How to Construct Random Functions

(Extended Abstract)

Oded Goldreich  Shafi Goldwasser  Silvio Micali
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA  02139

Abstract

This paper develops a constructive theory of randomness for functions based on computational complexity.

We present a deterministic polynomial-time algorithm that transforms pairs \((g,r)\), where \(g\) is any one-way (in a very weak sense) function and \(r\) is a random \(k\)-bit string, to polynomial-time computable functions \(f_r: \{1, \ldots, 2^k\} \rightarrow \{1, \ldots, 2^k\}\). These \(f_r\)'s cannot be distinguished from random functions by any probabilistic polynomial time algorithm that asks and receives the value of a function at arguments of its choice.

The result has applications in cryptography, random constructions and complexity theory.

1. Introduction

Measuring randomness has attracted much attention in the second half of this century. However most of the previous work focused on measuring the randomness of strings.

In Kolmogorov Complexity ([Kol], [Sol], [ZL], [Ch], [L2], [L3], [L4], [ML], [Sch] and [Ga]) the measure of randomness of a string is the length of its shortest description: randomness is an inherent property of individual strings. This approach is non-constructive and far from being applicable to pseudo-random string generation. (Interesting generalizations of Kolmogorov Complexity have been considered in [A], [Si], [H] and [W].)

In [BM] and [Y] (following a result of [Sh]) a constructive approach to the randomness of strings is introduced based on computational complexity. In this approach a set of strings is random if elements randomly selected in it retain, with respect to polynomial-time computation, properties of elements randomly selected in the set of all strings.

In this paper we further develop this latter approach by introducing a constructive theory of randomness for functions. In particular,

1. We introduce a computational complexity measure of the randomness of functions.
   (Loosely speaking, we call a function random if no polynomial time algorithm, asking for the values of the function at arguments of its choice, can 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. Notice the analogy with the Turing Test for intelligence.)
2. Assuming the existence of one-way functions, we present an algorithm for constructing functions that achieve maximum randomness with respect to the above measure.

Our result solves, and was motivated by, an open problem of [BBS].
Organization of the paper

In the rest of this section we informally discuss the notion of a poly-random collection: a set of easy to select and to evaluate functions that achieve randomness with respect to polynomial-time computation. We compare this new notion with the previously considered notions of one-way functions and Cryptographically Strong Pseudo-Random Bit generators (CSPRB generators). In section 2 we briefly recall the basic definitions and results about CSPRB generators and the Blum Blum Shub open problem. In section 3 we formally define poly-random collections and show how to construct a poly-random collection given any one-way function. In section 4 we characterize poly-random collections as extremely hard prediction problems. In section 5 we briefly discuss various applications of poly-random collections. We conclude this paper with some reflections on the internal coherence of polynomial-randome...
Choose and fix \( f \in F_k \). Let a probabilistic poly(k) time algorithm \( A \) ask for the value of \( f \) on polynomially many (in \( k \)) arguments of its choice: \( y_1, y_2, \ldots, y_k \). Then let \( A \) choose an argument \( x \) (\( x \neq y_i \), for all \( i \)'s) as an exam. If \( A \) is now given two numbers in random order, one of which is \( f(x) \) and the other a random \( k \)-bit number, it cannot guess which of the two is \( f(x) \) with probability greater than \( 1/2 \).

Not only that \( f(x) \) cannot be computed from the values of \( f \) at other arguments, but it cannot even be recognized when given! The above test is a complete characterization of poly-random collections (see section 4).

1.3. Comparison with CSPRB Generators

CSPRB generators are deterministic programs that stretch a (random) \( k \)-bit long seed to a \( k' \)-bit long (pseudo-random) sequence that is indistinguishable from a \( k' \)-bit long truly random sequence for some constant \( i > 0 \) (see section 2.1). Their existence has interesting implications with respect to probabilistic computation.

Performing a probabilistic polynomial-time computation that requires \( k^t \) random bits is trivial if we are willing to flip \( k^t \) coins. Interestingly, CSPRB generators guarantee the same result of the computation by flipping only \( k \) coins.

We now address the problem of efficiently simulating more complex probabilistic computations: computations with a random oracle.

A random oracle (see Bennett and Gill [BG]) is a special case of a random function: it associates the result of a single coin toss to each string. Notice that computing with a random oracle has advantages over computing with a coin. The bit associated with each string \( x \), not only is random, but does not change in time. That is, if one asks twice for the bit associated with string \( x \), then he gets the same (random) result. The advantages of computing with a random oracle are clarified by all the applications listed in section 5.

It is trivial to simulate a random oracle that is queried on \( k^t \) strings if one is willing to use \( O(k^{t+1}) \) bits of storage:

For each query \( q \), generate a random (or pseudo-random) bit \( b \) and store some encoding of the pair \((q, b)\) so as to be able to recognize whether a query occurred before and give the same answer.

Clearly, if the queries cannot be compressed (as for random queries) then this simple simulation would require at least \( k^{t+1} \) bits of storage. An interesting feature of poly-random collections is that they guarantee the same result of any computation with a random oracle for \( k \)-bit strings (by using only \( k \) coin flips and) by storing only \( k \) bits! This can be done by randomly selecting and storing a \( k \)-bit index specifying a function in a poly-random collection.

Poly-random collections allow to share randomness in a distributed environment

An additional advantage of poly-random collections is that they enable many parties to efficiently share a random function \( f \) in a distributed environment. By sharing \( f \) we mean that if \( f \) is evaluated at different times by different parties on the same argument \( x \), the same value \( f(x) \) will be obtained. Such sharing is efficient as it can be achieved by only flipping \( k \) coins, using \( k \) bits of storage (per party) and without exchanging any messages at all. Again, each party (processor) will simply have in memory a common, randomly selected \( k \)-bit string specifying a function \( f \) in a poly-random collection.

1.4. Conventions

All definitions and results in this paper are stated with respect to the Turing Machine computational model. The results can also be stated and proved in terms of circuit complexity.

Also, all definitions and results are stated with respect to the uniform probability distribution. The results can be stated and proved with respect to more general probability distributions.
The parameter $k$, when given as input to any algorithm discussed in this paper, will be presented in unary.

Let $A$ be a multiset with distinct elements $a_1, \ldots, a_n$ occurring with multiplicities $m_1, \ldots, m_n$ respectively. Then $|A| = \sum_{i=1}^{n} m_i$. By writing $a \in_R A$ we mean that the element $a$ has been randomly selected from the multiset $A$, i.e., an element occurring in $A$ with multiplicity $m$ is chosen with probability $\frac{m}{|A|}$.

2. CSPRB Generators

In this section we recall some of the basic definitions and results concerning Cryptographically Strong Pseudo-Random Bit generators (CSPRB generator).

2.1. The Notion of a CSPRB Generator

Improving a result of Shamir [Sh], Blum and Micali [BM] introduced the notion of a Cryptographically Strong Pseudo-Random Bit generator (CSPRB generator). Let $P$ be a polynomial. A CSPRB generator, $G$, is a deterministic poly($k$)-time program that stretches a $k$-bit long randomly selected seed into a $P(k)$-bit long sequence (called a CSPRB sequence) that passes all next-bit-tests:

Let $P$ and $S = \bigcup_{k} S_k$ be as above. A polynomial time statistical test for strings is a probabilistic polynomial-time algorithm $T$ that, on input a $P(k)$-bit string, outputs only 0 or 1.

The multiset $S$ passes the test $T$ 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 $T$ outputs 1 on $s \in_R S_k$ and $p_k^R$ the probability that $T$ outputs 1 on a randomly selected $P(k)$-long bit sequence.

Yao [Y] shows that by substituting $\epsilon$ by $\frac{1}{\text{poly}(k)}$ in the definition of the next-bit-test the following theorem can be proved.

Theorem 1 (Yao [Y]): A multiset $S = \bigcup_{k} S_k$ of bit-sequences passes the next-bit-test if and only if it passes all polynomial-time statistical tests for strings.

Thus, CSPRB sequences pass all polynomial-time statistical tests for strings. Theorem 4 generalizes the above theorem. The reader can derive a proof of Theorem 1 from the proof of Theorem 4.

2.3. Implementations of CSPRB Generators

Blum and Micali [BM] presented an algorithmic scheme for constructing CSPRB generators based on a general complexity theoretic assumption (a sketch can be found in the Appendix). They also presented the first instance of their scheme based on a specific assumption: the intractability assumption of the discrete logarithm problem (DLP). Namely, if the next bit in the sequences produced by their generator could be predicted with probability greater than $\frac{1}{2} + \epsilon$, then there would exist a poly($k, \epsilon^{-1}$) algorithm for solving the DLP for a fraction $\epsilon$ of all primes of length $k$.

Other instances of CSPRB generators based on various number theoretic assumptions appeared in [Y] [BBS] [GMT'] [ICS] [VV1] [LW] [ACGS].
More generally, Yao [Y] showed how to obtain CSPRB generators if any (weak) one-way permutation is given. Let us be more formal.

**Definition (Yao):** Let $D_k \subseteq I_k$. Let $f_k : D_k \rightarrow D_k$ be a sequence of permutations and let the function $f$ be defined as follows: $f(x) = f_k(x)$ if $x \in D_k$. $f$ is said to be a *one-to-one one-way function* if

1) $f$ is polynomial-time computable.

2) $f$ is (moderately) hard to invert: there exists a polynomial $Q$ such that for every polynomial-time algorithm $A$ and for all sufficiently large $k$, $A(x) \neq f_k^{-1}(x)$ for at least a fraction $\frac{1}{Q(k)}$ of the $x \in D_k$.

3) There exists a probabilistic polynomial-time algorithm that, on input $k$, select an $x \in D_k$ with uniform probability distribution.

**Theorem 2 (Yao [Y]):** Given a weak one-to-one one-way function, it is possible to construct CSPRB generators.

A sketch of the construction used by Yao is given in the Appendix.

Levin [L5] pointed out that Theorem 2 still holds with respect to "locally one-way" functions, a notion weaker than the above defined notion of a one-way permutation. More over he exhibits a function that is locally one-way if any locally one-way function exists. An informal sketch of Levin's definition is given in the Appendix.

2.4. CSPRB Generators With Direct Access.

Blum, Blum and Shub [BBS] present an interesting CSPRB generator whose sequences pass all polynomial time statistical tests if and only if *squaring modulo a Blum-integer*\(^{(1)}\) is a weak one-to-one one-way function.\(^{(2)}\)

Notice that, even though a CSPRB sequence generated with a $k$-bit long seed consists of polynomially many (in $k$) bits, a CSPRB generator and a seed $s$ define an infinite (ultimately periodic) bit-sequence $b_0, b_1, \ldots$. An interesting feature first present in Blum Blum Shub's generator is that knowledge of the seed and of the factorization of the modulus allows direct access to each of the first $2^k$ bits. I.e. if $\log i < k$, the $i$th bit in the string, $b_i$, can be computed in poly($k$) time. This is due to the special weak one-way permutation on which the security of their generator is based. However, this directly-accessible 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 all polynomial time statistical tests for strings. Indeed, it may be the case that, given $b_1, \ldots, b_k$ and $b_{2^k+k+1}, \ldots, b_{2^k+k}$ it is easy to compute any other bit in the string.

The Blum Blum Shub open problem consists of whether direct access to exponentially far away bits in their pseudo-random pad is a "randomness preserving" operation. This problem has also been discussed by Angluin and Lichtenstein [AL].

Notice that there is a natural one-to-one correspondence between "randomness preserving" directly-accessible $k \cdot 2^k$-bit long strings and random functions from $I_k$ to $I_k$. By constructing a poly-random collection $F = \{F_k\}$, we virtually construct $k \cdot 2^k$-bit strings $\{y = f(1)f(2)\ldots f(2^k)\}_F$ which can be directly accessed in a "randomness preserving" manner. This practically solves the Blum Blum Shub problem in a strong sense since we construct poly-random collections not only if squaring modulo a Blum-integer is a one-way permutation, but given any one-way permutation.

\(^{(1)}\) A Blum integer is an integer of the form $p_1p_2$ where $p_1$ and $p_2$ are distinct primes both congruent to 3 mod 4.

\(^{(2)}\) This generator has been proved [BBS] to be cryptographically strong based on the intractability of deciding Quadratic Residuosity modulo a Blum-integer. Recently, it has been pointed out [VV2] that, the results in [ACGS] imply that this generator is cryptographically strong based on a weaker assumption: the intractability of factoring Blum-integers.
3. Constructing Poly-Random Collections

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 $k$ as input 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 $k$:

$$|p_k^T - p_k^H| < \frac{1}{Q(k)}$$

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

The above definition can be interpreted as follows. A function $f$ is "judged" to be random depending on its input-output relation. The test $T$ consists of two phases. First it gathers information about $f$ by getting $f$'s values at arguments of its choice. Then it outputs its "verdict": 0 (if it "thinks" that $f \in F_k$) or 1 (if it "thinks" that $f \in H_k$). If the collection $F$ passes the test $T$, then the output of $T$ given oracle $O_f$ gives no information on whether $f \in F_k$ or $f \in R H_k$. In either case $T$ will output 1 with essentially the same probability.

Passing all polynomial-time statistical tests for functions is an extremely general randomness criterion. This can be intuitively argued as follows. Should some efficient algorithm $A$ find any dependencies among the selected input-output pairs of $f \in R F_k$, it can be converted to a statistical test $T_A$ that will halt outputting 0 (i.e. judging that $f \in R F_k$) when detecting these dependencies. Since such dependencies cannot be found when $f \in R H_k$, the collection $F = \{F_k\}$ will not pass the test $T_A$.

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 $F$

We construct poly-random collections given any CSPRB generator $G$ that stretches a seed $x \in I_k$ into a 2$k$-bit long sequence, $G(x) = b_1^x \ldots b_2^x$. By Theorem 2, such generator $G$ can be constructed given any one-way permutation.

Let $S_k$ be the multiset of the 2$k$-bit sequences output by $G$ on seeds of length $k$. Recall that $S = \bigcup_{k} S_k$ passes all polynomial-time statistical tests for strings.

Let $x \in I_k$. By $G(x)$ we denote the first $k$ bits output by $G$ on input $x$ i.e. $G(x) = b_1^x \ldots b_k^x$. By $G(x)$ we denote the next $k$ bits output by $G$. I.e. $G_1(x) = b_{k+1}^x \ldots b_{2k}^x$. Let $\alpha = \alpha_1 \alpha_2 \ldots \alpha_r$ be a binary string. We define $G_{\alpha_1 \alpha_2 \ldots \alpha_r}(x) =$ $G_{\alpha_r \ldots \alpha_2 \alpha_1}(G_{\alpha_1}(x)) \ldots$.

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

$$f_x(y) = G_{y_1 y_2 \ldots y_k}(x).$$

Set $F_k = \{f_x\} \forall x \in I_k$ 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 $f_x: I_k \rightarrow I_k$ 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 $G(s)$ is stored in $v$'s left-son, $v_l$, and $G(s)$ 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 \( y \). See figure 1.

![Tree Diagram](image)

**fig 1**

Efficiency Consideration

Let \( T_k \) denote the (worst case) number of steps used in the computation of the CSPKB sequence \( G(x) \) on input \( x \in I_k \). Clearly, computing \( f_x(y) \) on inputs \( x \) and \( y \) can be done in at most \( k \cdot T_k \) steps. Thus, the efficiency of the evaluation of a function in our poly-random collection is reduced to the efficiency of the underlying CSPRB generator. The latter question is referred to in the Appendix.

3.3. The Poly-Randomness of \( F \)

Note that the collection \( F \) just defined satisfies conditions 1 (indexing) and 2 (poly-time evaluation) of a poly-random collection. The main theorem shows that condition 3 (pseudo-randomness) is also satisfied. We prove the main theorem using a (new) variant of Yao’s statistical test.

**Definition (population test):** Let \( P \) and \( P_1 \) be polynomials and \( S_k = \bigcup_k S_k \) be a set of sequences, where \( S_k \) consists of \( P(k) \)-bit sequences. A polynomial-time population test for strings is a probabilistic polynomial-time algorithm \( T \) that, on input \( P_1(k) \) strings each \( P(k)-\)bit long, outputs either 0 or 1. We say that \( S \) passes the test \( T \) if for any polynomial \( Q \), for all sufficiently large \( k \):

\[
|p_k^S - p_k^B| < \frac{1}{Q(k)}
\]

where \( p_k^S \) denotes the probability that \( T \) outputs 1 on \( P_1(k) \) randomly selected strings in \( S_k \) and \( p_k^B \) denotes the probability that \( T \) outputs 1 on \( P_1(k) \) random bit-strings each of length \( P(k) \).

**Lemma:** A set of bit-sequences \( S = \bigcup_k S_k \) passes all polynomial-time statistical tests if and only if it passes all polynomial-time population tests.

The proof of the Lemma can be easily obtained by techniques similar to the ones used for proving Theorem 4.

**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 \( p_k^F(p_k^B) \) be the probability that \( T \) outputs 1 when given the input parameter \( k \) and access to an oracle \( O_f \) for a function \( f \in F_k \) (\( f \in F_H \)).

Assume, for contradiction, that for some polynomial \( Q \) and for infinitely many \( k \),

\[
|p_k^F - p_k^B| > \frac{1}{Q(k)}.
\]

Let us consider computations of \( T \) in which, instead of an oracle \( O_f \), an algorithm \( A_i \) answers \( T \)’s queries. For \( 0 \leq i \leq k \) and for each computation of \( T \) with oracle \( A_i \), \( A_i \) is defined as follows.

Let \( y = y_1 y_2 \cdots y_k \) be a query to \( A_i \). Then \( A_i \) responds as follows:

If \( y \) is the first query with prefix \( y_1 \cdots y_i \), \( A_i \) selects a string \( r \in I_k \) at random, stores the pair \( (y_1 \cdots y_i, r) \), and answers \( G_{y_1 \cdots y_i}(r) \).

Else, \( A_i \) retrieves the pair \( (y_1 \cdots y_i, v) \) and answers \( G_{y_1 \cdots y_i}(v) \).

(In terms of the tree representation of \( f_x \), \( A_i \) 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 subsection based on the actual values in level \( i \).

For \( 0 \leq i \leq k \), \( p_k^i \) is defined to be the probability that \( T \) outputs 1 when given \( k \) as input and access to the oracle \( A_i \).
Note that $p_k^0 = p_k^F$ and that $p_k^k = p_k^H$.

We will reach a contradiction by exhibiting a polynomial-time population test for strings, $A$, so that $S$ will not pass $A$.

Let $k$ be such that $|p_k^0 - p_k^i| > \frac{1}{Q(k)}$, without loss of generality let $p_k^0 - p_k^k > \frac{1}{Q(k)}$. On input $k$, with probability greater than $1 - \frac{1}{8k \cdot Q(k)}$, $A$ finds an $i$ ($0 \leq i < k$) such that $p_k^i - p_k^{i+1} > \frac{1}{2k \cdot Q(k)}$.

Algorithm $A$ does so by running a polynomial-time Monte-Carlo experiment using $T$ as a subroutine.

Let now $R_k$ be the set of all $2k$-bit long strings $S_k$ be as in section 3.2.

Algorithm $A$ 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 $y = y_1...y_k$ on the oracle tape.

If $y$ is the first query with prefix $y_1...y_i$, $A$ picks at random, in the set $U_k$,

$u = u_0u_1$ ($u_0u_1$ is the concatenation of $u_0$ and $u_1$, and $|u_0| = |u_1| = k$). $A$ stores the pairs $(y_1...y_i, u_0)$ and $(y_1...y_i, u_1)$. $A$ answers

$G_{y_1...y_i, u_0}(u_0)$ if $y_{i+1} = 0$ and

$G_{y_1...y_i, u_1}(u_1)$ if $y_{i+1} = 1$.

Else $A$ retrieves the pair $(y_1...y_{i+1}, v)$ and answers $G_{y_1...y_{i+1}, v}(v)$ if $i \leq k - 2$ and $v$ if $i = k - 1$.

Note that, when $U_k = S_k$, $A$ simulates the computation of $T$ with oracle $A_1$. When instead $U_k = R_k$, $A$ simulates the computation of $T$ with oracle $A_{i+1}$. Since $T$'s output differs, in a measurable way, on these two computations for infinitely many $k$, letting $A$ output the same bit that subroutine $T$ does, we have reached a contradiction.

QED

3.4. Generalized Poly-Random Collections

Let $P_1$ and $P_2$ be polynomials. In some applications, we would like to have random functions from $I_{P_1(k)} \rightarrow I_{P_2(k)}$ (e.g. in hashing we might want functions from $I_{1000}$ into $I_{10}$). We meet this need by constructing a generalized poly-random collection $\{F_k^{P_1,P_2}\}$. The modified construction can be simply described in terms of two different CSPRB generators: $G$ as above and $G'$ mapping $k$ random input bits to $P_2(k)$ pseudo-random bits. For $x \in I_k$ the function $f_x \in P_1^{P_1,P_2}$ is defined as follows: on input $y \in I_{P_1(k)}(I_{P_2}(y) = G'(G_y(x)))$. By a proof similar to the one of the Main Theorem one can prove that the collection $\{F_k^{P_1,P_2}\}$ possesses properties (1), (2) and (3) of poly-random collections.

3.5. A Universal Statistical Test

Our definition of a poly-random collection consists of passing all polynomial-time statistical tests for functions. In fact it is enough to consider one universal polynomial-time statistical test for functions (a collection will pass this universal test if and only if it passes all tests). Essentially, this universal test will guess a program of a statistical test and then execute it. Further details will be given in the full version of this paper. Similarly, universal tests exist also for all the other classes of tests mentioned in this paper.

4. Prediction Problems and Poly-Random Collections

Physics may be viewed as a prediction problem. This problem may seem to be tractable if

1) There is an a priori guarantee that the "laws of nature" are "simple" (the functions one needs to predict can be computed in polynomial time once some trapdoor information is given).

2) It is possible to conduct selected experiments (one is given temporary access to an oracle for the function).

3) The goal is only to approximately predict the "laws of nature" (the function).
Note that the ability to perform selected experiments (query the function) is a much more powerful tool than learning from given examples. The power of this tool is hereafter demonstrated.

**An Example:**

Consider the set $C$ of all integers product of two primes of equal length. No efficient algorithm is known for factoring the integers $n \in C$; furthermore, the question whether such an efficient algorithm exists constitutes one of the oldest computational problems. For $n \in C \cap I_k$, we define the following functions $f_n : I_k \rightarrow I_k$ as follows: $f_n(x) =$ the smallest square root of $x^2 \mod n$ if $\gcd(x, n) = 1$, and 0 otherwise. These functions are "simple", i.e. are polynomial-time computable if the trapdoor information (the factorization of $n$) is given. If the factorization of $n$ is not part of the input then these $f_n$'s may be hard to compute: Rabin [Ra] proved that factoring $n \in C$ is probabilistic polynomial-time reducible to computing $f_n(y)$ on input $n$ and $y$. However, a simple extension of Rabin's proof shows that (even when the index $n$ is not a part of the input), these "simple" functions can be computed after being given temporary access to an oracle $(O_A)$ which on query $q$ returns the value of the function at argument $q$ (i.e. $f_n(q)$). In fact, after asking the oracle a few questions, $n$ can be easily computed and factored.

One might therefore wonder whether for all "simple" functions $f$, temporary access to an oracle for $f$ may enable one to hereafter easily compute $f$. We answer this question negatively in a strong sense, under the assumption that one-way permutations exist. Given any one-way permutation $g$, we construct "simple" functions $f^g$ that cannot be predicted (even in a weaker sense than discussed above).

**Remark:** The $f^g$'s we construct cannot be weakly predicted after temporary access to an oracle for $f$, even if the one-way permutation $g$ at the base of the construction can be easily computed after temporary access to an oracle for $g$.

**Formal Setting**

Let $F$ be a collection of functions satisfying conditions 1 (indexing) and 2 (poly-time evaluation) of a poly-random collection. Let $A$ be a probabilistic polynomial-time algorithm 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 $x_1, \ldots, x_f$. Then, algorithm $A$ outputs $x \in I_k$ such that $x \neq x_1, \ldots, x_f$. This $x$ will be called the chosen exam. At this point $A$ is disconnected from $O_f$ and is presented $f(x)$ and $y \in I_k$ in random order. $A$ is asked to guess which of the two is $f(x)$.

Let $Q$ be a polynomial. We say that $A$ $Q-$Queries-and-Learns $F$ if on input $k$ the probability that $A$ guesses correctly which-is-which is greater than $\frac{1}{2} + \frac{1}{Q(k)}$.

We say that $F$ cannot be polynomially-inferred if there exists no probabilistic polynomial time algorithm $A$ and polynomial $Q$ such that $A$ can $Q-$query-and-learns $F$.

Note that polynomially-inferring the collection $F$ is a much more easy task than predicting $f \in F_k$ in the sense discussed in the beginning of this section.

**Theorem 4:** $F$ can not be polynomially-inferred if and only if $F$ passes all polynomial-time statistical tests for functions.

**Proof:** Assume, on one hand, that $F$ can be polynomially-inferred. Let $Q$ be a polynomial and $A$ be a probabilistic algorithm that $Q-$queries-and-learns $F$. Clearly, $A$ can not $Q-$queries-and-learns $H = \{H_k\}$. Thus $A$ can be used to construct a statistical test $T_A$ which distinguishes $F$ from $H$ as follows:

On input $k$, $T_A$ initiates $A$ with input $k$ and answers $A$'s queries by forwarding them to the oracle $O_f$ ($f \in F_k$ or $f \in H_k$). When $A$ asks to be examined on the exam $x$, $T_A$ queries $O_f$ on $x$, picks randomly $y \in I_k$ and returns $y$ and $f(x)$ to $A$ in random order. If $A$ guess right the identity of $f(x)$ then $T_A$ outputs 1; otherwise $T_A$ outputs 0. Note that the
probability that $T_A$ outputs 1 is exactly $\frac{1}{2}$ when $f \in R H_k$; while it (the probability $T_A$ outputs 1) is greater than $\frac{1}{2} + \frac{1}{Q(k)}$ when $f \in R F_k$.

Assume, on the other hand, that $F$ does not pass the statistical test $T$. Then there exist a polynomial, $Q$, such that $|p^F_k - p^{H}_k| > \frac{1}{Q(k)}$, where $p^F_k$ and $p^{H}_k$ are defined, as in section 3.1, relative to $T$. Let $P$ be a polynomial. Without loss of generality, given $k$ as input, $T$ always asks $P(k)$ oracle queries and all queries are different. Without loss of generality assume that $p^F_k - p^{H}_k > \frac{1}{Q(k)}$. We will construct a probabilistic polynomial time algorithm, $A_T$, that $2 \cdot P(k) \cdot Q(k)$ queries and learns $F$.

For $f \in F_k$, the pseudo-oracle $O_f^k$ is formally defined as follows:

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

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

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

Define $p^F_k$ to be the probability that $T$ outputs 1 when given access to the oracle $O_f^k$. Here the probability is taken over all $f \in F_k$ and all possible computations of $T$. Note that $p^F_k = p^{H}_k$ and $p^{P(k)}_k = p^F_k$.

On input $k$ with probability $1 - \frac{1}{8P(k)Q(k)}$, $A_T$ finds an $i$ $(0 \leq i < P(k))$, such that $p^F_k - p^{P(k)}_k > \frac{1}{2 \cdot P(k)Q(k)}$, by running a Monte-Carlo experiment.

$A_T$ uses $T$ as follows: $A_T$ starts $T$ on the same input $k$ it receives. $A_T$ answers the first $i$ queries of $T$ using the oracle $O_f^k$. When $T$ asks for its $i+1$st query, $x_{i+1}$, $A_T$ outputs $x_{i+1}$ as its $(A_T$'s) chosen exam. Upon receiving $y = f(x_{i+1})$ and $y \in R I_k$, $A_T$ chooses randomly $z \in \{f(x_{i+1}), y\}$ and writes $z$ on $T$'s answer tape (i.e. as the $i+1$st oracle answer). $A_T$ answers all subsequent queries of $T$ by randomly selecting $k$-bit strings. If $T$ outputs 1 then $A_T$ guesses that $z \in R I_k$; otherwise then $A_T$ guesses that $z = f(x_{i+1})$.

Qed

5. Applications

In this section we briefly discuss some of the problems which can be solved using a poly-random collection. Our solutions are the first which are proved secure under the general assumption that one-way permutations exist. A detailed discussion of these applications is presented in [GGM2]. Brassard [B] has pointed out that application 5.2 could be possible if the BBS open problem had a positive solution.

5.1. Storageless Distribution of Secret Identification Numbers

Consider a distributed system with one or more servers and many users each having a distinct name. The problem is to distribute, to each user, a secret user-identification number (ID) such that the ID is verifiable by the servers but infeasible to compute by any other user. An example of such a problem is assigning calling card numbers to telephone customers.

Our solution uses the poly-random collection $F = \{F_f\}$ in order to assign random secret IDs to the users. First, the servers jointly pick a $f \in R F_k$ in secrecy, and each server stores the $k$-bit index of $f$. (This is all the servers need to store!) Then, every user $X$ in the system is assigned an ID $f(X)$.

Note that each server can verify whether a given number is the ID of Alice, by computing $f(Alice)$. However, it is infeasible for any set of users to compute the ID of any user not in the set.

5.2. Message Authentication and Time-Stamping

Using poly-random collections it is possible, for the first time, to construct deterministic, memoryless, authentication schemes which are highly robust, as discussed in the following concrete setting.

Assume that all the employees of a large bank communicate through a public network. As an adversary may be able to inject messages, the employees need to authenticate the messages they sent to each other (e.g. "transfer sum $S$ from account $A$ to account $B$"). A solution may consist of appending to
the message $m$ an authentication tag which is hard to compute by an adversary. In particular, we propose the following. Let all employees have access to authentication machines which compute a function $f_s$ in a poly-random collection. The tag associated with a message $m$ is $f_s(m)$. We can tradeoff security for the length of the tag. For example, if one uses only the first 20 bits of $f_s(m)$ as an authentication tag, then the chance that an adversary could successfully authenticate a message is about 1 in a million.

To avoid playback of previously authenticated messages, it is common practice to use time-stamps. Namely, authenticate $m$ concatenated with date it was sent. So far, time-stamping was only a heuristic as an adversary who sees the message $m$ authenticated with date $D$ could conceivably authenticate $m$ with another date (say $D+1$). Using our solution for message authentication, time-stamping makes playback provably hard. This is the case as for a random function $f(x)$ is totally unrelated to $f(x+1)$, and therefore the same holds (with respect to polynomial-time adversaries) for poly-random collections.

Another threat to the Bank’s security is the loyalty of its own employees. They have the authenticating computer at their disposal and can use it to launch a chosen message attack against the scheme, so that when they are fired they can forge transactions. Our message authentication scheme remains secure even when the employees are not trustworthy, if each message to be authenticated is automatically time stamped by the computer. An employee who leaves the bank, after having widely experimented with the machine, will not be able to authenticate even one new message.

5.3. An Identify Friend Or Foe System

The members of an exclusive society are well known for their brotherhood spirit. Upon meeting each other, anywhere in the world, they extend hospitality, favors, advice, money etc. Naturally, they face the danger of imposters trying to take advantage of their generosity. Thus, upon meeting each other, they must execute a protocol for establishing membership. As they meet in public places (busses, trains, theatre), they must be careful not to yield information that can lead to future successful impersonations. They go around carrying pocket computers on which they may make calculations.

Clearly a password scheme will not suffice in this context, as the conversations are public. An interactive identification scheme is needed where the ability to ask questions does not enable future successful impersonations. Note that the questions that $A$ may ask member $B$, must be picked from an exponential range to prevent an active imposter from asking all possible questions, receiving all possible answers and thereafter successfully impersonating as a member (or to prevent a passive imposter from having a non-negligible probability of being asked a question that he overheard the answer to).

Using our poly-random collection, we can fully solve this problem. Let the president of the society choose a $k$-bit random string $s$, specifying a function $f_s$ in a poly-random collection. Each member receives a computer which calculates $f_s$. When member $A$ meets $B$, he asks $z ?$ where $z \in B I_k$. Only if $B$ answers $f_s(z)$, will member $A$ be convinced that $B$ is a member. In addition, if the computers that calculate $f_s$ can be manufactured so that they can not be duplicated, then losing a computer does not compromised the security of the entire scheme; it just allows one non-member to enjoy the privileges of the society.

Note that using any of the "known" one-way functions in the role of $f_s$ may not work here, since ability to ask questions may compromise the security of the entire society as for the case of Rabin’s function (see section 4).

5.4. Dynamic Hashing

Poly-random collections from long bit-strings to short bit-strings constitute very good hash functions. Note that such hash functions have advantages, with respect to polynomial-time computation, over the Universal Hashing scheme suggested by Carter and
Wegman [CW]. In their scheme the hash functions perform well with respect to a fixed a priori probability distribution for the keys. Our scheme performs well even if an adversary does not fix his key distribution a priori, but can dynamically change the key distribution during the hashing process upon seeing the hash function values on previous keys.

Such a scheme may be useful in applications where accessing memory is more expensive than evaluating the hash functions.

5.5. Speeding-up CSPRBC Generator

Assume that $G$ is an "inherently-sequential" CSPRBC generator. That is, on input a $k$-bit seed, computing the $i$-th bit in the output sequence of $G$ takes time $i\cdot T(k)$. Assume that our application (see example below) requires to compute the bits in the $\text{poly}(k)$-bit long sequence output by $G$ in arbitrary order, and that only $O(k)$ bits of storage are available. Then it would be desirable to access the bits in the pseudo-random sequence "directly" rather than "sequentially".

Using $G$ to construct a function in a poly-random collection, we effectively construct an exponentially (in $k$) long pad each bit of which can be accessed in time $k\cdot 2k\cdot T(k)$.

Example: protecting a data base

Suppose that one would like to store a huge data base on a public computer while maintaining the information contained in it private. To achieve this one may encrypt each of the records of the data base, place the encrypted records on the public computer and store only a relatively small secret key on his home computer. Suppose that encryption has been done by using the sequence output by a CSPRBC generator as a one-time pad. In this case the private key consists of the input seed to the generator. To retrieve the information on a record one has to access the segment of the pseudo-random pad used for encrypting it.

6. Concluding Remarks

The Notion of Polynomial Pseudo-Randomness

A CSPRBC generator can be viewed as a tool for simulating a source of truly random coin tosses. Consider the following source of randomness: a probabilistic polynomial-time Turing Machine (TM) that, on input the security parameter $k$, outputs polynomially many bits. Using a CSPRBC generator, one can construct a probabilistic polynomial-time TM that, on input $k$, simulates the source using only $k$ internal coin tosses. The simulation is perfect with respect to all polynomially bounded observers.

Let us now consider interactive sources. An Interactive Source is an interactive, probabilistic, polynomial-time TM which answers queries presented to it by an inspection machine (another interactive, probabilistic, polynomial-time TM). The interaction consists of a sequence of interleaved queries and answers. In this extended abstract, we considered a special case of interaction and showed how such interactive sources can be perfectly simulated by a poly-random collection, using only $k$ internal coin tosses and $k^c$ bits of storage (for some fixed $c$). We believe that this case captures the notion of polynomial pseudo-randomness.

A Tool for Cryptographic Protocol Design

As shown in the applications mentioned in section 5.1, 5.2 and 5.3, the poly-random collections are a powerful tool in cryptographic protocol design. The following methodology for protocol design appears fruitful. First, design a protocol which uses truly random functions, and prove it correct. Then, replace the truly random functions by functions randomly selected from a poly-random collection. This implementation will provably maintain all properties of the original protocol with respect to polynomially bounded adversaries. Also note that if two independent random functions are substituted by two functions randomly selected from a poly-random collection, then the latter will be totally uncorrelated (as
the former ones). This provable independence is
very useful in protocol design.

Recently, Luby and Rackoff [LR] used poly-
random collections to construct collections of poly-
random permutations. This result leads to the con-
struction of ideal private key cryptosystems.

Acknowledgements

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

Leonid Levin relentlessly encouraged us to get
this result and, once obtained, helped up to better
understand it in the course of so many inspiring dis-
cussions. Thank you Leonid!

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

We are very grateful to Albert Mayer for
quickly rescuing us from a fearful dead end.

Many thanks to Michael Ben-Or, Steve Cook,
Tom Leighton, Gary Miller, Charles Rackoff and
Mike Sipser for several helpful discussions.

Oded Goldreich would like to thank Dassi Levi
for her existence.

Appendix

Sufficient Conditions for Constructing CSPRB Gen-
erators

Let $D_k \subseteq I_k$ and $B_k : D_k \to \{0, 1\}$. Let $g_k$ be a permutation over $D_k$. Let $D = \bigcup_k D_k$, $B = \{B_k\}$ and
g = $\{g_k\}$. Blum and Micali [BM] showed that CSPRB

generators can be constructed under the following
conditions:

1) The Domain is accessible: there exists a proba-
abilistic polynomial-time algorithm that on input
$k$, chooses $x \in D_k$ with uniform probability dis-
tribution.

2) There exists a polynomial-time algorithm that
on input $k$ and $x \in D_k$, computes $g_k(x)$.

3) Let $A$ be a probabilistic polynomial-time algo-

rithm and $Q$ be a polynomial. Then for all

sufficiently large $k$:

$$A(x) \neq B_k(x) \text{ for at least for a fraction} \frac{1}{2} - \frac{1}{Q(k)} \text{ of the } x \in D_k.$$

4) There exists a polynomial-time algorithm that
on input $k$ and $x \in D_k$, computes $B_k(g_k(x))$.

Note that the above conditions imply that $g$ is a
one-way permutation as defined in section 2.3. Yao
[Y] showed that the existence of a one-way permuta-
tion (over an accessible domain) is a sufficient condi-
tion for constructing CSPRB generators.

A Sketch of Yao’s Construction

Yao’s construction [Y] can be viewed as a

method to construct $B$ and $g$ as above, when given

any one-way permutation $h = \{h_k\}$ over the accessible
domain $E = \bigcup_k E_k$. Recall that no polynomial algo-

rithm can invert $h$ without being mistaken on a $\frac{1}{k^c}$

fraction of the domain, for some constant $c$, when $k$ is

sufficiently large.

Set $D_k$ to be the cartesian product of $k^{2q}$
copies of $E_k$.

Set $g_k(x_1x_2\cdots x_{k2q}) = h_k(x_1)h_k(x_2)\cdots h_k(x_{k2q})$, where $x_j \in E_k$.

Set $B_k^{(i)}(x)$ to be the $i$th bit of $h_k^{-1}(x)$, where $x \in E_k$ and

$$B_k(x_1x_2\cdots x_{k2q}) = \bigcup_{i=1}^{k^{2q-1}} B_k^{(i)}(x_{k2q-i+1}x_{k2q-i}).$$

Then $\bigcup_k \{g_k\}$ and $\{B_k\}$ defined above satisfy all

4 conditions of the Blum-Micali scheme (a proof of

this appears in [GI]).

A sketch of Levin’s definition

A function (algorithm) $A$ is $(t,e)$—one-way on

an input $x \in I_k$ if

1) There exists an $i$ such that $A^i(x) = x$. 476
2) The computation of $A$ on input $x$ takes time at most $t(k)$.

3) An optimal inverting algorithm (for $A$) requires at least time $e(k)$ in order to compute and verify $x$ on input $A(x)$. (The existence of an optimal inverting algorithm for NP-search problems was pointed out in [L6].)

A function (algorithm), $A$, is locally one-way if there exist a polynomial $t$ and a function $e$ which grows faster than any polynomial such that $A$ is $(t,e)$-one-way on at least a $\frac{1}{t(k)}$ fraction of the inputs in $I_k$.

Levin has pointed out a universal algorithm, $u$, (with $k^2$ time bound) which is locally one-way, unless no function is locally one-way. Furthermore, in case $u$ is locally one-way it is $(t_u,e_u)$-locally one-way, where $t_u(k) = k^2$ and $e_u$ grows faster than any polynomial. Note that, $u$ can be used in Yao's construction (of a CSPRB generator) instead of the given one-way permutation.

On the Running Time of the known CSPRB Generators

The running time of CSPRB generators should be compared with respect to the intractability assumption on which they are based. Basing a generator on any weak one-way permutation, though very appealing from a theoretical point of view, seems to have a practical drawback: slow running time (see Yao's construction above). It seems that in order to get fast generators, one would have to rely on stronger assumptions (i.e. on the intractability of specific problems). Let us consider the following two assumptions:

1) The Intractability Assumption for the Discrete Logarithm Problem (DLA): It is infeasible to compute discrete logarithms modulo all but a negligible fraction of the primes. (For a precise formulation of DLA see [BM].)

2) The Intractability Assumption for the Integer Factorization Problem (FA): It is infeasible to factor all but a negligible fraction of the Blum Integers. (For a precise formulation of FA see [BG].)

The fastest CSPRB generator known under DLA is presented in [LW]. It produces $O(\log k)$ bits of output at the cost of one modular exponentiation of $k$-bit integers.

The fastest CSPRB generators known under FA can be obtained by the results in [ACGS]. In particular, $O(\log k)$ bits of output can be produced at the cost of one modular multiplication of $k$-bit integers.

References


To appear in SIAM Jour. on Computing


[GMR] S. Goldwasser, S. Micali and R.L. Rivest, A "Paradoxical" Signature Scheme, these proceedings


[Si] M. Sipser, A Complexity Theoretic Approach to


