JOURNAL OF COMPUTER AND SYSTEM SCIENCES 50, 600–610 (1995)

Cryptographic Lower Bounds for Learnability of Boolean Functions on the Uniform Distribution

MICHAEL KHARITONOV*

D. E. Shaw & Co., 120 West 45th Street, New York, New York 10036

Received November 15, 1992; revised September 13, 1993

We investigate cryptographic lower bounds on the number of samples and on computational resources required to learn several classes of boolean circuits on the uniform distribution. Under the assumption that solving $n \times n^{1+\epsilon}$ subset sum is hard, we construct (using the results of Impagliazzo and Naor and Goldreich, Goldwasser, and Micali) a pseudo-random function generator that can be computed by shallow boolean circuits. From this we conclude that learning AC^1 circuits on the uniform distribution requires $\Omega(n^{\log\log n})$ different samples, or, alternatively, that learning AC^1 circuits on the uniform distribution with a polynomial number of samples is as hard as solving $n \times n^{1+\epsilon}$ subset sum. We also show that no algorithm can learn NC^1 circuits on the uniform distribution with a polynomial number of samples. Using the weaker assumption that solving $n \times (1+\epsilon)n$ subset sum is hard, we show that the class of NC circuits can not be learned with $n^{\log^c n}$ samples for any constant c.

1. INTRODUCTION

The use of cryptography to prove hardness results in computational learning is almost as old as the field itself. Valiant, in his seminal paper on learning [Val84], described how the existence of a chosen plaintext secure encryption scheme can be used to show that unrestricted polynomial size circuits are not learnable. Valiant pointed out that a pseudo-random function generator of Goldreich, Goldwasser, and Micali [GGM86] can be used to show that, if one-way functions exist, arbitrary boolean circuits are not predictable (the formal presentation of this argument can be found in [PW88]). A modified pseudo-random function generator construction was used by Blum [Blum89] to separate the PAC learning model from the absolute mistake-bound model of learning.

Recently, Kearns and Valiant [KV89] used specifc public key encryption schemes to prove the unpredictability of NC^1 circuits (which are equivalent to boolean formulas); using prediction preserving reductions of Pitt and Warmuth [PW88] they proved the unpredictability of deterministic finite acceptors and several other important classes. Angluin and Kharitonov [AK91] used chosen cyphertext secure public key encryption and secure digital signatures to prove similar results for prediction with membership queries. While these recent results were important in proving hardness of distribution-free learning, their common drawback is that distributions of examples used for this purpose were cryptographically oriented and unnatural. The possibility of using "malicious" distributions to show non-learnability highlighted the fact that the distribution independence requirement makes PAC learning of relatively simple concepts very difficult.

At the same time, significant progress has been achieved in constructing learning algorithms that work well on specific distributions. This is especially true for the uniform distribution. Linial, Mansour, and Nisan [LMN89] used Fourier analysis to construct a $O(n^{\text{poly}\log n})$ algorithm to learn AC^0 functions on the uniform distribution. Several improvements and generalizations of this result followed, reducing both sample and time complexity and extending it to product distributions, although the running time of resulting algorithms remained super-polynomial. Very recently Mansour [Man92] used Fourier analysis to obtain a $O(n^{\log\log n})$ algorithm for learning DNF boolean formulas on the uniform distribution.

These successful efforts to construct learning algorithms that perform relatively well on the uniform distribution prompted us to look at the cryptographic limitations on this type of learning. The only previous hardness of this type was the original observation of Valiant [Val84] which applied to unrestricted boolean circuits.

We use a specific cryptographic assumption, namely that a subset sum problem of dimension $n \times n^{1+\epsilon}$ is hard, to construct pseudo-random function generators in AC^1, NC^1, and NC. While too weak for cryptographic purposes, these generators can be used to prove lower bounds on sample complexity (as well as time complexity) of learning algorithms on the uniform distribution. We show, in effect, that if a learner does not have the computational resources to solve a specific problem which is well studied and widely

* Supported by a Fannie and John Hertz Fellowship, by NSF Presidential Young Investigator Grant CCR-8858097 with matching funds from AT&T, DEC, and 3M, a grant from Powell Foundation, and a grant from Mitsubishi Electric Laboratories. This work was done while the author was at Stanford University. E-mail: misha@cs.stanford.edu.

0022-0000/95 $6.00

presumed to be "hard on average," his only strategy for learning specially constructed concept classes that can be evaluated by boolean circuits of low depth requires a large number of labeled samples. As far as we know, this is the first application of cryptography to establish a *computational* sample bound for learning algorithms.

Throughout this paper we will use the boolean circuit model of computation. All the circuits will be assumed to be over the standard boolean basis $\wedge$, $\vee$, $\neg$ and of size polynomial in the length of their inputs (the number of boolean inputs). We will discuss polynomial size circuits of unrestricted fan-in and fan-in two, and of various bounded depth. Let n denote the number of boolean inputs to a circuit. Following the standard notation we will denote by NC^i the class of fan-in two circuits whose size is polynomial in n and whose depth is $O(\log^i n)$. We will denote by AC^i the class circuits with unrestricted fan-in whose size is polynomial in n and whose depth is $O(\log^i n)$. Clearly $NC^i \subseteq AC^i \subseteq NC^{i+1}$. Let $NC = \bigcup NC^i$ for all i. Recall that the class NC^1 has the same expressive power as the class of polynomial size boolean formulas.

We will use the parameter n as the *security parameter*. The length of inputs and the running times of efficient algorithms are bounded by polynomials in n, and the assumptions (and thus the theorems derived from them) are assumed to hold for all sufficiently large values of n.

2. PRELIMINARIES

In this section we formally define the model of concept representations and efficient prediction with membership queries on specific distributions. The definitions are identical to those in [AK91], except that distribution independence is no longer required.

2.1. *Representations of Concepts*

For simplicity we will use a binary alphabet. Let X denote $\{0, 1\}^*$; binary strings will represent both examples and concept names. If x is a string, $|x|$ denotes its length. For any natural number n, $X^{[n]} = \{x \in X : |x| \leq n\}$.

A *representation of concepts* $\mathscr{C}$ is any subset of $X \times X$. We interpret an element $\langle u, x \rangle$ of $X \times X$ as consisting of a *concept name* (or *concept representation*) u and an *example* x. The example x is a member of the concept u if and only if $\langle u, x \rangle \in \mathscr{C}$.

Define the *concept represented by* u as

$$\kappa_{\mathscr{C}}(u) = \{x : \langle u, x \rangle \in \mathscr{C}\}.$$

The *set of concepts represented by* $\mathscr{C}$ is

$$\{\kappa_{\mathscr{C}}(u) : u \in X\}.$$

If a set A of binary strings is a concept represented in $\mathscr{C}$, then we define $\text{size}_{\mathscr{C}}(A)$ to be the length of the shortest string u such that $\kappa_{\mathscr{C}}(u) = A$.

For example, the class of boolean formulas is represented as follows. We fix a straightforward binary encoding of general boolean formulas over the variables $X_1, X_2, \ldots$ and the basis AND ($\wedge$), OR ($\vee$), and NOT ($\neg$). Then $\langle u, x \rangle$ is an element of $\mathscr{C}_{BF}$ if and only if u represents a positive positive integer n and a boolean formula ϕ over the variables $X_1, \ldots, X_n$ such that $|x| = n$ and the assignment $X_i = x_i$ for $i = 1, \ldots, n$ satisfies the formula ϕ.

Boolean circuits are represented in a similar way. Any boolean circuit $\aleph$ on n inputs $X_1, \ldots, X_n$ is represented by $\{\langle u, x \rangle\}$, where u represents the positive integer n and the encoding of $\aleph$, $|x| = n$, and the assignment $X_i = x_i$ for $i = 1, \ldots, n$ causes the output of $\aleph$ to be 1. The class $\mathscr{C}_{\mathscr{P}/\text{poly}}$ of polynomial size boolean circuits is specified analogously to boolean formulas: $\langle u, x \rangle$ is an element of $\mathscr{C}_{\mathscr{P}/\text{poly}}$ if and only if u represents a positive integer n and a boolean circuit $\aleph$ over the inputs $X_1, \ldots, X_n$, and iff $|x| = n$ and the assignment $X_i = x_i$ for $i = 1, \ldots, n$ causes the output of $\aleph$ to be 1.

When we say that a binary string u *encodes* or *represents* an object (such as a boolean formula or a circuit of some type), we assume that a certain fixed and reasonable encoding scheme is used. With respect to any such scheme we can assume that all binary strings represent concepts by assigning an empty set to all binary strings that do not form a valid encoding of an object. Unless stated otherwise, all numbers are represented in binary.

Since each representation of concepts $\mathscr{C}$ is a set of pairs of binary strings, we can investigate its computational complexity, that is, the computational complexity of the *evaluation problem* for $\mathscr{C}$, i.e., the complexity of deciding whether $\langle u, x \rangle \in \mathscr{C}$ or not (or, alternatively, the complexity of $\mathscr{C}$ as a binary language). Buss [Buss87] showed that boolean formula evaluations can be done by NC^1 circuits, hence $\mathscr{C}_{BF}$ is in NC^1.

We will want to discuss the learnability of standard classes of restricted boolean circuits, such as AC^1 (the class of circuits of logarithmic depth) and NC^1 (the class of circuits of logarithmic depth and fan-in two). Since we use the non-uniform model, asymptotic restrictions on size or depth only make sense for families of circuits, where there is one circuit for each input size. Our definition of concept representation, however, does not provide for the learnability of a *family* of circuits; for each input size, the corresponding circuit from the family defines a separate concept. Thus each family of circuits defines a concept class. Because the running time of a learning algorithm is bounded by a polynomial in the size of the target concept representation as well the size of the input, the learnability of the class $\mathscr{C}_{\mathscr{P}/\text{poly}}$ of polynomial size circuits is well defined. But the asymptotic restriction on the *depth* of circuits is lost. We can, however, still discuss the learnability of restricted

Boolean circuits by considering the circuit complexity of concept representation classes. We say that an algorithm successfully learns a class $\mathcal{F}$ of Boolean circuits if it can learn every representation class which is in $\mathcal{F}$ (as a language). Thus, for example, we will show that the class AC^1 is not learnable by constructing a concept class whose representation is in AC^1 but is not learnable in polynomial time.

2.2. *Prediction with Membership Queries*

A *prediction with membership queries algorithm*, or *pwm-algorithm*, is a possibly randomized algorithm A that takes as input a bound s on the size of the target concept representation, a bound n on the length of examples, and an accuracy bound ε. It may make three different kinds of oracle calls, the responses to which are determined by the unknown target concept c and the specified distribution D on $X^{[n]}$, as follows:

1. A membership query takes a string $x \in X$ as input and returns 1 if $x \in c$ and 0 otherwise.

2. A request for a random classified example takes no input and returns a pair $\langle x, b \rangle$, where x is a string chosen independently according to D and $b = 1$ if $x \in c$ and $b = 0$ otherwise, and

3. A request for an element to predict takes no input and returns a string x chosen independently according to D.

A may make any number of membership queries or requests for random classified examples. However, A must eventually make one and only one request for an element to predict, and then eventually halt with an output of 1 or 0 without making any further oracle calls. The output is interpreted as A's guess of how the target concept classifies the element returned by the request for an element to predict. *A runs in polynomial time* if its running time (counting one step per oracle call) is bounded by a polynomial in s, n, and $1/\varepsilon$.

We say that *A successfully predicts* a representation of concepts $\mathcal{C}$ on a distribution D if and only if for all positive integers s and n, for all positive rationals ε, for all concept names $u \in X^{[s]}$, when A is run with inputs s, n, and ε, and oracles determined by $c = \kappa_{\mathcal{C}}(u)$ and D, A asks membership queries that are in X and the probability is at most ε that the output of A is not equal to the correct classification of x by $\kappa_{\mathcal{C}}(u)$, where x is the string returned by the (unique) request for an element to predict. Recall that if $\varepsilon = \frac{1}{2} - 1/n^c$ for some constant $c > 0$ we say that *A weakly predicts* $\mathcal{C}$ [KV89].

A representation of concepts $\mathcal{C}$ is *polynomially predictable with membership queries on a distribution D* if and only if there is a pwm-algorithm A that runs in polynomial time and successfully predicts $\mathcal{C}$ on D.

In particular we will focus on the case where D is the uniform distribution on $X^{[n]}$ and show some classes of Boolean circuits to be unpredictable even in this case. Recall that a weak learning algorithm that is successful on a specific distribution does not necessarily imply the existence of a strong learning algorithm that is successful on the same distribution. (Unlike in the distribution independent model, where weak and strong learning were shown to be equivalent [Sch89].) Thus it is even more desirable to show the hardness of weak prediction.

3. PSEUDO-RANDOM FUNCTION GENERATORS

3.1. *Basic Definitions*

Let us first review the definitions of a pseudo-random generator and a pseudo-random function generator. For a complete and formal treatment we refer the reader to the original paper by Goldreich, Goldwasser, and Micali [GGM86] which introduced the notion of a pseudo-random function generator and showed how to construct one from a pseudo-random generator.

A *pseudo-random generator* is a polynomial time computable function $f: \{0,1\}^n \to \{0,1\}^{l(n)}$, where $l(n) > n$, such that on a random input it produces an output that no polynomial time statistical test A can distinguish with non-negligible probability of success from a truly random sequence of the same length. More formally, we require that for any polynomial time computable function $A: \{0,1\}^{l(n)} \to \{0,1\}$ (i.e., the expected running time of an algorithm computing A is polynomial in the size of its inputs) and for all c, the *success probability*

$$\delta_A(n) = |\Pr_{x \in \{0,1\}^n}[A(f(x)) = 1] - \Pr_{y \in \{0,1\}^{l(n)}}[A(y) = 1]| < 1/n^c,$$

where $x \in \{0,1\}^n$ and $y \in \{0,1\}^{l(n)}$ are chosen uniformly at random. We will call $l(n)$ the *stretch factor* of the pseudo-random generator.

A *n, k, m pseudo-random function generator* is a function $g: \{0,1\}^n \times \{0,1\}^k \to \{0,1\}^m$ with the following properties:

1. *Efficient evaluation.* For any $x \in \{0,1\}^n$ and any $y \in \{0,1\}^k$ the running time to compute $g(x, y)$ is polynomial in n.

2. *Pseudo-randomness.* Fix $x \in \{0,1\}^n$ uniformly at random and denote $g(x, y)$ by $g_x(y)$. Then no polynomial times test that has access to g_x as an oracle can distinguish between g_x and a function chosen at random from the set of all functions $H_{k,m} = \{h: \{0,1\}^k \to \{0,1\}^m\}$. More formally, we require that for any randomized polynomial time oracle Turing machine T^f (i.e., the expected running time of T^f is polynomial in the size of its inputs) with access to an

oracle for computing a function $f: \{0,1\}^k \to \{0,1\}^m$ and for all c, the *success probability*

$$\delta_T(n) = |\Pr_{x \in \{0,1\}^n}[T^{g_x}(1^n, 1^k, 1^m) = 1] - \Pr_{f \in H_{k,m}}[T^f(1^n, 1^k, 1^m) = 1]| < 1/n^c,$$

where $x \in \{0,1\}^n$ and $f \in H_{k,m}$ are chosen uniformly at random.

This definition of the pseudo-random function generator is the same as the original definition in [GGM86], except that there mainly the case $k = m = n$ is considered. (The case where k and m are polynomial in n is also mentioned in [GGM86].) Note that in our definition only n is used as the *security parameter*, in the sense that running times and success probabilities of statistical tests are bound by polynomials in n alone. Because of the running time bounds it is clear that k and m are bounded from above by polynomials in n, but they are not bounded from below. For our purposes, we can set $m = 1$ and k will range as low as $\log n$.

3.2. *Prediction Hardness*

It was shown in [GGM86] that a collection of pseudo-random functions is not predictable in a very strong sense. Valiant also observed in [Va84] that the existence of a pseudo-random function generator implies that the class $\mathcal{P}$/poly of unrestricted polynomial size boolean circuits is not PAC learnable, even when membership queries are available. This observation, together with the notions of polynomial predictability and prediction preserving reduction, is formally presented in [PW88].

It is clear from the definition of the pseudo-random function generator that any polynomial time algorithm has probability less than $\frac{1}{2} + 1/n^c$ for any $c > 0$ of successfully predicting the output of a pseudo-random function generator on any input not previously seen in the learning stage. Therefore we have the following observation.

OBSERVATION 1. *Let g be a n, k, 1 pseudo-random function generator. Then the concept class $\mathcal{G}_{n,k} = \{\langle x, y\rangle : x \in \{0,1\}^n, y \in \{0,1\}^k, g_x(y) = 1\}$ cannot be predicted in time polynomial in n with or without membership queries on any non-trivial polynomial time samplable distribution D over $\{0,1\}^k$ as long as k is large enough to ensure that with high probability the prediction challenge (chosen according to D) will not coincide with any of the labeled examples (also chosen according to D) or membership queries.*

Proof. Assume to the contrary that a polynomial time prediction algorithm with membership queries A is successful for $\mathcal{G}_{n,k}$. Let us transform it into a successful statistical test T for the pseudo-random function generator g. Given an oracle for computing a boolean function f, T^f simulates the algorithm A with inputs n (bounding the size of the concept) and k (bounding the length of examples) and error parameter 1/4. The oracle queries of A are answered as follows:

1. When A makes a membership query with string y, if $|y| = k$, then T^f makes an oracle call and returns $f(y)$ back to A. If $|y| \neq k$, then T^f returns 0.

2. When A requests a random classified example, T^f samples from the distribution D over $\{0,1\}^k$ to obtain an example y and then makes an oracle call to obtain $f(y)$. T^f returns $\langle y, f(y)\rangle$ back to A.

3. When A requests an element to predict, T^f samples from the distribution D over $\{0,1\}^k$ to obtain a string y which is returned to A as the element to predict.

Let $p(n,k)$ be the polynomial upper bound on the running time of A (recall that k is bounded from above by a polynomial in n by the definition of a pseudo-random function generator). If within $p(n,k)$ steps A makes a prediction b on an element y, T^f makes an oracle call to obtain $f(y)$ and outputs 1 if $f(y) = b$ and 0 otherwise. If within $p(n,k)$ steps A fails to make a prediction (or request a challenge), T^f outputs 1 with probability $\frac{1}{2}$ and 0 with probability $\frac{1}{2}$.

If $f = g_x$ for any $x \in \{0,1\}^n$, then from correctness of A it follows that with probability at least $\frac{3}{4}$ A will make within $p(n,k)$ steps a correct prediction b on a challenge y, i.e., $b = f(y)$. Thus $\Pr_{x \in \{0,1\}^n}[T^{g_x}(n,k,1) = 1] \geqslant \frac{3}{4}$.

If, on the other hand, f is a random Boolean function on k variables (i.e., chosen at random from $H_{k,1}$), then, unless the prediction challenge x was seen previously as a labeled example or a membership query, T^f outputs 1 with probability $\frac{1}{2}$ irrespective of whether A makes a prediction or not. If, for example, k is large enough that the probability of y being a new element is at least $\frac{3}{4}$, then $\Pr_{f \in H_{k,1}}[T^f(n,k,1) = 1] \leqslant \frac{5}{8}$. Therefore,

$$|\Pr_{x \in \{0,1\}^n}[T^{g_x}(1^n, 1^k, 1^m) = 1] - \Pr_{f \in H_{k,m}}[T^f(1^n, 1^k, 1^m) = 1]| \geqslant \tfrac{1}{8}$$

which makes T a successful statistical test for the pseudo-random function generator g. ∎

Note that in the definitions of polynomial predictability (or learnability) a learner is allowed time polynomial in the size of the examples, which is k, as well as the size of the concept representation, which is n. But, as was mentioned above, $k < p(n)$ for some polynomial p, so this is not a problem. In fact, in our applications we will always have $k \leqslant n$.

Another way to restate Observation 1 is to say that in order to learn $\mathcal{G}_{n,k}$ a prediction algorithm must collect enough samples to assure that the probability of prediction challenge coinciding with one of the samples is sufficiently

high. Let us compute the number of labeled examples necessary to *weakly* predict $\mathcal{G}_{n,k}$ on the uniform distribution over $\{0,1\}^k$. Recall that a weak learning algorithm (defined in [KV89]) is required to have a prediction error $\varepsilon \leqslant \frac{1}{2} - 1/n^c$ for some $c > 0$. Consider the general case where prediction error must be $\varepsilon \leqslant \frac{1}{2} - \delta$, where δ possibly depends on n. Choose $x \in \{0,1\}^n$ uniformly at random thus fixing the target concept $\{y \in \{0,1\}^k : g_x(y) = 1\}$. Suppose that a total of $M = u \cdot 2^k$, for some $0 < u < 1$, different labeled examples were obtained during the learning stage via sampling and/or membership queries. Then with probability $1-u$ the newly chosen random prediction challenge y is not among these labeled examples, and in this case the probability that any polynomial time learning algorithm wrongly predicts $g_x(y)$ must be $> \frac{1}{2} - 1/n^{c'}$ for any constant $c' > 0$. (Otherwise the learner would successfully be able to differentiate between g_x and a random function.) In general, if the prediction error probability ε must be $\leqslant \frac{1}{2} - \delta$, then, solving the inequality

$$\frac{1}{2} - \delta \geqslant \varepsilon > (1-u)\left(\frac{1}{2} - \frac{1}{n^{c'}}\right)$$

for u, we see that the number of needed samples must be greater than $(1-(1-2\delta)n^{c'}/(n^{c'}-2)) \cdot 2^k$ for any constant $c' > 0$ and, hence, greater than $\delta 2^k$. (In the case of weak learning, we have $\delta = 1/n^c$ for some $c > 0$.)

Consider the case when $k = \omega(\log n)$. We would like to say that the number of samples needed to learn $\mathcal{G}_{n,k}$ on the uniform distribution is at least $\delta 2^k$. But just taking $\delta 2^k$ random samples requires more than polynomial time, while the security of a pseudo-random function generator was defined in terms of polynomial (in n) time tests. However, if we assume that for the pseudo-random function generator g any test with running time bounded by $n^b 2^{bk}$ (for all constants $b > 0$) has success probability less than $1/n^c$ (for all constants $c > 0$), then we can still claim that any such algorithm requires at least $\delta 2^k$ different labeled examples to learn $\mathcal{G}_{n,k}$.

The goal of this paper is to show that boolean circuits of small depth can evaluate the concept class $\mathcal{G}_{n,k}$ when a specific cryptographic assumption is used to construct the underlying pseudo-random function generator. Then Observation 1 allows us to conclude that the classes of such boolean circuits require super-polynomial numbers of labeled examples and are not polynomially predictable over the uniform distribution, with or without membership queries.

3.3. *The Original Construction*

Goldreich, Goldwasser, and Micali [GGM86] constructed a pseudo-random function generator using as a basic building block a pseudo-random generator f that stretches a n-bit long random string into a $2n$-bit long pseudo-random string.

When used to implement a n, k, n pseudo-random function generator, their construction can best be visualized as a rooted complete binary tree of depth k whose nodes are labeled with n-bit strings and whose edges are labeled 0 or 1. Given a n-bit string x, let $f_0(x)$ denote the first n bits of $f(x)$ and let $f_1(x)$ denote the last n bits of $f(x)$ (recall that $f(x)$ is $2n$ bits long.) To compute a particular function g_x, the root of this tree is labeled with the n-bit string x. Now all nodes and edges of the tree can be labeled in the following manner: if an internal node v is labeled with a string z, then v's left child v_l is labeled with $f_0(z)$ and v's right child v_r is labeled with $f_1(z)$. The edge (v, v_l) is labeled with 0 and the edge (v, v_r) is labeled with 1. The n bit string $g_x(y)$ is the label of the leaf that is reached from the root via the path labeled with the k-bit string y.

This tree, of course, is of exponential (in k) size and is never explicitly constructed in its entirety. When traversing it, we compute children's labels "on the fly" using the label of their parent and the pseudo-random generator f.

In [GGM86] it is shown that if the underlying pseudo-random generator is secure, then such a construction leads to a cryptographically secure pseudo-random function generator.

THEOREM 1 (Goldreich, Goldwasser, and Micali). *Assuming the underlying pseudo-random generator is secure, the function $g(x, y)$ constructed above is a n, k, n pseudo-random function generator.*

To prove this theorem, they introduced a so-called hybrid argument, which shows how to transform a successful test for a pseudo-random function generator constructed above into a successful statistical test for the underlying pseudo-random generator.

In the construction outlined above we fixed $m = n$. In the case where $m < n$, including $m = 1$ which is used for our learning application, we can simply truncate the labels of the leaves to the desired length. If $m > n$ is desired, the pseudo-random generator f can be used to expand the labels of the leaves to the desired length without affecting the pseudo-randomness of the function generator.

3.4. *A Shallow Construction*

It was remarked in [GGM86] that computing $g_x(y)$ on n-bit input x and k bit input y requires $k \cdot T_n$ steps, where T_n denotes the number of steps necessary for the computation of $f(y)$, a pseudo-random generator that stretches n random bits into $2n$ pseudo-random bits. When the circuit model of computation is used, the number of steps translates into the depth of the circuit computing $g_x(y)$. In order to be polynomially unpredictable, a concept (and its complement) must contain a super-polynomial number of examples, hence the

length of the longest examples must be $\omega(\log n)$. Therefore, the original construction is not suitable for showing unpredictability of circuits of logarithmic depth.

Levin [Lev87] proposed to cryptographically hash the inputs to g_x in order to reduce the number of steps by reducing the depth of the binary tree. For a learning problem where examples are chosen from the uniform distribution (without membership queries), however, this is equivalent to simply padding the shorter inputs from k bits to n.

In order to reduce the number of steps in the computation of g_x, and thus the depth of the hard to predict circuits, we will use a pseudo-random generator f with stretch $l(n) = n^2$ to implement a n, k, n pseudo-random function generator. This transforms the binary tree of depth k in the original construction into a complete rooted n-ary tree of depth $d = k/\log n$. The nodes are still labeled with n-bit strings, but the edges are now labeled with $\log n$-bit strings which are concatenated to form a label of a path from the root to a leaf. For simplicity and without loss of generality, we assume n to be a power of 2 and assume $k = d \log n$ for some (not necessarily constant) integer d. For any n-bit string z let $f_0(z)$ denote the first n bits of the n^2-bit long string $f(z)$, $f_1(z)$ the second n bits of $f(z)$, and so on until $f_{n-1}(z)$. Similarly, for any n^2 bit string w let w_0 denote the first n bits of w, w_1 the second n bits of w, and so on until w_{n-1}.

To compute a particular function g_x, the root of the generator tree is still labeled with the n-bit string x. The nodes and edges of the tree can now be labeled as follows: if an internal node v is labeled with a string z, then v's leftmost child v_0 is labeled with $f_0(z)$, the second child from the left is labeled with $f_1(z)$, and so on until the rightmost child (v has n children in total) which is labeled with $f_{n-1}(z)$. Each edge (v, v_i), where i ranges from 0 to $n-1$, is labeled with the number i written as a $\log n$-bit binary string. The n bit string $g_x(y)$ is the label of the leaf that is reached from the root via the path of length d whose labels, when concatenated, form the k-bit string y. Formally, for any k-bit string w let w^0 denote the number between 0 and $n-1$ whose binary expansion appears as the first $\log n$ bits of w; let w^1 denote the number between 0 and $n-1$ whose binary expansion appears as the second $\log n$ bits of w; and so on until w^{d-1}, whose binary expansion appears as the last $\log n$ bits of w. Then

$$g_x(y) = f_{x^{d-1}}(\cdots(f_{x^1}(f_{x^0}(y)))\cdots).$$

Once again, the tree never needs to be explicitly constructed in its entirety. The labels of children $v_0, v_1, \ldots, v_{n-1}$ of a node v are computed from the label of v during the traversal from the root to a leaf.

THEOREM 2. *Assuming that the underlying pseudo-random generator is secure, the function $g(x, y)$ constructed above is a n, k, n, pseudo-random function generator.*

Sketch of Proof. The hybrid argument applies with minor modifications. It shows how to transform a successful test for a pseudo-random function generator g constructed above into a successful statistical test for the underlying pseudo-random generator f with stretch n^2. The complete proof is presented in the Appendix. ∎

For our application we will use only a single bit of the output of $g_x(y)$ to obtain a n, k, 1 pseudo-random function generator.

Recall that at the end of Section 3.2 we mentioned that in order to claim a $O(2^k)$ lower bound on the number of samples needed to learn a representation class $\mathscr{G}_{n,k}$ based on a pseudo-random function generator, we need to assume that this generator is secure against all tests with running time $O(\text{poly}(n2^k))$. From the proof of Theorem 2 it is clear that this is the case if all tests for the underlying pseudo-random generator with running time $O(\text{poly}(n2^k))$ have success probability less than $1/\text{poly}(n2^k)$.

4. THE SUBSET SUM PROBLEM

This section is a partial summary of results presented in [IN89]; we generally adopt the definitions and notation established there.

4.1. *The Basics*

The subset sum problem of dimension $n \times l(n)$ is defined as follows: given n numbers, $\mathbf{a} = (a_1, a_2, \ldots, a_n)$, each $l(n)$ bits long, and a number T, find a subset $S \subseteq \{1, \ldots, n\}$ such that $\sum_{i \in S} a_i = T \bmod 2^{l(n)}$. Alternatively, the problem can be restated as that of inverting the function

$$f(\mathbf{a}, S) = \mathbf{a}, \sum_{i \in S} a_i \bmod 2^{l(n)}.$$

In the applications we consider, the numbers $a_1, a_2, \ldots, a_n$ can be chosen once and fixed thereafter, allowing f to be viewed as a function mapping an n bit string S to an $l(n)$ bit string; we will denote it $f_{\mathbf{a}}(S)$ instead of $f(\mathbf{a}, S)$. Thus the computation of $f_{\mathbf{a}}(S)$ requires the addition of n numbers fixed in advance, each number of length $l(n)$. Such a computation can be performed by a NC^1 circuit, i.e., a circuit of fan-in 2 and depth $O(\log n)$.

Recall that a function f defined on inputs of size n is called a *one-way function* if it is computable in time polynomial in n and is hard to invert on random inputs. More formally, it is required that for any randomized algorithm A with polynomial expected running time and for all c,

$$\Pr_{x \in \{0,1\}^n}[f(A(f(x))) = f(x)] < 1/n^c,$$

where $x \in \{0, 1\}^n$ is chosen uniformly at random.

When we say that the $n \times l(n)$ subset sum is hard we mean that the corresponding function f is a one-way function. The many interesting properties of the subset sum are discussed in [IN89].

4.2. *The Hardness of the Subset Sum*

While the subset sum problem is $\mathcal{NP}$-hard, this fact does not imply hardness of a random instance. In fact, several well-known $\mathcal{NP}$-hard problems can be solved efficiently on a randomly chosen instance. The hardness of an instance of the subset sum problem where both $\mathbf{a}$ and the subset S are chosen uniformly at random has been studied extensively, since many cryptographic schemes based on the subset sum have been proposed in the past. Only one of these schemes has not yet been broken (it is due to Chor and Rivest [CR88]). These schemes, however, use the subset sum as a *trapdoor* function, providing on purpose an easy way to solve the subset sum with the knowledge of a secret key. The trapdoor property is not needed in our application of the subset sum (as well as in other applications discussed in [IN89]); in fact, the general subset sum problem defined in Section 4.1 is not believed to be a trapdoor.

The difficulty of solving a random instance of the subset sum seems to depend on the value of $l(n)$. For the case $l(n) > n^2$, Lagarias and Odlyzko [LO85] and Brickell [Br83] have constructed an efficient algorithm which solves this problem for almost all instances. Their algorithms use the lattice base reduction algorithm of Lenstra, Lenstra, and Lovasz [LLL82] to find the shortest vector in a lattice. It is at this stage that the $l(n) > n^2$ condition appears.

In [IN89] it is shown that the $n \times n$ subset sum is the most secure. For the applications that require $l(n) > n$, they propose to use $l(n) = cn$, where $1 < c \leqslant \frac{3}{2}$. For our purposes we will need to use $l(n) = n^{1+\varepsilon}$ for any constant $\varepsilon > 0$, where ε can be arbitrarily small. When we talk about $n \times n^{1+\varepsilon}$ subset sum we assume some "very small" constant ε. How secure is $n \times n^{1+\varepsilon}$ subset sum? For small values of ε no algorithm significantly better than exhaustive search is currently known to solve this problem. Even though for cryptographic applications such a subset sum is not believed to be as secure as, say, factoring Blum integers, it seems highly unlikely that a general-purpose polynomial, or even quasi-polynomial time learning algorithm can be used to solve for all $\varepsilon > 0$, a random instance of $n \times n^{1+\varepsilon}$ subset sum.

Using the hidden bit result of Goldreich and Levin [GL89] in an elegant and novel way, Impagliazzo and Naor prove in [IN89] the following theorem.

THEOREM 3 (Impagliazzo and Naor). *For $l(n) > n$, if the $n \times l(n)$ subset sum is a one-way function, then it is aslo a pseudo-random generator with stretch factor $l(n)$.*

Therefore we can use the subset sum as a building block to construct a very efficient pseudo-random function generator.

5. PREDICTION HARDNESS FOR BOOLEAN CIRCUITS

In this section we first describe how the subset sum can be used to implement a pseudo-random function generator that can be evaluated by boolean circuits of low depth. We then discuss the implications of this construction for the learnability of specific classes of circuits on the uniform distribution.

5.1 *Implementing a Pseudo-Random Function Generator with the Subset Sum*

We want to use the subset sum to implement the pseudo-random function generator construction described in Section 3.4. Recall that there we used a pseudo-random generator with stretch n^2 as a building block. We cannot directly use the $n \times n^2$ subset sum because, as mentioned in Section 4.2, $n \times n^2$ subset sum is not very secure. We could use the $n \times n^{1+\varepsilon}$ subset sum directly, but it is also possible to first use it to construct the desired pseudo-random generator with stretch n^2. We accomplish this by *cascading* $\lceil 1/\varepsilon \rceil$ subset sums of dimensions $n \times n^{1+\varepsilon}$, $n^{1+\varepsilon} \times n^{1+2\varepsilon}$, ..., $n^{2-\varepsilon} \times n^2$ and using the output of each instance as the input of the next one. Let $f_{\mathbf{a}_i}(S_i)$ where $1 \leqslant i \leqslant \lceil 1/\varepsilon \rceil$, denote the function computed by the ith subset sum on the list. Then, since $f_{\mathbf{a}_i}(S_i)$ is a pseudo-random generator for each i, it is easy to see that the function $f_{\mathbf{a}_1, \ldots, \mathbf{a}_{\lceil 1/\varepsilon \rceil}}: \{0,1\}^n \to \{0,1\}^{n^2}$ defined as $f_{\mathbf{a}_1, \ldots, \mathbf{a}_{\lceil 1/\varepsilon \rceil}} = f_{\mathbf{a}_{\lceil 1/\varepsilon \rceil}}(\cdots(f_{\mathbf{a}_1}(S_1))\cdots)$ is a pseudo-random generator with stretch n^2. Denote $f_{\mathbf{a}_1, \ldots, \mathbf{a}_{\lceil 1/\varepsilon \rceil}}$ by $f_{\mathbf{a}}$. Note that since there is a constant number of $f_{\mathbf{a}_i}$'s and each one of them is computed by a $O(\log n)$ depth fan-in 2 circuit, i.e., in NC^1, then so is $f_{\mathbf{a}}$. It is easy to see (via a hybrid argument) that any test for $f_{\mathbf{a}}$ can be used to obtain a test for one of the $f_{\mathbf{a}_i}$'s, with the success probability decreasing by just a constant factor. Thus we can obtain a pseudo-random generator with stretch n^2 without any loss of security.

Another possibility is to use in this construction the $n \times (1+\varepsilon)n$ subset sum for some $\varepsilon > 0$, which is probably much more secure than the $n \times n^{1+\varepsilon}$ subset sum. In this case, to obtain a pseudo-random generator with stretch n^2, we need to cascade a total of $\log n / \log(1+\varepsilon)$ subset sums. Hence the pseudo-random generator is computed by a $O(\log^2 n)$ depth fan-in 2 circuit, i.e., in NC^2. Once again, it is easy to see that the resulting pseudo-random generator is as secure as its components (the corresponding success probabilities are within a factor of $\text{poly}(\log n)$.)

Now we can use $f_{\mathbf{a}}$ as described in Section 3.4 to implement a n, k, 1 pseudo-random function generator g. What will then be the depth of a fan-in 2 circuit necessary to compute $g_x(y)$? Recall that to compute $g_x(y)$ we must traverse a path of length $k/\log n$. At each node we must compute the labels of its children using the pseudo-random generator f, which requires a circuit of depth $O(\log n)$. We then must chose one of the children based on the value of $\log n$ bits of

x, which clearly can also be done by a circuit of depth $O(\log n)$. Therefore $g_x(y)$ can be computed by a fan-in 2 circuit of depth $O(k)$. Hence we have our main theorem:

THEOREM 4. *For the n, k, 1 pseudo-random function generator g based on the hardness of the $n \times n^{1+\varepsilon}$ subset sum, where $k = \Omega(\log n)$, the concept class $\mathcal{G}_{n,k} = \{\langle y, x\rangle : y \in \{0,1\}^n, x \in \{0,1\}^k, g_x(y) = 1\}$ can be evaluated by a family of polynomial (in n) size fan-in 2 boolean circuits of depth $O(k)$. If the $n \times (1+\varepsilon)n$ subset sum is used, the depth of polynomial size fan-in 2 boolean circuits that can evaluate $\mathcal{G}_{n,k}$ is $O(k \log n)$.*

5.2. *Predictability of Specific Circuits*

From Observation 1 and Theorem 4, assuming that the $n \times n^{1+\varepsilon}$ subset sum is hard, we can obtain sample lower bounds for predicting boolean circuits of specific depths on the uniform distribution (and other specific distributions as well). These bounds remain the same whether membership queries are allowed or not. As discussed in Section 3.2, strong lower bounds require stronger assumptions on the hardness of the subset sum. In this section, when we say that a subset sum problem of a given dimension is $r(n)$-hard, we mean that any algorithm whose (expected) running time is bounded by $\text{poly}(r(n))$ has less than $1/\text{poly}(r(n))$ probability of successfully inverting the random instance of the problem. When $r(n)$ is ommitted, by default $r(n) = n$. In the subsequent discussion k is always less than n, so we can pad k-bit long examples to n bits, with the last $n-k$ bits ignored by the target circuits.

5.2.1. *AC^1 Circuits.* Let $k = \Theta(\log n \log\log n)$. Then by Theorem 4 the n, k, 1 pseudo-random function generator g can be evaluated by polynomial size fan-in 2 boolean circuits of depth $O(\log n \log\log n)$, which in turn can be transformed in a straight forward way into polynomial size circuits of depth $O(\log n)$ with unrestricted fan-in, i.e., AC^1. Hence, by Observation 1, learning AC^1 circuits with probability of error $\leqslant \frac{1}{2} - \delta$ requires $\Omega(\delta \cdot n^{\log\log n})$ labeled samples, i.e., a super-polynomial number of samples even for weak learning.

COROLLARY 1. *Weakly learning AC^1 circuits on the uniform distribution with a polynomial number of samples is as hard as solving $n \times n^{1+\varepsilon}$ subset sum. Assuming that $n \times n^{1+\varepsilon}$ subset sum is $n^{\log\log n}$-hard, weakly learning AC^1 circuits on the uniform distribution requires $\Omega(n^{\log\log n})$ different labeled samples.*

5.2.2. *NC^1 Circuits.* Since it takes a circuit of depth $O(\log n)$ to compute a subset sum, NC^1 is the smallest class for which our technique can be applied. Let $k = c \log n$ for a constant c. Theorem 4 states that the depth of fan-in 2 boolean circuits that represent $\mathcal{G}_{n,k}$ is $O(k) = O(\log n)$. Let us take a closer look at the size of these circuits. For each value of n, the circuit representing $\mathcal{G}_{n,k}$ consists of $c = k/\log n$ identical sub-circuits, each sub-circuit consisting of a n to n^2 pseudo- random generator and a "multiplexer," which is a circuit that takes $\log n$ input bits and uses these bits to choose one of n consecutive n-bit substrings from the n^2-bit long output of the pseudo random generator. Clearly the size and the depth of such a sub-circuit does not depend on k. Let the size of the sub-circuit be bn^a and the depth be $d \log n$ for some some constants a, b, d. Hence the size of the total circuit for $\mathcal{G}_{n,k}$ is cbn^a and the depth is $cd \log n$.

By Observation 1 predicting these circuits on the uniform distribution with probability of error $\leqslant \frac{1}{2} - \delta$ requires more than $\delta 2^k = \delta n^c$ labeled examples. The significance of this is that for a fixed n the size of the target circuits depends linearly on c, but the number of necessary samples depends exponentially on c. In effect, we thus show that for any constants c and c', there is a parameterized family of NC^1 circuits $\{C_n : n \in \mathbf{N}\}$ such that for all n large enough and assuming that the $n \times n^{1+\varepsilon}$ subset sum is hard, the circuit C_n of size $s = \text{poly}(n)$ is not learnable with $O(s^c, n^{c'})$ labeled n-bit long examples. But, by definition of polynomial predictability, prediction algorithm's running time (and hence the number of samples) must be polynomial in the *size* of the target concept representation as well as the length of examples. Therefore we have shown the following.

COROLLARY 2. *Assuming that solving $n \times n^{1+\varepsilon}$ subset sum is hard, there exists no algorithm that predicts NC^1 circuits with a polynomial number of samples.*

What are the implications of Corollary 2 on the learnability of boolean formulas? The concept class represented by polynomial size boolean formulas is exactly the same as the class evaluated by NC^1 circuits. The transformation from NC^1 circuits to boolean formulas provides for each circuit a formula equivalent to it whose size is exponential in the *depth* of the circuit. In case of NC^1 this results in a formula of polynomial size, but the argument that proves Corollary 2 results in a bound on the number of samples necessary to learn a boolean formula which is only linear in the size of the target formula. Thus Corollary 2 does not provide a useful lower bound for the learnability of boolean formulas.

5.2.3. *NC Circuits.* Let $k = \Theta(\log^i n)$. Then by Theorem 4 the n, k, 1 pseudo-random function generator g can be evaluated by polynomial size fan-in 2 boolean circuits of depth $O(\log^i n)$; i.e., in NC^i. Hence, by Observation 1, learning NC^i circuits with probability of error $\leqslant \frac{1}{2} - \delta$ requires $\Omega(\delta \cdot n^{\log^{i-1} n})$ labeled samples.

Now instead of $n \times n^{1+\varepsilon}$ let us assume that $n \times (1+\varepsilon)n$ subset sum is hard (as discussed in Section 4.2, this version of the subset sum problem appears to be more secure) and use it to construct the n, k, 1 pseudo-random function generator as described in Section 5.1. Then by Theorem 4 this

generator can be evaluated by polynomial size fan-in 2 boolean circuits of depth $O(\log^{i+1} n)$, i.e., in NC^{i+1}. Learning these circuits with probability of error $\leqslant \frac{1}{2} - \delta$ still requires $\Omega(\delta \cdot n^{\log^{i-1} n})$ labeled samples. Recall that the class NC is defined as $NC = \bigcup NC^i$ for all i. Therefore we have the following corollary.

COROLLARY 3. *For any constant $c > 0$, assuming $n \times (1+\varepsilon)n$ subset sum is $n^{\log^c n}$-hard, weakly learning NC circuits on the uniform distribution requires $\Omega(n^{\log^c n})$ labeled samples.*

6. CONCLUSIONS AND OPEN PROBLEMS

In this paper we used a cryptographic assumption, the hardness of the subset sum problem, and a cryptographic primitive, the pseudo-random function generator, to prove lower bounds on the number of samples (and thus the amount of time) needed to successfully predict classes of boolean functions on the uniform distribution. These bounds hold when the learner is allowed membership queries. Recently, a new technique was developed in [Kh93] that allows us to extend hardness results for the uniform distribution to other non-trivial distributions. It is easy to show (although we do not describe it here) that this technique applies directly to the proofs in this paper and can be used to extend our results to *all* non-trivial distributions.

The results presented here can be improved in a number of ways. The biggest challenge is to prove similar bounds for circuits of lower depth. Perhaps the method (described in [IN89]) which uses the subset sum to implement a pseudo-random generator in AC^0 can be used for this purpose. While some success in this direction was achieved in [Kh93] using an explicit sub-exponential assumption on the hardness of factoring, the subset sum assumption is a very promising alternative. Also, stronger lower bounds for AC^1 and NC^1 are desirable.

A major effort in the field of cryptography resulted recently in moving from reliance on specific cryptographic assumptions, such as the hardness of inverting the subset sum or factoring large numbers, to general cryptographic assumptions, such as the existence of one-way functions and trapdoor functions. Moving from specific to general cryptographic assumptions for the purpose of showing the hardness of learning is an important task for future research. An even more ambitious goal is to move from cryptographic to general complexity assumptions, such as $\mathcal{R} \neq \mathcal{NP}$.

APPENDIX

Proof of Theorem 2. The proof shows how to transform a successful test for a pseudo-random function generator g constructed in Section 3.4 into a successful statistical test for the underlying pseudo-random generator f with stretch n^2. We closely follow the proof for the original construction of [GGM86] as presented in [Luby92].

Given a binary string v and a *number* $0 \leqslant u \leqslant n-1$, let $v * u$ denote the string formed by concatenating v and the log n-bit long binary expansion of u. For example, $v * 3$ denotes the string v followed by $\log n - 2$ zeroes followed by two ones; $f_1(z) * w^1$ consists of bits $n+1$ through $2n$ from the string $f(z)$ followed by bits 1 through $\log n$ from the string w. Also let $v_{\rightarrow i}$ denote the i log n-bit long prefix of v.

We can now describe our construction of the pseudo-random function generator g as follows. Let $\lambda \in \{0,1\}^0$ denote the empty string; let

$$\{0,1\}^{\leqslant m} = \bigcup_{i=0}^{\lceil m/\log n \rceil} \{0,1\}^{i \log n}.$$

(Recall that for simplicity and without loss of generality, we assumed n to be a power of 2.) Given $x \in \{0,1\}^n$ define the function G_x: $\{0,1\}^{\leqslant k} \to \{0,1\}^n$ inductively on the length of its input as follows:

- $G_x(\lambda) = x$.
- Let $y \in \{0,1\}^{\leqslant (k - \log n)}$. We call the n-tuple $(y * 0, y * 1, \ldots, y * (n-1))$ of strings from $\{0,1\}^{\leqslant k}$ the *children* of y and denote it by $\hat{y}$; we call y the parent of $\hat{y}$. Once $G_x(y)$ is defined, we define the value of G_x on the ith child $y * (i-1)$ of y to be $G_x(y * (i-1)) = f_{i-1}(G_x(y))$.

Now given $x \in \{0,1\}^n$ and $y \in \{0,1\}^k$ we define $g(x, y) = G_x(y)$ which can be computed by successively computing for all $1 \leqslant i \leqslant d$ (recall that we assumed without loss of generality that $k = d \log n$) the value of $G_x(\lambda * y^1 * \cdots * y^i)$ as described above.

Assume that T is a polynomial time statistical test which is successful for the pseudo-random function generator g. We use it to construct an oracle machine S^T which is a successful polynomial time statistical test for the pseudo-random generator f. Let $\delta(n)$ be the success probability of T for g. Let

$$q_0 = \Pr_{x \in \{0,1\}^n}[T^{g_x}(1^n, 1^k, 1^n) = 1]$$

and

$$q_1 = \Pr_{h'' \in H_{k,n}}[T^{h''}(1^n, 1^k, 1^n) = 1],$$

where $x \in \{0,1\}^n$ and $h'' \in H_{k,n}$ are chosen uniformly at random. Then, without loss of generality, $\delta(n) = q_0 - q_1$. Let $m(n)$ be a polynomial that bounds the maximum (over all functions $h'' \in H_{k,n}$) number of queries made by $T^{h''}(1^n, 1^k, 1^n)$. (Actually, the number of queries only must be polynomial on the average. This can be addressed in the proof, but would make it more complicated.)

Now let us describe the execution of S^T on an input $z \in \{0,1\}^{n^2}$. S^T on an input $z \in \{0,1\}^{n^2}$. S^T simulates an

execution of $T^{h'}$, where $h' \in H_{k,n}$ is a hybrid between a random function $h'' \in H_{k,n}$ and g_x for a random x. Actually, in the process of answering the oracle calls of $T^{h'}$, S^T will be computing a more general function h: $\{0,1\}^{\leqslant k} \to \{0,1\}^n$ which is a hybrid between a random function H: $\{0,1\}^{\leqslant k} \to \{0,1\}^n$ and G_x. Initially, S^T chooses a random $s \in \{0,1\}^n$ and sets $h(\lambda) = s$. The values of h on other inputs are computed inductively during the simulation of T. Let $y \in \{0,1\}^k$ be an input to the oracle call made by T. To compute $h(y)$, S^T computes in sequence the values of h on the elements of the following d n-tuples:

$$\langle (0, 1, ..., n-1), (y_{\wp 1} * 0, y_{\wp 1} * 1, ..., y_{\wp 1} * (n-1)), ...,$$
$$(y_{\wp d-1} * 0, y_{\wp d-1} * 1, ..., y_{\wp d-1} * (n-1)) \rangle.$$

The elements of each n-tuple are always processed together. Somewhere during the computation of h on the last n-tuple, S^T computes $h(y)$, which is the output of the oracle call. To be consistent, S^T stores all computed input/output pairs of h; whenever it needs to compute h on some input $u \in \{0,1\}^{\leqslant k}$, S^T first checks if $h(u)$ was already computed and, if so, outputs the previously stored value.

During the simulation of T, sometimes S^T has to compute the values of h on the elements of the n-tuple $\hat{u}$ which was not previously seen. We call it a *new n-tuple*, and we order all n-tuples by the order in which they occur in the simulation. Note that when a new n-tuple $\hat{u}$ occurs during the simulation, the value of $h(u)$ on its parent u has already been computed. Since for every query of T there are at most d new n-tuples on which h must be computed, we can order all new n-tuples and say that for $1 \leqslant i \leqslant dm(n)$ we let $\hat{u}_i$ denote the ith new n-tuple. Note that the value of $\hat{u}_i$ depends strongly on the history of this particular simulation.

Initially, S^T picks uniformly at random $1 \leqslant l \leqslant dm(n)$; recall that $h(\lambda) = s$ where s is a random n bit string. Also, S^T chooses $n(l-1)$ random n-bit long strings $r_1, r_2, ..., r_{n(l-1)}$. For all $i = 1, ..., l-1$, when the ith new n-tuple $\hat{u}_i$ occurs during the simulation, for each $j = 0, ..., n-1$ the algorithm S^T sets $h(u_i * j) = r_{(i-1)n+j+1}$. In other words, the value of h on each element of each of the first i n-tuples is set to be a new random n-bit string. When the lth new n-tuple $\hat{u}_l$ occurs during the simulation, for each $j = 0, ..., n-1$ the algorithm S^T sets $h(u_l * j) = z_j$. (Recall that z is the n^2-bit long string that S^T is testing for randomness.) For all $i = l+1, ..., dm(n)$, when the ith new n-tuple $\hat{u}_i$ occurs during the simulation, for each $j = 0, ..., n-1$, the algorithm S^T sets $h(u_i * j) = f_j(h(u_i))$. In other words, after $\hat{u}_l$ is processed, the function h is computed in the same way as G_x.

When the simulation is finished and the test T outputs a bit, S^T outputs the same bit.

For the analysis of S^T, let $X \in \{0,1\}^n$ and $Y \in \{0,1\}^{n^2}$ be uniformly distributed random variables. Let $p_{0,i}$ be the probability that S^T outputs 1 when the distribution on its input z is $f(X)$ and when $l = i$. Let $p_{1,i}$ be the probability that S^T outputs 1 when the distribution on its input z is Y and when $l = i$. Then the total probability that S^T outputs 1 when the distribution on its input z is $f(X)$ is

$$p_0 = \frac{1}{dm(n)} \sum_{i=1}^{dm(n)} p_{0,i}$$

while the total probability that S^T outputs 1 when the distribution on its input z is Y is

$$p_1 = \frac{1}{dm(n)} \sum_{i=1}^{dm(n)} p_{1,i}.$$

We now prove the following claims:

1. For all $i = 1, ..., dm(n)-1$ we have $p_{0,i+1} = p_{1,i}$. This claim follows from the following observations:
   - When $l = i+1$ and z is distributed according to $f(X)$:
     - For the ith new n-tuple: S^T assigns the values of h on the elements of the n-tuple uniformly at random.
     - For the $(i+1)$th new n-tuple: S^T uses pieces of its input $f(X)$ to assign the values of h on the elements of the n-tuple.
   - When $l = i$ and z is distributed according to Y:
     - For the ith new n-tuple: S^T uses pieces of its random input Y to assign the values of h on the elements of the n-tuple.
     - For the $(i+1)$th new n-tuple: S^T uses pieces of the value $f(h(u_i))$ to assign the values of h on the elements of the n-tuple $\hat{u}_{i+1}$, where $h(u_i)$ was previously assigned at random (either to a substring Y_j of the random input Y or to a random value chosen by S^T.)

2. $p_{0,1} = q_0$. This is true because the simulation of T by S^T in this case is the same as if $h(\lambda) = X$ and the rest of the values of h are computed in the same way as the values of G_X. Thus the oracle calls of T are answered according to g_X.

3. $p_{1,dm(n)} = q_1$. This is true because the total number of oracle calls made by T during the simulation is at most $m(n)$ and for each call there are at most d new n-tuples, hence there are at most $dm(n)$ new n-tuples in total. Since in this case S^T assigns the values of h on the elements of all new n-tuples uniformly at random, the oracle calls of T are being evaluated according to a random function.

Let $\delta'(n)$ be the success probability of S^T for distinguishing $f(X)$ from Y. Then

$$\delta'(n) = p_0 - p_1 = \frac{d}{dm(n)} \sum_{i=1}^{dm(n)} (p_{0,i} - p_{1,i})$$
$$= \frac{p_{0,1} - p_{1,dm(n)}}{dm(n)} = \frac{\delta(n)}{dm(n)}.$$

Therefore, S^T is a successful statistical test for the pseudo-random generator f. ∎

ACKNOWLEDGMENTS

I am grateful to Moni Naor for many useful discussions concerning the properties and applications of the subset sum. Thanks to Yoav Freund, Moti Yung, and the anonymous referee for helpful comments. I thank the International Computer Science Institute, Berkeley, where some of this work was done.

REFERENCES

[AK91] D. Angluin and M. Kharitonov, When won't membership queries help?, in "Proceedings, 23d ACM Symposium on Theory of Computing," pp. 444–454, ACM, New York, 1991.

[Blum89] A. Blum, Separating distribution-free and mistake-bounded learning models over the Boolean domain, *in* "Proceedings, 31st IEEE Symposium on Foundations of Computer Science," pp. 211–218, IEEE, New York, 1990.

[Br83] E. F. Brickell, Solving low density knapsacks, *in* "Proceedings, Crypto 83," pp. 25–37.

[Buss87] S. R. Buss, The Boolean formula value problem is in Alogtime, *in* "Proceedings, 19th Annual ACM Symposium on Theory of Computing," pp 123–131, ACM, New York, 1987.

[CR88] B. Chor and R. L. Rivest, A knapsack type public key cryptosystem based on arithmetic in finite fields, *IEEE Trans. Inform. Theory* **34** (1988), 901–909.

[GGM86] O. Goldreich, S. Goldwasser, and S. Micali, How to construct random functions, *J. Assoc. Comput. Mach.* **33** (1986), 792–807.

[GL89] O. Goldreich and L. Levin, A hard-core predicate for all one-way functions, *in* "Proceedings, 21st Symposium on the Theory of Computing, 1989."

[Has86] J. Hastad, "Computational Limitations for Small Depth Circuits," MIT Ph.D. thesis, MIT Press, Cambridge, MA, 1986.

[IN89] R. Impagliazzo and M. Naor, Efficient cryptographic schemes provably as secure as subset sum, *in* "Proceedings, 30th IEEE Symposium on Foundations of Computer Science," pp. 236–243, IEEE, New York, 1989.

[KV89] M. Kearns and L. Valiant, Cryptographic limitations on learning boolean formulae and finite automata, *in* "Proceedings, 21st ACM Symposium on Theory of Computing," pp. 433–444, ACM, New York, 1989.

[Kh93] M. Kharitonov, Cryptographic hardness of distribution-specific learning, *in* "Proceedings, 25th ACM Symposium on Theory of Computing," pp. 372–381, ACM, New York, 1993.

[LO85] J. C. Lagarias and A. M. Odlyzko, Solving low density subset sum problems, *J. Assoc. Comput. Mach.* **32** (1985), 229–246.

[LLL82] A. K. Lenstra, H. W. Lenstra, and L. Lovasz, Factoring polynomials with rational coefficients, *Math. Ann.* **261** (1982), 515–534.

[Lev87] L. Levin, One way functions and pseudorandom generators, *Combinatorica* **7**, No. 4 (1987), 357–363.

[LMN89] N. Linial, Y. Mansour, and N. Nisan, Constant depth circuits, Fourier transform, and learnability, *in* "Proceedings, 30th IEEE Symposium on Foundations of Computer Science," pp. 574–579, IEEE, New York, 1989.

[Luby92] M. Luby, "Pseudo-randomness and Applications," Princeton Univ. Press, Princeton, NJ, 1992.

[Man92] Y. Mansour, An $O(n^{\log\log n})$ learning algorithm for DNF under the uniform distribution, *in* "Proceedings, 1992 Workshop on Computational Learning Theory," Morgan Kaufmann, San Mateo, CA, 1992.

[PW88] L. Pitt and M. Warmuth, Reductions among prediction problems: On the difficulty of predicting automata, *in* "Proceedings, Third Annual Structure in Complexity Theory Conference," pp. 60–69, IEEE Comput. Soc. Press, Washington, DC, 1988.

[Sch89] R. E. Schapire, The strength of weak learnability, *in* "Proceedings, 30th Annual Symposium on Foundations of Computer Science," pp. 28–33, IEEE, New York, 1989.

[Val84] L. G. Valiant, A theory of the learnable, *Comm. ACM* **27** (1984), 1134–1142.