An Efficient Discrete Log Pseudo Random Generator

Sarvar Patel and Ganapathy S. Sundaram

Bell Labs
67 Whippany Rd, Whippany, NJ 07981, USA
sarvar@bell-labs.com, ganeshs@bell-labs.com

Abstract. The exponentiation function in a finite field of order p (a prime number) is believed to be a one-way function. It is well known that $O(\log\log p)$ bits are simultaneously hard for this function. We consider a special case of this problem, the discrete logarithm with short exponents, which is also believed to be hard to compute. Under this intractibility assumption we show that discrete exponentiation modulo a prime p can hide $n-\omega(\log n)$ bits ($n = \lceil \log p \rceil$ and $p = 2q+1$, where q is also a prime). We prove simultaneous security by showing that any information about the $n-\omega(\log n)$ bits can be used to discover the discrete log of $g^s \bmod p$ where s has $\omega(\log n)$ bits. For all practical purposes, the size of s can be a constant c bits. This leads to a very efficient pseudo-random number generator which produces $n-c$ bits per iteration. For example, when $n = 1024$ bits and $c = 128$ bits our pseudo-random number generator produces a little less than 900 bits per exponentiation.

1 Introduction

A function f is said to be *one-way* if it is *easy* to compute but *hard* to invert. With appropriate selection of parameters, the discrete exponentiation function over a finite field, $g^x \bmod p$,is believed to be a one-way function (where g is a generator of the cyclic group of non zero elements in the finite field). The intractability of its inverse, the discrete logarithm problem, is the basis of various encryption, signature and key agreement schemes. Apart from finite fields, other *finite groups* have been considered in the context of discrete exponentiation. One such example is the group of points on an *elliptic curve* over a finite field. Koblitz and Miller (independently) [15], [17], considered the group law on an elliptic curve to define a public key encryption scheme suggesting that *elliptic curve addition* is also a one-way function. Another number theoretic problem that is considered to be *hard* is the problem of factoring integers. Examples of functions relying on factoring which are believed to be one-way are the RSA and Rabin functions. Closely related to factoring is the problem of deciding quadratic residuosity modulo a composite integer.

A concept which is intimately connected to one-way functions is the notion of hard bits, which was first introduced by Blum & Micali. Informally, a hard bit $B(.)$ for a one way function $f(.)$ is a *bit* which is as hard to compute as it is to invert f. Blum and Micali showed that *the most significant bit* is a hard

bit for the discrete logarithm problem over a finite field. To be precise, their notion of most significant bit corresponds to the Boolean predicate which is one if the index of the exponent is greater than $\frac{p-1}{2}$ and zero otherwise. They defined and proved this hard bit and successfully used it to show the importance of hard bits in secure pseudo-random bit generation. Soon after, the hard bits of RSA & Rabin functions were also discovered by Ben-Or *etal* [2] which led to a new secure pseudo-random bit generator. Blum, Blum and Shub [3] used the quadratic residue problem over a composite integer to design yet another secure pseudo-random bit generator. Their work was based on the security of the quadratic residue problem which was investigated by Goldwasser and Micali [8]. Later Goldreich and Levin [7] proved that all one-way functions have a hard bit. More generally they were able to show that for any one-way function a logarithmic number of one bit predicates are simultaneously hard. This led to the work of [9], where they proved how to use any one-way function to build secure pseudo-random bit generators. The use of pseudo-random bits in cryptography relates to one time pad style encryption and bit commitment schemes, to name a few.

All the above generators based on one bit predicates suffer from the same problem, namely they are too *slow*. All of them output one bit per modular exponentiation. The concept of *simultaneous hardness* is the first step in speeding things up. Intuitively, the notion of simultaneous hardness applied to a group of bits associated to a one-way function f states that it is computationally as hard as the inverse of the one-way function to succeed in computing any information whatsoever about the given group of bits given only $f(x)$. Using this notion one can extract collections of bits per operation and hence the speed up. Long and Wigderson [16] and Peralta [20] showed that $\log\log p$ bits of the discrete log modulo a prime number p are simultaneously hard. On the other hand the works of Vazirani and Vazirani [24] and Alexi *etal* [1] address the notion of simultaneous hardness of RSA and Rabin bits. Later Kaliski [12] showed individual hardness of bits (in the Blum Micali sense) of the elliptic curve group addition problem using a novel oracle proof technique applicable to any finite Abelian group. His methods extend to show simultaneous hardness (stated but not proved in the paper) of $\log n$ bits where n is the order of the group. More recently, Hastad, Schrift and Shamir [10], have designed a much more efficient generator which produces $\frac{n}{2}$ bits per iteration where n is the number of bits of the modulus. The one-way function they have considered is the discrete exponentiation function modulo a composite integer (to be precise a Blum integer). Once again the method of generation relies on the proof that $\frac{n}{2}$ bits of every iteration are simultaneously hard. The use of a composite modulus allows them to relate individual and simultaneous hardness of bits to factoring the modulus. In all these works the common strings are the results of Yao contained in his seminal work [25] which laid the foundations to a complexity theoretic approach to cryptography which paved the way for a quantification of security in terms of known hard problems.

In this paper we construct a very efficient cryptographic pseudo-random bit

generator attached to modular exponentiation in a finite field of cardinality p (where p is a prime number of the form $2q + 1$, and q is also prime). This assumption on the structure of the finite field holds for the entire paper. We show that $n - \omega(\log n)$ bits of every iteration are simultaneously secure (where $2^{O(\log n)}$ is a polynomial value in n and $O(\log n)$ is the order of the number of bits needed to represent a polynomial in n. Note that $2^{\omega(\log n)}$ is greater than any polynomial value in n and $\omega(\log n)$ is the order of the number of bits needed to represent it.) Hence each iteration produces more bits than any other method discovered so far. In fact, the construction that we present here is maximal since if we extract more bits then only $O(\log n)$ would have to be guessed, which can be exhaustively searched in polynomial time (since $2^{O(\log n)}$ is polynomial in n). The novelty in this work is to relate the security of the random bit generation to the problem of solving the discrete logarithm with short exponents. The motivation for this technique is derived from the above mentioned work of [10] where although they are using a modular exponentiation function modulo a composite, the security of the system is related to factoring the underlying modulus. In a similar but not so obvious sense, we use exponentiation in a finite field for the generation but relate the security to the strength of the discrete log problem (over the same prime modulus) but with *short* exponents. The proofs are simple and rely on known techniques. In this paper an oracle for the $i - th$ bit gives the value of $i - th$ bit when the binary representation is used for the argument. This is a different representation of the i-th bit than that used by Blum-Micali and Long-Wigderson.

The paper is organized as follows: In section 2 we discuss the discrete log problem and in particular the short exponent discrete log problem. Details of the oracles and hardness of bits are formalized in this section. In section 3 we show that the trailing $n - \omega(\log n)$ bits are individually hard with respect to the discrete logarithm problem with short exponents. In section 4 we prove simultaneous hardness of the trailing $n - \omega(\log n)$ bits. Once again this is with respect to the discrete log with short exponents problem. In section 5 we discuss the design of the pseudo-random generator and provide the proof of security and conclude in section 6. In the appendix, we discuss some extensions of this work to include other Abelian groups and possible ways to improve the efficiency of the pseudo random generator.

2 The Discrete Logarithm Problem

We first define the discrete logarithm problem. Let p be a prime and g a generator for $Z_p{}^*$, the multiplicative cyclic group of nonzero elements in the finite field of order p. Then for $1 \leq x \leq p - 1$ the function which maps x to $g^x \bmod p$ defines a permutation.

Problem 1. The discrete logarithm problem is to find x given $y \epsilon Z_p{}^*$ such that $g^x \bmod p \equiv y$.

Let $n = \lceil \log p \rceil$ be the length of p in binary, then $g^x \bmod p$ is computable in Poly (n) time. However, there is no known deterministic or randomized algorithm which can compute the discrete logarithm in Poly (n) number of steps. The best algorithm to compute the discrete logarithm in a finite field of order p, is the index calculus method. Even this is infeasible if p is appropriately large (e.g. 1024 bits) since the complexity is subexponential and not polynomial in n. On the other hand for primes such that $p-1$ consists of only small factors, there are very fast algorithms whose complexity is equal to the complexity of the discrete log in a field whose cardinality is equal to its largest prime factor. This algorithm is due to Pohlig and Hellman [21].

2.1 Discrete Logarithm with Short Exponents

For efficiency purposes the exponent x is sometimes restricted to c bits (e.g. $c = 128$ or 160 bits) since this requires fewer multiplications. There are square root time algorithms to find x in $\frac{c}{2}$ steps, due to Shanks [14] and Pollard [22]. Thus c should be at least 128 bits to provide 64 bits of security. By this we mean an attacker should perform at least 2^{64} operations in order to crack the discrete logarithm using these algorithms. At the moment, there is no faster way to discover the discrete logarithm even with x so restricted. In particular, the complexity of index calculus algorithms is a function of the size of the entire group and does not depend on the size of the exponent.

We will also restrict x, in particular, we will restrict it to be slightly greater than $O(\log n)$ bits, but not to save on multiplications. The size of the exponent will be denoted $\omega(\log n)$, described in section 1. Hence, even with the square root attack one needs greater than $2^{O(\log n)}$ steps or greater than a polynomial in n number of steps.

The hard problem we consider in this paper is the inverse of this special case of the discrete exponentiation function. In other words:

Problem 2. Let s be an integer which is significantly smaller compared to p. The DLSE problem is to find s given $y \epsilon Z_p^*$ such that $g^s \bmod p \equiv y$.

The DLSE problem has been studied by [19] in the context of the Diffie-Hellman key agreement scheme. The use of short exponents in the Diffie-Hellman protocol is to speed up the process of exponentiation. Typically the cost of computing g^x is linearly related to the bit length of x, hence real-time computing costs have motivated the use of low order exponents. Care is necessary to ensure that such optimizations do not lead to security weaknesses. The above mentioned paper [19], presents a set of attacks and methods to rectify the situation. In particular their conclusions suggest the use of *safe primes.*

Another example of the use of shorter exponents is in the generation of digital signatures. The digital signature standard (DSS) published by the NIST [6] is based on the discrete logarithm problem. It is a modification of the ElGamal signature scheme. The ElGamal scheme usually leads to a signature having $2n$ bits, where n is the number of bits of p (the modulus). For potential applications

a shorter signature is desirable. DSS modifies the ElGamal scheme so that a 160-bit message is signed using a 320 bit signature but computations are all done modulo a 512 bit prime. The methodology involves the restrictions of all computations to a subgroup of size 2^{160}. The assumed security of the scheme is based on two different but very related problems. First of these is the discrete log in the entire group which uses a 512 bit modulus, where the index calculus algorithm applies. The second is the discrete log problem in the subgroup of the cyclic group of nonzero elements in the finite field. Here Shanks'square root algorithm reduces the complexity to $O(2^{80})$ since the exponent is only 160-bits. Although the DLSE and the subgroup discrete log problems are not equivalent, the square root time attacks apply to both problems.

2.2 Hardness of Bits

As indicated in the introduction, the notion of hard bits is intimately connected to that of a one-way function. In this paper we define a mild generalization of hard bits.

Definition 3. Let $f(x)$ and $f'(s)$ be one-way functions. Let $B :\rightarrow \{0,1\}$ be a Boolean predicate. Given $f(x)$ for some x, the predicate $B(x)$ is said to be f'-hard if computing $B(x)$ is as hard as inverting $f'(s)$, i.e. discovering s.

When f and f' are the same as are x and s, then we have the usual definition of hard bits. For example, discovering the Blum-Micali bit is as hard as computing the discrete logarithm. But in this paper we allow f and f' to be different. An example of this phenomenon, is discrete exponentiation modulo a composite modulus. Here the discrete logarithm in the ring of integers modulo a composite is a hard function, and so is factoring. So $f(x) = g^x \bmod m$ and $f'(s) = f'(s = p, q) = m$. Clearly, there are boolean predicates $B(x)$ which are f-hard but there may be other predicates which are f'-hard, but not f-hard. That is computing $B(x)$ is as hard as factoring the modulus m, but may be not as hard as the discrete log modulo a composite [10]. In this paper we consider a similar situation. We consider the one-way function of discrete exponentiation, but we prove that the $n-\omega(\log n)$ bits of the exponent are *DLSE-simultaneously hard*. That is for us $f(x) = g^x \bmod p$ and $f'(s) = g^s \bmod p$ where s is a short exponent. The best previous result showed simultaneous hardness of $\frac{n}{2}$ of the bits [10], but our result shows simultaneous hardness for almost all the n bits. Our results are maximal. In other words, in a pseudo-random generator, if in any iteration we hide only $O(\log n)$ or fewer bits, then any attacker can compute the seed of the generator by making a polynomial number of guesses. Hence one cannot further improve on these results regarding number of bits produced per iteration.

2.3 Binary Representation

The number x can be represented in binary as $b_n \cdot 2^{n-1} + b_{n-1} \cdot 2^{n-2} + \ldots + b_2 \cdot 2^1 + b_1 \cdot 2^0$ where b_i is either 0 or 1. The $i-th$ bit problem is to discover the value

of b_i of x. The $i-th$ bit is hard if computing it is as difficult as computing the inverse of $f'(s)$. If we had an perfect oracle, $O^i(g,p,y)$, which outputs the value of b_i then the bit is hard if there is a Poly(n) time algorithm which makes Poly(n) queries to the oracle $O^i(g,p,\cdot)$ and computes the entire value of s. We know the least significant bit is not hard because there is a Poly (n) time algorithm to compute it, namely by computing the Legendre symbol.

An imperfect oracle, $O_\epsilon(p,g,\cdot)$, is usually defined as an oracle which outputs the correct bit value with probability greater than $\frac{1}{2}+\frac{1}{Poly(n)}$. Some of the most significant bits of x, in fact $O(\log n)$ most significant bits, can be biased, but as we shall see later they do not affect us.

2.4 Blum-Micali Representation

In this paper, we will use the binary representation when we discuss the security of the i-th bit, however, we want to mention another interpretation of the i^{th} bit. Blum-Micali introduced a particular bit predicate, $B(x)$ and showed its hardness. $B(x)$ is 0 if $1 \leq x \leq \frac{p-1}{2}$ and $B(x)$ is 1 if $\frac{p-1}{2} < x \leq p-1$. This is sometimes referred to as the most significant bit of x and it is clearly different from the most significant bit of x under the binary representation. Others [16] have extended the definitions to define the k most significant bits. Often the Blum-Micali representation is used to refer to the most significant bits, while the binary representation is used for the least significant bits. In this paper we will use the binary representation when referring to the i^{th} bit, unless specified otherwise.

3 Individual Hardness of Bits

In this section, we discuss the security of the trailing $n-\omega(\log n)$ bits, where $\omega(\log n)$ is as defined earlier. To be precise we show that except the least significant bit, all the $n-\omega(\log n)$ lower bits are individually hard with respect to the DLSE problem. Based on definition 3, this amounts to proving the bits of the discrete logarithm are DLSE-hard. The proof techniques we employ are variations of techniques from [4] and [20].

Let $O^i(g,y,p)$ be a perfect oracle which gives the i^{th} bit (for any $i \in [2, n-\omega(\log n)]$. Note that i increases from right to left and $i=1$ for the least significant bit. Given this oracle we show that in polynomial number of steps we can compute the short exponent discrete logarithm. In addition, we prove hardness of individual bits by showing that given an imperfect oracle $O^i_\epsilon(g,y,p)$ with ϵ advantage to predict the i^{th} bit (for any i in the prescribed range), we can turn this into an algorithm to compute the discrete logarithm of a short exponent in probabilistic polynomial time by making a polynomial number of queries to this oracle. For the rest of the paper we will refer to lower k bits to mean *lower k bits excluding the least significant bit*, for any k.

Theorem 4. *The lower $n-\omega(\log n)$ bits are individually DLSE-hard.*

Proof: According to definition 3, it is enough to show that given $O^i(g,y,p)$ (where g is a generator of the group in question) we can compute $\log y$ for all y such that $s = \log y$ is a short exponent. In this paper we assume that $p-1 = 2q$, where q is an odd integer.

(a) *Perfect Oracles* - $O^i(g,y,p)$. We are given g^s and g and we know in advance that s is small (consisting of $\omega(\log n)$ bits). Now, computing the least significant bit is always easy, via the Legendre symbol. Hence we compute it and set it to zero. Let $i = 2$ and suppose we have an oracle for the 2nd bit. If this is a perfect oracle then we discover the second bit. Once this is known then we set it to zero and we will continue to refer to the new number as g^s. Next we compute the square roots of g^s. The roots are $g^{\frac{s}{2}}$ and $g^{\frac{s}{2}+\frac{p-1}{2}}$ where we refer to the former as the principal square root. Since the two least significant bits of g^s are zero, we know that the principal square root has LSB equal to zero (or equivalently Legendre symbol one). This allows us to identify the principal square root. Now run the oracle on the principal square root and compute the second least significant bit. This bit is really the third least significant bit of s. Once again, set this bit to zero and repeat the process. Clearly, in poly(n) steps we would have computed s one bit at a time from right to left, given an oracle for the second bit.

Now, in general when we are given the oracle for the i^{th} bit ($i > 2$) we square g^s $i-2$ times. Then the 2nd LSB of s is at the i^{th} position, and we run the oracle to compute this bit; we zero this bit and once again compute square roots. The principal square root corresponds to the root with LSB equal to zero. Now the $(i+1)^{th}$ bit of s can be computed by running the oracle on the principal square root. Continue this process and in c steps where $c = \log s$, we would know s.

(b) *Imperfect Oracles* - $O^i_\epsilon(g,y,p)$. Suppose we have an imperfect oracle which succeeds in finding the i^{th} bit in only ϵ more than fifty percent of the $x \in Z_p^*$. Then we can concentrate the stochastic advantage and turn this oracle into an oracle which answers *any specific instance* correctly with arbitrarily high probability.

We divide the proof into two parts

(i) The lower $2 \leq i < n - \omega(\log n) - O(\log n)$ bits are individually hard.

(ii) The middle $n-\omega(\log n)-O(\log n) \leq i \leq n-\omega(\log n)$ bits are individually hard.

(i) Let $i = 2$ and suppose we have an imperfect oracle for the 2nd bit whose advantage is ϵ, i.e., the oracle can give the correct answer on ϵ more than fifty percent of the possible inputs (and we do not know which ones). Then let $\{r_j\}$ be a sequence of polynomial number of random numbers between 1 and $p-1$. We run the oracle on g^{s+r_j}, where the LSB of s is zero. Via the weak law of large numbers [4], a simple counting of the majority of $1's$ and $0's$ of the oracle output (after neutralizing the effect of the random number) for the second LSB yields this bit with high probability. Now compute the square roots and pick the principal square root as earlier. Once again repeat the process with a fresh set of random numbers to discover the next bit. In $c = \log s$ steps we recover a candidate and verify that $g^{candidate} = g^s \bmod p$. If they are not equal then the

whole process is repeated. Clearly in poly(n) steps we would have discovered s one bit at a time from right to left. The details of the proofs are omitted, and we refer to [4] or [20] for further details. The only aspect that needs additional mention is the fact, when we randomize it is possible that for some r_j when we add them to the exponent we may exceed $p-1$. We refer to this as cycling. Assuming that we pick our random numbers uniformly, then we argue that the probability of this cycling is negligible because most of the leading bits of g^s are zero.

Suppose $i > 2$. Then we square g^s $i-1$ times, and repeat the above process and conclude that any oracle which has an ϵ advantage will lead to a polynomial time algorithm to compute s. The probability of cycling is still negligible for $2 \leq i < n - \omega(\log n) - O(\log n)$ because even in the extreme case when $i = n - \omega(\log n) - O(\log n)$ the chance of cycling is $\frac{1}{2^{\omega(\log n)}}$ or less than one over any polynomial.

(ii) The proof of this step is also similar to the second part of the proof of (i) except that one has to set the initial t bits of s to zero by guessing, before we start the randomizing process. Even when $i = n - \omega(\log n)$ and s has been shifted so that the 2^{nd} least significant bit is in the i^{th} position, the probability of cycling can be bounded by $\frac{1}{Poly(n)}$ for any Polynomial in n. Here t is up to $O(\log n)$ number of bits and hence the probability of cycling is bounded above by $\frac{1}{Poly(n)}$ and hence we need to increase the number of queries by a certain amount corresponding to the drop in advantage due to cycling. Once again the details are omitted for brevity (see [4]) and will be included in an expanded version of this paper.

4 Discrete Logarithm Hides Almost n Bits

In this section we prove the simultaneous hardness of $n - \omega(\log n)$ lower bits of the index in modular exponentiation. Intuitively, given a generator g of a finite field of order p, and g^x for some x then we show that gaining *any information* about the trailing $n - \omega(\log n)$ bits is *hard*. Here hardness is with respect to the DLSE problem. In other words, for any prime p given a random generator g and a random element g^x of the finite field, any information on the relevant bits of x can be converted into an poly(n) algorithm to solve the DLSE problem. Now, the phrase *gaining any information* is rather vague, and this is clarified by the concept of simultaneous security which is defined below for any generic one-way function.

Definition 5. Let f be a one-way function. A collection of k bits, $B^k(x)$ is said to be simultaneously secure for f if $B^k(x)$ is easy to compute given x and for every Boolean predicate B an oracle which computes $B(B^k(x))$ correctly with probability greater than $\frac{1}{2}$ given only $f(x)$ can be used to invert f in Poly(n) time.

In this paper we will be employing a modified notion of simultaneous security relative to a possibly different one-way function.

Definition 6. Let f and f' be one-way functions. A k-bit predicate B^k is said to be f'-simultaneously hard if given $f(x)$, for every non-trivial Boolean predicate B on k bits, an oracle which outputs $B(B^k(x))$ can be used to invert f' in polynomial time. If B^k is a f' hard predicate then we say the bits of $B^k(x)$ are f'-simultaneously hard.

The above definition, although precise, is not easy to apply when understanding simultaneous security. A more working definition is provided in definition 7, phrased in the language of the discrete logarithm problem over a prime modulus.

Definition 7. The bits of the exponentiation function $g^x \bmod p$ at location $j \leq i \leq k$ are DLSE-simultaneously hard if the $[j,k]$ bits of the discrete logarithm of $g^x \bmod p$ are polynomially indistinguishable from a randomly selected $[j,k]$ bit string for random chosen $(g, p, g^x \bmod p)$. In addition any polynomial distinguishability will lead to an oracle which solves the DLSE problem in polynomial time.

Once again, proving polynomial indistinguishability of a group of bits as above is difficult. But the notion relative hardness helps alleviate this problem and in fact leads to a test of simultaneous security.

Definition 8. The i^{th} bit, $j \leq i \leq k$, of the function $g^x \bmod p$ is relatively hard to the right in the interval $[j,k]$ if no polynomial time algorithm can, given a random admissible triplet $(g, p, g^x \bmod p)$ and in addition given the $k-i$ bits of the discrete logarithm of g^x to its right, computes the i^{th} bit of the discrete logarithm of g^x with probability of success greater than $\frac{1}{2} + \frac{1}{poly(n)}$ for any polynomial $poly(n)$ where $n = \log p$.

Based on this definition, we have a test for simultaneous security. The statement of this test is the following fact.

Fact *Definitions* 7 *and* 8 *are equivalent.*

The proof of this equivalence is implied by the well-known proof of the universality of the next bit test due to Yao [25]. Now, using this fact and the intractibility of the DLSE problem we show that the trailing $n - \omega(\log n)$ bits are simultaneously hard.

Theorem 9. *The $n-\omega(\log n)$ trailing bits of $g^x \bmod p$ are simultaneously hard, with respect to the DLSE problem.*

Proof: Based on the above fact, it is sufficient to show that every trailing bit of x (given g and g^x) is relatively hard to the right in the interval $[2, n-\omega(\log n)]$. Following the definitions and theorem above we know that, in order to show simultaneous security, we are allowed to use only a weak oracle: given g^x, to predict the i^{th} bit of x , all the $i-1$ trailing bits of the unknown x should also be given to the oracle. Such a weak oracle may not work in general.

Assume the theorem is false. Then, for some $i \in [2, n-\omega(\log n)]$ there exists an oracle which when supplied with the trailing $i-1$ bits of a generic x succeeds

in predicting the i^{th} bit of x with advantage ϵ (where ϵ is $\frac{1}{poly(n)}$). Now pick an element $S = g^s$ where s is a short exponent. We can shift s to the left by squaring S the appropriate number of times. Now all the bits to the right of the i^{th} bit are zero. Since $0 \leq i < n - \omega(\log n)$ we can shift s by $i-1$ bits to the left without cycling. Recall, by cycling we mean the exponent exceeds $p-1$ and hence its remainder modulo $p-1$ replaces the exponent. Now the 2nd LSB of s rests on the i^{th} bit and we can run the oracle repeatedly by multiplying by $g^r \bmod p$ where r is a random number between 0 and $p-1$. In order to make sure that the probability of cycling is low we may have to set the $t = O(\log n)$ leading bits of s to zero which we can exhaustively guess and run the algorithm for each guess. Since we will continue to have an $\epsilon' \geq \epsilon - \frac{1}{t}$ advantage we can deduce the bit from the oracle in poly(n) time. We know the 2nd LSB of s in this manner. We set that bit to zero, and take the square root of the number. Of the two roots we should pick the one which is the quadratic residue because all the lower bits are zero to begin with and hence the square root should have a zero in the LSB. Now the next bit of s is in the i^{th} position and we can run the oracle repeatedly to discover this bit and so on to recover all the bits of s. At the end of the algorithm we have a candidate and we can see if $g^{candidate}$ equals S. If it does then we stop or else repeat the algorithm with another guess for t bits or different random numbers r. Note the oracle is very weak unlike the case for the individual bit oracle. The oracle here will tell you the i th bit with ϵ advantage provided you also supply all the $i-1$ bits to the right of i. However we are able to do this because all the bits to the right of the shifted s are known to be zero, since we started with a short exponent. Now we have shown that for every i such that $2 \leq i < \omega(\log n)$ we can use this weak oracle to discover s thus we have shown the trailing bits to be simultaneously hard provided the function $g^s \bmod p$ with s of size $\omega(\log n)$ is hard to invert.

5 Pseudo Random Bit Generator

In this section we provide the details of the new pseudo-random bit generator. In particular we extend the scheme used by Blum-Micali [4] to extract more bits. This is the same scheme that Long-Wigderson [16] used in their generator but their output consisted of $\log n$ bits per iteration. In our new scheme we produce $n - \omega(\log n)$ bits per iteration. Recall from section 2 that the Blum-Micali scheme used a mildly different definition of "bits". We use the same definition of bits as [10], but we do not encounter the difficulties they did in defining the generation scheme since our exponentiation induces a permutation on Z_p^*.

NEW GENERATOR *Pick a seed x_0 from Z_p^*. Define $x_{i+1} = g^{x_i} \bmod p$. At the i^{th} step ($i > 0$) output the lower $n - \omega(\log n)$ bits of x_i, except the least significant bit.*

5.1 Proof of Security

Suppose A is an ϵ-distinguisher of the l (l is poly in n) long output of our generator, then there is a (ϵ/l)-distinguisher for some output at the i th step. By appropriately running the generator then there is a (ϵ/l)-distinguisher for $n - \omega(\log n)$ bits of s_0. According to our definitions in the previous section, due to Yao [25], we can use a distinguisher to create a weak oracle which will tell us the i-th bit of s provided we also give it the rightmost $i-1$ bits of s.

Now we note that we can use this to discover s given $g^s \bmod p$ where s has $\omega(\log n)$ bits. We repeatedly invoke the "weak oracle" by setting $s_0 = g^s g^r$. Thus we can discover the i bit in poly(n) time. Using techniques shown in theorem 9 we can discover the entire s. So if the output sequence of our generator is ϵ-distinguishable then in poly(n) time we can discover s of our exponentiation function. Assuming it is intractable to invert the function $g^s \bmod p$ where s has $\omega(\log n)$ bits (i.e., short exponent) then the output sequence of our generator is polynomially indistinguishable.

6 Conclusion

We have shown that the discrete logarithm mod a prime p hides $n - \omega(\log n)$ bits by showing the simultaneous hardness of those bits. The hardness in this result is with respect to the discrete logarithm problem with short exponents, i.e., DLSE-simultaneously hard (as defined in section 2 of this paper). This allows us to extract $n - \omega(\log n)$ bits at a time for pseudo-random generation and other applications. As an example for n of size 1024 bits and s of size 128 bits this allows us to extract almost 900 bits per exponentiation. Spoken informally, we note that the security of this example is 2^{64} since it takes $O(2^{64})$ for the best known algorithm to crack a modular exponentiation with 128 bits. Also, if one desires more security at every step then we can decrease the number of bits extracted at every stage. This generator outputs the maximal number of bits from a single iteration. Extracting any more bits in any iteration leads to a prediction of bits - since we would then be hiding $O(\log n)$ or fewer bits and hence in polynomial number of guesses we would know the complete exponent in every iteration.

References

[1] W. Alexi, B. Chor, O. Goldreich and C. P. Schnorr, RSA/Rabin bits are 1/2+1/poly(log N) secure, *Proceedings of 25th FOCS*, 449–457, 1984.

[2] M. Ben-Or, B. Chor, A. Shamir, On the cryptographic security of single RSA bits, *Proceedings of 15th STOC*, 421–430, 1983.

[3] L. Blum, M. Blum, and M. Shub, A simple secure pseudo-random number generator, *SIAM J. Computing*, 15 No. 2:364–383, 1986.

[4] M. Blum, and S. Micali, How to generate cryptographically strong sequences of pseudo random bits, *SIAM J. Computing*, 13 No. 4:850–864, 1984.

[5] R. B. Boppana, and R. Hirschfeld, Pseudorrandom generators and complexity classes, *Advances in Computing Research*, 5 (S. Micali, Ed.), JAI Press, CT.

[6] U. S. Department of Commerce/ N. I. S. T, *Digital Signature Standard*, FIPS 186, May 1994.

[7] O. Goldreich, and L. A. Levin, A hard-core predicate for all one way functions, *Proceedings of 21st STOC*, 25–32, 1989.

[8] S. Goldwasser, and A. Micali, Probabilistic encryption, *Journal of Computer and Systems Science*, 28: 270-299, 1984.

[9] J. Hastad, R. Impagliazzo, L. A. Levin, and M. Luby, Construction of pseudo-random generator from any one-way function, *SIAM J. Computing*, to appear.

[10] J. Hastad, A. W. Schrift, and A. Shamir, The discrete logarithm modulo a composite modulus hides $O(n)$ bits, *Journal of Computer and System Sciences*, 47: 376–404, 1993.

[11] R. Impagliazzo, L. A. Levin, and M. Luby, Pseudo-random generation from one-way functions, *Proceddings of 20th STOC*, 12–24, 1988.

[12] B. S. Kaliski, A pseudo-random bit generator based on elliptic logarithms, *Advances in Cryptology - CRYPTO '86 (LNCS 263)*, 84–103, 1987.

[13] J. Kilian, S. Micali, and R. Ostrovsky, Minimum resource zero-knowledge proofs, *Procedings of 30th FOCS*, 474–489, 1989.

[14] D. E. Knuth, *The Art of Computer Programming (vol 3): Sorting and Searching*, Addison Wesley, 1973.

[15] N. Koblitz, Elliptic curve cryptosystems, *Mathematics of Computation*, 48:203–209, 1987.

[16] D. L. Long, and A. Wigderson, The discrete log hides $O(\log n)$ bits, *SIAM J. Computing*, 17:363–372, 1988.

[17] V. Miller, Elliptic curves and cryptography, *Advances in Cryptology - CRYPTO '85 (LNCS 218)*, 417–426, 1986.

[18] M. Naor, Bit commitment using pseudo-randomness, *Advances in Cryptology - CRYPTO '89 (LNCS 435)*, 128–136, 1989.

[19] P. van Oorschot, M. Wiener, On Diffie-Hellman key agreement with short exponents, *Advances in Cryptology - EUROCRYPT '96 (LNCS 1070)*, 332–343, 1996.

[20] R. Peralta, Simultaneous security of bits in the discrete log, *Advances in Cryptology - EUROCRYPT '85 (LNCS 219)*, 62–72, 1986.

[21] S. C. Pohlig, and M. E. Hellman, An improved algorithm for computing over $GF(p)$ and its cryptographic significance, *IEEE Trans. IT*, 24: 106–110, 1978.

[22] J. M. Pollard, Monte Carlo methods for index compution (mod p), *Mathematics of Computation*, 32, No. 143:918–924, 1978.

[24] U. V. Vazirani, and V. V. Vazirani, Efficient and secure pseudo-random number generators, *Proceedings of 25th FOCS*, 458–463, 1984.

[25] A. C. Yao, Theory and applications of trapdoor functions, *Proceedings of 23rd FOCS*, 80–91, 1982.

7 Appendix

In this section we discuss some extensions of our results which will be addressed in the future.

7.1 Improving Efficiency of Computations

Let us focus on the mechanics of the generator. We start with a finite field, and a generator g of its multiplicative cyclic group. Let x_0 be a secret seed. Then we define $x_{i+1} = g^{x_i}$ iteratively. The output of the generator are the trailing $n - \omega(\log n)$ bits of x_i for all $i > 0$, where $n = \log p$.

Although the number of bits generated per iteration is large, each iteration involves a large exponent and this could impact on the speed of the generator. Instead, we could start with p, g, and x_0 as earlier but at each stage we define $x_{i+1} = g^{s_i}$ where s_i = leading $\omega(\log n)$ bits of x_i. This will ensure that at each stage we are using short exponents and hence guarantee a significant speed up. This raises some interesting questions.

Question 10. Will this speed impact the security of the generator?

Note that when we restrict our exponents we no longer have a permutation. Hence the simple construction used here is inapplicable. A possible method of settling this problem is outlined in Hastad-*etal* in the context of discrete logarithms over composite moduli [10]. In particular, exploiting a certain hashing lemma proved in [11] they construct a *perfect extender* and the pseudo-random generation is achieved through repeated applications of the extender to a random seed.

Question 11. Are there efficient extenders which guarantee the same level of security (as the DLSE) but yet perform short exponent exponentiation at each step?

7.2 Discrete Logarithms in Abelian Groups

Let G be a finite Abelian group. Let $g \in G$ and let $y = g^x$ (where x is unknown and we are using the multiplicative notation to denote the group operation). The discrete logarithm problem in the subgroup generated by g asks for the value of x given g and y.

In this context, Kaliksi [12] has shown that under the intractibility assumption of the discrete log in the subgroup generated by g the individual bits of x are hard. In this paper the Blum-Micali notion of bits is employed, and the proof of individual hardness is based on a novel and new oracle proof technique. The main idea being, the identification of bits is based on a correlation function which automatically accommodates cycling and changes in bits due to randomization. In addition, he completely avoids the computation of square roots which is central to several of the other works on individual bit security. This paper also states that $\log n$ bits are simultaneously hard. Presumably, the techniques of Long-Wigderson once applied in the framework of generic Abelian groups yields this result.

Now, we note that assuming the discrete logarithm problem with short exponents is also hard in the chosen Abelian group our results on simultaneous hardness of the trailing bits may be applicable. This result will be very useful when applied to the group of points on an elliptic curve over a finite field.

7.3 Discrete Logarithms in Small Subgroups

The security of the digital signature standard (DSS) is based on the intractability of the discrete logarithm in small subgroups (DLSS). This leads to a natural question:

Question 12. Are there k-bit predicates attached to the input of the discrete exponentiation function that are simultaneously hard with respect to DLSS? In particular, is $k = n - \omega(\log n)$?