How to Construct Random Functions

ODED GOLDREICH, SHAFI GOLDWASSER,
AND SILVIO MICALI

Massachusetts Institute of Technology, Cambridge, Massachusetts

Abstract. A constructive theory of randomness for functions, based on computational complexity, is developed, and a pseudorandom function generator is presented. This generator is a deterministic polynomial-time algorithm that transforms pairs (g, r), where g is *any* one-way 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.

Categories and Subject Descriptors: F.0 [**Theory of Computation**]: General; F.1.1 [**Computation by Abstract Devices**]: Models of Computation—*computability theory*; G.0 [**Mathematics of Computing**]: General; G.3 [**Mathematics of Computing**]: Probability and Statistics—*probabilistic algorithms*; *random number generation*

General Terms: Algorithms, Security, Theory

Additional Key Words and Phrases: Cryptography, one-way functions, prediction problems, randomness

> *I have set up on a Manchester computer a small programme using only 1000 units of storage, whereby the machine supplied with one sixteen figure number replies with another within two seconds. I would defy anyone to learn from these replies sufficient about the programme to be able to predict any replies to untried values.*
>
> A. TURING

1. *Introduction*

What is meant by saying that certain functions "behave randomly"?

In this paper we provide a precise answer to the above question. We then present an efficient way to construct functions that behave randomly, if one-way functions exist. We conclude by demonstrating applications of our construction.

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*.

O. Goldreich was supported in part by a Weizmann postdoctoral fellowship; S. Goldwasser was supported in part by an IBM faculty development award (1983) and National Science Foundation grant DCR 85-09905; and S. Micali was supported by a National Science Foundation grant DCR 84-13577 and an IBM faculty development Award (1984).

Authors' present addresses: O. Goldreich, Computer Science Department, Technion, Haifa 32000 Israel; S. Goldwasser and S. Micali, Laboratory for Computer Science, Massachusetts Institute of Technology, 545 Technology Square, Cambridge, MA 02139.

In Kolmogorov Complexity [10, 13, 22, 24–26, 30, 34, 37, and 42] the measure of randomness of a string is the length of its *shortest description.* Kolmogorov-randomness is an inherent property of *individual* strings. This approach is nonconstructive and far from being applicable to pseudorandom string generation. Notably, the set of Kolmogorov-random strings is nonrecursive.

Interesting generalizations of Kolmogorov complexity have been considered in [1], [19], [36], and [38]. Here a string s is "random" if it cannot be produced by a program that is both efficient (polynomial-time) and shorter than s. The approach remains far from pseudorandom number generation. In fact no efficient algorithm that uses less than k truly random bits can output a k-bit string random in the above sense.

Recently, a constructive approach to the randomness of strings, based on computational complexity, has emerged [8, 41]. In this approach a *set* S of strings is *polynomial random* (*poly-random*) if programs that run in polynomial time lead to identical results when fed either with elements randomly selected in S or with elements randomly selected in the set of all strings. This approach is constructive in the following way: There exists a deterministic polynomial-time algorithm that, upon input of a k-bit string, outputs a poly(k)-bit string, such that, if one-way functions exist, then the set of all output strings is poly-random.

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 poly-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 poly-random functions. Our work was motivated by an open problem of [9] and [7].

In the rest of this introduction we informally discuss the notion of a poly-random collection: a set of functions easy to select and evaluate, which 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 pseudorandom bit generators (CSB generators).

1.1. POLY-RANDOM COLLECTIONS. Let I_k denote the set of all k-bit strings. Consider the set H_k of all functions from I_k into I_k. Note that the cardinality of H_k is $2^{k \cdot 2^k}$. Thus, to specify a function in H_k, we would need $k2^k$ bits: an impractical task even for a moderately large k. Assume now that for all integer k one randomly selects a subset $\bar{H}_k \subseteq H_k$ of cardinality 2^k. Let $\bar{H}$ denote the collection $\{\bar{H}_k\}$. This way each function in $\bar{H}_k$ has a unique k-bit index. However, with probability 1, there is no polynomial-time algorithm that, given the k-bit index of a function $f \in \bar{H}_k$ and $x \in I_k$, evaluates $f(x)$.

Our goal is to make "random functions" accessible for applications. That is, to construct functions that can be easily specified and evaluated and yet cannot be distinguished from functions chosen at random in H_k. Thus we restrict ourselves to choosing functions from a multiset F_k (whose elements are in H_k), where the

collection $F = \{F_k\}$ has the following properties:

(1) *Indexing*: Each function in F_k has a unique k-bit index associated with it: $F_k = \{f_i \mid i \in I_k\}$. Thus picking randomly a function $f \in F_k$ is easy, if k random bits are available.
(2) *Poly-time Evaluation*: There exists a polynomial algorithm that (for all $k \geq 1$), upon input of an index $i \in I_k$ and an argument $x \in I_k$, computes $f_i(x)$.
(3) *Pseudorandomness*: No probabilistic algorithm that runs in time polynomial in k can distinguish the functions in F_k from the functions in H_k. (See Section 3.1 for a precise definition.)

Such a collection of functions F is called a *poly-random collection*. Loosely speaking, despite the fact that the functions in F_k are easy to select and evaluate, they will exhibit, to an examiner with polynomially bounded resources, all the properties of functions randomly selected in H_k.

The above definition is highly constructive. We transform *any* CSB generator (a high-quality pseudorandom bit generator, discussed in Section 2) to a poly-random collection. It has been shown (see the discussion in Section 2.3) that CSB generators can be constructed if one-way functions exist.

1.2 Comparison with One-Way Functions. Informally, one-way functions are functions that are easy to compute, but hard to invert for some nonnegligible fraction of the instances. We construct random functions from any one-way function. This confirms the great potential present in the notion of a one-way function. However, this power needs to be carefully brought out.

Although the inverse of a one-way function is somewhat unpredictable, this does not mean that it is random. In fact, all functions that are currently believed to be one-way satisfy various algebraic identities (e.g., the Rivest–Shamir–Adelman (RSA) function [33] is a multiplicative permutation; thus given its inverse on x and y, one can easily infer its inverse at $x \cdot y$). This clearly does not happen with truly random functions and in fact will not happen with a function randomly selected from a poly-random collection $\{F_k\}$. In particular, our construction hides all the algebraic identities that may be satisfied by the one-way function upon which it is based from any observer with polynomially bounded resources. In fact, the following property holds for poly-random collections:

> Randomly 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: $y_1, y_2, \ldots, y_{k^c}$. 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 substantially greater than $\frac{1}{2}$.

Thus, if f is selected in a poly-random collection, not only the value of f at argument 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 CSB Generators. In this subsection, we address the problem of simulating probabilistic polynomial-time computations so as to save basic resources such as coin tosses and storage. As we shall see, CSB generators allow one to save coin tosses in probabilistic polynomial-time computations, whereas poly-random collections allow one to save both coin tosses and storage in polynomial-time computation with a random oracle.

CSB generators are efficient deterministic programs that stretch a (random) k-bit-long input seed to a k^t-bit-long output (pseudorandom) sequence, for some constant $t > 0$. These sequences are indistinguishable, in polynomial time, from k^t-bit-long truly random sequences (see Section 2.1 for a detailed discussion). Thus, we can replace the coin tosses in a probabilistic poly(k)-time computation by the bit sequence generated by a CSB generator on a random k-bit string and still get almost the same results.

We now address the problem of efficiently simulating more complex probabilistic computations: computations with a random oracle. A random oracle [4] is a special case of a random function: it associates the result of a single coin toss to each string. In computing with a random oracle, an algorithm queries the oracle with a string q and receives q's associated bit (denoted $b(q)$). Since $b(q)$ does not change with time, the algorithm need not store the pair $(q, b(q))$ but rather query the oracle on q whenever it needs $b(q)$. The advantages of computing with a random oracle are clarified by all the applications listed in Section 5.

A polynomial-time computation that queries a random oracle on k^t strings of length k can be *trivially* simulated using a CSB generator and only k coins (see below). However, this trivial simulation of the oracle requires k^{t+1} bits of storage.

> Store a randomly selected k-bit string s and denote by b_i the ith bit produced by a CSB generator on input s. Let q_i be the ith new query (i.e., a query never asked before). Then set $b(q_i) = b_i$, append (an encoding of) q_i to a list of past queries, and answer b_i. The ordered list of past queries enables us to recognize whether a query has occurred before and, if so, to give the same answer.

Note that the list of past queries is indeed necessary. Such a list may not be significantly compressable (e.g., for randomly selected queries). Thus, in the worst case the simulation requires at least k^{t+1} bits of storage.

An interesting property of poly-random collections is that they guarantee the same result for any polynomial-time 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 f in a poly-random collection. The bit associated with each string x will be the first bit of $f(x)$.

Sharing Randomness in a Distributed Environment. An additional advantage of poly-random collections is that they enable many parties to *share efficiently* 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 can be achieved by flipping k coins to specify a function f in a poly-random collection. These k bits are communicated to and stored by each processor. No further messages need to be exchanged between the processors to share f.

1.4 Notation and Conventions. This paper does not deal with Kolmogorov complexity. Whenever we refer to a random element, we mean an element randomly selected from the appropriate set.

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. That is, an element occurring in A with multiplicity m is chosen with probability $m/|A|$.

For uniformity of notation, the parameter k, when given as input to any algorithm discussed in this paper, is presented in unary. (We analyze the running time of our

algorithms with respect to the *length* of their input, and some algorithms in this paper have only k as input.)

We chose Turing machines as the basic computational model for this paper. One can easily transform the statements and the proofs of all our results from the Turing machine model to the circuit model.

1.5 ORGANIZATION OF THE PAPER. In Section 2 we briefly recall the basic definitions and results concerning CSB generators and the easy-access open problem. In Section 3 we formally define poly-random collections and show how to construct them 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.

2. *CSB Generators*

A pseudorandom number generator is a deterministic and efficient algorithm that stretches a random input number (seed) into a long pseudorandom number sequence. The pseudorandom sequence should have "some" statistical properties present in truly random sequences (e.g., it should have approximately as many 0's as 1's). Many statistical properties of the linear congruential pseudorandom number sequence $x_{i+1} = a \cdot x_i + b (\text{mod } n)$ have been studied by Knuth [21]. However, unlike truly random sequences, the next number in a linear congruential sequence can be easily computed from the preceding ones, even when x_0, a, b, and n are not given [31]. See also [12], [20], and [23].

Shamir [35] presents a pseudorandom number generator for which computing the next number in the sequence from the preceding ones is as hard as inverting the RSA function. However, though unpredictable, the numbers in such a sequence may not appear "random" (e.g., their high-order bits may be easy to predict). All such problems provably do not arise for CSB generators.

2.1 THE NOTION OF A CSB GENERATOR. Blum and Micali [8] introduced the notion of a cryptographically strong pseudorandom bit generator (CSB generator). Let P be a polynomial. A CSB 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 CSB sequence) that passes all *next-bit-tests*:

> Let P be a polynomial, S_k a multiset[1] consisting of $P(k)$-bit sequences and $S = \bigcup_k S_k$. A *next-bit-test* for S is a probabilistic polynomial-time algorithm T that on input k and the first i bits in a string $s \in_R S_k$ outputs a bit b. Let p_k^i denote the probability that b equals the $i+1$st bit of s. We say that S *passes the next-bit-test* T if, for all polynomials Q, for all sufficiently large k, and for all integers $i \in [0, P(k))$;
>
> $$\left| p_k^i - \frac{1}{2} \right| < \frac{1}{Q(k)}.$$[2]

A more general definition of string randomness has been suggested by Yao [41].

[1] We use multisets instead of sets since it may be that a CSB generator outputs the same CSB sequence on two different seeds.

[2] In the original version of [8] a fixed $\epsilon > 0$ appeared instead of $1/Q(k)$. The replacement by $1/Q(k)$ was suggested by Yao [41]. This was of crucial importance for proving Theorem 1 [41].

2.2 POLYNOMIAL-TIME STATISTICAL TESTS FOR STRINGS

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

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

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

Of special interest is the case in which the polynomial $P_1(k)$ is the constant 1, so that the statistical test receives as an input a single string.

The following definition plays an important role in relating the above definitions of randomness. We say that a multiset $S = \bigcup_k S_k$ is *samplable* if there is a probabilistic polynomial-time algorithm that, given as input k, outputs $s \in_R S_k$.

THEOREM 1. (Yao [41]). *Let $S = \bigcup_k S_k$ be a samplable multiset of bit sequences. Then the following three statements are equivalent*:

(*i*) *S passes the next-bit-test.*
(*ii*) *S passes all polynomial-time statistical tests for strings.*
(*iii*) *S passes all polynomial-time statistical tests whose input consists of a single string in S.*

Notice that CSB sequences form a samplable multiset. Therefore,

(*) *CSB sequences pass all polynomial-time statistical tests.*

Actually, only statement (*) explicitly appears in [41]. However, its proof contains all the ideas needed for proving the equivalence of the three conditions of Theorem 1. The reader can derive a proof of Theorem 1 from the proof of Theorem 4 (which can be viewed as a generalization of Theorem 1).

2.3 IMPLEMENTATION OF CSB GENERATORS. Blum and Micali [8] presented an algorithmic scheme for constructing CSB generators based on a general complexity theoretic assumption (a sketch can be found in Section A1 of the Appendix). They also presented the first instance of their scheme based on a specific complexity 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 ϵ of all primes of length k. A more efficient CSB generator based on the DLP can be derived from [28].

Other instances of CSB generators based on various number-theoretic assumptions have been found. CSB generators based on the quadratic residuosity problem appeared in [41] and [7]. The first CSB generator based on factoring appears in [41]. More efficient generators based on factoring have been obtained by a sequence of stronger results [2; 5, 18, 39,]. As pointed out in [40], the results in [2] imply that the quadratic residuosity generators in [41] and [7] are in fact also based on factoring.

More generally, Yao [41] has shown how to obtain CSB generators if any one-way *permutation* is given. Levin [27] shows how to obtain CSB generators using a seemingly weaker condition: The existence of one-way functions (defined below).

Definition (Levin). Let $D_k \subseteq I_k$. Let f_k: $D_k \to D_k$ be a sequence of functions and let the function f be defined as follows: $f(x) = f_k(x)$ if $x \in D_k$. Let f^i denote f applied i times. Let $D_k^i \subseteq D_k$ such that $y \in D_k^i$ if $y = f^i(x)$ for some $x \in D_k$. f is a *one-way function* if

(1) f is polynomial-time computable;
(2) f is hard to invert; that is, for every probabilistic polynomial-time algorithm A and for all sufficiently large k, for every $1 \le i \le k^3$, $A(x) \ne f_k^{-1}(x)$ for at least a constant fraction of the $x \in D_k^i$;
(3) $\cup D_k$ is samplable.

THEOREM 2. (Levin [27]). *There exists a one-way function if and only if there exists a CSB generator.*

The above theorem is constructive. Levin shows a particular generator that is a CSB generator if any CSB generator exists. Levin makes use of a construction due to Yao [41], which is sketched in Section A2 of the Appendix.

2.4 CSB GENERATORS WITH EASY ACCESS. Notice that, even though a CSB sequence generated with a k-bit-long seed consists of polynomially many (in k) bits, a CSB generator and a seed s define an infinite (ultimately periodic) bit-sequence $b_0, b_1, \ldots$. An interesting feature first present in the generator of Blum et al. [7] is that knowledge of the seed allows easy access to each of the first 2^k bits; that is, if $\log i < k$, the ith bit in the string b_i can be computed in poly(k) time. This is due to the *special* one-way permuation on which the security of their generator is based. However, this easily accessible exponentially long bit-string *may not* appear "random." Blum et al. only prove that any *single* polynomially long interval of *consecutive* bits in the string passes all polynomial-time statistical tests for strings, provided that squaring mod a *Blum-integer*[3] n is a one-way permutation (over the squares mod n). Indeed, it may be the case that, given $b_1, \ldots, b_k$ and $b_{2^{\sqrt{k}}+1} \ldots, b_{2^{\sqrt{k}}+k}$, it is easy to compute any other bit in the string.

The *easy-access* open problem consists of whether easy access to exponentially far away bits in their pseudorandom pad is a "randomness preserving" operation. This problem was posed by Brassard [9] and Blum et al. [7]. The problem was also discussed by Angluin and Lichtenstein [3].

Notice that there is a natural one-to-one correspondence between "randomness preserving" easily 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 $\{s_f = f(1)f(2) \cdots f(2^k) \mid f \in F_k\}$ which can be easily accessed in a "randomness preserving" manner. This practically solves the easy-access problem. In fact, our construction demonstrates a different way to achieve the benefits that a positive answer to the easy-access problem would have provided. Even better, we construct poly-random collections not only if squaring modulo a Blum-integer is a one-way function, but given any one-way function.

[3] 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.

3. *Constructing Poly-Random Collections*

In this section we show how to construct collections of 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 $f \in F_k$, $f: I_k \to I_k$.

3.1 Polynomial-Time Statistical Tests for Functions

Definition. 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 $f: I_k \to I_k$, 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;

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

where p_k^F denotes the probability that T outputs 1 on input k and access to an oracle O_f for a function $f \in_R F_k$ and p_k^H denotes 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). Here the probabilities are taken over all the possible choices of $f \in F_k$ or H_k and the internal coin tosses of T.

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 $f \in_R F_k$) or 1 (if it "thinks" that $f \in_R H_k$). If the collection F passes the test T, then the output of T when given access to an oracle O_f gives no information on whether $f \in_R F_k$ 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. For example, suppose that some efficient algorithm A can find dependencies among input–output pairs of $f \in_R F_k$; then A can be converted to a statistical test T_A that will output 0 upon A's detection of such dependencies (i.e., judging that $f \in_R F_k$). Since such dependencies cannot be found when $f \in_R H_k$, the collection $F = \{F_k\}$ will not pass the test T_A. (For a more detailed discussion see Section 4.)

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

3.2 The Construction of F.

In this section we show how to use a CSB generator to construct a poly-random collection. In other words we show that pseudorandomness for strings implies pseudorandomness for functions. Since a CSB generator can be explicitly constructed if one-way functions exist, so can poly-random collections.

In particular, our construction utilizes *any* CSB generator G that stretches a seed $x \in I_k$ into the $2k$-bit-long sequence $G(x) = b_1^x \cdots b_{2k}^x$.

Let S_k be the multiset of the $2k$-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 $G_0(x)$ we denote the first k bits output by G on input x. That is, $G_0(x) = b_1^x \cdots b_k^x$. By $G_1(x)$ we denote the next k bits output by G. That is, $G_1(x) = b_{k+1}^x \cdots b_{2k}^x$. Let $\alpha = \alpha_1\alpha_2 \cdots \alpha_t$ be a binary string. We define $G_\alpha(x) = G_{\alpha_t}(\cdots(G_{\alpha_2}(G_{\alpha_1}(x)))\cdots)$.

For $x \in I_k$, the function $f_x: I_k \to I_k$ is defined as follows:

$$f_x(y) = G_y(x).$$

Let $F_k = \{f_x\}_{x \in I_k}$. Then $F = \{F_k\}$ is the desired collection.[4]

The reader may find it useful to picture a function $f_x: I_k \to I_k$ as a rooted full binary tree of depth k with k-bit strings stored in the nodes and edges labeled 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 labeled 0 and the edge (v, v_r) is labeled 1. The string $f_x(y)$ is then stored in the leaf reachable from the root following the edge path labeled y. See Figure 1.

We remark that computing $f_x(y)$ on inputs x and y requires $k \cdot T_k$ steps, where T_k denotes the number of steps for computing $G(x)$ on input $x \in I_k$. Also note that the functions in F_k may not be one-to-one.

3.3 THE POLY-RANDOMNESS OF F. The collection F just defined satisfies conditions 1 (indexing) and 2 (poly-time evaluation) of a poly-random collection (see Section 1.1). The main theorem shows that condition 3 (pseudorandomness) is also satisfied.

THEOREM 3 (MAIN THEOREM). *Let F be a collection of functions constructed as in Section 3.2 using a CSB generator G. Then F passes all polynomial-time statistical tests for functions.*

PROOF. Let us first give an overview of the proof. We assume, for contradiction, that there exists some probabilistic polynomial-time statistical test for functions T that F does not pass. We then use T to construct a polynomial-time statistical test for strings, A_T. We reach a contradiction by showing that the set of CSB sequences produced by G does not pass A_T.

Let us consider computations of the statistical test T in which T's queries are answered by one of the following probabilistic algorithms A_i $(i = 0, 1, \ldots, k)$ (instead of being answered by an oracle O_f).

Algorithm A_i answers T's queries as follows.[5] Let $y = y_1y_2 \cdots y_k$ be a query of T. Then

if y is the first query with prefix $y_1 \cdots y_i$
then A_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 A_i retrieves the pair $(y_1 \cdots y_i, v)$ and answers $G_{y_{i+1}\cdots y_k}(v)$.

(Conceptually, algorithm A_i starts with a full binary tree of depth k and stores random k-bit strings in all nodes of level i. In the nodes of succeeding levels, it stores k-bit strings deterministically computed by applying G as follows. If the k-bit string s is stored in an internal node v, then $G_0(s)$ is stored in v's left-son and $G_1(s)$ is stored in v's right-son. The algorithm answers query q with the string stored in the leaf reachable from the root following the edge path q.)

[4] In the next subsection we show that F is a poly-random collection. We do not know whether this is also true when one defines $f_x(y) = G_x(y)$.

[5] We extend our notation by letting $G_\lambda(x) = x$, where λ denotes the empty string.

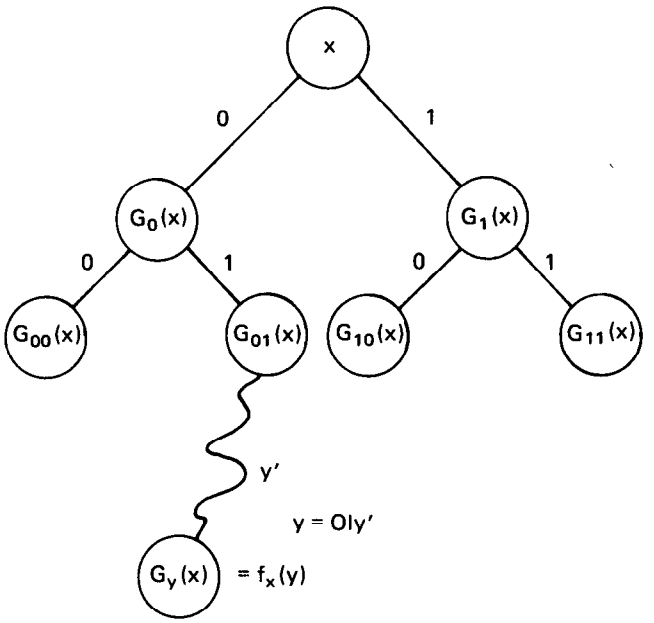

FIG. 1. The string $fx(y)$ is then stored in the leaf reachable from the root following the edge-path labeled y.

Define p_k^i to be the probability that T outputs 1 when given k as input and its queries are answered by algorithm A_i, $0 \le i \le k$.

Define p_k^F (p_k^H, respectively) to be the probability that T outputs 1 when given k as input and access to an oracle O_f for a function $f \in_R F_k$ ($f \in_R H_k$, respectively). Note that $p_k^0 = p_k^F$ and that $p_k^k = p_k^H$.

As F is assumed not to pass T, there exists a polynomial Q and infinitely many k so that $|p_k^F - p_k^H| > 1/Q(k)$. Equivalently, $|p_k^0 - p_k^k| > 1/Q(k)$. We denote by K the set of all such k's.

We are now ready to describe the polynomial-time statistical test A_T for strings. Let P_1 be a polynomial such that the test T makes at most $P_1(k)$ queries on input k. On input $k \in K$ and a set U_k of $P_1(k)$ strings, each $2k$ bits long, the test A_T performs a two-stage computation. In the first stage, A_T picks i between 0 and $k-1$ with uniform probability. In stage two, algorithm A_T gives k as input to algorithm T and answers T's oracle queries consistently using the set U_k as follows.

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

if y is the first query with prefix $y_1 \cdots y_i$
then A_T picks the next string in U_k. Let $u = u_0u_1$ be such a string (u_0u_1 is the concatenation of u_0 and u_1, and $|u_0| = |u_1| = k$). Then A_T stores the pairs $(y_1 \cdots y_i0, u_0)$ and $(y_1 \cdots y_i1, u_1)$ and answers

$$G_{y_{i+2}\cdots y_k}(u_0) \quad \text{if } y_{i+1} = 0 \quad \text{and} \quad G_{y_{i+2}\cdots y_k}(u_1) \quad \text{if } y_{i+1} = 1.$$

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

Note that, if U_k consists of $2k$-bit strings output by the CSB generator G on randomly selected k-bit input seeds, then A_T simulates a computation of T with oracle A_i. If, instead, U_k consists of randomly selected $2k$-bit strings, then A_T simulates a computation of T with oracle A_{i+1}.

The probability that A_T outputs 1 when U_k is a randomly chosen subset of the $2k$-bit strings output by the CSB generator G is $\sum_{i=0}^{k-1} (1/k) \cdot p_k^i$. On the other hand, the probability that A_T outputs 1 when U_k is a randomly chosen subset of all $2k$-bit strings, is $\sum_{i=0}^{k-1} (1/k) \cdot p_k^{i+1}$. When $k \in K$, these probabilities differ by at least $(1/k) \cdot |p_k^0 - p_k^k| > 1/(k \cdot Q(k))$. Thus, the sequences produced by G do not pass the statistical test A_T and we reach a contradiction. □

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 $I_{P_1(k)} \to I_{P_2(k)}$ (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 CSB generators: G mapping k bit strings into $2k$ bit strings and G' mapping k random input bits into $P_2(k)$ pseudorandom bits. For $x \in I_k$ the function $f_x \in F_k^{P_1,P_2}$ is defined as follows: on input $y \in I_{P_1(k)}$, $f_x(y) = G'(G_y(x))$. By a proof similar to that of the Main Theorem, we can prove that the collection, $\{F_k^{P_1,P_2}\}$ possesses properties (1)–(3) of poly-random collections.

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."
(2) It is possible to conduct *selected* experiments.
(3) The goal is only to *approximately infer* the "laws of nature."

Similarly, in complexity theory, one may conjecture that all functions f that are "simple" (i.e., that are easy to evaluate given some hidden key) can be "approximately inferred" after temporary access to an oracle for f. In this section we show that this is not the case, under the assumption that one-way functions exist.

4.1 FORMAL SETTING. Let $F = \{F_k\}$ be a collection of functions and A a probabilistic polynomial-time algorithm capable of oracle calls. 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 $x_1, \ldots, x_j$. Then algorithm A outputs $x \in I_k$ such that $x \neq x_1, \ldots, x_j$. This x is called the *chosen exam.* At this point A is disconnected from O_f and is presented the two values $f(x)$ and $y \in_R I_k$ in random order. We say that A *passes the exam* if it correctly guesses which of the two values is $f(x)$. Let Q be a polynomial. We say that A *Q-infers* the collection F if, given input k, for infinitely many k, it passes the exam with probability at least $1/2 + 1/Q(k)$. Here the probability is taken over all the possible choices of $f \in F_k$, the internal coin tosses of A, all possible choices of y, and the random order between $f(x)$ and y.

We say that a collection of functions F can be *polynomially inferred* if there exist a polynomial Q and a probabilistic polynomial-time algorithm A that Q-infers F.

Polynomially inferring a collection F is a very weak kind of a prediction problem. However, if one-way functions exist, it is still an infeasible task, as follows from Theorems 3 and 4.

THEOREM 4. *Let $F = \{F_k\}$ be a collection of functions satisfying the conditions* 1 (*indexing*) *and* 2 (*polynomial-time evaluation*) *of a poly-random collection. Then F cannot be polynomially inferred if and only if it passes all polynomial-time statistical tests for functions.*

PROOF. Assume, first, that there exists a probabilistic polynomial-time algorithm A that Q-infers the collection F. Then F does not pass the statistical test for functions, T_A, described here:

> On input k and access to an oracle O_f ($f \in_R F_k$ or $f \in_R H_k$), the test T_A invokes the inferring algorithm A with input k. For every query q made by A, the test T_A asks O_f for $f(q)$ and returns the answer to A. Finally, when A outputs the string x as its chosen exam, T_A queries O_f on x, randomly picks $y \in I_k$, and returns y and $f(x)$ to A in random order. If A correctly identifies $f(x)$, then T_A outputs 1; otherwise T_A outputs 0.

For all k, when $f \in_R H_k$ the probability that T_A outputs 1 is exactly $\frac{1}{2}$. On the other hand, for infinitely many k, when $f \in_R F_k$ the probability that T_A outputs 1 is greater than $\frac{1}{2} + 1/Q(k)$. Thus, F does not pass the test T_A.

Conversely, assume that F does not pass a statistical test T. That is, there exists a polynomial Q such that for infinitely many k, $|p_k^F - p_k^H| > 1/Q(k)$, where p_k^F (p_k^H, respectively) is the probability that T outputs 1 on input k and access to an oracle O_f for $f \in_R F_k$ ($f_R \in_R H_k$, respectively). Without loss of generality, we may assume that, for infinitely many k, $p_k^F - p_k^H > 1/Q(k)$, and let K denote the set of all such k. Also, without loss of generality, during the same computation, T never asks the same query twice and, on input k, asks exactly $P(k)$ queries (for some polynomial P).

We construct a probabilistic polynomial-time algorithm A_T that uses T as a subroutine and $2 \cdot P(k) \cdot Q(k)$-infers F. On input k and access to an oracle O_f ($f \in_R F_k$), the algorithm A_T proceeds as follows. It first chooses i between 0 and $P(k) - 1$ with uniform probability. (We later refer to i as the *index*.) Next A_T invokes T with input k and uses the oracle O_f to answer T's first i queries. When T asks for its $i + 1$st query, x_{i+1}, then A_T outputs x_{i+1} as its chosen exam. Upon receiving $f(x_{i+1})$ and y, where $y \in_R I_k$, A_T randomly chooses $z \in \{f(x_{i+1}), y\}$ and gives z as an answer to query x_{i+1}. Next, algorithm A_T continues to answer the queries x_{i+2} through $x_{P(k)}$ of T by randomly selected k-bit strings. Finally, T outputs a bit and halts. If T's output was a 0, then A_T guesses that $z \in_R I_k$; otherwise A_I, guesses that $z = f(x_{i+1})$.

In analyzing the probability that A_T makes a correct guess, the following concept of a (k, i, g)-*experiment* (where $g \in F_k$) will be useful:

Run T with input k and answer its queries as follows. Let x_j be the jth query of T.

if $j \leq i$, **then** answer $g(x_j)$; **else** answer with a random k-bit string.

Let p_k^i be the probability that T outputs 1 in a (k, i, g)-*experiment* when $g \in_R F_k$. Note that $p_k^0 = p_k^H$ and $p_k^{P(k)} = p_k^F$.

Let us calculate the probability that A_T makes a correct guess on input $k \in K$ and access to oracle O_f for $f \in_R F_k$. (In this calculation, k is fixed and the probabilities are taken over all possible choices of $f \in F_k$ and the internal coin tosses of A_T with uniform distribution.) Consider executions of A_T. Let A_T^i denote

the event "Algorithm A_T chose *index* = i". Then,

$$\begin{aligned}
&\text{prob}(A_T) \text{ is correct} \\
&= \sum_{i=0}^{P(k)-1} \text{prob}(A_T^i) \cdot \text{prob}(A_T \text{ is correct} \mid A_T^i) \\
&= \frac{1}{P(k)} \cdot \sum_{i=0}^{P(k)-1} [\text{prob}(z \in_R I_k \mid A_T^i) \\
&\qquad \cdot \text{prob}(A_T \text{ guesses } z \in_R I_k \mid z \in_R I_k \text{ and } A_T^i) \\
&\qquad + \text{prob}(z = f(x_{i+1}) \mid A_T^i) \\
&\qquad \cdot \text{prob}(A_T \text{ guesses } z = f(x_{i+1}) \mid z = f(x_{i+1}) \text{ and } A_T^i)] \\
&= \frac{1}{P(k)} \cdot \sum_{i=0}^{P(k)-1} \left[\frac{1}{2} \cdot \text{prob}(T \text{ outputs } 0 \mid z \in_R I_k \text{ and } A_T^i)\right. \\
&\qquad \left. + \frac{1}{2} \cdot \text{prob}(T \text{ outputs } 1 \mid z = f(x_{i+1}) \text{ and } A_T^i)\right] \\
&= \frac{1}{2 \cdot P(k)} \cdot \sum_{i=0}^{P(k)-1} ((1 - p_k^i) + p_k^{i+1}) \geq \frac{1}{2} + \frac{1}{2 \cdot P(k) \cdot Q(k)}.
\end{aligned}$$

□

COROLLARY. *Poly-random collections can not be polynomially inferred.*

Remark. Our construction of poly-random collections has a "strengthening effect." Assume that $F^{(g)}$ is a poly-random collection constructed given the one-way function g. Then the functions in $F^{(g)}$ cannot be polynomially inferred, even if g and/or g^{-1} is polynomially inferable.

5. *Cryptographic Applications and Further Improvements*

Poly-random collections constitute a very powerful tool in a cryptographic setting. The functions in such collections are easy to select and compute with, but retain all the desired statistical properties of random functions with respect to adversaries that are bounded to polynomial-time computation. This suggests the following methodology for protocol design. First, design a protocol that (magically) uses truly random functions and prove it correct. This step is often very easy. Then, replace the truly random functions by functions randomly selected from a poly-random collection. This replacement will provably maintain all properties of the original protocol with respect to polynomially bounded adversaries.

This methodology has provided rigorous solutions to such cryptographic problems as message authentication with time stamping, storageless distribution of secret identification numbers, identifying friend or foe systems, and cryptographically strong hashing. A detailed discussion of these applications is presented in [15].

Levin and Goldreich pointed out in [17a] that poly-random collections can be used to make the Goldwasser–Micali–Rivest [17] signature scheme "memoryless." The use of poly-random collections is crucial in the fair contract signing protocol of Ben-Or et al. [6].

Recently, Luby and Rackoff [29] used poly-random collections to construct collections of poly-random permutations. This result leads to the construction of "ideal private key cryptosystems."

Levin [27] proposed a modification of our poly-random construction, that can be carried out in poly(log k) steps. This implies that, if there exists a CSB generator

that works in NC, then there exists a poly-random collection of functions that can be evaluated in NC.

Appendix

A1. Sufficient Conditions for Constructing CSB Generators. 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 [8] showed that CSB generators can be constructed under the following conditions:

(1) The domain is accessible: There exists a *probabilistic* polynomial-time algorithm that on input k, chooses $x \in D_k$ with uniform probability distribution.
(2) The permutation is easy to evaluate: There exists a polynomial-time algorithm that on input k and $x \in D_k$, compute $g_k(x)$.
(3) The predicate is inapproximable: Let A be any *probabilistic* polynomial-time algorithm and Q be any polynomial. Then for all sufficiently large k:

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

(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 [41] showed that the existence of a one-way permutation is a sufficient condition for constructing CSB generators.

A2. A Sketch of Yao's Construction. Yao's construction [41] 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 $E = \bigcup_k E_k$. By the definition of a one-way permutation [41], no polynomial algorithm can invert h without being mistaken on a $1/k^c$ fraction of the domain, for some constant c, when k is sufficiently large.

Set D_k to be the Cartesian product of k^{2c+1} copies of E_k.
Set $g_k(x_1 x_2 \cdots x_{k^{2c+1}}) = h_k(x_1)h_k(x_2) \cdots h_k(x_{k^{2c+1}})$, where $x_j \in E_k$.
Set $B_k^i(x)$ to be the ith bit of $h_k^{-1}(x)$, where $x \in E_k$ and

$$B_k(x_1 x_2 \cdots x_{k^{2c+1}}) = \bigoplus_{i=1}^{k} \bigoplus_{j=1}^{k^{2c}} B_k^i(x_{k^{2c}(i-1)+j})$$

where $\oplus$ denotes the exclusive-OR function.

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

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

We are particularly grateful to Ron Rivest who assisted us all along with many insights and precious criticism and to Albert Meyer for quickly rescuing us from a fearful dead end. Many thanks to Michael Ben-Or, Steve Cook, Tom Leighton, Mike Luby, Gary Miller, Charles Rackoff, and Mike Sipser for several helpful discussions.

REFERENCES

(Note: References [11], [14], [16], and [32] are not cited in text.)

1. ADELMAN, L. Time, Space and Randomness. Tech. Memo 131, Laboratory for Computer Science MIT, Cambridge, Mass., 1979.
2. ALEXI, W., CHOR, B., GOLDREICH, O., AND SCHNORR, C. P. RSA and Rabin functions: Certain parts are as hard as the whole. *SIAM J. Comput.*, to appear. (An earlier version appeared in *Proceedings of the 25th IEEE Symposium on Foundations of Computer Science.* IEEE, New York, 1984, pp. 449–457.)
3. ANGLUIN, D., AND LICHTENSTEIN, D. Provable security of cryptosystems: A survey. Tech. Rep. 288, Dept. of Computer Science, Yale Univ. New Haven, Conn., 1983.
4. BENNETT, C. H., AND GILL, J. Relative to a random oracle, A, $P^A \neq NP^A \neq$ co-NP^A with probability 1. *SIAM J. Comput. 10* (1981), 96–113.
5. BEN-OR, M., CHOR, B., AND SHAMIR, A. On the cryptographic security of single RSA bits. In *Proceedings of the 15th ACM Symposium on Theory of Computing* (Boston, Mass., Apr. 25–27). ACM, New York, 1983, pp. 421–430.
6. BEN-OR, M., GOLDREICH, O., MICALI, S., AND RIVEST, R. L. A fair protocol for signing contracts. In *Automata, Languages and Programming, 12th Colloquium*, W. Brauer, Ed. Lecture Notes in Computer Science, vol. 194. Springer-Verlag, New York, 1985, pp. 43–52.
7. BLUM, L., BLUM, M., AND SHUB, M. A simple unpredictable pseudo-random number generator. *SIAM J. Comput. 15* (May 1986), 364–383.
8. BLUM, M., AND MICALI, S. How to generate cryptographically strong sequences of pseudo-random bits. *SIAM J. Comput. 13* (Nov. 1984), 850–864.
9. BRASSARD, G. On computationally secure authentication tags requiring short secret shared keys. In *Advances in Cryptology: Proceedings of Crypto-82*, D. Chaum, R. L. Rivest and A. T. Sherman, Eds. Plenum Press, New York, 1983, pp. 79–86.
10. CHAITIN, G. J. On the length of programs for computing finite binary sequences. *J. ACM 13*, 4 (Oct. 1966), 547–570.
11. DIFFIE, W., AND HELLMAN, M. E. New directions in cryptography. *IEEE Trans. Inf. Theory IT-22* (Nov. 1976), 644–654.
12. FREIZE, A. M., KANNAN, R., AND LAGARIAS, J. C. Linear congruential generators do not produce random sequences. In *Proceedings of the 25th Symposium on Foundations of Computer Science.* IEEE, New York, 1984, pp. 480–484.
13. GACS, P. On the symmetry of algorithmic information. *Sov. Math. Dokl. 15* (1974), 1477.
14. GOLDREICH, O., GOLDWASSER, S., AND MICALI, S. How to construct random functions. Tech. Memo 244, Laboratory for Computer Science, MIT, Cambridge, Mass., Nov. 1983.
15. GOLDREICH, O., GOLDWASSER, S., AND MICALI, S. On the cryptographic applications of random functions. In *Advances in Cryptology: Proceedings of Crypto-84.* B. Blakely, Ed. Lecture Notes in Computer Science, vol. 196. Springer-Verlag, New York, 1985, pp. 276–288.
16. GOLDWASSER, S. Probabilistic encryption: Theory and applications. Ph.D. dissertation, Dept. of Computer Science, Univ. of California, Berkeley, Calif., 1984.
17. GOLDWASSER, S., MICALI, S., AND RIVEST, R. L. A "paradoxical" signature scheme. In *Proceedings of the 25th IEEE Symposium on Foundations of Computer Science.* IEEE, New York, 1984, pp. 441–448.

17a. GOLDWASSER, A., MICALI, S., AND RIVEST, R. L. A digital signature scheme secure against adaptive chosen method attack. *SIAM J. Comput.* to appear.

18. GOLDWASSER, S., MICALI, S., AND TONG, P. Why and how to establish a private code on a public network. In *Proceedings of the 23rd IEEE Symposium on Foundations of Computer Science.* IEEE, New York, 1982, pp. 134–144.
19. HARTMANIS, J. Generalized Kolmogorov complexity and the structure of feasible computations. In *Proceedings of the 24th IEEE Symposium on Foundations of Computer Science.* IEEE, New York, 1983, pp. 439–445.
20. HASTAD, J., AND SHAMIR, A. The cryptographic security of truncated linearly related variables. In *Proceedings of the 17th ACM Symposium on Theory of Computing* (Providence, R.I., May 6–8). ACM, New York, 1985, pp. 356–362.
21. KNUTH, D. *The Art of Computer Programming: Seminumerical Algorithms*, vol. 2. 2nd ed. Addison-Wesley, Reading, Mass. 1981.
22. KOLMOGOROV, A. Three approaches to the concept of "The amount of information," *Prob. Inf. Transm. 1*, 1 (1965).
23. LAGARIAS, J., AND REEDS, J. Extrapolation of nonlinear recurrences. Submitted for publication.
24. LEVIN, L. A. On the notion of a random sequence. *Sov. Math. Dokl. 14*, 5 (1973), 1413.

25. LEVIN, L. A. Various measures of complexity for finite objects (axiomatic descriptions). *Sov. Math. Dokl. 17*, 2 (1976), 522–526.
26. LEVIN, L. A. Randomness conservation inequalities, information and independence in mathematical theories. *Inf. Control 61* (1984), 15–37.
27. LEVIN, L. A. One-way function and pseudorandom generators. In *Proceedings of the 17th ACM Symposium on Theory of Computing* (Providence, R.I., May 6–8). ACM, New York, 1985, pp. 363–365.
28. LONG, D. L., AND WIGDERSON, A. How discreet is discrete log? In preparation. A preliminary version appeared in *Proceedings of the 15th ACM Symposium on Theory of Computing* (Boston, Mass., Apr. 25–27). ACM, New York, 1983, pp. 413–420.
29. LUBY, M., AND RACKOFF, C. Pseudo random permutation generators and cryptographic composition. In *Proceedings of the 18th ACM Symposium on Theory of Computing* (Berkeley, Calif., May 28–30). ACM, New York, 1986, pp. 356–363.
30. MARTIN-LOF, P. The definition of random sequences. *Inf. Control 9* (1966), 602–619.
31. PLUMSTEAD, J. Inferring a sequence generated by a linear congruence. In *Proceedings of the 23rd IEEE Symposium on Foundations of Computer Science*. IEEE, New York, 1982, pp. 153–159.
32. RABIN, M. O. Digitalized signatures and public key functions as intractable as factoring. Tech. Rep. 212, Laboratory for Computer Science, Cambridge, Mass., 1979.
33. RIVEST, R., SHAMIR, A., AND ADLEMAN, L. A method for obtaining digital signatures and public key cryptosystems. *Commun. ACM, 21*, 2 (Feb. 1978), 120–126.
34. SCHNORR, C. P. Zufaelligkeit und Wahrscheinlichkeit. Lecture Notes in Mathematics, vol. 218. Springer-Verlag, New York, 1971.
35. SHAMIR, A. On the generation of cryptographically strong pseudorandom sequences. *ACM Trans. Comput. Syst. 1*, 1 (Feb. 1983), 38–44.
36. SIPSER, M. A complexity theoretic approach to randomness. In *Proceedings of the 15th ACM Symposium on Theory of Computing* (Boston, Mass., Apr. 25–27). ACM, New York, 1983, 330–335.
37. SOLOMONOFF, R. J. A formal theory of inductive inference. *Inf. Control, 7*, 1 (1964), 1–22.
38. WILBER, R. E. Randomness and the density of hard problems. In *Proceedings of 24th IEEE Symposium on Foundations of Computer Science*. IEEE, New York, 1983, pp. 335–342.
39. VAZIRANI, U. V., AND VAZIRANI, V. V. RSA bits are .732 + ϵ secure. In *Advances in Cryptology: Proceedings of Crypto-83*, D. Chaum, Ed. Plenum Press, New York, 1984, pp. 369–375.
40. VAZIRANI, U. V., AND VAZIRANI, V. V. Efficient and secure pseudo-random number generation. In *Proceedings of the 25th IEEE Symposium on Foundations of Computer Science*. IEEE, New York, 1984, pp. 458–463.
41. YAO, A. C. Theory and applications of trapdoor functions. In *Proceedings of the 23rd IEEE Symposium on Foundations of Computer Science*. IEEE, New York, 1982, pp 80–91.
42. ZVONKIN, A. K., AND LEVIN, L. A. The complexity of finite objects and the algorithmic concepts of randomness and information. *UMN (Russian Math. Surveys), 25*, 6 (1970), 83–124.

RECEIVED OCTOBER 1984; REVISED NOVEMBER 1985; ACCEPTED NOVEMBER 1985