Synthesizers and Their Application to the Parallel Construction of Pseudo-Random Functions *

Moni Naor [†] Omer Reingold [‡]

Abstract

A pseudo-random function is a fundamental cryptographic primitive that is essential for encryption, identification and authentication. We present a new cryptographic primitive called pseudo-random synthesizer and show how to use it in order to get a parallel construction of a pseudo-random function. We show several NC^1 implementations of synthesizers based on concrete intractability assumptions as factoring and the Diffie-Hellman assumption. This yields the first parallel pseudo-random functions (based on standard intractability assumptions) and the only alternative to the original construction of Goldreich, Goldwasser and Micali. In addition, we show parallel constructions of synthesizers based on other primitives such as weak pseudo-random functions or trapdoor one-way permutations. The security of all our constructions is similar to the security of the underlying assumptions. The connection with problems in Computational Learning Theory is discussed.

*A preliminary version of this paper appeared at the *Proc. 35th IEEE Symp. on Foundations of Computer Science* (1995) pp. 170-181

[†]Incumbent of the Morris and Rose Goldman Career Development Chair, Dept. of Applied Mathematics and Computer Science, Weizmann Institute of Science, Rehovot 76100, Israel. Research supported by BSF grant no. 94-00032 and a grant from the Israel Science Foundation administered by the Israeli Academy of Sciences. E-mail: naor@wisdom.weizmann.ac.il.

[‡]Dept. of Applied Mathematics and Computer Science, Weizmann Institute of Science, Rehovot 76100, Israel. Research supported by a Clore Scholars award and by a grant from the Israel Science Foundation administered by the Israeli Academy of Sciences. E-mail: reingold@wisdom.weizmann.ac.il.

1 Introduction

A pseudo-random function, as defined by Goldreich, Goldwasser and Micali [26], is a function that is indistinguishable from a truly random function to a (polynomial-time bounded) observer who can access the function as a black-box (i.e. can provide inputs of his choice and gets to see the value of the function on these inputs). Pseudo-random functions are the key component of private-key cryptography. They allow parties who share a common key to send secret messages to each other, to identify themselves and to authenticate messages [16, 27, 40]. In addition, they have many other applications, essentially in any setting that calls for a random function that is provided as a black-box [9, 12, 19, 23, 24, 41, 51].

Goldreich, Goldwasser and Micali provided a construction of such functions. For roughly a decade, this was the only known construction even under specific assumptions such as "factoring is hard". Their construction is sequential in nature and consists of n successive invocations of a pseudo-random generator (where n is the number of bits in the input to the function). Our goal in this paper is to present an alternative construction for pseudo-random functions that can be implemented in $\log n$ phases.

We introduce a new cryptographic primitive which we call *pseudo-random synthesizer*. A pseudo-random synthesizer is a two variable function, $S(\cdot,\cdot)$, so that if many (but polynomially bounded) random assignments, $\langle x_1,\ldots,x_m\rangle$ and $\langle y_1,\ldots,y_m\rangle$, are chosen to both variables, then the output of S on all the combinations of these assignments, $(f(x_i,y_j))_{i,j=1}^m$, is indistinguishable from random to a polynomial-time observer. Our main results are:

1. A construction of pseudo-random functions based on pseudo-random synthesizers. Evaluating such a function involves $\log n$ phases, where each phase consists of several parallel invocations of a synthesizer (with a total of n invocations altogether).

2. Constructions of parallel (NC^1) synthesizers based on standard number-theoretic assumptions such as "factoring is hard", RSA (it is hard to extract roots modulo a composite) and Diffie-Hellman. In addition, a very simple construction based on a problem from learning. The key-generating algorithm of these constructions is sequential for RSA and factoring, non-uniformly parallel for Diffie-Hellman and parallel for the learning problem.

3. An extremely simple (and also parallel) construction of synthesizers based on what we call a weak pseudo-random function. A weak pseudo-random function is indistinguishable from a truly random function to a (polynomial-time bounded) observer who gets to see the value of the function on *uniformly distributed inputs* (instead of any input of its choice). This construction almost immediately implies constructions of synthesizers based on trapdoor one-way permutations and based on any hard-to-learn problem (under the definition of [13]).

Taking (1) and (2) together we get a pseudo-random function that can be evaluated in NC^2.

We note that our constructions do not weaken the security of the underlying assumptions. Take for instance the construction that is based on factoring. If there is an algorithm for breaking this construction in time t and success α (success α means that the observer has advantage of at least α in distinguishing the pseudo-random function from the random one), then there is an algorithm that works in time $poly(t)$ and factors Blum-integers with probability $\alpha/poly(t)$. See [32, 40] for a discussion of security preserving reductions [1].

[1] In their terminology, such a reduction is called poly-preserving. In fact, most of our reductions (as the reduction from the security of the pseudo-random functions to the security of the pseudo-random synthesizers) are linear-preserving. The only place were our reductions are not linear-preserving is when they rely on the hard-card bits of [2, 28].

Our constructions of pseudo-random functions have additional attractive properties. First, it is possible to obtain from the constructions a sharp time-space tradeoff. Loosely speaking, by keeping m strings as the key we can reduce the amount of work for computing the functions from n invocations of the synthesizer to about $\frac{n}{\log m}$ invocations in $\log n - \log\log m$ phases (thus, also reducing the parallel-time complexity). In addition, the construction obtains a nice incremental property. For any y of Hamming distance one from x, given the computation of $f(x)$ we can compute $f(y)$ with only $\log n$ invocations of the synthesizer (we can also make this property hold for $y = x + 1$). We discuss both properties in Section 6.

Applications of NC-Computable Pseudo-Random Functions

The class NC has been criticized as a model for parallel computation for two main reasons:

- It ignores communication delays and other parameters that determine the execution time on an actual parallel machine.
- It over-emphasizes latency rather than the speed-up of problems.

These criticisms seem less valid for the problem of constructing pseudo-random functions, since (a) It is likely that it will be implemented in a special purpose circuit (as there are DES chips) and (b) For some applications of pseudo-random functions minimizing the latency of computing the functions is essential. Such an application is the encryption of messages on a network, where the latency of computing the function is added to the latency of the network. Furthermore, if the complexity of evaluating a synthesizer on a given input is comparable to that of a pseudo-random generator, then the work performed by our construction is comparable to the one in [26] and we can get optimal speed-up.

Note that many of the applications of pseudo-random functions preserve the parallel-time complexity of the functions. An important example is the Luby and Rackoff [41] construction of pseudo-random *permutations* from pseudo-random functions. Their construction is very simple and involves four invocations of a pseudo-random function in order to evaluate the pseudo-random permutation at a given point (see also [45] for an "optimal" construction that requires only two invocations). Therefore, our constructions yield (strong) pseudo-random *permutations* in NC as well.

There is a deep connection between pseudo-random functions and hardness results for learning. Since a random function cannot be learned, if a concept class is strong enough to contain pseudo-random functions we cannot hope to learn it efficiently. Since no construction of pseudo-random functions in NC was known, several ways of bypassing this were suggested [3, 37, 38]. However, these are weaker unlearnability-results than the one obtained by pseudo-random functions. The existence of pseudo-random functions in a concept class implies that there exists a distribution of concepts in this class that is hard for *every* learning algorithm, for *every* "non-trivial" distribution on inputs *even* when membership queries are allowed. Finding such a *distribution of concepts* is still of interest to learning theory [33]. We discuss the connection between our work and learning-theory in Section 9.

Another application of pseudo-random functions in complexity was suggested by the work of Razborov and Rudich [53] on Natural Proofs. They showed that if a circuit-class contains pseudo-random functions (that are secure against a subexponential-time adversary) then there are no, what they called, Natural Proofs (which include all known lower bound techniques) for separating this class from $P/poly$. Given our constructions, the existence of Natural Proofs for separating NC from $P/poly$ would imply that several well-established intractability assumptions are false.

The question of whether pseudo-random functions exist in NC is also interesting in contrast to the lower bound of Linial, Mansour and Nisan [39] that there are no pseudo-random functions in AC^0.

Previous Work

In addition to introducing pseudo-random functions, Goldreich, Goldwasser and Micali [26] have suggested a construction of such functions from pseudo-random generators that expand the input by a factor of two (like the one in [34]). As mentioned above, the GGM construction is sequential in nature. An idea of Levin [42] is to select some secret hash function h and apply the GGM construction to $h(x)$ instead of x. If $|h(x)| = \log^2 n$, then the depth of the GGM-tree is only $\log^2 n$ and presumably we get a pseudo-random function in NC. The problem with this idea is that we have decreased the security significantly: with probability $1/n^{\log n}$ the function can be broken, irrespective of the security guaranteed by the pseudo-random generator. To put this construction in the "correct" light, suppose that for security parameter k we have some problem whose solution requires time 2^k (on instance of length polynomial in k). If we would like to have security $1/2^k$ for our pseudo-random function, then the Levin construction requires depth k whereas our construction requires depth $\log k$.

Impagliazzo and Naor [34] have provided parallel constructions for several other cryptographic primitives based on the hardness of subset sum (and factoring). The primitives include pseudo-random generators that expand the input by a constant factor[2], universal one-way hash functions and strong bit-commitments.

Blum et. al. [13] proposed a way of constructing in parallel several cryptographic primitives based on problems that are hard to learn. We extend their result by showing that hard-to-learn problems can be used to obtain synthesizers and thus pseudo-random functions.

A different line of work [1, 4, 47, 48, 49, 50, 54], more relevant to derandomization and saving random bits, is to construct *bit*-generators such that their output is indistinguishable from a truly random source to an observer of restricted computational power (e.g. generators against polynomial-size constant-depth circuits). Most of these constructions need no unproven assumptions.

In a subsequent work [46] we describe constructions of pseudo-random functions (and other cryptographic primitives) that are at least as secure as the decisional version of the Diffie-Hellman assumption or as the assumption that factoring is hard. These functions can be computed in NC^1 (in fact, even in TC^0) and are much more efficient than the concrete constructions of this paper. It is interesting to note that [46] is motivated by this paper and in particular by the concept of pseudo-random synthesizers.

Organization of the Paper

In Section 3 we define pseudo-random synthesizers and collections of pseudo-random synthesizers and discuss their properties. In Section 4 we describe our parallel construction of pseudo-random functions from pseudo-random synthesizers and in Section 5 we prove its security. In Section 6 we describe a related construction of pseudo-random functions. In addition, we discuss the time-space tradeoff and the incremental property of our constructions. In Section 7 we discuss the relations between pseudo-random synthesizers and other cryptographic primitives. In Section 8 we describe constructions of pseudo-random synthesizers based on several number-theoretic assumptions. In Section 9 we show how to construct pseudo-random synthesizers from hard-to-learn problems and

[2] They also provided a construction of AC^0 pseudo-random generators with small expansion.

consider a very simple concrete example. We also discuss the application of parallel pseudo-random functions to learning-theory. In Section 10 we suggest topics for further research.

2 Preliminaries

2.1 Notation

- $\mathbb{N}$ denotes the set of all natural numbers.
- I^n denotes the set of all n-bit strings, $\{0,1\}^n$.
- U_n denotes the random variable uniformly distributed over I^n.
- Let X be any random variable, we denote by $X^{k \times \ell}$ the $k \times \ell$ matrix whose entries are independently identically distributed according to X. We denote by X^k the vector $X^{1 \times k}$.
- We identify functions of two variables and functions of one variable in the natural way. I.e, by letting $f : I^n \times I^n \mapsto I^k$ be equivalent to $f : I^{2n} \mapsto I^k$ and letting $f(x, y)$ be the same value as $f(x \circ y)$ (where $x \circ y$ stands for x concatenated with y).
- Let x be any bit-string, we denote by $|x|$ its length (i.e. the number of bits in x). This should not be confused with the usage of $|\cdot|$ as absolute value.
- For any two bit-strings of the same length, x and y, the inner product mod 2 of x and y is denoted by $x \odot y$.

2.2 Pseudo-Random Functions

For the sake of completeness and concreteness, we briefly review in this section the concept of pseudo-random functions almost as it appears in [25]. Another good reference on pseudo-random functions is [40]. Informally, a pseudo-random function ensemble is an efficient distribution of functions that cannot be efficiently distinguished from the uniform distribution. That is, an efficient algorithm that gets a function as a black box cannot tell (with non-negligible success probability) from which of the distributions it was sampled. To formalize this, we first define function ensembles and efficient function ensembles:

Definition 2.1 (function ensemble) *Let ℓ and k be any two $\mathbb{N} \mapsto \mathbb{N}$ functions. An $I^\ell \mapsto I^k$* function ensemble *is a sequence $F = \{F_n\}_{n \in \mathbb{N}}$ of random variables, such that the random variable F_n assumes values in the set of $I^{\ell(n)} \mapsto I^{k(n)}$ functions. The* uniform $I^\ell \mapsto I^k$ function ensemble*, $R = \{R_n\}_{n \in \mathbb{N}}$, has R_n uniformly distributed over the set of $I^{\ell(n)} \mapsto I^{k(n)}$ functions.*

Definition 2.2 (efficiently computable function ensemble)
A function ensemble, $F = \{F_n\}_{n \in \mathbb{N}}$, is efficiently computable *if there exist probabilistic polynomial-time algorithms, $\mathcal{I}$ and $\mathcal{V}$, and a mapping from strings to functions, ϕ, such that $\phi(\mathcal{I}(1^n))$ and F_n are identically distributed and $\mathcal{V}(i, x) = (\phi(i))(x)$.*

We denote by f_i the function assigned to i (i.e. $f_i \stackrel{\text{def}}{=} \phi(i)$). We refer to i as the key of f_i and to $\mathcal{I}$ as the key-generating algorithm of F.

For simplicity, we concentrate in the definition of pseudo-random functions and in their construction on length-preserving functions. The distinguisher, in our setting, is defined to be an oracle machine that can make queries to a length preserving function (which is either sampled from the pseudo-random function ensemble or from the uniform function ensemble). We assume that on input 1^n the oracle machine makes only n-bit queries. For any probabilistic oracle machine, $\mathcal{M}$, and any $I^n \mapsto I^n$ function, O, we denote by $\mathcal{M}^O(1^n)$ the distribution of $\mathcal{M}$'s output on input 1^n and with access to O.

Definition 2.3 (efficiently computable pseudo-random function ensemble) *An efficiently computable $I^n \mapsto I^n$ function ensemble, $F = \{F_n\}_{n\in\mathbb{N}}$, is* pseudo-random *if for every probabilistic polynomial-time oracle machine $\mathcal{M}$, every polynomial $p(\cdot)$, and all sufficiently large n's*

$$\left|\Pr\left[\mathcal{M}^{F_n}(1^n) = 1\right] - \Pr\left[\mathcal{M}^{R_n}(1^n) = 1\right]\right| < \frac{1}{p(n)}$$

where $R = \{R_n\}_{n\in\mathbb{N}}$ is the uniform $I^n \mapsto I^n$ function ensemble.

At the rest of this paper the term "pseudo-random functions" is used as an abbreviation for "efficiently computable pseudo-random function ensemble".

Remark 2.1 *In the definition above and in the rest of the paper, we interpret "efficient computation" as "probabilistic polynomial-time" and "negligible" as "smaller than $1/poly$". This is a rather standard choice and it significantly simplifies the presentation of the paper. However, from each one of the proofs in this paper one can easily extract a more quantitative version of the corresponding result. As mentioned in the introduction, the different reductions of this paper are security-preserving in the sense of [32, 40].*

3 Pseudo-random Synthesizers

As mentioned above, we introduce in this paper a new cryptographic primitive called a pseudo-random synthesizer. In this section we define pseudo-random synthesizers and describe their properties.

3.1 Motivation

Pseudo-random synthesizers are efficiently computable functions of two variables. The significant feature of such a function, S, is that given polynomially-many uniformly distributed assignments, $\langle x_1, \ldots, x_m\rangle$ and $\langle y_1, \ldots, y_m\rangle$, for both variables, the output of S on all the combinations of these assignments, $(f(x_i, y_j))_{i,j=1}^m$, is pseudo-random (i.e, is indistinguishable from random to a polynomial-time observer). This is a strengthening of an important property of pseudo-random *generators* — the indistinguishability of a polynomial sample:

A pseudo-random (bit) generator [14, 61], is a polynomial-time computable function, $G : \{0,1\}^* \mapsto \{0,1\}^*$, such that $\forall x \in I^n, |G(x)| = \ell(n) > n$ and $G(U_n)$ is pseudo-random (i.e. $\{G(U_n)\}_{n\in\mathbb{N}}$ and $\{U_{\ell(n)}\}_{n\in\mathbb{N}}$ are computationally indistinguishable). It turns out that this definition implies that: Given polynomially-many uniformly distributed assignments, $\langle z_1, \ldots, z_m\rangle$, the sequence $\{(G(z_i)\}_{i=1}^m$, is pseudo-random.

The major idea behind the definition of pseudo-random synthesizers is to obtain a function, S, such that $\{(S(z_i)\}_{i=1}^m$ remains pseudo-random even when the z_i's *are not completely independent.*

More specifically, pseudo-random synthesizers require that $\{(S(z_i)\}_{i=1}^m$ remains pseudo-random even when the z_i's are of the form $\{x_i \circ y_j\}_{i,j=1}^m$. This paper shows that (under some standard intractability assumptions) it is possible to obtain such a function S and that this property is indeed very powerful. As a demonstration to their strength, we note below that pseudo-random synthesizers are useful even when no restriction is made on their output length (which is very different than what we have for pseudo-random generators).

Remark 3.1 *It is important to note that there exist pseudo-random generators that are* not *pseudo-random synthesizers. An immediate example is a generator which is defined by $G(x \circ y) \stackrel{\text{def}}{=} G'(x) \circ y$, where G' is also a pseudo-random generator. A more natural example is the subset-sum generator [34], $G = G_{a_1,a_2,\ldots,a_n}$, which is defined by $G(z) = \sum_{z_i=1} a_i$. This is not a pseudo-random synthesizer (for fixed values $a_1, a_2, \ldots, a_n$) since for every four $n/2$-bit strings, x_1, x_2, y_1 and y_2, we have that $G(x_1 \circ y_1) + G(x_2 \circ y_2) = G(x_1 \circ y_2) + G(x_2 \circ y_1)$.*

3.2 Formal Definition

We first introduce an additional notation to formalize the phrase "all different combinations":

Notation 3.1 *Let f be an $I^{2n} \mapsto I^{\ell}$ function and let $X = \{x_1, \ldots, x_k\}$ and $Y = \{y_1, \ldots, y_m\}$ be two sequences of n-bit strings. We define $\mathbf{C}_f(X, Y)$ to be the $k \times m$ matrix $(f(x_i, y_j))_{i,j}$ ($\mathbf{C}$ stands for combinations).*

We can now define what a pseudo-random synthesizer is:

Definition 3.1 (pseudo-random synthesizer) *Let ℓ be any $\mathbb{N} \mapsto \mathbb{N}$ function and let $S : \{0,1\}^* \times \{0,1\}^* \mapsto \{0,1\}^*$ be a polynomial-time computable function such that $\forall x, y \in I^n, |S(x,y)| = \ell(n)$. Then S is a* pseudo-random synthesizer *if for every probabilistic polynomial-time algorithm, $\mathcal{D}$, every two polynomials $p(\cdot)$ and $m(\cdot)$, and all sufficiently large n's*

$$\left|\Pr\left[\mathcal{D}(\mathbf{C}_S(X,Y)) = 1\right] - \Pr\left[\mathcal{D}((U_{\ell(n)})^{m(n) \times m(n)}) = 1\right]\right| < \frac{1}{p(n)}$$

where X and Y are independently drawn from $(U_n)^{m(n)}$. (I.e. for random X and Y the matrix $\mathbf{C}_S(X,Y)$ cannot be efficiently distinguished from a random matrix.)

3.3 Expanding the Output Length

In Definition 3.1 no restriction was made on the output-length function, ℓ, of the pseudo-random synthesizer. However, our parallel construction of pseudo-random functions uses (parallel) pseudo-random synthesizers *with linear output length*, $\ell(n) = n$. The following lemma shows that any synthesizer, S, can be used to construct another synthesizer S', with large output-length, such that S and S' have the same parallel time complexity. Therefore, for the construction of pseudo-random functions in NC it is enough to show the existence of synthesizers with *constant output length* in NC.

Lemma 3.1 *Let S be a pseudo-random synthesizer with arbitrary output-length function, ℓ, in NC^i (resp. AC^i). Then for every constant $0 < \epsilon < 2$, there exists a pseudo-random synthesizer S' in NC^i (resp. AC^i) such that its output-length function, ℓ', satisfies $\ell'(n) = \Omega(n^{2-\epsilon})$.*

Proof. For every constant $c > 0$, define S^c as follows: Let $k_n \stackrel{\text{def}}{=} \max\{k \in \mathbb{Z} : k^{c+1} < n\}$. On input $x, y \in I^n$, regard the first k_n^{c+1} bits of x and y as two length-k_n^c sequences, X and Y, of k_n-bit strings. $S^c(x,y)$ is defined to be $\mathbf{C}_S(X,Y)$ (viewed as a single bit-string rather than a matrix). Notice that the following properties hold for S^c:

1. *S^c is indeed a pseudo-random synthesizer*: For any polynomial $m(\cdot)$, let X' and Y' be independently drawn from $(U_n)^{m(n)}$ and let X and Y be independently drawn from $(U_{k_n})^{m(n)\cdot k_n^c}$. By the definition of S^c, the distributions $\mathbf{C}_{S^c}(X',Y')$ and $\mathbf{C}_S(X,Y)$ are identical. Taking into account the fact that n is polynomial in k_n, we conclude that every polynomial-time distinguisher for S^c is also a polynomial-time distinguisher for S. Since S is a pseudo-random synthesizer so is S^c.

2. *Let ℓ_c denote the output-length function of S^c, then $\ell_c(n) = \Omega(n^{2-\frac{2}{c+1}})$*: Since c is a constant and $n < (k_n+1)^{c+1}$, for every n it holds that
$$\ell_c(n) = (k_n)^{2c} \cdot l(k_n) \geq (k_n)^{2c} = \Omega(n^{\frac{2c}{c+1}}) = \Omega(n^{2-\frac{2}{c+1}})$$

3. *S^c is in NC^i (resp. AC^i)*: Immediate from the definition of S^c.

Thus, by taking S' to be S^c for some $c > \frac{2}{\epsilon} - 1$ we obtain the lemma. □

The construction of Lemma 3.1 has the advantage that it is very simple and that the parallel time complexity of S and S' is identical. Nevertheless, it has an obvious disadvantage: The security of S' is related to the security of S on a much smaller input length. For example, if $\ell(n) = 1$ and $\ell'(n) = n$ then the security of S' on k^2-bit strings is related to the security S on k-bit strings. This results in a substantial increase in the time and space complexity of any construction that uses S'.

We now show an alternative construction to the one of Lemma 3.1 that is more security-preserving. The alternative construction uses a pseudo-random generator G that expands the input by a factor of 2 and relies on the GGM-Construction:

Corollary 3.2 *(of [26]) Let G be a pseudo-random generator in NC^i (resp. AC^i) such that $\forall s, |G(s)| = 2|s|$. Then for every polynomial $p(\cdot)$ there exists a pseudo-random generator G' in NC^{i+1} (resp. AC^{i+1}) such that $\forall s, |G'(s)| = p(|s|) \cdot |s|$.*

G' is defined as follows: On input s it computes $G(s) = s_0 \circ s_1$ and recursively generates $\frac{p(|s|)\cdot|s|}{2}$ bits from s_0 and $\frac{p(|s|)\cdot|s|}{2}$ bits from s_1. The number of levels required is $\lceil \log p(|s|) \rceil = O(\log |s|)$. Using Corollary 3.2 we get:

Lemma 3.3 *Let S be a pseudo-random synthesizer with arbitrary output-length function, ℓ, in NC^i (resp. AC^i). Let G be a pseudo-random generator in NC^j (resp. AC^j) such that $\forall s, |G(s)| = 2|s|$. Let k denote $\max\{i, j+1\}$. Then for every positive constant c, there exists a pseudo-random synthesizer S' in NC^k (resp. AC^k) such that its output-length function, ℓ', satisfies $\ell'(n) = \Omega(n^{2c} \cdot l(n))$.*

Furthermore, the construction of S' is linear-preserving in the sense of [32, 40] (the exact meaning of this claim is described below).

Proof.(sketch) S' is defined as follows: On input $x, y \in I^n$, compute $X = G'(x) = \{x'_1, \ldots, x'_{\lceil n^c \rceil}\}$ and $Y = G'(y) = \{y'_1, \ldots, y'_{\lceil n^c \rceil}\}$, where G' is the pseudo-random generator that is guaranteed to exist by Corollary 3.2. $S'(x,y)$ is defined to be $\mathbf{C}_S(X,Y)$.

It is immediate that S' is in NC^k (resp. AC^k) and that $\ell'(n) = \Omega(n^{2c} \cdot l(n))$. It is also not hard to verify that S' is indeed a pseudo-random synthesizer and (from the proof of Corollary 3.2) that the construction of S' is linear-preserving in the following sense:

Assume that there exists an algorithm that works in time $t(n)$ and distinguishes $\mathbf{C}_{S'}(X',Y')$ from $(U_{\ell'(n)})^{m'(n) \times m'(n)}$ with bias $\alpha(n)$, where X' and Y' are independently drawn from $(U_n)^{m'(n)}$. Let $m(n) = m'(n) \cdot \lceil n^c \rceil$. Then one of the following holds:

1. The same algorithm distinguishes $\mathbf{C}_S(X,Y)$ from $(U_{\ell(n)})^{m(n) \times m(n)}$ with bias $\alpha(n)/2$, where X and Y are independently drawn from $(U_n)^{m(n)}$.

2. There exists an algorithm that works in time $t(n) + m^2(n) \cdot poly(n)$ and distinguishes $G(U_n)$ from random with bias $\alpha(n)/O(m(n))$.

$\square$

The construction of Lemma 3.3 is indeed more security-preserving than the construction of Lemma 3.1 (since the security of S' relates to the security of S and G on the same input length). However, the time complexity of S' is still substantially larger than the time complexity of S, and the parallel time complexity of S' might also be larger. Given the drawbacks of both construction, it seems that a direct construction of efficient and parallel synthesizers with linear output length is very desirable.

3.4 Collection of Pseudo-Random Synthesizers

A natural way to relax the definition of a pseudo-random synthesizer is to allow a distribution of functions for every input length rather than a single function. To formalize this we use the concept of an efficiently computable function ensemble (of Definition 2.2).

Definition 3.2 (collection of pseudo-random synthesizers) *Let ℓ be any $\mathbb{N} \mapsto \mathbb{N}$ function and let $S = \{S_n\}_{n \in \mathbb{N}}$ be an efficiently computable $I^{2n} \mapsto I^{\ell}$ function ensemble. S is* a collection of $I^{2n} \mapsto I^{\ell}$ pseudo-random synthesizers *if for every probabilistic polynomial-time algorithm, $\mathcal{D}$, every two polynomials $p(\cdot)$ and $m(\cdot)$, and all sufficiently large n's*

$$\left| \Pr\left[\mathcal{D}(\mathbf{C}_{S_n}(X,Y)) = 1\right] - \Pr\left[\mathcal{D}((U_{\ell(n)})^{m(n) \times m(n)}) = 1\right] \right| < \frac{1}{p(n)}$$

where X and Y are independently drawn from $(U_n)^{m(n)}$.

As shown below, a collection of pseudo-random synthesizers is sufficient for our construction of pseudo-random functions. Working with a collection of synthesizers (rather than a single synthesizer) enables us to move some of the computation into a preprocessing stage during the key-generation. This is especially useful if all other computations can be done in parallel.

Note that Lemma 3.1 and Lemma 3.3 easily extend to collections of synthesizers.

4 A Parallel Construction of Pseudo-Random Functions

This section describes the construction of pseudo-random functions, using pseudo-random synthesizers as building blocks. The intuition of this construction is best explained through the concept of a k-dimensional pseudo-random synthesizer. This is a natural generalization of the "regular" (two-dimensional) synthesizer. Informally, an efficiently computable function of k variables, S^k, is a k-dimensional pseudo-random synthesizer if:

Given polynomially-many, uniformly-chosen, assignments for each variable, $\{\{a_{j,i}\}_{i=1}^m\}_{j=1}^k$, *the output of* S^k *on all the combinations* $M = \left(S^k(a_{1,i_1}, a_{2,i_2}, \ldots, a_{k,i_k})\right)_{i_1,i_2,\ldots,i_k=1}^m$ *cannot be efficiently distinguished from uniform by an algorithm that can access* M *at points of its choice*

Note that this definition is somewhat different from the two-dimensional case. For any constant k (and in particular for $k = 2$) the matrix M is of polynomial size and we can give it as an input to the distinguisher. In general, M might be too large and therefore we let the distinguisher "access M at points of its choice".

Using this concept, the construction of pseudo-random functions can be described in two steps:

1. A parallel construction of an n-dimensional synthesizer, S^n, from a two-dimensional synthesizer, S, that has output length $\ell(n) = n$. This is a recursive construction, where the $2k$-dimensional synthesizer, S^{2k}, is defined using a k-dimensional synthesizer, S^k:

$$S^{2k}(x_1, x_2, \ldots, x_{2k}) \stackrel{\text{def}}{=} S^k(S(x_1, x_2), S(x_3, x_4), \ldots, S(x_{2k-1}, x_{2k}))$$

2. An immediate construction of the pseudo-random function, f, from S^n:

$$f_{\langle a_{1,0}, a_{1,1}, a_{2,0}, a_{2,1}, \ldots a_{n,0}, a_{n,1}\rangle}(x) \stackrel{\text{def}}{=} S^n(a_{1,x_1}, a_{2,x_2}, \ldots, a_{n,x_n})$$

In fact, pseudo-random functions can be constructed from a *collection* of synthesizers. In this case, for each level of the recursion a different synthesizer is sampled from the collection. As will be noted below, for some collections of synthesizers (as those constructed in this paper) it is enough to sample a single synthesizer for all levels.

4.1 Formal Definition

The following operation on sequences is used in the construction:

Definition 4.1 *For every function* $S : I^{2n} \mapsto I^n$ *and every sequence,* $L = \{\ell_1, \ell_2, \ldots, \ell_k\}$, *of* n*-bit strings define* $\mathbf{SQ}_S(L)$ *to be the sequence* $L' = \{\ell'_1, \ldots, \ell'_{\lceil \frac{k}{2} \rceil}\}$, *where* $\ell'_i = S(l_{2i-1}, \ell_{2i})$ *for* $i \leq \lfloor \frac{k}{2} \rfloor$ *and if* k *is odd, then* $\ell'_{\lceil \frac{k}{2} \rceil} = \ell_k$ *(*$\mathbf{SQ}$ *stands for squeeze).*

We now turn to the construction itself:

Construction 4.1 (Pseudo-Random Functions) *Let* $S = \{S_n\}_{n \in \mathbb{N}}$ *be a collection of* $I^{2n} \mapsto I^n$ *pseudo-random synthesizers and let* $\mathcal{I}_S$ *be a probabilistic polynomial-time key-generating algorithm for* S *(as in Definition 2.2). For every possible value,* k, *of* $\mathcal{I}_S(1^n)$, *denote by* s_k *the corresponding* $I^{2n} \mapsto I^n$ *function. The function ensemble* $F = \{F_n\}_{n \in \mathbb{N}}$ *is defined as follows:*

- (key-generation) *On input* 1^n, *the probabilistic polynomial-time key-generating algorithm* $\mathcal{I}_F$ *outputs a pair* $(\vec{a}, \vec{k})$, *where* $\vec{a} = \{a_{1,0}, a_{1,1}, a_{2,0}, a_{2,1}, \ldots, a_{n,0}, a_{n,1}\}$ *is sampled from* $(U_n)^{2n}$ *and* $\vec{k} = \{k_1, k_2, \ldots, k_{\lceil \log n \rceil}\}$ *is generated by* $\lceil \log n \rceil$ *independent executions of* $\mathcal{I}_S$ *on input* 1^n *(i.e. is sampled from* $(\mathcal{I}_S(1^n))^{\lceil \log n \rceil}$*).*
- (evaluation) *For every possible value,* $(\vec{a}, \vec{k})$, *of* $\mathcal{I}_F(1^n)$ *the function* $f_{\vec{a},\vec{k}} : I^n \mapsto I^n$ *is defined as follows: On an* n*-bit input,* $x = x_1 x_2 \ldots x_n$, *the function outputs the single value in*

$$\mathbf{SQ}_{s_{k_1}}(\mathbf{SQ}_{s_{k_2}}(\ldots \mathbf{SQ}_{s_{k_{\lceil \log n \rceil}}}(\{a_{1,x_1}, a_{2,x_2}, \ldots, a_{n,x_n}\}) \ldots))$$

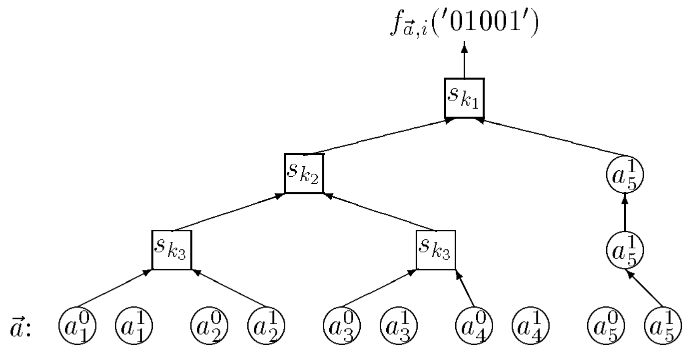

Figure 1: Computing the Value of the Pseudo-Random Function for $n = 5$

Finally, F_n is defined to be the random variable that assumes as values the functions $f_{\vec{a},\vec{k}}$ with the probability space induced by $\mathcal{I}_F(1^n)$.

The evaluation of $f_{\vec{a},\vec{k}}(x)$ can be thought of as a recursive labeling process of a binary tree with n leaves and depth $\lceil \log n \rceil$. The i^{th} leaf has two possible labels, $a_{i,0}$ and $a_{i,1}$. The i^{th} input bit, x_i selects one of these labels a_{i,x_i}. The label of each internal node at depth d is the value of $s_{k_{d+1}}$ on the labels of its children. The value of $f_{\vec{a},\vec{k}}(x)$ is simply the label of the root. (Figure 1 illustrates the evaluation of $f_{\vec{a},\vec{k}}$ for $n = 5$.) We note that this labeling process is very different than the one associated with the GGM-Construction [26]. First, the binary tree is of depth $\lceil \log n \rceil$ instead of depth n as in [26]. Secondly, the labeling process is bottom-up instead of top-down as in [26] (i.e. starting at leaves instead of the root). Moreover, here each input defines a different labeling of the tree whereas in [26] the labeling of the tree is fully determined by the key (and the input only determines a leaf such that its label is the value of the function on this input).

4.2 Efficiency of the Construction

It is clear that F is efficiently computable (given that S is efficiently computable). Furthermore, the parallel time complexity of functions in F_n is larger by a factor of $O(\log n)$ than the parallel time complexity of functions in S_n. The parallel time complexity of $\mathcal{I}_S$ and $\mathcal{I}_F$ is identical.

We note that, for simplicity, the parameter n serves a double role. n is both the length of inputs to $f_{\vec{a},\vec{k}} \in F_n$ and the security parameter for such a function (the second role is expressed by the fact that the strings in $\vec{a}$ are n-bit long). In practice, however, these roles would be separated. The security parameter would be determined by the quality of the synthesizers and the length of inputs to the pseudo-random functions would be determined by their application. In fact, one can usually use a pseudo-random function with a reasonably small input-length (say 160-bit long to prevent a "birthday attack"). This is implied by the suggestion of Levin [42] to pair-wise independently hash the input before applying the pseudo-random function (this idea is described with more details in the introduction).

4.3 Reducing the Key-Length

An apparent disadvantage of Construction 4.1 is the large key-length of a function $f_{\vec{a},\vec{k}} \in F_n$. In particular, the sequence $\vec{a}$ is defined by $2n^2$ bits. However, this is not truly a problem since: (a) In Section 6.1 a related construction is described (Construction 6.1) where $\vec{a}$ consists of a constant number of strings (and is therefore defined by $O(n)$ bits). (b) The truly random sequence $\vec{a}$ can be replaced by a pseudo-random sequence without increasing the depth of the construction (by more than a constant factor). This is achieved as follows: Let G be a pseudo-random generator that expands the input by a factor of 2. Let G' be the pseudo-random generator that can be constructed from G according to Corollary 3.2 for $p(n) = 2n$ (i.e. by using $\lceil \log n + 1 \rceil$ levels of the recursion). Then $\vec{a}$ can be replaced by $G'(\tilde{a})$, where $\tilde{a}$ is an n-bit seed.

In addition to $\vec{a}$, the key of $f_{\vec{a},\vec{k}} \in F_n$ consists of $\lceil \log n \rceil$ keys of functions in S_n. It turns out that for some collections of synthesizers (such as those described in this paper) this overhead can be eliminated as well. This is certainly true when using a *single* synthesizer instead of a collection. Moreover, from the proof of security for Construction 4.1 one can easily extract the following claim: If the collection of synthesizers remains secure even when it uses a public key (i.e. if $\mathbf{C}_{s_k}(X, Y)$ remains pseudo-random even when the distinguisher sees k), then the $\lceil \log n \rceil$ keys can be replaced with a single one (i.e. the same key can be used at all levels of the recursion).

5 Security of the Construction

Theorem 5.1 *Let S and F be as in Construction 4.1 and let $R = \{R_n\}_{n \in \mathbb{N}}$ be the uniform $I^n \mapsto I^n$ function ensemble. Then F is an efficiently computable pseudo-random function ensemble. Furthermore, any efficient distinguisher, $\mathcal{M}$, between F and R yields an efficient distinguisher, $\mathcal{D}$, for S such that the success probability of $\mathcal{D}$ is smaller by a factor of at most $\lceil \log n \rceil$ than the success probability of $\mathcal{M}$.*

To prove Theorem 5.1, we use of a hybrid argument (for details about this proof technique, see [25]): We first define a sequence of $\lceil \log n \rceil + 1$ function distributions such that the two extreme distributions are R_n and F_n. We then show that any distinguisher for two neighboring distributions can be transformed into a distinguisher for the pseudo-random synthesizers. For simplicity, we define those hybrid-distributions in case $n = 2^\ell$. The definition easily extends to a general value of n such that Claim 5.1 still holds.

For any $0 \leq j \leq \ell$, denote by H_n^j the j^{th} hybrid-distribution. The computation of functions in H_n^j may be described as a labeling process of a binary tree with n leaves and depth ℓ (an analogous description for F_n appears in Section 4). Here, the labeling process starts with nodes at depth $\ell - j$. The i^{th} such node has 2^{2^j} possible labels, $\{a_{i,s} : s \in I^{2^j}\}$ (which are part of the key). The i^{th} 2^j-bit substring of the input, x_i, selects one of these labels, a_{i,x_i}. The rest of the labeling process is the same as it was for functions in F_n: The label of each node at depth $d < \ell - j$ is the value of $s_{k_{d+1}}$ on the labels of its children. The value of the function on this input is simply the label of the root.

Another way to think of H_n^j is via the concept of a k-dimensional synthesizer (see Section 4). As was the case for F^n, the construction of functions in H_n^j can be described in two steps: (1) A recursive construction of a $2^{\ell-j}$-dimensional synthesizer, $S^{2^{\ell-j}}$, from a two-dimensional synthesizer, S. (2) An immediate construction of the pseudo-random function, f, from $S^{2^{\ell-j}}$:

$$f_{\{a_{r,s}:1 \leq r \leq 2^{\ell-j}, s \in I^{2^j}\}}(x_1 \circ x_2 \ldots \circ x_{2^{\ell-j}}) \stackrel{\text{def}}{=} S^{2^{\ell-j}}(a_{1,x_1}, a_{2,x_2}, \ldots, a_{2^{\ell-j},x_{2^{\ell-j}}})$$

We turn to the formal definition of the hybrid-distributions:

Definition 5.1 *Let $\mathcal{I}_S$ be the key-generating algorithms of S. Let n, ℓ and j be three integers such that $n = 2^\ell$ and $0 \leq j \leq \ell$. For every sequence, $\vec{k} = \{k_1, k_2, \ldots, k_{\ell-j}\}$ of possible values of $\mathcal{I}_S(1^n)$ and for every length-$2^{2^j}2^{\ell-j}$ sequence of n-bit strings, $\vec{a} = \{a_{r,s} : 1 \leq r \leq 2^{\ell-j}, s \in I^{2^j}\}$ the function*

$f_{\vec{a},\vec{k}} : I^n \mapsto I^n$ *is defined as follows: On input* $x = x_1 \circ x_2 \ldots \circ x_{2^{\ell-j}}$*, where* $\forall 1 \le i \le 2^{\ell-j}, x_i \in I^{2^j}$ *the function outputs the single value in*

$$\mathbf{SQ}_{s_{k_1}}(\mathbf{SQ}_{s_{k_2}}(\ldots \mathbf{SQ}_{s_{k_{\ell-j}}}(a_{1,x_1}, a_{2,x_2}, \ldots, a_{2^{\ell-j},x_{2^{\ell-j}}})\ldots))$$

H_n^j *is the random variable that assumes as values the functions* $f_{\vec{a},\vec{k}}$ *defined above, where the* k_i*'s are independently distributed according to* $\mathcal{I}_S(1^n)$ *and* $\vec{a}$ *is independently distributed according to* $(U_n)^{2^{2^j} 2^{\ell-j}}$.

This definition immediately implies that:

Claim 5.1 H_n^0 *and* F_n *are identically distributed and* $H_n^{\lceil \log n \rceil}$ *and* R_n *are identically distributed.*

The proof below shows that for every $0 \le j < \ell$ the two neighboring ensembles H_n^j and H_n^{j+1} are computationally indistinguishable. As shown below, this implies Theorem 5.1 by a standard hybrid argument.

Proof. (of Theorem 5.1) As mentioned in Section 4, it is obvious that F is an efficiently computable function ensemble. Assume that F is not pseudo-random. By the definition of pseudo-random function ensembles, there exists a polynomial-time oracle machine, $\mathcal{M}$, and a polynomial $p(\cdot)$ so that for infinitely many n's

$$\left| \Pr\left[\mathcal{M}^{F_n}(1^n) = 1\right] - \Pr\left[\mathcal{M}^{R_n}(1^n) = 1\right] \right| > \frac{1}{p(n)}$$

where $R = \{R_n\}_{n \in \mathbb{N}}$ is the uniform $I^n \mapsto I^n$ function ensemble. Let $t(\cdot)$ be a polynomial that bounds the number queries that $\mathcal{M}$ makes on input 1^n.

Given $\mathcal{M}$, we define the probabilistic polynomial-time algorithm $\mathcal{D}$ that distinguishes the output of S_n from random. Let $m = m(n)$ be defined by $m(n) \stackrel{\text{def}}{=} t(n) \cdot n$. For every n, the input of $\mathcal{D}$ is an $m(n) \times m(n)$ matrix, $B = (b_{i,j})$, whose entries are n-bit strings. As part of its algorithm, $\mathcal{D}$ invokes $\mathcal{M}$ on input 1^n. The definition of m allows $\mathcal{D}$ to answer all the queries of $\mathcal{M}$ (which are bounded by $t(n)$). It will be shown below that $\mathcal{D}$ distinguishes between the following two distributions of B: (a) $\mathbf{C}_{S_n}(X, Y)$ where X and Y are independently drawn from $(U_n)^{m(n)}$. (b) $(U_n)^{m(n) \times m(n)}$.

For simplicity of presentation, we only define the algorithm that $\mathcal{D}$ performs for $n = 2^\ell$. It is easy to extend this definition to a general value of n such that Claim 5.2 and Claim 5.3 still hold. On input $B = (b_{i,j})_{i,j=1}^{m(n)}$, the algorithm is defined as follows:

1. Choose $0 \le J < \ell$ uniformly at random.
2. Generate $\vec{k} = \{k_1, k_2, \ldots, k_{\ell-J-1}\}$ by $\ell - J - 1$ independent executions of $\mathcal{I}_S$ on input 1^n.
3. Extract $2^{\ell-J-1}$ sub-matrices of B: For $1 \le i \le 2^{\ell-J-1}$, denote by $B^i = \left(b^i_{u,v}\right)_{u,v=1}^{t(n)}$ the $t(n) \times t(n)$ diagonal sub-matrix of B defined by
$$b^i_{u,v} \stackrel{\text{def}}{=} b_{u+((i-1)\cdot t(n)+1), v+((i-1)\cdot t(n)+1)}$$
4. Invoke $\mathcal{M}$ on input 1^n. Denote by $q^r = q^r_1 \circ q^r_2 \ldots \circ q^r_{2^{\ell-J}}$ the r^{th} query $\mathcal{M}$ makes, where $q^r_i \in I^{2^J}$ for $1 \le i \le 2^{\ell-J}$. On each of these queries $\mathcal{D}$ answers as follows:

For every $1 \leq i \leq 2^{\ell-J-1}$, denote by $a_{i,q^r_{2i-1}\circ q^r_{2i}}$ the entry $b^i_{u,v}$ of B^i where

$$u = \min\{1 \leq j \leq r : q^j_{2i-1} = q^r_{2i-1}\} \ and \ v = \min\{1 \leq j \leq r : q^j_{2i} = q^r_{2i}\}$$

Answer the query with the single value in

$$\mathbf{SQ}_{s_{k_1}}(\ldots\mathbf{SQ}_{s_{k_{\ell-J-1}}}(\{a_{1,q^r_1\circ q^r_2}, \ldots, a_{2^{\ell-J-1},q^r_{2^{\ell-J}-1}\circ q^r_{2^{\ell-J}}}\})\ldots)$$

5. Output whatever $\mathcal{M}$ outputs.

It is obvious that $\mathcal{D}$ is a polynomial-time algorithm. To show that $\mathcal{D}$ is also a distinguisher for the pseudo-random synthesizers, we first state and prove the following two claims:

Claim 5.2 *For every* $0 \leq J < \ell$,

$$\Pr\left[\mathcal{D}((U_n)^{m(n)\times m(n)}) = 1 | J = j\right] = \Pr\left[\mathcal{M}^{H^{j+1}_n}(1^n) = 1\right]$$

Proof. As part of its algorithm, $\mathcal{D}$ denotes some of B's entries by names of the form "$a_{r,s}$", where $1 \leq r \leq 2^{\ell-J-1}$ and $s \in I^{2^{J+1}}$. Note that $\mathcal{D}$ never denotes an entry of B by two different names. Assume, for the sake of the proof, that any name $a_{r,s}$ that was not used by $\mathcal{D}$ is assigned an independently and uniformly distributed n-bit string. Denote by $\vec{a}$ the sequence $\{a_{r,s} : 1 \leq r \leq 2^{\ell-J-1}, s \in I^{2^{J+1}}\}$. It is easy to verify that:

1. When B is uniformly distributed, the distribution of $\vec{a}$ is identical to $(U_n)^{2^{2^{J+1}}2^{\ell-J-1}}$.
2. $\mathcal{D}$ answers every query, q, of $\mathcal{M}$ with the value $f_{\vec{a},\vec{k}}(q)$, where $f_{\vec{a},\vec{k}}$ is as in the definition of H^{J+1}_n.

The claim immediately follows from (1) and (2) and from the definition of $\vec{k}$. □

Claim 5.3 *Let* X *and* Y *be independently drawn from* $(U_n)^{m(n)}$. *Then for every* $0 \leq J < \ell$,

$$\Pr\left[\mathcal{D}(\mathbf{C}_{S_n}(X,Y)) = 1 | J = j\right] = \Pr\left[\mathcal{M}^{H^j_n}(1^n) = 1\right]$$

Proof. Let $X = \{x_1, x_2, \ldots, x_{m(n)}\}$ and $Y = \{y_1, y_2, \ldots, y_{m(n)}\}$ be independently drawn from $(U_n)^{m(n)}$ and let s_k be drawn from S_n. Assume that the input of $\mathcal{D}$ is $B = \mathbf{C}_{s_k}(X,Y)$. For the sake of the proof, define the vector $\vec{a'} = \{a'_{i,s} : 1 \leq i \leq 2^{\ell-J}, s \in I^{2^J}\}$ as follows:

- If $\mathcal{D}$ denoted by $a_{i,q^r_{2i-1}\circ q^r_{2i}}$ the entry $b^i_{u,v}$ of B^i, then define $a'_{2i-1,q^r_{2i-1}}$ to be $x_{(i-1)\cdot t(n)+u}$ and $a'_{2i,q^r_{2i}}$ to be $y_{(i-1)\cdot t(n)+v}$. Note that $a_{i,q^r_{2i-1}\circ q^r_{2i}} = s_k(a'_{2i-1,q^r_{2i-1}}, a'_{2i,q^r_{2i}})$.
- For all other values in $\vec{a'}$ assign an independently and uniformly distributed n-bit string.

It is easy to verify that the distribution of $\vec{a'}$ is identical to $(U_n)^{2^{2^J}2^{\ell-J}}$. Let $\vec{k'}$ be the sequence $\{k_1, k_2, \ldots, k_{\ell-J-1}, k\}$ and let $f_{\vec{a'},\vec{k'}}$ be as in the definition of H^J_n. We now have that the answer $\mathcal{D}$ gives to the r^{th} query, q^r, of $\mathcal{M}$ is:

$$\begin{aligned}
& \mathbf{SQ}_{s_{k_1}}(\ldots\mathbf{SQ}_{s_{k_{\ell-J-1}}}(\{a_{1,q^r_1\circ q^r_2}, \ldots, a_{2^{\ell-J-1},q^r_{2^{\ell-J}-1}\circ q^r_{2^{\ell-J}}}\})\ldots) \\
= \ & \mathbf{SQ}_{s_{k_1}}(\ldots\mathbf{SQ}_{s_{k_{\ell-J-1}}}(\mathbf{SQ}_{s_k}(\{a'_{1,q^r_1}, a'_{2,q^r_2}, \ldots, a'_{2^{\ell-J},q^r_{2^{\ell-J}}}\}))\ldots) \\
= \ & f_{\vec{a'},\vec{k'}}(q^r)
\end{aligned}$$

From this fact and from the definition of $\vec{a'}$ and $\vec{k'}$, we immediately get the claim. □

By Claims 5.1, 5.2 and 5.3, we can now conclude the following: Let X and Y be independently drawn from $(U_n)^{m(n)}$, then for infinitely many n's

$$\begin{aligned}
&\left|\Pr\left[\mathcal{D}(\mathbf{C}_{S_n}(X,Y))=1\right]-\Pr\left[\mathcal{D}((U_n)^{m(n)\times m(n)})=1\right]\right| \\
=\;& \frac{1}{\lceil\log n\rceil}\cdot\left|\sum_{j=0}^{\lceil\log n\rceil-1}\Pr\left[\mathcal{D}(\mathbf{C}_{S_n}(X,Y))=1|J=j\right]-\sum_{j=0}^{\lceil\log n\rceil-1}\Pr\left[\mathcal{D}((U_n)^{m(n)\times m(n)})=1|J=j\right]\right| \\
=\;& \frac{1}{\lceil\log n\rceil}\cdot\left|\sum_{j=0}^{\lceil\log n\rceil-1}\Pr\left[\mathcal{M}^{H_n^j}(1^n)=1\right]-\sum_{j=0}^{\lceil\log n\rceil-1}\Pr\left[\mathcal{M}^{H_n^{j+1}}(1^n)=1\right]\right| \\
=\;& \frac{1}{\lceil\log n\rceil}\cdot\left|\Pr\left[\mathcal{M}^{H_n^0}(1^n)=1\right]-\Pr\left[\mathcal{M}^{H_n^{\lceil\log n\rceil}}(1^n)=1\right]\right| \\
=\;& \frac{1}{\lceil\log n\rceil}\cdot\left|\Pr\left[\mathcal{M}^{F_n}(1^n)=1\right]-\Pr\left[\mathcal{M}^{R_n}(1^n)=1\right]\right| \\
>\;& \frac{1}{p(n)\cdot\lceil\log n\rceil}
\end{aligned}$$

This contradicts the assumption that S is a collection of pseudo-random synthesizers and completes the proof of Theorem 5.1. □

Corollary 5.2 *For any collection of pseudo-random synthesizers, S, such that its functions are computable in NC^i there exists an efficiently computable pseudo-random function ensemble, F, such that its functions are computable in NC^{i+1}. Furthermore, the corresponding key-generating algorithms, $\mathcal{I}_S$ and $\mathcal{I}_F$, have the same parallel time complexity.*

Proof. By Lemma 3.1, we can construct from, S, a new collection of $I^{2n} \mapsto I^n$ pseudo-random synthesizers, S', such that its functions are computable in NC^i. By Theorem 5.1, we can construct from S' an efficiently computable pseudo-random function ensemble, F, such that its functions are computable in NC^{i+1}. Both constructions preserve the parallel time complexity of the key-generating algorithms. □

6 A Related Construction and Additional Properties

Though designed to enable efficient computation in parallel, Construction 4.1 obtains some additional useful properties. In this section we describe two such properties: a rather sharp time-space tradeoff and an incremental property. We also show how to adjust the construction in order to improve upon these properties.

6.1 Time-Space Tradeoff

Construction 4.1 has the advantage of a sharp time-space tradeoff. In order to get an even sharper tradeoff, we describe an alternative construction of pseudo-random functions. The best way to understand the revised construction is by viewing the computation process backwards: Every function on n-bits is defined by the length-2^n sequence of all its values. Assume that we could sample and store two length-$\lceil\sqrt{2^n}\rceil$ sequences, X and Y, of random strings as the key of a pseudo-random function. In this case, given a pseudo-random synthesizer, S, we can define the 2^n values of the pseudo-random function to be the entries of the matrix $\mathbf{C}_S(X,Y)$. In order to reduce the key size, we can replace the random sequences, X and Y, with pseudo-random sequences. Such

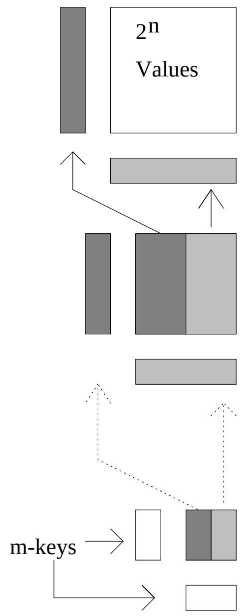

Figure 2: Illustration of the Alternative Construction

sequences X and Y can be obtained *together* from $\mathbf{C}_S(X', Y')$, where X' and Y' are two shorter random sequences (of length approximately $\sqrt{2 \cdot 2^{n/2}}$). By continuing this process of reducing the key size $\log n$ times, we get a key with constant number of strings (see Figure 2 for an illustration of the construction).

In order to understand where the original construction is "wasteful" in the size of the key we can describe it in similar terms: The 2^n values of the function are still the values of $\mathbf{C}_S(X, Y)$ for two sequences X and Y (in the description of the computation as a tree labeling process these are all the possible labels of the root's children) but then we get X and Y *separately* as $\mathbf{C}_S(X', Y')$ and $\mathbf{C}_S(X'', Y'')$. By the time the sequences have constant-length, there are $O(n)$ of those.

Returning to the new construction, note that if we allow a key of m strings we only need $t \approx \log n - \log\log m$ of the steps described above. Computing such functions requires t phases and in each phase several parallel invocations of S. The total number of invocations of S is $2^t - 1 \approx \frac{n}{\log m}$. This seems to be a relatively sharp time-space tradeoff and, to the best of our knowledge, one that cannot be obtained by the GGM-construction.

For some applications, like the protection of the data on a disk, we need pseudo-random functions with reasonably small amount of entries. In this case, by storing relatively few strings, we can achieve a very easy-to-compute function. For example, 512 random ℓ-bit strings define a pseudo-random $I^{30} \mapsto I^{\ell}$ function. Computing this function requires only 3 invocations of a pseudo-random synthesizer in 2 phases.

We formalize the definition of the alternative construction:

Construction 6.1 (Alternative Construction of Pseudo-Random Functions)
Let $S = \{S_n\}_{n\in\mathbb{N}}$ be a collection of $I^{2n} \mapsto I^n$ pseudo-random synthesizers and let $\mathcal{I}_S$ be a probabilistic polynomial-time key-generating algorithm for S (as in Definition 2.2). For every possible value, k, of $\mathcal{I}_S(1^n)$, denote by s_k the corresponding $I^{2n} \mapsto I^n$ function. The function ensemble $F = \{F_n\}_{n\in\mathbb{N}}$ is defined as follows:

- (key-generation) *Let m_j denote the value $2^j + 2$ and let t_n denote the smallest integer t such* *that $m_t \geq n$. On input 1^n, the probabilistic polynomial-time key-generating algorithm $\mathcal{I}_F$ outputs a pair $(\vec{a}, \vec{k})$, where $\vec{a} = \{a_0, a_1, \ldots, a_{2^{m_0}-1}\}$ is generated according to $(U_n)^{2^{m_0}}$ and $\vec{k} = \{k_1, k_2, \ldots, k_{t_n}\}$ is generated by t_n independent executions of $\mathcal{I}_S$ on input 1^n (i.e. is sampled from $(\mathcal{I}_S(1^n))^{t_n}$).*

- (evaluation) *For every possible value, $(\vec{a}, \vec{k})$, of $\mathcal{I}_F(1^n)$ and every j such that $0 \leq j \leq t_n$, define the function $f^j_{\vec{a},\vec{k}} : I^{m_j} \mapsto I^n$ in recursion on j: For $x \in I^{m_0}$, define $f^0_{\vec{a},\vec{k}}(x)$ to be a_x. For any $j > 0$ and $x = x_1 \circ x_2 \in I^{m_j}$ (x_1 and x_2 are $(2^{j-1}+1)$-bit strings) define $f^j_{\vec{a},\vec{k}}(x)$ to be $s_{k_j}(f^{j-1}_{\vec{a},\vec{k}}(0, x_1), f^{j-1}_{\vec{a},\vec{k}}(1, x_2))$.*

  For every $x \in I^n$ the value of the function $f_{\vec{a},\vec{k}} : I^n \mapsto I^n$ on x is defined to be $f^{t_n}_{\vec{a},\vec{k}}(x')$, where x' is obtained by padding x with $m_{t_n} - n$ zeros.

Finally, F_n is defined to be the random variable that assumes as values the functions $f_{\vec{a},\vec{k}}$ with the probability space induced by $\mathcal{I}_F(1^n)$.

The proof of security for Construction 6.1 is omitted since it is almost identical to the proof of security for Construction 4.1.

We can now state the exact form of the time-space tradeoff under the notation of Construction 6.1. If $\vec{a}$ contains 2^{m_i} strings instead of 2^{m_0}, then we can define $f^i_{\vec{a},\vec{k}}(x)$ to be a distinct value in $\vec{a}$ for every $x \in I^{m_i}$ and keep the recursive definition of $f^j_{\vec{a},\vec{k}}$ as before for $j > i$. In this case, computing $f_{\vec{a},\vec{k}}(x)$ can be done in $t_n - i$ phases with a total of $2^{t_n - i} - 1$ invocations of the synthesizers. The next lemma follows (for simplicity this lemma is stated in terms of a synthesizer instead of a collection of synthesizers):

Lemma 6.1 *Let S be a pseudo-random synthesizer with output-length function $\ell(n) = n$. Assume that S can be computed in parallel-time $D(n)$ and work $W(n)$ (on n-bit inputs). Then for every $m = m(n)$ such that $2^{m_0} \leq m(n) < 2^n$ there exists an efficiently computable pseudo-random function ensemble $F = \{F_n\}_{n \in \mathbb{N}}$ such that the key of a function in F_n is a sequence of at most $m(n)$ random n-bit strings and this function can be computed in parallel-time* $(\log n - \log\log m(n) + O(1))D(n)$ *and using work of $O(\frac{n}{\log m(n)})W(n)$.*

6.2 Incremental Property

We now describe an observation of Mihir Bellare that gives rise to an interesting incremental property of our construction. (For the formulation and treatment of incremental cryptography, see the work of Bellare, Goldreich and Goldwasser [7, 8].)

Let f be any function in F_n, where $F = \{F_n\}_{n \in \mathbb{N}}$ is the pseudo-random function ensemble defined in Construction 4.1. Let $x, y \in I^n$ be of Hamming distance one (x and y differ on exactly one bit). Then given the computation of $f(x)$ (including all intermediate values), we only need additional $\log n$ invocations of the pseudo-random synthesizers (instead of n) in order to evaluate $f(y)$. The easiest way to see the correctness of this observation is to recall the description of the computation of $f(x)$ as a labeling process on a depth-$\log n$ binary tree. The only labels that change as a result of flipping one bit of x are those of the nodes on a path from one leaf to the root (i.e. $\log n + 1$ labels).

If a Gray-code representation[3] of numbers is used, we get a similar observation for the computation of $f(x)$ and $f(x+1)$: Given the computation of one of these values, computing the other requires only additional $\log n$ invocations of the pseudo-random synthesizers. It is not hard to imagine situations where one of these incremental properties is useful.

The observation regarding the computation of $f(x)$ and $f(y)$, for x and y of Hamming distance one, also holds for the functions of Construction 6.1. The observation regarding the computation of $f(x)$ and $f(x+1)$ holds if we use a different representation of numbers (this representation is similar to a Gray-code, though a bit more complicated).

7 Construction of Pseudo-Random Synthesizers Based on General Cryptographic Primitives

Sections 7-9 are mostly devoted to showing parallel constructions of pseudo-random synthesizers. In this section we provide a simple construction of pseudo-random synthesizers based on what we call weak pseudo-random functions. This construction immediately implies a construction of pseudo-random synthesizers based on trapdoor one-way permutations (and an additional construction, based on any hard-to-learn problem, which is considered in Section 9). An interesting line for further research is the parallel construction of pseudo-random synthesizers from other cryptographic primitives. In particular, we do not know of such a construction from pseudo-random generators or directly from one-way functions.

7.1 Weak Pseudo-Random Functions

The reason pseudo-random functions are hard to construct is that they must endure a very powerful kind of an attack. The adversary (the distinguisher) may query their values at every point and may adapt its queries based on the answers it gets. We can weaken the opponent by letting the only access it has to the function be a polynomial sample of random points and the value of the function at these points. We call functions that look random to such an adversary *weak* pseudo-random functions. In this section it is shown that weak pseudo-random functions yield pseudo-random synthesizers in a straightforward manner. We therefore get a parallel construction of (standard) pseudo-random functions from weak pseudo-random functions.

For simplicity, we define weak pseudo-random functions as length-preserving. In their definition we use the following notation:

Notation 7.1 *For every function f and every sequence $X = \{x_1 \ldots x_k\}$ of values in the domain of f, denote by $\mathbf{V}(X, f)$ the sequence $\{x_1, f(x_1), x_2, f(x_2) \ldots x_k, f(x_k)\}$ ($\mathbf{V}$stands for values).*

Definition 7.1 (collection of weak pseudo-random functions) *An efficiently computable $I^n \mapsto I^n$ function ensemble $F = \{F_n\}_{n\in\mathbb{N}}$, is a* collection of weak pseudo-random functions *if for every probabilistic polynomial-time algorithm, $\mathcal{D}$, every two polynomials $p(\cdot)$ and $m(\cdot)$, and all sufficiently large n's*

$$\left|\Pr\left[\mathcal{D}(\mathbf{V}((U_n)^{m(n)}, F_n)) = 1\right] - \Pr\left[\mathcal{D}((U_n)^{2m(n)}) = 1\right]\right| < \frac{1}{p(n)}$$

[3] A permutation, P, on I^n is called a Gray-code representation if for every $0 \le x < 2^n$ the Hamming distance between $P(x)$ and $P(x+1 \bmod 2^n)$ is one. Such a P defines a Hamiltonian-cycle on the n-dimensional cube. It is not hard to see that an easy-to-compute P can be defined.

Let F be a collection of weak pseudo-random functions and let $\mathcal{I}$ be the polynomial-time key-generating algorithm for F. Lemma 7.1 shows how to construct a pseudo-random synthesizer from F and $\mathcal{I}$. Since the random bits of $\mathcal{I}$ can be replaced by pseudo-random bits, we can assume that $\mathcal{I}$ only uses n truly random bits on input 1^n. In fact, this is only a simplifying assumption which is not really required for the construction of pseudo-random synthesizers. For every $r \in I^n$, denote by $\mathcal{I}_r(1^n)$ the value of $\mathcal{I}(1^n)$ when $\mathcal{I}$ uses r as its random bits.

Lemma 7.1 *Let F and $\mathcal{I}$ be as above and define $S : \{0,1\}^* \times \{0,1\}^* \mapsto \{0,1\}^*$ such that $\forall x, y \in I^n, S(x,y) = f_{\mathcal{I}_y(1^n)}(x)$. Then S is a pseudo-random synthesizer.*

Proof. It is obvious that S is efficiently computable. Assume, in contradiction to the lemma, that S is not a pseudo-random synthesizer. Then there exists a probabilistic polynomial-time algorithm, $\mathcal{D}$, and polynomials $p(\cdot)$ and $m(\cdot)$, such that for infinitely many n's

$$\left|\Pr\left[\mathcal{D}(\mathbf{C}_S(X,Y)) = 1\right] - \Pr\left[\mathcal{D}((U_n)^{m(n)\times m(n)}) = 1\right]\right| > \frac{1}{p(n)}$$

where X and Y are independently drawn from $(U_n)^{m(n)}$.

For every n and every $0 \leq i \leq m(n)$, define the i^{th} hybrid distribution H_n^i over $m(n) \times m(n)$ matrices as follows: The first i columns are distributed according to $\mathbf{C}_S(X,Y)$, where X is drawn from $(U_n)^{m(n)}$ and Y is independently drawn from $(U_n)^i$. The last $m(n) - i$ columns are independently distributed according to $(U_n)^{m(n)\times(m(n)-i)}$. It is immediate that for infinitely many n's

$$\left|\Pr\left[\mathcal{D}(H_n^{m(n)}) = 1\right] - \Pr\left[\mathcal{D}(H_n^0) = 1\right]\right| > \frac{1}{p(n)}$$

We now define a distinguisher $\mathcal{D}'$ for F. Given $\{x_1, z_1, x_2, z_2, \ldots, x_{m(n)}, z_{m(n)}\}$ as its input, $\mathcal{D}'$ performs the following algorithm:

1. Define $X = \{x_1, \ldots, x_{m(n)}\}$ and $Z = \{z_1, \ldots, z_{m(n)}\}$.
2. Uniformly choose $0 < J \leq m(n)$.
3. Sample Y from $(U_n)^{J-1}$ and generate an $m(n)\times m(n)$ matrix B whose first $J-1$ columns are $\mathbf{C}_S(X,Y)$, its J^{th} column is Z^t and the last $m(n) - J$ columns are independently distributed according to $(U_n)^{m(n)\times(m(n)-J)}$.
4. Output $\mathcal{D}(B)$.

It is obvious that $\mathcal{D}'$ is efficiently computable. It is also easy to verify that

$$\Pr\left[\mathcal{D}'(\mathbf{V}((U_n)^{m(n)}, F_n)) = 1 | J = j\right] = \Pr\left[\mathcal{D}(H_n^j) = 1\right]$$

and that

$$\Pr\left[\mathcal{D}'((U_n)^{2m(n)}) = 1 | J = j\right] = \Pr\left[\mathcal{D}(H_n^{j-1}) = 1\right]$$

Thus, by a standard hybrid argument, we get that for infinitely many n's

$$\left|\Pr\left[\mathcal{D}'(\mathbf{V}((U_n)^{m(n)}, F_n)) = 1\right] - \Pr\left[\mathcal{D}'((U_n)^{2m(n)}) = 1\right]\right| > \frac{1}{p(n)m(n)}$$

in contradiction to the assumption that F is a collection of weak pseudo-random functions. We can therefore conclude the lemma. □

Notice that S (as in Lemma 7.1) obeys an even more powerful requirement than is needed by the definition of a pseudo-random synthesizer: For random X and Y the matrix $\mathbf{C}_S(X, Y)$ cannot be efficiently distinguished from a random matrix *even* if we allow the distinguisher access to X.

Corollary 7.2 *(of Lemma 7.1) If there exist weak pseudo-random functions that can be sampled and evaluated in NC, then there also exist a pseudo-random synthesizer in NC and (standard) pseudo-random functions that can be sampled and evaluated in NC.*

7.2 Trapdoor One-Way Permutations

We now describe a rather simple construction of weak pseudo-random functions from a collection of trapdoor permutations. Therefore, given Lemma 7.1, we get a construction of a pseudo-random synthesizer out of a collection of trapdoor permutations. This pseudo-random synthesizer is in NC if the trapdoor permutations can be sampled and inverted in NC (in fact, there is an additional requirement of a hard-core predicate in NC but this is already satisfied by [28]). Since we have no concrete example of this sort, we only give a brief and informal description of the construction (for formal definitions of trapdoor one-way permutations and hard-core bits, see e.g. [25, 40]).

Let $F = \{F_n\}_{n \in \mathbb{N}}$ be a permutation ensemble such that every $f_i \in F_n$ is a permutation over a domain D_n. Informally, F is a collection of trapdoor one-way permutations if the key generating algorithm $\mathcal{I}_F$ of F outputs both a public-key, i, and a trapdoor-key, $t(i)$, and we have that:

- Given i, the function f_i is easy to compute everywhere but hard to invert on the average.
- Given $t(i)$ the function f_i is easy to compute and to invert everywhere.

Let $F = \{F_n : D_n \mapsto D_n\}_{n \in \mathbb{N}}$ be a collection of trapdoor one-way permutations. Assume that the collection is one-way for the uniform distribution over the inputs (i.e. it is hard to compute x given $F_n(x)$, where x is uniformly distributed in D_n). Let the sequence of functions $\{b_n : D_n \mapsto I^1\}_{n \in \mathbb{N}}$ be a hard-core predicate for F. Informally, this means that given $F_n(x)$ for a uniformly distributed x, it is hard to guess $b_n(x)$ with probability which is non-negligibly better than half. We can now define a collection of weak pseudo-random functions $G = \{G_n\}_{n \in \mathbb{N}}$ in the following way:

> *For every $x \in I^n$, denote by $D_n(x)$ the element in D_n sampled using x as the random bits. For every key i of $f_i \in F_n$, define $g_i : I^n \mapsto I^1$ as follows:*
>
> $$\forall x \in I^n, g_i(x) \stackrel{\text{def}}{=} b_n(f_i^{-1}(D_n(x)))$$
>
> *(Note that computing $g_i(x)$ requires knowledge of the trapdoor-key $t(i)$.) Let G_n be the random variable that assumes as values the functions g_i with the probability space induced by the distribution over the keys in F_n.*

Claim 7.1 *(without proof) The function ensemble G which is defined above is a collection of weak pseudo-random functions.*

8 Number-Theoretic Constructions of Pseudo-Random Synthesizers

In this section we present several NC^1 constructions of pseudo-random synthesizers based on concrete, frequently-used, intractability assumptions. The first construction is at least as secure as the Diffie-Hellman [21] assumption. As we shall see in Section 8.3, we also get a construction that is at least as secure as factoring (since the Diffie-Hellman assumption modulo a composite is not stronger than factoring [44, 58] (also see [10])). Finally, we show two constructions that are at least as secure as the RSA assumption [55]. Although the RSA assumption is not weaker than factoring, the constructions based on RSA might have other advantages. For example, under the assumption that $\Omega(n)$ least-significant bits are simultaneously hard for RSA, we get pseudo-random synthesizers with linear output length. In addition, the constructions based on RSA and their proof of security use several interesting ideas that might be useful elsewhere. We first address issues that are common to all constructions.

The evaluation of our pseudo-random synthesizers in NC^1 relies on a preprocessing stage. This stage can be performed as part of the (sequential) key-generating algorithm. In this idea we follow the work of Kearns and Valiant, [37]. In their context, the additional data is "forced" into the input whereas in our context it is added to the key.

The analysis of the parallel-time complexity of the synthesizers uses previous results on the parallel-time complexity of arithmetic operations (see Karp and Ramachandran [36] for a review). In particular, we use the result of Beame, Cook and Hoover, [6]. They showed that iterated multiplication (multiplying n numbers of length n) and additional related operations can be performed by log-depth circuits (these circuits can be constructed efficiently, though sequentially). The results of [6] enable the computation of modular exponentiation in NC^1 given preprocessing that only depends on the base. This follows from the fact that computing $b^e \bmod N$ is reduced to an iterated multiplication (and an additional modular reduction) given the values $b^i \bmod N$ (where $0 \leq i \leq$ the length of e).

The pseudo-random synthesizers constructed in this section are Boolean functions[4]. Section 3.3 showed two methods for expanding the output-length of pseudo-random synthesizers. The method of Lemma 3.1 requires a pseudo-random generator that expands the input by a factor of two. A natural choice for this purpose (in the case of the synthesizers which are described in this section) is the pseudo-random generators of Blum, Blum and Shub [11] or the one of Hastad, Schrift and Shamir [30]. Given appropriate preprocessing, both generators can be computed in NC^1 and their security is based on the assumption that factoring integers (Blum-integers in [30]) is hard.

We note that all the constructions of this section give *collections* of pseudo-random synthesizers. However, the security of theses synthesizers does not rely on keeping their key private. As discussed in Section 4.3, this allows us to use a single synthesizer at all the levels of Construction 4.1 (and of Construction 6.1).

8.1 Common Tools

In our constructions, we use the result of Goldreich and Levin [28] which gives a hard-core predicate for "any" one-way function:

Theorem 8.1 *([28]) Let f be any one-way function. For every probabilistic polynomial-time algorithm, $\mathcal{A}$, for every polynomial, $p(\cdot)$ and all sufficiently large n's*

$$\Pr\left[\mathcal{A}(f(x), r) = r \odot x\right] < \frac{1}{2} + \frac{1}{p(n)}$$

where x and r are independently drawn from U_n (recall that $r \odot x$ denotes the inner product mod 2 *of r and x).*

[4] In fact, all these synthesizers can be made to output a logarithmic number of bits. Furthermore, given stronger assumptions they may output an even larger number of bits. See Remark 8.2 for an example

In fact, we use their result in a slightly different context. Loosely speaking, if given $f(x)$ it is hard to compute $g(x)$, then given $f(x)$ it is also hard to guess $g(x) \odot r$. An important improvement on the application of [28] in the context of the Diffie-Hellman assumption was made by Shoup [57].

In addition, the proof of security for all the constructions uses the next-bit prediction tests of Blum and Micali [14]. The equivalence between pseudo-random ensembles and ensembles that pass all polynomial-time next-bit tests was shown by Yao [61].

8.2 The Diffie-Hellman Assumption

We now define a collection of pseudo-random synthesizers that are at least as secure as the Diffie-Hellman assumption (**DH-Assumption**). This assumption was introduced in the seminal paper of Diffie and Hellman [21] (as a requirement for the security of their key-exchange protocol). The validity of the DH-Assumption was studied quite extensively over the last two decades. A few notable representatives of this research are [15, 43, 57]. Maurer [43] and Boneh and Lipton [15] have shown that in several settings the DH-Assumption is equivalent to the assumption that computing the discrete-log is hard. In particular, for any specific prime P there is an efficient reduction (given some information that only depends on P) of the discrete-log problem in $\mathbb{Z}_P^*$ to the DH-Problem in $\mathbb{Z}_P^*$. Shoup [57] has shown that the DH-Assumption holds against what he calls "generic"-algorithms.

For concreteness, we state the DH-Assumption in the group $\mathbb{Z}_P^*$, where P is a prime. However, our construction works just as well given the DH-Assumption in other groups. We use this fact in Section 8.3 to get pseudo-random synthesizers which are at least as secure as factoring. In order to formalize the DH-Assumption in $\mathbb{Z}_P^*$, we need to specify the distribution of P. One possible choice is to let P be a uniformly distributed prime of a given length. However, there are other possible choices. For example, it is not inconceivable that P can be fixed for any given length. To keep our results general, we let P be generated by *some* polynomial-time algorithm IG_{DH} (where IG stands for instance generator):

Definition 8.1 (IG_{DH}) *The Diffie-Hellman instance generator, IG_{DH}, is a probabilistic polynomial-time algorithm such that on input 1^n the output of IG_{DH} is distributed over n-bit primes.*

In addition, we need to specify the distribution of a generator, g, of $\mathbb{Z}_P^*$. It can be shown that if the DH-Assumption holds for some distribution of g, then it also holds if we let g be a uniformly distributed generator of $\mathbb{Z}_P^*$ (since there exists a simple randomized-reduction of the DH-Problem for any g to the DH-Problem with a uniformly distributed g).

All exponentiations in the rest of this subsection are in $\mathbb{Z}_P^*$ (the definition of P will be clear by the context). To simplify the notations, we omit the expression "mod P" from now on. We can now formally state the DH-Assumption (for the instance generator given IG_{DH}):

Assumption 8.1 *(Diffie-Hellman [21]) For every probabilistic polynomial-time algorithm, $\mathcal{A}$, for every polynomial, $q(\cdot)$ and all sufficiently large n's*

$$\Pr\left[\mathcal{A}(P, g, g^a, g^b) = g^{ab}\right] < \frac{1}{q(n)}$$

where the distribution of P is $IG_{DH}(1^n)$, the distribution of g is uniform over the set of generators of $\mathbb{Z}_P^*$ and the distribution of $\langle a, b\rangle$ is $(U_n)^2$.

Based on this assumption we define a collection of $I^{2n} \mapsto I^1$ pseudo-random synthesizers, S_{DH}:

Definition 8.2 *For every n-bit prime, P, every generator, g, of $\mathbb{Z}_P^*$ and every $r \in I^n$, define $s_{P,g,r} : I^{2n} \mapsto I^1$ by:*

$$\forall x, y \in I^n, s_{P,g,r}(x, y) \stackrel{\text{def}}{=} g^{xy} \odot r$$

Let S_n to be the random variable that assumes as values the functions $s_{P,g,r}$, where the distribution of P is $IG_{DH}(1^n)$, the distribution of g is uniform over the set of generators of $\mathbb{Z}_P^*$ and the distribution of r is U_n. The function ensemble S_{DH} is defined to be $\{S_n\}_{n\in\mathbb{N}}$.

Note that in a subsequent work [46] we show a direct and efficient construction of n-dimensional pseudo-random synthesizers based on the (stronger) *decisional* version of the DH-Assumption. This construction gives very efficient pseudo-random functions.

Theorem 8.2 *If the DH-Assumption (Assumption 8.1) holds, then S_{DH} is a collection of $I^{2n} \mapsto I^1$ pseudo-random synthesizers.*

Proof. It is obvious that $S_{DH} = \{S_n\}_{n\in\mathbb{N}}$ is efficiently computable. Assume that S_{DH} is not a collection of pseudo-random synthesizers. Then there exists a polynomial $m(\cdot)$ such that the ensemble $E = \{E_n\}$ is not pseudo-random, where $E_n = \mathbf{C}_{S_n}(X, Y)$ for X and Y that are independently drawn from $(U_n)^{m(n)}$. Therefore, there exists an efficient next-bit prediction test, $\mathcal{T}$, and a polynomial $q(\cdot)$ such that for infinitely many n's it holds that:

> *Given a prefix of E_n of uniformly chosen length, $\mathcal{T}$ succeeds to predict the next bit with probability greater than $\frac{1}{2} + \frac{1}{q(n)}$.*

We now show how to use $\mathcal{T}$ in order to define an efficient algorithm $\mathcal{A}$ such that for infinitely many n's

$$\Pr\left[\mathcal{A}(P, g, g^a, g^b, r) = g^{ab} \odot r\right] > \frac{1}{2} + \frac{1}{q(n)}$$

where the distribution of P, g, a and b is as in the DH-Assumption and r is drawn from U_n. By Theorem 8.1, this means that g^{ab} can also be efficiently computed which contradicts the DH-Assumption and completes the proof of the lemma.

In the definition of $\mathcal{A}$ we use the fact that in order to compute $g^{xy} = (g^x)^y = (g^y)^x$ it is enough to either know g^x and y or g^y and x (i.e. it is not required to know both x and y). This enables $\mathcal{A}$ to define a matrix which is distributed according to E_n such that one of its entries is $g^{ab} \odot r$ (the value $\mathcal{A}$ tries to guess) and all other entries can be computed by $\mathcal{A}$. It is now possible for $\mathcal{A}$ to guess $g^{ab} \odot r$ by invoking $\mathcal{T}$ on the appropriate prefix of this matrix.

In more details, on input $\langle P, g, g^a, g^b, r\rangle$ the algorithm $\mathcal{A}$ is defined as follows:

1. Uniformly choose $1 \le i, j \le m(n)$.
2. Define $X = \{x_1, \ldots, x_{m(n)}\}$ and $Y = \{y_1, \ldots, y_{m(n)}\}$ by setting $x_i = a$, $y_j = b$ and independently drawing all other values from U_n.
3. Define $B = (b_{u,v})_{u,v=1}^{m(n)}$ to be $\mathbf{C}_{s_{P,g,r}}(X, Y)$.

   Note that $\mathcal{A}$ knows all the values of X and Y *except* x_i and y_j. Therefore, $\mathcal{A}$ can compute all the entries of B *except* $b_{i,j} = g^{ab} \odot r$.
4. Invoke $\mathcal{T}$ and feed it with all the entries of B up to $b_{i,j}$ (i.e. the first $i-1$ rows and the first $j-1$ entries of the i^{th} row).
5. Output $\mathcal{T}$'s prediction of $b_{i,j}$.

It is obvious that $\mathcal{A}$ is efficient. Furthermore, since the distribution of B is exactly E_n, it is immediate that for infinitely many n's $\Pr\left[\mathcal{A}(P,g,g^a,g^b,r) = g^{ab} \odot r\right] > \frac{1}{2} + \frac{1}{q(n)}$, where the distribution of P, g, a, b and r is as above. This contradicts the DH-Assumption and proves the lemma. $\Box$

Corollary 8.3 *If the DH assumption (Assumption 8.1) holds, then there exist pseudo-random functions that are computable in NC^2 (given a sequential precomputation which is part of the key-generating algorithm).*

Proof. By Theorem 8.2, given that the DH-Assumption holds, S_{DH} is a collection of pseudo-random synthesizers. If the key-generating algorithm precomputes $g^{2^i} \bmod P$ for $1 \leq i \leq n$, then the functions of S_{DH} can be evaluated in NC^1. This precomputation reduces any modular exponentiation (with g as the base) to an iterated multiplication and an additional modular reduction (see also the discussion at the beginning of this section). By Corollary 5.2, there exist pseudo-random functions in NC^2 (the key-generating algorithm in both cases is sequential). $\Box$

Remark 8.1 *Assume that $IG_{DH}(1^n)$ has a single possible value, P, for every n. Then S_{DH} can be transformed into a synthesizer rather than a collection of synthesizers. In this case, the key-generating algorithm of the pseudo-random functions we get is in "non-uniform" NC.*

8.3 Composite Diffie-Hellman Assumption and Factoring

The collection of pseudo-random synthesizers, S_{DH}, is at least as secure as the DH-Assumption *modulo a prime*. As mentioned above, the DH-Assumption in any other group gives a corresponding construction of pseudo-random synthesizers with practically the same proof of security. We now consider the DH-Assumption *modulo a Blum-integer* (**composite DH-Assumption**). McCurley [44] and Shmuely [58] have shown that the composite DH-Assumption is implied by the assumption that factoring Blum-integers is hard (see also [10] for definitions and proof that are more consistent with our setting). We therefore get a simple construction of pseudo-random synthesizers which is at least as secure as factoring. In the subsequent, we give the relevant definitions and claims. We omit the proofs (since they are practically the same as in Section 8.2).

To formalize the composite DH-Assumption we let this composite be generated by *some* polynomial-time algorithm IG_F. We restrict the output of IG_F to the set of Blum-integers. This restriction is quite standard and it is meant to simplify the reduction of the composite DH-Assumption to factoring.

Definition 8.3 (IG_F) *The factoring instance generator, IG_F, is a probabilistic polynomial-time algorithm such that on input 1^n its output, N, is distributed over $2n$-bit integers, where $N = P \cdot Q$ for two n-bit primes, P and Q, such that $P \equiv Q \equiv 3 \bmod 4$ (such an integer is known as a Blum-integer).*

We note that the most natural distribution of $IG_F(1^n)$ is the uniform distribution over $2n$-bit Blum-integers. Furthermore, it is essential that $IG_F(1^n)$ would have many possible values since otherwise factoring would be non-uniformly easy (in this respect it is very different from the case of the Diffie-Hellman instance generator, IG_{DH}).

All exponentiations in the rest of this subsection are in $\mathbb{Z}_N^*$ (the definition of N will be clear by the context). To simplify the notations, we omit the expression "mod N" from now on. We can now define both the composite DH-Assumption and the assumption that factoring Blum-integers is hard (for the instance generator given IG_F):

Assumption 8.2 *(Composite Diffie-Hellman) For every probabilistic polynomial-time algorithm, $\mathcal{A}$, for every polynomial, $q(\cdot)$ and all sufficiently large n's*

$$\Pr\left[\mathcal{A}(N, g, g^a, g^b) = g^{ab}\right] < \frac{1}{q(n)}$$

where the distribution of N is $IG_F(1^n)$, the distribution of g is uniform over the set of quadratic-residues in $\mathbb{Z}_N^*$ and the distribution of $\langle a, b\rangle$ is $(U_{2n})^2$.

Assumption 8.3 *(Factoring) For every probabilistic polynomial-time algorithm, $\mathcal{A}$, for every polynomial, $q(\cdot)$ and all sufficiently large n's*

$$\Pr[\mathcal{A}(P \cdot Q) \in \{P, Q\}] < \frac{1}{q(n)}$$

where the distribution of $N = P \cdot Q$ is $IG_F(1^n)$.

We define a collection of $I^{4n} \mapsto I^1$ pseudo-random synthesizers, S_F (in analogy to the definition of S_{DH}):

Definition 8.4 *For every $2n$-bit Blum-integer, N, every quadratic-residue, g, in $\mathbb{Z}_N^*$ and every $r \in I^{2n}$, define $s_{N,g,r} : I^{2n} \mapsto I^1$ by:*

$$\forall x, y \in I^{2n}, s_{N,g,r}(x, y) \stackrel{\text{def}}{=} g^{xy} \odot r$$

Let S_n to be the random variable that assumes as values the functions $s_{N,g,r}$, where the distribution of N is $IG_F(1^n)$, the distribution of g is uniform over the set of quadratic-residues in $\mathbb{Z}_N^*$ and the distribution of r is U_{2n}. The function ensemble S_{DH} is defined to be $\{S_n\}_{n\in\mathbb{N}}$.

In the same way Theorem 8.2 was proven we get that:

Theorem 8.4 *If the composite DH-Assumption (Assumption 8.2) holds, then S_F is a collection of $I^{4n} \mapsto I^1$ pseudo-random synthesizers.*

Since the composite DH-Assumption (Assumption 8.2) is implied by the factoring assumption (Assumption 8.3) we get that:

Corollary 8.5 *(of Theorem 8.4 and of [10, 44, 58]) If factoring Blum-integers is hard (Assumption 8.3), then S_F is a collection of $I^{4n} \mapsto I^1$ pseudo-random synthesizers.*

Finally, we can conclude that:

Corollary 8.6 *If factoring Blum-integers is hard (Assumption 8.3), then there exist pseudo-random functions that are computable in NC^2 (given a sequential precomputation which is part of the key-generating algorithm).*

8.4 The RSA Assumption

We now define two collections of pseudo-random synthesizers under the assumption that the RSA-permutations of Rivest, Shamir and Adleman [55] are indeed one-way (i.e. under the assumption that it is hard to extract roots modulo a composite). This assumption is not weaker than the factoring assumption. However, the constructions based on RSA might have other advantages. For example, the second RSA-construction gives pseudo-random synthesizers with linear output length under the assumption that $\Omega(n)$ least-significant bits are simultaneously hard for RSA. Another reason to include these constructions is that they use several interesting techniques that might be useful elsewhere (e.g. the multiple role played by the subset product function).

As was the case with the previous assumptions, we keep the definition of the RSA-Assumption general by using *some* polynomial-time instance generator, IG_{RSA}:

Definition 8.5 **(IG_{RSA})** *The RSA instance generator, IG_{RSA}, is a probabilistic polynomial-time algorithm such that on input 1^n its output is distributed over pairs $\langle N, e\rangle$. Where $N = P \cdot Q$ is a $2n$-bit integer, P and Q are two n-bit primes and $e \in \mathbb{Z}^*_{\varphi(N)}$ (i.e. e is relatively prime to the order of $\mathbb{Z}^*_N$ which is denoted by $\varphi(N)$).*

All exponentiations in the rest of this subsection are in $\mathbb{Z}^*_N$ (the definition of N will be clear by the context). To simplify the notations, we omit the expression "mod N" from now on. We can now define the RSA-Assumption (for the instance generator given IG_{RSA}):

Assumption 8.4 *(RSA [55]) For every probabilistic polynomial-time algorithm, $\mathcal{A}$, for every polynomial, $q(\cdot)$ and all sufficiently large n's*

$$\Pr\left[\mathcal{A}(N, e, m^e) = m\right] < \frac{1}{q(n)}$$

where the distribution of $\langle N, e\rangle$ is $IG_{RSA}(1^n)$ and m is uniformly distributed in $\mathbb{Z}^*_N$.

The RSA-Assumption gives a collection of trapdoor one-way permutations: the public key is $\langle N, e\rangle$, the function $f_{N,e}$ is defined by $f_{N,e}(m) = m^e$ and the trapdoor-key is $\langle N, e, d\rangle$, where $d = e^{-1} \bmod \mathbb{Z}^*_{\varphi(N)}$ (which enables efficient inversion by the formula $m = (m^e)^d$). In Section 7 we showed a general construction of pseudo-random synthesizers out of trapdoor one-way permutations. However, a straightforward application of this construction to the RSA collection gives very inefficient synthesizers. In the following few paragraphs we describe this construction, the reasons it is inefficient and some of the ideas and tools that allow us to get more efficient synthesizers (which are also computable in NC^1).

Applying the construction of Section 7 to the RSA collection (using the Goldreich-Levin [28] hard-core predicate) gives the following collection of synthesizers: The key of each synthesizer is a uniformly distributed string r and for every $x, y \in I^n$, $s_r(x, y) \stackrel{\text{def}}{=} m^d \odot r$, where x samples the trapdoor-key $\langle N, e, d\rangle$ and y samples a uniformly chosen element $m \in \mathbb{Z}^*_N$. The most obvious drawback of this definition is that computing the value $s_r(x, y)$ consists of sampling an RSA trapdoor-key. In particular, computing $s_r(x, y)$ consists of sampling a Blum-integer, N. This (rather heavy) operation might be acceptable as part of the key-generating algorithm of the pseudo-random synthesizers (or functions) but is extremely undesirable as part of their evaluation.

In the direct constructions of pseudo-random synthesizers based on RSA, we manage to "push" the composite N into the key of the synthesizers (thus overcoming the drawback described in the previous paragraph). Nevertheless, we are still left with the following problem: Computing m^d in

NC^1 requires precomputation that depends on m. To enable this precomputation, it seems that m needs to be part of the key as well. However, in the construction which is described above, m depends on the input and is uniformly distributed for a random input. In order to overcome this problem, we show a method of sampling m almost uniformly at random in a way that facilitates the necessary preprocessing. This method uses the *subset product* functions. We first define these functions and then describe the way they are used in our context.

Definition 8.6 *Let G be a finite group and let $\vec{y} = \{y_1, \ldots, y_n\}$ be an n-tuple of elements in G. For any n-bit string, $x = x_1 \ldots x_n$, define the subset product $SP_{G,\vec{y}}(x)$ to be the product in G of the elements y_i such that $x_i = 1$.*

The following lemma was shown by Impagliazzo and Naor [34] and is based on the leftover hash lemma of [31, 35]:

Lemma 8.7 *([34]) Let G be a finite group, $n > c \log |G|$ and $c > 1$. Then for all but an exponentially small fraction of the choices of $\vec{y} \in (G)^n$, the distribution $SP_{G,\vec{y}}(U_n)$ is statistically indistinguishable within an exponentially small amount from the uniform distribution over G.*

Let N be a $2n$-bit integer, Lemma 8.7 gives a way of defining a collection of functions $\{f_k : I^{3n} \mapsto \mathbb{Z}_N^*\}_k$ which solves the problem of sampling an almost uniformly distributed element $m \in \mathbb{Z}_N^*$ for which m^d can be computed in NC^1. This collection is $\{f_{\vec{g}}\}_{\vec{g}} = \{SP_{\mathbb{Z}_N^*,\vec{g}}\}_{\vec{g}}$, where $\vec{g} = \{g_1, \ldots, g_{3n}\}$ is a sequence of $3n$ elements in $\mathbb{Z}_N^*$. The functions $\{f_{\vec{g}}\}_{\vec{g}}$ have the following properties:

1. For almost all choices of the key $\vec{g}$ we have that $f_{\vec{g}}(U_{3n})$ is almost uniformly distributed in $\mathbb{Z}_N^*$.

2. Following preprocessing that depends only on the key, $\vec{g}$, each value $(f_{\vec{g}}(x))^y$ can be computed in NC^1. The values that need to be precomputed are $g_i^{2^j}$, where $1 \leq i \leq 3n$ and $0 \leq j \leq$ the length of y. With these values the computation of $(f_{\vec{g}}(x))^y$ is reduced to a single iterated multiplication (and an additional modular reduction).

8.4.1 The First RSA Construction

For our first RSA construction we need to assume that it is hard to extract the e^{th} root modulo a composite *when e is a large prime.* To formalize this, we assume that for every possible value $\langle N, e\rangle$ of $IG_{RSA}(1^n)$ we have that e is a $2n$-bit prime (which in particular means that $e \in \mathbb{Z}_{\varphi(N)}^*$). Based on this version of the RSA-Assumption we define a collection of $I^{6n} \mapsto I^1$ pseudo-random synthesizers, S_{RSA1}:

Definition 8.7 *Let N be a $2n$-bit integer, let $\vec{g} = \{g_1, \ldots, g_{3n}\}$ be a sequence of $3n$ elements in $\mathbb{Z}_N^*$ and let r be a $2n$-bit string. Define the function $s_{N,\vec{g},r} : I^{6n} \mapsto I^1$ by:*

$$\forall x, y \in I^{3n}, s_{N,\vec{g},r}(x, y) \stackrel{\text{def}}{=} (g_x)^y \odot r$$

where $g_x = SP_{\mathbb{Z}_N^*,\vec{g}}(x)$. Let S_n to be the random variable that assumes as values the functions $s_{N,\vec{g},r}$, where the distribution of N is induced by $IG_{RSA}(1^n)$ and $\vec{g}$ and r are uniformly distributed in their range. The function ensemble S_{RSA1} is defined to be $\{S_n\}_{n\in\mathbb{N}}$.

Note that the only reason we let y be a $3n$-bit number (instead of a $2n$-bit number) is to make both inputs of $s_{N,\vec{g},r}$ be of the same length (which not really necessary for our constructions).

Theorem 8.8 *If the RSA-Assumption (Assumption 8.4) holds when for every possible value $\langle N, e\rangle$ of $IG_{RSA}(1^n)$ we have that e is a $2n$-bit prime. Then S_{RSA1} is a collection of $I^{6n} \mapsto I^1$ pseudo-random synthesizers.*

Proof. It is obvious that $S_{RSA1} = \{S_n\}_{n\in\mathbb{N}}$ is efficiently computable. Assume that S_{RSA1} is not a collection of pseudo-random synthesizers. Then there exists a polynomial $m(\cdot)$ such that the ensemble $E = \{E_n\}$ is not pseudo-random, where $E_n = \mathbf{C}_{S_n}(X, Y)$ for X and Y that are independently drawn from $(U_{3n})^{m(n)}$. Therefore, there exists an efficient next-bit prediction test, $\mathcal{T}$, and a polynomial $q(\cdot)$ such that for infinitely many n's it holds that:

> *Given a prefix of E_n of uniformly chosen length, $\mathcal{T}$ succeeds to predict the next bit with probability greater than $\frac{1}{2} + \frac{1}{q(n)}$.*

We now show how to use $\mathcal{T}$ in order to define an efficient algorithm $\mathcal{A}$ such that for infinitely many n's

$$\Pr\left[\mathcal{A}(N, e, m^e, z, r) = m^z \odot r\right] > \frac{1}{2} + \frac{1}{2q(n)}$$

where the distribution of N, e and m is as in the RSA-Assumption (with the restriction that e is a $2n$-bit prime), r is drawn from U_{2n} and z is uniformly distributed over the set of $3n$-bit integers that are relatively prime to e. By Theorem 8.1, this means that m^z can also be efficiently computed. Following Shamir [56], we note that given any z such that $\gcd(e, z) = 1$ and given m^z it is easy to compute m. The reason is that if $gcd(e, z) = 1$ then m can be computed by the formula $m = (m^e)^a (m^z)^b$ where $a, b \in \mathbb{Z}$ satisfy that $ae + bz = 1$ (and can be efficiently computed as well). Thus, the existence of such an algorithm $\mathcal{A}$ contradicts the RSA-Assumption and completes the proof of the lemma.

The algorithm $\mathcal{A}$ defines a matrix B which is *almost* identically distributed as E_n. One of the entries of B is $m^z \odot r$ (the value $\mathcal{A}$ tries to guess) and all other entries can be computed by $\mathcal{A}$. It is now possible for $\mathcal{A}$ to guess $m^z \odot r$ by invoking $\mathcal{T}$ on the appropriate prefix of this matrix. In more details, on input $\langle N, e, m^e, z, r\rangle$ the algorithm $\mathcal{A}$ is defined as follows:

1. Uniformly choose $1 \leq i, j \leq m(n)$.

2. Define the values $\{h_1, \ldots, h_{m(n)}\}$ and $\{d_1, \ldots, d_{m(n)}\}$ by setting $h_i = m$, uniformly drawing all other h_u's from $\mathbb{Z}_N^*$, setting $d_j = z \cdot e^{-1} \bmod \varphi(N)$ and drawing all other d_v's from U_{3n}.

3. Define $B = (b_{u,v})_{u,v=1}^{m(n)}$ by setting $b_{u,v} = \left(((h_u)^e)^{d_v}\right) \odot r$.

   Note that $\mathcal{A}$ can compute any entry $b_{u,v}$ *except* $b_{i,j} = m^z \odot r$. The reason is that if $v \neq j$ then $\mathcal{A}$ knows both d_v and $(h_u)^e$ and if $u \neq i$ then $\mathcal{A}$ can compute $b_{u,j} = \left(((h_u)^e)^{z \cdot e^{-1}}\right) \odot r = (h_u)^z \odot r$ since it knows both h_u and z.

4. Invoke $\mathcal{T}$ and feed it with all the entries of B up to $b_{i,j}$ (i.e. the first $i-1$ rows and the first $j-1$ entries of the i^{th} row).

5. Output $\mathcal{T}$'s prediction of $b_{i,j}$.

It is obvious that $\mathcal{A}$ is efficient. In order to complete the proof, we need to show that if $\langle N, e, m^e, z, r\rangle$ are distributed as above, then B and E_n are of exponentially small statistical distance. This would imply that (for infinitely many n's) if we feed $\mathcal{T}$ with the bits of B up to $b_{i,j}$ it predicts $b_{i,j} = m^z \odot r$ with probability greater than, say, $\frac{1}{2} + \frac{1}{2q(n)}$. As argued above, this would contradict the RSA-Assumption and would complete the proof.

To see that B and E_n are indeed statistically close notice that:

- Since $e \in \mathbb{Z}^*_{\varphi(N)}$ and $\forall 1 \le u \le m(n)$ the value m_u is uniformly distributed in $\mathbb{Z}^*_N$, we have that $\forall 1 \le u \le m(n)$ the value $(m_u)^e$ is also uniformly distributed in $\mathbb{Z}^*_N$.

  By Lemma 8.7, we therefore have that the distribution of $\{(m_u)^e\}_{1\le u\le m(n)}$ is statistically close to the distribution of $\{g_{x_u} = SP_{\mathbb{Z}^*_N,\vec{g}}(x_u)\}_{1\le u\le m(n)}$ for uniformly distributed values $\{x_1, \ldots, x_{m(n)}\} \in (I^{3n})^{m(n)}$ and $\vec{g} = \{g_1, \ldots, g_{3n}\} \in (\mathbb{Z}^*_N)^{3n}$.

- For z that is chosen from U_{3n} the distribution of $z \cdot e^{-1} \bmod \varphi(N)$ and $U_{3n} \bmod \varphi(N)$ are statistically close. Since e is a large prime, even given the restriction that z is relatively prime to e these distributions are statistically close.

Given these two observations it is easy to verify that B and E_n are indeed of exponentially small statistical distance. $\square$

Claim 8.1 *The functions in S_{RSA1} can be evaluated in NC^1 (given a sequential precomputation which is part of the key-generating algorithm).*

Proof. Given that the key-generating algorithm precomputes $(g_i)^{2^j}$ for $1 \le i, j \le 3n$, the evaluation of functions in S_{RSA1} is reduced to an iterated multiplication and an additional modular reduction. $\square$

8.4.2 The Second RSA Construction

The security of S_{RSA1} depends on the assumption that it is hard to extract the e^{th} root modulo a composite, where e is a large prime. Here, we define another collection of synthesizers under the assumption that it is hard to extract the e^{th} root modulo a composite, N, without any restriction on the distribution of $e \in \mathbb{Z}^*_{\varphi(N)}$. However, we introduce a new restriction on the possible values of the composite N:

Definition 8.8 *Let G_n be the set of $2n$-bit integers $N = P \cdot Q$ such that P and Q are two n-bit primes and $\varphi(N)$ has no odd factor smaller than n^2.*

It is easy to verify that if $N \in G_n$ then a sequence of $3n$ uniformly-chosen odd-values, $\vec{d} = \{d_1, \ldots, d_{3n}\} \in \mathbb{Z}_N$, have a constant probability to be in $\left(\mathbb{Z}^*_{\varphi(N)}\right)^{3n}$. By Lemma 8.7, given such a sequence, it is easy to almost uniformly sample any polynomial number of values in $\mathbb{Z}^*_{\varphi(N)}$ *even without knowledge of $\varphi(N)$*. This can be done by using the subset product function[5] $SP_{\mathbb{Z}^*_{\varphi(N)},\vec{d}}$. Notice that here the subset product function serves an additional role to the one already described above.

Sieve theory shows that G_n is not too sparse. For example, denote by $B(x)$ the number of primes p smaller than x such that $(p-1)/2$ is the product of two primes each of which is larger than $p^{1/4}$. Then there exists a positive constant c such that $B(x) \ge \frac{cx}{\log^2 x}$. See [52] for several results of this sort (which are more than sufficient for our purpose). As a result we get that: (a) If the RSA-assumption holds for a uniformly distributed value of N, then it also holds under the restriction $N \in G_n$. (b) The uniform distribution over G_n can be efficiently sampled (using Bach's algorithm [5]). Given (a) and (b) it seems that this restriction is rather reasonable.

[5]Actually, without knowledge of $\varphi(N)$, we cannot really compute $SP_{\mathbb{Z}^*_{\varphi(N)},\vec{d}}$. However, for every input x, we can still compute y such that $SP_{\mathbb{Z}^*_{\varphi(N)},\vec{d}}(x) = y \bmod \varphi(N)$. Such a value y would be just as good as $SP_{\mathbb{Z}^*_{\varphi(N)},\vec{d}}(x)$ for our proof.

Based on the RSA-Assumption with the restriction that $N \in G_n$, we define a collection of $I^{6n} \mapsto I^1$ pseudo-random synthesizers, S_{RSA2}. In the definition of S_{RSA2}, we use the least-significant bit (LSB) instead of the Goldreich-Levin hard-core bit. Alexi et. al. [2] showed that LSB is a hard-core bit for RSA. Fischlin and Schnorr [22] have recently provided a stronger reduction for this bit.

Definition 8.9 *Let N be a $2n$-bit integer, let $\vec{g} = \{g_1, \ldots, g_{3n}\}$ be a sequence of $3n$ elements in $\mathbb{Z}_N^*$ and let $\vec{d} = \{d_1, \ldots, d_{3n}\}$ be a sequence of $3n$ elements in $\mathbb{Z}_{\varphi(N)}^*$. Define the function $s_{N,\vec{g},\vec{d}} \colon I^{6n} \mapsto I^1$ by:*

$$\forall x, y \in I^{3n}, s_{N,\vec{g},\vec{d}}(x, y) \stackrel{\text{def}}{=} LSB((g_x)^{d_y})$$

where $g_x = SP_{\mathbb{Z}_N^*,\vec{g}}(x)$. and $d_y = SP_{\mathbb{Z}_{\varphi(N)}^*,\vec{d}}(y)$. Let S_n to be the random variable that assumes as values the functions $s_{N,\vec{g},\vec{d}}$, where the distribution of N is induced by $IG_{RSA}(1^n)$ and $\vec{g}$ and $\vec{d}$ are uniformly distributed in their range. The function ensemble S_{RSA2} is defined to be $\{S_n\}_{n\in\mathbb{N}}$.

Theorem 8.9 *If the RSA-Assumption (Assumption 8.4) holds when for every possible value $\langle N, e\rangle$ of $IG_{RSA}(1^n)$ we have that $N \in G_n$. Then S_{RSA2} is a collection of $I^{6n} \mapsto I^1$ pseudo-random synthesizers.*

Proof. It is obvious that $S_{RSA2} = \{S_n\}_{n\in\mathbb{N}}$ is efficiently computable. Assume that S_{RSA2} is not a collection of pseudo-random synthesizers. Then there exists a polynomial $m(\cdot)$ such that the ensemble $E = \{E_n\}$ is not pseudo-random, where $E_n = \mathbf{C}_{S_n}(X, Y)$ for X and Y that are independently drawn from $(U_{3n})^{m(n)}$. Therefore, there exists an efficient next-bit prediction test, $\mathcal{T}$, and a polynomial $q(\cdot)$ such that for infinitely many n's it holds that:

> *Given a prefix of E_n of uniformly chosen length, $\mathcal{T}$ succeeds to predict the next bit with probability greater than $\frac{1}{2} + \frac{1}{q(n)}$.*

We now show how to use $\mathcal{T}$ in order to define an efficient algorithm $\mathcal{A}$ such that for infinitely many n's

$$\Pr\left[\mathcal{A}(N, e, m^e) = LSB(m)\right] > \frac{1}{2} + \frac{1}{2q(n)}$$

where the distribution of N, e and m is as in the RSA-Assumption (with the restriction that $N \in G_n$). By [2], this contradicts the RSA-Assumption and completes the proof of the lemma.

The basic idea in the definition of the algorithm $\mathcal{A}$ is similar to the proof of Theorem 8.8: the algorithm $\mathcal{A}$ defines a matrix B which is *almost* identically distributed as E_n. One of the entries of B is $LSB(m)$ (the value $\mathcal{A}$ tries to guess) and all other entries can be computed by $\mathcal{A}$. It is now possible for $\mathcal{A}$ to guess $LSB(m)$ by invoking $\mathcal{T}$ on the appropriate prefix of this matrix. In more details, on input $\langle N, e, m^e\rangle$ as above, the algorithm $\mathcal{A}$ is defined as follows:

1. Uniformly choose $1 \le i, j \le m(n)$.
2. Define $\hat{e}$ to be $e \cdot d$, where d is almost uniformly distributed in $\mathbb{Z}_{\varphi(N)}^*$ (such a value d can be sampled because $N \in G_n$).

   Note that $\hat{e}$ is almost uniformly distributed in $\mathbb{Z}_{\varphi(N)}^*$
3. Define the values $\{h_1, \ldots, h_{m(n)}\}$ and $\{d_1, \ldots, d_{m(n)}\}$ by setting $h_i = m$, uniformly drawing all other h_u's from $\mathbb{Z}_N^*$, setting $d_j = \hat{e}^{-1} \bmod \varphi(N)$ and sampling all other d_v's almost uniformly from $\mathbb{Z}_{\varphi(N)}^*$.

4. Define $B = (b_{u,v})_{u,v=1}^{m(n)}$ by setting $b_{u,v} = LSB\left(\left((h_u)^{\hat{e}}\right)^{d_v}\right)$.

   Note that $\mathcal{A}$ can compute any entry $b_{u,v}$ *except* $b_{i,j} = LSB(m)$. The reason is that if $v \neq j$ then $\mathcal{A}$ knows both d_v and $(h_u)^{\hat{e}}$ and if $u \neq i$ then $\mathcal{A}$ can compute $b_{u,j} = LSB\left(\left((h_u)^{\hat{e}}\right)^{\hat{e}^{-1}}\right) = LSB((h_u))$ since it knows h_u.

5. Invoke $\mathcal{T}$ and feed it with all the entries of B up to $b_{i,j}$ (i.e. the first $i-1$ rows and the first $j-1$ entries of the i^{th} row).

6. Output $\mathcal{T}$'s prediction of $b_{i,j}$.

It is obvious that $\mathcal{A}$ is efficient. It is also easy to verify that if $\langle N, e, m^e \rangle$ is distributed as above, then B and E_n are of exponentially small statistical distance. Therefore, for infinitely many n's if we feed $\mathcal{T}$ with the bits of B up to $b_{i,j}$ it predicts $b_{i,j} = LSB(m)$ with probability greater than, say, $\frac{1}{2} + \frac{1}{2q(n)}$. As argued above, this contradicts the RSA-Assumption and completes the proof of the lemma. $\square$

Claim 8.2 *The functions in S_{RSA2} can be evaluated in NC^1 (given a sequential precomputation which is part of the key-generating algorithm).*

Proof. Given that the key-generating algorithm precomputes $(g_i)^{2^j}$ for $1 \leq i, j \leq 3n$, the evaluation of functions in S_{RSA2} is reduced to two iterated multiplication and two modular reductions. $\square$

Remark 8.2 *Since Alexi et. al. [2] showed that the $\log n$ least-significant bits are simultaneously hard for RSA we can adjust the functions in S_{RSA2} to output $\log n$ bits. If we make a stronger assumption, that $\Omega(n)$ bits are simultaneously hard for RSA, we get a direct construction of pseudo-random synthesizers with linear output size. Although the stronger assumption is not known to be equivalent to the RSA-Assumption it is still quite standard.*

9 Pseudo-Randomness and Learning-Theory

9.1 Synthesizers Based on Hard-to-Learn Problems

The "traditional" connection between cryptography and learning theory is using cryptographic assumptions to deduce computational non-learnability results. Blum, Furst, Kearns and Lipton [13] have suggested that the other direction is interesting as well. They have shown how to construct several cryptographic primitives out of *hard-to-learn* functions, in a way that preserves the degree of parallelism of the functions. A major motivation for presenting such constructions is the simplicity of function classes that are believed to be hard for efficient learning.

We show that, under the definitions of [13], pseudo-random synthesizers can easily be constructed from distributions of functions that are hard to learn. Thus (by the constructions shown in this paper), two additional cryptographic primitives can be constructed in parallel out of hard-to-learn functions: (1) pseudo-random generators with large expansion ratio (without assuming, as in [13], that the functions are hard to learn with membership queries) and (2) pseudo-random functions.

There is a difference between standard learning-theory definitions and standard cryptographic definitions. Loosely speaking, a collection of concepts is hard to learn if for every efficient algorithm there exists a distribution over the concepts that is hard to learn *for this specific algorithm.* In cryptographic settings the order of quantifiers is reversed: the hard distribution should be hard for *every* efficient algorithm. In order for hard-to-learn problems to be useful in cryptographic settings an average-case learning model is introduced in [13].

Informally describing one of the definitions in [13], we can say that a distribution ensemble of functions, $F = \{F_n\}_{\in \mathbb{N}}$, is not *weakly predictable on the average* with respect to a distribution D on the inputs, if the following holds: There is no efficient algorithm that can predict $f(\tilde{x})$ with probability $\frac{1}{2} + \frac{1}{poly(n)}$, given $\tilde{x}$ and a polynomial sequence $\{\langle x_i, f(x_i)\rangle\}$, where f is distributed according to F_n and all the inputs are independently distributed according to D.

It is easy to verify that a distribution ensemble of functions, F, is not weakly predictable on the average with respect to the uniform distribution if and only if it is a collection of weak pseudo-random functions. Thus, by Lemma 7.1, such a distribution defines a pseudo-random synthesizer S, where $S(x, y)$ is simply $f_{\mathcal{I}_y(1^n)}(x)$ (recall that $f_{\mathcal{I}_y(1^n)}$ denotes the function that is sampled from F_n using y as random bits). Using S we can construct pseudo-random generators and pseudo-random functions. Moreover, by Lemma 3.1. the pseudo-random generator we construct may have a large expansion ratio ($n^{1-\epsilon}$ for every $\epsilon > 0$). The pseudo-random generator constructed in [13] under the same assumption has expansion ratio bounded by $1 + 1/n$.

9.1.1 A Concrete Hard-to-Learn Problem

Consider the following distribution on functions with parameters k and n. Each function is defined by two, uniformly distributed, disjoint sets $A, B \subset \{1, \ldots, n\}$ each of size k. Given an n-bit input, the output of the function is the exclusive-or of two values: the parity of the bits indexed by A and the majority of the bits indexed by B. In [13], it is estimated that these functions (for $k = \log n$) cannot be weakly predictable without using "profoundly" new ideas. If indeed this distribution of functions is not weakly predictable on the average (for any k), then it defines an extremely efficient synthesizer. Therefore, using the constructions of this paper, we get efficient parallel pseudo-random functions.

9.2 The Application of Pseudo-Random Functions to Learning Theory

As observed by Valiant [59], if a concept class contains pseudo-random functions, then we can deduce a very strong unlearnability result for this class. Informally, it means that there exists a distribution of concepts in this class that is hard for *every* learning algorithm, for *every* "non-trivial" distribution on inputs *even* when membership queries are allowed. Since no parallel pseudo-random functions were known before the current work, this observation could not have been applied to NC.

Nevertheless, other techniques based on cryptographic assumptions were used in [3, 37, 38] to show hardness results for NC^1 and additional classes. For example, Kharitonov [38] used the following fact: after preprocessing, a polynomial-length pseudo-random bit-string (based on [11]) can be produced in NC^1 (the length of the string can stay undetermined at the preprocessing stage). The existence of pseudo-random functions in NC might still be of interest to computational learning theory because the result it implies is stronger than previous results. To briefly state the difference, we note that the results of [3, 37] use a very specific distribution on the inputs that is hard-to-learn and the results of [38] strongly rely on the order of quantifiers in learning-theory models which was mentioned above (e.g. for any given learning algorithm [38] shows a different hard concept which can still be easily learned by an algorithm which has a somewhat larger running-time).

10 Further Research

In Sections 7-9 we discussed the existence of pseudo-random synthesizers in NC. Additional work should be done in this area. The most obvious question is what are the general assumptions (in cryptography or in other fields) that imply the existence of pseudo-random synthesizers in NC. In particular, whether there exist parallel constructions of pseudo-random synthesizers out of pseudo-random generators or directly from one-way functions.

It is also of interest to find parallel constructions of pseudo-random synthesizers based on other concrete intractability assumptions. A task of practical importance is to derive more efficient concrete constructions of pseudo-random synthesizers in order to get efficient constructions of pseudo-random functions. As described in Section 3.3, an important contribution to the efficiency of the pseudo-random functions would be a direct construction of synthesizers with linear output length.

An extensive research field deals with pseudo-random generators that "fool" algorithms performing space-bounded computations. This kind of generators can be constructed without any (unproven) assumptions; see [4, 47, 48, 50] for definitions, constructions and applications. It is possible that the concept of pseudo-random synthesizers and the idea of our construction can be applied to the "world" of space-bounded computations. As a motivation remark, note that the construction in [47] bares some resemblance to the GGM construction.

In some sense we can think of the inner product function as a pseudo-random synthesizer for space bounded computation. Let $IP(x, y)$ be the inner product of x and y (mod 2) and let X and Y be random length-m sequences of n-bit strings. For some constant $0 < \alpha < 1$ and $s = \alpha n$ it can be shown that $\mathbf{C}_{IP}(X, Y)$ is a pseudo-random generator for $SPACE(s)$ with parameter $\epsilon = 2^{-\Omega(s)}m^2$ (when $\mathbf{C}_{IP}(X, Y)$ is given row by row). The only fact we use is that approximating IP is "hard" in the communication complexity model (see [20, 60]).

One might also try to apply the concept of pseudo-random synthesizers for other classes of algorithms. For example [1, 49] construct pseudo-random generators for polynomial-size constant-depth circuits, and in general for any class for which hard problems are known.

Our primary motivation for introducing pseudo-random synthesizers is the parallel construction of pseudo-random functions. The special characteristics of pseudo-random synthesizers lead us to believe that other desired applications may exist. For instance, pseudo-random synthesizers easily define a pseudo-random generator with large output length and the ability to directly compute subsequences of the output. This and the properties discussed in Section 6 suggests that pseudo-random synthesizers may be useful for software implementations of pseudo-random generators or functions. Another possible application of the idea of Construction 4.1 that should be examined is to convert encryption methods that are not immune to chosen plain-text attacks into ones that are immune.

As mentioned in the introduction, in a subsequent work [46] we describe constructions of pseudo-random functions (and other cryptographic primitives) based on several number-theoretic assumptions. These functions can be computed in NC^1 (in fact, even in TC^0) and are very efficient. We note that [46] is motivated by the current work and that it can be described as a direct and efficient construction of n-dimensional pseudo-random synthesizers (see Section 4). The question that arises is whether there exist other such constructions of n-dimensional pseudo-random synthesizers.

An alternative direction for constructing parallel pseudo-random functions is to try and generalize the philosophy behind the Data Encryption Standard (DES) while maintaining its apparent efficiency. Some interesting ideas and results on the generalization of DES can be found in Cleve's work [17, 18].

Acknowledgments

We thank the two anonymous referees and Benny Pinkas for their many helpful comments. We thank Shafi Goldwasser and Jon Sorenson who brought [52] to our attention and Mihir Bellare for his observation described in Section 6.2.

References

[1] M. Ajtai and A. Wigderson, Deterministic simulations of probabilistic constant depth circuits, *ADVCR: Advances in Computing Research*, vol. 5, 1989. Preliminary version: *Proc. 26th Symp. on Foundations of Computer Science*, 1985, pp. 11-19.

[2] W. B. Alexi, B. Chor, O. Goldreich and C. P. Schnorr, RSA and Rabin functions: certain parts are as hard as the whole, *SIAM J. Comput.*, vol. 17(2), 1988, pp. 194-209.

[3] D. Angluin and M. Kharitonov, When won't membership queries help?, *J. Comput. System Sci.*, vol. 50, 1995, pp. 336-355.

[4] L. Babai, N. Nisan and M. Szegedy, Multiparty protocols, pseudorandom generators for logspace, and time-space tradeoffs, *J. Comput. System Sci.*, vol. 45(2), 1992, pp. 204-232.

[5] E. Bach, How to generate factored random numbers, *SIAM J. Comput.*, vol. 17(2), 1988, pp. 179-193.

[6] P. W. Beame, S. A. Cook and H. J. Hoover, Log depth circuits for division and related problems, *SIAM J. Comput.*, vol. 15, 1986, pp. 994-1003.

[7] M. Bellare, O. Goldreich and S. Goldwasser, Incremental cryptography: the case of hashing and signing, *Advances in Cryptology - CRYPTO '94*, Lecture Notes in Computer Science, vol. 839, Springer-Verlag, 1994, pp. 216-233.

[8] M. Bellare, O. Goldreich and S. Goldwasser, Incremental Cryptography with Application to Virus Protection, *Proc. 27th Ann. ACM Symp. on Theory of Computing*, 1995, pp. 45-56.

[9] M. Bellare and S. Goldwasser, New paradigms for digital signatures and message authentication based on non-interactive zero knowledge proofs, *Advances in Cryptology - CRYPTO '89*, Lecture Notes in Computer Science, vol. 435, Springer-Verlag, 1990, pp. 194-211.

[10] E. Biham, D. Boneh and O. Reingold, Generalized Diffie-Hellman modulo a composite is not weaker than factoring, *Theory of Cryptography Library*, Record 97-14 at:
`http://theory.lcs.mit.edu/~tcryptol/homepage.html`

[11] L. Blum, M. Blum and M. Shub, A simple secure unpredictable pseudo-random number generator, *SIAM J. Comput.*, vol. 15, 1986, pp. 364-383.

[12] M. Blum, W. Evans, P. Gemmell, S. Kannan, M. Naor, Checking the correctness of memories, *Algorithmica*, 1994, pp. 225-244. Preliminary version: *Proc. 31st Symp. on Foundations of Computer Science*, 1990.

[13] A. Blum, M. Furst, M. Kearns and R. J. Lipton, Cryptographic primitives based on hard learning problems, *Advances in Cryptology - CRYPTO '93*, Lecture Notes in Computer Science, vol. 773, Springer-Verlag, 1994, pp. 278-291.

[14] M. Blum and S. Micali, How to generate cryptographically strong sequence of pseudo-random bits, *SIAM J. Comput.*, vol. 13, 1984, pp. 850-864.

[15] D. Boneh and R. Lipton, Algorithms for Black-Box fields and their application to cryptography, *Advances in Cryptology - CRYPTO '96*, LNCS, vol. 1109, Springer, 1996, pp. 283-297.

[16] G. Brassard, **Modern cryptology**, Lecture Notes in Computer Science, vol. 325, Springer-Verlag, 1988.

[17] R. Cleve, *Methodologies for designing block ciphers and cryptographic protocols*, Part I, Ph.D. Thesis, University of Toronto, 1989.

[18] R. Cleve, Complexity theoretic issues concerning block ciphers related to D.E.S., *Advances in Cryptology - CRYPTO '90*, Lecture Notes in Computer Science, vol. 537, Springer-Verlag, 1991, pp. 530-544.

[19] B. Chor, A. Fiat and M. Naor, Tracing traitors, *Advances in Cryptology - CRYPTO '94*, Springer-Verlag, 1994, pp. 257-270.

[20] B. Chor and O. Goldreich, Unbiased bits from sources of weak randomness and probabilistic communication complexity, *SIAM J. Comput.*, vol 17, 1988, pp. 230-261. Preliminary version in: *Proc. 26th IEEE Symp. on Foundations of Computer Science*, 1985, pp. 429-442.

[21] W. Diffie and M. Hellman, New directions in cryptography, *IEEE Trans. Inform. Theory*, vol. 22(6), 1976, pp. 644-654.

[22] R. Fischlin and C. P. Schnorr, Stronger security proofs for RSA and Rabin bits, *Advances in Cryptology - EUROCRYPT '97*, Lecture Notes in Computer Science, vol. 1233, Springer-Verlag, 1997, pp. 267-279.

[23] O. Goldreich, Two remarks concerning the Goldwasser-Micali-Rivest signature scheme, *Advances in Cryptology - CRYPTO '86*, Lecture Notes in Computer Science, vol. 263, Springer-Verlag, 1987, pp. 104-110.

[24] O. Goldreich, Towards a theory of software protection, *Proc. 19th Ann. ACM Symp. on Theory of Computing*, 1987, pp. 182-194.

[25] O. Goldreich, **Foundations of Cryptography (Fragments of a Book)**, 1995. Electronic publication: `http://www.eccc.uni-trier.de/eccc/info/ECCC-Books/eccc-books.html` (Electronic Colloquium on Computational Complexity).

[26] O. Goldreich, S. Goldwasser and S. Micali, How to construct random functions, *J. of the ACM.*, vol. 33, 1986, pp. 792-807.

[27] O. Goldreich, S. Goldwasser and S. Micali, On the cryptographic applications of random functions, *Advances in Cryptology - CRYPTO '84*, Lecture Notes in Computer Science, vol. 196, Springer-Verlag, 1985, pp. 276-288.

[28] O. Goldreich and L. Levin, A hard-core predicate for all one-way functions, *Proc. 21st Ann. ACM Symp. on Theory of Computing*, 1989, pp. 25-32.

[29] S. Goldwasser and S. Micali, Probabilistic encryption, *J. Comput. System Sci.*, vol. 28(2), 1984, pp. 270-299.

[30] J. Hastad, A. W. Schrift and A. Shamir, The discrete logarithm modulo a composite hides $O(n)$ bits, *J. of Computer and System Sciences*, vol. 47, 1993, pp. 376-404.

[31] J. Hastad, R. Impagliazzo, L. A. Levin and M. Luby, Construction of a pseudo-random generator from any one-way function, To appear in *SIAM J. Comput.* Preliminary versions by Impagliazzo et. al. in *21st STOC*, 1989 and Hastad in *22nd STOC*, 1990.

[32] A. Herzberg and M. Luby, Public randomness in cryptography, *Advances in Cryptology - CRYPTO '92*, Lecture Notes in Computer Science, vol. 740, Springer-Verlag, 1992, pp. 421-432.

[33] R. Impagliazzo and L. Levin, No better ways to generate hard NP instances than picking uniformly at random, *Proc. 31st IEEE Symp. on Foundations of Computer Science*, 1990 pp. 812-821.

[34] R. Impagliazzo and M. Naor, Efficient Cryptographic schemes provably secure as subset sum, *J. of Cryptology*, vol 9, 1996, pp. 199-216.

[35] R. Impagliazzo and D. Zuckerman, Recycling random bits, *Proc. 30th IEEE Symposium on Foundations of Computer Science*, 1989, pp. 248-253.

[36] R. M. Karp and V. Ramachandran, Parallel algorithms for shared-memory machines, in: J. van Leeuwen, ed., **Handbook of Theoretical Computer Science, vol. A** MIT Press, 1990, pp. 869-941.

[37] M. Kearns and L. Valiant, Cryptographic limitations on learning Boolean formulae and finite automata, *J. of the ACM.*, vol. 41(1), 1994, pp. 67-95.

[38] M. Kharitonov, Cryptographic hardness of distribution-specific learning, *Proc. 25th ACM Symp. on Theory of Computing*, 1993, pp. 372-381.

[39] N. Linial, Y. Mansour and N. Nisan, Constant depth circuits, Fourier transform, and learnability, *J. of the ACM.*, vol 40(3), 1993, pp. 607-620.

[40] M. Luby, **Pseudo-randomness and applications**, Princeton University Press, 1996.

[41] M. Luby and C. Rackoff, How to construct pseudorandom permutations and pseudorandom functions, *SIAM J. Comput.*, vol. 17, 1988, pp. 373-386.

[42] L. A. Levin, One-way function and pseudorandom generators, *Proc. 17th Ann. ACM Symp. on Theory of Computing*, 1985, pp. 363-365.

[43] U. Maurer, Towards the equivalence of breaking the Diffie-Hellman protocol and computing discrete logarithms, *Advances in Cryptology - CRYPTO '94*, LNCS, vol. 740, Springer, 1994, pp. 271-281.

[44] K. McCurley, A key distribution system equivalent to factoring, *J. of Cryptology*, vol 1, 1988, pp. 95-105.

[45] M. Naor and O. Reingold, On the construction of pseudo-random permutations: Luby-Rackoff revisited, To appear in: *J. of Cryptology.* Preliminary version in: *Proc. 29th Ann. ACM Symp. on Theory of Computing*, 1997. pp. 189-199.

[46] M. Naor and O. Reingold, Number-Theoretic constructions of efficient pseudo-random functions, *Proc. 38th IEEE Symp. on Foundations of Computer Science*, 1997.

[47] Nisan, N., Pseudorandom generators for space-bounded computation, *Combinatorica*, vol. 12(4), 1992, pp. 449-461.

[48] N. Nisan, $RL \subseteq SC$, *Proc. 24th Ann. ACM Symp. on Theory of Computing*, 1992, pp. 619-623.

[49] N. Nisan and A. Wigderson, Hardness vs. randomness, *J. Comput. System Sci.*, vol. 49(2), 1994, pp. 149-167. Preliminary version: *Proc. 29th IEEE Symp. on Foundations of Computer Science*, 1988, pp. 2-12.

[50] N. Nisan and D. Zuckerman, Randomness is linear in space, *J. Comput. System Sci.*, vol. 52, 1996, pp. 43–52. Preliminary version: More deterministic simulations in logspace *Proc. 25th Ann. ACM Symp. on Theory of Computing*, 1993, pp. 235-244.

[51] R. Ostrovsky, An efficient software protection scheme, *Proc. 22nd Ann. ACM Symp. on Theory of Computing*, 1990, pp. 514-523.

[52] M. Ram Murty, Artin's conjecture for primitive roots, *The Mathematical Intelligencer*, vol. 10(4), Springer-Verlag, 1988 pp. 59-67.

[53] A. Razborov and S. Rudich, Natural proofs, *J. of Computer and System Sciences*, vol. 55(1), 1997, pp. 24-35. Preliminary version: *Proc. 26th Ann. ACM Symp. on Theory of Computing*, 1994, pp. 204-213.

[54] J. H. Reif and J. D. Tygar, Efficient parallel pseudorandom number generation, *SIAM J. Comput.*, vol. 17(2), 1988, pp. 404-411.

[55] R. L. Rivest, A. Shamir, and L. M. Adleman, A method for obtaining digital signature and public key cryptosystems, *Comma. ACM*, vol. 21, 1978, pp. 120-126.

[56] A. Shamir, On the generation of cryptographically strong pseudo-random number sequences, *ACM Trans. Comput. Sys.*, 1983, pp. 38-44.

[57] V. Shoup, Lower bounds for discrete logarithms and related problems, *Proc. Advances in Cryptology - EUROCRYPT '97*, Lecture Notes in Computer Science, Springer-Verlag, 1997, pp. 256-266.

[58] Z. Shmuely, Composite Diffie-Hellman public-key generating systems are hard to break, Technical Report No. 356, Computer Science Department, Technion, Israel, 1985.

[59] L. G. Valiant, A theory of the learnable, *Comm. ACM*, vol. 27, 1984, pp. 1134-1142.

[60] U. V. Vazirani, Strong communication complexity or generating quasi-random sequences from two communicating semi-random sources, *Combinatorica*, vol. 7, 1987. Preliminary version in: *Proc. 17th Ann. ACM Symp. on Theory of Computing*, 1985, pp. 366-378.

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