Proceedings of the 1989 Winter Simulation Conference
E.A. MacNair, K.J. Musselman, P. Heidelberger (eds.)

ABOUT POLYNOMIAL-TIME "UNPREDICTABLE" GENERATORS

Pierre L'Ecuyer and René Proulx
Département d'informatique
Université Laval
Ste-Foy, Qué., Canada, G1K 7P4.

ABSTRACT

So-called "perfect" or "unpredictable" pseudorandom generators have been proposed recently by people from the area of cryptology. Many people got aware of them from an optimistic article in the New York Times (Gleick (1988)). These generators are usually based on nonlinear recurrences modulo some integer m. Under some (yet unproven) complexity assumptions, it has been proven that no polynomial-time statistical test can distinguish a sequence of bits produced by such a generator from a sequence of truly random bits.

In this paper, we give some theoretical background concerning this class of generators and we look at the practicality of using them for simulation applications. We examine in particular their ease of implementation, their efficiency, periodicity, the ease of jumping ahead in the sequence, the minimum size of modulus that should be used, etc.

1. INTRODUCTION

In the recent years, a growing interest has raised for "cryptographically strong" (or "perfect", or "unpredictable") pseudorandom generators. The introduction of such generators is an outgrowth of a body of results and ideas which, as stated by Yao (1982), form a *theory of information* based on *computational complexity*. They are of primary interest in the field of cryptology (public key cryptosystems, generation of random pads, message authentication, key exchange schemes, etc.). See Brassard (1988). They are not well known to the simulation community. The purpose of this paper is to introduce them and look at the practicality of their use for simulation applications.

Pseudorandom generators are in fact deterministic algorithms that stretch a short (truly) random seed into a long sequence that is supposed to appear and behave like a true random sequence. The most commonly used in practice are still the linear congruential generators and its variants (matrix form, combinations, etc.) (Knuth (1981), Bratley et al. (1987), L'Ecuyer (1988, 1989)). These generators are quite efficient and show good statistical behavior when their parameters are well chosen. However, efficient algorithms have been designed to infer sequences produced by linear congruential generators, even when the multiplier, increment and modulus are unknown, by looking at the first few numbers generated (Plumstead (1982)). Discarding the low order bits of the numbers in the output still fails to assure unpredictability (Boyar (1989a)), Stern (1987), Frieze et al. (1984, 1988)). Finally, general results for multiple linear congruential recurrences and for one-term polynomial congruential recurrences of an arbitrary degree with unknown coefficients have been established which provide efficient inference methods for predicting the sequences of numbers produced (Boyar (1989b), Lagarias and Reeds (1988)). Though the distributions generated by these commonly used pseudorandom generators may satisfy various statistical tests of randomness these results indicate that the sequences obtained are not unpredictable as one would expect a random sequence to be.

The generators that we examine in sections 3 and 4 of this paper are based on nonlinear polynomial (monomial) one-term recurrences, but their output consists in a fraction of the low order bits of the successive terms in the recurrence. Hence these generators produce sequences of (pseudorandom) bits, which in turn can be blocked for the generation of integers or rationals. Under some unproven but reasonable complexity assumptions (e.g. no algorithm can systematically factorize integers in time polynomial in the size of the smallest factor), these generators are provably "unpredictable" in the sense that from a given sequence of output bits, no polynomial-time algorithm can guess the next bit (or previous bit) significantly better than by flipping a fair coin. Also, no polynomial-time statistical test can distinguish the output (sequence of bits produced) from a truly random sequence of bits. More precise definitions of these properties are given in the next section.

Section 2 introduces basic concepts, definitions and properties. Theorems 1 and 3 are related to theorems stated by Yao (1982), Goldreich et al. (1986) and Levin (1987). Theorem 4 is a modification of theorem 1 in Blum and Micali (1984). Our definitions and proofs are strongly inspired from these authors, but with some modifications and extensions. In sections 3 and 4, we examine the BBS generator as proposed by Blum, Blum and Schub (1986) and improved by Vazirani and Vazirani (1984), and the SPG generator, suggested by Micali and Schnorr (1988). Practical aspects of implementation of these generators are discussed. The last section concludes by commenting on the practicality of these generators for simulation applications. Our conclusion is far from being as optimistic as Gleick (1988).

2. PT-PERFECT GENERATORS

We assume the "Turing machine" model of computation. *Probabilistic* algorithms are allowed to toss a fair coin, once per step, to produce truly random bits. As in Brassard and Bratley (1987), for any $f : \mathbb{N} \rightarrow [0, \infty)$, we define $O(f(n))$ as the *set* of functions $g : \mathbb{N} \rightarrow [0, \infty)$ such that for some constants $c > 0$ and $n_0 \in \mathbb{N}$, $g(n) \leq cf(n)$ for all $n \geq n_0$.

A *polynomial-time* (PT) algorithm is one that halts in (worst-case) time $g(n) \in O(f(n))$, where $f(\cdot)$ is a polynomial and n is the size of the input. As usual (in many complexity studies), PT algorithms are considered *efficient*, while any problem for which no PT algorithm exists is considered *hard*. Of course, this should be interpreted with care, since in practice, it is possible (although rare) to have a PT algorithm A and a non-PT algorithm B such that A runs *slower* than B for all problem instances that can be solved in reasonable time on a fast computer.

2.1. Polynomial-time generators

DEFINITION 1. A *Pseudo-Random Number Generator* (PRNG) is a sequence of structures $G = \{G_n = (\ell_n, S_n, \mu_n, f_n, U_n, g_n),\ n \geq 1\}$, where $\ell_n = \ell(n) \in O(n)$ and for each n, G_n is defined as follows. S_n is a finite set of bit strings of uniform length ℓ_n (i.e. each $s \in S_n$ can be represented by a string of at most ℓ_n bits). S_n is called the *state* space and ℓ_n the *size* of the generator. μ_n is a probability distribution on $(S_n, \mathcal{B}(S_n))$, called the *initial distribution*, where $\mathcal{B}(S_n)$ is the set of all subsets of S_n. U_n is a finite set called the *output* space (or *observation* space), $f_n : S_n \rightarrow S_n$ is the *transition function* and $g_n : S_n \rightarrow U_n$ is the *output function* (or *observation function*). We will denote by c_n the cardinality of the set U_n and let $\nu_n = \lceil \lg c_n \rceil$ (where lg is the log in base 2), the minimal number of bits to represent all the output symbols. We assume that $\nu_n = \nu(n) \in O(n)$. For a fixed n, the generator operates as follows:

(1) Select the initial state $s_0 \in S_n$ according to μ_n (or an approximation of μ_n, see below); let $u_0 := g_n(s_0)$;

(2) for $i := 1, 2, \ldots$, let $s_i := f_n(s_{i-1})$ and $u_i := g_n(s_i)$.

The sequence of *observations* $(u_0, u_1, u_2, \ldots)$ is the (observable) output of the generator. The initial state s_0 is called the *seed*. Often, in practice, s_0 cannot be generated exactly according to μ_n, but one can use an approximating distribution $\tilde{\mu}_n$.

DEFINITION 2. Let $\{(S_n, \mu_n),\ n \geq 1\}$ be a sequence such that each S_n is a finite set and μ_n is a probability distribution on the subsets of S_n (as above). We say that μ_n is *polynomially accessible* if there exists a constant v and a probabilistic polynomial time (in n) algorithm, using $O(n^v)$ true random bits, whose output follows (for each n) a distribution $\tilde{\mu}_n$ on S_n such that for each constant t,

$$\sum_{s \in S_n} |\mu_n(s) - \tilde{\mu}_n(s)| \in O(n^{-t}). \qquad (1)$$

DEFINITION 3. A *polynomial-time PRNG* (PT-PRNG) is a PRNG for which there are deterministic (worst-case) polynomial-time (in n) procedures to compute f_n and g_n, and which has a polynomially accessible initial distribution μ_n.

We will use slight abuses of notation. each time we say that something is drawn according to μ_n, we mean in fact that in practice, it is drawn according to some $\tilde{\mu}_n$ that satisfies definition 2. Once s_0 is fixed, all the rest can be computed deterministically and "efficiently" (provided s_0, f_n and g_n are known). In particular, the distribution μ_n for s_0 induces a probability distribution over the set of sequences of observations $(u_0, u_1, \ldots)$. We will also denote it by μ_n. Other authors set $v = 1$ in definition 2, but most currently conjectured "unpredictable" generators need more than $O(n)$ truly random bits. Of course, all this is interesting only if the generator can output more bits than the number of true random bits required to get its seed.

2.2. Statistical tests and unpredictability

DEFINITION 4. A *polynomial-time statistical test* (PT-ST), for a PT-PRNG $G = \{G_n,\ n \geq 1\}$, is a sequence of probabilistic algorithms $T = \{T_n,\ n \geq 1\}$ and two constants v and w such that T_n takes as input a sequence of observations of length n^w, say $(u_0, u_1, \ldots, u_{n^w-1})$, halts in (worst-case) time at most n^v and outputs 0 or 1. Let $p(G_n, T_n)$ be the probability that T_n outputs 1 when the sequence $(u_0, \ldots, u_{n^w-1})$ is drawn from μ_n. Let $p^*(G_n, T_n)$ be the probability that T_n outputs 1 if all the observations u_i's are replaced by i.i.d. random variates truly uniformly distributed over U_n. We assume that such uniform variates can be drawn in polynomial time. We say that the PT-PRNG *passes the test* if for every constant t,

$$|p(G_n, T_n) - p^*(G_n, T_n)| \in O(n^{-t}). \qquad (2)$$

DEFINITION 5. A PT-PRNG is called *PT-perfect* (a PT-perfect PRNG, or PT-perfect generator) if it passes every possible PT-ST.

DEFINITION 6. A *PT-prediction test* for a PT-PRNG is a PT-ST, such that T_n uses as input $\tilde{U}_n \subset \{u_0, u_1, \ldots, u_{n^w-1}\}$ (a strict subsequence of the output), selects k such that $0 \leq k < n^w$ and $u_k \notin \tilde{U}_n$, and computes a value $\tilde{u}_k \in U_n$, which is a *prediction* of u_k given $\tilde{U}_n$. It then observes u_k and if $u_k = \tilde{u}_k$ (correct prediction), it outputs 1, otherwise it outputs 0. In the case where $\tilde{U}_n = \{u_i, \ldots, u_j\}$ for some $0 \leq i \leq j < n^w$ and $k = j+1 < n^w$ [$k = i-1 \geq 0$], it is called a *PT-next observation* [*PT-previous observation*] test.

REMARK 1. In the case where the u_i's are truly i.i.d. uniform over U_n, the probability $p^*(G_n, T_n)$ of a correct prediction is $1/c_n$. Therefore, the generator passes the prediction test iff for every constant t, $|\Pr(\text{correct prediction}) - 1/c_n| \in O(n^{-t})$.

DEFINITION 7. A PT-PRNG is called *PT-unpredictable* [*PT-unpredictable to the right, PT-unpredictable to the left*] if it passes every possible PT-prediction [PT-next observation, PT-previous observation] test.

THEOREM 1. For a PT-PRNG G, assume $c_n \in O(n^v)$ for some constant v. Then, the following are equivalent:

(a) G is PT-perfect;

(b) G is PT-unpredictable;

(c) G is PT-unpredictable to the right;

(d) G is PT-unpredictable to the left.

PROOF. Obviously, (a) implies (b), and (b) implies (c) and (d). Hence, it suffices to prove that (c) or (d) implies (a). Suppose that some PT-ST $\{T_n,\ n \geq 1\}$ (with constants v, w) is not passed. For some constant t, and all n in some infinite subset $\tilde{N}$ of $\mathbb{N} = \{1, 2, 3, \ldots\}$, $|p(G_n, T_n) - p^*(G_n, T_n)| > n^{-t}$. We will construct a PT-next observation test that is not passed, showing that (c) cannot be true. For each i, $0 \leq i \leq n^w$, consider the procedure A_i obtained by feeding the PT-ST as follows. Draw a sequence of observations of length i, say $(\tilde{u}_0, \ldots, \tilde{u}_{i-1})$, according to μ_n. For $k \geq i$, draw $\tilde{u}_k$ uniformly from U_n. Give $(\tilde{u}_0, \ldots, \tilde{u}_{n^w-1})$ as input to T_n. Output the same value as T_n. Let p_i be the probability that this procedure A_i outputs 1. Note that $p_0 = p^*(G_n, T_n)$ and $p_{n^w} = p(G_n, T_n)$. Let $n \in \tilde{N}$. Without loss of generality, we can assume that $p(G_n, T_n) > p^*(G_n, T_n)$ (otherwise, inverse the result of the PT-ST). There is an integer k, $0 \leq k < n^w$, such that $p_{k+1} - p_k > n^{-(t+w)}$. Define the following PT-next observation test $\tilde{T}_n$ for u_k. Apply procedure A_k and if it outputs 1, take $\tilde{u}_k$ as a guess, otherwise draw $\tilde{\tilde{u}}_k$ uniformly from U_n and take its value as a guess. If the guess is correct (equals to u_k), $\tilde{T}_n$ outputs 1, otherwise it outputs 0. The probability that the guess is correct is Pr(guess correct) $= \Pr(A_k \text{ outputs 0 and } \tilde{\tilde{u}}_k = u_k) + \Pr(A_k \text{ outputs 1 and } \tilde{u}_k = u_k) = (1 - p_k)/c_n + \Pr(A_k \text{ outputs } 1 \mid \tilde{u}_k = u_k)/c_n = (1 + p_{k+1} - p_k)/c_n \geq (1 + n^{-(t+w)})/c_n$. Therefore, this PT-next observation test is not passed, so that (c) implies (a). The proof that (d) implies (a) is similar. This time, A_i uses $(\tilde{u}_0, \ldots, \tilde{u}_{n^w-1})$ where $\tilde{u}_0, \ldots, \tilde{u}_i$ are random, uniformly drawn from U_n, and the sequence $(\tilde{u}_{i+1}, \ldots, \tilde{u}_{n^w-1})$ follows μ_n. Assume that $p(G_n, T_n) < p^*(G_n, T_n)$ and choose k such that $-(p_k - p_{k-1}) > n^{-(t+w)}$. The PT-previous observation test is defined in the same way. ∎

THEOREM 2. A necessary condition for a PT-PRNG to be PT-perfect is that the distribution of each u_i should be approximately uniform over U_n, with polynomially negligible error. More specifically, for any $u \in U_n$, integer $i \geq 0$ and constant t,

$$|\mu_n(u_i = u) - 1/c_n| \in O(n^{-t}).$$

PROOF. Otherwise, the generator does not pass the PT-prediction test that just guesses $u_i = u$ without looking at anything else. (Note that this is not ruling out the possibility that $\mu_n(u_i = u) = 0$ for some u, since c_n may for instance increase exponentially with n.) ∎

2.3. Bit tests

We now examine a little further the case where the output is a sequence of bits, i.e. $U_n = \{0,1\}$ for each n. In this case, the PT-prediction, PT-next observation and PT-previous observation tests are called respectively *PT-bit prediction*, *PT-next bit* and *PT-previous bit* tests. The terms PT-unpredictable, PT-unpredictable to the right and PT-unpredictable to the left can also be replaced respectively by PT-bit, PT-previous bit and PT-next bit unpredictable.

Obviously, any PT-PRNG can generate a sequence of bits: just concatenate the u_i's, viewed as bit strings of length ν_n. But nothing guarantees that these bits will look random, even if the initial generator is PT-perfect. For instance, if $c_n = 3$ (and $\nu_n = 2$), only about one-third of the bits will be ones. We will see that if U_n satisfies a simple property (below), a generator is PT-perfect if and only if its associated bit generator is PT-perfect.

DEFINITION 8. A PT-PRNG is called *bit-complete* if for each n, U_n is the set of *all* bit strings of length ν_n (and $c_n = 2^{\nu_n}$).

DEFINITION 9. Let $G = \{G_n,\ n \geq 1\}$ be a bit-complete PT-PRNG. The *bit-generator associated to G* is the PT-PRNG $\tilde{G} = \{\tilde{G}_n = (\tilde{\ell}_n, \tilde{S}_n, \tilde{\mu}_n, \tilde{f}_n, \tilde{U}_n, \tilde{g}_n),\ n \geq 1\}$, where $\tilde{\ell}_n = \ell_n + \lceil \lg \nu_n \rceil$, $\tilde{S}_n = S_n \times \{1, \ldots, \nu_n\}$, $\tilde{U}_n = \{0,1\}$ (so that $\tilde{c}_n = 2$ and $\tilde{\nu}_n = 1$) and for each $(s,j) \in \tilde{S}_n$,

$$\tilde{f}_n(s,j) = \begin{cases} (f_n(s), 1) & \text{if } j = \nu_n; \\ (s, j+1) & \text{if } 1 \leq j < \nu_n; \end{cases}$$

$$\tilde{g}_n(s,j) = j\text{-th bit of } g_n(s);$$

$$\tilde{\mu}_n(s,j) = \begin{cases} \mu_n(s) & \text{if } j = 1; \\ 0 & \text{otherwise.} \end{cases}$$

THEOREM 3. Let $G = \{G_n,\ n \geq 1\}$ be a bit-complete PT-PRNG, with associated bit-generator $\tilde{G} = \{\tilde{G}_n,\ n \geq 1\}$. Then the following are equivalent:

(a) G is PT-perfect;

(b) $\tilde{G}$ is PT-perfect;

(c) $\tilde{G}$ is PT-bit unpredictable;

(d) $\tilde{G}$ is PT-next bit unpredictable;

(e) $\tilde{G}$ is PT-previous bit unpredictable.

PROOF. Any PT-ST T for G can be transformed into a PT-ST $\tilde{T}$ for $\tilde{G}$ and vice-versa. Bits generated from $\tilde{G}$ can be regrouped into packets and given to T, while the observations obtained from G can be cut down into bits (as above) and given to $\tilde{T}$. Because the generator is bit-complete, each u_i is truly uniformly distributed in S_n if and only if each bit of u_i is truly uniformly distributed in $\{0,1\}$. Therefore, G passes the test T if and only if $\tilde{G}$ passes the test $\tilde{T}$. This proves the equivalence between (a) and (b), and the remainder follows from Theorem 1. ∎

The above Theorem is related to Theorem 1 in Goldreich et al. (1986). Note that we don't say that a generator *must* be bit-complete for its associated bit generator to be PT-perfect. Even if it is bit-complete, we don't say either that all elements of U_n must have positive probability to appear somewhere in the output sequence. For instance, if there exists a PT-perfect PRNG with ν_n linear in n (c_n exponential in n), having a small number of elements of U_n with zero probability should be acceptable.

The next result concerns the periodicity of PT-perfect generators. Since S_n is finite, the output sequence will eventually become periodic. For a given $s_0 \in S_n$, the *period* is

$$\begin{aligned} \rho(s_0) = \min\{j - i + 1 \mid j > i \geq 0, \\ u_{i+k} = u_{j+k} \text{ for all } k \geq 0\} \end{aligned}$$

and the *transient* is

$$\tau(s_0) = \min\{i \geq 0 \mid u_{i+k} = u_{i+k+\rho(s_0)} \text{ for all } k \geq 0\}.$$

THEOREM 4. Let w be a constant and G a PT-perfect PT-PRNG. Let $S'_n = \{s_0 \in S_n \mid \rho(s_0) + \tau(s_0) \leq n^w\}$. This the set of initial states such that the sequence of observations becomes periodic after a transient plus initial cycle of total length at most n^w. Then, for any constant t, $\mu_n(S'_n) \in O(n^{-t})$.

PROOF. Whenever $s_0 \in S'_n$, from $u_0, \ldots, u_{n^w}$, one can compute u_{n^w+1} in polynomial time (in n). If $\mu_n(S'_n) \geq n^{-t}$ for an infinite number of values of n, one can design a PT-next observation test for u_{n^w+1}, which computes u_{n^w+1} if $s_0 \in S'_n$ and guesses at random otherwise. The probability of a correct prediction is $\Pr(\text{correct}) = \mu_n(S'_n) + (1 - \mu_n(S'_n))/c_n \geq 1/c_n + n^{-t}(1 - 1/c_n) \geq 1/c_n + n^{-t}/2$ (since $c_n \geq 2$). Therefore, G fails the test and cannot be PT-perfect. ∎

2.4. One-way and random functions

The notion of *one-way function* (Yao (1982)) is strongly related to the notion of PT-perfect PRNG. Roughly speaking, a *one-way function* h is such that $h(x) = h_n(x)$ for $x \in \{0,1\}^n$, where $h_n : \{0,1\}^n \rightarrow \{0,1\}^n$ can be computed in PT (in n), but for which any PT algorithm trying to recover x from $h(x)$ will fail for at least a constant fraction (in n and i) of each set $D_n^i = h_n^i(\{0,1\}^n)$ (where h_n^i is the i-fold composition of h_n). More precise definitions are given in Levin (1987) and Goldreich et al. (1986). Yao (1982) studied the notion of one-way function and showed how, from any one-way *permutation*, one can construct a PT-perfect PRNG. Levin (1987) went further by proving that a one-way function exists *if and only if* a PT-perfect PRNG exists. (His definitions of PT-ST and one-way function were slightly different from Yao's).

Another related notion is the one of *random function* (Goldreich et al. (1986)). Informally, a *PT-random function* is like a PT-PRNG, except that the seed s_0 determines a function $h : \{0,1\}^n \rightarrow \{0,1\}^n$ that can be specified and evaluated in PT, but cannot be distinguished in PT from a function randomly selected from the set of all 2^{k2^k} possible functions of that form. Goldreich et al. (1986) show how, from any PT-perfect bit PRNG (and from any one-way function), one can construct a PT-random function. Luby and Rackoff (1988) also show that from any PT-random function, one can construct a PT-perfect *permutation* generator. Micali and Schnorr (1988) use the construction of Goldreich et al. (1986) to transform every PT-perfect generator into one that can be accelerated efficiently by parallel evaluation.

2.5. Some presumed PT-perfect generators

Various generators proposed recently have been *proved* to be PT-perfect, under some yet *unproven* complexity assumption. See for instance Yao (1982), Blum and Micali (1984), Blum et al. (1986), Alexi et al. (1988), Reif and Tygar (1988), Micali and Schnorr (1988). All of these are in fact based on *presumed* one-way functions. In the next sections, we examine in more detail two of these generators.

3. THE BBS GENERATOR

3.1. Definition

Blum, Blum and Schub (1986) have proposed the following generator. Let $N = pq$ be a n-bit *Blum integer* (i.e. a product of two distinct primes p and q, both congruent to 3 modulo 4), such that p and q have approximately the same size. Let y be a positive integer relatively prime to N and let $x_0 = y^2 \bmod N$. For $i = 1, 2, \ldots,$ let

$$x_i := x_{i-1}^2 \bmod N. \tag{3}$$

At each step, the generator outputs the last ν_n bits of x_i. In the original version (Blum et al. (1986)), one had $\nu_n = 1$ (the output was the parity of x_i) and the generator was based on the quadratic residuosity assumption. But Vazirani and Vazirani (1984) have shown that under the FISH assumption below, taking $\nu_n = \nu(n) \in O(\lg n)$ yields a PT-perfect generator.

ASSUMPTION FISH (Factoring is hard). Any PT algorithm whose purpose is to factorize n-bit Blum integers will fail on at least a constant fraction (in n) of the possible inputs. In other words, the multiplication function defined by $h_n(p,q) = N$ is one-way.

We can define a BBS generator G as follows. For simplicity, we assume that n is even. Let $\ell_n = n$, $D_n = \{N \mid N$ is a Blum integer with two $(n/2)$-bit factors $\}$ and $S_n = \{s = (N,x) \mid N \in D_n$ and x is a quadratic residue modulo N, relatively prime to $N\}$. (Some members of D_n can have in fact $n-1$ bits, but this is not a problem.) Let P_n be the set of all $(n/2)$-bit primes that are congruent to 3 mod 4. μ_n is defined as follows. Pick p and q randomly (uniformly) from P_n. Let $N := pq \in D_n$. Then pick y uniformly among the integers in $\{2, \ldots, N-1\}$ that are relatively prime to N, and let $x := y^2 \bmod N$. Define f_n by $f_n(N,x) := (N, x^2 \bmod N)$. Let $\nu_n = \lfloor \gamma \lg n \rfloor$ for some constant $\gamma > 0$, $U_n = \{0,1\}^{\nu_n}$ and $g_n(N,x) := x \bmod 2^{\nu_n}$. Under the FISH assumption, this BBS generator is PT-perfect.

The best currently available factorization programs can routinely find prime factors of up to about 50 decimal digits (150 to 200 bits). Current progress is also quite fast in that area. Therefore, in order to rely on the FISH asssumption, n must be *much* larger than say 400. Perhaps $n = 500$ is still acceptable for today, while $n = 1000$ would be safer for a few years to come.

3.2. Period length

Blum et al. (1986) have shown that the sequence $\{s_i = (N, x_i),\ i \geq 0\}$ is *purely* periodic with period p that divides $\lambda(\lambda(N))$. That is $x_0^{2^p} \bmod N = x_0$. Here λ is the Carmichael's function defined by $\lambda(1) = \lambda(2) = 1$; $\lambda(4) = 2$; $\lambda(2^e) = 2^{e-2}$ for $e > 2$; $\lambda(p^e) = p^e - p^{e-1}$

for p odd prime and $e \geq 1$; $\lambda(p_1^{e_1} \cdots p_k^{e_k}) =$ l.c.m. $(\lambda(p_1^{e_1}), \ldots, \lambda(p_k^{e_k}))$ if $p_1, \ldots, p_k$ are prime. In particular, if $N = pq$ is a Blum integer, $\lambda(N) =$ l.c.m. $(p-1, q-1)$.

Blum et al. also prove that the following conditions are *sufficient* for the period to be *equal* to $\lambda(\lambda(N))$. A prime p is called a *special prime* if $p = 2p_1 + 1 = 2(2p_2 + 1) + 1$ where p_1 and p_2 are prime. Let $D'_n = \{N \in D_n \mid N = pq$, p and q are special primes and 2 is a quadratic residue with respect to (i.e. modulo) at most one (or none) of the two integers $p_1 = (p-1)/2$ and $q_1 = (q-1)/2\}$. Let $S'_n = \{(N, x) \in S_n \mid N \in D'_n$ and $k = \lambda(N)/2$ is the smallest positive integer for which $x^k \bmod N = 1\}$. If $(N, x_0) \in S'_n$, then the period of the x_i's is $\lambda(\lambda(N))$.

We can define a PRNG G' by modifying G as follows. Replace S_n by S'_n, P_n by the set P'_n of $(n/2)$-bit special primes, and μ_n by μ'_n defined as follows. Pick p uniformly from P'_n. Then pick values of q uniformly from P'_n until $N = pq \in D'_n$. Then pick values of y uniformly from $\{2, \ldots, N-1\}$ until $(N, x = y^2 \bmod N) \in S'_n$.

Following a remark by Blum et al., an element of S'_n can be found (on the average) after a polynomial number of trials (using a polynomial number of truly random bits). Indeed, the fraction of $(n/2)$-bit integers that are in P'_n is presumed to be asymptotically $((n/2)\ln 2)^{-3}$. For instance, for $(n/2) = 256$, this is about one in 15 millions. Therefore, for large n, picking numbers from D'_n could be very time consuming in practice. We did the following empirical investigation. For different values of n, we look at how much CPU time (in minutes) it took to find a *first* $(N, x) \in S'_n$ according to μ'_n. The results appear in table 1. We also indicate the period length associated with that seed (N, x) and the approximate fraction of $(n/2)$-bit integers that are special primes. Note that for $n > 128$, we were unable to find elements of S'_n in reasonable time. It took us 155 hours of CPU time to find two 128-bit special primes p and q, but for which 2 is a quadratic residue with respect to *both* $(p-1)/2$ and $(q-1)/2$. This experiment, like all others reported in this paper, were done on a MicroVax-II, using our own software (based on L'Ecuyer et al. (1988)). The "truly random bits" were replaced by the output of the "fast" generator proposed in L'Ecuyer (1988). To generate "random" primes, we followed the approach discussed in Beauchemin et al. (1988).

Since selecting from S_n is much faster than from S'_n, it could be interesting to analyze the periods obtained in the former case. Since the period divides $\lambda(\lambda(N))$, it can be computed easily (eventually by exhaustive

Table 1: Finding a first element of S'_n

n	CPU (min)	period	fraction s.p.
32	3.5	2.8×10^8	.000733
64	30.3	1.6×10^{18}	.000092
95	450.5	4.2×10^{27}	.000027
128	1610.9	2.5×10^{37}	.000011

search) *provided* that $\lambda(N)$ and $\lambda(\lambda(N))$ can be factorized. Now, factorization becomes the hurdle. Table 2 reports another empirical investigation. For different values of n, we generated a large number of seeds (N, x) according to μ_n. Table 2 gives the number of those for which $\lambda(N)$ and $\lambda(\lambda(N))$ were successfully factorized and the period was computed. It also gives the approximate minimum, maximum and average of these periods. For each "seed", the allocated CPU time for factorization was 15 min. for $n = 64$ and 30 min. for $n = 128$.

Table 2: Approximate period when selecting from S_n

n	nb. gen.	$\lambda(\lambda(N))$ *fact.*	min. period	max. period	aver. period
64	20	20	10^{11}	10^{16}	10^{15}
128	30	13	10^{25}	10^{33}	10^{32}

3.3. Efficiency and other properties

When p and q (and thus N and $\lambda(N)$) and the generator's state s_i are known, one can efficiently *jump ahead* in the sequence, or generate the sequence *backwards*. Indeed, since $x_i^{\lambda(N)} \bmod N = 1$,

$$x_{i+j} = x_i^{2^j} \bmod N = x_i^{2^j \bmod \lambda(N)} \bmod N.$$

These exponentiations are easy to compute (see e.g. Brassard and Bratley (1987)). Theorem 10b in Blum et al. (1986) shows how to compute x_{i-j} given N, x_i and $\lambda(N)$. They also give an efficient algorithm to factorize N when the period of the x_i's is known.

Recall that saying that G is PT-perfect means that for large enough n, it will pass any PT-ST *which uses as input only the u_i's*, were u_i represents the last ν_n bits of x_i. The generator may not pass a test that knows x_i or the factors of N. Blum et al. (1986) have shown that even if N is known by the test, the generator with

$\nu_n = 1$ remains PT-previous bit unpredictable. Therefore, from theorem 1 or 3, it is also PT-perfect. They also show that under the quadratic residuosity assumption, no algorithm knowing only N and x_i can guess efficiently x_{i-1}. N can be observed, but must remain random. It is part of the seed. On the other hand, we can redefine S_n by *fixing* N, but then, the generator will certainly not be PT-perfect. In fact, the output sequence may be really bad for some specific values of N. But the proportion of such values probably goes down to zero exponentially fast (or faster) with n.

An important consideration in selecting a generator is its ease of implementation and speed. Computing $x^2 \bmod N$ for large N using a standard computer requires a non-negligible programming effort and/or can be CPU-time consuming. For our implementations, we used the package described in L'Ecuyer et al. (1988). We implemented instances of BBS generators for different values of n and with $\gamma = 1$. The bits generated ($\nu_n = \lfloor \lg n \rfloor$ per iteration) were regrouped into packets of 23 and each packet was used to produce a floating-point number between 0 and 1. Table 3 gives the CPU time (seconds) per 1000 numbers generated, on a MicroVax-II. For comparison, the 32-bit generator proposed in L'Ecuyer (1988) needs approximately 0.2 seconds per 1000 numbers in the same environment. In our implementation, the product modulo N is performed in quadratic time in n and $\lfloor \lg n \rfloor$ bits are obtained each time. Therefore, the time required per floating-point number is in $O(n^2/\lg n)$. "Faster" multiplication algorithms permit in principle to reduce this asymptotic time-per-number to $O(n^{1+\epsilon}/\lg n)$ for any $\epsilon > 0$ (Brassard and Bratley (1987)), but this reduction is effective only for *huge* values of n. For n around 500 to 1000, using divide-and-conquer may yield a small improvement in practice.

Table 3: Time to generate 1000 $U(0,1)$ values for BBS

n	CPU time (sec)
32	15.1
64	24.2
95	38.9
128	48.5
256	130.0

4. THE SPG GENERATOR

Micali and Schnorr (1988) have proposed a Sequential Polynomial Generator (SPG) which they claim is almost as efficient as the usual linear congruential generator. They conjecture it to be PT-perfect. It is defined as follows.

Again, assume n even. Let $d \geq 2$ and $\gamma \geq 0$ be two constants, $\zeta_n = \lfloor 2n/d \rfloor + \gamma$, $\ell_n = n + \zeta_n$, $\nu_n = n - \zeta_n$ and $M(N) = \lfloor (N-1)/2^{\nu_n} \rfloor$. Let $D_n = \{N = pq \mid p \text{ and } q \text{ are two } (n/2)\text{-bit primes}\}$ and $S_n = \{s = (N, x) \mid N \in D_n \text{ and } 0 \leq x \leq M(N)\}$. μ_n is defined by picking p and q uniformly from the set of (n/2)-bit primes, setting $N = pq$, computing $M(N)$, then picking x uniformly from the set $\{2, \ldots, M(N)\}$. Define

$$f_n(N, x) := \lfloor (x^d \bmod N)/2^{\nu_n} \rfloor;$$

$$g_n(N, x) := (x^d \bmod N) \bmod 2^{\nu_n}$$

and $U_n = \{0,1\}^{\nu_n}$. Note that $M(N)$ represents the largest possible output from f_n. At each step, the generator computes the n-bit value $x^d \bmod N$, whose first ζ_n bits make up the next value of x (determines the next state), while the last $\nu_n = n - \zeta_n$ bits are the output.

Two special cases of this generator are conjectured by Micali and Schnorr to be PT-perfect. (They do not state their conjectures exactly as we do below, but the idea is essentially the same.) In the first case, d is assumed odd and D_n is replaced by its subset $D'_n = \{N \in D_n \mid (p-1)(q-1) \text{ is relatively prime to } d\}$. Call this generator G_1.

CONJECTURE SPG1. Generator G_1 is PT-perfect.

This conjecture is related to the security of the RSA encryption scheme. The next conjecture allows even d, which yields more efficient generators (e.g. if d is a power of two). It is justified by the work of Alexi et al. (1988). Call G_2 the SPG generator obtained when D_n is replaced by its subset $D'_n = \{N \in D_n \mid N \text{ is a Blum integer}\}$.

CONJECTURE SPG2. Generator G_2 is PT-perfect.

Micali and Schnorr suggest $n = 512$ and $d = 15$ or 16 for practical applications.

We did empirical investigations with SPG generators, with different values of d and n. Concerning the period, we have observed that in most cases, x_i eventually reaches the absorbing state 0 (in which case $\rho(s_0) = 1$). For some other cases (seeds), the generator is not reaching zero but gets into a very short cycle. This is reminiscent of the infamous "Middle-square" method (Knuth

(1981)). On the other hand, the average transient (μ_n-expectation of $\tau(N, x_0)$) is more important and increases rapidly with n. According to theorem 4, it should increase faster than any polynomial. Table 4 summarizes the results of our experiments. For different values of d and n we picked a number of different seeds, and computed the approximate minimum and average of the period and transient.

Table 4: Approx. transient plus period for the SPG

d	n	nb. seeds	aver. ρ	aver. $\tau+\rho$	min. $\tau+\rho$
8	32	200	2.6	11.2	2
16	64	200	3.3	9.6	3
12	64	200	6.5	22.5	3

We also performed similar speed tests as for the BBS generator.

Table 5: Time to generate 1000 $U(0,1)$ values for SPG

d	n	CPU (sec)
8	128	7.6
15	128	7.0
15	256	10.3
16	256	9.7
15	512	16.5

5. CONCLUSION

We gave a framework for studying PT-PRNGs and derived some properties of PT-perfect generators. We also examined some practical aspects for the BBS and SPG generators, two classes of generators that are conjectured to be PT-perfect. Both use a modulus N that some might view as a *parameter* of the generator. But since the PT-perfectness conjecture is justified only for *random* N, we should view N as part of the seed.

Generating a random seed requires generating random primes, which itself requires a number of random bits. That number is in $O(n^v)$ for some v, but usually not in $O(n)$. Some will argue that it just postpones the problem of random number generation and they are partly right. But there is effectively a gain if from that seed, one generates n^w bits for $w > v$.

In practice, a BBS or SPG generator may have a proportion of seeds for which the behavior of the output sequence is totally unacceptable. But this proportion should go down to zero with n faster than one over any polynomial in n. In practice, "acceptable" safety can usually be obtained by taking n large enough. But "large enough" can sometimes be *very very large*. Our empirical investigations suggest that for such "large enough" n, straightforward (software) implementations of the BBS and SPG generators are not competitive in terms of speed with other generators currently in use, which are acceptable for most simulation applications.

ACKNOWLEDGEMENTS

This work has been supported by NSERC-Canada grant # A5463 and FCAR-Quebec grant # EQ2831 to the first author. It was performed while the first author was visiting the Laboratoire d'Automatique de Nantes (U. A. CNRS no. 823), ENSM, Nantes, France.

REFERENCES

Alexi, W., Chor, B., Goldreich, O. and Schnorr, C. P. (1988). RSA and Rabin Functions: Certain Parts are as Hard as the Whole. *SIAM J. on Computing*, **17**, 2, 194–209.

Beauchemin, P., Brassard, G., Crépeau, C., Goutier, C. and Pomerance, C. (1988). The Generation of Random Numbers That Are Probably Prime. *J. of Cryptology*, **1**, 53–64.

Blum, L., Blum, M. and Schub, M. (1986). A Simple Unpredictable Pseudo-Random Number Generator. *SIAM J. Comput.*, **15**, 2, 364–383. Preliminary version published in *Proceedings of CRYPTO 82*, 61–78.

Blum, M. and Micali, S. (1984). How to Generate Cryptographically Strong Sequences of Pseudo-Random Bits. *SIAM J. on Computing*, **13**, 4, 850–864.

Boyar, J. (1989a). Missing Low Order Bits in a Linear Congruential Generator. To appear in *J. of Cryptology*.

Boyar, J. (1989b). Inferring Sequences Produced by Pseudo-Random Number Generators. *J. of the ACM*, **36**, 1, 129–141.

Brassard, G. (1988). *Modern Cryptology: A Tutorial*. Lectures Notes in Computer Science, Springer-Verlag, New York.

Brassard, G. and Bratley P. (1987). *Algorithmique, conception et analyse* (in french), Masson, Paris, and Les Presses de l'Université de Montréal. English version: *Algorithmics, Theory and Practice*, Prentice-Hall, 1988.

Bratley, P., Fox, B. L. and Schrage, L. E. (1987). *A Guide to Simulation*, second edition. Springer-Verlag, New York.

Frieze, A. M., Kannan, R. and Lagarias, J. C. (1984). Linear Congruential Generators Do Not Produce Random Sequences. *Proceedings of the 25th IEEE Symposium on Foundations of Computer Science*, 480–484.

Frieze, A. M., Hastad, J., Kannan, R. and Lagarias, J. C. and Shamir, A. (1988). Reconstructing Truncated Integer Variables Satisfying Linear Congruences. *SIAM J. on Computing*, **17**, 2, 262–280.

Gleick, J. (1988). The Quest for True Randomness Finally Appears Successful. *The New York Times*, Thuesday, April 19, 1988, C1 and C8.

Goldreich, O., Goldwasser, S. and Micali, S. (1986). How to Construct Random Functions. *J. of the ACM*, **33**, 4, 792–807.

Knuth, D. E. (1981). *The Art of Computer Programming : Seminumerical Algorithms*, vol. 2, second edition. Addison-Wesley.

Lagarias, J. C. and Reeds, J. (1988). Unique Extrapolation of Polynomial Recurrences. *SIAM J. on Computing*, **17**, 2, 342–362.

L'Ecuyer, P. (1988). Efficient and Portable Combined Random Number Generators. *Communications of the ACM*, **31**, 6, 742–749 and 774.

L'Ecuyer, P. (1989). A Tutorial on Uniform Variate Generation. In these *Proceedings*.

L'Ecuyer, P., Perron, G. and Blouin, F. (1988). SENTIERS: Un logiciel Modula-2 pour l'arithmétique sur les grands entiers. Report no. DIUL-RT-8802, dépt. d'informatique, Univ. Laval.

Levin, L. A. (1987). One-Way Functions and Pseudorandom Generators. *Combinatorica*, **7**, 4, 357–363.

Luby, M. and Rackoff, C. (1988). How to Construct Pseudorandom Permutations From Pseudorandom Functions. *SIAM J. Computing*, **17**, 2, 373–386.

Micali, S. and Schnorr, C. P. (1988). Super-Efficient, Perfect Random Number Generators. Manuscript, Laboratory for Computer Science, M.I.T.

Morain, F. (1988). Implementation of the Atkin-Goldwasser-Kilian Primality Testing Algorithm. Rapport de recherche no. 911, INRIA, Rocquencourt, France.

Plumstead, J. B. (Boyar) (1982). Inferring a Sequence Generated by a Linear Congruence. *Proceedings of the 23rd IEEE Symposium on Foundations of Computer Science*, 153–159.

Rabin, M. O. (1979). Digitalized Signature and Public-Key Functions as Intractable as Factorization. Report no. MIT/LCS/TR-212, Laboratory for Computer Science, M.I.T., Cambridge.

Rabin, M. O. (1980). Probabilistic Algorithms for Primality Testing. *J. Number Theory*, **12**, 128–138.

Reif, J. H. and Tygar, J. D. (1988). Efficient Parallel Pseudorandom Number Generation. *SIAM J. Computing*, **17**, 2, 404–411.

Shamir, A. (1981). On the Generation of Cryptographically Strong Pseudo-Random Sequences. *Proceedings of the 8th International Colloquium on Automata, Languages and Programming*, Springer--Verlag, 544–550. Also in *ACM Trans. on Computer Systems*, 1, 1 (1983) 38–44.

Stern, J. (1987). Secret Linear Congruential Generators are not Cryptographically Secure. *Proceedings of the 28th IEEE Symposium on Foundations of Computer Science*, 421–426.

Vazirani, U. and Vazirani, V. (1984). Efficient and Secure Pseudo-Random Number Generation. *Proceedings of the 25th IEEE Symposium on Foundations of Computer Science*, 458–463.

Vazirani, U. and Vazirani, V. (1983). Trapdoor Pseudo-Random Number Generators with Applications to Protocol Design. *15-th Annual ACM Symp. on Theory of Computing*, 23–30.

Yao, A. C. (1982). Theory and Applications of Trapdoor Functions. *Proceedings of the 23rd IEEE Symposium on Foundations of Computer Science*, 80–91.

AUTHOR'S BIOGRAPHIES

PIERRE L'ECUYER is an associate professor in the Computer Science Department at Laval University, Ste-Foy, Québec, Canada. He received the B.Sc. degree in mathematics in 1972, and was a college teacher in mathematics from 1973 to 1978. He then received the M.Sc. degree in operations research and the Ph.D. degree in

computer science, in 1980 and 1983 respectively, both from the University of Montreal. His research interests are in Markov renewal decision processes, sensitivity analysis and optimization of discrete-event stochastic systems, random number generation, and discrete-event simulation software. He is a member of ACM, IEEE, ORSA and SCS.

Pierre L'Ecuyer
Département d'informatique
Pavillon Pouliot
Université Laval
Ste-Foy, G1K 7P4
Canada
(418) 656-3226

RENÉ PROULX is a scientific consultant in mathematical modeling for SOMAPRO Ltd., Québec. He is also a part-time master's student in computer science at Laval University. He received a B.Sc. in mathematics in 1978 and a M.Sc. in mathematics in 1983, both from Laval University. His professional interests are in modeling and analysis of physical systems, simulation and software development.

René Proulx
Somapro, Ltd.
84 St-Louis,
Québec, G1R 3Z5
Canada
(418) 692-3903