξ.

A sequence $\xi \in \Sigma^\infty$ is Turing machine computable if there exists a Turing machine M to calculate the bits $\xi[0]$, $\xi[1]$, $\cdots$. In the following, we prove a theorem which says that, for each Turing machine computable non p-random sequence ξ, there exists a martingale F such that the process of F succeeding on ξ can be efficiently observed in time $O(n^2)$. The theorem is useful in the characterizations of p-random sequences and in the characterization of LIL-test waiting period.

Theorem 4.2: For a sequence $\xi \in \Sigma^\infty$ and a polynomial time computable martingale F, F succeeds on ξ iff there exists a martingale F' and a non-decreasing $O(n^2)$-time computable (with respect to the unary representation of numbers) function from $h : N \to N$ such that $F'(\xi[0..n-1]) \geq h(n)$ for all n.

Proof. In order to construct a martingale F' and a non-decreasing function $h : N \to N$ to satisfy the condition of the theorem, we first construct a martingale F' and a polynomial time computable function $d : \Sigma^* \to N$ such that,

1) For all $x \sqsubseteq y$, $d(x) \leq d(y)$ and $F'(x) \geq d(x)$.
2) For any sequence $\xi \in \Sigma^\infty$, if F succeeds on ξ then we have $\lim_n d(\xi[0..n-1]) = \infty$.

We construct d and F' by induction. Without loss of generality, we may assume that $F(\lambda) = 1$. First let $F'(\lambda) = F(\lambda) = 1$ and $d(\lambda) = F(\lambda) - 1 = 0$.

For induction, assume that $F'(x)$ and $d(x)$ have been defined for all $|x| \leq n$. Fix a string x of length n and $b \in \Sigma$, let $l(xb) = \frac{F(xb)}{F(x)}$ if $F(x) \neq 0$ and let $l(xb) = 0$ otherwise. For the definition of $F'(xb)$ and $d(xb)$, we distinguish the following two cases.

1) $d(x) + 1 \geq F'(x)$: Let $F'(xb) = d(x) + (F'(x) - d(x))l(xb)$ and $d(xb) = d(x)$.
2) $d(x) + 1 < F'(x)$: Let $F'(xb) = d(x) + 1 + (F'(x) - d(x) - 1)l(xb)$ and $d(xb) = d(x) + 1$.

It is straightforward that both F' and d are polynomial time computable, F' is a martingale, $d(x) \leq d(y)$ for $x \sqsubseteq y$, and $F'(x) > d(x)$ for all $x \in \Sigma^*$. Next we show that for strings $x, y \in \Sigma^*$, if $d(x) < F'(x) \leq d(x)+1$ and $F'(xy') \leq d(x)+1$ for all $y' \sqsubseteq y$, then we have

$$F'(xy) = \frac{F(xy)}{F(x)} \cdot (F'(x) - d(x)) + d(x). \qquad (3)$$

We use induction on y to prove (3). If $y \in \Sigma$, then (3) follows from the construction. Assume that that (3) holds for $y \in \Sigma^*$ and $F'(xy) \leq d(x) + 1$. Then, by the construction, $d(xy) = d(x)$ and

$$\begin{aligned} F'(xyb) &= d(xy) + (F'(xy) - d(xy))l(xyb) \\ &= d(xy) + (F'(xy) - d(x))\tfrac{F(xyb)}{F(xy)} \\ &= d(x) + \left(\tfrac{F(xy)}{F(x)} \cdot (F'(x) - d(x))\right) \cdot \tfrac{F(xyb)}{F(xy)} \\ &= d(x) + \tfrac{F(xyb)}{F(x)} \cdot (F'(x) - d(x)). \end{aligned}$$

where $b = 0, 1$. Thus (3) holds for yb and the induction is complete.

Next we show that for a sequence $\xi \in \Sigma^\infty$, if F succeeds on ξ, then $\lim_n d(\xi[0..n-1]) = \infty$. We prove by induction that, for each $k \in N$, there exists $n \in N$ such that $d(\xi[0..n-1]) > k$. By the construction, $d(\lambda) \geq 0$.

Assume that $k+1 \geq F'(\xi[0..n_1-1]) > d(\xi[0..n_1-1]) = k$ for some $n_1 \in N$. Then, by (3),

$$F'(\xi[0..n-1]) = d(\xi[0..n_1-1]) + \frac{F(\xi[0..n-1])}{F(\xi[0..n_1-1])} \cdot (F'(\xi[0..n_1-1]) - d(\xi[0..n_1-1]))$$

for $n \geq n_1$ until $F'(\xi[0..n-1]) > d(\xi[0..n_1-1]) + 1 = k+1$. Since F succeeds on ξ, there exists $n_2 > n_1$ such that

$$F(\xi[0..n_2-1]) > \frac{F(\xi[0..n_1-1])}{F'(\xi[0..n_1-1]) - d(\xi[0..n_1-1])}.$$

Hence there exists $n_3 \leq n_2$ such that

$$F'(\xi[0..n_3-1]) > d(\xi[0..n_1-1]) + 1 = k+1$$

and $d(\xi[0..n_3]) \geq k+1$.

Now we are ready to construct the non-decreasing function h from d by induction. Let $h(0) = 0$ and assume that $h(n)$ is defined already. Using the Turing machine M to search for a string $x \sqsubseteq \xi[0..n]$ such that $d(x) \geq h(|x|) + 1 = h(n) + 1$. If such an x is found in n steps, then let $h(s+1) = h(s) + 1$. Otherwise let $h(s+1) = h(s)$.

It is straightforward that h is an n^2-time computable (with respect to the unary representation of numbers), unbounded, nondecreasing function and $F'(\xi[0..n-1]) \geq h(n)$ for almost all n. This completes the proof of the Theorem. □

In the following, we establish the relationship between computational indistinguishability and p-randomness. Fix a standard polynomial time computable and invertible pairing function $\langle \cdot, \cdot \rangle : N \times N \to N$ such that, for each $i \in N$, there is a real $\alpha(i) > 0$ satisfying

$$|\{m : \langle i, m \rangle \leq 2^n\}| \geq \alpha(i) \cdot 2^n \text{ for almost all } n.$$

For each $i \in N$, use $\langle i, \cdot \rangle$ as a selection function to obtain a subsequence from ξ:

$$\xi_i = \xi[\langle i, 0 \rangle]\xi[\langle i, 1 \rangle]\xi[\langle i, 2 \rangle]\xi[\langle i, 3 \rangle] \cdots$$

In order to establish the relationship between complexity theoretic pseudorandom concepts and cryptographic indistinguishability concepts, we first define a martingale ensemble based on the series of sequences $\xi_0, \xi_1, \xi_2, \cdots$ derived from ξ. Let $r : N \to N^+$ be a non-decreasing function and $\{F^r_{n,\xi}\}_{n \in N}$ be a martingale ensemble defined by

$$F^r_{n,\xi}(x) = 2^n \cdot \frac{|\{\xi_i : x \sqsubseteq \xi_i, i < r(n)\}|}{r(n)}$$

for $x \in \Sigma^n$ and $F^r_{n,\xi}(x)$ for other $x \in \Sigma^*$ is defined correspondingly according to martingale ensemble requirements.

For the sake of convenience and completeness of the description, we present the following theorem in two parts: Martin-Löf randomness based result and p-randomness based result. For those who are not familiar with Martin-Löf randomness concepts, they may skip part one of the theorem or check [17], [23], [27] for details.

Theorem 4.3:
1) For a Martin-Löf random sequence $\xi \in \Sigma^\infty$, there exists a non-decreasing function $r(n)$ such that the martingale ensembles $\{F^r_{n,\xi}\}_{n \in N}$ and $\{F^U_n\}_{n \in N}$ are computationally indistinguishable.
2) For a p-random sequence $\xi \in \Sigma^\infty$, there exist a real number $\varepsilon > 0$, a polynomial $p(n)$, and a non decreasing function $r(n)$ with $2^{\varepsilon n} \leq r(n) \leq 2^{p(n)}$ such that the martingale ensembles $\{F^r_{n,\xi}\}_{n \in N}$ and $\{F^U_n\}_{n \in N}$ are computationally indistinguishable.

Sketch of Proof. We describe the proof for part one of the theorem. The proof arguments for part two are similar to that of part one with resource constraints and the details could be found in the full version of this paper. We first note that if a sequence $\xi \in \Sigma^\infty$ is Martin-Löf random, then all of the sequences $\xi_0, \xi_1, \xi_2, \cdots$ are Martin-Löf random. For a contradiction, assume that there is a Turing machine approximable martingale F that succeeds on ξ_i for some $i \in N$. Then we can easily convert the martingale F to another Turing machine approximable martingale F' that succeeds on ξ, which is a contradiction. Similarly, we can show that for any $n > 0$, the following sequence β is Martin-Löf random:

$$\beta = \xi_0[0..n-1]\xi_1[0..n-1]\xi_2[0..n-1] \cdots$$

By the fact that Martin-Löf random sequences are normal with respect to any given $n > 0$, we have $\lim_{n \to \infty} F^r_{n,\xi}(x) = 1 = F_U(x)$ for all $x \in \Sigma^*$. In other words, appropriate non-decreasing function r could be chosen such that the two given ensembles are computationally indistinguishable. □

The readers may wonder whether the other directions of Theorem 4.3 hold also. By Ville's construction of the counter example for a von Mises' "Kollektiv" that does not satisfy the law of the iterated logarithm, the other directions of Theorem 4.3 do not hold. Indeed, if we choose independent Martin-Löf random sequences $\xi_0, \xi_2, \xi_4, \cdots$ and let $\xi_{2i+1} = \xi_{2i}$ for all $i \in N$, then it can be shown that the martingale ensembles $\{F^r_{n,\xi}\}_{n\in N}$ and $\{F^U_n\}_{n\in N}$ are computationally indistinguishable for an appropriately chosen non-decreasing function r though the resulting sequence ξ is not random in any sense. It is an *open* question whether it is feasible to build some kind of equivalence between the cryptographic indistinguishability concepts and the complexity theoretic randomness concepts.

V. Stochastic Properties of Pseudorandom Sequences

It is shown in [27] that p-random sequences are stochastic in the sense of von Mises and satisfy common statistical laws such as the law of the iterated logarithm. It is not difficult to show that all p-random sequences pass the NIST SP800-22 [22] tests for $\alpha = 0.01$ since each test in [22] could be converted to a polynomial time computable martingale which succeeds on all sequences that do not pass this test. However, none of the sequences generated by pseudorandom generators are p-random since from the generator algorithm itself, a martingale can be constructed to succeed on sequences that it generates.

Since there is no efficient mechanism to generate p-random sequences, pseudorandom generators are commonly used to produce long sequences for cryptographic applications. While the required uniformity property (see NIST SP800-22 [22]) for pseudorandom sequences is equivalent to the law of large numbers, the scalability property (see [22]) is equivalent to the invariance property under the operation of "admissible place selection" rules. Since p-random sequences satisfy common statistical laws, it is reasonable to expect that pseudorandom sequences produced by pseudorandom generators satisfy these laws also (see, e.g., [22]).

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 says that, for a pseudo random sequence ξ, the limit of $\frac{S(\xi[0..n-1])}{n}$ is $\frac{1}{2}$, which corresponds to the frequency (Monobit) test in NIST SP800-22 [22]. But it says nothing about the reduced deviation $S^*(\xi[0..n-1])$. It is intuitively clear that, for a pseudorandom sequence ξ, $S^*(\xi[0..n-1])$ will sooner or later take on arbitrary large values (though slowly). The law of the iterated logarithm (LIL), which was first discovered by Khintchine [16], gives an optimal upper bound $\sqrt{2\ln\ln n}$ for the fluctuations of $S^*(\xi[0..n-1])$. It was showed in Wang [27] that this law holds for p-random sequences also.

Theorem 5.1: (LIL for p-random sequences [27]) For a sequence $\xi \in \Sigma^\infty$, let

$$S_{lil}(\xi[0..n-1]) = \frac{2\sum_{i=0}^{n-1}\xi[i] - n}{\sqrt{2n\ln\ln n}}.$$

Then for each p-random sequence $\xi \in \Sigma^\infty$ we have both

$$\limsup_{n\to\infty} S_{lil}(\xi[0..n-1]) = 1 \text{ and } \liminf_{n\to\infty} S_{lil}(\xi[0..n-1]) = -1.$$

In other words, if we let

$$\mathbf{Y}_k = \left\{\xi \in \Sigma^\infty : S_{lil}(\xi[0..n-1]) > 1 + \frac{1}{k} \text{ infinitely often}\right\},$$

and

$$\mathbf{X}_k = \left\{\xi \in \Sigma^\infty : S_{lil}(\xi[0..n-1]) > 1 - \frac{1}{k} \text{ finitely often}\right\},$$

then there is a polynomial time computable martingale F_{lil} that succeeds on all sequences in $(\bigcup_{k=1}^\infty \mathbf{X}_k)\bigcup(\bigcup_{k=1}^\infty \mathbf{Y}_k)$.

VI. The LIL tests

Theorem 5.1 shows that pseudorandom sequences should satisfy the law of the iterated logarithm (LIL). In particular, for small values of k, there are simple martingales that succeed on all sequences in $\mathbf{X}_k$ of Theorem 5.1. For example, we can show that there is an $O(n^5)$-time computable martingale that succeeds on all sequences in $\mathbf{X}_{10}$ (see full version of this paper for details). Thus it is expected that pseudorandom sequences for cryptographic applications should not be included in $\mathbf{X}_k$ with $k \le 10$. Though NIST SP800-22 test suite does not include a test for LIL, it includes three tests that are related to cusum test [8]: "the cumulative sums (cusums) test", "the random excursions test", and "the random excursions variants". The limiting distribution of cusum test is related to Wiener process. But a sequence that passes the cusum test does not necessarily satisfy LIL. Our experiments in Section VII confirms this fact. We first propose two LIL based tests.
Weak LIL Test: For given $\alpha \in (0, 0.25]$ and $n_1 < n_2$, we say that a sequence ξ does not pass the weak (α, n_1, n_2)-LIL test if $-1+\alpha < S_{lil}(\xi[0..n-1]) < 1-\alpha$ for all $n \in [n_1, n_2]$.
Strong LIL Test: For given $\alpha \in (0, 0.25]$ and $n_1 < n_2$, we say that a sequence ξ does not pass the strong (α, n_1, n_2)-LIL test if $S_{lil}(\xi[0..n-1]) > -1+\alpha$ for all $n \in [n_1, n_2]$ or $S_{lil}(\xi[0..n-1]) < 1-\alpha$ for all $n \in [n_1, n_2]$.

By the definition, a sequence ξ passes the weak (α, n_1, n_2)-LIL test if S_{lil} reaches either $1-\alpha$ or $-1+\alpha$ in the testing period $[n_1, n_2]$, while a sequence ξ passes the strong (α, n_1, n_2)-LIL test if S_{lil} reaches both $1-\alpha$ and $-1+\alpha$ in the testing period $[n_1, n_2]$.

In the following, we provide justifications for these two tests. The DeMoivre-Laplace theorem is a normal approximation to the binomial distribution, which says 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 give the variant of DeMoivre-Laplace limit theorem in the following. To shorten our notations, we will use $S^*(\xi, n) = S^*(\xi[0..n-1])$ and

$S_{lil}(\xi, n) = S_{lil}(\xi[0..n-1])$ in the remaining part of the paper unless stated otherwise.

Theorem 6.1: (DeMoivre-Laplace [10, Chapter VII.7, p193]) Let $u(n) \to \infty$ and $u(n)^3/\sqrt{n} \to 0$. Then we have

$$Prob[\{\xi : S^*(\xi, n) > u(n)\}] \simeq (u(n)\sqrt{2\pi})^{-1} e^{-u(n)^2/2}.$$

We first show that a uniformly chosen sequence should pass the weak (α, n_1, n_1+1)-LIL test with a high probability.

Theorem 6.2:
1) Let $\alpha = 0.01$, $n_1 = 3 \times 2^{24}$ and $n_2 = n_1+1$. A uniformly chosen sequence $\xi \in \{0,1\}^\infty$ passes the weak (α, n_1, n_2)-LIL test with probability at least 2%.
2) Let $\alpha = 0.1$, $n_1 = 3 \times 2^{24}$ and $n_2 = n_1 + 1$. A uniformly chosen sequence $\xi \in \{0,1\}^\infty$ passes the weak (α, n_1, n_2)-LIL test with probability at least 3.6%.

Proof. For given α, n_1, n_2, let $\mathcal{WR}_{(\alpha,n_1,n_2)}$ be the set of sequences that pass the weak (α, n_1, n_2)-LIL test. That is,

$$\mathcal{WR}_{(\alpha,n_1,n_2)} = \{\xi : \exists n \in [n_1, n_2]\, (|S_{lil}(\xi, n)| \geq 1-\alpha)\}.$$

Furthermore, let $u(n) = (1-\alpha)\sqrt{2\ln\ln n}$ and

$$\mathcal{R}_{(n,\alpha)} = \{\xi : |S^*(\xi, n)| \geq u(n)\}.$$

Then we have $\mathcal{R}_{(n_1,\alpha)} \subseteq \mathcal{WR}_{(\alpha,n_1,n_2)}$. By Theorem 6.1, we have

$$\begin{aligned} Prob\left[\mathcal{WR}_{(\alpha,n_1,n_2)}\right] &\geq Prob[\mathcal{R}_{(n_1,\alpha)}] \\ &\simeq 2 \times (u(n_1)\sqrt{2\pi})^{-1} e^{-(1-\alpha)^2 \ln\ln n_1} \\ &= \frac{1}{(1-\alpha)\sqrt{\pi \ln\ln n_1}(\ln n_1)^{(1-\alpha)^2}}. \end{aligned} \tag{4}$$

Note that $\ln n_1 \simeq 17.7341$ and $\ln\ln n_1 \simeq 2.8755$. By substituting $n_1 = 3 \times 2^{24}$ and $\alpha = 0.01$ in (4), we get

$$Prob\left[\mathcal{WR}_{(\alpha,n_1,n_2)}\right] \geq 0.02.$$

By substituting $n_1 = 3 \times 2^{24}$ and $\alpha = 0.1$ in (4), we get

$$Prob\left[\mathcal{WR}_{(\alpha,n_1,n_2)}\right] \geq 0.036.$$

This completes the proof of the Theorem. □

Furthermore, if we choose $\alpha = 0.01$, $n_1 = 3 \times 2^{24}$ (= 6MB) and $n_2 = 5 \times 2^{31}$ ($\simeq$ 1.34GB), then $Prob\left[\mathcal{WR}_{(\alpha,n_1,n_2)}\right] \gg 0.02$. Indeed, by Theorem 5.1, there exists an efficient martingale that succeeds on all sequences in $\overline{\mathcal{WR}_{(\alpha,n_1,n_2)}}$. By Theorem 4.2, it can be effectively observed that this martingale succeeds on $\overline{\mathcal{WR}_{(\alpha,n_1,n_2)}}$. Based on these facts and some involved analysis, it could be shown that $Prob\left[\mathcal{WR}_{(\alpha,n_1,n_2)}\right] \geq 0.14$. Similarly, for $\alpha = 0.1$, $n_1 = 3 \times 2^{24}$ (= 6MB) and $n_2 = 5 \times 2^{31}$ ($\simeq$ 1.34GB), we have $Prob\left[\mathcal{WR}_{(\alpha,n_1,n_2)}\right] \geq 0.2556$. It should be noted that the above probabilities 0.14 and 0.2556 are based on the most conservative calculations. We *conjecture* that these probabilities could be improved to at least 0.5 by more accurate calculations.

Next we show that a uniformly chosen sequence should pass the strong LIL test with a non-negligible probability.

Theorem 6.3: Let $\alpha = 0.2$, $n_1 = 3 \times 2^{24}$, and $n_2 = 128n_1$. With a probability of at least 0.2652%, a uniformly chosen sequence $\xi \in \{0,1\}^\infty$ passes the strong (α, n_1, n_2)-LIL test.

Proof. For given α, n_1, n_2, let $\mathcal{SR}_{(\alpha,n_1,n_2)}$ be the set of sequences that pass the strong (α, n_1, n_2)-LIL test. That is,

$$\mathcal{SR}_{(\alpha,n_1,n_2)} = \left\{\xi : \begin{array}{l} \exists n \in [n_1, n_2]\, (S_{lil}(\xi, n) \geq 1-\alpha) \\ \exists n \in [n_1, n_2]\, (S_{lil}(\xi, n) \leq -1+\alpha) \end{array}\right\}.$$

Furthermore, let $u(n) = (1-\alpha)\sqrt{2\ln\ln n}$ and

$$\mathcal{T}_{(n,\alpha)} = \left\{\xi : \begin{array}{l} S^*(\xi, n) \geq u(n) \text{ and} \\ S^*(\xi, 128n) \leq -u(128n) \end{array}\right\}.$$

Then we have $\mathcal{T}_{(n_1,\alpha)} \subseteq \mathcal{SR}_{(\alpha,n_1,n_2)}$. Now let

$$\mathcal{D}_n = \left\{\xi : \frac{S^*(\xi[n..128n-1])}{\sqrt{2\ln\ln(127n)}} \leq -0.95\right\}$$

and

$$\mathcal{E}_n = \left\{\xi : 0.8 \leq \frac{S^*(\xi, n)}{\sqrt{2\ln\ln n}} \leq 1.5\right\}.$$

For any sequence $\xi \in \mathcal{D}_{n_1} \cap \mathcal{E}_{n_1}$, we have

$$2S(\xi[n_1..128n_1-1]) - 127n_1 \leq -0.95\sqrt{2 \times 127n_1 \ln\ln(127n_1)} \tag{5}$$

and

$$2 \times S(\xi, n_1) - n_1 \leq 1.5\sqrt{2n_1 \ln\ln n_1} \tag{6}$$

By adding (5) and (6) together and dividing it by $\sqrt{128n_1}$, we get

$$\begin{aligned} S^*(\xi, 128n_1) &\leq \tfrac{1.5\sqrt{2n_1\ln\ln n_1} - 0.95\sqrt{2\times127n_1\ln\ln(127n_1)}}{\sqrt{128n_1}} \\ &\simeq (25520.1312 - 189639.4950)/80264.8799 \\ &= -2.0447 \\ &< -1.9975 \\ &\simeq -0.8 \times 2.4969 \\ &\simeq -u(128n_1) \end{aligned} \tag{7}$$

By (7), we have $\xi \in \mathcal{T}_{(n_1,\alpha)}$. In other words, $\mathcal{D}_{n_1} \cap \mathcal{E}_{n_1} \subseteq \mathcal{T}_{(n_1,\alpha)}$. Thus we have

$$\begin{aligned} Prob[\mathcal{SR}_{(\alpha,n_1,n_2)}] &\gg 2 \times Prob[\mathcal{T}_{(n_1,\alpha)}] \\ &\geq 2 \times Prob[\mathcal{D}_{n_1} \cap \mathcal{E}_{n_1}] \\ &\geq 2 \times Prob[\mathcal{D}_{n_1}] \cdot Prob[\mathcal{E}_{n_1}] \end{aligned} \tag{8}$$

By Theorem 6.1 and by the symmetry property for the distributions of 0s and 1s in ξ, we have

$$\begin{aligned} Prob\left[\mathcal{D}_{n_1}\right] &\simeq (u(127n_1)\sqrt{2\pi})^{-1} \times e^{-0.95^2 \ln\ln(127n_1)} \\ &= \frac{1}{0.95\sqrt{\pi \ln\ln(127n_1)}(\ln(127n_1))^{0.95^2}} \\ &\simeq 0.0201896 \end{aligned} \tag{9}$$

and

$$\begin{aligned}
Prob\,[\mathcal{E}_{n_1}] &\simeq (0.8\sqrt{2\pi\cdot 2\ln\ln n_1})^{-1}\times e^{-0.8^2\ln\ln(n_1)} \\
&\quad -(1.5\sqrt{2\pi\cdot 2\ln\ln n_1})^{-1}\times e^{-1.5^2\ln\ln(n_1)} \\
&= \frac{1}{0.8\sqrt{\pi\ln\ln(n_1)}(\ln(n_1))^{0.8^2}} \\
&\quad -\frac{1}{1.5\sqrt{\pi\ln\ln(n_1)}(\ln(n_1))^{1.5^2}} \\
&\simeq 0.0660299-0.0003437 \\
&= 0.0656862
\end{aligned} \tag{10}$$

By substituting (9) and (10) into (8), we get

$$\begin{aligned}
Prob[\mathcal{SR}_{(\alpha,n_1,n_2)}] &\gg 2\times 0.0656862\times 0.0201896 \\
&\simeq 0.002652
\end{aligned}$$

This completes the proof of the Theorem. □

The probability 0.2652% in Theorem 6.3 is based on the most conservative calculations. We *conjecture* that this probability could be improved to at least 5% by a more accurate calculation. In other words, a randomly chosen sequence should pass the strong (α, n_1, n_2)-LIL with a high probability.

VII. Experimental results

We have carried out weak LIL tests on the following pseudorandom sequence generators: SHA1PRNG (Java), NIST DRBG [22], and Fortuna-PRNG (Schneier and Ferguson [11]). For the weak LIL-test, we used the parameters $\alpha = 0.01$, $n_1 = 3 \times 2^{24}$ ($=$ 6MB) and $n_2 \geq 5 \times 2^{31}$ ($\simeq$ 1.34GB). In a summary, our experimental results provide evidence that sequences generated by commonly used pseudorandom generators (e.g., Java SHA1PRNG and hash function based DRBG in NIST SP800-90A) do not pass the weak (α, n_1, n_2)-LIL tests (thus they can not pass strong LIL tests either) with above parameters. On the other hand, our experiments show that sequences generated by NIST ECC-DRBG pass the weak (α, n_1, n_2)-LIL tests with $\alpha = 0.01$, $n_1 = 3 \times 2^{24}$ and $n_2 = 15 \times 2^{26} \simeq$ 120MB. We do not know whether sequences generated by NIST ECC-DRBG can pass the weak (α, n_1, n_2)-LIL tests with $\alpha = 0.01$, and larger n_1, n_2 (e.g., $n_1 \geq$ 120MB) since we do not have resources to generate large number of longer sequences based on NIST ECC-DRBG. Furthermore, we have not got sufficient resources to carry out the strong LIL test on ECC-DRBG either.

Following the comments after Theorem 6.2, sequences that do not pass the weak (α, n_1, n_2)-LIL tests could be distinguished from uniformly chosen sequences with a probability of at least 14%. In the random oracle model, the adversary is allowed to make a polynomial number of queries to the random oracle. Thus in the instantiated protocol the adversary may trigger the instantiated random oracle to output 6MB to 1.34GB bits (which is certainly feasible). The above results show that protocols such as RSA-OAEP [3], [7], [24] in SSL with security proof using the random oracle model should not be instantiated with SHA1PRNG (Java) and hash function based DRBG (NIST SP800-90A) pseudorandom generators.

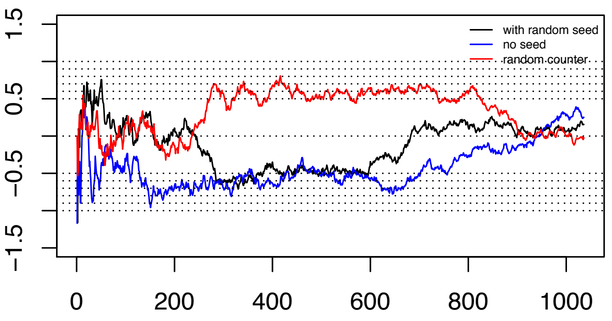
We have also run NIST SP800-22 tests [22], [20] on sequences that we have generated. The test tool [20] only checks the first 1,215,752,192 bits ($\simeq$145MB) of sequences that we provided since the software uses 4-byte `int` data type for integer variables. The initial 145MB of each sequence that we have generated passes NIST tests with P-values larger than $\alpha = 0.01$ except for the "longest run of ones in a block" test which failed for several sequences. NIST recommends that a test fails if the P-value is less than a pre-selected threshold value $\alpha \in [0.001, 0.01]$.

Though weak LIL tests require the value $S_{lil}(\xi[0..n-1])$ to reach either -1 or 1 in the given period. It does not require the fluctuation scale for $S_{lil}(\xi[0..n-1])$ when n increases. By Theorem 5.1, for a true random sequence ξ, $S_{lil}(\xi[0..n-1])$ should take large fluctuation scales even if the testing period $[n_1, n_2]$ is not long enough to carry out the strong LIL tests. No sequence (except NIST ECC-DRBG generated sequences) that we have generated have the large fluctuation scale for $S_{lil}(\xi[0..n-1])$. Figure 11 presents a sequence's S_{lil} curve that has a close looking to a true pseudorandom sequence's S_{lil} curve with a large fluctuation scale.

A. Java SHA1PRNG API based sequences

The pseudorandom generator SHA1PRNG API in Java generates sequences $\mathrm{SHA1}'(s,0)\mathrm{SHA1}'(s,1)\cdots$, where s is an optional seeding string of arbitrary length, the counter i is 64 bits long, and $\mathrm{SHA1}'(s,i)$ is the first 64 bits of $\mathrm{SHA1}(s,i)$. When no seed is provided, Java provides random

Fig. 1. Typical results for Java SHA1PRNG API



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 1.34GB long. The LIL test is then run on these sequences and we observed similar trend curves for all sequences. Specifically, we observed that $S_{lil}(\xi[0..n-1]) \leq 0.5$ for $n >$ 5MB on average and then $-0.75 \leq S_{lil}(\xi[0..n-1]) \leq 0.3$ for $n >$ 6MB 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 $\text{SHA1}'(x_1)\text{SHA1}'(x_2)\cdots$ using SHA1PRNG API without seeds, where $x_1x_2x_3\cdots$ are pseudorandom sequences generated by AES128 with different keys and each x_i is 64 bits long. We then run the weak LIL test and observed that S_{lil} values for these sequences take larger values though they still fail the weak LIL test. In a summary, for three kinds of sequences (with seeds, without seeds, and with random counters) that we have generated, none of them passes the weak (α, n_1, n_2)-LIL test with $\alpha = 0.1$, $n_1 = 6$MB, and $n_2 = 1.34$GB.

Figure 1 shows three typical LIL test results. The black line is for a sequence with seed $s = \text{SHA256}(\texttt{0xD2029649D2029649})$. The blue line is for a sequence without a seed. The red line is for a sequence without a seed but with counters replaced by random strings from $x_1x_2x_3\cdots = \text{AES}(k,0)\text{AES}(k,1)\cdots$, where $k = \texttt{0x0E14533E1F056F7C7E192B3F4C4D7E6F}$. To reduce the size of the figure, we use the scale $10000n^2$ for the x-axis. In other words, Figure 1 shows the values $S_{lil}(\xi[0..10000n^2-1])$ for $1 \le n \le 1037$. The readers may ask: does $S_{lil}(\xi[0..m-1])$ 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. 2. More results for Java SHA1PRNG API

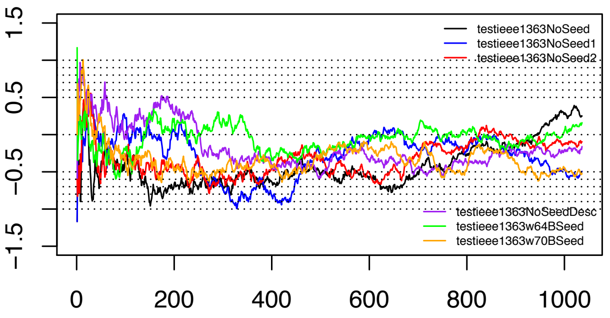


Figure 2 shows another result on six sequences generated by SHA1PRNG API using the same scale $10000n^2$ for the x-axis. The first three lines (black, blue, red) are for sequences that are generated by SHA1PRNG API without seeds. The fourth line (purple) is for a sequence that is generated by SHA1PRNG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

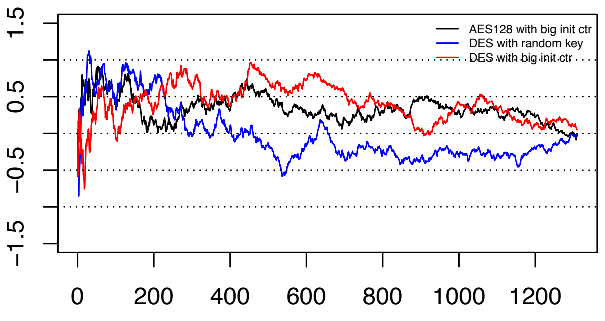
B. NIST SP800-90A based sequences

NIST SP800.90A [1] 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.

1) Block cipher based DRBG: For block cipher based generators, we generated one hundred of sequences of the format $\text{AES128}(k,V)\text{AES128}(k,V+1)\cdots$ and $\text{DES}(k,V)\text{DES}(k,V+1)\cdots$ 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 [1]. Each sequence is 2.15GB 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 VII-A 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. Thus they do not pass the weak (α, n_1, n_2)-LIL test with $\alpha = 0.01$, $n_1 = 6$MB, and $n_2 = 2.15$GB.

As an example, Figure 3 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 $E_2(\cdot)$

Fig. 3. Results for AES and DES based generators

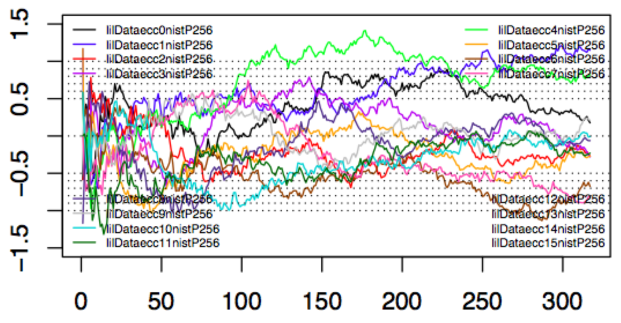


denote AES128 and DES encryption functions respectively. These lines are for sequences $E_1(k, ctr_0)E_1(k, ctr_0+1)\cdots$, $E_2(k_1,0)E_2(k_1,1)\cdots$, and $E_2(k_2, ctr_0)E_2(k_2, ctr_0+1)\cdots$ respectively, where k, k_1, k_2 are random keys and ctr_0 is a random value of 8 bytes.

2) 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 $R_i = s_iQ$ 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-

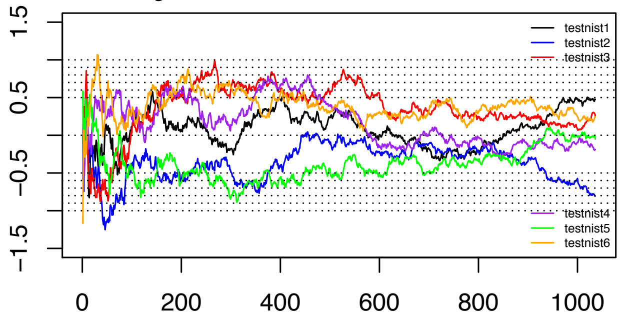
384 and P-521). After outputting the random bits, the internal state of the generator is updated to $s_{i+1} = x(s_i P)$ where $x(s_i P)$ denotes the x-coordinate of the elliptic curve point $s_i P$. In our experiments, we generated 16 random sequences lilDataecc0nistP256, $\cdots$, lilDataecc15nistP256 using the curve P-256 with the initial states s_0 =SHA(0), $\cdots$, 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 120MB long. Since it takes 26 hours 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. Figure 4 shows the test results for these 16 random sequences. For these relatively "short" sequences (120MB long each), they pass the weak (α, n_1, n_2)-LIL test with $\alpha = 0.01$, $n_1 = 6$MB, and $n_2 = 120$MB. We have not got enough resources to carry out the weak LIL test for large n_1, n_2 (e.g., $n_1 \geq 120$MB) and to carry out the strong LIL test on NIST ECC-DRBG .

Fig. 4. Results for NIST ECC-DRBG



3) Hash function based DRBG: For hash function based DRBG in NIST SP800.90A, a hash function G is used to generate sequences $G(V)G(V+1)G(V+2)\cdots$ where V is a seedlen-bit counter that is derived from the secret seeds, seedlen is 440 for SHA1, and the value of V is revised after at most 2^{19} bits are output. We generated several hundreds

Fig. 5. Results for NIST SHA1-DRBG



of sequences with randomly chosen seeds for SHA1 and SHA256 based DRBG. None of these sequences pass the weak (α, n_1, n_2)-LIL test with $\alpha = 0.01$, $n_1 = 6$MB, and $n_2 = 1.34$GB ($n_2 = 2.15$GB for SHA256). As an example, Figure 5 shows the test results on six typical sequences generated by SHA1-based DRBG with the scale $10000n^2$ for the x-axis.

C. 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 to 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, $S_{lil}(\xi[0..n-1]) \leq 0$ for SHA1/SHA256 based sequences and $S_{lil}(\xi[0..n-1]) \geq 0$ 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. 6. Results for non-NIST SHA1/SHA2/SHA3-DRBG

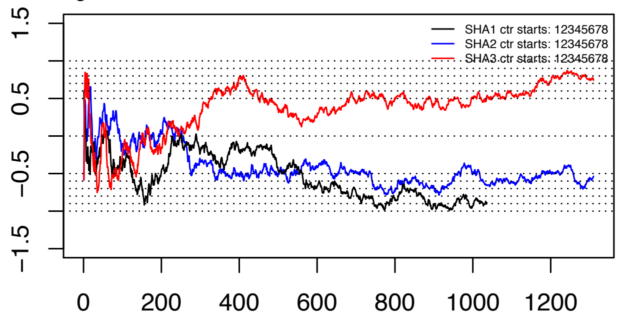
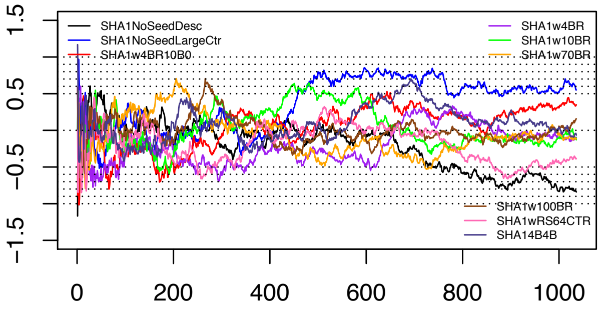


Figure 6 shows the test results on three typical sequences generated by SHA1, SHA256, and Keccak256 using the scale $1000n^2$ for the x-axis. These sequences are generated with empty seeds and with a 32-bit counter starting at 12345678. The hash functions are called 2^{26} times. Thus the SHA1 based sequence is 1.34GB and the SHA256/Keccak256 based sequences are 2.15GB. It is observed that, for SHA1 and SHA256 based sequences, we have $S_{lil}(\xi[0..n-1]) \leq 0$ when n is sufficiently large and, for sequences generated using Keccak256, we have $S_{lil}(\xi[0..n-1]) \geq 0$ when n is sufficiently large.

Figure 7 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 1.34GB 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 $G_1(ctr_0)\cdots G_1(0)$ with a 4-byte decreasing counter that starts at $ctr_0 = 2^{26} - 1$. Line SHA1NoSeedLargeCtr is for the sequence $G_1(ctr_0)G_1(ctr_0+1)\cdots G_1(ctr_0+2^{26}-1)$ where ctr_0 is a random 4-byte integer. The line SHA1w4BR10B0 is for the sequence $G_1(s, v_0, 0)\cdots$ where s is a 4-byte

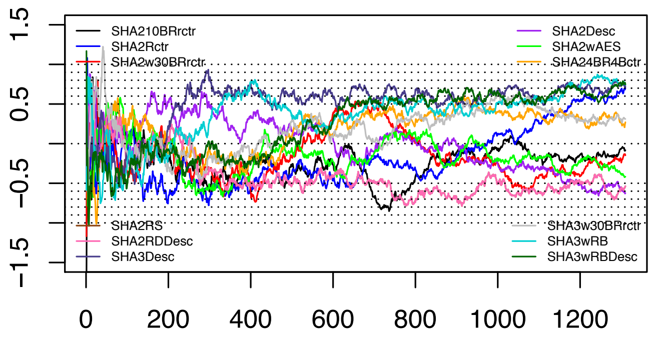
Fig. 7. Results for some sequences based on SHA1



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 $G_1(s,0)G_1(s,1)\cdots$ 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 $G_1(s,0)G_1(s,1)\cdots$ 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 $G_1(s,0)\cdots$ where s is a 70-byte random seed and ctr is a 4-byte counter starting at 0. The line SHA1w100BR is for $G_1(s,ctr_0)G_1(s,ctr_0+1)\cdots$ where s is a 100-byte random seed and ctr_0 is a 4-byte random integer. The line SHA1wRS64CTR is for the sequence $G_1(s,0)G_1(s,1)\cdots$ with a 8-byte counter and a 8-byte random seed s. The line SHA14B4B is for $G_1(s,ctr_0)G_1(s,ctr_0+1)\cdots$ where s is a 4-byte random seed and ctr_0 is a 4-byte random integer.

Figure 8 shows some LIL test results on 12 sequences based on SHA256 and Keccak256 hash functions using the scale $10000n^2$ for the x-axis. Each sequence is 2.15GB 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

Fig. 8. Results for some sequences based on SHA2/SHA3

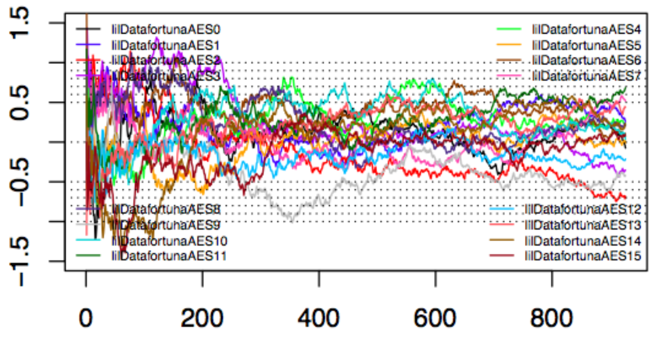


SHA210BRrctr is for the sequence $G_2(s,ctr_0)G_2(s,ctr_0+1)\cdots$ where ctr_0 is a 4-byte random integer and s is a 210-byte random seed. The line SHA2Rctr is for the sequence $G_2(ctr_0)G_2(ctr_0+1)\cdots$ where ctr_0 is a 4-byte random integer. Line SHA2w30BRrctr is for the sequence $G_2(s,ctr_0)G_2(s,ctr_0+1)\cdots$ where s is a 30-byte random seed and ctr_0 is a 4-byte random integer. The line SHA2Desc is for the sequence $G_2(2^{26}-1)G_2(2^{26}-2)\cdots G_2(0)$ 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 $x_0x_1\cdots$ is a pseudorandom sequence and x_i is 8 bytes. The line SHA24BR4Bctr is for the sequence $G_2(s,ctr_0)G_2(s,ctr_0+1)\cdots$ where ctr_0 is a 4-byte random integer and s is a 24-byte random seed. The line SHA2RS is for the sequence $G_2(s,0)G_2(s,1)\cdots$ where s is a 500-byte random seed the counter is 4 bytes. Line SHA2RDDesc is for the sequence $G_2(s,2^{26}-1)G_2(s,2^{26}-2)\cdots G_1(s,0)$ where s is a 100-byte random seed and the counter is 4 bytes. The line SHA3Desc is for the sequence $G_3(2^{26}-1)G_3(2^{26}-2)\cdots G_3(0)$ where ctr_0 is a 4-byte random integer. The line SHA3w30BRrctr is for the sequence $G_3(s,ctr_0)G_3(s,ctr_0+1)\cdots$ where s is a 30-byte random seed and ctr_0 is a 4-byte random integer. The line SHA3wRB is for $G_3(s,0)G_3(s,1)\cdots$ where s is a 4-byte random seed and the counter is 4 bytes. The line SHA3wRBDesc is for the sequence $G_3(s,2^{26}-1)G_3(s,2^{26}-2)\cdots G_3(s,0)$ where s is a 8-byte random seed and the counter is 4 bytes.

D. Fortuna PRNG

Fortuna pseudorandom number generator (Schneier and Ferguson [11]) uses block ciphers such as AES in counter mode and the key is changed each time after at most 1MB of data is generated. We uses AES-128 as the underlying block cipher to instantiate Fortuna PRNG. In total, we generated 100 sequences: fortunaAES0, $\cdots$, fortunaAES99. Each of these sequence is 1GB long. Specifically, for the generation of sequence fortunaAESi, we run AES-128 in counter mode and use keys SHA1$(i\|j)$ for $0\le j\le 2^{10}$. Each of the AES key SHA1$(i\|j)$ is used to encrypt 2^{16} consecutive counters. Figures 9 shows the LIL-test results for the sequences fortunaAES0, $\cdots$, lilDatafortunaAES15. In Figures 9, most of the curves lie strictly 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 weak LIL test. More testing is needed to confirm whether Fortuna PRNG passes the weak LIL-test.

Fig. 9. Results for Fortuna-PRNG



VIII. How to seed a generator?

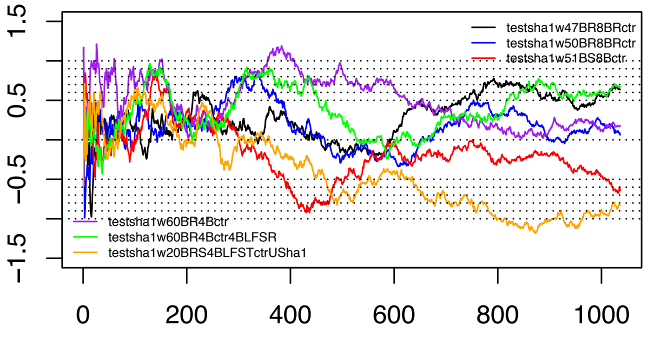
Results in Sections VII-B and VII-C show that the fluctuation scales of S_{lil} for SHA1/SHA256 and Keccak256 generated sequences are quite flat. In order to improve the S_{lil} fluctuation scale within the interval $[-1,1]$ for sequences generated by pseudorandom generators, we need a better seeding approach. In existing hash function designs (e.g., SHA1/SHA2/SHA3 [21], [5]), the input to the hash function is padded with a bit 1 followed by 0s (and the length of the message itself in case of SHA1 and SHA2) so that the size of the padded message is a multiple of the hash function message block size. The message blocks are then processed one by one and the hash values are updated correspondingly. If the combined inputs (based on seeds and counters) to the generators are small (e.g., smaller than 440 bits for SHA1 and SHA256) such as in DRBG [22], then each hash function call needs to process only one message block and there is no chance for the initial hash values (or internal state of the sponge function in SHA3) to be dynamically changed. Furthermore, if counter mode is used and consecutive counters are not significantly changed, then inputs to consecutive primitive function calls are almost identical (only a few bits of difference). In [22], one counter V can be used for at most 2^{19} bits of output. If the combined inputs (based on seeds and counters) to the generators are larger than 448 bits but smaller than 512 bits for SHA1/SHA256 based generators, then the padded inputs have the form M_1M_2 where M_2 consists mainly of the padded 0-bits. Thus for each hash function call, the hash function processes the same last message block M_2 with different first hash values (or internal hash function state). We conjecture that these "sparse" inputs to the generators may reduce the randomness property (or *reveal the non-randomness property*) of the underlying primitives. Thus it is reasonable to design a better seeding process for pseudorandom generators. In [22], the seeding information is used to derive a start counter V of seedlen bits which is 440 for SHA1/SHA2. The length of V is chosen in such a way that each hash function call will only have one message block to process. The value of V is revised after at most 2^{19}-bit output using $G(V+G(0\text{x}03||V)+C+reseed\_counter)$ where C contains the entropy of the original seeding. As we have observed in the experiments, the generated sequences show strong non-randomness properties with stable S_{lil} values. We recommend revising the seeding process in such a way that each hash function call has significantly different last message block and different internal hash algorithm state when the last message block is processed. Specifically, we recommend the following seeding approach with two choices for the value of $vLen$ which is defined using $seedln$ from [22].

Approach: The seeding information is converted to a series of values V_0, V_1, $\cdots$, V_T using a second independent pseudorandom generator such that each V_i is of $vLen$ bits and T is the maximal number of requests between re-seeding as defined in [22]. The generated pseudorandom sequence is $G(V_0)\cdots G(V_T)$ where G is a hash function or block cipher primitive. For the first choice of $vLen$, we set $vLen = seedlen$. The second choice is for hash function based generators only and we set $vLen = seedlen + u$ where u is the hash function G's message block size.

The values of $V_0, V_1, \cdots, V_T$ are generated from the seeding information using a second independent pseudorandom generator such as another block cipher or another hash function based generator or linear feedback shift registers (LFSR). For the first choice of $vLen$, we achieve the same efficiency of [22] by having one message block for each primitive function call. The advantage of the second choice for $vLen$ is that if G is a hash function, then the hash function internal states (or the first hash values) are dynamically changed for each hash function call and we expect this will produce better randomness properties within the generated sequences. Our experiments show that sequences generated with the above proposed approach have better fluctuation scales for the value S_{lil} compared with the results in Sections VII-B and VII-C.

Figure 10 shows the LIL test results on 6 typical sequences based on SHA1 hash function with the proposed seeding approach and dynamic inputs using the scale $10000n^2$ for the x-axis. Each sequence is 1.34GB long (2^{26} calls to the hash function). In the following, we use G_1 to denote the SHA1 hash function. The LFSRs that we used are based on the feedback polynomial $x^{32}+x^{22}+x^2+x+1$. Compared

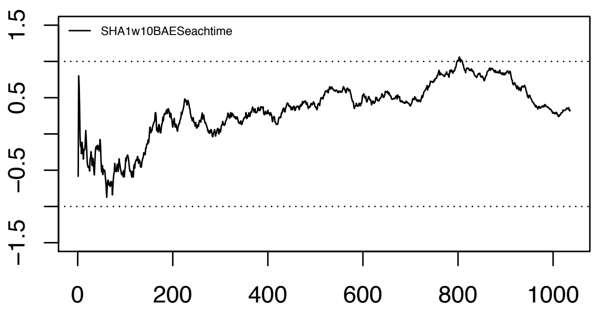
Fig. 10. Result for SHA1 with dynamic inputs



to results in Figure 8, Figure 10 have much better S_{lil} fluctuation scales as we have expected and these sequences have passed the weak (α, n_1, n_2)-LIL test with $\alpha = 0.01$, $n_1 = $ 6MB, and $n_2 = $ 1.34GB. The line sha1w47BR8BRctr is for the sequence $G_1(x_0,0)G_1(x_1,1)\cdots$ where x_i is a 47-byte string generated from the seed s using AES128 counter mode and the counter is 8 bytes. The line sha1w50BR8BRctr is for the sequence $G_1(x_0)G_1(x_1)\cdots$ where x_i is a 58-byte string generated from 15 LFSRs with different initial values based on the seed s. The line sha1w51BS8Bctr is for the sequence $G_1(x,y_0)G_1(x,y_1)\cdots$ where x is a 51-byte fixed seed and y_i is a 8-byte counter generated from two LFSRs with different initial values based on the seed s. The line sha1w60BR4Bctr is for $G_1(x_0,y_0)G_1(x_1,y_1)\cdots$, where x_i is a 60-byte string generated using AES128 in counter mode and y_i is a 8-byte random string that is generated using two LFSRs with different initial values based on the seed s. The line sha1w60BR4Bctr4BLFSR is for $G_1(x_0,0,y_0)G_1(x_1,1,y_1)\cdots$, where x_i is a 60-byte string generated using AES128 in counter mode, y_i is a 4-byte string generated using one LFSR with seed s, and the counter is

4 bytes. The line sha1w20BRS4BLFSTctrUSha1 is for the sequence $G_1(x_0, y_0)G_1(x_1, y_1)\cdots$, where x_i is a 20-byte string generated using SHA1 with counter mode and y_i is a 4-byte dynamic counter generated using one LFSR with the seed s.

Fig. 11. Result for SHA1 with random counter mode



As another example, Figure 11 shows the test result on a sequence $G_1(x_0)G_1(x_1)\cdots$ where x_i is a 10-byte sequence generated by AES128 with counter mode and with key $k =$ `0x0E14533E 1F056F7C7E192B3F4C4D7E6F`. It is clear that the LIL curve in Figure 11 has very good S_{lil} fluctuation scale and passes the weak (α, n_1, n_2)-LIL test with $\alpha = 0.01$, $n_1 =$ 6MB, and $n_2 = 1.34$GB. This example further justifies our proposed seeding approach since we used a second AES128 based pseudorandom generator to produce the input to the SHA1 based pseudorandom generator.

References

[1] E. Barker and J. Kelsey. *NIST SP 800-90A: Recommendation for Random Number Generation Using Deterministic Random Bit Generators*. NIST, 2012.

[2] M. Bellare and P. Rogaway. Random oracles are practical: a paradigms for designing efficient protocols. In *Proc. ACM CCS*, pages 62–73, 1993.

[3] M. Bellare and P. Rogaway. Optimal asymmetric encryption. In *Advances in Cryptology–EUROCRYPT'94*, pages 92–111. Springer, 1995.

[4] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. Sponge-based pseudo-random number generators. *CHES 2010*, pages 33–47, 2010.

[5] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. The keccak reference. *NIST winning algorithm of SHA3*, 2012.

[6] M. Blum and S. Micali. How to generate cryptographically strong sequences of pseudorandom bits. *SIAM J. Comput.*, 13:850–864, 1984.

[7] D. Boneh. Simplified oaep for the rsa and rabin functions. In *Advances in Cryptology–CRYPTO 2001*, pages 275–291. Springer, 2001.

[8] R.L. Brown, J. Durbin, and J.M. Evans. Techniques for testing the constancy of regression relationships over time. *J. Royal Stat. Soc. Ser. B (Methodological)*, pages 149–192, 1975.

[9] R. Canetti, O. Goldreich, and S. Halevi. The random oracle methodology, revisited. *J. Assoc. Comput. Math.*, 51(4):557–594, 2004.

[10] W. Feller. *Introduction to probability theory and its applicatons*, volume I. John Wiley & Sons, Inc., New York, 1968.

[11] Niels Ferguson and Bruce Schneier. *Practical cryptography*, volume 141. Wiley New York, 2003.

[12] I. Goldberg and D. Wagner. Randomness and the netscape browser. *Dr Dobb's Journal-Software Tools for the Professional Programmer*, 21(1):66–71, 1996.

[13] S. Goldwasser and S. Micali. Probabilistic encryption. *J. Comput. Sys. Sci.*, 28(2):270–299, 1984.

[14] Oracle Inc. Class SecureRandom. http://docs.oracle.com/javase/1.4.2/docs/api/java/security/SecureRandom.html, 2004.

[15] Oracle Inc. Java$^{\mathrm{tm}}$ cryptography architecture – API specification & reference. http://docs.oracle.com/javase/1.5.0/docs/guide/security/CryptoSpec.html, 2004.

[16] A. Khintchine. Über einen satz der wahrscheinlichkeitsrechnung. *Fund. Math*, 6:9–20, 1924.

[17] J. H. Lutz. Almost everywhere high nonuniform complexity. *J. Comput. System Sci.*, 44:220–258, 1992.

[18] P. Martin-Löf. The definition of random sequences. *Inform. and Control*, 9:602–619, 1966.

[19] U. Maurer, R. Renner, and C. Holenstein. Indifferentiability, impossibility results on reductions, and applications to the random oracle methodology. *Theory of Cryptography*, pages 21–39, 2004.

[20] NIST. Statistical test suite, http://csrc.nist.gov/groups/ST/toolkit/rng/, 2010.

[21] Federal Information Processing Standards Publication. Fips pub 180-4, secure hash standard (SHS). *US Department of Commerce*, 2011.

[22] A. Rukhin, J. Soto, J. Nechvatal, M. Smid, E. Barker, S. Leigh, M. Levenson, M. Vangel, D. Banks, A. Heckert, J. Dray, and S. Vo. *A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications*. NIST SP 800-22, 2010.

[23] C. P. Schnorr. *Zufälligkeit und Wahrscheinlichkeit. Eine algorithmische Begründung der Wahrscheinlichkeittheorie*. Lecture Notes in Math. 218. Springer Verlag, 1971.

[24] V. Shoup. Oaep reconsidered. In *CRYPTO 2001*, pages 239–259. Springer, 2001.

[25] J. Ville. *Étude Critique de la Notion de Collectif.* Gauthiers-Villars, Paris, 1939.

[26] R. von Mises. Grundlagen der wahrscheinlichkeitsrechung. *Math. Z.*, 5:52–89, 1919.

[27] Y. Wang. Resource bounded randomness and computational complexity. *Theoret. Comput. Sci.*, 237:33–55, 2000.

[28] A. C. Yao. Theory and applications of trapdoor functions. In *Proc. 23rd IEEE FOCS*, pages 80–91, 1982.