EFFICIENT, PERFECT RANDOM NUMBER GENERATORS

S. Micali
Laboratory for Computer Science
MIT

C.P.Schnorr*
Fachbereich Mathematik/Informatik
Universität Frankfurt

Abstract We describe a method that transforms every perfect random number generator into one that can be accelerated by parallel evaluation. Our method of parallelization is perfect, m parallel processors speed the generation of pseudo-random bits by a factor m; these parallel processors need not to communicate. Using sufficiently many parallel processors we can generate pseudo-random bits with nearly any speed. These parallel generators enable fast retrieval of substrings of very long pseudo-random strings. Individual bits of pseudo-random strings of length 10^{20} can be accessed within a few seconds. We improve and extend the RSA-random number generator to a polynomial generator that is almost as efficient as the linear congruential generator. We question the existence of polynomial random number generators that are perfect and use a prime modulus.

*) Research performed while visiting the Department of Computer Science of the University of Chicago.
MIT - Patent Pending

S. Goldwasser (Ed.): Advances in Cryptology - CRYPTO '88, LNCS 403, pp. 173-198, 1990.
© Springer-Verlag Berlin Heidelberg 1990

1. Introduction

A *random number generator* (RNG) is an efficient algorithm that transforms short random seeds into long pseudo-random strings. A classical RNG is the linear congruential generator (LCG) that is based on the recursion $x_{i+1} := ax_i + b \pmod N$. It is well known that the LCG passes certain statistical tests, e.g. for a clever choice of the parameters a,b,N it generates well mixed numbers (see Knuth 1980). There are more elaborate statistical tests which the LCG fails. Stern (1987) shows that the sequence generated by the LCG can be inferred even if the parameters a,b,N and the seed x_0 are all unknown.

The concept of *perfect* random number generator has been introduced by Blum, Micali (1982) and Yao (1982). A RNG is *perfect* if it passes all polynomial time statistical tests, i.e. the distribution of output sequences cannot be distinguished from the uniform distribution of sequences of the same length. So far the proofs of perfectness are all based on unproven complexity assumptions. This is because we cannot prove superpolynomial complexity lower bounds.

Perfect random number generators have been established for example based on the discrete logarithm by Blum, Micali (1982), based on quadratic residuosity by Blum, Blum, Shub (1986), based on one way functions by Yao (1982), based on RSA encryption and factoring by Alexi, Chor, Goldreich and Schnorr (1984). All these RNG's are less efficient than the LCG. The RSA/RABIN-generator is the most efficient of these generators. It successively generates log n pseudo-random bits by one modular multiplication with a modulus N that is n bit long. The modulus N must be at least 512 bits long.

We extend and accelerate the RSA-generator in various ways. We give evidence for more powerful complexity assumptions that yield more efficient generators. Let $N = pq$ be product of two large random primes p and q and let d be a natural number that is relatively prime to $\varphi(N) = (p-1)(q-1)$. The number d must be small compared to log N so that the interval $[1,N^{2/d}]$ is sufficiently large. We conjecture that the following distributions are indistinguishable by efficient statistical tests (see Hypothesis 2.1):

- the distribution of $x^d \pmod N$ for random $x \in [1,N^{2/d}]$.
- the uniform distribution on $[1,N]$.

This hypothesis is closely related to the security of the RSA-scheme. Under this

hypothesis the transformation

$$[1,N^{2/d}] \ni x \mapsto x^d \pmod N \in [1,N]$$

stretches short random seeds $x \in [1,N^{2/d}]$ into a pseudo-random numbers $x^d \pmod N$ in the interval $[1,N]$. We build various random number generators on this transformation. The sequential polynomial generator (SPG) generates from random seed $x \in [1,N^{2/d}]$ a sequence of numbers $x = x_1, x_2, \ldots, x_1, \ldots \in [1,N^{2/d}]$. The $n(1-2/d)$ least significant bits of the binary representation of $x_i^d \pmod N$ are the output of x_i and the $2n/d$ most significant bits form the successor x_{i+1} of x_i.

It follows from a general argument of Goldreich, Goldwasser, Micali (1986) and the above hypothesis that all these generators are perfect, i.e. the distribution of output strings is indistinguishable, by efficient statistical tests, from the uniform distribution of binary strings of the same length. The sequential generator is nearly as efficient as the LCG. Using a modulus N, that is n bit long, it outputs $n(1-2/d)$ pseudo-random bits per iteration step. The costs of an iteration step $x \mapsto x^d \pmod N$ with $x \in [1,N^{2/d}]$ corresponds to the costs of about one full multiplications modulo N. This is because the evaluation of $x^d \pmod N$ over numbers $x \leq N^{2/d}$ consists almost entirely of multiplications with small numbers that do not require modular reduction.

We extend the SPG to a parallel polynomial generator (PPG). The PPG generates from random seed $x \in [1,N^{2/d}]$ a tree. The nodes of this iteration tree are pseudo-random numbers in $[1,N^{2/d}]$ with outdegree at most $d/2$. To compute the successor nodes $y(1),\ldots,y(s)$ and the output string of node y we stretch y into a pseudo-random number $y^d \pmod N$ that is n bits long. Then the successors $y(1),\ldots,y(s)$ of y are obtained by partitioning the most significant bits of $y^d \pmod N$ into $s \leq d/2$ bit strings of length $\lfloor 2n/d \rfloor$. The output of node y consists of the remaining least significant bits of y^d(mod N). Any collection of subtrees of the iteration tree can be independently processed in parallel once the corresponding roots are given. In this way m parallel processors can speed the generation of pseudo-random bits by a factor m. These parallel processors need not to communicate; they are given pseudo-independent input strings and their output strings are simply concatenated. The concatenated output of all nodes of the iteration tree is pseudo-random, i.e. the parallel generator is perfect. The PPG enables fast retrieval of substrings of the pseudo-random output. To access a node of the iteration tree we follow the path from the root to this node. After retrieving a bit the subsequent

bits in the output can be generated at full speed. Iteration trees of depth at most 60 are sufficient for practical purposes; they generate pseudo-random strings of length 10^{20} (for outdegree 2) such that individual bits can be retrieved within a few seconds.

The parallel generator is based on a method that has been invented by Goldreich, Goldwasser and Micali (1984) for the construction of random functions. Our contribution consists of the observation that this construction can be applied to speed every perfect random number generator by a factor m using m parallel processors. Using this principle and sufficiently many parallel processors we can generate pseudo-random bits with almost any speed. This important method of parallelization applies to all perfect RNG's but the RSA-generator is particularly suited for this method. Our method of parallelization does not apply to imperfect RNG's as the LCG since this method can further detoriate a weak generator.

The paper is organized as follows. In section 2 we formulate our basic Hypothesis which is somewhat stronger than the assumption that factoring large integers is difficult. We give support to this hypothesis and show that a weak version of it follows from the assumption that the RSA-scheme is safe. We present in section 3 sequential and parallel random number generators that are based on this hypothesis. In the open problem session we question whether there exist perfect pseudo-random number generators that use a prime modulus. This would lead to pseudo-random number generators which use a modulus that is only 224 bits long.

2. The Complexity Assumption for the Polynomial Random Generator

Let P(x) be a polynomial of degree $d \geq 2$ with integer coefficients and let N be an integer that is n bits long, i.e. $2^{n-1} \leq N < 2^n$. We denote $l = \lfloor 2n/d \rfloor$. Residue classes modulo N are identified with the corresponding integers in the interval [1,N].

The polynomial generator is based on the transformation

$$[1,M] \ni x \to P(x) \bmod N \qquad (1)$$

where x ranges over a sufficiently large subinterval [1,M] of [1,N]. We would like that the outputs of (1), for random $x \in [1,M]$ and given N, M and P, be indistinguishable from random $y \in [1,N]$. The following conditions and restrictions are clearly necessary.

- the modulus N must be difficult to factor since given the factorization of N we can easily invert (1).

- The interval $[1,M]$ must be so large that a random seed $x \in [1,M]$ cannot be easily recovered from $P(x) \pmod N$ by guessing x. M must be sufficiently large to make $P(x)/N$ large for almost all $x \in [1,M]$. This is because we can easily invert (1) provided that $P(x)/N$ is small.

- $P(x)$ must not be a square polynomial. If $P(x) = Q(x)^2$ for some polynomial Q then the Jacobi-symbol $\left(\frac{P(x)}{N}\right)$ is 1 for all x whereas $\text{prob}\left[\left(\frac{y}{N}\right) = 1\right] = \text{prob}\left[\left(\frac{y}{N}\right) = -1\right]$ for random $y \in [1,N]$. Since the Jacobi-symbol can be evaluated efficiently we can distinguish $P(x) \bmod N$ from random numbers $y \in [1,N]$.

- $P(x)$ must not be a linear transform of a square polynomial. If $P(x) = aQ(x)^2 + b$ we can, from $P(x) \bmod N$, recover $Q(x)^2 \bmod N$ and check that $\left(\frac{Q(x)^2}{N}\right) = 1$.

We choose $N,M,P(x)$ as to correspond to these conditions. Let N be a random number that is uniformly distributed over the set

$$S_n = \left\{ N \in \mathbb{N} \;\middle|\; \begin{array}{l} N = p \cdot q \text{ for distinct primes } p,q \\ \text{such that } 2^{n/2-1} < p,q < 2^{n/2} \end{array} \right\}$$

of integers that are products of two distinct primes which each is n/2 bits long. We choose the interval length M proportional to $2^{2n/d}$, $M = \theta(2^{2n/d})$; i.e. $1/c \le 2^{2n/d} / M \le c$ for some absolute constant $c > 0$. Then M is proportional to $N^{2/d}$ for all $N \in S_n$. The choice for the polynomials $P(x)$ seems to be subject to only a few restrictions. We are going to study a particular class of permutation polynomials where the hypothesis below can be justified by known theory. These are the RSA-polynomials $P(x) = x^d$ with d relatively prime to $\varphi(N) = (p-1)(q-1)$.

Rivest, Shamir and Adleman (1978) have invented the RSA-cryptoscheme that is based on the multiplicative group

$$\mathbb{Z}_N^* = \{\, x (\text{mod } N) \mid \gcd(x,N) = 1 \,\}$$

of residue classes modulo N that are relatively prime to N. The integer N is product of two odd primes, $N = p \cdot q$. The order of the group $\mathbb{Z}_N^*$ is $\varphi(N) = (p-1)(q-1)$. The transformation

$$x \mapsto x^d \pmod N \qquad (2)$$

with $\gcd(\varphi(N),d) = 1$ is a permutation on the residue classes modulo N, i.e. it permutes the integers in the interval [1,N]. The inverse transformation is given by $x \mapsto x^e \pmod N$ where $e = d^{-1} \bmod \varphi(N)$. The permutation (2) with $\gcd(\varphi(N),d) = 1$ and $d \neq 1$ is an RSA-*enciphering function*. The enciphering key d does not reveal the inverse key e provided that $\varphi(N)$ is unknown. Knowledge of $\varphi(N)$ is equivalent to knowing the factorization $N = p \cdot q$. The security of the RSA-scheme relies on the assumption that RSA-enciphering $x \to x^d \pmod N$ is difficult to invert when d, N are given but $\varphi(N)$ and $e = d^{-1} \bmod \varphi(N)$ are unknown. All known methods for inverting RSA-enciphering require the factorization of N.

We are going to show that the following hypothesis is closely related to the security of the RSA-scheme. Our random number generators will rely on this hypothesis.

Hypothesis 2.1 *Let $d \geq 3$ be an odd integer and $l = \lfloor 2n/d \rfloor$. For random $N \in S_n$ such that $\gcd(d,\varphi(N)) = 1$ and for all $M = \theta(2^l)$ the following distributions on $[1,N]$ are indistinguishable by polynomial time statistical tests:*

- *the uniform distribution on $[1,N]$,*
- *$x^d \pmod N$ for random $x \in [1,M]$.*

We explain the hypothesis in more detail. The concept of a *statistical test* has been introduced by Yao (1982). A polynomial time *statistical test* is a sequence $T = (T_n)_{n \in \mathbb{N}}$ of probabilistic algorithms with a uniform polynomial time bound $n^{O(1)}$. According to Yao it is sufficient to consider statistical tests with 0,1-output. Let

$$p_n^T = \text{prob}[T_n(y,N) = 1]$$

be the probability that T_n outputs 1. The probability space is that of all integers $N \in S_n$ with $\gcd(d,\varphi(N)) = 1$, all numbers $y \in [1,N]$ and all 0-1 sequences of internal coin tosses, with uniform distribution. Let $\overline{p}_n^T(M)$ be the same probability with random numbers $y \in [1,N]$ replaced by $y = x^d \pmod N$ for random $x \in [1,M]$ and fixed d. The

hypothesis means that for every polynomial time statistical test T and all $M_n = \theta(2^l)$

$$\lim_n |p_n^T - \overline{p}_n^T(M_n)|\, n^t = 0 \quad \text{for all } t > 0. \qquad (3)$$

In particular the hypothesis means that any polynomial time algorithm can at most factor a negligible fraction of the integers in S_n. There are algorithms that can efficiently factor a very small fraction of the integers in S_n, e.g. Pollard's ρ-method efficiently factors all integers $N = p \cdot q$ such that either p-1 or q-1 is a product of small primes. But no algorithm is known that can factor in polynomial time a n^{-t}-fraction of the integers in S_n for some fixed $t > 0$.

We introduce some useful terminology. We say that the statistical test T *ϵ_n-rejects* RSA-ciphertexts $x^d(\text{mod } N)$ of random $x \in [1,M_n]$ if $|p_n^T - \overline{p}_n^T(M_n)| \geq \epsilon_n$ for infinitely many n. If (3) holds for all polynomial time statistical tests T we call RSA-ciphertexts $x^d(\text{mod } N)$ of random messages $x \in [1,M_n]$ *pseudo-random* in [1,N]. In this case the distributions of $x^d(\text{mod } N)$ for random $x \in [1,M_n]$ and the uniform distribution on [1,N] are called *indistinguishable*.

In general two sequences of distributions $(D_n)_{n \in \mathbb{N}}$ and $(\overline{D}_n)_{n \in \mathbb{N}}$ are called *indistinguishable* if for every pol. time statistical test $(T_n)_{n \in \mathbb{N}}$, that is given random inputs with respect to D_n ($\overline{D}_n$, resp.) the probability p_n^T ($\overline{p}_n^T$, resp.) of output 1 satisfy $\lim_n |p_n^T - \overline{p}_n^T| n^t = 0$ for all $t > 0$. In case of indistinguishable distributions D_n, $\overline{D}_n$, where D_n is the uniform distribution on set C_n, random elements with respect to $\overline{D}_n$ are called *pseudo-random* in C_n. In case of pseudo-random pairs (x,y) we call x and y *pseudo-independent*. A random number generator is called *perfect* if it transforms random seeds into pseudo-random strings.

It can easily be seen that the Hypothesis 2.1 can only fail if RSA-enciphering leaks partial information on RSA-messages.

Fact 2.2 *Suppose Hypothesis 2.1 fails. Then given d and N we can distinguish between RSA-ciphertexts $x^d(\text{mod } N)$ of random messages $x \in [1,N]$ and of random messages $x \in [1,M_n]$ for some $M_n = \theta(2^l)$.*

Proof The transformation $x \to x^d(\text{mod } N)$ permutes the integers in the interval [1,N].

The RSA-enciphering $x^d (\bmod N)$ of random messages $x \in [1,N]$ is uniformly distributed over $[1,N]$. If Hypothesis 2.1 fails the uniform distribution can be distinguished from RSA-ciphertexts $x^d (\bmod N)$ for random $x \in [1,M_n]$; i.e. RSA-ciphertexts $x^d (\bmod N)$ would leak information on whether the message x is contained in $[1,M_n]$. **QED**

Fact 2.2 does not mean that the RSA-scheme breaks down if the hypothesis fails. This is because messages in the interval $[1,2^l]$ are rather unlikely. Nevertheless the hypothesis is close to the security of the RSA-scheme. Using the following Theorem 2.3 we can relate the hypothesis to RSA-security (see Corollary 2.5).

Theorem 2.3 *Alexi, Chor, Goldreich, Schnorr (1984)*
Let d,N be integers such that $gcd(d,\varphi(N)) = 1$. Every probabilistic algorithm AL, which given the RSA-enciphering $x^d (\bmod N)$ of a message x, has an ε_N-advantage in guessing the least significant bit of the message x, can be transformed (uniformly in N) into a probabilistic algorithm $\overline{AL}$ for deciphering arbitrary RSA-ciphertexts. The deciphering algorithm $\overline{AL}$, when given for input $x^d (\bmod N)$, d and N, terminates after at most $O(\varepsilon_N^{-8} n^3)$ elementary steps and outputs x with probability at least 1/2.

We count for *elementary steps* the $\mathbb{Z}_N$-operations (addition, multiplication, division), RSA-encryptions and calls for algorithm AL at unit cost. We say that algorithm AL has an ε_N*-advantage* in guessing the least significant bit of x if

$$\text{prob}[AL(x^d (\bmod N),N) = x(\bmod 2)] \geq \frac{1}{2} + \varepsilon_N.$$

The probability space is the set of all $x \in [1,N]$ and all 0-1 sequences of internal coin tosses, with uniform probability.

By Theorem 2.3 the security of the RSA-scheme with parameters N, d implies that the following two distributions cannot be distinguished given only N and d:

- the uniform distribution on $[1,N]$,
- $x^d (\bmod N)$ for random, even $x \in [1,N]$.

Everyone who is able to distinguish these distributions can decode arbitrary RSA-ciphertexts $x^d (\bmod N)$ given only N and d. We will present in Corollary 2.4 a more formal version of this statement.

We say that a probabilistic algorithm AL ε_N*-rejects* the distribution D on [1,N] if

$$|p^A - \overline{p}^A| \geq \varepsilon_N$$

where p^A ($\overline{p}^A$, resp.) is the probability that AL on input $y \in [1,N]$ outputs 1. The probability space is the set of all $y \in [1,N]$, distributed according to D (with uniform distribution, resp.) and of all 0-1 sequences of internal coin tosses of algorithm AL. Using this notion we can reformulate Theorem 2.3 as follows.

Corollary 2.4 *Let d, N be integers such that* $gcd(d,\varphi(N)) = 1$*. Every probabilistic algorithm AL, that* ε_N*-rejects RSA-ciphertexts* $x^d (mod\ N)$ *of even random messages x can be transformed (uniformly in N) into a probabilistic algorithm for decoding arbitrary RSA-ciphertexts. This deciphering algorithm terminates after at most* $O(\varepsilon_N^{-8} n^3)$ *elementary steps (i.e.* $\mathbb{Z}_N$*-operations, RSA-encryptions and calls for AL).*

We next show that Corollary 2.4 remains valid if we replace RSA-ciphertexts of random even messages x, by RSA-ciphertexts of random messages $x \in [1,N/2]$.

Corollary 2.5 *Let d, N be odd integers such that* $gcd(d,\varphi(N)) = 1$*. Every probabilistic algorithm AL, that* ε_N*-rejects RSA-ciphertexts* $x^d (mod\ N)$ *of random messages* $x \in [1,N/2]$*, can be transformed (uniformly in N) into a probabilistic algorithm for decoding arbitrary RSA-ciphertexts. This deciphering algorithm terminates after at most* $O(\varepsilon_n^{-8} n^3)$ *elementary steps.*

Proof For odd N and all $x \in [1,N]$ we have

$$x \in [1,N/2] \iff 2x (mod\ N) \text{ is even}$$

(i.e. $x \in [1,N/2]$ iff the representative of $2x (mod\ N)$ in [1,N] is even).

We see from this equivalence that the following distributions are identical for odd N:

- $x^d (mod\ N)$ for random $x \in [1,N/2]$,
- $2^{-d} y^d (mod\ N)$ for random even $y \in [1,N]$.

Moreover we can transform in polynomial time $y^d (mod\ N)$ into $2^{-d} y^d (mod\ N)$. Thus an ε_N-rejection of RSA-encipherings $x^d (mod\ N)$ of random messages $x \in [1,N/2]$ can be transformed (uniformly in N) into an ε_N-rejection of RSA-ciphertexts $y^d (mod\ N)$ of random even $y \in [1,N]$. Corollary 2.5 follows from Corollary 2.4 by this transformation.

QED

Under the assumption that the RSA-scheme is safe Corollary 2.5 proves a slight

modification of our hypothesis. The interval $[1,2^l]$ of Hypothesis 2.1 is replaced by the interval $[1,N/2]$ in this modification. This poses the question whether the length of the interval is crucial for the hypothesis to be valid. We next show that Hypothesis 2.1, with the interval $[1,2^l]$ replaced by the interval $[1,N\ 2^{-\lceil \log n \rceil}]$, is valid if the RSA-scheme is safe.

Theorem 2.6 *Let d, N be odd integers such that $gcd(d,\varphi(N)) = 1$. Every probabilistic algorithm AL, that ε_N-rejects RSA-ciphertexts $x^d (mod\ N)$ of random messages $x \in [1, N2^{-k}]$, can be transformed (uniformly in N) into a probabilistic algorithm for decoding arbitrary RSA-ciphertexts. This deciphering algorithm terminates after at most $O(2^{2k}\ \varepsilon_N^{-8}\ n^3)$ elementary steps.*

Proof Under the assumption that the RSA-scheme is safe, Alexi et alii (1984) have shown that the log n least significant bits of RSA-messages x are pseudo-random when given $x^d (\text{mod } N)$, d and N. Their proof transforms every algorithm AL, that ε_N-rejects RSA-encipherings $x^d (\text{mod } N)$ of random messages x satisfying $x = 0 (\text{mod } 2^k)$, (uniformly in N) into a probabilistic algorithm for deciphering arbitrary RSA-ciphertexts. This RSA-deciphering procedure terminates after at most $O(2^{2k}\ \varepsilon_N^{-8}\ n^3)$ elementary steps (i.e. $\mathbb{Z}_N$-operations, RSA-encipherings and calls for algorithm AL).

For odd N and all $x \in [1,N]$ we obviously have

$$x \in [1,N2^{-k}] \quad \Leftrightarrow \quad 2^k x (\text{mod } N) = 0 (\text{mod } 2^k)$$

(i.e. $x \in [1,N\ 2^{-k}]$ iff the representative of $2^k x (\text{mod } N)$ in $[1,N]$ is a multiple of 2^k).
Therefore the following two distributions are identical for odd N:

- $x^d (\text{mod } N)$ for random $x \in [1,N2^{-k}]$,
- $2^{-kd} y^d (\text{mod } N)$ for random $y \in [1,N]$ satisfying $y = 0 (\text{mod } 2^k)$.

Moreover we can transform in polynomial time $y^d (\text{mod } N)$ into $2^{-kd} y^d (\text{mod } N)$. Thus an ε_N-rejection of RSA-ciphertexts $x^d (\text{mod } N)$ of random messages $x \in [1,N\ 2^{-k}]$ can be transformed (uniformly in N) into an ε_N-rejection of RSA-ciphertexts $y^d (\text{mod } N)$ of random messages y satisfying $y = 0 (\text{mod } 2^k)$. Corollary 2.6 follows from this transformation and the above mentioned proof of Alexi et alii (1984). **QED**

Notice that the time bound for the RSA-deciphering algorithm of Corollary 2.6 is polynomially related to the time bound of algorithm AL provided that $k \leq \log n$. Hence if Hypothesis 2.1 fails, with the interval $[1,2^l]$ replaced by the interval $[1,\ N2^{-\lceil \log n \rceil}]$,

then RSA-ciphertexts can be deciphered in probabilistic polynomial time. Also if Hypothesis 2.1 fails, with the interval $[1,2^l]$ replaced by the interval $[1, N2^{-\lfloor\sqrt{n}\rfloor}]$, then RSA-ciphertexts can be deciphered in time $e^{O(\sqrt{n})}$. However the fastest known algorithm for RSA-deciphering, via factoring N, requires about $e^{0.693\sqrt{n \log n}}$ steps, where $0.693 \approx \log 2$. Thus if Hypothesis 2.1 fails for the interval $[1, N2^{-\lfloor\sqrt{n}\rfloor}]$, then we can speed up the presently known attacks to the RSA-scheme.

It remains the question whether the computational properties of the distribution $x^d \pmod N$ change when x ranges over very small integers x. In fact Hypothesis 2.1 does not hold for the interval $[1,N^{1/d}]$ since we have $x^d < N$ for all $x \in [1,N^{1/d}]$ and therefore RSA-ciphertexts $x^d \pmod N$ can easily be deciphered for $x \in [1,N^{1/d}]$. On the other hand the d-powers x^d are of order N^2 for almost all numbers $x \in [1,2^l]$. We conjecture that this is sufficient to make the task of deciphering $x^d \pmod N$ hard. This is justified because inverting the squaring

$$x \rightarrow x^2 \pmod N$$

is known to be as hard as factoring N, and the squares x^2 are of order N^2, too.

We are going to study the question whether Hypothesis 2.1 should be extended to polynomials P(x) that are more general than RSA-polynomials $P(x) = x^d$ with $\gcd(d,\varphi(N)) = 1$. There is an obvious extension of Hypothesis 2.1 to arbitrary exponents $d \geq 2$. It seems that the condition $\gcd(d,\varphi(N)) = 1$ is not necessary for odd d. This is because no extension of the Jacobi-symbol is known for residues $x^d \pmod N$ of odd prime powers d. On the other hand we must modify the hypothesis for even d since the Jacobi-symbol gives efficient information on the quadratic residuosity. We formulate the extended hypothesis so that it can be applied in the proof of Theorem 3.1 to establish perfect RNG's. For reasons of efficiency we are particularly interested in even exponents d and in exponents that are powers of 2.

Extension to even d of Hypothesis 2.1 *For random $N \in S_n$, all $M = \theta(2^l)$, $l = \lfloor 2n/d \rfloor$, and random $x \in [1,M]$ the following holds.*

(1) $y := x^d \pmod N$ is a pseudo-random quadratic residue modulo N.

(2) Partitioning y into disjoint sections $z := \lfloor y\, 2^{-n+l} \rfloor$ and $y \pmod{2^{n-l}}$ yields pseudo-random numbers in $[1, N\, 2^{-n+l}]$ and $[1,2^{n-l}]$.

(3) $z^d \pmod N$ and $y \pmod{2^{n-l}}$ are pseudo-independent.

Article (1) of the extended hypothesis can be justified by the work of Alexi et alii (1984) for the case that N is a *Blum-integer*, i.e. N is product of two primes p and q such that $p \equiv 3 \pmod 4$ and $q \equiv 3 \pmod 4$. One can prove that distinguishing $x^d \pmod N$, for random $x \in [1, N n^{-1}]$ from random quadratic residues modulo N is equivalent, by probabilistic polynomial time reductions, to factoring N. Article (2) means that neither z nor $y \pmod{2^{n-l}}$ contains efficient information on the quadratic residuosity of y. Article (3) means that the dependence of z and $y \pmod{2^{n-l}}$, via the quadratic residuosity of y, gets hidden by the transformation $z \to z^d \pmod N$.

Next we consider arbitrary polynomials P(x) of degree d. We are going to show that some elementary methods for distinguishing random numbers $y \in [1,N]$ and P(x) mod N for random $x \in [1,N^{2/d}]$ do not work. Theorem 2.7 is a first step in this direction. This problem clearly deserves further study.

In general we can invert the transformation

$$x \to P(x) \bmod N \tag{1}$$

only if the factorization $N = pq$ is given. Then, using Berlekamps algorithm for polynomial factorization we invert (1) modulo p and modulo q and apply the Chinese remainder construction. This can be done in probabilistic time $(nd)^{O(1)}$. Without knowing the factorization of N we do not know how to invert (1). In the particular case that P(x) divides $x^{\varphi(N)}$ we can invert (1) provided that we know the cofactor $x^{\varphi(N)}/P(x)$, but in this case we can even factor N.

Can we invert (1) for small integers x ? If $|P(x)| / N$ is small we can guess $z = P(x)$ and factorize $P(x) - z$. Theorem 2.7 below shows that $|P(x)|/N$ is large for almost all $x \in [1,N^{2/d}]$ provided that P(x) has degree at most d. A degree bound is necessary since there exist polynomials of degree $N^{2/d}$ that vanish on the intervall $[1,N^{2/d}]$.

Theorem 2.7 *Let A,B,d be integers such that* $M \geq (BN)^{1/d}\ 16Ad$, *and let* $P(x) \in \mathbb{Z}[x]$ *have degree d. Then we have* $\text{prob}[|P(x)| \leq BN] \leq 1/A$ *for random* $x \in [1,M]$.

Proof Let $x_1,\dots,x_k$ be the distinct real numbers in [0,N] satisfying $P(x_i)^2 = B^2N^2$ for $i=1,\dots,k$. We have $k \leq 2d$ since $P(x)^2$ has degree 2d. We partition the real interval [0,M]

into 4Ad intervals I of length M/(4Ad). A fundamental theorem in approximation theory (see e.g. Stiefel (1969), p. 236) implies that

$$\max[P(x)^2 \mid x \in I] \geq \left(\frac{M}{8Ad}\right)^{2d} 2^{-2d+1}$$

for each of these intervals I. Hence

$$\max[\,|P(x)|\; x \in I\,] > \left(\frac{M}{16Ad}\right)^{d} \geq BN.$$

This shows that every interval I, that contains an integer x satisfying $|P(x)| \leq BN$, must also contain some point x_i, $1 \leq i \leq k$. The intervals I that contain some point x_i can have at most

$$2d\left(\frac{M}{4Ad} + 1\right) \leq \frac{M}{2A} + 2d$$

integer points. This accounts for at most a fraction of

$$\frac{1}{2A} + \frac{2d}{M} \leq 1/A$$

of the points in [1,M]. **QED**

3. The Sequential and the Parallel Polynomial Generator

In this section we build several RNG's on polynomials P(x) of degree $d \geq 2$ that have the following **generator** property. The *generator property* formulates Hypothesis 2.1 for arbitrary polynomials P(x).

Definition The polynomial P(x) has the *generator property* if for random $N \in S_n$, all M proportional to $N^{2/d}$ and random $x \in [1,M]$ the number P(x) mod N is pseudo-random in [1,N].

The generator property means that P stretches random seeds $x \in [1,N^{2/d}]$ into pseudo-random numbers P(x) mod N in the interval [1,N]. By Hypothesis 2.1 RSA-polynomials $P(x) = x^d$ with $\gcd(d,\varphi(N)) = 1$ and $d \geq 3$ have the generator property.

The *sequential polynomial generator* (SPG) generates a sequence of numbers $x = x_1, x_2, \ldots, x_i, \ldots$ in $[1, N^{2/d}]$ that are represented by bit strings of length $l := \lfloor 2n/d \rfloor$. The *output* at x_i, $Out(x_i) \in \{0,1\}^{n-l}$, is the bit string consisting of the $n-l$ least significant bits of the binary representation of $P(x_i)$ mod N. The *successor* x_{i+1} of x_i is the number corresponding to the other bits of $P(x_i)$ mod N,

$$x_{i+1} := \lceil P(x_i) \bmod N \; / \; 2^{n-l} \rceil \, .$$

The sequential polynomial generator can be figured by the following infinite tree

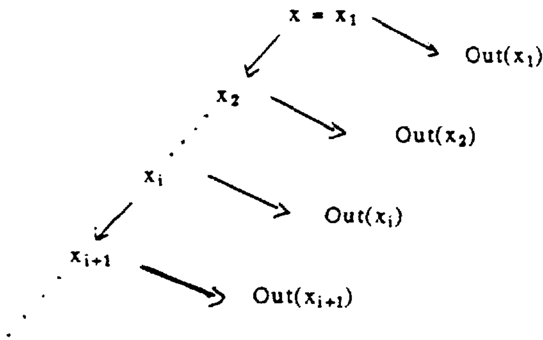

figure of the sequential polynomial generator (SPG)

Let the *k-output* of the SPG

$$SPG_{k,P}(x,N) = \prod_{i=1}^{k} Out(x_i)$$

be the concatenated output of the first k steps.

Notice that the most significant bits of $P(x_i)$ mod N are biased depending on the most significant bits of N. Even though the most significant bits of $P(x_i)$ mod N are not pseudo-random we can form from these bits the successor x_{i+1} of x_i. This is because the generator property and Hypothesis 2.1 imply that $P(x_i)$ mod N is pseudo-random if x_i is random in $[1,M]$, *for all* M proportional to 2^l.

Theorem 3.1 *Suppose that P has the generator property. Then for random $N \in S_n$, random*

$x \in [1, N^{2/d}]$ and polynomially bounded k (i.e. $k = k(n) = n^{O(1)}$) the k-output $SPG_{k,P}(x,N)$ of the sequential polynomial generator is pseudo-random.

Proof For random $N \in S_n$ and random $x_1 \in [1, N^{2/d}]$ the number $P(x_1) \bmod N \in [1,N]$ is pseudo-random. It follows that the bit string $Out(x_1) \in \{0,1\}^{n-l}$ is pseudo-random and that the number $x_2 \in [1, 2^l]$ is pseudo-random. We also see that the pair $(Out(x_1), x_2)$ is pseudo-random. It follows from the generator property and since x_2 is pseudo-random that

$$(Out(x_1)\, Out(x_2),\ x_3) = (SPG_{2,P}(x_1,N),\ x_3)$$

is pseudo-random, too. To prove this claim we replace in a statistical test $T = (T_n)_{n \in \mathbb{N}}$ for $z := (Out(x_1)\, Out(x_2),\ x_3)$ the pair $(Out(x_2),\ x_3)$ (the string $Out(x_1)$, resp.) by random objects generated through internal coin tosses. This transforms T into statistical tests for $P(x_1) \bmod N$ ($P(x_2) \bmod N$, resp.). If z is ε_n-rejected then either $P(x_2) \bmod N$ or $P(x_1) \bmod N$ is $(\varepsilon_n/2)$-rejected. In either case this yields a statistical test that $(\varepsilon_n/2)$-rejects $P(x_1) \bmod N$.

By induction on k the same argument proves that

$$(SPG_{k,P}(x_1,N),\ x_{k+1})$$

is pseudo-random for every fixed k. The pseudo-randomness also holds if $k = k(n)$ is polynomially bounded in n, i.e. $k = n^{O(1)}$. Using the above argument we can transform a test that ε_n-rejects $(SPG_{k,P}(x_1,N),\ x_{k+1})$ into a test that (ε_n/k)-rejects $P(x_1) \bmod N$. QED

It is important that the above proof also applies to polynomials $P(x) = x^d$ with even d. Instead of using the generator property of P we can use the extension to even d of Hypothesis 2.1. Speaking informally, it does not hurt that $x^d (\bmod N)$ ranges over quadratic residues since the output merely contains the least significant bits of $x^d (\bmod N)$ and these bits give no efficient information on the quadratic residuosity of $x^d (\bmod N)$. E.g. we can use for random bit generation the polynomial $P(x) = x^8$ which yields particular efficient RNG's.

PRACTICAL SEQUENTIAL POLYNOMIAL GENERATORS: The modulus N and the number $N^{2/d}$ must be fixed in practical applications. We study the complexity conditions

that N and $N^{2/d}$ must satisfy to prevent an efficient analysis of the generator output.

It must be practically impossible to factor the modulus N. For this let N be product of two random primes p and q which each is at least 256 bits long. The numbers p-1, p+1, q-1, q+1 must each have at least some prime factor which is larger than 2^{80}.

The number $N^{2/d}$ must be so large that, given x^d(mod N), it is practically impossible to find $x \in [1,N^{2/d}]$ by efficient search methods. Pollard (1988) has proposed the following method to search for an input x that is product $x = uv$ of two numbers $u,v \in [1,N^\alpha]$:

1. Generate the set $S_1 = \{u^d (\text{mod } N) \mid u \in [1,N^\alpha]\}$ and sort this set.
2. Generate the set $S_2 = \{x^d v^{-d} (\text{mod } N) \mid v \in [1,N^\alpha]\}$ and sort this set.
3. Test whether S_1 and S_2 have a common element. If $u^d = x^d v^{-d} (\text{mod } N) \in S_1 \cap S_2$ then one has found $x = uv$.

Pollard's attack performs $O(N^\alpha)$ arithmetical steps modulo N and stores N^α residues modulo N. It is most efficient when x is product of two numbers in $[1,N^{1/d}]$. In order to make Pollard's attack infeasible it is sufficient that $N^{1/d}$ is at least 2^{64}.

Example 1: Let N be $n = 512$ bits long and let $\gcd(7,\varphi(N)) = 1$. We choose $d = 7$, $P(x) = x^7$. Let $Out(x_i)$ consist of the 365 least significant bits of $P(x_i)$ mod N and let x_{i+1} be the number corresponding to the 128 most significant bits of $P(x_i)$ mod N. We compute x^7(mod N) by computing x^2, x^4, $x^7 = x \cdot x^2 \cdot x^4$. Only the last multiplication requires modular reduction. The other multiplications are with small numbers. The costs of one iteration step correspond to one full modular multiplication. Thus this SPG iteratively outputs 384 pseudo-random bits at the cost of one full modular multiplication with a modulus that is 512 bits long.

Example 2: Another suitable polynomial is $P(x) = x^8$ even though this polynomial does not have the generator property. The computation of x^8(mod N) is particularly easy; we compute x^2, x^4, x^8 by successive squaring. The SPG with $P(x) = x^8$ iteratively outputs 384 bits at the cost of one full modular multiplication with a modulus N that is 512 bits long.

Efficient public key encoding and decoding. We can use the above RNG's to generate a one-time-pad for message encoding. When given the seed x_1 of the one-time-pad, encoding and decoding can be done at a speed of about $n(1-2/d)$ bits per multiplication modulo N. A public key coding scheme as e.g. RSA can be used to encode and to decode the seed x_1.

The parallel polynomial generator. The *parallel polynomial generator* (PPG) generates from random seed $x \in [1, N^{2/d}]$ a tree with root x and outdegree at most d/2. The nodes of this *iteration tree* are pseudo-random numbers in $[1, N^{2/d}]$ that are represented by bit strings of length l.

The *successors* $y(1),\dots,y(s)$ of a node y with degree s and the *output string* Out(y) of node y are defined as follows. Let $b_1,\dots,b_n$ be the bits of the binary representation of P(y) mod N, with b_1 being the most significant bit, i.e.

$$\sum_{i=1}^{n} b_i \, 2^{n-i} = P(y) \bmod N .$$

We partition the sl most significant bits into s block with l bits in each block. The corresponding numbers

$$y(j) := 1 + \sum_{i=1}^{l} b_{(j-1)l+i} \, 2^{l-i} \qquad \text{for } j = 1,\dots,s$$

are the successors of node y in the iteration tree. The *output* Out(y) at node y consists of the remaining low order bits of P(y) mod N,

$$\mathrm{Out}(y) = b_{sl+1} \cdots b_n .$$

For convenience we denote the nodes on level k of the iteration tree as $x(j_1,\dots,j_k)$; $x(j_1,\dots,j_{k-1})$ is the direct predecessor of $x(j_1,\dots,j_k)$ and j_k ranges from 1 to s_{k-1} = "outdegree of $x(j_1,\dots,j_{k-1})$". For simplicity we let the outdegree of node $x(j_1,\dots,j_k)$ be a function depending on k only; we assume that $s_k \geq 1$.

The parallel polynomial generator can be figured by the following infinite tree

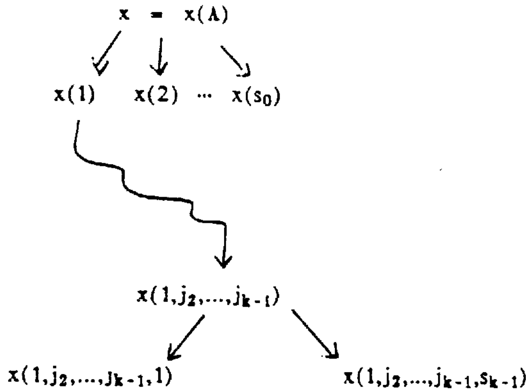

figure of the parallel polynomial generator (PPG)

We define the *k-output* $PPG_{k,P}(x,N)$ of the PPG with seed x as the concatenation of all bit strings $Out(x(j_1,...,j_i))$ on levels i with $0 \le i \le k$, with respect to any efficient enumeration order, as e.g. preorder traversal, postorder traversal, inorder traversal or enumeration by levels.

In the particular case that all outdegrees are one, i.e. $s_0 = s_1 = ... = s_k = 1$, the parallel and the sequential polynomial generator coincide. The argument of Goldreich, Goldwasser and Micali (1986) extends Theorem 3.1 from the SPG to arbitrary PPG's, provided that we process at most polynomially many nodes in the iteration tree. This yields the following theorem.

Theorem 3.2 *Suppose that P has the generator property. Then for random $N \in S_n$, random $x \in [1,2^l]$ the k-output $PPG_{k,P}(x,N)$ of the parallel polynomial generator is pseudo-random provided that the length of $PPG_{k,P}(x,N)$ is polynomially bounded.*

Idea of proof There is a straightforward way to extend the proof of Theorem 3.1. Suppose that the k-output $PPG_{k,P}(x,N)$ collects the outputs of $\overline{k}$ nodes. Then every statistical test that ε_n-rejects $PPG_{k,P}(x,N)$ for random $x \in [1,N^{2/d}]$ and random $N \in S_n$ can be transformed into a statistical test that $(\varepsilon_n/\overline{k})$-rejects $P(x) \bmod N$. QED

For the output of the PPG we can use any efficient enumeration for the nodes of the iteration tree. To support parallel evaluation we can adjust the shape of the iteration tree and the enumeration order to the number of available parallel processors. For m parallel processors we can use any iteration tree consisting of m isomorphic subtrees attached to the root; we can enumerate, in any order, the m-tuples of corresponding nodes in these subtrees. The enumeration within the subtrees can be chosen to support fast retrieval; for this we can enumerate the nodes e.g. in preorder traversal or in inorder traversal. It is an obvious but important observation that m processors can speed the pseudo-random bit generation of the PPG by a factor m. Once we are given m nodes on the same level of the iteration tree we can process the subtrees below these nodes independently by m parallel processors. These processors do not need to communicate.

Corollary 3.3 *Using m processors in parallel we can speed the pseudo-random bit generation of the parallel polynomial generator by a factor m.*

PRACTICAL PARALLEL POLYNOMIAL GENERATORS

Let N be product of two random primes so that N is 512 bits long. Let $P(x) = x^8$.

Example 3: We construct from random $x \in [1,2^{128}]$ a tree with 4 nodes per level.

1. Stretch a random seed $x \in [1,2^{128}]$ into $x^8 (\text{mod } N)$.
2. Partition the binary representation of $x^8 (\text{mod } N)$ into 4 bit strings $x(1),\ldots,x(4)$ of length 128. Put $k = 1$ and let $PPG_{1,P}(x,N)$ the empty string.
3. For $j = 1,\ldots,4$ let $x(j\ 1^k) \in I_{128}$ consist of the 128 most significant bits of the binary representation of $x(j\ 1^{k-1})^4 \text{ mod } N$, and let $Out(x(j\ 1^k)) \in I_{384}$ consist of the remaining 384 least significant bits.
4. $$PPG_{k+1,P}(x,N) = PPG_{k,P}(x,N) \prod_{j=1}^{4} Out(x(j\ 1^k))$$

   $k := k + 1$, go to 3.

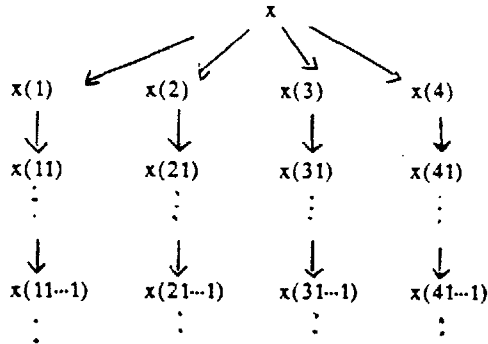

Figure of the PPG of example 3

Using 4 parallel processors this PPG iteratively generates $4 \cdot 384 = 1536$ pseudo-random bits in the time for one full modular multiplication with a modulus N that is 512 bits long. With current processors for smart cards such a full modular multiplication can be done in less than 0.2 sec. Thus 4 parallel processors can generate about 9000 pseudo-random bits per sec.

Example 4: We construct from random $x \in [1,2^{128}]$ a complete tree of outdegree 2.

1. Choose a random seed $x \in [1,2^{128}]$ for root of the tree.
2. For every node $y \in [1,2^{128}]$ of the tree compute the successors y(1), y(2) and the output Out(y) by partitioning the binary representation B of $y^8 (\mathrm{mod}\ N)$ as

$$B = B_1\ B_2\ \mathrm{Out}(y) \in I_{128}^2 \times I_{256}\ ,$$

   and compute for $i = 1,2$

$$y(i) := 1 + \text{"the number with binary representation } B_i\text{"}.$$

The main interest in such a PPG comes from fast retrieval methods.

Fast retrieval for the PPG. If the PPG has a complete iteration tree one can efficiently retrieve substrings of the output. Consider example 4 with a complete iteration tree of outdegree 2. Level k of the tree has 2^k nodes and the first k levels have $2^{k+1} - 1$ nodes in total. Suppose the nodes of the tree are enumerated in preorder traversal. Each node yields 256 output bits. To retrieve node y we follow the path from the root to y. This requires processing and storage of at most k nodes and can be done at the costs of about k full modular multiplications. Once we have retrieved node y and stored the path from

the root to node y, the bit string that follows Out(y) in the output can be generated using standard retrieval methods at the speed of 256 bits per modular multiplication. For most practical applications the depth k will be at most 60 which permits to generate a pseudo-random string that is $3.7 \cdot 10^{20}$ bits long. We see that retrieval of substrings is very efficient, it merely requires a preprocessing stage of a few seconds to retrieve the initial segment of the substring.

Theorem 3.4 *Every node y of depth k in the iteration tree of the PPG can be accessed and processed at the costs of O(k) modular multiplications.*

k	10	20	30	40	50	60
# nodes in the first k levels	2047	$2 \cdot 10^6$	$2.1 \cdot 10^9$	$2.2 \cdot 10^{12}$	$2.25 \cdot 10^{15}$	$2.3 \cdot 10^{18}$
# output bits	$5.2 \cdot 10^5$	$5.7 \cdot 10^8$	$5.5 \cdot 10^{11}$	$5.6 \cdot 10^{14}$	$5.8 \cdot 10^{17}$	$5.9 \cdot 10^{20}$

Table: retrieval performance of the PPG, example 4

Parallelization and fast retrieval for arbitrary perfect RNG's. It is an important observation that the above methods of parallelization and of efficient retrieval apply to every perfect RNG $(G_n)_{n \in \mathbb{N}}$. The parallel version of the generator associates an iteration tree to a random seed. For example let $G_n : I_n \to I_{3n}$ stretch a random strings in I_n into pseudo-random strings in I_{3n}. We construct from random seed $x \in I_n$ a binary iteration tree with nodes in I_n. Let x be the root of the tree. Construct the two successors y(1), y(2) and the output Out(y) of node y by partitioning $G_n(y) \in I_{3n}$ into three substrings of length n,

$$G_n(y) = y(1)\ y(2)\ \mathrm{Out}(y)\ .$$

Let $PG_{k,G}(x)$ be the concatenated output of all nodes with depth at most k (compare with the definition of $PPG_{k,P}(x,N)$).

Theorem 3.5 *Let $(G_n)_{n \in \mathbb{N}}$ be any perfect RNG. Then for random seed $x \in I_n$ the concatenated output $PG_{k,G}(x)$ of all nodes with depth $\leq k$ is pseudo-random provided that its length is polynomially bounded in n.*

We illuminate our method of parallelization in applying it to some less efficient versions of the RSA/Rabin generator. Let N be a product of two random primes such that N is 512 bits long and $\gcd(3, \varphi(N)) = 1$.

Example 5: From random seed $x \in [1,N]$ we generate the sequence of numbers $x_1, x_2, \ldots, x_i, \ldots \in [1,N]$ as

$$x_1 = x\ , \quad x_{i+1} = x_i^3 (\mathrm{mod}\ N)\ .$$

Under the assumption that the RSA-enciphering $x \to x^3 (\mathrm{mod}\ N)$ is safe for the *particular* N, Alexi et alii (1984) have shown that about the 16 least significant bits of x_i are pseudo-independent from x_{i+1}. This suggest the following output of x_i

$$\mathrm{Out}(x_i) = \text{"the 16-least significant bits of } x_i\text{"}.$$

Thus for random $x_1 \in [1,N]$ and under the assumption that RSA-enciphering is safe we obtain pseudo-random bit strings $\prod_{i=1}^{100} \mathrm{Out}(x_i)$ of length 1600. We apply a binary tree construction to the function

$$G : I_{512} \to I_{1600}$$

that stretches the binary representation of $x_1 \in [1,N]$ into $\prod_{i=1}^{100} \mathrm{Out}(x_i)$. The binary tree has nodes in I_{512}. The successors y(1), y(2) and the output of node y are obtained by partitioning G(y) into two successor strings of length 512 and an output string $\mathrm{Out}_G(y) \in I_{576}$. Processing a node of the binary iteration tree costs 200 modular multiplication.

Example 6: We can accelerate this generator under the reasonable assumption that the 448 least significant bits of the number x and the number $x^3 (\mathrm{mod}\ N)$ are pseudo-independent for random $x \in [1,N]$. We set

$$\mathrm{Out}(x_i) := \text{"the 448 least significant bits of } x_i\text{"}\ .$$

The assumption implies that $\prod_{i=1}^{3} \mathrm{Out}(x_i) \in I_{1344}$ is pseudo-random for random $x_1 \in [1,N]$. We apply the binary tree construction to the function

$$G : I_{512} \to I_{1344}$$

that stretches the binary representation of $x_1 \in [1,N]$ into $\prod_{i=1}^{3} \mathrm{Out}(x_i)$. The successors $y(1), y(2) \in I_{512}$ and the output $\mathrm{Out}_G(y) \in I_{320}$ of node y are obtained by partitioning $G(y) \in I_{1344}$ into two strings in I_{512} and $\mathrm{Out}_G(y) \in I_{320}$. Processing a node of the binary tree costs 6 modular multiplications.

Example 7: We can further speed up this generator under the assumption that the 448 least significant bits of random $x \in [1,N]$ and the number $x^2 \pmod N$ are pseudo-independent. (It follows from Alexi et alii (1984) that the 16 least significant bits of random $x \in [1,N]$ and the number $x^2 \pmod N$ are pseudo-independent if factoring the *particular* N is hard. Under this assumption we can replace the iteration $x_i := x_{i+1}^3 \pmod N$ by $x_{i+1} := x_i^2 \pmod N$. As in Example 5 we associate with a random $x \in [1,N]$ a binary iteration tree with nodes in I_{512}. Processing a node of this tree costs about 4 modular multiplications and yields 320 pseudo-random bits for output.

It is interesting to compare the efficiency of these parallel RNG's with the parallel RNG's based on Hypothesis 2.1. For the latter RNG's in examples 1-4 the cost per node of the iteration tree is about 1 multiplication modulo N. This shows that the new perfect RNG's are more suitable for our method of parallelization and fast retrieval.

4. Open Problems: Random Number Generators Based on a Prime Modulus

In Hypothesis 2.1 we need that the modulus N is difficult to factor. This is because given the factorization of N and given $x^d \pmod N$ we can recover $x = x^{de} \pmod N$ using the inverse exponent $e = d^{-1} \pmod{\varphi(N)}$. Now suppose we are only given the least significant bits of $x^d \pmod N$. Then we cannot easily recover x even if $d^{-1} \pmod{\varphi(N)}$ is known. This poses the question whether Hypothesis 2.1 can be extended to arbitrary prime moduli p.

Problem 4.1. Let p be an arbitrary prime, $2^{n-1} < p < 2^n$, let d be relatively prime to p-1, $d \geq 3$ and let $l \geq \lfloor 2n/d \rfloor$. Is it true that for random $x \in [1,2^l]$ and $y := x^d \pmod p$ the $n - l$ least significant bits of y are pseudo-random?

If this pseudo-randomness does not hold for all primes we ask whether it holds for random primes.

Problem 4.2. Let $d \geq 3$, $l \geq \lfloor 2n/d \rfloor$ and let p be a random prime such that $2^{n-1} < p < 2^n$ and $\gcd(d,p-1) = 1$. Is it true that for random $x \in [1,2^l]$ and $y := x^d \pmod p$ the $n-l$ least significant bits of y are pseudo-random?

If we replace in Problem 4.2 the prime modulus p by a random composite modulus in S_n

the pseudo-randomness in question follows from Hypothesis 2.1. These problems are important since this would modify Hypothesis 2.1 so that it is no more related to the difficulty of factoring the modulus. We consider the random number generators that would follow.

The sequential generator using a prime modulus The SPG generates from a random seed $x_1 \in [1,2^l]$ a sequence of numbers $x_1, x_2, \ldots, x_i \in [1,2^l]$ that are represented by bit strings of length l. The *output* at x_i, $out(x_i) \in \{0,1\}^{n-2l}$, is the bit string consisting of the n-2l least significant bits of the binary representation of $x_i^d \pmod p$. The *successor* x_{i+1} of x_i is the number corresponding to the next l least significant bits of $x_i^d \pmod p$; these are the bits in positions $n-l, \ldots, n-2l+1$ from the left.

Corollary 4.3 *If pseudo-randomness holds in problem 4.2, then the above SPG transforms for random prime p with $2^{n-1} < p < 2^n$ and every k with $k = n^{O(1)}$ a random seed $x_1 \in [0,\ldots,p-1]$ into a pseudo-random output $\prod_{i=1}^{k} out(x_i)$.*

In practical applications the number l must be so large that, given the n-l least significant bits of $x^d \pmod p$, it is practically impossible to find $x \in [1,2^l]$. Now Pollard's attack (see section 3) does not work since the most significant bits of $x^d \pmod p$ are unknown. Therefore it would be sufficient to start with a random seed x_1 that is 64 bits long.

Example 8: Let p be a prime that is 224 bits long, let $\gcd(p-1,7) = 1$, $d = 7$ and $l = 64$. The output $Out(x_i)$ consists of the 96 least significant bits of $x_i^7 \pmod p$, the successor x_{i+1} of x_i is formed by the next 64 least significant bits of $x_i^7 \pmod p$. The 64 most significant bits of $x_i^7 \pmod p$ are not used at all. Each iteration step generates 96 pseudo-random bits roughly at the cost of one full modular multiplication with a modulus that is 224 bits long.

If we choose a 512 bit long prime modulus p and $d = 7$, $l = 64$ then we can output 384 pseudo random bits per iteration. This achieves the same performance that is obtained with a composite modulus of the same length, see example 1. However using a prime modulus that is about 224 bits long the arithmetic can be done with much smaller numbers, and thus the generator can be implemented on a cheaper chip.

Acknowledgement The second author wishes to thank the Department of Computer Science of the University of Chicago for supporting the research of this paper which was done during a stay at this department. He also wishes to thank A.K. Lenstra and A. Shamir for very inspiring discussions during this work.

References

Alexi, W., Chor, B., Goldreich, O., and Schnorr, C.P.: RSA and Rabin Functions: certain parts are as hard as the whole. Proceeding of the 25th Symposium on Foundations of Computer Science, 1984, pp. 449-457; also: Siam Journal on Comput., 17,2 (1988).

Blum, L., Blum, M. and Shub, M.: A simple unpredictable pseudo-random number generator. Siam J. on Computing (1986), pp. 364-383.

Blum, M. and Micali, S.: How to generate cryptographically strong sequences of pseudo-random bits. Proceedings of the 25th IEEE Symposium on Foundations of Computer Science, IEEE, New York (1982); also Siam J. Comput. 13 (1984), pp. 850-864.

Goldreich, O., Goldwasser, S., Micali, S.: How to Construct Random Functions. Proceedings of the 25th IEEE Symposium on Foundations of Computer Science, IEEE, New York, (1984); also Journal ACM 33,4 (1986), pp. 792-807.

Knuth, D.E.: The Art of Computer Programming. Vol. 2, second edition. Addison Wesley (1981).

Luby, M. and Rackoff, Ch.: Pseudo-random permutation generators and cryptographic composition. Proceedings of the 18th ACM Symposium on the Theory of Computing, ACM, New York (1985) pp. 356-363.

Pollard, J.: private communication (1988).

Stern, J.: Secret linear congruential generators are not cryptographically secure. Proceedings of the 28th IEEE-Symposium on Foundations of Computer Science (1987) pp. 421-426.

Stiefel, E.: Einführung in die numerische Mathematik. Teubner, Stuttgart (1969).

Yao, A.C.: Theory and applications of trapdoor functions. Proceedings of the 25th IEEE Symposium on Foundations of Computer Science, IEEE, New York (1982), pp. 80-91.