An Improved Pseudo-random Generator Based on Discrete Log

Rosario Gennaro

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

Abstract. Under the assumption that solving the discrete logarithm problem modulo an n-bit prime p is hard even when the exponent is a small c-bit number, we construct a new and improved pseudo-random bit generator. This new generator outputs $n - c - 1$ bits per exponentiation with a c-bit exponent.
Using typical parameters, $n = 1024$ and $c = 160$, this yields roughly 860 pseudo-random bits per small exponentiations. Using an implementation with quite small precomputation tables, this yields a rate of more than 20 bits per modular multiplication, thus much faster than the the squaring (BBS) generator with similar parameters.

1 Introduction

Many (if not all) cryptographic algorithms rely on the availability of truly random bits. 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 (PRBG) was introduced in papers by Blum and Micali [4] and Yao [20]. 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 [4] 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 [8] where it was shown that secure PRBGs exists if any one-way function does.

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 [14] and Peralta [17], 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.* [10] show that if one considers discrete-log modulo a 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. product of two primes of identical size and both $\equiv 3 \bmod 4$.) Under this assumption, Blum *et al.* [3] 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 the intractability of factoring as the underlying assumption. Up to this date, this is the most efficient provably secure PRBG.

In [16] Patel and Sundaram propose a very interesting variation on the Blum-Micali generator. They showed that if solving the discrete log problem modulo an n-bit prime p is hard even when the exponent is small (say only c bits long with $c < n$) then 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 if 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.

Our Contribution. In this paper we show that it is possible to construct a high-rate discrete-log based secure PRBG. Under the same assumption introduced in [16] 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 $f : \{0,1\}^c \longrightarrow Z_p^*$ defined as $f(x) = g^x \bmod p$ is is a one-way function then it also has also 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 pseudo-random bit generator.

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. Using the precomputation scheme proposed in [13] one can show that such exponentiation will cost on average roughly 40 multiplications, using a table of only 12 Kbytes. Thus we obtain a rate of more than 21 pseudo-random bits per modular multiplication. Different tradeoffs between memory and efficiency can be obtained.

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 $x = x_n x_{n-1} \dots x_1$ 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.

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 [7]) 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*

$$\delta_{\mathcal{D},X_n} = Prob[x \leftarrow X_n \ ; \ \mathcal{D}(x) = 1]$$

$$\delta_{\mathcal{D},Y_n} = Prob[y \leftarrow Y_n \ ; \ \mathcal{D}(y) = 1]$$

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 to define pseudo-random number generators [4,20]. There are several equivalent definitions, but the following one is sufficient for our purposes. Consider a family of functions

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

where $k_n < n$. 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 G_n is a* cryptographically strong pseudo-random bit generator *if the function 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

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

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* AG_n *is a* cryptographically strong pseudo-random generator *over* A_n *if the function* 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 they may not be perfect, 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 will 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

$$f : Z_{p-1} \longrightarrow Z_p^*$$

$$f(x) = g^x \bmod p$$

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 [5] on public-key cryptography). The best known algorithm to compute discrete logarithms is the so-called *index calculus* method [1] which however runs in time sub-exponential in n.

In some applications (like the one we are going to 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 a integer which we can think 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 index-calculus method 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 [12] or the *rho* algorithm by Pollard [18] can compute the discrete log of y in $O(2^{c/2})$ time.

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 $x \leq 2^c$. In [16] 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 will adopt it as the basis of our results as well.

Assumption 1 (c-DLSE [16]) *Let $PRIMES(n)$ be the set of n-bit 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*

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

This assumption is somewhat supported by a result by Schnorr [19] who proves that no *generic* algorithm can compute c-bits discrete logarithms in less than $2^{c/2}$ generic steps. A generic algorithm is restricted to only perform group operations and cannot take advantage of specific properties of the encoding of group elements.

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).

2.4 Hard Bits for Discrete Logarithm

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). 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).

A Boolean predicate Π is said to be *hard* 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 [4] prove that the predicate

$$\Pi : Z_{p-1} \longrightarrow \{0,1\}$$

$$\Pi(x) = (x \leq \frac{p-1}{2})$$

is hard for the discrete logarithm function. Recently Håstad and Näslund [9] proved that every bit of the binary representation of x (except the least significant one) is hard for the discrete log function.

In terms of *simultaneous* security of several bits, Long and Wigderson [14] and Peralta [17] showed that there are $O(\log\log p)$ predicates which are *simultaneously hard* for discrete log. Simultaneously hard means that the whole collection of bits looks "random" even when given $y = g^x$. A way to formalize this (following [20]) is 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: there exists $O(\log\log p)$ Boolean predicates

$$\Pi_i : Z_{p-1} \longrightarrow \{0,1\} \quad \text{for } i = 1, \ldots, O(\log\log p)$$

such that for every $1 \leq j \leq O(\log\log p)$, if there exists a probabilistic polynomial-time algorithm $\mathcal{A}$ and a polynomial $P(\cdot)$ such that

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

then there exists a probabilistic polynomial time algorithm $\mathcal{A}'$ which on input g^x computes x with non-negligible probability.

2.5 The Patel-Sundaram Generator

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

In [16] 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[1]:

Theorem 1 ([16]). *For sufficiently large n, if p is a n-bit prime such that $p \equiv 3 \bmod 4$ 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*

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

[1] We point out that [16] requires that p be a *safe* prime, i.e. such that $(p-1)/2$ is also a prime; but a close look at their proof reveals that $p \equiv 3 \bmod 4$ suffices.

We refer the reader to [16] for a proof of this Theorem.

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 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 a n-bit prime, $p \equiv 3 \bmod 4$, and $c = \omega(\log n)$. Consider the following function (which we call PSG for Patel-Sundaramam Generator):

$$\mathsf{PSG}_{n,c} : Z_{p-1} \longrightarrow Z_p^* \times \{0,1\}^{n-c-1}$$

$$\mathsf{PSG}_{n,c}(x) = (g^x \bmod p, x_2, \ldots, x_{n-c})$$

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 [16], 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

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

In the next section we show our new generator and we prove that if it is not secure that 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 a n-bit prime, $p \equiv 3 \bmod 4$ and $c = \omega(\log n)$. Let g be a generator of Z_p^* and denote with $\hat{g} = g^{2^{n-c}} \bmod p$. Recall that if s is an integer we denote with s_i the i^{th}-bit in its binary representation.

The function we consider is the following.

$$\mathsf{RG}_{n,c} : Z_{p-1} \longrightarrow Z_p^*$$

$$\mathsf{RG}_{n,c}(s) = \hat{g}^{(s \text{ div } 2^{n-c})} g^{s_1} \bmod p$$

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).

The function RG induces a distribution over Z_p^* in the usual way. We denote it with $RG_{n,c}$ the following probability distribution over Z_p^*

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

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 a n-bit prime, with $p \equiv 3 \bmod 4$ 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 modified Patel-Sundaram generator 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 will contradict Theorem 1 and ultimately the c-DLSE Assumption.

Sketch of 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

$$\delta_{\mathcal{D},U_n} = Prob[x \leftarrow Z_p^* \ ; \ \mathcal{D}(p, g, x, c) = 1]$$

$$\delta_{\mathcal{D},RG_{n,c}} = Prob[s \leftarrow Z_{p-1} \ ; \ \mathcal{D}(p, g, \mathsf{RG}_{n,c}(s), c) = 1]$$

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

More in 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 $r^{(0)}, r^{(1)}, \ldots, r^{(k)}$ 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 pseudo-random bit generator (see Definition 2).*

Sketch of 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 position $2, 3, \ldots, n-c$ of a random element of Z_{p-1} and thus their bias is bounded by 2^{-c} (i.e. the statistical distance between the distribution of $r^{(0)}$ and the uniform distribution over $\{0,1\}^{n-c-1}$ is bounded by 2^{-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}.

More formally, the proof follows a hybrid argument. 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 for a specific index i we must have a distinguisher $\mathcal{D}_1$ 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^* (just take the bits in position $2, 3, \ldots, n-c$ of the input and pass them to $\mathcal{D}_2$). This contradicts Lemma 1 and ultimately the c-DLSE Assumption. □

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 [16] 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 w.l.o.g.

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

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 $2 \le j < n-c-\log P(n)$ and in time $O((n-c)cP^3(n)T(n))$ if $n-c-\log P(n) \le j \le n-c$ (the hidden constant is very small). This is a very crude analysis of the efficiency of the reduction in [16] and it is quite possible to improve on it (details in the final paper).

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

Comparison with the BBS generator. The BBS generator was introduced by Blum *et al.* in [3] under the assumption that deciding quadratic residuosity modulo a composite is hard. The generator works by repeatedly squaring $\bmod N$ 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 [2] reduction is not very tight and it was recently improved by Fischlin and Schnorr in [6]. 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 k bits per iteration, the complexity grows by a factor of 2^{2k} and the reduction quickly becomes more expensive than known factoring algorithms. Notice instead that the reduction in [16] (and thus in ours) depends only linearly on the number of bits outputted.

Concrete Parameters. Let's fix $n = 1024$ and $c = 160$. With these parameters we can safely assume that the complexity of the best known algorithms to break c-DLSE [1,12,17] is beyond the reach of today's computing capabilities.

For moduli of size $n = 1024$, the results in [6] seem to indicate that in practice, for $n = 1024$ we can output around 4 bits per iteration of the BBS generator, if we want to rule out adversaries which run in time $T = 1$ MIPS-year (roughly 10^{13} instructions) and predict the next bit with advantage $1/100$ (which is quite high). This yields a rate of 4 bits per modular squaring.

Using the same indicative parameters suggested in [6], we can see that we can safely output all $n - c - 1 \approx 860$ bits in one iteration of the Patel-Sundaram generator. Since the security of our scheme is basically the same as their generator we can also output all 860 bits in our scheme as well.

Thus we obtain 860 bits at the cost of roughly 240 multiplications, which yields a rate of about 3.5 bits per modular multiplication. Thus the basic implementation of our scheme has efficiency comparable to the BBS generator. In the next section we show how to improve on this, by using precomputation tables.

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[2] 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 this values to compute fastly $\hat{g}^s$ 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 $.5c$ multiplications on average. The cost is an increase to $O(cn)$

[2] As well as in the Patel-Sundaram one, or in the Blum-Micali one

bits of required memory. With this simple improvement one iteration of our generator will require roughly 80 multiplications, which yields a rate of more that 10 pseudo-random bits per multiplication. The size of the table is about 20 Kbytes.

Lim and Lee [13] present more flexible trade-offs between memory and computation time to compute exponentiations over a fixed basis. Their approach is applicable to our scheme as well. In short, the [13] precomputation scheme is governed by two parameters h, v. The storage requirement is $(2^h - 1)v$ elements of the field. The number of multiplications required to exponentiate to a c-bit exponent is $\lceil \frac{c}{h} \rceil + \lceil \frac{c}{hv} \rceil - 2$ in the worst case.

Using the choice of parameters for 160-bit exponents suggested in [13] we can get roughly 40 multiplications with a table of only 12 Kbytes. This yields a rate of more than 21 pseudo-random bits per multiplication. A large memory implementation (300 Kbytes) will yield a rate of roughly 43 pseudo-random bits per multiplication.

In the final version of this paper we will present a more complete concrete analysis of our new scheme.

5 Conclusions

In this paper we presented a secure pseudo-random bit generator whose efficiency is comparable to 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 [16]. 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 $f : \{0,1\}^c \longrightarrow Z_p^*$ defined as $f(x) = g^x$ is a one-way function. Our results indicate that f has also strong pseudo-randomness properties over Z_p^*. In particular it is possible to think of it as pseudo-random generator itself. We are aware of only one other example in the literature of a one-way function with this properties, in [11] based on the hardness of subset-sum problems.

The 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 [15].) It would be particularly interesting to see a pseudo-random bit generator that beats the rate of the squaring generator, even if at the cost of a stronger assumption on factoring or RSA.

Acknowledgments

This paper owes much to the suggestions and advices of Shai Halevi. Thanks also to the other members of the CRYPTO committee for their suggestions and to Dario Catalano for reading early drafts of the paper.

References

1. L. Adleman. *A Subexponential Algorithm for the Discrete Logarithm Problem with Applications to Cryptography.* 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. Computing, 17(2):194–209, April 1988.
3. L. Blum, M. Blum and M. Shub. *A Simple Unpredictable Pseudo-Random Number Generator.* SIAM J.Computing, 15(2):364–383, May 1986.
4. M. Blum and S. Micali. *How to Generate Cryptographically Strong Sequences of Pseudo-Random Bits.* SIAM J.Computing, 13(4):850–864, November 1984.
5. W. Diffie and M. Hellman. *New Directions in Cryptography.* IEEE Trans. Inf. Theory, IT-22:644–654, November 1976.
6. R. Fischlin and C. Schnorr. *Stronger Security Proofs for RSA and Rabin Bits.* J.Crypt., 13(2):221–244, Spring 2000.
7. S. Goldwasser and S. Micali. *Probabilistic Encryption.* JCSS, 28:270–299, 1988.
8. J. Håstad, R. Impagliazzo, L. Levin and M. Luby. *A Pseudo-Random Generator from any One-Way Function.* SIAM J.Computing, 28(4):1364-1396, 1999.
9. J. Håstad and M. Näslund. The Security of Individual RSA Bits. IEEE FOCS, pp.510–519, 1998.
10. J. Håstad, A. Schrift and A. Shamir. *The Discrete Logarithm Modulo a Composite Hides $O(n)$ Bits.* JCSS, 47:376-404, 1993.
11. R. Impagliazzo and M. Naor. *Efficient Cryptographic Schemes Provably as Secure as Subset Sum.* J.Crypt., 9(4):199–216, 1996.
12. D. Knuth. *The Art of Computer Programming (vol.3): Sorting and Searching.* Addison-Wesley, 1973.
13. C.H. Lim and P.J. Lee. *More Flexible Exponentiation with Precomputation.* CRYPTO'94, LNCS 839, pp.95–107.
14. D. Long and A. Wigderson. *The Discrete Log Hides $O(\log n)$ Bits.* SIAM J.Computing, 17:363–372, 1988.
15. M. Naor and O. Reingold. *Number-Theoretic Constructions of Efficient Pseudo-Random Functions.* IEEE FOCS, pp.458–467, 1997.
16. S. Patel and G. Sundaram. *An Efficient Discrete Log Pseudo Random Generator.* CRYPTO'98, LNCS 1462, pp.304–317, 1998.
17. R. Peralta. *Simultaneous Security of Bits in the Discrete Log.* EUROCRYPT'85, LNCS 219, pp.62–72, 1986.
18. J. Pollard. *Monte-Carlo Methods for Index Computation (mod p).* Mathematics of Computation, 32(143):918–924, 1978.
19. C. Schnorr *Security of Allmost ALL Discrete Log Bits.* Electronic Colloquium on Computational Complexity. Report TR98-033. Available at `http://www.eccc.uni-trier.de/eccc/`.
20. A. Yao. *Theory and Applications of Trapdoor Functions.* IEEE FOCS, 1982.