MIT/LCS/TM-244

HOW TO CONSTRUCT RANDOM FUNCTIONS

ODED GOLDREICH

SHAFI GOLDWASSER

SILVIO MICALI

November 1983

How to Construct Random Functions

Oded Goldreich Shafi Goldwasser Silvio Micali

Laboratory for Computer Science
M I T
Cambridge, MA 02139

Abstract

We assume that functions that are one-way in a very weak sense exist. We prove that in probabilistic polynomial time it is possible to construct **deterministic** polynomial time computable functions $g: \{1,...,2^k\} \rightarrow \{1,...,2^k\}$ that cannot be distinguished by any probabilistic polynomial time algorithm from a **random** function.

Loosely speaking, g provides random access to a $k2^k$-bit long pad whose entries record the outcome of independent coin flips.

This complexity theoretic result has many important applications in Cryptography, Protocols and Hashing.

Keywords : Randomness, random function, pseudo-random number generation.

The first author was supported in part by a Weizman Postdoctoral Fellowship. The second author was supported in part by the International Business Machines Corporation under the IBM/MIT Joint Research Program, Faculty Development Award agreement dated August 9, 1983.

1. Introduction

Measuring the randomness of a **string** has attracted much attention in the second half of this century. Kolmogorov [6] and Levin [8] measure the randomness of a string based on the length of its shortest description. Shamir [10], Blum and Micali [3], and Yao [12] introduce algorithmic predictability measures. Adleman [1] and Sipser [11] take an intermediate approach.

However, very little is known about measuring the randomness of a **function**. In this paper,

1) We introduce an algorithmic measure of the randomness of a function.

   (Loosely speaking, a function is random if any polynomial time algorithm, which asks for the values of the function at various points, cannot distinguish a computation during which it receives the true values of the function, from a computation during which it receives the outcome of independent coin flips.)

2) Based on a complexity theoretic assumption, we present functions that achieve maximum algorithmic randomness.

Let I_k denote the set of all k-bit strings, and H_k the set of all functions from I_k into I_k. Note that the cardinality of H_k is 2^{k2^k}. Thus to specify a function in H_k we would need $k2^k$ bits: an impractical task even for a moderately large k. Even more, assume that one randomly selects subsets $H_k^{\#} \subseteq H_k$ of cardinality 2^k so that each function in $H_k^{\#}$ has a unique k-bit index; then there is no polynomial time Turing Machine that, given k, the index of a function $f \in H_k^{\#}$ and $x \in I_k$, will evaluate $f(x)$. As there are many applications where we would like to use "random" functions, we must restrict ourselves to choose functions from a subset $F_k \subseteq H_k$ where the collection $F = \{F_k\}$ has the following properties:

1) There is a probabilistic polynomial algorithm A_F that on input k, will select with uniform probability a function in F_k.

2) There is a polynomial time Turing Machine that evaluates, on any input, any function in F_k, given its index.

3) No probabilistic algorithm that runs in time polynomial in k can distinguish the functions in F_k from random functions. (see section 3.1 for a precise definition).

Such a collection of functions F will be called a *poly-random* collection. Loosely speaking, despite the fact that the functions in F are easy to select and easy to

evaluate, they will exhibit, to an examiner with polynomially bounded resources, all the properties of randomly selected functions. E.g. for $f \in F$, it is computationally infeasible to find x and y such that $f(x)$ is easily computable from $f(y)$.

Assuming one-to-one one-way functions (formally defined in section 2.3) exist, we will prove that a poly-random collection of functions exists. The proof is constructive: given any one-to-one one-way function we will construct an algorithm A_F for a poly-random collection of functions F. The construction is in two steps: first, we use Yao's construction (Theorem 2 and appendix C) to transform a one-to-one one-way function into a high quality pseudo-random bit generator, called a Blum-Micali-generator (defined in section 2.1); next, we use any Blum-Micali-generator to construct a poly-random collection.

Our result provides:

1) Perfectly-secure (deterministic!) private key encryption **functions.**

   Such encryption functions exhibit to an examiner with polynomially bounded resources, all the properties of a random function. Moreover, for all message spaces with any probability distribution, the encrypted messages are provably secure against both chosen plaintext and chosen ciphertext attacks.

2) A powerful tool for cryptographic protocols design.

   Given a protocol that has already been proved secure when it uses random (encryption) functions, it is hard to find a provably secure implementation of it. This is due to the fact that "concrete" (encryption) functions may have unaccounted identities known to adversary who may successfully exploit them. (E.g. for two functions f and g, an adversary may find an x such that $f(g(x))=g(f(x))$). Our result suggest the following methodology.

   Design your protocol using random (encryption) functions so that you can prove that it is secure. The protocol will be executed by using (encryption) functions randomly selected from a poly-random collection. Such an implementation will be provably secure. Otherwise, an adversary, who efficiently breaks such an implementation, would provide an efficient way to distinguish random functions from functions in a poly-random collection.

3) "Ultimate" hashing: Choose and fix your hashing function. Now let an adversary choose the keys to be hashed and allow him to see the hashing values of the keys he picked. Yet, he cannot force collisions! This hashing scheme can be viewed as (and provides the first solution to) the two-parties authentication-tag scheme defined by Brassard [4].

4) A secure secret password distribution scheme, in which passwords are not stored, but can be quickly recomputed at any time.

2. Pseudo-Random Bit Generators

Throughout this paper, k will represent a security parameter. To all the algorithms in the paper k will be presented in unary. With P_1, P_2,... we will denote particular polynomials depending upon the desired design parameters. All polynomials in this paper are positive. The number of bits in the binary representation of an integer x will be denoted by $|x|$.

A *pseudo-random number generator* is a deterministic polynomial time algorithm that when given a k-bit *seed* outputs a $P_1(k)$-long *pseudo-random number sequence* $x_1,x_2,..,x_{P_1(k)}$. The pseudo-random sequence should pass "some" statistical tests. E.g. the sequence should have approximately as many 0's as 1's.

Many statistical properties of the familiar linear congruential pseudo-random number-sequence $x_{i+1}=a\cdot x_i+b \ (mod\ n)$ have been extensively studied by Knuth [7].

In a very nice paper [10], Shamir introduces unpredictability as a measure for the randomness of a sequence. He presents a pseudo-random number generator for which computing the $i+1$st number, x_{i+1}, from $x_1,x_2,..,x_i$ is as hard as inverting the RSA function [9].

Blum and Micali [3] point out that, even though x_{i+1} may be hard to compute from the preceding x_i's, every bit of x_{i+1} may be easy to predict with high probability. Even worse, consider a sequence of numbers such that the low-order bits of each x_i consist only of 0's. The next number in the sequence can still be hard to predict, but it would be against our intuition to call such a sequence pseudo-random. They thus introduce the notion of a *pseudo-random bit generator* that stretches a k-bit long seed into a $P_1(k)$-bit long sequence passing the Blum-Micali test when the seed is unknown.

2.1 Blum-Micali-Generators

The Blum-Micali Test

Let P' be a polynomial and $S = \bigcup_k S_k$ be a set of binary sequences such that all sequences $s \in S_k$ consist of $P'(k)$ bits. Let M be a probabilistic polynomial-time Turing Machine that on input the first i, $i < P'(k)$, bits of $s \in S_k$, outputs a bit b. Let $p_{k,i}^M$ denote the probability that b = the $i+1$st bit of s. The probability is taken over the possible choices of s in S_k and all M's internal coin tosses. We say that S passes the *Blum-Micali test* if for any polynomial P, for all probabilistic polynomial-time Turing Machines M, for all sufficiently large k and for all $i < P'(k)$,

$$p_{k,i}^M < \frac{1}{2} + \frac{1}{P(k)}.$$

Definition : Let G be a deterministic polynomial time algorithm that, on input $x \in I_k$, outputs a $P'(k)$-bit long sequence. Let $S = \bigcup_k S_k$, where S_k is the set of sequences output by G on seeds of size k. G is a *Blum-Micali-generator* if S passes the Blum-Micali-test. If so, S will be the set of the *Blum-Micali-sequences*.

Blum and Micali [3] presented a general algorithmic scheme that constructs Blum-Micali-generators from any "unapproximable predicate" with a "friendship function". They also gave the first implementation of their scheme and showed that the sequences it produced pass the Blum-Micali test if and only if the Discrete Logarithm Problem is "intractable". For more details, see appendix A.

Subsequently, Yao [12] showed that the Blum-Micali-test is a complete polynomial time statistical test (see next section for a precise definition). Therefore, Blum-Micali-sequences pass all polynomial time statistical tests.

2.2 Yao's Statistical Test

Let P' be a polynomial and $S = \bigcup_k S_k$ be a set of sequences, where S_k consists of $P'(k)$-bit sequences. A *polynomial time statistical test for strings* is a probabilistic polynomial time Turing Machine M that outputs only 0 or 1. We say that S *passes the test* M if for any polynomial Q, for all sufficiently large k:

$$|p_k^S - p_k^R| < \frac{1}{Q(k)}$$

where p_k^S denotes the probability that M outputs 1 on a randomly selected element of S_k and p_k^R the probability that M outputs 1 on a $P'(k)$-long random bit sequence.

Theorem 1 (Yao):

A set $S = \bigcup_k S_k$ of bit sequences passes the Blum-Micali-test if and only if it passes all polynomial time statistical tests for strings.

Equivalently,

The sequences generated by a Blum-Micali-generator pass all polynomial time statistical tests for strings.

The reader can derive a proof of Theorem 1 from the proof of Theorem 4.

2.3 Relevant Implementations of a Blum-Micali-Generator

Weak one-to-one one-way functions

The following definition is due to Yao. Let $f_k : I_k \rightarrow I_k$ be a sequence of permutations such that there is a polynomial-time algorithm that on input $x \in I_k$ computes $f_k(x)$. Let the function f be defined as follows: $f(x) = f_k(x)$ if $x \in I_k$. We say that f is a *one-to-one one-way function* if for all polynomial-time Turing Machines M there is a polynomial P such that, for all sufficiently large k

$$M(x) \neq f_k^{-1}(x) \text{ for at least a fraction } \frac{1}{P(k)} \text{ of the } x \in I_k.$$

Theorem 2 (Yao [12]): Given a weak one-to-one one-way function, it is possible to implement Blum-Micali-generators.(1)

For details on Yao's construction, see appendix C.

Blum, Blum and Shub [2] present an interesting implementation of the Blum-Micali scheme and prove that the generated sequences pass the Blum-Micali test if and only if deciding Quadratic Residuosity modulo a *Blum –integer*(2) whose factorization is not known, is "intractable". Equivalently, if and only if *squaring modulo a Blum-integer* is a weak one-to-one one-way function.

---

(1) As a matter of fact, one can prove that there exist a weak one-to-one one-way function if and only if there exists an unapproximable predicate with a relative friendship function.
(2) A Blum integer is an integer of the form $p_1 \cdot p_2$ where p_1 and p_2 are distinct primes both congruent to 3 mod 4.

An interesting feature of their generator is that knowledge of the seed and of the factorization of the modulus allows easy access to each bit in an exponentially long bit string (i.e. if k denotes the length of the seed and $|i| \leq k$, then the i-th bit in the string can be computed in poly(k) time). This is due to the **special** weak one-to-one one-way function on which the security of their generator is based. However, this exponentially long bit string **may not** appear "random". Blum, Blum and Shub only prove that any **single** polynomially long interval of **consecutive** bits in the string passes the Blum-Micali-test. Indeed, it may be the case that, given the first k bits in the string and k bits starting from the $2^{\sqrt{k}}$-th one, it is possible to easily compute any other bit in the string. For details on the Blum, Blum Shub generator, see appendix B.

The first implementation of a Blum-Micali-generator which passes the Blum-Micali test if and only if deciding Quadratic Residuosity modulo Blum-integers is "intractable", is due to Yao [12]. However, his generator does not possess the "easy-access" feature as in the Blum,Blum and Shub generator. Goldwasser, Micali and Tong [5] propose an implementation of a Blum-Micali-generator that passes the Blum-Micali test if and only if factoring a special class[3] of Blum-integers is "intractable". Their generator has the same "easy-access" feature as in the Blum, Blum and Shub generator. (Again, their generator is based on a special weak one-to-one one-way function and the "randomness" of bits exponentially apart is not proved).

In the next section, we show that if **any** weak one-to-one one-way function exists, we can construct sets S_k^{ex} containing 2^k-bit strings and provide "easy-access" to each bit in any string $s \in S_k^{ex}$. We **prove** that these "easily-accessed" bits are indistinguishable from bit sequences chosen at random. More precisely, let M be a probabilistic polynomial time Turing Machine capable of calls to an oracle O_s for a 2^k-bit string $s \in S_k^{ex}$. On input k and access to the oracle O_s, the j-th query of M is a k-bit string i_j; the oracle's answer is b_{i_j}, the i_j-th bit in s. Without loss of generality, we can assume that M will ask $P_2(k)$ queries on input k. The set of strings S_k is defined as follows. A $P_2(k)$-bit string s' belongs to S_k if it is a sequence of oracle-answers during a computation of M on input k and access to an oracle O_s for some $s \in S_k^{ex}$. Then $S = \bigcup_k S_k$ passes the Blum-Micali test.

(3) in this class $n = p_1 \cdot p_2$ such that $p_1 \equiv p_2 \equiv 3 \bmod 8$ or $p_1 \equiv p_2 \equiv 7 \bmod 8$. Half of the Blum-integers are of this form.

3. Constructing random functions

In this section we show how to construct functions that pass all "polynomially bounded" statistical tests.

A *collection of functions*, F, is a collection $\{F_k\}$, such that for all k and all $f \in F_k$, $f: I_k \rightarrow I_k$.

3.1 Polynomial Time Statistical Tests For Functions

A *polynomial time statistical test for functions* is a probabilistic polynomial time algorithm T that, given an input k and access to an oracle O_f for a function $f: I_k \rightarrow I_k$, outputs either 0 or 1. Algorithm T can query the oracle O_f only by writing on a special query-tape some $y \in I_k$ and will read the oracle answer, $f(y)$, on a separate answer-tape. As usual, O_f prints its answer in one step.

Let $F = \{F_k\}$ be a collection of functions. We say that F passes the test T if for any polynomial Q, for all sufficiently large k:

$$|p_k^F - p_k^R| < \frac{1}{Q(k)}$$

where p_k^F denotes the probability that T outputs 1 on input k and access to an oracle for a function f randomly chosen in F_k. p_k^R is the probability that T outputs 1 when given the input parameter k and access to an oracle O_f for a function f randomly picked in H_k (i.e. a random function).

We now exhibit a collection F that passes all polynomial time statistical tests, under the assumption that there exists a weak one-to-one one-way function.

3.2 The Construction of F

Assume that there exists a weak one-to-one one-way function g. Then, by Theorem 2, one can construct a Blum-Micali-generator G. Recall that G is a function defined on all bit strings such that if $x \in I_k$, $G(x) = b_1^x, \ldots, b_{P_1(k)}^x$. Let $S = \bigcup_k S_k$ be defined as follows. S_k is the set of all the first $2k$ bits output by G on seeds of length k. Then S passes all polynomial size statistical tests for strings.

Let $x \in I_k$ be a seed for G. $G_0(x)$ denotes the first k bits output by G on input x; $G_1(x)$ denotes the next k bits output by G. Let $\alpha = \alpha_1\alpha_2\cdots\alpha_t$ be a binary string. We define $G_{\alpha_1\alpha_2\cdots\alpha_t}(x) = G_{\alpha_t}(\cdots(G_{\alpha_2}(G_{\alpha_1}(x)))\cdots)$.

Let $x \in I_k$. The function f_x: I_k -> I_k is defined as follows:

For $y = y_1 y_2 \cdots y_k$, $f_x(y) = G_{y_1 y_2 \cdots y_k}(x)$.

Define $F_k = \{ f_x \}_{x \in I_k}$ and $F = \{F_k\}$.

Note that a function in F_k needs not be one-to-one.

The reader may find it useful to picture a function $f_x: I_k$-> I_k, as a full binary tree of depth k with k-bit strings stored in the nodes and edges labelled 0 or 1. The k-bit string x will be stored in the root. If a k-bit string s is stored in an internal node, v, then $G_0(s)$ is stored in v's left-son, v_l, and $G_1(s)$ is stored in v's right-son, v_r. The edge (v,v_l) is labelled 0 and the edge (v,v_r) is labelled 1. The string $f_x(y)$ is then stored in the leaf reachable from the root following the edge path labelled y.

3.3 The Poly-Randomness of F

Note that the collection F just defined satisfies conditions (1) and (2) of a poly-random collection. The following theorem shows that also condition (3) is satisfied.

Main Theorem (Theorem 3): The collection of functions F passes all polynomial time statistical tests for functions.

Proof: Let T be a polynomial time test for functions. Let p_k^F (p_k^R) be the probability that T outputs 1 when given the input parameter k and access to an oracle O_f for a function f randomly picked in F_k (H_k).

Assume, for contradiction, that for some polynomial Q and for infinitely many k, $|p_k^F - p_k^R| > \frac{1}{Q(k)}$.

The "random" oracle O_i is defined as follows for each i, $0 \leq i \leq k$.

Let $y = y_1 y_2 \cdots y_k$ be a query to O_i. Then O_i responds as follows:

If y is the first query with prefix $y_1 \cdots y_i$, O_i selects a string $r \in I_k$ at random, stores the pair ($y_1 \cdots y_i$, r), and answers $G_{y_{i+1} \cdots y_k}(r)$.

Else , O_i retrieves the pair ($y_1 \cdots y_i$, v) and answers $G_{y_{i+1} \cdots y_k}(v)$.

(In terms of the tree representation of f_x, O_i stores random k-bit strings in the nodes of level i. The nodes of higher level will contain k-bit strings deterministically computed as in the previous section based on the actual values in level i).

For $0 \leq i \leq k$, p_k^i is defined to be the probability that A outputs 1 when given k as input and access to the oracle O_i.

Note that $p_k^0 = p_k^F$ and that $p_k^k = p_k^R$.

We will reach a contradiction by exhibiting a polynomial statistical test for strings, B, so that S will not pass B. On input k, whenever k is such that $|p_k^0 - p_k^k| > \frac{1}{Q(k)}$, B finds an i ($0 \leq i < k$) such that $|p_k^i - p_k^{i+1}| > \frac{1}{k \cdot Q(k)}$. Algorithm B does so by running a Monte-Carlo experiment using T as a subroutine.

Let now R_k be the set of all $2k$-bit long strings and S_k be as in section 3.2. Algorithm B gives k as input to algorithm T and answers T's oracle queries consistently using the set U_k as follows. (U_k is either R_k or S_k).

Assume T writes $y = y_1 \cdots y_k$ on the oracle tape.

If y is the first query with prefix $y_1 \cdots y_i$, B picks at random, in the set U_k, $u = u'u''$ ($u'u''$ is the concatenation of u' and u'', and $|u'| = |u''| = k$). B stores the pairs ($y_1 \cdots y_i 0$, u') and ($y_1 \cdots y_i 1$, u''). B answers

$$G_{y_{i+2} \cdots y_k}(u') \text{ if } y_{i+1} = 0 \text{ and}$$
$$G_{y_{i+2} \cdots y_k}(u'') \text{ if } y_{i+1} = 1.$$

Else B retrieves the pair ($y_1 \cdots y_{i+1}$, v) and answers $G_{y_{i+2} \cdots y_k}(v)$.

Note that, when $U_k = S_k$, B simulates the computation of T with oracle O_i. When instead $U_k = R_k$, B simulates the computation of T with oracle O_{i+1}. Since T's output differs, in a measurable way, on these two computations for infinitely many k, letting B output the same bit that subroutine T does, we have reached a contradiction.

Qed

Remark: The Main Theorem as well as the poly-randomness of F hold when applying the construction of sec. 3.2 to any Blum-Micali-generator G.

3.4 Generalizations

In some applications, we would like to have random functions from $I_{P_3(k)} \to I_{P_4(k)}$. E.g. in hashing we might want functions from I_k into I_{10}. We meet this need by introducing the collection $F = \{F_k\}$ defined as follows: For $x \in I_k$, $f_x \in F_k$ is a function from $I_{P_3(k)}$ into $I_{P_4(k)}$ defined as follows. Let $y = y_1 \cdots y_{P_3(k)}$. Define $f_x(y) = \Gamma_{P_4(k)}[G_{y_1 \cdots y_{P_3(k)}}(x)]$, where $\Gamma_{P_4(k)}(z)$ are the first

$P_4(k)$ bits output by G when fed input $z \in I_k$, where G is a Blum-Micali generator.

Such an F is also a poly-random collection: properties (1) and (2) trivially hold, and property (3) can be proved in a way similar to the Main Theorem.

3.5 A Complete Polynomial Time Statistical Test For Functions

We present a particular polynomial-time statistical test for functions and show that a collection of functions passes this test if and only if it passes all polynomial time statistical tests for functions. This is a natural generalization of Theorem 1.

The Query-and-Learn Test

Let $F = \{F_k\}$ be a collection of functions. Let A be a probabilistic Turing Machine capable of oracle calls as in section 3.1. On input k and access to an oracle O_f for a function $f \in F_k$, algorithm A carries out a computation during which it queries O_f about $x_1,...,x_j$. Then algorithm A outputs $x \in I_k$ such that $x \neq x_1, \ldots, x_j$. This x will be called the *chosen exam.* At this point A is disconnected from O_f and is presented a $y \in I_k$. The string y is, with equal probability, either f(x) or a random k-bit string. After some computation, A outputs either 0 or 1 and halts. We say that A has *queried-and-learned* if either A outputs 1 and $y = f(x)$, or A outputs 0 and $y \neq f(x)$.

Let P be a polynomial. We say that A *P-queries-and-learns* F if, for all sufficiently large k, the probability that A guesses correctly on input k and access to an oracle O_f for $f \in F_k$, is at least $\frac{1}{2} + \frac{1}{P(k)}$. The probability is taken over all the possible choices of $f \in F_k$, y, and A's internal coin tosses.

Let T_k^A denote the maximum number of steps of A on input k and access to an oracle O_f for $f \in F_k$. We say that F passes the *Query-and-Learn* test if, for any polynomial P and all probabilistic Turing Machine A that P-predicts F, T_k^A grows faster than any polynomial in k.

Theorem 4 : F passes the Query-and-Learn test if and only if F passes all polynomial time statistical tests.

Proof : The "if part" can be easily proved noticing that for any polynomial P, no Turing Machine A that is given as input k and access to an oracle for a function $f \in H_k$ can query-and-learn.

We now prove the "only if" part. Assume that there exists a polynomial time statistical test A such that $|p_k^F - p_k^R| > \frac{1}{P_5(k)}$, where p_k^F and p_k^R are defined, as in section 3.1, relative to A. Without loss of generality, given k as input, A always asks $P_6(k)$ oracle queries and all queries are different. We will construct a probabilistic polynomial time Turing Machine, B, that $P_5(k) \cdot P_6(k)$-queries-and-learns F.

For $f \in F_k$, O_f^i is formally defined as follows:

Let x_j be the j-th query presented to O_f^i.

If $j \leq i$, O_f^i answers with $f(x_j)$,

Else O_f^i answers with a random k-bit string.

Define p_k^i to be the probability that A outputs 1 when given access to the oracle O_f^i. Here the probability is taken over all $f \in F_k$ and all possible computations of A. Note that $p_k^0 = p_k^R$ and $p_k^{P_6(k)} = p_k^F$.

On input k, B finds an i $(0 \leq i < P_6(k))$, such that $|p_k^i - p_k^{i+1}| > \frac{1}{P_5(k) \cdot P_6(k)}$, by running a Monte-Carlo experiment.

B uses A as a subroutine as follows: B starts A on the same input k it receives. B answers the first i queries of A using the oracle O_f. When A asks for its $i+1$st query, x_{i+1}, B outputs x_{i+1} as its chosen exam. Upon receiving y (equal to $f(x)$ or a randomly selected k-bit string), B writes y on A's answer-tape. B answers all subsequent queries of A by randomly selecting k-bit strings. B outputs the same bit that A does, and halts.

The reader can convince himself that B $P_5(k) \cdot P_6(k)$-queries-and-learns F.

Qed

Appendix A: The Blum-Micali Scheme for Constructing Blum-Micali-Generators

A.1 The general scheme

The scheme starts from an "unapproximable" predicate, B, and a "friendship function", f, for B.

Let $B=\{B_k\}$ be a collection of predicates, where B_k: $X_k \subseteq I_k$-> {0,1} and there is a probabilistic polynomial-time Turing Machine M that, on input k, selects an $x \in X_k$ with uniform probability. We say that B is an *unapproximable predicate* if for any polynomial P and for all polynomial-time Turing Machines M, for all sufficiently large k,

$$M(x)=B_k(x) \text{ for at most a fraction } \frac{1}{2}+\frac{1}{P(k)} \text{ of the } x \in I_k.$$

Let $f=\{f_k\}$ be a set of functions. For all k, f_k is a permutation on X_k. f is a *friendship* function for the unapproximable predicate B if there exist a polynomial time algorithm that computes $B(f(x))$ for all k and all inputs $x \in X_k$.

Let $x,y \in X_k$. One can interpret $B_k(y)$ as the secret of y which is hard to guess given y. One can interpret $f(x)$ as the best friend of x. x "knows" the secret of his best friend.

The Blum-Micali scheme takes B and f as above and a polynomial Q and constructs a Blum-Micali-generator as follows:

given $x \in X_k$, output the $Q(k)$-bit long sequence

$$B_k(f_k^{Q(k)}(x)),\ B_k(f_k^{Q(k)-1}(x)),\ ....,\ B_k(f_k(x)).$$

The proof that these sequences pass the Blum-Micali-test can be found in [3].

A.2 The Discrete Logarithm Implementation

Let p be a prime. The set of integers [1, p-1] forms a cyclic group under multiplication mod p. Such group is denoted by Z_p^*. Let g be a generator for Z_p^*. The function $f_{p,g}$: x -> $g^x \bmod p$, for $x \in Z_p^*$, defines a permutation on Z_p^* computable in Poly($|p|$) Time. The *Discrete Logarithm Problem* (DLP) with inputs p,g and y consists in finding the $x \in Z_p^*$ such that $g^x \bmod p = y$. Such an x will be denoted by $index_g(y)$ whenever no ambiguity may arise about p. Let Q be a polynomial. A probabilistic Turing Machine M *Q-solves the DLP mod a k-bit prime p if it halts in expected $Q(k)$* time and $M(p,g,y)=index_g(y)$ for any generator g for Z_p^* and any $y \in Z_p^*$.

The Intractability Assumption for the DLP:

For any polynomials P and Q and for all probabilistic Turing Machines M, for all sufficiently large k, M Q-solves the DLP mod p for at most a fraction $\frac{1}{P(k)}$ of the k-bit primes p.

Blum and Micali show that, from the intractability assumption for the DLP, one can extract an unapproximable predicate and its relative frienship function. (Loosely, $B(g^x \bmod p) = 1$ if $x \leq \frac{p-1}{2}$ and 0 otherwise, will be the unapproximable predicate and $g^x \bmod p$ its relative friendship function).

Appendix B: The Blum, Blum and Shub Implementation

In terms of the Blum-Micali scheme, Blum Blum and Shub use the Quadratic Residuosity Problem, modulo a Blum-integer whose factorization is not known, as the unapproximable predicate. They look at the quadratic residues, q, mod n. Each such a q, has exactly one square root, r, that is itself a quadratic residue. The secret of q is defined to be the the last bit in the binary representation of r. The friendship function is squaring mod n (a permutation over the quadratic residues mod n when n is a Blum integer). The i-th bit in the sequence generated on input x is the secret of $x^{2^i} \bmod n$.

The generator, knowing the factorization of n, can compute the i-th bit in the sequence, i< n, in time polynomial in |n|. First it computes $y = 2^i \bmod \varphi(n)$ by repeated squaring, then computes $x^y = x^{2^i} \bmod n$ again by repeated squaring, where $\varphi(n)$ is Euler's totient function.

Appendix C: Yao's construction

The following shows how to implement the Blum-Micali scheme using any weak one-to-one one-way function (i.e. how to construct an unapproximable predicate and its relative friendship function).

Let f be a weak one-to-one one-way function. For each i, $1 \leq i \leq k$, the predicates B_k^i: I_k-> {0,1} ,1 ≤ i ≤ k, are defined as follows:

$B_k^i(x)$ is the i-th bit of $f_k^{-1}(x)$.

Loosely speaking, since f_k is hard to invert for at least a fraction $\frac{1}{P_7(k)}$ of the $x \in I_k$, then for some i ($1 \leq i \leq k$) B_k^i is hard to compute for at least a fraction $\frac{1}{k \cdot P_7(k)}$ of the $x \in I_k$. Let $P_8(k) = k \cdot P_7(k)$ and $P_9(k) = k \cdot (P_8(k))^3$.

Let $\bigoplus$ denote the "exclusive or" function. Let X_k be the cartesian product of $P_9(k)$ copies of I_k, i.e. $X_k = (I_k)^{P_9(k)}$.

Let $x = (x_1, \ldots, x_{P_9(k)}) \in X_k$. The unapproximable predicate $\{ B_k : X_k \rightarrow \{0,1\} \}$ is defined as follows: $B_k(x) = \bigoplus_{i=1}^{k} (\bigoplus_{j=1}^{(P_8(k))^3} B_k^j (x_{(i-1)(P_8)^3+j}))$, for all k.

The friendship function $\{ g_k : X_k \rightarrow X_k \}$ is so defined

$$g_k(x) = (f_k(x_1), \ldots, f_k(x_{P_9(k)})).$$

Acknowledgements (so far)

Our greatest thanks go to Benny Chor for sharing with us much of the labor involved in this research.

We are particularly grateful to Ron Rivest who assisted us all along with many insights and precious criticism.

Leonid Levin relentesly encouraged us to get this result. Thank you Lenia!

Many thanks to Michael Ben-Or, Tom Leighton, Albert Meyer and Mike Sipser for several helpful discussions.

Oded Goldreich would like to thank Dassi Levi for existing.

References

[1] L. Adleman, *Time, space and randomness*, MIT/LCS/TM -131, 1979

[2] L. Blum, M. Blum and M. Shub, *A simple secure psuedo random number generator*, Advances in Cryptology: Proc. of CRYPTO-82, ed. D. Shaum, R.L. Rivest and A.T. Sherman. Plenum press 1983, pp 61-78.

[3] M. Blum and S. Micali, *How to generate cryptographically strong sequences of pseudo-random bits*, Proc. 23rd IEEE Symp. on Foundations of Computer Science, 1982, pp 112-117

A better is version available from authors.

[4] G. Brassard, *On computationally secure authentication tags requiring short secret shared keys*, Advances in Cryptology: Proc. of CRYPTO-82, ed. D. Shaum, R.L. Rivest and A.T. Sherman. Plenum press 1983, pp 79-86.

[5] S. Goldwasser, S. Micali and P. Tong, *Why and how to establish a private code on a public network*, Proc. 23rd IEEE Symp. on Foundations of Computer Science, 1982, pp 134-144

[6] A. Kolmogorov, *Three approaches to the concept of "the amount of information"*, Probl. of Inf. Transm. **1/1**, 1965

[7] D. Knuth, *The Art of Computer Programming: Seminumerical Algorithms*, Vol. 2, Addison-Wesley, 1981.

[8] L. Levin, *Various measures of complexity for finite objects (axiomatic descriptions) Soviet Math. Dokl. 17/2 (1976) pp 522-526*

[9] R. Rivest, A. Shamir, and L. Adleman, *A method for obtaining digital signatures and public key cryptosystems*, Commun. ACM, vol. 21, Feb. 1978, pp 120-126

[10] A. Shamir, *On the Generation of Cryptographically Strong Pseudo-random Sequences*, 8th International Colloquium on Automata, Languages, and Programming, Lect. Notes in Comp. Sci. **62**, Springer Verlag, 1981

[11] M. Sipser, *A complexity theoretic approach to randomness*, Proc. 15th ACM Symp. on Theory of Computing, 1983, pp 330-335.

[12] A.C. Yao, *Theory and applications of trapdoor functions*, Proc. 23rd IEEE Symp. on Foundations of Computer Science, 1982, pp 80-91.