J. Cryptology (2005) 18: 91–110
DOI: 10.1007/s00145-004-0215-y

Journal of
CRYPTOLOGY



# An Improved Pseudo-Random Generator Based on the Discrete Logarithm Problem*

Rosario Gennaro
IBM T.J.Watson Research Center, P.O. Box 704,
Yorktown Heights, NY 10598, U.S.A.
rosario@watson.ibm.com

Communicated by Matthew Franklin

Received 17 June 2002 and revised 13 July 2003
Online publication 28 September 2004

**Abstract.** Under the assumption that solving the discrete logarithm problem modulo an $n$-bit safe prime $p$ is hard even when the exponent is a small $c$-bit number, we construct a new pseudo-random bit generator. This new generator outputs $n - c - 1$ bits per exponentiation with a $c$-bit exponent and is among the fastest generators based on hard number-theoretic problems.

**Key words.** Pseudorandomness, Discrete logarithm.

## 1. Introduction

Many (if not all) cryptographic algorithms rely on the availability of some form of randomness. However, perfect randomness is a scarce resource. Fortunately, for almost all cryptographic applications, it is sufficient to use pseudo-random bits, i.e. sources of randomness that "look" sufficiently random to the adversary.

This notion can be made more formal. The concept of cryptographically strong pseudo-random bit generators (PRBGs) was introduced in papers by Blum and Micali [5] and Yao [35]. Informally a PRBG is cryptographically strong if it passes all polynomial-time statistical tests or, in other words, if the distribution of sequences output by the generator cannot be distinguished from truly random sequences by any polynomial-time judge.

Blum and Micali [5] presented the first cryptographically strong PRBG under the assumption that modular exponentiation modulo a prime $p$ is a one-way function. This breakthrough result was followed by a series of papers that culminated in [12] where it was shown that secure PRBGs exists if any one-way function does.

* A preliminary version of this paper appeared in the *Proceedings of CRYPTO* 2000 [9]. The main differences between the two versions are summarized in Section 1.3.

To extract a single pseudo-random bit, the Blum–Micali generator requires a full modular exponentiation in $Z_p^*$. This was improved by Long and Wigderson [22] and Peralta [26], who showed that up to $O(\log\log p)$ bits could be extracted by a single iteration (i.e. a modular exponentiation) of the Blum–Micali generator. Håstad et al. [15] show that if one considers discrete-log modulo an $n$-bit composite then almost $n/2$ pseudo-random bits can be extracted per modular exponentiation.

Better efficiency can be gained by looking at the quadratic residuosity problem in $Z_N^*$ where $N$ is a Blum integer (i.e. the product of two primes of identical bitsize and both $\equiv 3 \bmod 4$.) Under this assumption, Blum et al. [4] construct a secure PRBG for which each iteration consists of a single squaring in $Z_N^*$ and outputs a pseudo-random bit. Alexi et al. [2] showed that one can improve this to $O(\log\log N)$ bits and rely only on the intractability of factoring as the underlying assumption. Up to this date, this is the most efficient provably secure PRBG based on number-theoretic assumptions.

In [25] Patel and Sundaram (using ideas already present in [15]) propose a very interesting variation on the Blum–Micali generator. They consider the following variation of the discrete log problem: they assume that it is hard to solve the discrete log problem modulo an $n$-bit prime $p$ even when the exponent is small (say only $c$ bits long with $c < n$); we call this the *Discrete Log with Short Exponents Assumption*. They show then, that it is possible to extract up to $n - c - 1$ bits from one iteration of the Blum–Micali generator. However, the iterated function of the generator itself remains the same, which means that one gets $n - c - 1$ bits per full modular exponentiations. Patel and Sundaram left open the question whether it was possible to modify their generator so that each iteration consisted of an exponentiation with a small $c$-bit exponent. We answer their question in the affirmative.

### 1.1. *Our Contribution*

We show that it is possible to construct a high-rate discrete-log-based secure PRBG. Under the Discrete Log with Short Exponents Assumption considered in [25], we present a generator that outputs $n-c-1$ bits per iteration, which consists of a single exponentiation with a $c$-bit exponent.

The basic idea of the new scheme is to show that if the function $f\colon \{0,1\}^c \longrightarrow Z_p^*$ defined as $f(x) = g^x \bmod p$ is a one-way function, then it also has strong pseudo-randomness properties over $Z_p^*$. In particular, it is possible to think of it as pseudo-random generator itself. By iterating the above function and outputting the appropriate bits, we obtain an efficient PRBG.

Another attractive feature of this generator (which is shared by the Blum–Micali and Patel–Sundaram generators as well) is that all the exponentiations are computed over a fixed basis, and thus precomputation tables can be used to speed them up.

Using typical parameters $n = 1024$ and $c = 160$ we obtain roughly 860 pseudo-random bits per 160-bit exponent exponentiations. Higher parameters must be considered if one takes into account a "concrete security analysis" which includes the security degradation which is brought about by the security proof. Interestingly, the *rate* of our generator (i.e. the number of output bits per modular multiplication) *increases* as the parameters go up. However, the overall efficiency does not since the cost of modular multiplications increases as parameters go up, but, overall, the decrease in efficiency

is much slower than in previously known generators (since the increase in the rate compensates, in part, for the increase in the cost of multiplications). The efficiency of the generator can be greatly increased by using a precomputation scheme such as the one proposed in [21]. See Section 4 for a detailed discussion of these points.

The same technique can be used to speed up the generator proposed in [15], by a factor of roughly 2, under the assumption that factoring is hard.

### 1.2. *Other Related Work*

Micali and Schnorr in [23] consider functions $f\colon \{0,1\}^n \longrightarrow \{0,1\}^n$ with the following property: when $f$ is fed with a random input from $\{0,1\}^c$ (with $c < n$), then the resulting distribution is computationally indistinguishable from the uniform distribution over $\{0,1\}^n$. Then clearly $f$ can be immediately used as a pseudo-random generator: choose $x = x_1 \cdots x_n$ at random in $\{0,1\}^n$, output the top $n-c$ bits and iterate by setting $x := f(x_1 \cdots x_c)$.

Micali and Schnorr do not present any function which can be proven to have this property under some standard cryptographic assumption. Thus they specifically make this assumption about known cryptographic functions (like RSA or discrete log).

The idea behind our generator is similar to [23], but we are able to provably demonstrate that this property of the discrete log function with small exponents follows from the much weaker assumption of its one-wayness.

Håstad and Naslund in [14] consider pseudo-random generators based on symmetric cryptographic primitives (such has block ciphers). The constructions in [14] are thoroughly analyzed and their security can be proven under some pseudo-randomness assumption on the underlying primitives. The constructions are very efficient and clearly beat number-theoretic constructions, such as ours.

Independently from our work, Goldreich and Rosen [10] noticed the improvement by a factor of 2 to the generator proposed in [15].

### 1.3. *Editorial Note*

This paper is a revised version of [9]. In that version no mention was made of an attack on the discrete log problem with short exponents which was discovered by van Oorschot and Wiener in [33]. In order to avoid their attack it is sufficient to restrict the class of prime moduli to *safe* primes (i.e. primes $p$ such that $(p-1)/2$ is also a prime.) This was not specified in [9] and we correct it in this version. See Section 2.3 for the technical details.

The current version also differs in the treatment of the "concrete security analysis" of the generator (Section 4) and in the details of the security proofs.

### 1.4. *Organization*

The paper is organized as follows. In Section 2 we summarize notations, definitions and prior work. Section 3 presents the main contribution of our paper, the new generator and its security proof. In Section 4 we discuss the efficiency of our construction compared with other generators in the literature.

## 2. Preliminaries

In this section we summarize notations, definitions and prior work which is relevant to our result. In the following we denote with $\{0,1\}^n$ the set of $n$-bit strings. If $x \in \{0,1\}^n$ then we write $x = x_n x_{n-1} \cdots x_1$ where each $x_i \in \{0,1\}$. If we think of $x$ as an integer then we have $x = \sum_i x_i 2^{i-1}$ (that is, $x_n$ is the most significant bit). With $R_n$ we denote the uniform distribution over $\{0,1\}^n$.

### 2.1. *Pseudo-Random Number Generators*

Let $X_n$, $Y_n$ be two arbitrary probability ensembles over $\{0,1\}^n$. In the following we denote with $x \leftarrow X_n$ the selection of an element $x$ in $\{0,1\}^n$ according to the distribution $X_n$, and with $Prob_{X_n}[x]$ we denote the probability of such an event.

We say that $X_n$ and $Y_n$ have *statistical distance* bounded by $\Delta(n)$ if the following holds:

$$\sum_{x \in \{0,1\}^n} |Prob_{X_n}[x] - Prob_{Y_n}[x]| \leq \Delta(n).$$

We say that $X_n$ and $Y_n$ are *statistically indistinguishable* if for every polynomial $P(\cdot)$ and for sufficiently large $n$ we have that

$$\Delta(n) \leq \frac{1}{P(n)}.$$

We say that $X_n$ and $Y_n$ are *computationally indistinguishable* (a concept introduced in [11]) if any polynomial-time machine cannot distinguish between samples drawn according to $X_n$ or according to $Y_n$. More formally:

**Definition 1.** Let $X_n$, $Y_n$ be two families of probability distributions over $\{0,1\}^n$. Given a Turing machine $\mathcal{D}$ consider the following quantities:

$$\begin{aligned} \delta_{\mathcal{D},X_n} &= Prob[x \leftarrow X_n;\ \mathcal{D}(x) = 1], \\ \delta_{\mathcal{D},Y_n} &= Prob[y \leftarrow Y_n;\ \mathcal{D}(y) = 1]. \end{aligned}$$

We say that $X_n$ and $Y_n$ are *computationally indistinguishable* if for every probabilistic polynomial time $\mathcal{D}$, for every polynomial $P(\cdot)$ and for sufficiently large $n$ we have that

$$|\delta_{\mathcal{D},X_n} - \delta_{\mathcal{D},Y_n}| \leq \frac{1}{P(n)}.$$

We now move on to define pseudo-random number generators [5], [35]. There are several equivalent definitions, but the following one is sufficient for our purposes. Consider a family of functions

$$\mathsf{G}_n\colon \{0,1\}^{k_n} \longrightarrow \{0,1\}^n,$$

where $k_n < n$. $\mathsf{G}_n$ induces a family of probability distributions (which we denote with $G_n$) over $\{0,1\}^n$ as follows:

$$Prob_{G_n}[y] = Prob[y = \mathsf{G}_n(s);\ s \leftarrow R_{k_n}].$$

**Definition 2.** We say that $\mathsf{G}_n$ is a *cryptographically strong* (*or secure*) *PRBG* if the function $\mathsf{G}_n$ can be computed in polynomial time and the two families of probability distributions $R_n$ and $G_n$ are computationally indistinguishable.

The input of a pseudo-random generator is usually called the *seed*.

### 2.2. *Pseudo-Randomness over Arbitrary Sets*

Let $A_n$ be a family of sets such that for each $n$ we have $2^{n-1} \leq |A_n| < 2^n$ (i.e. we need $n$ bits to describe elements of $A_n$). We denote with $U_n$ the uniform distribution over $A_n$. Also let $k_n$ be a sequence of numbers such that for each $n$, $k_n < n$. Consider a family of functions

$$\mathsf{AG}_n\colon \{0,1\}^{k_n} \longrightarrow A_n.$$

$\mathsf{AG}_n$ induces a family of probability distributions (which we denote with $AG_n$) over $A_n$ as follows:

$$Prob_{AG_n}[y] = Prob[y = \mathsf{AG}_n(s);\ s \leftarrow R_{k_n}].$$

**Definition 3.** We say that $\mathsf{AG}_n$ is a *cryptographically strong* (*or secure*) *pseudo-random generator* over $A_n$ if the function $\mathsf{AG}_n$ can be computed in polynomial time and the two families of probability distributions $U_n$ and $AG_n$ are computationally indistinguishable.

A secure pseudo-random generator over $A_n$ is already useful for applications in which one needs pseudo-random elements of that domain. Indeed, no adversary will be able to distinguish if $y \in A_n$ was truly sampled at random or if it was computed as $\mathsf{AG}_n(s)$ starting from a much shorter seed $s$. An example of this is to consider $A_n$ to be $Z_p^*$ for an $n$-bit prime number $p$. If our application requires pseudo-random elements of $Z_p^*$ then such a generator would be sufficient.

However, as *bit* generators it may not be secure, since if we look at the bits of an encoding of the elements of $A_n$, then their distribution may be biased. This, however, is not going to be a problem for us since we use pseudo-random generators over arbitrary sets as a tool in the proof of our main pseudo-random *bit* generator.

### 2.3. *The Discrete Logarithm Problem*

Let $p$ be a prime. We denote with $n$ the binary length of $p$. It is well known that $Z_p^* = \{x : 1 \leq x \leq p-1\}$ is a cyclic group under multiplication mod$p$. Let $g$ be a generator of $Z_p^*$. Thus the function

$$f\colon Z_{p-1} \longrightarrow Z_p^*,$$
$$f(x) = g^x \bmod p$$

is a permutation. The inverse of $f$ (called the *discrete logarithm* function) is conjectured to be a function hard to compute (the cryptographic relevance of this conjecture first appears in the seminal paper by Diffie and Hellman [7] on public-key cryptography). In spite of extensive research into this problem, the best known algorithms for its solution still run in time sub-exponential in $n$ (see [1] and [19]).

In some applications (like the one we describe in this paper) it is important to speed up the computation of the function $f(x) = g^x$. One possible way to do this is to restrict its input to small values of $x$. Let $c$ be an integer which we can think of as depending on $n$ ($c = c(n)$). Assume now that we are given $y = g^x \bmod p$ with $x \leq 2^c$. It appears to be reasonable to assume that computing the discrete logarithm of $y$ is still hard even if we know that $x \leq 2^c$. Indeed, the running time of the sub-exponential algorithms mentioned before depends only on the size $n$ of the whole group. Depending on the size of $c$, different methods may actually be more efficient. Indeed, the so-called *baby-step giant-step* algorithm by Shanks [18] or the *rho* and *lambda* algorithms by Pollard [28] (as improved in [34] and [29]) can compute the discrete log of $y$ in $O(2^{c/2})$ time. If one restricts the field to *generic* algorithms (i.e. algorithms that can only perform group operations and cannot take advantage of specific properties of the encoding of group elements) then it can be proven that this is the best that can be done (see [31] and [30]).

If the complete factorization of $p - 1$ is known, then the running time of these algorithms can be improved by using the Pohlig–Hellman decomposition [27]. This is done by reducing the original discrete log problem, into several "smaller" problems (one for each distinct prime factor in $p - 1$).

Van Oorschot and Wiener in [33] present a new method of combining the Pollard *lambda* method with a partial Pohlig–Hellman decomposition. Their end result is that for *random* primes, using short exponents is *not* secure. However, one of the stated ways to avoid their attack is to restrict the moduli to be *safe* primes $p$ (i.e. such that $(p - 1)/2$ is also a prime) since in this case the Polhig–Hellman decomposition is useless.

Define with $\omega(\cdot)$ any function (defined over the integers) that grows faster than the logarithmic function (i.e. for any constant $\gamma$ there exists an integer $n_\gamma$ such that $\omega(n) > \gamma \log n$ for $n > n_\gamma$). Thus if we set $c = \omega(\log n)$, there are no known polynomial-time algorithms that can compute the discrete log of $y = g^x \bmod p$ when $x$ is chosen at random in $[0..2^c]$ and $p$ is a safe prime. In [25] it is explicitly assumed that *no* such efficient algorithm can exist. This is called the *Discrete Logarithm with Short c-Bit Exponents* (*c-DLSE*) Assumption and we adopt it as the basis of our results as well.

**Assumption 1** ($c$-DLSE [25]). *Let SPRIMES($n$) be the set of $n$-bit safe primes and let $c$ be a quantity that grows faster than* $\log n$ (*i.e.* $c = \omega(\log n)$). *For every probabilistic polynomial-time Turing machine $\mathcal{I}$, for every polynomial $P(\cdot)$ and for sufficiently large $n$ we have that*

$$Prob\begin{bmatrix} p \leftarrow SPRIMES(n); \\ x \leftarrow R_c; \\ \mathcal{I}(p, g, g^x, c) = x \end{bmatrix} \leq \frac{1}{P(n)}.$$

Informally the assumption states that for any polynomial-time machine $\mathcal{I}$ (for inverter) that runs on input a safe prime $p$, and the value $g^x$ with $x$ randomly chosen in $[0..2^c]$, the probability that $\mathcal{I}$ solves the discrete log problem (i.e. output $x$) is negligible (can be made smaller than the inverse of any polynomial by choosing a large enough security parameter $n$).

In practice, given today's computing power and discrete-log computing algorithms, it seems to be sufficient to set $n = 1024$ and $c = 160$. This implies a "security level" of $2^{80}$ (intended as work needed in order to "break" 160-DLSE). See [20] for a way to

estimate the size of these parameters, based on the current state of knowledge on the discrete log problem, and the current computing environment.[1]

*Remark.* The $c$-DLSE assumption is somewhat non-standard and should be considered with care. Apart from the attacks mentioned above, the discrete log with small exponent can be attacked with algorithms that try to exploit the low Hamming weight of the exponent (see [32]). However, in our scenario, the asymptotic complexity of this attack is higher than $2^{c/2}$.

On the other hand, there is evidence supporting the $c$-DLSE Assumption. Besides the lower bounds on generic algorithms mentioned above, there are some results on the inapproximability of the discrete log when restricted to a small set (see [6]).

### 2.4. *Hard Bits for Discrete Logarithm*

The discussion in the previous section basically states that the function $f(x) = g^x \bmod p$ is widely considered to be one-way (i.e. a function easy to compute but not to invert). We strengthened this assumption, to claim that $f$ remains one-way even when $x$ is chosen in a restricted domain. In this section we recall the basic notions of hard-core bits for a one-way function, and how this notion is used to construct PRBGs.

It is well known that even if $f$ is a one-way function, it does not hide all information about its preimages. For the specific case of the discrete logarithm, it is well known that given $y = g^x \bmod p$ it is easy to guess the least significant bit of $x \in Z_{p-1}$ by testing to see if $y$ is a quadratic residue or not in $Z_p^*$ (there is a polynomial-time test to determine that, see [3]).

A Boolean predicate $\Pi$ is said to be *hard* (or *hard-core*) for a one-way function $f$ if any algorithm $\mathcal{A}$ that, given $y = f(x)$, guesses $\Pi(x)$ with probability substantially better than $1/2$, can be used to build another algorithm $\mathcal{A}'$ that on input $y$ computes $x$ with non-negligible probability.

Blum and Micali in [5] prove that the predicate

$$\Pi\colon Z_{p-1} \longrightarrow \{0, 1\},$$
$$\Pi(x) = \left[ x \leq \frac{p-1}{2} \right]$$

is hard for the discrete logarithm function. For the case of safe primes Schnorr in [30] proves that every bit (except the least significant one) of the binary representation of $x$ is hard for the discrete log function. For the case of general primes this result was extended by Håstad and Näslund [13] who proved that every bit of $x$ is hard, except the $s$ least significant ones, where $s$ is the maximum integer such that $2^s$ divides $p - 1$.

The above results talk about *individual* hardness of each bit. We say that a collection of $k$ bits $x_{i_1}, \ldots, x_{i_k}$ in the the binary representation of $x$ is *simultaneously* hard-core if the whole collection of bits looks "random" to a polynomial-time judge who is given the value $g^x$. Even if a collection of $k$ bits $x_{i_1}, \ldots, x_{i_k}$ in the binary representation of $x$ are individually hard-core, it does not guarantee that the whole collection "looks random".

[1] Although the above parameters may be sufficient today for the security of the $c$-DLSE Assumption, they may not be sufficient for the security of the PRBG we propose, if one takes into account a concrete security analysis. See Section 4 for details.

A way to formalize the concept of simultaneous hardness is (following [35]) to say that it is not possible to guess the value of the $j$th predicate even after seeing $g^x$ and the value of the previous $j-1$ predicates over $x$. Formally, we say that $k$ predicates $\Pi_1, \ldots, \Pi_k$,

$$\Pi_i \colon Z_{p-1} \longrightarrow \{0,1\} \qquad \text{for} \quad i = 1, \ldots, k,$$

are simultaneously hard-core for the discrete-log function if any probabilistic polynomial-time algorithm $\mathcal{A}$ such that

$$Prob[x \leftarrow Z_{p-1};\ \mathcal{A}(g^x, \Pi_1(x), \ldots, \Pi_{j-1}(x)) = \Pi_j(x)] \geq \frac{1}{2} + \frac{1}{P(n)}$$

for a polynomial $P(\cdot)$, can be used to construct another probabilistic polynomial-time algorithm $\mathcal{A}'$ which on input $g^x$ computes $x$ with non-negligible probability.

In terms of *simultaneous* security of several bits, Long and Wigderson [22] and Peralta [26] showed that there are $O(\log\log p)$ predicates which are *simultaneously hard* for discrete log.

### 2.5. *The Patel–Sundaram Generator*

Let $p$ be an $n$-bit prime such that $p \equiv 3 \bmod 4$ and let $g$ be a generator of $Z_p^*$. Denote with $c$ a quantity that grows faster than $\log n$, i.e. $c = \omega(\log n)$.

In [25] Patel and Sundaram prove that under the $c$-DLSE Assumption the bits $x_2, x_3, \ldots, x_{n-c}$ are simultaneously hard for the function $f(x) = g^x \bmod p$. More formally:

**Theorem 1** [25]. *For sufficiently large $n$, if $p$ is an $n$-bit safe prime and if the $c$-DLSE Assumption holds, then for every $j$, $2 \leq j \leq n-c$, for every polynomial-time Turing machine $\mathcal{A}$, for every polynomial $P(\cdot)$ and for sufficiently large $n$ we have that*

$$|Prob[x \leftarrow Z_{p-1};\ \mathcal{A}(g^x, x_2, \ldots, x_{j-1}) = x_j] - \tfrac{1}{2}| \leq \frac{1}{P(n)}.$$

For sake of completeness we present a proof of this theorem in the Appendix.

Theorem 1 immediately yields a secure PRBG. Start with $x^{(0)} \in_R Z_{p-1}$. Set $x^{(i)} = g^{x^{(i-1)}} \bmod p$. Set also $r^{(i)} = x_2^{(i)}, x_3^{(i)}, \ldots, x_{n-c}^{(i)}$. The output of the generator will be $r^{(0)}, r^{(1)}, \ldots, r^{(k)}$ where $k$ is the number of iterations.

Notice that this generator outputs $n-c-1$ pseudo-random bits at the cost of a modular exponentiation with a random $n$-bit exponent.

## 3. Our New Generator

We now show that under the $c$-DLSE Assumption it is possible to construct a PRBG which is much faster than the Patel–Sundaram one. In order to do this we first revisit the construction of Patel and Sundaram to show how one can obtain a pseudo-random generator over $Z_p^* \times \{0,1\}^{n-c-1}$.

Then we construct a function from $Z_{p-1}$ to $Z_p^*$ which induces a pseudo-random distribution over $Z_p^*$. The proof of this fact is by reduction to the security of the modified

Patel–Sundaram generator. This function is not a generator yet, since it does not stretch its input.

We finally show how to obtain a pseudo-random *bit* generator, by iterating the above function and outputting the appropriate bits.

### 3.1. *The Patel–Sundaram Generator Revisited*

As usual let $p$ be an $n$-bit safe prime and let $c = \omega(\log n)$. Consider the following function (which we call PSG for Patel-Sundaramam Generator):

$$\mathsf{PSG}_{n,c}\colon Z_{p-1} \longrightarrow Z_p^* \times \{0,1\}^{n-c-1},$$
$$\mathsf{PSG}_{n,c}(x) = (g^x \bmod p, x_2, \ldots, x_{n-c}).$$

That is, on input a random seed $x \in Z_{p-1}$, the generator outputs $g^x$ and $n-c-1$ consecutive bits of $x$, starting from the second least significant.

An immediate consequence of the result in [25], is that under the $c$-DLSE assumption $\mathsf{PSG}_{n,c}$ is a secure pseudo-random generator over the set $Z_p^* \times \{0,1\}^{n-c-1}$. More formally, if $U_n$ is the uniform distribution over $Z_p^*$, then the distribution induced by $\mathsf{PSG}_{n,c}$ over $Z_p^* \times \{0,1\}^{n-c-1}$ is computationally indistinguishable from the distribution $U_n \times R_{n-c-1}$.

In other words, for any probabilistic polynomial-time Turing machine $\mathcal{D}$, we can define

$$\delta_{\mathcal{D},UR_n} = Prob[y \leftarrow Z_p^*;\ r \leftarrow R_{n-c-1};\ \mathcal{D}(y,r) = 1],$$
$$\delta_{\mathcal{D},PSG_{n,c}} = Prob[x \leftarrow Z_{p-1};\ \mathcal{D}(\mathsf{PSG}_{n,c}(x)) = 1],$$

then for any polynomial $P(\cdot)$ and for sufficiently large $n$, we have that

$$|\delta_{\mathcal{D},UR_n} - \delta_{\mathcal{D},PSG_{n,c}}| \leq \frac{1}{P(n)}.$$

In the next section we show our new generator and we prove that if it is not secure then we can show the existence of a distinguisher $\mathcal{D}$ that contradicts the above.

### 3.2. *A Preliminary Lemma*

We also assume that $p$ is an $n$-bit safe prime and $c = \omega(\log n)$. Let $g$ be a generator of $Z_p^*$ and denote $\hat{g} = g^{2^{n-c}} \bmod p$. Recall that if $s$ is an $n$-bit integer we denote with $s_i$ the $i$th bit in its binary representation, i.e. $s = \sum_i^n s_i 2^{i-1}$. With div we denote the integer division function, thus $s$ div $2^k$ is going to be the $(n-k)$-bit integer $s_n \cdots s_{k+1}$, i.e. $s$ div $2^k = \sum_{i=1}^{n-k} s_{k+i} 2^{i-1}$.

The function we consider is the following:

$$\mathsf{RG}_{n,c}\colon Z_{p-1} \longrightarrow Z_p^*,$$
$$\mathsf{RG}_{n,c}(s) = \hat{g}^{(s \text{ div } 2^{n-c})} g^{s_1} \bmod p.$$

That is we consider modular exponentiation in $Z_p^*$ with base $g$, but only after zeroing the bits in positions $2, \ldots, n-c$ of the input $s$ (these bits are basically ignored). In other

words, writing the exponent in binary form we have[2]

$$\mathsf{RG}_{n,c}(s) = g^{s_n s_{n-1} \cdots s_{n-c+1} 0 \cdots 01}.$$

We denote with $RG_{n,c}$ the following probability distribution over $Z_p^*$ (which is induced by the function $\mathsf{RG}$ in the usual way):

$$Prob_{RG_{n,c}}[y] = Prob[y = \mathsf{RG}_{n,c}(s);\ s \leftarrow Z_{p-1}].$$

The following lemma states that the distribution $RG_{n,c}$ is computationally indistinguishable from the uniform distribution over $Z_p^*$ if the $c$-DLSE assumption holds.

**Lemma 1.** *Let $p$ be an $n$-bit safe prime and let $U_n$ be the uniform distribution over $Z_p^*$. If the $c$-DLSE Assumption holds, then the two distributions $U_n$ and $RG_{n,c}$ are computationally indistinguishable* (*see Definition* 1).

The proof of the lemma goes by contradiction. We show that if $RG_{n,c}$ can be distinguished from $U_n$, then the Patel–Sundaram generator $\mathsf{PSG}$ is not secure. We do this by showing that any efficient distinguisher between $RG_{n,c}$ and the uniform distribution over $Z_p^*$ can be transformed into a distinguisher for $\mathsf{PSG}_{n,c}$. This contradicts Theorem 1 and ultimately the $c$-DLSE Assumption.

**Proof.** Assume for the sake of contradiction that there exists a distinguisher $\mathcal{D}$ and a polynomial $P(\cdot)$ such that for infinitely many $n$'s we have that

$$\delta_{\mathcal{D},U_n} - \delta_{\mathcal{D},RG_{n,c}} \geq \frac{1}{P(n)},$$

where

$$\begin{aligned} \delta_{\mathcal{D},U_n} &= Prob[x \leftarrow Z_p^*;\ \mathcal{D}(p,g,x,c) = 1], \\ \delta_{\mathcal{D},RG_{n,c}} &= Prob[s \leftarrow Z_{p-1};\ \mathcal{D}(p,g,\mathsf{RG}_{n,c}(s),c) = 1]. \end{aligned}$$

We show how to construct a distinguisher $\hat{\mathcal{D}}$ that "breaks" $\mathsf{PSG}$.

In order to break $\mathsf{PSG}_{n,c}$ we are given as input $(p, g, y, r, c)$ with $y \in Z_p^*$ and $r \in \{0,1\}^{n-c-1}$ and we want to guess if it comes from the distribution $U_n \times R_{n-c-1}$ or from the distribution $PSG_{n,c}$ of outputs of the generator $\mathsf{PSG}_{n,c}$. The distinguisher $\hat{\mathcal{D}}$ will follow this algorithm:

1. Consider the integer $z := r \circ 0$ where $\circ$ means concatenation. Set $w := yg^{-z} \bmod p$;
2. Output $\mathcal{D}(p, g, w, c)$.

Why does this work? Assume that $(y, r)$ was drawn according to $\mathsf{PSG}_{n,c}(x)$ for some random $x \in Z_{p-1}$. Then $w = g^u$ where $u = 2^{n-c}(x \text{ div } 2^{n-c}) + x_1 \bmod p - 1$. That

[2] This is why we use the value $\hat{g}$ as the basis in the first definition of $\mathsf{RG}$, otherwise the bits $s_n \cdots s_{n-c+1}$ would be placed in positions 1 to $c$.

is, the discrete log of $w$ in base $g$ has the $n-c-1$ bits in position $2, \ldots, n-c$ equal to 0 (this is because $r$ is identical to those $n-c-1$ bits of the discrete log of $y$ by the assumption that $(y, r)$ follows the $PSG_{n,c}$ distribution). Thus once we set $\hat{g} = g^{2^{n-c}}$ we get $w = \hat{g}^{x \text{ div } 2^{n-c}} g^{x_1} \bmod p$, i.e. $w = \mathsf{RG}_{n,c}(x)$. Thus if $(y, r)$ is drawn according to $PSG_n$ then $w$ follows the same distribution as $RG_n$.

On the other hand, if $(y, r)$ was drawn with $y$ randomly chosen in $Z_p^*$ and $r$ randomly chosen in $\{0,1\}^{n-c-1}$, then all we know is that $w$ is a random element of $Z_p^*$.

Thus $\hat{\mathcal{D}}$ will guess the correct distribution with the same advantage as $\mathcal{D}$ does. Which contradicts the security of the PSG generator. □

### 3.3. *The New Generator*

It is now straightforward to construct the new generator. The algorithm receives as a seed a random element $s$ in $Z_{p-1}$ and then it iterates the function RG on it. The pseudo-random bits outputted by the generator are the bits ignored by the function RG. The output of the function RG will serve as the new input for the next iteration.

In more detail, the algorithm $\mathsf{IRG}_{n,c}$ (for Iterated-RG generator) works as follows. Start with $x^{(0)} \in_R Z_{p-1}$. Set $x^{(i)} = \mathsf{RG}_{n,c}(x^{(i-1)})$. Set also $r^{(i)} = x_2^{(i)}, x_3^{(i)}, \ldots, x_{n-c}^{(i)}$. The output of the generator will be $r^{(0)}, r^{(1)}, \ldots, r^{(k-1)}$ where $k$ is the number of iterations (chosen such that $k = poly(n)$ and $k(n-c-1) > n$).

Notice that this generator outputs $n-c-1$ pseudo-random bits at the cost of a modular exponentiation with a random $c$-bit exponent (i.e. the cost of the computation of the function RG).

**Theorem 2.** *Under the c-DLSE Assumption,* $\mathsf{IRG}_{n,c}$ *is a secure PRBG* (*see Definition* 2).

**Proof.** We first notice that, for sufficiently large $n$, $r^{(0)}$ is an almost uniformly distributed $(n-c-1)$-bit string. This is because $r^{(0)}$ is composed of the bits in positions $2, 3, \ldots, n-c$ of a random element of $Z_{p-1}$ and thus it is possible to bound the statistical distance between the distribution of $r^{(0)}$ and the uniform distribution over $\{0,1\}^{n-c-1}$ with $2^{1-c}$. This can be easily seen as follows: write $p-1 = A2^{n-c} + B$ where $A$ (resp. $B$) is the integer represented by the top $c$ (resp. bottom $n-c$) bits of $p-1$.

A specific string $r^{(0)}$ appears with probability $2A/(p-1)$. Indeed, if you consider all of the elements of $Z_{p-1}$ and partition them in $2^{n-c-1}$ sets based on the values of the bits in positions $2, 3, \ldots, n-c$ then each set will have exactly $2A$ elements (the factor of 2 comes from the last bit). On the other hand, a uniformly chosen $(n-c-1)$-bit string will appear with probability $2^{-(n-c-1)}$. Thus the statistical distance $\Delta$ between the two distributions can be expressed as

$$\Delta = \Sigma \left| \frac{1}{2^{n-c-1}} - \frac{2A}{p-1} \right| = 1 - \frac{A2^{n-c}}{A2^{n-c} + B} = \frac{B}{A2^{n-c} + B}.$$

Now recall that $2^{c-1} < A < 2^c$ and $2^{n-c-1} < B < 2^{n-c}$, thus we can bound $\Delta$ as follows:

$$\Delta < \frac{2^{n-c}}{2^{n-c}2^{c-1} + 2n - c - 1} = \frac{1}{2^{c-1} + 2^{-1}} < 2^{1-c}.$$

Now by virtue of Lemma 1 we know that all the values $x^{(i)}$ follow a distribution which is computationally indistinguishable from the uniform one on $Z_p^*$. By the same argument as above it follows that all the $r^{(i)}$ must follow a distribution which is computationally indistinguishable from $R_{n-c-1}$.

The proof follows a hybrid argument, which is a standard proof method that can be explained as follows. If there is a distinguisher $\mathcal{D}$ between the distribution induced by $\mathsf{IRG}_{n,c}$ and the distribution $R_{k(n-c-1)}$, then we can construct a distinguisher $\mathcal{D}_1$ that for a specific index $i$ distinguish between the distribution followed by $r^{(i)}$ and the uniform distribution $R_{n-c-1}$. Now that implies that it is possible to distinguish the distribution followed by $x^{(i)}$ and the uniform distribution over $Z_p^*$ This contradicts Lemma 1 and ultimately the $c$-DLSE Assumption. The distinguisher $\mathcal{D}_1$ is built by analyzing the behavior of the distinguisher $\mathcal{D}$ over *hybrid* distributions that are somewhat "in between" $\mathsf{IRG}_{n,c}$ and $R_{k(n-c-1)}$: i.e. up to a specific index $i$ they follow $R_{k(n-c-1)}$, but from index $i+1$ they behave like $\mathsf{IRG}_{n,c}$.

We now work out the technical details more formally. For each $i = 0, \ldots, k$, consider the hybrid distribution $H_i$ over $\{0,1\}^{k(n-c-1)}$ defined by the following experiment: (1) choose $x \in_R Z_{p-1}$; (2) Run $\mathsf{IRG}_{n,c}(x)$ but only for $k-i$ iterations and let $r^{(i)}, r^{(i+1)}, \ldots, r^{(k-1)}$ be the output; (3) for each $j = 0, \ldots, i-1$ set $r^{(j)}$ as a random $(n-c-1)$-bit string. Clearly, $H_0$ is the distribution induced by $\mathsf{IRG}_{n,c}$. On the other hand, $H_k$ is the uniform distribution $R_{k(n-c-1)}$.

Let $\mathcal{D}$ be a distinguisher between the distribution induced by $\mathsf{IRG}_{n,c}$ and the distribution $R_{k(n-c-1)}$ which succeeds with non-negligible probability $\pi$. Denote

$$\pi_i = Prob[x \leftarrow H_i; \mathcal{D}(x) = 1].$$

Then we have that $|\pi_0 - \pi_k| \geq \pi$. However, that means that there must exist one index $i = 1, \ldots, k$ such that

$$|\pi_{i-1} - \pi_i| \geq \frac{\pi}{k},$$

which is still non-negligible.

Now we describe the distinguisher $\mathcal{D}_1$ which contradicts Lemma 1. $\mathcal{D}_1$ runs on input a value $x \in Z_p^*$ and it attempts to determine if $x$ follows the uniform distribution $U_n$ or the distribution $RG_{n,c}$ over $Z_p^*$. The distinguisher $\mathcal{D}_1$ first runs $\mathsf{IRG}_{n,c}(x)$ but only for $k-i$ iterations and let $r^{(i)}, r^{(i+1)}, \ldots, r^{(k-1)}$ be the output. Then for each $j = 0, \ldots, i-1$ set $r^{(j)}$ as a random $(n-c-1)$-bit string. It then runs $\mathcal{D}$ on the resulting string $\vec{r} = r^{(0)}, r^{(1)}, \ldots, r^{(k-1)}$ and outputs whatever $\mathcal{D}$ outputs.

Consider the string $\vec{r} = r^{(0)}, r^{(1)}, \ldots, r^{(k-1)}$ defined above. If $x$ was uniformly chosen in $Z_p^*$ then clearly this experiment is identical to the one defining the distribution $H_i$, thus $\vec{r}$ follows the distribution $H_i$. On the other hand, if $x$ was selected according to the distribution $RG_{n,c}$ it is not hard to see that $\vec{r}$ follows the distribution $H_{i-1}$. Thus $\mathcal{D}_1$ distinguishes between the two distributions with non-negligible advantage $\pi/k$. □

### 3.4. *A Comment on the HSS Generator*

In [15] Håstad et al. show that under the assumption that factoring RSA moduli is hard, the modular exponentiation function simultaneously hides half of its input bits.

In other words, let $N = pq$ be the product of two large primes and let $g \in Z_N^*$ be an element of sufficiently high order $O_g$ (see [15] for details). Then the function

$$f\colon [1..O_g] \longrightarrow Z_N^*,$$
$$f(x) = g^x \bmod N$$

is one-way under the assumption that factoring is hard. The paper [15] shows that either the top or the bottom half of the bits of $x$ are simultaneously hard for this function. Let $n = |N|$.

It is suggested in [15] to use the above results to construct the following pseudo-random generator. Start from a random $x \in [1..O_g]$. To simplify the notation denote the binary expansion of $x$ as $X_{\mathrm{T}} \circ x_{\mathrm{B}}$, where $X_{\mathrm{T}}$ are the top half of the bits, and $x_{\mathrm{B}}$ the bottom half. The output of the generator is the value $(f(x) = g^x, x_{\mathrm{B}})$. By the above result, this is pseudo-random over the set $G \times \{0, 1\}^{|O_g/2|}$ (where $G$ is the group generated by $g$). This can be transformed into a PRBG by applying a universal hash function to this value. Thus with one full exponentiation the above generator produces roughly $n/2$ extra pseudorandom bits.

It is not hard to see that an argument similar to the one of Lemma 1 allows us to state that the function $f$, when computed over inputs in $[1..O_g]$ randomly chosen but with the bottom half bits equal to zero, induces a pseudorandom distribution over $G$. As in Lemma 1 the basic idea of the proof is to show that if $(g^x, x_{\mathrm{B}})$ is pseudo-random over $G \times \{0, 1\}^{|O_g/2|}$, then

$$\frac{g^x}{g^{x_{\mathrm{B}}}} = g^{x - x_{\mathrm{B}}} = g^{x_T \circ \vec{0}}$$

must be pseudo-random over the group $G$.

Thus we can get roughly $n/2$ extra pseudo-random bits by computing a modular exponentiation with an exponent which is only half the one used by [15]. The net result is an improvement by a factor of 2 in the speed of the generator proposed in [15].

A similar result was independently achieved in [10].

## 4. Efficiency Analysis

Our new generator is very efficient. It outputs $n - c - 1$ pseudo-random bits at the cost of a modular exponentiation with a random $c$-bit exponent, or roughly $1.5c$ modular multiplications in $Z_p^*$. Compare this with the Patel–Sundaram generator where the same number of pseudo-random bits would cost $1.5n$ modular multiplications. Moreover, the security of our scheme is tightly related to the security of the Patel–Sundaram one, since the reduction from our scheme to theirs is quite immediate.

So far we have discussed security in asymptotic terms. If we want to instantiate practical parameters we need to analyze more closely the *concrete* security of the proposed scheme.

A close look at the proof of security in [25] shows the following. If we assume that Theorem 1 fails, i.e. that for some $j$, $2 \le j \le n - c$, there exists an algorithm $\mathcal{A}$ which runs in time $T(n)$, and a polynomial $P(\cdot)$ such that without loss of generality

$$Prob[x \leftarrow Z_{p-1};\ \mathcal{A}(g^x, x_2, \ldots, x_{j-1}) = x_j] > \frac{1}{2} + \frac{1}{P(n)},$$

then we have an algorithm $\mathcal{I}^{\mathcal{A}}$ to break $c$-DLSE which runs in time $O((n-c)cP^2(n)T(n))$ if $2 \leq j < n - c - \log P(n)$ and in time $O((n-c)cP^3(n)T(n))$ if $n - c - \log P(n) \leq j \leq n - c$ (the hidden constant is very small).

The security of our PBRG is by reduction to Theorem 1. The reduction is very tight, i.e. does not add to the complexity, except for the hybrid argument in Theorem 2 which introduces a factor of $k$ in the reduction. Basically if we denote with $\ell$ the total number of bits output by the generator (say before reseeding it), then $k = \ell/(n-c)$ and thus we can upper bound the complexity of the reduction with the term $O(\ell cP^3(n)T(n))$.

In order to be able to say that our PRBG is secure we need to make sure that this complexity is smaller than the time to break $c$-DLSE with the best known algorithm (which we know today is $2^{c/2}$).

*The BBS Generator.* The BBS generator was introduced by Blum et al. in [4] under the assumption that deciding quadratic residuosity modulo a composite is hard. The generator works by repeatedly squaring mod$N$ a random seed in $Z_N^*$ where $N$ is a Blum integer ($N = PQ$ with $P$, $Q$ both primes of identical size and $\equiv 3 \bmod 4$.) At each iteration it outputs the least significant bit of the current value. The rate of this generator is thus of 1 bit/squaring. In [2] Alexi et al. showed that one can output up to $k = O(\log \log N)$ bits per iteration of the squaring generator (and this while also relaxing the underlying assumption to the hardness of factoring). The actual number $k$ of bits that can be outputted depends on the concrete parameters adopted.

The reduction in [2] is not very tight and was recently improved by Fischlin and Schnorr in [8]. The complexity of the reduction quoted there is

$$O(n \log n P^2(n)T(n) + n^2 P^4(n) \log n)$$

(here $P(n)$, $T(n)$ refers to a machine which guesses the next bit in one iteration of the BBS generator in time $T(n)$ and with advantage $1/P(n)$).

If we want to output $m$ bits per iteration, the complexity grows by a factor of $2^{2m}$ and the reduction quickly becomes more expensive than known factoring algorithms. We do not consider this variation, since this increase in the reduction will force us to choose a larger security parameter $n$. This in turn will make the multiplications more expensive, so the factor of $m$ saved in the number of multiplications, will be offset by the larger individual cost of each multiplication. Notice instead that the reduction in [25] (and thus in our PRBG) depends only linearly on the number of bits outputted.

Again, if $\ell$ is the total number of bits output, then we must introduce a hybrid argument on the number $k = \ell/m$ of iterations, which increases the complexity by a factor of $k$. Thus we can upper bound the complexity of the BBS reduction with the term

$$O\left(\ell \frac{2^{2k}}{k}(n \log n P^2(n)T(n) + n^2 P^4(n) \log n)\right).$$

*How to Compare.* We are going to fix some bounds on the security we are willing to tolerate, in particular, on the time $T$ which we assume any adversary will run for, and the advantage $1/P$ with which he can correctly guess. We are also going to fix a bound on $\ell$ the number of bits output by the generator. For both our generator and for the BBS generator this bounds will yield some concrete time bounds on the complexity of the

reduction. Using known estimates on the complexity of the discrete log and factoring problem we will estimate the security parameters (i.e. the values $n$ and $c$) which make the reductions meaningful (i.e. for which the complexity of the reduction is inferior to the complexity of the best known factoring/dlog algorithm). At this point we will be able to make concrete statements on the comparative efficiency of the two generators.

*Concrete Parameters*. Here is how we fix the parameters. We set, following [8], $\ell = 10^7$ and $P = 10^2$ (i.e. we bound the adversary's probability by $1/100$). Also, following [20], we define $T = 3.5 \cdot 10^{10}$ MIPS/year, the upper bound on the running time of the adversary (this value of $T$ is the estimation given in [20] for the amount of unfeasible computation in the year 2003).

In order to make the computation simple, we approximate $c$ with the value $c = 3 \cdot 10^2$, and $n = 10^3$. Notice that this somewhat increases the complexity of the reduction of our generator, while it underestimates the complexity of the reduction of the BBS generator (thus the end result will eventually underestimate the improvement of our generator compared with BBS).

With the above parameters, the complexity of both reductions will be $\approx 10^{26}$ MIPS/year. We now extrapolate from [20] the values $n = 3000$ and $c = 225$: for these values the fastest algorithms for (i) factoring, (ii) computing general discrete log and (iii) computing discrete logs with short exponents all have complexity larger than $10^{26}$ in 2003.

So now we can do a comparison. The BBS generator will output one bit per modular squaring; considering that squaring is twice as fast than modular multiplication, we have that the BBS rate is of 2 bits per modular multiplication. Our PRBG instead outputs $3000 - 225 = 2775$ bits per modular exponentiation with an exponent of 225 bits, which on average will cost around 350 multiplications. This yields a rate of about 7 bits per multiplication, which is about 3.5 times faster than the BBS generator.

### 4.1. *Using Precomputed Tables*

The most expensive part of the computation of our generator is to compute $\hat{g}^s \bmod p$ where $s$ is a $c$-bit value.

We can take advantage of the fact that in our generator[3] the modular exponentiations are all computed over the same basis $\hat{g}$. This feature allows us to precompute powers of $\hat{g}$ and store them in a table, and then use these values to compute $\hat{g}^s$ quickly for any $s$.

The simplest approach is to precompute a table $T$:

$$T = \{\hat{g}^{2^i} \bmod p;\ i = 0, \ldots, c\}.$$

Now, one exponentiation with base $\hat{g}$ and a random $c$-bit exponent can be computed using only $0.5c$ multiplications on average. The cost is an increase to $O(cn)$ bits of required memory.

With this simple improvement one iteration of our generator will require roughly 113 multiplications, which yields a rate of more that 24 pseudo-random bits per multiplication. The size of the table is about 84 kbytes (225 values stored).

[3] As well as in the Patel–Sundaram one or in the Blum–Micali one.

Lim and Lee [21] present more flexible tradeoffs between memory and computation time to compute exponentiations over a fixed basis. Their approach is applicable to our scheme as well. We refer to [21] for details, but the following table summarizes the results: the first column is the memory requirement (in kbytes and number of values stored) and the second column is the efficiency (number of multiplications and corresponding rate of pseudo-random bits per multiplication):

| Storage size (kbytes (# values)) | # Mults/Rate (# bits per mult) |
|---|---|
| 16 (42) | 86/32 |
| 58 (155) | 52/53 |
| 189 (504) | 41/67 |

*Remark*. Notice that in the above we are talking about modular multiplications of 3000-bit numbers. Our PRBG has the nice feature that increasing the security parameter increases the bit/multiplication rate of the generator. Indeed, notice that when the security parameter is increased in order to tolerate a stronger adversary, the length of the modulus increases by a larger amount, compared with the length of the exponent. This is a natural consequence of the fact that for the discrete log problem, we have sub-exponential algorithms in the size of the modulus, but only exponential ones in the size of the exponent.

It could appear that this feature is what allows our generator to win over the BBS one, due to the large security parameters produced by the concrete security analysis. We point out, that even for the parameters used today ($n = 1024$ and $c = 160$), assuming that both the BBS and our generator are secure, we still outperform BBS by a factor of almost 2 (without using tables). Indeed, with these parameters our generator still outputs 860 bits per modular exponentiation with 160-bit exponents (on average 240 multiplications) for a rate of 3.5 bits per modular multiplication. Using tables the rate can be pushed to 21 (with 12 kbytes of storage) or 43 (with 300 kbytes). On the other hand, BBS is still stuck at 1 bit per modular squaring, which is comparable with a rate of 2 bits per modular multiplication (as we pointed out above, the version of BBS that outputs more bits per iteration cannot really be considered practical, due to the high cost of the security reduction).

This points to a nice feature of our generator. If the algorithmic state of the art remains the same, the increase of security parameters has a much lower negative impact on the efficiency of our generator. Indeed, the increased cost of modular multiplication is offset in part by the increase in the rate of the generator.

*Implementation Results*. The efficiency of the scheme was confirmed by experimental implementation results. These results confirmed the theoretical analysis in comparison with the BBS generator. They also allowed us to compare our generator with heuristic (i.e. not provably secure) ones based on collision-resistant hashing and/or block ciphers. Although we could not hope to be competitive with respect to these generators, it was surprising to find out that our generator is only slower by a factor of 10, which may not be a too high price to pay in exchange for provable security in certain applications.

The implementation tests were also carried out on tamper-proof dedicated cryptographic devices with specific hardware support for modular exponentiations and multiplications. The results over this platform indicated that precomputation tables are actually not useful, since the overhead of managing the tables is higher than the cost of performing exponentiations in hardware.

A report on the implementation tests can be found in [16].

## 5. Conclusions

In this paper we presented a secure PRBG whose efficiency is comparable with the squaring (BBS) generator. The security of our scheme is based on the assumption that solving discrete logarithms remains hard even when the exponent is small. This assumption was first used by Patel and Sundaram in [25]. Our construction, however, is much faster than theirs since it only uses exponentiations with small inputs.

An alternative way to look at our construction is the following. Under the $c$-DLSE assumption the function $f\colon \{0,1\}^c \longrightarrow Z_p^*$ defined as $f(x) = g^x$ is a one-way function. Our results indicate that $f$ also has strong pseudo-randomness properties over $Z_p^*$. In particular, it is possible to think of it as a pseudo-random generator itself. We are aware of only one other example in the literature of a one-way function with these properties [17] based on the hardness of subset-sum problems.

The $c$-DLSE Assumption is not as widely studied as the regular discrete log assumption so it needs to be handled with care. However, it seems a reasonable assumption to make.

It would be nice to see if there are other cryptographic primitives that could benefit in efficiency from the adoption of stronger (but not unreasonable) number-theoretic assumptions. Examples of this are already present in the literature (e.g. the efficient construction of pseudo-random functions based on the Decisional Diffie–Hellman problem in [24]). It would be particularly interesting to see a PRBG that beats the rate of the squaring generator, even if at the cost of a stronger assumption on factoring or RSA inversion (ruling out pseudo-randomness assumptions like the ones considered in [23]).

## Acknowledgments

This paper owes much to the suggestions and advice of Shai Halevi. Hugo Krawczyk pointed out the remark in Section 4.1. I also thank Mihir Bellare, Dario Catalano and Paul van Oorschot for useful comments and suggestions.

## Appendix. A proof of Theorem 1

In order to prove Theorem 1 we first need to show that the bits in positions $2, \ldots, n-c$ are individually secure. Then we prove simultaneous security.

Before we embark on the proof, we make a remark about computing square roots modulo $p$. It is well known that square roots modulo $p$ can be efficiently computed. Assume now that $p \equiv 3 \bmod 4$ (which is the case for safe primes), and that we are given a quadratic residue $y = g^z$. There are two square roots for $y$: $y_1 = g^{z/2}$ and

$-y_1 = g^{z/2+(p-1)/2}$. Since $(p-1)/2$ is odd, only one of them is a quadratic residue: we call this the *principal square root* of $y$. Since quadratic residuosity modulo $p$ can be efficiently recognized, then principal square roots can be efficiently computed. A last observation is that if $z$ is such that its two least significant bits are zeros, then the principal square root of $y$ is $y_1 = g^{z/2}$.

*Individual Security.* Let $i$ be an integer $1 \leq i \leq n-c$ and assume that we have a polynomial-time Turing Machine $\mathcal{A}_i$ which on input $p, g, g^x$ computes correctly the $i$th bit $x_i$ of $x$ with probability (over $x$) $\frac{1}{2}+\varepsilon(n)$ where $\varepsilon(n) > 1/P(n)$ for some polynomial $P(\cdot)$.

In order to contradict the $c$-DLSE Assumption we show how to build an algorithm $\mathcal{A}$ which uses $\mathcal{A}_i$ and, given $y = g^z$ (for $z \in_R [0..2^c]$), computes $z$ with non-negligible probability. Let $\gamma = 1 - \log \varepsilon = O(\log n)$. We split the proof into two parts: the first case has $1 \leq i < n-c-\gamma$. The second one is $n-c-\gamma \leq i \leq n-c$.

If $1 \leq i < n-c-\gamma$ the inversion algorithm $\mathcal{A}$ works as follows. We are given $y = g^z$ and we know that $z < 2^c$. We compute $z$ bit by bit; let $z_i$ denote the $i$th bit of $z$. To compute $z_1$ we just check if $y$ is a quadratic residue or not. We then "zero" out $z_1$ by setting $y \leftarrow yg^{-z_1}$. From now on, the discrete log of $y$ has a zero on the last significant bit.

To compute $z_2$ we square $y$ $i-1$ times, computing $y_i = y^{2^{i-1}} \bmod p$. This will place $z_2$ in the $i$th position (with all zeros to its right). Since $\mathcal{A}_i$ may be correct only slightly more than half of the time, we need to randomize the query. Thus we choose $r \in_R Z_{p-1}$ and run $\mathcal{A}_i$ on $\hat{y} = y_i g^r \bmod p$. Notice the following:

- Given the assumptions on $z$ and $i$ we know that $y_i = y^{2^{i-1}} = g^{2^{i-1}z}$ and $2^{i-1}z$ is not taken $\bmod p-1$ since it will not "wrap around".
- $\log_g(\hat{y}) = 2^{i-1}z + r \bmod p-1$. However, since $2^{i-1}z$ has at least $\gamma$ leading zeros the probability (over $r$) that $2^{i-1}z + r$ wraps around is $\leq \varepsilon/2$.
- Since $z_2$ has all zeros to its right, there are no carrys in the $i$th position of the sum. Thus by subtracting $r_i$ from $\mathcal{A}_i$'s answer we get $z_2$ unless $2^{i-1}z + r$ wraps around or $\mathcal{A}_i$ provides a wrong answer.

In conclusion we get the correct $z_2$ with probability $\frac{1}{2}+\varepsilon/2$, thus by repeating the process several (polynomially many) times and taking majority we get the correct $z_2$ with very high probability.

Once we get $z_2$, we "zero" it in the value $y$ and then take the principal square root of $y$, i.e. we set $y \leftarrow \sqrt{yg^{-2z_2}}$. Notice that the argument of the square root procedure is a value whose discrete log has the least two significant bits set to zero. So the discrete log of the new $y$ is $z/2$ (see the remark at the beginning of the proof). This places $z_3$ in the place that was previously $z_2$. We can now repeat the above process to discover it, and iterate to compute all the other bits of $z$.

Since each bit is determined with very high probability, the value $z = z_c \cdots z_1$ will be correct with non-negligible probability.

If $n-c-\gamma < i < n-c$ the above procedure may fail since now $2^i z$ does not have $\gamma$ leading zeros anymore. We fix this problem by guessing the $\gamma$ leading bits of $z$ (i.e. $z_{c-\gamma}, \ldots, z_c$). This is only a polynomial number of guesses.

For each guess, we "zero" those bits (let $\alpha$ be the $\gamma$-bit integer corresponding to each guess and set $y \leftarrow yg^{-2^{c-\gamma}\alpha}$). Now we are back in the situation we described above and we can run the inversion algorithm. This will give us a polynomial number of guesses for $z$, and we can then test which is the correct one.

*Simultaneous Security.* Notice that in the above inversion algorithm, every time we query $\mathcal{A}_i$ with the value $\hat{y}$ we know all the bits in position $1, \ldots, i-1$ of $\log_g(\hat{y})$. Indeed, these are the first $i-1$ bits of the randomizer $r$. Thus we can substitute the above oracle with the weaker one $\hat{\mathcal{A}}_i$ which expects $\hat{y}$ and the bits of $\log_g(\hat{y})$ in position $1, \ldots, i-1$. □

## References


[1] L. Adleman. A Subexponential Algorithm for the Discrete Logarithm Problem with Applications to Cryptography. *Proc. IEEE FOCS*, pp. 55–60, 1979.

[2] W. Alexi, B. Chor, O. Goldreich and C. Schnorr. RSA and Rabin Functions: Certain Parts Are as Hard as the Whole. *SIAM J. Comput.*, 17(2):194–209, April 1988.

[3] E. Bach and J. Shallit. *Algorithmic Number Theory, Volume I.* MIT Press, Cambridge, MA, 1996.

[4] L. Blum, M. Blum and M. Shub. A Simple Unpredictable Pseudo-Random Number Generator. *SIAM J. Comput.*, 15(2):364–383, May 1986.

[5] M. Blum and S. Micali. How to Generate Cryptographically Strong Sequences of Pseudo-Random Bits. *SIAM J. Comput.*, 13(4):850–864, November 1984.

[6] D. Coppersmith and I. Sharplinski. On Polynomial Approximation of the Discrete Logarithm and the Diffie–Hellman Mapping. *J. Cryptology*, 13(3):339–360, Summer 2000.

[7] W. Diffie and M. Hellman. New Directions in Cryptography. *IEEE Trans. Inform. Theory*, IT-22:644–654, November 1976.

[8] R. Fischlin and C. Schnorr. Stronger Security Proofs for RSA and Rabin Bits. *J. Cryptology*, 13(2):221–244, Spring 2000.

[9] R. Gennaro An Improved Pseudo-Random Generator Based on Discrete Log. *Proc. CRYPTO '*2000, pp. 469–481. LNCS 1880. Springer-Verlag, Berlin, 2000.

[10] O. Goldreich and V. Rosen. On the Security of Modular Exponentiations with Applications to the Construction of Pseudorandom Generators. *J. Cryptology*, 16(2):71–93, Spring 2003.

[11] S. Goldwasser and S. Micali. Probabilistic Encryption. *J. Comput. System Sci.*, 28:270–299, 1988.

[12] J. Håstad, R. Impagliazzo, L. Levin and M. Luby. A Pseudo-Random Generator from any One-Way Function. *SIAM J. Comput.*, 28(4):1364–1396, 1999.

[13] J. Håstad and M. Näslund. The Security of Individual RSA Bits. *Proc. IEEE FOCS*, pp. 510–519, 1998.

[14] J. Håstad and M. Näslund. Practical Constructions and Analysis of Pseudo-Randomness Primitives. *Proc. ASIACRYPT '*01, pp. 442–459. LNCS 2248. Springer-Verlag, Berlin, 2001.

[15] J. Håstad, A. Schrift and A. Shamir. The Discrete Logarithm Modulo a Composite Hides $O(n)$ Bits. *J. Comput. System Sci.*, 47:376–404, 1993.

[16] N. Howgrave-Graham, J. Dyer and R. Gennaro. Pseudo-Random Number Generation on the IBM 4758 Secure Crypto Coprocessor. *Proc. CHES '*01, pp. 93–102. LNCS 2162. Springer-Verlag, Berlin, 2001.

[17] R. Impagliazzo and M. Naor. Efficient Cryptographic Schemes Provably as Secure as Subset Sum. *J. Cryptology*, 9(4):199–216, 1996.

[18] D. Knuth. *The Art of Computer Programming* (*vol.* 3): *Sorting and Searching*. Addison-Wesley, Reading, MA, 1973.

[19] A.K. Lenstra, H.W. Lenstra, M.S. Manasse and J.M. Pollard. The Number Field Sieve. *Proc. STOC '*90, pp. 564–572. ACM Press, New York, 1990.

[20] A.K. Lenstra and E. Verheul. Selecting Cryptographic Key Sizes. *J. Cryptology*, 14(4):255–293. Autumn 2001.

[21] C.H. Lim and P.J. Lee. More Flexible Exponentiation with Precomputation. *Proc. CRYPTO '*94, pp. 95–107. LNCS 839. Springer-Verlag, Berlin, 1994.

[22] D. Long and A. Wigderson. The Discrete Log Hides $O(\log n)$ Bits. *SIAM J. Comput.*, 17:363–372, 1988.
[23] S. Micali and C. Schnorr. Efficient, Perfect Polynomial Random Number Generators. *J. Cryptology*, 3(3):157–172, Summer 1991.
[24] M. Naor and O. Reingold. Number-Theoretic Constructions of Efficient Pseudo-Random Functions. *Proc. IEEE FOCS*, pp. 458–467, 1997.
[25] S. Patel and G. Sundaram. An Efficient Discrete Log Pseudo Random Generator. *Proc. CRYPTO* '98, pp. 304–317. LNCS 1462. Springer-Verlag, Berlin, 1998.
[26] R. Peralta. Simultaneous Security of Bits in the Discrete Log. *Proc. EUROCRYPT* '85, pp. 62–72. LNCS 219. Springer-Verlag, Berlin, 1986.
[27] S.C. Pohlig and M.E. Hellman. An Improved Algorithm for Computing Logarithms over $GF(p)$ and its Cryptographic Significance. *IEEE Trans. Inform. Theory*, IT-24(1):106–110, January 1978.
[28] J.M. Pollard. Monte-Carlo Methods for Index Computation (mod $p$). *Math. Comp.*, 32(143):918–924, 1978.
[29] J.M. Pollard. Kangaroos, Monopoly and Discrete Logarithms. *J. Cryptology*, 13(4):437–447, Autumn 2000.
[30] C. Schnorr. Security of Almost ALL Discrete Log Bits. Electronic Colloquium on Computational Complexity. Report TR98-033. Available at `http://www.eccc.uni-trier.de/eccc/`.
[31] V. Shoup. Lower Bounds for Discrete Logarithms and Related Problems. *Proc. EUROCRYPT* '97, pp. 256–266. LNCS 1233. Springer-Verlag, Berlin, 1997.
[32] D. Stinson. Some Baby-Step Giant-Step Algorithms for the Low Hamming Weight Discrete Logarithm Problem. *Math. Comp.*, 71:379–391, 2002.
[33] P.C. van Oorschot and M. Wiener. On Diffie–Hellman Key Agreement with Short Exponents. *Proc. EUROCRYPT* '96, pp. 332–343. LNCS 1070. Springer-Verlag, Berlin, 1996.
[34] P.C. van Oorschot and M. Wiener. Parallel Collision Search with Cryptanalytic Applications. *J. Cryptology*, 12(1):1–28. Winter 1999.
[35] A. Yao. Theory and Applications of Trapdoor Functions. *Proc. IEEE FOCS*, pp. 80–91, 1982.