Pseudorandom Generators: A Primer

Oded Goldreich
Department of Computer Science and Applied Mathematics
Weizmann Institute of Science, Rehovot, Israel.

July 1, 2008

Contents

Preface **1**

1 Introduction **3**

1.1 The Third Theory of Randomness 4
1.2 Organization 6
1.3 Standard Conventions 6
1.4 The General Paradigm 8
1.4.1 Three fundamental aspects 8
1.4.2 Notational conventions 9
1.4.3 Some instantiations of the general paradigm 10

2 General-Purpose Pseudorandom Generators **11**

2.1 The Basic Definition 11
2.2 The Archetypical Application 13
2.3 Computational Indistinguishability 15
2.3.1 The general formulation 15
2.3.2 Relation to statistical closeness 16
2.3.3 Indistinguishability by Multiple Samples 17
2.4 Amplifying the stretch function 20
2.5 Constructions 21
2.5.1 Background: one-way functions 21
2.5.2 A simple construction 23
2.5.3 An alternative presentation 23
2.5.4 A necessary and sufficient condition 25
2.6 Non-uniformly strong pseudorandom generators 25
2.7 Stronger (Uniform-Complexity) Notions 27
2.7.1 Fooling stronger distinguishers 27
2.7.2 Pseudorandom Functions 28
2.8 Conceptual reflections 30

3 Derandomization of Time-Complexity Classes **32**

3.1 Defining Canonical Derandomizers 32
3.2 Constructing Canonical Derandomizers 35
3.2.1 The construction and its consequences 35

3.2.2 Analyzing the construction (i.e., proof of Theorem 3.5) 37
3.2.3 Construction 3.4 as a general framework 39
3.3 Reflections Regarding Derandomization 40

4 Space-Bounded Distinguishers **42**
4.1 Definitional Issues 42
4.2 Two Constructions 45
4.2.1 Sketches of the proofs of Theorems 4.2 and 4.3 46
4.2.2 Derandomization of space-complexity classes 49

5 Special Purpose Generators **52**
5.1 Pairwise-Independence Generators 53
5.1.1 Constructions 53
5.1.2 Applications (a brief review) 55
5.2 Small-Bias Generators 56
5.2.1 Constructions 57
5.2.2 Applications (a brief review) 58
5.2.3 Generalization 59
5.3 Random Walks on Expanders 60
5.3.1 Background: expanders and random walks on them 60
5.3.2 The generator 62

Notes **63**

Bibliography **67**

Preface

Indistinguishable things are identical.[1]

G.W. Leibniz (1646–1714)

This primer to the theory of pseudorandomness is based on a fresh view at the *question of randomness*, which has been taken by complexity theory. Underlying this view is the postulate that a distribution is random (or rather pseudorandom) if it cannot be told apart from the uniform distribution by any efficient procedure. Thus, (pseudo)randomness is not an inherent property of an object, but is rather subjective to the observer.

At the extreme, this approach says that the question of whether the world is deterministic or allows for some free choice (which may be viewed as sources of randomness) is irrelevant. *What matters is how the world looks to us and to various computationally bounded devices.* That is, if some phenomenon looks random then we may just treat it as if it were random. Likewise, if we can generate sequences that cannot be told apart from the uniform distribution by any efficient procedure, then we can use these sequences in any efficient randomized application instead of the ideal coin tosses that are postulated in the design of this application.

The pivot of the foregoing approach is the notion of *computational indistinguishability*, which refers to pairs of distributions that cannot be told apart by efficient procedures. The most fundamental incarnation of this notion associates efficient procedures with polynomial-time algorithms, but other incarnations that restrict attention to other classes of distinguishing procedures also lead to important insights. Likewise, the *effective generation* of pseudorandom objects, which is of major concern, is actually a general paradigm with numerous useful incarnations (which differ in the computational complexity limitations imposed on the generation process).

Pseudorandom generators are efficient deterministic procedures that stretch short random seeds into longer pseudorandom sequences. Thus, a generic formulation of pseudorandom generators consists of specifying three fundamental aspects – the *stretch measure* of the generators; the class of distinguishers that the generators

[1] This is Leibniz's *Principle of Identity of Indiscernibles.* Leibniz admits that counterexamples to this principle are conceivable but will not occur in real life because God is much too benevolent. We thus believe that he would have agreed to the theme of this text, which asserts that *indistinguishable things should be considered as if they were identical.*

are supposed to fool (i.e., the algorithms with respect to which the *computational indistinguishability* requirement should hold); and the resources that the generators are allowed to use (i.e., their own *computational complexity*).

The archetypical case of pseudorandom generators refers to efficient generators that fool any feasible procedure; that is, the potential distinguisher is any probabilistic polynomial-time algorithm, which may be more complex than the generator itself (which, in turn, has time-complexity bounded by a fixed polynomial). These generators are called general-purpose, because their output can be safely used in any efficient application. Such (general-purpose) pseudorandom generators exist if and only if one-way functions exist.

In contrast to such (general-purpose) pseudorandom generators, for the purpose of derandomization a relaxed definition of pseudorandom generators suffices. In particular, for such a purpose, one may use pseudorandom generators that are somewhat more complex than the potential distinguisher (which represents a randomized algorithm to be derandomized). Following this approach, adequate pseudorandom generators yield a full derandomization of $\mathcal{BPP}$ (i.e., $\mathcal{BPP} = \mathcal{P}$), and such generators can be constructed based on the assumption that some problems in $\mathcal{E}$ have no sub-exponential size circuits.

It is also beneficial to consider pseudorandom generators that fool space-bounded distinguishers and generators that exhibit some limited random behavior (e.g., outputting a pair-wise independent or a small-bias sequence). Such (special-purpose) pseudorandom generators can be constructed without relying on any computational complexity assumption.

Note: The study of pseudorandom generators is part of complexity theory (cf, e.g., [19]); in fact, the current primer is an abbreviated (and somewhat revised) version of [19, Chap. 8].

Chapter 1

Introduction

The "question of randomness" has been puzzling thinkers for ages. Aspects of this question range from philosophical doubts regarding the existence of randomness (in the world) and reflections on the meaning of randomness (in our thinking) to technical questions regarding the measuring of randomness. Among many other things, the second half of the 20th century has witnessed the development of three theories of randomness, which address different aspects of the foregoing question.

The first theory (cf., [13]), initiated by Shannon [51], views randomness as representing *lack of information*, which in turn is modeled by a probability distribution on the possible values of the missing data. Indeed, Shannon's Information Theory is rooted in probability theory. Information Theory is focused at distributions that are not perfectly random (i.e., encode information in a redundant manner), and characterizes perfect randomness as the extreme case in which the *information contents* is maximized (i.e., in this case there is no redundancy at all). Thus, perfect randomness is associated with a unique distribution – the uniform one. In particular, by definition, one cannot (deterministically) generate such perfect random strings from shorter random seeds.

The second theory (cf., [33, 34]), initiated by Solomonov [52], Kolmogorov [30], and Chaitin [11], views randomness as representing lack of structure, which in turn is reflected in the length of the most succinct (effective) description of the object. The notion of a succinct and *effective description* refers to a process that transforms the succinct description to an explicit one. Indeed, this theory of randomness is rooted in computability theory and specifically in the notion of a universal language (equiv., universal machine or computing device). It measures the randomness (or complexity) of objects in terms of the shortest program (for a fixed universal machine) that generates the object.[1] Like Shannon's theory, Kolmogorov Complexity is quantitative and perfect random objects appear as an extreme case. However, following Kolmogorov's approach one may say that a single object, rather than a distribution over objects, is perfectly random. Still, by definition, one cannot (deterministically) generate strings of high Kolmogorov Complexity from short

[1] We mention that Kolmogorov's approach is inherently intractable (i.e., Kolmogorov Complexity is uncomputable).

random seeds.

1.1 The Third Theory of Randomness

The third theory, which is the focus of the current primer, views randomness as an effect on an observer and thus as being relative to the *observer's abilities* (of analysis). The observer's abilities are captured by its computational abilities (i.e., the complexity of the processes that the observer may apply), and hence this theory of randomness is rooted in complexity theory. This theory of randomness is explicitly aimed at providing a notion of randomness that, unlike the previous two notions, allows for an efficient (and deterministic) generation of random strings from shorter random seeds. The heart of this theory is the suggestion to view objects as equal if they cannot be told apart by any efficient procedure. Consequently, a distribution that cannot be efficiently distinguished from the uniform distribution will be considered random (or rather called pseudorandom). Thus, randomness is not an "inherent" property of objects (or distributions) but is rather relative to an observer (and its computational abilities). To illustrate this perspective, let us consider the following mental experiment.

> Alice and Bob play "head or tail" in one of the following four ways. In each of them, Alice flips an unbiased coin and Bob is asked to guess its outcome *before* the coin hits the floor. The alternative ways differ by the knowledge Bob has before making his guess.
>
> In the first alternative, Bob has to announce his guess before Alice flips the coin. Clearly, in this case Bob wins with probability 1/2.
>
> In the second alternative, Bob has to announce his guess while the coin is spinning in the air. Although the outcome is *determined in principle* by the motion of the coin, Bob does not have accurate information on the motion. Thus we believe that, also in this case, Bob wins with probability 1/2.
>
> The third alternative is similar to the second, except that Bob has at his disposal sophisticated equipment capable of providing accurate *information* on the coin's motion as well as on the environment effecting the outcome. However, Bob cannot process this information in time to improve his guess.
>
> In the fourth alternative, Bob's recording equipment is directly connected to a *powerful computer* programmed to solve the motion equations and output a prediction. It is conceivable that in such a case Bob can substantially improve his guess of the outcome of the coin.

We conclude that the randomness of an event is relative to the information and computing resources at our disposal. At the extreme, even events that are fully determined by public information may be perceived as random events by an observer that lacks the relevant information and/or the ability to process it. Our focus will be on the lack of sufficient processing power, and not on the lack of sufficient information. The lack of sufficient processing power may be due either to the formidable amount of computation required (for analyzing the event in question) or to the fact that the observer happens to be very limited.

A natural notion of pseudorandomness arises – a distribution is *pseudorandom* if no efficient procedure can distinguish it from the uniform distribution, where efficient procedures are associated with (probabilistic) polynomial-time algorithms. This specific notion of pseudorandomness is indeed the most fundamental one, and much of this text is focused on it. Weaker notions of pseudorandomness arise as well – they refer to indistinguishability by weaker procedures such as space-bounded algorithms, constant-depth circuits, etc. Stretching this approach even further one may consider algorithms that are designed on purpose so not to distinguish even weaker forms of "pseudorandom" sequences from random ones (where such algorithms arise naturally when trying to convert some natural randomized algorithm into deterministic ones; see Chapter 5).

The foregoing discussion has focused at one aspect of the pseudorandomness question – the resources or type of the observer (or potential distinguisher). Another important aspect is whether such pseudorandom sequences can be generated from much shorter ones, and at what cost (or complexity). A natural approach requires the generation process to be efficient, and furthermore to be fixed before the specific observer is determined. Coupled with the aforementioned strong notion of pseudorandomness, this yields the archetypical notion of pseudorandom generators – those operating in (fixed) polynomial-time and producing sequences that are indistinguishable from uniform ones by *any* polynomial-time observer. In particular, this means that the distinguisher is allowed more resources than the generator. Such (general-purpose) pseudorandom generators (discussed in Chapter 2) allow to decrease the randomness complexity of *any efficient application*, and are thus of great relevance to randomized algorithms and cryptography. The term *general-purpose* is meant to emphasize the fact that the same generator is good for all efficient applications, including those that consume more resources than the generator itself.

Although general-purpose pseudorandom generators are very appealing, there are important reasons for considering also the opposite relation between the complexities of the generation and distinguishing tasks; that is, allowing the pseudorandom generator to use more resources (e.g., time or space) than the observer it tries to fool. This alternative is natural in the context of derandomization (i.e., converting randomized algorithms to deterministic ones), where the crucial step is replacing the random input of an algorithm by a pseudorandom input, which in turn can be generated based on a much shorter random seed. In particular, when derandomizing a probabilistic polynomial-time algorithm, the observer (to be fooled by the generator) is a fixed algorithm. In this case employing a more complex generator merely means that the complexity of the derived deterministic algorithm is dominated by the complexity of the generator (rather than by the complexity of the original randomized algorithm). Needless to say, allowing the generator to use more resources than the observer that it tries to fool makes the task of designing pseudorandom generators potentially easier, and enables derandomization results that are not known when using general-purpose pseudorandom generators. The usefulness of this approach is demonstrated in Chapters 3 through 5.

We note that the goal of all types of pseudorandom generators is to allow the generation of "sufficiently random" sequences based on much shorter random seeds. Thus, pseudorandom generators offer significant saving in the randomness complexity of various applications (and in some cases eliminating randomness altogether). Saving on randomness is valuable because many applications are severely limited in their ability to generate or obtain truly random bits. Furthermore, typically, generating truly random bits is significantly more expensive than standard computation steps. Thus, randomness is a computational resource that should be considered on top of time complexity (analogously to the consideration of space complexity).

1.2 Organization

We start by presenting some standard conventions (see Section 1.3). Next, in Section 1.4, we present the general paradigm underlying the various notions of pseudorandom generators. The archetypical case of general-purpose pseudorandom generators is presented in Chapter 2. We then turn to alternative notions of pseudorandom generators: generators that suffice for the derandomization of complexity classes such as $\mathcal{BPP}$ are discussed in Chapter 3; pseudorandom generators in the domain of space-bounded computations are discussed in Chapter 4; and special-purpose generators are discussed in Chapter 5.

The text is organized to facilitate the possibility of focusing on the notion of general-purpose pseudorandom generators (presented in Chapter 2). This notion is most relevant to computer science at large. Furthermore, the technical details presented in Chapter 2 are relatively simpler than those presented in Chapters 3 and 4.

The current primer is an abbreviated (and somewhat revised) version of [19, Chap. 8]. Additional connections between randomness and computation are discussed in other chapters of [19].

Preliminaries. We assume a basic familiarity with elementary probability theory and randomized algorithms (see, e.g., [38]). In particular, standard conventions regarding random variables (presented next) will be extensively used.

1.3 Standard Conventions

Throughout the entire text we refer only to *discrete* probability distributions. Specifically, the underlying probability space consists of the set of all strings of a certain length ℓ, taken with uniform probability distribution. That is, the sample space is the set of all ℓ-bit long strings, and each such string is assigned probability measure $2^{-\ell}$. Traditionally, *random variables* are defined as functions from the sample space to the reals. Abusing the traditional terminology, we use the term random variable also when referring to functions mapping the sample space into the set of binary strings. We often do not specify the probability space, but rather talk directly about random variables. For example, we may say that X is a random variable assigned values in the set of all strings such that $\Pr[X=00] = \frac{1}{4}$ and $\Pr[X = 111] = \frac{3}{4}$. (Such a random variable may be defined over the sample space $\{0,1\}^2$ such that $X(11) = 00$ and $X(00) = X(01) = X(10) = 111$.) One important case of a random variable is the output of a randomized process (e.g., a probabilistic polynomial-time algorithm).

All our probabilistic statements refer to random variables that are defined beforehand. Typically, we may write $\Pr[f(X) = 1]$, where X is a random variable defined beforehand (and f is a function). An important convention is that *all occurrences of the same symbol in a probabilistic statement refer to the same* (unique) *random variable*. Hence, if $B(\cdot,\cdot)$ is a Boolean expression depending on two variables, and X is a random variable then $\Pr[B(X,X)]$ denotes the probability that $B(x,x)$ holds when x is chosen with probability $\Pr[X=x]$. For example, for every random variable X, we have $\Pr[X=X] = 1$. We stress that if we wish to discuss the probability that $B(x,y)$ holds when x and y are chosen independently with identical probability distribution, then we will define *two* independent random variables each with the same probability distribution. Hence, if X and Y are two independent random variables then $\Pr[B(X,Y)]$ denotes the probability that $B(x,y)$ holds when the pair (x,y) is chosen with probability $\Pr[X=x]\cdot\Pr[Y=y]$. For example, for every two independent random variables, X and Y, we have $\Pr[X = Y] = 1$ only if both X and Y are trivial (i.e., assign the entire probability mass to a single string).

Throughout the entire text, U_n denotes a random variable uniformly distributed over the set of all strings of length n. Namely, $\Pr[U_n=\alpha]$ equals 2^{-n} if $\alpha \in \{0,1\}^n$ and equals 0 otherwise. We often refer to the distribution of U_n as the uniform distribution (neglecting to qualify that it is uniform over $\{0,1\}^n$). In addition, we occasionally use random variables (arbitrarily) distributed over $\{0,1\}^n$ or $\{0,1\}^{\ell(n)}$, for some function $\ell:\mathbb{N}\to\mathbb{N}$. Such random variables are typically denoted by X_n, Y_n, Z_n, etc. We stress that in some cases X_n is distributed over $\{0,1\}^n$, whereas in other cases it is distributed over $\{0,1\}^{\ell(n)}$, for some function ℓ (which is typically a polynomial). We often talk about probability ensembles, which are infinite sequence of random variables $\{X_n\}_{n\in\mathbb{N}}$ such that each X_n ranges over strings of length bounded by a polynomial in n.

Statistical difference. The statistical distance (a.k.a variation distance) between the random variables X and Y is defined as

$$\frac{1}{2}\cdot\sum_v |\Pr[X = v] - \Pr[Y = v]| \;=\; \max_S\{\Pr[X \in S] - \Pr[Y \in S]\}. \tag{1.1}$$

We say that X is δ-close (resp., δ-far) to Y if the statistical distance between them is at most (resp., at least) δ.

1.4 The General Paradigm

We advocate a unified view of various notions of pseudorandom generators. That is, we view these notions as incarnations of a general abstract paradigm, to be presented in this section. A reader who is interested only in one of these incarnations, may still use this section as a general motivation towards the specific definitions used later. On the other hand, some readers may prefer reading this section after studying one of the specific incarnations.

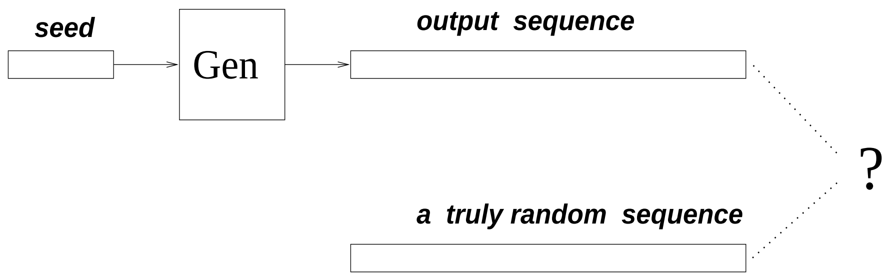

Figure 1.1: Pseudorandom generators – an illustration.

1.4.1 Three fundamental aspects

A generic formulation of pseudorandom generators consists of specifying three fundamental aspects – the *stretch measure* of the generators; the class of distinguishers that the generators are supposed to fool (i.e., the algorithms with respect to which the *computational indistinguishability* requirement should hold); and the resources that the generators are allowed to use (i.e., their own *computational complexity*). Let us elaborate.

Stretch function: A necessary requirement from any notion of a pseudorandom generator is that the generator is a *deterministic algorithm* that stretches short strings, called seeds, into longer output sequences.[2] Specifically, this algorithm stretches k-bit long seeds into $\ell(k)$-bit long outputs, where $\ell(k) > k$. The function $\ell : \mathbb{N} \to \mathbb{N}$ is called the stretch measure (or stretch function) of the generator. In some settings the specific stretch measure is immaterial (e.g., see Section 2.4).

Computational Indistinguishability: A necessary requirement from any notion of a pseudorandom generator is that the generator "fools" some non-trivial algorithms. That is, it is required that any algorithm taken from a predetermined class of interest cannot distinguish the output produced by the generator (when the generator is fed with a uniformly chosen seed) from a uniformly chosen sequence.

[2]Indeed, the seed represents the randomness that is used in the generation of the output sequences; that is, the randomized generation process is decoupled into a deterministic algorithm and a random seed. This decoupling facilitates the study of such processes.

Thus, we consider a class $\mathcal{D}$ of distinguishers (e.g., probabilistic polynomial-time algorithms) and a class $\mathcal{F}$ of (threshold) functions (e.g., reciprocals of positive polynomials), and require that the generator G satisfies the following: For any $D \in \mathcal{D}$, any $f \in \mathcal{F}$, and for all sufficiently large k's it holds that

$$|\,\mathsf{Pr}[D(G(U_k)) = 1] \,-\, \mathsf{Pr}[D(U_{\ell(k)}) = 1]\,| \;<\; f(k)\,, \tag{1.2}$$

where U_n denotes the uniform distribution over $\{0,1\}^n$, and the probability is taken over U_k (resp., $U_{\ell(k)}$) as well as over the coin tosses of algorithm D in case it is probabilistic. The reader may think of such a distinguisher, D, as of an observer that tries to tell whether the "tested string" is a random output of the generator (i.e., distributed as $G(U_k)$) or is a truly random string (i.e., distributed as $U_{\ell(k)}$). The condition in Eq. (1.2) requires that D cannot make a meaningful decision; that is, ignoring a negligible difference (represented by $f(k)$), D's verdict is the same in both cases.[3] The archetypical choice is that $\mathcal{D}$ is the set of all probabilistic polynomial-time algorithms, and $\mathcal{F}$ is the set of all functions that are the reciprocal of some positive polynomial.

Complexity of Generation: This aspect refers to the complexity of the generator itself, when viewed as an algorithm. The archetypical choice is that the generator has to work in polynomial-time (i.e., make a number of steps that is polynomial in the length of its input – the seed). Other choices will be discussed as well. We note that placing no computational requirements on the generator (or, alternatively, imposing very mild requirements such as upper-bounding the running-time by a double-exponential function), yields "generators" that can fool any subexponential-size circuit family.[4]

1.4.2 Notational conventions

We will consistently use k for denoting the length of the seed of a pseudorandom generator, and $\ell(k)$ for denoting the length of the corresponding output. In some cases, this makes our presentation a little more cumbersome (since a more natural presentation may specify some other parameters and let the seed-length be a function of the latter). However, our choice has the advantage of focusing attention on the fundamental parameter of pseudorandom generation process – the length of the random seed. We note that whenever a pseudorandom generator is used to "derandomize" an algorithm, n will denote the length of the input to this algorithm, and k will be selected as a function of n.

[3] The class of threshold functions $\mathcal{F}$ should be viewed as determining the class of noticeable probabilities (as a function of k). Thus, we require certain functions (i.e., those presented at the l.h.s of Eq. (1.2)) to be smaller than any noticeable function *on all but finitely many integers*. We call the former functions negligible. Note that a function may be neither noticeable nor negligible (e.g., it may be smaller than any noticeable function on infinitely many values and yet larger than some noticeable function on infinitely many other values).

[4] This fact can be proved via the probabilistic method; see [19, Exer. 8.1].

1.4.3 Some instantiations of the general paradigm

Two important instantiations of the notion of pseudorandom generators relate to polynomial-time distinguishers.

1. General-purpose pseudorandom generators correspond to the case that the generator itself runs in polynomial-time and needs to withstand *any probabilistic polynomial-time distinguisher*, including distinguishers that run for more time than the generator. Thus, the same generator may be used safely in any efficient application. (This notion is treated in Chapter 2.)

2. In contrast, pseudorandom generators intended for derandomization may run more time than the distinguisher, which is viewed as a fixed circuit having size that is upper-bounded by a fixed polynomial. (This notion is treated in Chapter 3.)

In addition, the general paradigm may be instantiated by focusing on the space-complexity of the potential distinguishers (and the generator), rather than on their time-complexity. Furthermore, one may also consider distinguishers that merely reflect probabilistic properties such as pair-wise independence, small-bias, and hitting frequency.

Chapter 2

General-Purpose Pseudorandom Generators

Randomness is playing an increasingly important role in computation: It is frequently used in the design of sequential, parallel and distributed algorithms, and it is of course central to cryptography. Whereas it is convenient to design such algorithms making free use of randomness, it is also desirable to minimize the usage of randomness in real implementations. Thus, general-purpose pseudorandom generators (as defined next) are a key ingredient in an "algorithmic tool-box" – they provide an automatic compiler of programs written with free usage of randomness into programs that make an economical use of randomness.

Organization of this chapter. Since this is a relatively long chapter, a short road-map seems in place. In Section 2.1 we provide the basic definition of general-purpose pseudorandom generators, and in Section 2.2 we describe their archetypical application (which was eluded to in the former paragraph). In Section 2.3 we provide a wider perspective on the notion of computational indistinguishability that underlies the basic definition, and in Section 2.4 we justify the little concern (shown in Section 2.1) regarding the specific stretch function. In Section 2.5 we address the existence of general-purpose pseudorandom generators. In Section 2.6 we motivate and discuss a non-uniform version of computational indistinguishability. We conclude by reviewing other variants and reflecting on various conceptual aspects of the notions discussed in this chapter (see Sections 2.7 and 2.8, resp.).

2.1 The Basic Definition

Loosely speaking, general-purpose pseudorandom generators are efficient deterministic programs that expand short randomly selected seeds into longer pseudorandom bit sequences, where the latter are defined as computationally indistinguishable from truly random sequences by *any* efficient algorithm. Identifying efficiency with polynomial-time operation, this means that the generator (being a fixed algorithm) works within *some fixed* polynomial-time, whereas the distinguisher may be *any* algorithm that runs in polynomial-time. Thus, the distinguisher is potentially more complex than the generator; for example, the distinguisher *may* run in time that is cubic in the running-time of the generator. Furthermore, to facilitate the development of this theory, we allow the distinguisher to be probabilistic (whereas the generator remains deterministic as stated previously). We require that such distinguishers cannot tell the output of the generator from a truly random string of similar length, or rather that the difference that such distinguishers may detect (or "sense") is negligible. Here a negligible function is a function that vanishes faster than the reciprocal of any positive polynomial.[1]

Definition 2.1 (general-purpose pseudorandom generator): *A deterministic polynomial-time algorithm G is called a* pseudorandom generator *if there exists a* stretch function, $\ell : \mathbb{N}\rightarrow\mathbb{N}$ (satisfying $\ell(k) > k$ for all k), *such that for any probabilistic polynomial-time algorithm D, for any positive polynomial p, and for all sufficiently large k's it holds that*

$$|\,\mathsf{Pr}[D(G(U_k)) = 1] \;-\; \mathsf{Pr}[D(U_{\ell(k)}) = 1]\,| \;<\; \frac{1}{p(k)} \tag{2.1}$$

where U_n denotes the uniform distribution over $\{0,1\}^n$ and the probability is taken over U_k (resp., $U_{\ell(k)}$) as well as over the internal coin tosses of D.

Thus, Definition 2.1 is derived from the generic framework (presented in Section 1.4) by taking the class of distinguishers to be the set of all probabilistic polynomial-time algorithms, and taking the class of (noticeable) threshold functions to be the set of all functions that are the reciprocals of some positive polynomial.[2] Indeed, the principles underlying Definition 2.1 were discussed in Section 1.4 (and will be further discussed in Section 2.3).

We note that Definition 2.1 does not make any requirement regarding the stretch function $\ell : \mathbb{N}\rightarrow\mathbb{N}$, except for the generic requirement that $\ell(k) > k$ for all k. Needless to say, the larger ℓ is, the more useful the pseudorandom generator is. Of course, ℓ is upper-bounded by the running-time of the generator (and hence by a polynomial). In Section 2.4 we show that any pseudorandom generator (even one having minimal stretch $\ell(k) = k+1$) can be used for constructing a pseudorandom generator having any desired (polynomial) stretch function. But before doing so, we rigorously discuss the "saving in randomness" offered by pseudorandom generators, and provide a wider perspective on the notion of computational indistinguishability that underlies Definition 2.1.

[1] Definition 2.1 requires that the functions representing the distinguishing gap of certain algorithms should be smaller than the reciprocal of any positive polynomial for all but finitely many k's, and the former functions are called *negligible*. The notion of negligible probability is robust in the sense that any event that occurs with negligible probability will occur with negligible probability also when the experiment is repeated a "feasible" (i.e., polynomial) number of times.

[2] The latter choice is naturally coupled with the association of efficient computation with polynomial-time algorithms: An event that occurs with noticeable probability occurs almost always when the experiment is repeated a "feasible" (i.e., polynomial) number of times.

2.2 The Archetypical Application

We note that "pseudo-random number generators" appeared with the first computers, and have been used ever since for generating random choices (or samples) for various applications. However, typical implementations use generators that are not pseudorandom according to Definition 2.1. Instead, at best, these generators are shown to pass *some* ad-hoc statistical test (cf., [29]). We warn that the fact that a "pseudo-random number generator" passes some statistical tests, does not mean that it will pass a new test and that it will be good for a future (untested) application. Needless to say, the approach of subjecting the generator to some ad-hoc tests fails to provide general results of the form "for *all* practical purposes using the output of the generator is as good as using truly unbiased coin tosses." In contrast, the approach encompassed in Definition 2.1 aims at such generality, and in fact is tailored to obtain it: The notion of computational indistinguishability, which underlines Definition 2.1, covers all possible efficient applications and guarantees that for all of them pseudorandom sequences are as good as truly random ones. Indeed, any efficient randomized algorithm maintains its performance when its internal coin tosses are substituted by a sequence generated by a pseudorandom generator. This substitution is spell-out next.

Construction 2.2 (typical application of pseudorandom generators): *Let G be a pseudorandom generator with stretch function $\ell:\mathbb{N}\to\mathbb{N}$. Let A be a probabilistic polynomial-time algorithm, and $\rho:\mathbb{N}\to\mathbb{N}$ denote its randomness complexity. Denote by $A(x,r)$ the output of A on input x and coin tosses sequence $r \in \{0,1\}^{\rho(|x|)}$. Consider the following randomized algorithm, denoted A_G:*

> *On input x, set $k = k(|x|)$ to be the smallest integer such that $\ell(k) \geq \rho(|x|)$, uniformly select $s \in \{0,1\}^k$, and output $A(x,r)$, where r is the $\rho(|x|)$-bit long prefix of $G(s)$.*

That is, $A_G(x,s) = A(x,G'(s))$, for $|s| = k(|x|) = \mathrm{argmin}_i\{\ell(i) \geq \rho(|x|)\}$, where $G'(s)$ is the $\rho(|x|)$-bit long prefix of $G(s)$.

Thus, using A_G instead of A, the randomness complexity is reduced from ρ to $\ell^{-1}\circ\rho$, while (as we show next) it is infeasible to find inputs (i.e., x's) on which the *noticeable behavior* of A_G is different from the one of A. For example, if $\ell(k) = k^2$, then the randomness complexity is reduced from ρ to $\sqrt{\rho}$. We stress that the pseudorandom generator G is *universal*; that is, it can be applied to reduce the randomness complexity of *any* probabilistic polynomial-time algorithm A.

Proposition 2.3 *Let A, ρ and G be as in Construction 2.2, and suppose that $\rho:\mathbb{N}\to\mathbb{N}$ is 1-1. Then, for every pair of probabilistic polynomial-time algorithms,* a finder F *and* a tester T, *every positive polynomial p and all sufficiently long n's*

$$\sum_{x\in\{0,1\}^n} \mathsf{Pr}[F(1^n)=x]\cdot|\,\Delta_{A,T}(x)\,| \;<\; \frac{1}{p(n)} \tag{2.2}$$

where $\Delta_{A,T}(x) \stackrel{\rm def}{=} \mathsf{Pr}[T(x,A(x,U_{\rho(|x|)}))=1] - \mathsf{Pr}[T(x,A_G(x,U_{k(|x|)}))=1]$*, and the probabilities are taken over the* U_m*'s as well as over the internal coin tosses of the algorithms* F *and* T*.*

Algorithm F represents a potential attempt to find an input x on which the output of A_G is distinguishable from the output of A. This "attempt" may be benign as in the case that a user employs algorithm A_G on inputs that are generated by some probabilistic polynomial-time application. However, the attempt may also be adversarial as in the case that a user employs algorithm A_G on inputs that are provided by a potentially malicious party. The potential tester, denoted T, represents the potential use of the output of algorithm A_G, and captures the requirement that this output be as good as a corresponding output produced by A. Thus, T is given x as well as the corresponding output produced either by $A_G(x) \stackrel{\rm def}{=} A(x,U_{k(|x|)})$ or by $A(x) = A(x,U_{\rho(|x|)})$, and it is required that T cannot tell the difference. In the case that A is a probabilistic polynomial-time *decision procedure*, this means that it is infeasible to find an x on which A_G decides incorrectly (i.e., differently than A). In the case that A is a *search procedure for some NP-relation*, it is infeasible to find an x on which A_G outputs a wrong solution.

Proof Sketch: The proposition is proven by showing that any triple (A,F,T) violating the claim can be converted into an algorithm D that distinguishes the output of G from the uniform distribution, in contradiction to the hypothesis. The key observation is that for every $x \in \{0,1\}^n$ it holds that

$$\Delta_{A,T}(x) = \mathsf{Pr}[T(x,A(x,U_{\rho(n)}))=1] - \mathsf{Pr}[T(x,A(x,G'(U_{k(n)})))=1], \qquad (2.3)$$

where $G'(s)$ is the $\rho(n)$-bit long prefix of $G(s)$. Thus, a method for finding a string x such that $|\Delta_{A,T}(x)|$ is large yields a way of distinguishing $U_{\ell(k(n))}$ from $G(U_{k(n)})$; that is, given a sample $r \in \{0,1\}^{\ell(k(n))}$ and using such a string $x \in \{0,1\}^n$, the distinguisher outputs $T(x,A(x,r'))$, where r' is the $\rho(n)$-bit long prefix of r. Indeed, we shall show that the violation of Eq. (2.2), which refers to $\mathsf{E}_{x\leftarrow F(1^n)}[|\Delta_{A,T}(x)|]$, yields a violation of the hypothesis that G is a pseudorandom generator (by finding an adequate string x and using it). This intuitive argument requires a slightly careful implementation, which is provided next.

As a warm-up, consider the following algorithm D. On input r (taken from either $U_{\ell(k(n))}$ or $G(U_{k(n)})$), algorithm D first obtains $x \leftarrow F(1^n)$, where n can be obtained easily from $|r|$ (because ρ is 1-1 and $1^n \mapsto \rho(n)$ is computable via A). Next, D obtains $y = A(x,r')$, where r' is the $\rho(|x|)$-bit long prefix of r. Finally D outputs $T(x,y)$. Note that D is implementable in probabilistic polynomial-time, and that

$$\begin{aligned} D(U_{\ell(k(n))}) &\equiv T(X_n, A(X_n, U_{\rho(n)}))\,, \text{ where } X_n \stackrel{\rm def}{=} F(1^n) \\ D(G(U_{k(n)})) &\equiv T(X_n, A(X_n, G'(U_{k(n)})))\,, \text{ where } X_n \stackrel{\rm def}{=} F(1^n). \end{aligned}$$

Using Eq. (2.3), it follows that $\mathsf{Pr}[D(U_{\ell(k(n))})=1] - \mathsf{Pr}[D(G(U_{k(n)}))=1]$ equals $\mathsf{E}[\Delta_{A,T}(F(1^n))]$, which implies that $\mathsf{E}[\Delta_{A,T}(F(1^n))]$ must be negligible (because otherwise we derive a contradiction to the hypothesis that G is a pseudorandom generator). This yields a weaker version of the proposition asserting that $\mathsf{E}[\Delta_{A,T}(F(1^n))]$ is negligible (rather than that $\mathsf{E}[|\Delta_{A,T}(F(1^n))|]$ is negligible).

In order to prove that $\mathsf{E}[|\Delta_{A,T}(F(1^n))|]$ (rather than to $\mathsf{E}[\Delta_{A,T}(F(1^n))]$) is negligible, we need to modify D a little. Note that the source of trouble is that $\Delta_{A,T}(\cdot)$ may be positive on some x's and negative on others, and thus it may be the case that $\mathsf{E}[\Delta_{A,T}(F(1^n))]$ is small (due to cancelations) even if $\mathsf{E}[|\Delta_{A,T}(F(1^n))|]$ is large. This difficulty can be overcome by determining the sign of $\Delta_{A,T}(\cdot)$ on $x = F(1^n)$ and changing the outcome of D accordingly; that is, the modified D will output $T(x, A(x, r'))$ if $\Delta_{A,T}(x) > 0$ and $1 - T(x, A(x, r'))$ otherwise. Thus, in each case, the contribution of x to the distinguishing gap of the modified D will be $|\Delta_{A,T}(x)|$. We further note that if $|\Delta_{A,T}(x)|$ is small then it does not matter much whether we act as in the case of $\Delta_{A,T}(x) > 0$ or in the case of $\Delta_{A,T}(x) \leq 0$. Thus, it suffices to correctly determine the sign of $\Delta_{A,T}(x)$ in the case that $|\Delta_{A,T}(x)|$ is large, which is certainly a feasible (approximation) task. Details can be found in [19, Sec. 8.2.2]. □

Conclusion. Although Proposition 2.3 refers to standard probabilistic polynomial-time algorithms, a similar construction and analysis applied to any efficient randomized process (i.e., any efficient multi-party computation). Any such process preserves its behavior when replacing its perfect source of randomness (postulated in its analysis) by a pseudorandom sequence (which may be used in the implementation). Thus, given a pseudorandom generator with a large stretch function, *one can considerably reduce the randomness complexity of any efficient application.*

2.3 Computational Indistinguishability

In this section we spell-out (and study) the definition of computational indistinguishability that underlies Definition 2.1.

2.3.1 The general formulation

The (general formulation of the) definition of computational indistinguishability refers to *arbitrary* probability ensembles. Here a probability ensemble is an infinite sequence of random variables $\{Z_n\}_{n\in\mathbb{N}}$ such that each Z_n ranges over strings of length that is polynomially related to n (i.e., there exists a polynomial p such that for every n it holds that $|Z_n| \leq p(n)$ and $p(|Z_n|) \geq n$). We say that $\{X_n\}_{n\in\mathbb{N}}$ and $\{Y_n\}_{n\in\mathbb{N}}$ are computationally indistinguishable if for every feasible algorithm A the difference $d_A(n) \stackrel{\text{def}}{=} |\mathsf{Pr}[A(X_n) = 1] - \mathsf{Pr}[A(Y_n) = 1]|$ is a negligible function in n. That is:

Definition 2.4 (computational indistinguishability): *The probability ensembles $\{X_n\}_{n\in\mathbb{N}}$ and $\{Y_n\}_{n\in\mathbb{N}}$ are* computationally indistinguishable *if for every probabilistic polynomial-time algorithm D, every positive polynomial p, and all sufficiently* *large n,*

$$|\mathsf{Pr}[D(X_n)=1] - \mathsf{Pr}[D(Y_n)=1]| \; < \; \frac{1}{p(n)} \tag{2.4}$$

where the probabilities are taken over the relevant distribution (i.e., either X_n or Y_n) *and over the internal coin tosses of algorithm D. The l.h.s. of Eq.* (2.4)*, when viewed as a function of n, is often called the* distinguishing gap *of D, where $\{X_n\}_{n\in\mathbb{N}}$ and $\{Y_n\}_{n\in\mathbb{N}}$ are understood from the context.*

We can think of D as representing somebody who wishes to distinguish two distributions (based on a given sample drawn from one of the distributions), and think of the output "1" as representing D's verdict that the sample was drawn according to the first distribution. Saying that the two distributions are computationally indistinguishable means that if D is a feasible procedure then its verdict is not really meaningful (because the verdict is almost as often 1 when the sample is drawn from the first distribution as when the sample is drawn from the second distribution). We comment that the absolute value in Eq. (2.4) can be omitted without affecting the definition, and we will often do so without warning.

In Definition 2.1, we required that the probability ensembles $\{G(U_k)\}_{k\in\mathbb{N}}$ and $\{U_{\ell(k)}\}_{k\in\mathbb{N}}$ be computationally indistinguishable. Indeed, an important special case of Definition 2.4 is when one ensemble is uniform, and in such a case we call the other ensemble pseudorandom.

2.3.2 Relation to statistical closeness

Two probability ensembles, $\{X_n\}_{n\in\mathbb{N}}$ and $\{Y_n\}_{n\in\mathbb{N}}$, are said to be statistically close (or statistically indistinguishable) if for every positive polynomial p and all sufficient large n the variation distance between X_n and Y_n is bounded above by $1/p(n)$. Clearly, any two probability ensembles that are statistically close are computationally indistinguishable. Needless to say, this is a trivial case of computational indistinguishability, which is due to information theoretic reasons. In contrast, we shall be interested in *non-trivial cases* (of computational indistinguishability), which correspond to probability ensembles that are statistically far apart.

Indeed, as claimed in Section 1.4 (see [19, Exer. 8.1]), there exist probability ensembles that are statistically far apart and yet are computationally indistinguishable. However, at least one of the two probability ensembles in this unconditional existential claim is *not* polynomial-time constructible.[3] We shall be much more interested in non-trivial cases of computational indistinguishability in which both ensembles are polynomial-time constructible. An important example is provided by the definition of pseudorandom generators. As we shall see (in Theorem 2.14), the existence of one-way functions implies the existence of pseudorandom generators, which in turn implies the existence of *polynomial-time constructible* probability ensembles that are statistically far apart and yet are computationally indistinguishable. We mention that this sufficient condition is also necessary (cf., [15]).

[3] We say that $\{Z_n\}_{n\in\mathbb{N}}$ is polynomial-time constructible if there exists a polynomial-time algorithm S such that $S(1^n)$ and Z_n are identically distributed.

2.3.3 Indistinguishability by Multiple Samples

The definition of computational indistinguishability (i.e., Definition 2.4) refers to distinguishers that obtain a single sample from one of the two relevant probability ensembles (i.e., $\{X_n\}_{n\in\mathbb{N}}$ and $\{Y_n\}_{n\in\mathbb{N}}$). A very natural generalization of Definition 2.4 refers to distinguishers that obtain several independent samples from such an ensemble.

Definition 2.5 (indistinguishability by multiple samples): *Let $s:\mathbb{N}\rightarrow\mathbb{N}$ be polynomially-bounded. Two probability ensembles, $\{X_n\}_{n\in\mathbb{N}}$ and $\{Y_n\}_{n\in\mathbb{N}}$, are* computationally indistinguishable by $s(\cdot)$ samples *if for every probabilistic polynomial-time algorithm, D, every positive polynomial $p(\cdot)$, and all sufficiently large n's*

$$\left|\mathsf{Pr}\left[D(X_n^{(1)},...,X_n^{(s(n))})=1\right]-\mathsf{Pr}\left[D(Y_n^{(1)},...,Y_n^{(s(n))})=1\right]\right| < \frac{1}{p(n)}$$

where $X_n^{(1)}$ through $X_n^{(s(n))}$ and $Y_n^{(1)}$ through $Y_n^{(s(n))}$ are independent random variables such that each $X_n^{(i)}$ is identical to X_n and each $Y_n^{(i)}$ is identical to Y_n.

It turns out that, in the most interesting cases, computational indistinguishability by a single sample implies computational indistinguishability by any polynomial number of samples. One such case is the case of polynomial-time constructible ensembles. We say that the ensemble $\{Z_n\}_{n\in\mathbb{N}}$ is polynomial-time constructible if there exists a polynomial-time algorithm S such that $S(1^n)$ and Z_n are identically distributed.

Proposition 2.6 *Suppose that $X \stackrel{\text{def}}{=} \{X_n\}_{n\in\mathbb{N}}$ and $Y \stackrel{\text{def}}{=} \{Y_n\}_{n\in\mathbb{N}}$ are both polynomial-time constructible, and s be a positive polynomial. Then, X and Y are computationally indistinguishable by a single sample if and only if they are computationally indistinguishable by $s(\cdot)$ samples.*

Clearly, for every polynomial $s \geq 1$, computational indistinguishability by $s(\cdot)$ samples implies computational indistinguishability by a single sample. We now prove that, for efficiently constructible ensembles, indistinguishability by a single sample implies indistinguishability by multiple samples.[4] The proof provides a simple demonstration of a central proof technique, known as the *hybrid technique*, which is a special case of the so-called reducibility argument (cf, e.g., [17, Sec. 2.3.3] or [19, Sec. 7.1.2]).

Proof Sketch:[5] Using the counter-positive, we show that the existence of an efficient algorithm that distinguishes the ensembles X and Y using several samples, implies the existence of an efficient algorithm that distinguishes the ensembles X and Y using a single sample. That is, starting from the distinguishability of $s(n)$-long sequences of samples (either drawn all from X_n or drawn all from Y_n), we consider *hybrid* sequences such that the i^{th} hybrid consists of i samples of X_n followed by $s(n)-i$ samples of Y_n. Note that the "homogeneous" sequences (which we

[4] The requirement that both ensembles are polynomial-time constructible is essential; see, [23].

[5] For more details see [17, Sec. 3.2.3].

assumed to be distinguishable) are the extreme hybrids (i.e., the first and last hybrids). The key observation is that distinguishing the extreme hybrids (towards the contradiction hypothesis) implies distinguishing neighboring hybrids, which in turn yields a procedure for distinguishing single samples of the two original distributions (contradicting the hypothesis that these two distributions are indistinguishable by a single sample). Details follow.

Suppose, towards the contradiction, that D distinguishes $s(n)$ samples of X_n from $s(n)$ samples of Y_n, with a distinguishing gap of $\delta(n)$. Denoting the i^{th} hybrid by H_n^i (i.e., $H_n^i = (X_n^{(1)},...,X_n^{(i)},Y_n^{(i+1)},...,Y_n^{(s(n))})$), this means that D distinguishes the extreme hybrids (i.e., H_n^0 and $H_n^{s(n)}$) with gap $\delta(n)$. It follows that D distinguishes a random pair of neighboring hybrids (i.e., D distinguishes H_n^i from H_n^{i+1}, for a randomly selected i) with gap at least $\delta(n)/s(n)$: the reason being that

$$\begin{aligned}
&\mathsf{E}_{i\in\{0,\dots,s(n)-1\}}\left[\mathsf{Pr}[D(H_n^i)=1]-\mathsf{Pr}[D(H_n^{i+1})=1]\right] \\
&= \frac{1}{s(n)}\cdot\sum_{i=0}^{s(n)-1}\left(\mathsf{Pr}[D(H_n^i)=1]-\mathsf{Pr}[D(H_n^{i+1})=1]\right) \qquad (2.5)\\
&= \frac{1}{s(n)}\cdot\Big(\mathsf{Pr}[D(H_n^0)=1]-\mathsf{Pr}[D(H_n^{s(n)})=1]\Big) \;=\; \frac{\delta(n)}{s(n)}\,.
\end{aligned}$$

The key step in the argument is transforming the distinguishability of neighboring hybrids into distinguishability of single samples of the original ensembles (thus deriving a contradiction). Indeed, using D, we obtain a distinguisher D' of single samples: Given a single sample, algorithm D' selects $i \in \{0,...,s(n)-1\}$ at random, generates i samples from the first distribution and $s(n)-i-1$ samples from the second distribution, invokes D with the $s(n)$-samples sequence obtained when placing the input sample in location $i+1$, and answers whatever D does. That is, on input z and when selecting the index i, algorithm D' invokes D on a sample from the distribution $(X_n^{(1)},...,X_n^{(i)},z,Y_n^{(i+2)},...,Y_n^{(s(n))})$. Thus, the construction of D' relies on the hypothesis that both probability ensembles are polynomial-time constructible. The analysis of D' is based on the following two facts:

1. When invoked on an input that is distributed according to X_n and selecting the index $i \in \{0,...,s(n)-1\}$, algorithm D' behaves like $D(H_n^{i+1})$, because $(X_n^{(1)},...,X_n^{(i)},X_n,Y_n^{(i+2)},...,Y_n^{(s(n))}) \equiv H_n^{i+1}$.

2. When invoked on an input that is distributed according to Y_n and selecting the index $i \in \{0,...,s(n)-1\}$, algorithm D' behaves like $D(H_n^i)$, because $(X_n^{(1)},...,X_n^{(i)},Y_n,Y_n^{(i+2)},...,Y_n^{(s(n))}) \equiv H_n^i$.

Thus, the distinguishing gap of D' (between Y_n and X_n) is captured by Eq. (2.5), and the claim follows. □

The hybrid technique – a digest: The hybrid technique constitutes a special type of a "reducibility argument" in which the computational indistinguishability of *complex* ensembles is proved using the computational indistinguishability of *basic* ensembles. The actual reduction is in the other direction: efficiently distinguishing the basic ensembles is reduced to efficiently distinguishing the complex ensembles, and *hybrid* distributions are used in the reduction in an essential way. The following three properties of the construction of the hybrids play an important role in the argument:

1. *The complex ensembles collide with the extreme hybrids.* This property is essential because our aim is proving something that relates to the complex ensembles (i.e., their indistinguishability), while the argument itself refers to the extreme hybrids.

   In the proof of Proposition 2.6 the extreme hybrids (i.e., $H_n^{s(n)}$ and H_n^0) collide with the complex ensembles that represent $s(n)$-ary sequences of samples of one of the basic ensembles.

2. *The basic ensemble are efficiently mapped to neighboring hybrids.* This property is essential because our starting hypothesis relates to the basic ensembles (i.e., their indistinguishability), while the argument itself refers directly to the neighboring hybrids. Thus, we need to translate our knowledge (i.e., computational indistinguishability) of the basic ensembles to knowledge (i.e., computational indistinguishability) of any pair of neighboring hybrids. Typically, this is done by efficiently transforming strings in the range of a basic distribution into strings in the range of a hybrid such that the transformation maps the first basic distribution to one hybrid and the second basic distribution to the neighboring hybrid.

   In the proof of Proposition 2.6 the basic ensembles (i.e., X_n and Y_n) were efficiently transformed into neighboring hybrids (i.e., H_n^{i+1} and H_n^i, respectively). Recall that, in this case, the efficiency of this transformation relied on the hypothesis that both the basic ensembles are polynomial-time constructible.

3. *The number of hybrids is small* (i.e., polynomial). This property is essential in order to deduce the computational indistinguishability of extreme hybrids from the computational indistinguishability of each pair of neighboring hybrids. Typically, the "distinguishability gap" established in the argument losses a factor that is proportional to the number of hybrids. This is due to the fact that the gap between the extreme hybrids is upper-bounded by the sum of the gaps between neighboring hybrids.

   In the proof of Proposition 2.6 the number of hybrids equals $s(n)$ and the aforementioned loss is reflected in Eq. (2.5).

We remark that in the course of an hybrid argument, a distinguishing algorithm referring to the complex ensembles is being analyzed and even invoked on arbitrary hybrids. The reader may be annoyed of the fact that the algorithm "was not designed to work on such hybrids" (but rather only on the extreme hybrids). However, *an algorithm is an algorithm*: once it exists we can invoke it on inputs of our choice, and analyze its performance on arbitrary input distributions.

2.4 Amplifying the stretch function

Recall that the definition of pseudorandom generators (i.e., Definition 2.1) makes a minimal requirement regarding their stretch; that is, it is only required that the output of such generators is longer than their input. Needless to say, we seek pseudorandom generators with a much more significant stretch, firstly because the stretch determines the saving in randomness obtained via Construction 2.2. It turns out (see Construction 2.7) that pseudorandom generators of any stretch function (and in particular of minimal stretch $\ell_1(k) \stackrel{\rm def}{=} k+1$) can be easily converted into pseudorandom generators of any desired (polynomially bounded) stretch function, ℓ. On the other hand, since pseudorandom generators are required (by Definition 2.1) to run in polynomial time, their stretch must be polynomially bounded.

Construction 2.7 *Let G_1 be a pseudorandom generator with stretch function $\ell_1(k) = k+1$, and ℓ be any polynomially bounded stretch function that is polynomial-time computable. Let*

$$G(s) \stackrel{\rm def}{=} \sigma_1\sigma_2\cdots\sigma_{\ell(|s|)} \tag{2.6}$$

where $x_0 = s$ and $x_i\sigma_i = G_1(x_{i-1})$, for $i = 1, ..., \ell(|s|)$. That is, σ_i is the last bit of $G_1(x_{i-1})$ and x_i is the $|s|$-bit long prefix of $G_1(x_{i-1})$.

Needless to say, G is polynomial-time computable and has stretch ℓ. An alternative construction is obtained by iteratively applying G on increasingly longer input lengths (see [19, Exer. 8.11]).

Proposition 2.8 *Let G_1 and G be as in Construction 2.7. Then G constitutes a pseudorandom generator.*

Proof Sketch: The proposition is proven using the *hybrid technique*, presented and discussed in Section 2.3. Here (for $i = 0, ..., \ell(k)$) we consider the hybrid distributions H_k^i defined by

$$H_k^i \stackrel{\rm def}{=} U_i^{(1)} \cdot g_{\ell(k)-i}(U_k^{(2)}),$$

where $\cdot$ denotes the concatenation of strings, $g_j(x)$ denotes the j-bit long prefix of $G(x)$, and $U_i^{(1)}$ and $U_k^{(2)}$ are independent uniform distributions (over $\{0,1\}^i$ and $\{0,1\}^k$, respectively). The extreme hybrids (i.e., H_k^0 and H_k^k) correspond to $G(U_k)$ and $U_{\ell(k)}$, whereas distinguishability of neighboring hybrids can be worked into distinguishability of $G_1(U_k)$ and U_{k+1}. Details follow.

Suppose that one could distinguish H_k^i from H_k^{i+1}. Defining $F(z)$ (resp., $L(z)$) as the first $|z|-1$ bits (resp., last bit) of z, and using $g_j(s) = L(G_1(s))\cdot g_{j-1}(F(G_1(s)))$ (for $j \geq 1$), we have

$$H_k^i \equiv U_i^{(1)} \cdot L(G_1(U_k^{(2)})) \cdot g_{(\ell(k)-i)-1}(F(G_1(U_k^{(2)})))$$

and

$$\begin{aligned} H_k^{i+1} &= U_{i+1}^{(1')} \cdot g_{\ell(k)-(i+1)}(U_k^{(2)}) \\ &\equiv U_i^{(1)} \cdot L(U_{k+1}^{(2')}) \cdot g_{(\ell(k)-i)-1}(F(U_{k+1}^{(2')})). \end{aligned}$$

Now, incorporating the generation of $U_i^{(1)}$ and the evaluation of $g_{\ell(k)-i-1}$ into the distinguisher, it follows that we distinguish $G_1(U_k^{(2)})$ from $U_{k+1}^{(2')}$, in contradiction to the pseudorandomness of G_1. For further details see [19, Sec. 8.2.4] (or [17, Sec. 3.3.3]). $\square$

Conclusion. In view of the foregoing, when talking about the mere existence of pseudorandom generators, in the sense of Definition 2.1, we may ignore the specific stretch function.

2.5 Constructions

The constructions surveyed in this section "transform" computational difficulty, in the form of one-way functions, into generators of pseudorandomness. We thus start by reviewing the definition of one-way functions as well as some related results.

2.5.1 Background: one-way functions

One-way functions are functions that are easy to compute but hard to invert (in an average-case sense).

Definition 2.9 (one-way functions): *A function $f:\{0,1\}^* \to \{0,1\}^*$ is called* one-way *if the following two conditions hold:*

1. Easy to evaluate: *There exist a polynomial-time algorithm A such that $A(x) = f(x)$ for every $x \in \{0,1\}^*$.*
2. Hard to invert: *For every probabilistic polynomial-time algorithm A', every positive polynomial p, and all sufficiently large n,*

$$\mathsf{Pr}_{x\in\{0,1\}^n}[A'(f(x),1^n) \in f^{-1}(f(x))] \ < \ \frac{1}{p(n)} \tag{2.7}$$

where the probability is taken uniformly over the possible choices of $x \in \{0,1\}^n$ and over the internal coin tosses of algorithm A'.

Algorithm A' is given the auxiliary input 1^n so as to allow it to run in time polynomial in the length of x, which is important in case f drastically shrinks its input (e.g., $|f(x)| = O(\log|x|)$). Typically (and, in fact, without loss of generality), the function f is length preserving, in which case the auxiliary input 1^n is redundant. Note that A' is not required to output a specific preimage of $f(x)$; any preimage (i.e., element in the set $f^{-1}(f(x))$) will do. (Indeed, in case f is 1-1, the string x is the only preimage of $f(x)$ under f; but in general there may be other preimages.) It is required that algorithm A' fails (to find a preimage) with overwhelming probability, when the probability is also taken over the input distribution. That is, f is "typically" hard to invert, not merely hard to invert in some ("rare") cases.

On hard-core predicates. Recall that saying that a function f is one-way means that given a typical y (in the range of f) it is infeasible to find a preimage of y under f. This does not mean that it is infeasible to find partial information about the preimage(s) of y under f. Specifically, it may be easy to retrieve half of the bits of the preimage (e.g., given a one-way function f consider the function f' defined by $f'(x,r) \stackrel{\text{def}}{=} (f(x),r)$, for every $|x| = |r|$). We note that hiding partial information (about the function's preimage) plays an important role in the construction of pseudorandom generators (as well as in other advanced constructs). With this motivation in mind, we will show that essentially any one-way function hides specific partial information about its preimage, where this partial information is easy to compute from the preimage itself. This partial information can be considered a "hard core" of the difficulty of inverting f. Loosely speaking, a *polynomial-time computable* (Boolean) predicate b, is called a hard-core of a function f if no feasible algorithm, given $f(x)$, can guess $b(x)$ with success probability that is non-negligibly better than one half.

Definition 2.10 (hard-core predicates): *A polynomial-time computable predicate $b : \{0,1\}^* \to \{0,1\}$ is called a* hard-core *of a function f if for every probabilistic polynomial-time algorithm A', every positive polynomial $p(\cdot)$, and all sufficiently large n's*

$$\mathsf{Pr}_{x\in\{0,1\}^n}\left[A'(f(x)) = b(x)\right] < \frac{1}{2} + \frac{1}{p(n)}$$

where the probability is taken uniformly over the possible choices of $x \in \{0,1\}^n$ and over the internal coin tosses of algorithm A'.

Note that for every $b : \{0,1\}^* \to \{0,1\}$ and $f : \{0,1\}^* \to \{0,1\}^*$, there exist obvious algorithms that guess $b(x)$ from $f(x)$ with success probability at least one half (e.g., the algorithm that, obliviously of its input, outputs a uniformly chosen bit). Also, if b is a hard-core predicate (of any function) then it follows that b is almost unbiased (i.e., for a uniformly chosen x, the difference $|\mathsf{Pr}[b(x)=0] - \mathsf{Pr}[b(x)=1]|$ must be a negligible function in n).

Since b itself is polynomial-time computable, the failure of efficient algorithms to approximate $b(x)$ from $f(x)$ (with success probability that is non-negligibly higher than one half) must be due either to an information loss of f (i.e., f not being one-to-one) or to the difficulty of inverting f. For example, for $\sigma \in \{0,1\}$ and $x' \in \{0,1\}^*$, the predicate $b(\sigma x') = \sigma$ is a hard-core of the function $f(\sigma x') \stackrel{\text{def}}{=} 0x'$. Hence, in this case the fact that b is a hard-core of the function f is due to the fact that f loses information (specifically, the first bit: σ). On the other hand, in the case that f loses no information (i.e., f is one-to-one) a hard-core for f may exist only if f is hard to invert. In general, the interesting case is when being a hard-core is a computational phenomenon rather than an information theoretic one (which is due to "information loss" of f). It turns out that any one-way function has a modified version that possesses a hard-core predicate.

Theorem 2.11 (a generic hard-core predicate): *For any one-way function f, the inner-product mod 2 of x and r, denoted $b(x,r)$, is a hard-core of $f'(x,r) = (f(x),r)$.*

In other words, Theorem 2.11 asserts that, given $f(x)$ and a random subset $S \subseteq [|x|]$, it is infeasible to guess $\oplus_{i\in S} x_i$ significantly better than with probability $1/2$, where $x = x_1 \cdots x_n$ is uniformly distributed in $\{0,1\}^n$.

2.5.2 A simple construction

Intuitively, the definition of a hard-core predicate implies a potentially interesting case of computational indistinguishability. Specifically, as will be shown in Proposition 2.12, if b is a hard-core of the function f, then the ensemble $\{f(U_n)\cdot b(U_n)\}_{n\in\mathbb{N}}$ is computationally indistinguishable from the ensemble $\{f(U_n)\cdot U_1'\}_{n\in\mathbb{N}}$. Furthermore, if f is 1-1 then the foregoing ensembles are statistically far apart, and thus constitute a non-trivial case of computational indistinguishability. If f is also polynomial-time computable and length-preserving, then this yields a construction of a pseudorandom generator.

Proposition 2.12 (A simple construction of pseudorandom generators): *Let b be a hard-core predicate of a polynomial-time computable 1-1 and length-preserving function f. Then, $G(s) \stackrel{\rm def}{=} f(s)\cdot b(s)$ is a pseudorandom generator.*

Proof Sketch: Considering a uniformly distributed $s \in \{0,1\}^n$, we first note that the n-bit long prefix of $G(s)$ is uniformly distributed in $\{0,1\}^n$, because f induces a permutation on the set $\{0,1\}^n$. Hence, the proof boils down to showing that distinguishing $f(s)\cdot b(s)$ from $f(s)\cdot\sigma$, where σ is a random bit, yields contradiction to the hypothesis that b is a hard-core of f (i.e., that $b(s)$ is *unpredictable* from $f(s)$). Intuitively, the reason is that such a hypothetical distinguisher also distinguishes $f(s)\cdot b(s)$ from $f(s)\cdot\overline{b(s)}$, where $\overline{\sigma} = 1-\sigma$, whereas distinguishing $f(s)\cdot b(s)$ from $f(s)\cdot\overline{b(s)}$ yields an algorithm for predicting $b(s)$ based on $f(s)$. For further details see [19, Sec. 8.2.5.1] (or [17, Sec. 3.3.4]). □

Combining Theorem 2.11, Proposition 2.12 and Construction 2.7, we obtain the following corollary.

Theorem 2.13 (A sufficient condition for the existence of pseudorandom generators): *If there exists 1-1 and length-preserving one-way function then, for every polynomially bounded stretch function ℓ, there exists a pseudorandom generator of stretch ℓ.*

Digest. The main part of the proof of Proposition 2.12 is showing that the (next bit) unpredictability of $G(U_k)$ implies the pseudorandomness of $G(U_k)$. The fact that (next bit) unpredictability and pseudorandomness are equivalent, in general, is proven explicitly in the alternative proof of Theorem 2.13 provided next.

2.5.3 An alternative presentation

Let us take a closer look at the pseudorandom generators obtained by combining Construction 2.7 and Proposition 2.12. For a stretch function $\ell:\mathbb{N}\to\mathbb{N}$, a 1-1 one-way function f with a hard-core b, we obtain

$$G(s) \stackrel{\text{def}}{=} \sigma_1\sigma_2\cdots\sigma_{\ell(|s|)}\,, \tag{2.8}$$

where $x_0 = s$ and $x_i\sigma_i = f(x_{i-1})b(x_{i-1})$ for $i = 1, ..., \ell(|s|)$. Denoting by $f^i(x)$ the value of f iterated i times on x (i.e., $f^i(x) = f^{i-1}(f(x))$ and $f^0(x) = x$), we rewrite Eq. (2.8) as follows

$$G(s) \stackrel{\text{def}}{=} b(s)\cdot b(f(s))\cdots b(f^{\ell(|s|)-1}(s))\,. \tag{2.9}$$

The pseudorandomness of G is established in two steps, using the notion of (next bit) unpredictability. An ensemble $\{Z_k\}_{k\in\mathbb{N}}$ is called unpredictable if any probabilistic polynomial-time machine obtaining a (random)[6] prefix of Z_k fails to predict the next bit of Z_k with probability non-negligibly higher than $1/2$. Specifically, we establish the following two results.

1. A **general result** asserting that *an ensemble is pseudorandom if and only if it is unpredictable.* Recall that an ensemble is pseudorandom if it is computationally indistinguishable from a uniform distribution (over bit strings of adequate length).

   Clearly, pseudorandomness implies polynomial-time unpredictability, but here we actually need the other direction, which is less obvious. Still, using a hybrid argument, one can show that (next-bit) unpredictability implies indistinguishability from the uniform ensemble. (Hint: The i^{th} hybrid consists of the i-bit long prefix of the distribution at hand augmented by an adequate number of totally random bits.)

2. A **specific result** asserting that the ensemble $\{G(U_k)\}_{k\in\mathbb{N}}$ is unpredictable *from right to left.* Equivalently, $G'(U_n)$ is polynomial-time unpredictable (from left to right (as usual)), where $G'(s) = b(f^{\ell(|s|)-1}(s))\cdots b(f(s))\cdot b(s)$ is the reverse of $G(s)$.

   Using the fact that f induces a permutation over $\{0,1\}^n$, observe that the $(j+1)$-bit long prefix of $G'(U_k)$ is distributed identically to $b(f^j(U_k))\cdots b(f(U_k))\cdot b(U_k)$. Thus, an algorithm that predicts the $j+1^{\text{st}}$ bit of $G'(U_n)$ based on the j-bit long prefix of $G'(U_n)$ yields an algorithm that guesses $b(U_n)$ based on $f(U_n)$.

Needless to say, G is a pseudorandom generator if and only if G' is a pseudorandom generator. We mention that Eq. (2.9) is often referred to as the Blum-Micali Construction.[7]

[6] For simplicity, we define unpredictability as referring to prefixes of a random length (distributed uniformly in $\{0, ..., |Z_k|-1\}$). A more general definition allows the predictor to determine the length of the prefix that it reads on the fly. This seemingly stronger notion of unpredictability is actually equivalent to the one we use, because both notions are equivalent to pseudorandomness.

[7] Given the popularity of the term, we deviate from our convention of not specifying credits in the main text. Indeed, this construction originates in [9].

2.5.4 A necessary and sufficient condition

Recall that given any one-way 1-1 length-preserving function, we can easily construct a pseudorandom generator. Actually, the 1-1 (and length-preserving) requirement may be dropped, but the currently known construction – for the general case – is quite complex.

Theorem 2.14 (On the existence of pseudorandom generators): *Pseudorandom generators exist if and only if one-way functions exist.*

To show that the existence of pseudorandom generators imply the existence of one-way functions, consider a pseudorandom generator G with stretch function $\ell(k) = 2k$. For $x, y \in \{0,1\}^k$, define $f(x,y) \stackrel{\text{def}}{=} G(x)$, and so f is polynomial-time computable (and length-preserving). It must be that f is one-way, or else one can distinguish $G(U_k)$ from U_{2k} by trying to invert f and checking the result: inverting f on the distribution $f(U_{2k})$ corresponds to operating on the distribution $G(U_k)$, whereas the probability that U_{2k} has an inverse under f is negligible.

The interesting direction of the proof of Theorem 2.14 is the construction of pseudorandom generators based on any one-way function. Since the known proof is quite complex, we only provide a very rough overview of some of the ideas involved. We mention that these ideas make extensive use of adequate hashing functions.

We first note that, in general (when f may not be 1-1), the ensemble $f(U_k)$ may not be pseudorandom, and so Construction 2.12 (i.e., $G(s) = f(s)b(s)$, where b is a hard-core of f) cannot be used *directly*. One idea underlying the known construction is hashing $f(U_k)$ to an almost uniform string of length related to its entropy.[8] But "hashing $f(U_k)$ down to length comparable to the entropy" means shrinking the length of the output to, say, $k' < k$. This foils the entire point of stretching the k-bit seed. Thus, a second idea underlying the construction is compensating for the loss of $k - k'$ bits by extracting these many bits from the seed U_k itself. This is done by hashing U_k, and the point is that the $(k - k')$-bit long hash value does not make the inverting task any easier. Implementing these ideas turns out to be more difficult than it seems, and indeed an alternative construction would be most appreciated.

2.6 Non-uniformly strong pseudorandom generators

Recall that we said that truly random sequences can be replaced by pseudorandom sequences without affecting any efficient computation that uses these sequences. The specific formulation of this assertion, presented in Proposition 2.3, refers to randomized algorithms that take a "primary input" and use a secondary "random

[8]This is done after guaranteeing that the logarithm of the probability mass of a value of $f(U_k)$ is typically close to the entropy of $f(U_k)$. Specifically, given an arbitrary one-way function f', one first constructs f by taking a "direct product" of sufficiently many copies of f'. For example, for $x_1, ..., x_{k^{2/3}} \in \{0,1\}^{k^{1/3}}$, we let $f(x_1, ..., x_{k^{2/3}}) \stackrel{\text{def}}{=} f'(x_1), ..., f'(x_{k^{2/3}})$.

input" in their computation. Proposition 2.3 asserts that it is infeasible to find a primary input for which the replacement of a truly random secondary input by a pseudorandom one affects the final output of the algorithm in a noticeable way. This, however, does not mean that such primary inputs do not exist (but rather that they are hard to find). Consequently, Proposition 2.3 falls short of yielding a (worst-case)[9] "derandomization" of a complexity class such as $\mathcal{BPP}$. To obtain such results, we need a stronger notion of pseudorandom generators, presented next. Specifically, we need pseudorandom generators that can fool all polynomial-size circuits, and not merely all probabilistic polynomial-time algorithms.[10]

Definition 2.15 (strong pseudorandom generator – fooling circuits): *A deterministic polynomial-time algorithm G is called a* non-uniformly strong pseudorandom generator *if there exists a* stretch function, *$\ell : \mathbb{N} \to \mathbb{N}$, such that for any family $\{C_k\}_{k\in\mathbb{N}}$ of polynomial-size circuits, for any positive polynomial p, and for all sufficiently large k's*

$$|\,\mathsf{Pr}[C_k(G(U_k)) = 1] \; - \; \mathsf{Pr}[C_k(U_{\ell(k)}) = 1]\,| \;\; < \;\; \frac{1}{p(k)}$$

Using such pseudorandom generators, we can "derandomize" $\mathcal{BPP}$.

Theorem 2.16 (derandomization of $\mathcal{BPP}$): *If there exists non-uniformly strong pseudorandom generators then $\mathcal{BPP}$ is contained in $\bigcap_{\varepsilon>0} \mathrm{DTIME}(t_\varepsilon)$, where $t_\varepsilon(n) \stackrel{\rm def}{=} 2^{n^\varepsilon}$.*

Proof Sketch: For any $S \in \mathcal{BPP}$ and any $\varepsilon > 0$, we let A denote a probabilistic polynomial-time decision procedure for S and G denote a non-uniformly strong pseudorandom generator stretching n^ε-bit long seeds into poly(n)-long sequences (to be used by A as secondary input when processing a primary input of length n). Combining A and G, we obtain an algorithm $A' = A_G$ (as in Construction 2.2). We claim that A *and* A' *may significantly differ in their* (expected probabilistic) *decision on at most finitely many inputs*, because otherwise we can use these inputs (together with A) to derive a (non-uniform) family of polynomial-size circuits that distinguishes $G(U_{n^\varepsilon})$ and $U_{\mathrm{poly}(n)}$, contradicting the the hypothesis regarding G. Specifically, an input x on which A and A' differ significantly yields a circuit C_x that

[9]Indeed, Proposition 2.3 yields an *average-case derandomization of* $\mathcal{BPP}$. In particular, for every polynomial-time constructible ensemble $\{X_n\}_{n\in\mathbb{N}}$, every Boolean function $f \in \mathcal{BPP}$, and every $\varepsilon > 0$, there exists a randomized algorithm A' of randomness complexity $r_\varepsilon(n) = n^\varepsilon$ such that the probability that $A'(X_n) \neq f(X_n)$ is negligible. A corresponding deterministic ($\exp(r_\varepsilon)$-time) algorithm A'' can be obtained, as in the proof of Theorem 2.16, and again the probability that $A''(X_n) \neq f(X_n)$ is negligible, where here the probability is taken only over the distribution of the primary input (represented by X_n). In contrast, worst-case derandomization, as captured by the assertion $\mathcal{BPP} \subseteq \mathrm{DTIME}(2^{r_\varepsilon})$, requires that the probability that $A''(X_n) \neq f(X_n)$ is zero.

[10]Needless to say, strong pseudorandom generators in the sense of Definition 2.15 satisfy the basic definition of a pseudorandom generator (i.e., Definition 2.1). We comment that the underlying notion of computational indistinguishability (by circuits) is strictly stronger than Definition 2.4, and that it is invariant under multiple samples (regardless of the constructibility of the underlying ensembles).

distinguishes $G(U_{|x|^\varepsilon})$ and $U_{\mathrm{poly}(|x|)}$, by letting $C_x(r) = A(x,r)$.[11] Incorporating the finitely many "bad" inputs into A', we derive a probabilistic polynomial-time algorithm that decides S while using randomness complexity n^ε.

Finally, emulating A' on each of the 2^{n^ε} possible random sequences (i.e., seeds to G) and ruling by majority, we obtain a deterministic algorithm A'' as required. That is, let $A'(x,r)$ denote the output of algorithm A' on input x when using coins $r \in \{0,1\}^{n^\varepsilon}$. Then $A''(x)$ invokes $A'(x,r)$ on every $r \in \{0,1\}^{n^\varepsilon}$, and outputs 1 if and only if the majority of these 2^{n^ε} invocations have returned 1. $\square$

We comment that stronger results regarding derandomization of $\mathcal{BPP}$ are presented in Section 3.

On constructing non-uniformly strong pseudorandom generators. Non-uniformly strong pseudorandom generators (as in Definition 2.15) can be constructed using any one-way function that is hard to invert by any non-uniform family of polynomial-size circuits, rather than by probabilistic polynomial-time machines. In fact, the construction in this case is simpler than the one employed in the uniform case (i.e., the construction underlying the proof of Theorem 2.14).

2.7 Stronger (Uniform-Complexity) Notions

The following two notions represent strengthening of the standard definition of pseudorandom generators (as presented in Definition 2.1). Non-uniform versions of these notions (strengthening Definition 2.15) are also of interest.

2.7.1 Fooling stronger distinguishers

One strengthening of Definition 2.1 amounts to explicitly quantifying the resources (and success gaps) of distinguishers. We choose to bound these quantities as a function of the length of the seed (i.e., k), rather than as a function of the length of the string that is being examined (i.e., $\ell(k)$). For a class of time bounds $\mathcal{T}$ (e.g., $\mathcal{T} = \{t(k) \stackrel{\mathrm{def}}{=} 2^{c\sqrt{k}}\}_{c\in\mathbb{N}}$) and a class of noticeable functions (e.g., $\mathcal{F} = \{f(k) \stackrel{\mathrm{def}}{=} 1/t(k) : t \in \mathcal{T}\}$), we say that a pseudorandom generator, G, is $(\mathcal{T},\mathcal{F})$-strong if for any probabilistic algorithm D having running-time bounded by a function in $\mathcal{T}$ (applied to k)[12], for any function f in $\mathcal{F}$, and for all sufficiently large k's, it holds that

$$|\,\mathsf{Pr}[D(G(U_k)) = 1] - \mathsf{Pr}[D(U_{\ell(k)}) = 1]\,| \;<\; f(k).$$

An analogous strengthening may be applied to the definition of one-way functions. Doing so reveals the weakness of the known construction that underlies the proof of Theorem 2.14: It only implies that for some $\varepsilon > 0$ ($\varepsilon = 1/8$ will do), for any

[11] Indeed, in terms of the proof of Proposition 2.3, the finder F consists of a non-uniform family of polynomial-size circuits that print the "problematic" primary inputs that are hard-wired in them, and the corresponding distinguisher D is thus also non-uniform.

[12] That is, when examining a sequence of length $\ell(k)$ algorithm D makes at most $t(k)$ steps, where $t \in \mathcal{T}$.

$\mathcal{T}$ and $\mathcal{F}$, the existence of "$(\mathcal{T},\mathcal{F})$-strong one-way functions" implies the existence of $(\mathcal{T}',\mathcal{F}')$-strong pseudorandom generators, where $\mathcal{T}' = \{t'(k) \stackrel{\rm def}{=} t(k^\varepsilon)/\mathrm{poly}(k) : t \in \mathcal{T}\}$ and $\mathcal{F}' = \{f'(k) \stackrel{\rm def}{=} \mathrm{poly}(k) \cdot f(k^\varepsilon) : f \in \mathcal{F}\}$. What we *would like* to have is an analogous result with $\mathcal{T}' = \{t'(k) \stackrel{\rm def}{=} t(\Omega(k))/\mathrm{poly}(k) : t \in \mathcal{T}\}$ and $\mathcal{F}' = \{f'(k) \stackrel{\rm def}{=} \mathrm{poly}(k) \cdot f(\Omega(k)) : f \in \mathcal{F}\}$.

2.7.2 Pseudorandom Functions

Recall that pseudorandom *generators* provide a way to efficiently generate long pseudorandom sequences from short random seeds. Pseudorandom functions are even more powerful: they provide efficient direct access to the bits of a huge pseudorandom sequence (which is not feasible to scan bit-by-bit). More precisely, a pseudorandom function is an efficient (deterministic) algorithm that given an k-bit *seed*, s, and an k-bit *argument*, x, returns an k-bit string, denoted $f_s(x)$, such that it is infeasible to distinguish the values of f_s, for a uniformly chosen $s \in \{0,1\}^k$, from the values of a truly random function $F : \{0,1\}^k \rightarrow \{0,1\}^k$. That is, the (feasible) testing procedure is given oracle access to the function (but not its explicit description), and cannot distinguish the case it is given oracle access to a pseudorandom function from the case it is given oracle access to a truly random function.

Definition 2.17 (pseudorandom functions): *A* pseudorandom function (ensemble), *is a collection of functions* $\{f_s : \{0,1\}^{|s|} \rightarrow \{0,1\}^{|s|}\}_{s \in \{0,1\}^*}$ *that satisfies the following two conditions:*

1. (efficient evaluation) *There exists an efficient* (deterministic) *algorithm that given a* seed*, s, and an* argument*, $x \in \{0,1\}^{|s|}$, returns $f_s(x)$.*
2. (pseudorandomness) *For every probabilistic polynomial-time oracle machine, M, every positive polynomial p and all sufficiently large k's*

$$\left|\, \mathsf{Pr}[M^{f_{U_k}}(1^k) = 1] - \mathsf{Pr}[M^{F_k}(1^k) = 1] \,\right| < \frac{1}{p(k)}$$

where F_k denotes a uniformly selected function mapping $\{0,1\}^k$ to $\{0,1\}^k$.

One key feature of pseudorandom functions is that they can be generated and shared by merely generating and sharing their seed; that is, a "random looking" function $f_s : \{0,1\}^k \rightarrow \{0,1\}^k$, is determined by its k-bit seed s. Thus, parties wishing to share a "random looking" function f_s (determining 2^k-many values), merely need to generate and share among themselves the k-bit seed s. (For example, one party may randomly select the seed s, and communicate it, via a secure channel, to all other parties.) Sharing a pseudorandom function allows parties to determine (by themselves and without any further communication) random-looking values depending on their current views of the environment (which need not be known a priori). To appreciate the potential of this tool, one should realize that sharing a pseudorandom function is essentially as good as being able to agree, on the fly, on the association of random values to (on-line) given values, where the latter are taken from a huge set of possible values. We stress that this agreement is achieved without communication and synchronization: Whenever some party needs to associate a random value to a given value, $v \in \{0,1\}^k$, it will associate to v the (same) random value $r_v \in \{0,1\}^k$ (by setting $r_v = f_s(v)$, where f_s is a pseudorandom function agreed upon beforehand). Concretely, the foregoing idea underlies the construction of secure private-key encryption and message-authentication schemes based on pseudorandom functions (cf. [18, Sec. 5.3.3&6.3.1]). In addition to numerous applications in cryptography, pseudorandom functions were also used to derive negative results in computational learning theory [56] and in the study of circuit complexity (cf., Natural Proofs [46]).

Theorem 2.18 (How to construct pseudorandom functions): *Pseudorandom functions can be constructed using any pseudorandom generator.*

Proof Sketch:[13] Let G be a pseudorandom generator that stretches its seed by a factor of two (i.e., $\ell(k) = 2k$), and let $G_0(s)$ (resp., $G_1(s)$) denote the first (resp., last) $|s|$ bits in $G(s)$. Let

$$G_{\sigma_{|s|}\cdots\sigma_2\sigma_1}(s) \stackrel{\text{def}}{=} G_{\sigma_{|s|}}(\cdots G_{\sigma_2}(G_{\sigma_1}(s))\cdots),$$

define $f_s(x_1x_2\cdots x_k) \stackrel{\text{def}}{=} G_{x_k\cdots x_2x_1}(s)$, and consider the function ensemble $\{f_s : \{0,1\}^{|s|} \to \{0,1\}^{|s|}\}_{s\in\{0,1\}^*}$. Pictorially, the function f_s is defined by k-step walks down a full binary tree of depth k having labels at the vertices. The root of the tree, hereafter referred to as the level 0 vertex of the tree, is labeled by the string s. If an internal vertex is labeled r then its left child is labeled $G_0(r)$ whereas its right child is labeled $G_1(r)$. The value of $f_s(x)$ is the string residing in the leaf reachable from the root by a path corresponding to the string x.

We claim that the function ensemble $\{f_s\}_{s\in\{0,1\}^*}$ is pseudorandom. The proof uses the hybrid technique (cf. Section 2.3): The i^{th} hybrid, denoted H_k^i, is a function ensemble consisting of $2^{2^i\cdot k}$ functions $\{0,1\}^k \to \{0,1\}^k$, each determined by 2^i random k-bit strings, denoted $\overline{s} = \langle s_\beta\rangle_{\beta\in\{0,1\}^i}$. The value of such function $h_{\overline{s}}$ at $x = \alpha\beta$, where $|\beta| = i$, is defined to equal $G_\alpha(s_\beta)$. Pictorially, the function $h_{\overline{s}}$ is defined by placing the strings in $\overline{s}$ in the corresponding vertices of level i, and labeling vertices of lower levels using the very rule used in the definition of f_s. The extreme hybrids correspond to our indistinguishability claim (i.e., $H_k^0 \equiv f_{U_k}$ and H_k^k is a truly random function), and the indistinguishability of neighboring hybrids follows from our indistinguishability hypothesis. Specifically, we show that the ability to distinguish H_k^i from H_k^{i+1} yields an ability to distinguish multiple samples of $G(U_k)$ from multiple samples of U_{2k} (by placing on the fly, halves of the given samples at adequate vertices of the $i+1^{\text{st}}$ level). □

Variants. Useful variants (and generalizations) of the notion of pseudorandom functions include Boolean pseudorandom functions that are defined over all strings

[13] See details in [17, Sec. 3.6.2].

(i.e., $f_s : \{0,1\}^* \to \{0,1\}$) and pseudorandom functions that are defined for other domains and ranges (i.e., $f_s : \{0,1\}^{d(|s|)} \to \{0,1\}^{r(|s|)}$, for arbitrary polynomially bounded functions $d, r : \mathbb{N} \to \mathbb{N}$). Various transformations between these variants are known (cf. [17, Sec. 3.6.4] and [18, Apdx. C.2]).

2.8 Conceptual reflections

We highlight several conceptual aspects of the foregoing computational approach to randomness. Some of these aspects are common to other instantiation of the general paradigm (esp., the one presented in Chapter 3).

Behavioristic versus Ontological. The behavioristic nature of the computational approach to randomness is best demonstrated by confronting this approach with the Kolmogorov-Chaitin approach to randomness. Loosely speaking, a string is *Kolmogorov-random* if its length equals the length of the shortest program producing it. This shortest program may be considered the "true explanation" to the phenomenon described by the string. A Kolmogorov-random string is thus a string that does not have a substantially simpler (i.e., shorter) explanation than itself. Considering the simplest explanation of a phenomenon may be viewed as an ontological approach. In contrast, considering the effect of phenomena on certain devices (or observations), as underlying the definition of pseudorandomness, is a behavioristic approach. Furthermore, there exist probability distributions that are not uniform (and are not even statistically close to a uniform distribution) and nevertheless are indistinguishable from a uniform distribution (by any efficient device). Thus, *distributions that are ontologically very different, are considered equivalent by the behavioristic point of view taken in the definition of computational indistinguishability.*

A relativistic view of randomness. We have defined pseudorandomness in terms of its observer. Specifically, we have considered the class of efficient (i.e., polynomial-time) observers and defined as pseudorandom objects that look random to any observer in that class. In subsequent chapters, we shall consider restricted classes of such observers (e.g., space-bounded polynomial-time observers and even very restricted observers that merely apply specific tests such as linear tests or hitting tests). Each such class of observers gives rise to a different notion of pseudorandomness. Furthermore, the general paradigm (of pseudorandomness) explicitly aims at *distributions that are not uniform and yet are considered as such from the point of view of certain observers.* Thus, our entire approach to pseudorandomness is relativistic and subjective (i.e., depending on the abilities of the observer).

Randomness and Computational Difficulty. Pseudorandomness and computational difficulty play dual roles: The general paradigm of pseudorandomness relies on the fact that *placing computational restrictions on the observer gives rise* *to distributions that are not uniform and still cannot be distinguished from uniform distributions.* Thus, the pivot of the entire approach is the computational difficulty of distinguishing pseudorandom distributions from truly random ones. Furthermore, many of the constructions of pseudorandom generators rely either on conjectures or on facts regarding computational difficulty (i.e., that certain computations that are hard for certain classes). For example, one-way functions were used to construct general-purpose pseudorandom generators (i.e., those working in polynomial-time and fooling all polynomial-time observers). Analogously, as we shall see in Sec. 3.2.3, the fact that parity function is hard for polynomial-size constant-depth circuits can be used to generate (highly non-uniform) sequences that fool such circuits.

Randomness and Predictability. The connection between pseudorandomness and unpredictability (by efficient procedures) plays an important role in the analysis of several constructions (cf. Sections 2.5 and 3.2). Here, we wish to highlight the intuitive appeal of this connection.

Chapter 3

Derandomization of Time-Complexity Classes

Let us take a second look at the process of derandomization that underlies the proof of Theorem 2.16. First, a pseudorandom generator was used to shrink the randomness-complexity of a BPP-algorithm, and then derandomization was achieved by scanning all possible seeds to this generator. A key observation regarding this process is that there is no point in insisting that the pseudorandom generator runs in time that is polynomial in its seed length. Instead, it suffices to require that the generator runs in time that is exponential in its seed length, because we are incurring such an overhead anyhow due to the scanning of all possible seeds. Furthermore, in this context, the running-time of the generator may be larger than the running time of the algorithm, which means that the generator need only fool distinguishers that take less steps than the generator. These considerations motivate the following definition of canonical derandomizers.

3.1 Defining Canonical Derandomizers

Recall that in order to "derandomize" a probabilistic polynomial-time algorithm A, we first obtain a functionally equivalent algorithm A_G (as in Construction 2.2) that has (significantly) smaller randomness-complexity. Algorithm A_G has to maintain A's input-output behavior on all (but finitely many) inputs. Thus, the set of the relevant distinguishers (considered in the proof of Theorem 2.16) is the set of all possible circuits obtained from A by hard-wiring any of the possible inputs. Such a circuit, denoted C_x, emulates the execution of algorithm A on input x, when using the circuit's input as the algorithm's internal coin tosses (i.e., $C_x(r) = A(x,r)$). Furthermore, the size of C_x is quadratic in the running-time of A on input x, and the length of the input to C_x equals the running-time of A (on input x).[1] Thus,

[1] Indeed, we assume that algorithm A is represented as a Turing machine and refer to the standard emulation of Turing machines by circuits. Thus, the aforementioned circuit C_x has size that is at most quadratic in the running-time of A on input x, which in turn means that C_x has

the size of C_x is quadratic in the length of its own input, and the pseudorandom generator in use (i.e., G) needs to fool each such circuit. Recalling that we may allow the generator to run in exponential-time (i.e., time that is exponential in the length of its own input (i.e., the seed))[2], we arrive at the following definition.

Definition 3.1 (pseudorandom generator for derandomizing $\mathrm{BPTIME}(\cdot)$)[3]: *Let $\ell : \mathbb{N} \to \mathbb{N}$ be a monotonically increasing function. A* canonical derandomizer of stretch *ℓ is a deterministic algorithm G that satisfies the following two conditions.*

1. *On input a k-bit long seed, G makes at most* $\mathrm{poly}(2^k \cdot \ell(k))$ *steps and outputs a string of length $\ell(k)$.*
2. *For every circuit D_k of size $\ell(k)^2$ it holds that*

$$|\Pr[D_k(G(U_k)) = 1] - \Pr[D_k(U_{\ell(k)}) = 1]| < \frac{1}{6}. \tag{3.1}$$

The circuit D_k represents a potential distinguisher, which is given an $\ell(k)$-bit long string (sampled either from $G(U_k)$ or from $U_{\ell(k)}$). When seeking to derandomize an algorithm A of time-complexity t, the aforementioned $\ell(k)$-bit long string represents a possible sequence of coin tosses of A, when invoked on a generic (primary) input of length $n = t^{-1}(\ell(k))$. Thus, for any $x \in \{0,1\}^n$, considering the circuit $D_k(r) = A(x, r)$, where $|r| = t(n) = \ell(k)$, we note that Eq. (3.1) implies that $A_G(x) = A(x, G(U_k))$ *maintains the majority vote of* $A(x) = A(x, U_{\ell(k)})$. On the other hand, the time-complexity of G implies that the straightforward deterministic emulation of $A_G(x)$ takes time $2^k \cdot (\mathrm{poly}(2^k \cdot \ell(k)) + t(n))$, which is upper-bounded by $\mathrm{poly}(2^k \cdot \ell(k)) = \mathrm{poly}(2^{\ell^{-1}(t(n))} \cdot t(n))$. This yields the following (conditional) derandomization result.

Proposition 3.2 *Let $\ell, t : \mathbb{N} \to \mathbb{N}$ be monotonically increasing functions and let $\ell^{-1}(t(n))$ denote the smallest integer k such that $\ell(k) \geq t(n)$. If there exists a canonical derandomizer of stretch ℓ then, for every time-constructible $t : \mathbb{N} \to \mathbb{N}$, it holds that* $\mathrm{BPTIME}(t) \subseteq \mathrm{DTIME}(T)$, *where* $T(n) = \mathrm{poly}(2^{\ell^{-1}(t(n))} \cdot t(n))$.

---

size that is at most quadratic in the length of its own input. (In fact, the circuit size can be made almost-linear in the running-time of A, by using a better emulation [45].) We note that many sources use the fictitious convention by which the circuit size equals the length of its input; this fictitious convention can be justified by considering a (suitably) padded input.

[2]Actually, in Definition 3.1 we allow the generator to run in time $\mathrm{poly}(2^k \ell(k))$, rather than in time $\mathrm{poly}(2^k)$. This is done in order not to trivially rule out generators of super-exponential stretch (i.e., $\ell(k) = 2^{\omega(k)}$). However, the condition in Eq. (3.1) does not allow for super-exponential stretch (or even for $\ell(k) = \omega(2^k)$). Thus, in retrospect, the two formulations are equivalent (because $\mathrm{poly}(2^k \ell(k)) = \mathrm{poly}(2^k)$ for $\ell(k) = 2^{O(k)}$).

[3]Fixing a model of computation, we denote by $\mathrm{BPTIME}(t)$ the class of decision problems that are solvable by a randomized algorithm of time complexity t that has two-sided error $1/3$. Using $1/6$ as the "threshold distinguishing gap" (in Eq. (3.1)) guarantees that if $\Pr[D_k(U_{\ell(k)}) = 1] \geq 2/3$ (resp., $\Pr[D_k(U_{\ell(k)}) = 1] \leq 1/3$) then $\Pr[D_k(G(U_k)) = 1] > 1/2$ (resp., $\Pr[D_k(G(U_k)) = 1] < 1/2$). As we shall see, this suffices for a derandomization of $\mathrm{BPTIME}(t)$ in time T, where $T(n) = \mathrm{poly}(2^{\ell^{-1}(t(n))} \cdot t(n))$ (and we use a seek of length $k = \ell^{-1}(t(n))$).

Proof Sketch: Just mimic the proof of Theorem 2.16, which in turn uses Construction 2.2. (Recall that given any randomized algorithm A and generator G, Construction 2.2 yields an algorithm A_G of randomness-complexity $\ell^{-1} \circ t$ and time-complexity $\text{poly}(2^{\ell^{-1}\circ t}) + t$.)[4] Observe that the complexity of the resulting deterministic procedure is dominated by the $2^k = 2^{\ell^{-1}(t(|x|))}$ invocations of $A_G(x,s) = A(x, G(s))$, where $s \in \{0,1\}^k$, and each of these invocations takes time $\text{poly}(2^{\ell^{-1}(t(|x|))}) + t(|x|)$. Thus, on input an n-bit long string, the deterministic procedure runs in time $\text{poly}(2^{\ell^{-1}(t(n))} \cdot t(n))$. The correctness of this procedure (which takes a majority vote among the 2^k invocations of A_G) follows by combining Eq. (3.1) with the hypothesis that $\Pr[A(x)=1]$ is bounded-away from $1/2$. Specifically, using the hypothesis $|\Pr[A(x)=1] - (1/2)| \geq 1/6$, it follows that the majority vote of $(A_G(x,s))_{s\in\{0,1\}^k}$ equals 1 if and only if $\Pr[A(x)=1] > 1/2$. Indeed, the implication is due to Eq. (3.1), when applied to the circuit $C_x(r) = A(x,r)$ (which has size at most $|r|^2$). $\square$

The goal. In light of Proposition 3.2, we seek canonical derandomizers with stretch that is as large as possible. The stretch cannot be super-exponential (i.e., it must hold that $\ell(k) = O(2^k)$), because there exists a circuit of size $O(2^k \cdot \ell(k))$ that violates Eq. (3.1) whereas for $\ell(k) = \omega(2^k)$ it holds that $O(2^k \cdot \ell(k)) < \ell(k)^2$. Thus, our goal is to construct a canonical derandomizer with stretch $\ell(k) = 2^{\Omega(k)}$. Such a canonical derandomizer will allow for a "full derandomization of $\mathcal{BPP}$":

Theorem 3.3 *If there exists a canonical derandomizer of stretch $\ell(k) = 2^{\Omega(k)}$, then $\mathcal{BPP} = \mathcal{P}$.*

Proof: Using Proposition 3.2, we get $\text{BPTIME}(t) \subseteq \text{DTIME}(T)$, where $T(n) = \text{poly}(2^{\ell^{-1}(t(n))} \cdot t(n)) = \text{poly}(t(n))$. $\blacksquare$

Reflections: Recall that a canonical derandomizer G was defined in a way that allows it to have time-complexity t_G that is larger than the size of the circuits that it fools (i.e., $t_G(k) > \ell(k)^2$ is allowed). Furthermore, $t_G(k) > 2^k$ was also allowed. Thus, if indeed $t_G(k) = 2^{\Omega(k)}$ (as is the case in Section 3.2), then $G(U_k)$ *can be distinguished from* $U_{\ell(k)}$ *in time* $2^k \cdot t_G(k) = \text{poly}(t_G(k))$ by trying all possible seeds.[5] We stress that the latter distinguisher is a uniform algorithm (and it works by invoking G on all possible seeds). In contrast, for a general-purpose pseudorandom generator G (as discussed in Chapter 2) it holds that $t_G(k) = \text{poly}(k)$, while

[4]Actually, given any randomized algorithm A and generator G, Construction 2.2 yields an algorithm A_G that is defined such that $A_G(x,s) = A(x, G'(s))$, where $|s| = \ell^{-1}(t(|x|))$ and $G'(s)$ denotes the $t(|x|)$-bit long prefix of $G(s)$. For simplicity, we shall assume here that $\ell(|s|) = t(|x|)$, and thus use G rather than G'. Note that given n we can find $k = \ell^{-1}(t(n))$ by invoking $G(1^i)$ for $i = 1, ..., k$ (using the fact that $\ell : \mathbb{N} \to \mathbb{N}$ is monotonically increasing). Also note that $\ell(k) = O(2^k)$ must hold (see Footnote 2), and thus we may replace $\text{poly}(2^k \cdot \ell(k))$ by $\text{poly}(2^k)$.

[5]We note that this distinguisher does not contradict the hypothesis that G is a canonical derandomizer, because $t_G(k) > \ell(k)$ definitely holds whereas $\ell(k) \leq 2^k$ typically holds (and so $2^k \cdot t_G(k) > \ell(k)^2$).

for every polynomial p it holds that $G(U_k)$ is indistinguishable from $U_{\ell(k)}$ in time $p(t_G(k))$.

3.2 Constructing Canonical Derandomizers

The fact that canonical derandomizers are allowed to be more complex than the corresponding distinguisher makes *some* of the techniques of Chapter 2 inapplicable in the current context. For example, the stretch function cannot be amplified as in Section 2.4. On the other hand, the techniques developed in the current section are inapplicable to Chapter 2. For example, the pseudorandomness of some canonical derandomizers (i.e., the generators of Construction 3.4) holds even when the potential distinguisher is given the seed itself. This amazing phenomenon capitalizes on the fact that the distinguisher's time-complexity does not allow for running the generator on the given seed.

3.2.1 The construction and its consequences

As in Section 2.5, the construction presented next transforms computational difficulty into pseudorandomness, except that here both computational difficulty and pseudorandomness are of a somewhat different form than in Section 2.5. Specifically, here we use Boolean predicates that are computable in exponential-time but are strongly inapproximable; that is, we assume *the existence of a Boolean predicate and constants $c, \varepsilon > 0$ such that for all but finitely many m, the* (residual) *predicate $f : \{0,1\}^m \to \{0,1\}$ is computable in time 2^{cm} but for any circuit C of size $2^{\varepsilon m}$ it holds that* $\Pr[C(U_m) = f(U_m)] < \frac{1}{2} + 2^{-\varepsilon m}$. (Needless to say, $\varepsilon < c$.) Such predicates exist under the assumption that the class $\mathcal{E}$ (where $\mathcal{E} = \bigcup_{c>0} \mathrm{DTIME}(2^{c \cdot n})$) contains predicates of (almost-everywhere) exponential circuit complexity [26]. With these preliminaries, we turn to the construction of canonical derandomizers with exponential stretch.

Construction 3.4 (The Nisan-Wigderson Construction):[6] *Let $f : \{0,1\}^m \to \{0,1\}$ and $S_1, ..., S_\ell$ be a sequence of m-subsets of $\{1, ..., k\}$. Then, for $s \in \{0,1\}^k$, we let*

$$G(s) \stackrel{\text{def}}{=} f(s_{S_1}) \cdots f(s_{S_\ell}) \tag{3.2}$$

where s_S denotes the projection of s on the bit locations in $S \subseteq \{1, ..., |s|\}$; that is, for $s = \sigma_1 \cdots \sigma_k$ and $S = \{i_1, ..., i_m\}$, we have $s_S = \sigma_{i_1} \cdots \sigma_{i_m}$.

Letting k vary and $\ell, m : \mathbb{N} \to \mathbb{N}$ be functions of k, we wish G to be a canonical derandomizer and $\ell(k) = 2^{\Omega(k)}$. One (obvious) necessary condition for this to happen is that the sets must be distinct, and hence $m(k) = \Omega(k)$; consequently, f must be computable in exponential-time. Furthermore, the sequence of sets $S_1, ..., S_{\ell(k)}$ must be constructible in $\mathrm{poly}(2^k)$ time. Intuitively, the function f should be strongly inapproximable, and furthermore it seems desirable to use a set

[6] Given the popularity of the term, we deviate from our convention of not specifying credits in the main text. This construction originates in [40, 43].

system with relatively small pairwise intersections (because this restricts the overlap among the various inputs to which f is applied). Interestingly, these conditions are essentially sufficient.

Theorem 3.5 (analysis of Construction 3.4): *Let $\alpha, \beta, \gamma, \varepsilon > 0$ be constants satisfying $\varepsilon > (2\alpha/\beta) + \gamma$, and consider the functions $\ell, m, T : \mathbb{N} \to \mathbb{N}$ such that $\ell(k) = 2^{\alpha k}$, $m(k) = \beta k$, and $T(n) = 2^{\varepsilon n}$. Suppose that the following two conditions hold:*

1. *There exists an exponential-time computable function $f : \{0,1\}^* \to \{0,1\}$ such that for every family of T-size circuits $\{C_n\}_{n \in \mathbb{N}}$ and all sufficiently large n it holds that*
$$\Pr[C_n(U_n) \neq f(U_n)] \geq \frac{1}{2} + \frac{1}{T(n)} \tag{3.3}$$
*In this case we say that f is T-*inapproximable*.*

2. *There exists an exponential-time computable function $S : \mathbb{N} \times \mathbb{N} \to 2^{\mathbb{N}}$ such that*
    (a) *For every k and $i \in [\ell(k)]$, it holds that $S(k,i) \subseteq [k]$ and $|S(k,i)| = m(k)$.*
    (b) *For every k and $i \neq j$, it holds that $|S(k,i) \cap S(k,j)| \leq \gamma \cdot m(k)$.*

Then, using G as defined in Construction 3.4 with $S_i = S(k,i)$, yields a canonical derandomizer with stretch ℓ.

Before proving Theorem 3.5 we mention that, for any $\gamma > 0$, a function S as in Condition 2 does exist for some $m(k) = \Omega(k)$ and $\ell(k) = 2^{\Omega(k)}$; see [19, Exer. 8.19]. We also recall that T-inapproximable predicates do exist under the assumption that $\mathcal{E}$ has (almost-everywhere) exponential circuit complexity (see [26] or [19, Sec. 8.2.1]). Thus, combining such functions f and S and invoking Theorem 3.5, we obtain a canonical derandomizer with exponential stretch based on the assumption that $\mathcal{E}$ has (almost-everywhere) exponential circuit complexity. Combining this with Theorem 3.3, we get the first part of the following theorem.

Theorem 3.6 (derandomization of BPP, revisited):

1. *Suppose that $\mathcal{E}$ contains a decision problem that has almost-everywhere exponential circuit complexity* (i.e., there exists a constant $\varepsilon_0 > 0$ such that, for all but finitely many m's, any circuit that correctly decides this problem on $\{0,1\}^m$ has size at least $2^{\varepsilon_0 m}$). *Then, $\mathcal{BPP} = \mathcal{P}$.*

2. *Suppose that, for every polynomial p, the class $\mathcal{E}$ contains a decision problem that has circuit complexity that is almost-everywhere greater than p. Then $\mathcal{BPP}$ is contained in $\bigcap_{\varepsilon > 0} \mathrm{DTIME}(t_\varepsilon)$, where $t_\varepsilon(n) \stackrel{\text{def}}{=} 2^{n^\varepsilon}$.*

Indeed, our focus is on Part 1, and Part 2 is stated for sake of a wider perspective. Both parts are special cases of a more general statement that can be proved by using a generalization of Theorem 3.5 that refers to arbitrary functions $\ell, m, T : \mathbb{N} \rightarrow \mathbb{N}$ (instead of the exponential functions in Theorem 3.5) that satisfy $\ell(k)^2 + \widetilde{O}(\ell(k) \cdot 2^{m'(k)}) < T(m(k))$, where $m'(k)$ replaces $\gamma \cdot m(k)$. We note that Part 2 of Theorem 3.6 supersedes Theorem 2.16. We also mention that, as in the case of general-purpose pseudorandom generators, the hardness hypothesis used in each part of Theorem 3.6 is necessary for the existence of a corresponding canonical derandomizer.

Additional comment. The two parts of Theorem 3.6 exhibit two extreme cases: Part 1 (often referred to as the "high end") assumes an extremely strong circuit lower-bound and yields "full derandomization" (i.e., $\mathcal{BPP} = \mathcal{P}$), whereas Part 2 (often referred to as the "low end") assumes an extremely weak circuit lower-bound and yields weak but meaningful derandomization. Intermediate results (relying on intermediate lower-bound assumptions) can be obtained via the aforementioned generalization, but tight trade-offs are obtained differently (cf., [54]).

3.2.2 Analyzing the construction (i.e., proof of Theorem 3.5)

Using the time complexity upper-bounds on f and S, it follows that G can be computed in exponential time. Thus, our focus is on showing that $\{G(U_k)\}$ cannot be distinguished from $\{U_{\ell(k)}\}$ by circuits of size $\ell(k)^2$; specifically, that G satisfies Eq. (3.1). In fact, we will prove that this holds for $G'(s) = s \cdot G(s)$; that is, G fools such circuits even if they are given the seed as auxiliary input. (Indeed, these circuits are smaller than the running time of G, and so they cannot just evaluate G on the given seed.)

We start by presenting the intuition underlying the proof. As a warm-up suppose that the sets (i.e., $S(k,i)$'s) used in the construction are disjoint. In such a case (which is indeed impossible because $k < \ell(k) \cdot m(k)$), the pseudorandomness of $G(U_k)$ would follow easily from the inapproximability of f, because in this case G consists of applying f to non-overlapping parts of the seed. In the actual construction being analyzed here, the sets (i.e., $S(k,i)$'s) are not disjoint but have relatively small pairwise intersection, which means that G applies f on parts of the seed that have relatively small overlap. Intuitively, such small overlaps guarantee that the values of f on the corresponding inputs are "computationally independent" (i.e., having the value of f at some inputs $x_1, ..., x_i$ does not help in approximating the value of f at another input x_{i+1}). This intuition will be backed by showing that, when fixing all bits that do not appear in the target input (i.e., in x_{i+1}), the former values (i.e., $f(x_1), ..., f(x_i)$) can be computed at a relatively small computational cost. Thus, the values $f(x_1), ..., f(x_i)$ do not (significantly) facilitate the task of approximating $f(x_{i+1})$. With the foregoing intuition in mind, we now turn to the actual proof.

The actual proof employs a reducibility argument; that is, assuming towards the contradiction that G' does not fool some circuit of size $\ell(k)^2$, we derive a contradiction to the hypothesis that the predicate f is T-inapproximable. The argument utilizes the relation between pseudorandomness and unpredictability (cf. Section 2.5). Specifically, *any circuit that distinguishes $G'(U_k)$ from $U_{\ell(k)+k}$ with gap $1/6$, yields a next-bit predictor of similar size that succeeds in predicting the next bit with probability at least $\frac{1}{2}+\frac{1}{6\ell'(k)} > \frac{1}{2}+\frac{1}{7\ell(k)}$*, where the factor of $\ell'(k) = \ell(k)+k < (1+o(1))\cdot\ell(k)$ is introduced by the hybrid technique (cf. Eq. (2.5)). Furthermore, given the non-uniform setting of the current proof, we may fix a bit location $i+1$ for prediction, rather than analyzing the prediction at a random bit location. Indeed, $i \geq k$ must hold, because the first k bits of $G'(U_k)$ are uniformly distributed. In the rest of the proof, we transform the foregoing predictor into a circuit that approximates f better than allowed by the hypothesis (regarding the inapproximability of f).

Assuming that a small circuit C' can predict the $i+1^{\text{st}}$ bit of $G'(U_k)$, when given the previous i bits, we construct a small circuit C for approximating $f(U_{m(k)})$ on input $U_{m(k)}$. The point is that the $i+1^{\text{st}}$ bit of $G'(s)$ equals $f(s_{S(k,j+1)})$, where $j = i-k \geq 0$, and so C' approximates $f(s_{S(k,j+1)})$ based on $s, f(s_{S(k,1)}), ..., f(s_{S(k,j)})$, where $s \in \{0,1\}^k$ is uniformly distributed. Note that this is the type of thing that we are after, except that the circuit we seek may only get $s_{S(k,j+1)}$ as input.

The first observation is that C' maintains its advantage when we fix the best choice for the bits of s that are not at bit locations $S_{j+1} = S(k,j+1)$ (i.e., the bits $s_{[k]\setminus S_{j+1}}$). That is, by an averaging argument, it holds that

$$\begin{aligned} & \max_{s' \in \{0,1\}^{k-m(k)}} \{\mathsf{Pr}_{s\in\{0,1\}^k}[C'(s, f(s_{S_1}), ..., f(s_{S_j})) = f(s_{S_{j+1}}) \,|\, s_{[k]\setminus S_{j+1}} = s']\} \\ \geq \;\; & p' \stackrel{\text{def}}{=} \mathsf{Pr}_{s\in\{0,1\}^k}[C'(s, f(s_{S_1}), ..., f(s_{S_j})) = f(s_{S_{j+1}})]. \end{aligned}$$

Recall that by the hypothesis $p' > \frac{1}{2}+\frac{1}{7\ell(k)}$. Hard-wiring the fixed string s' into C', and letting $\pi(x)$ denote the (unique) string s satisfying $s_{S_{j+1}} = x$ and $s_{[k]\setminus S_{j+1}} = s'$, we obtain a circuit C'' that satisfies

$$\mathsf{Pr}_{x\in\{0,1\}^{m(k)}}[C''(x, f(\pi(x)_{S_1}), ..., f(\pi(x)_{S_j})) = f(x)] \;\geq\; p'.$$

The circuit C'' is almost what we seek. The only problem is that C'' gets as input not only x, but also $f(\pi(x)_{S_1}), ..., f(\pi(x)_{S_j})$, whereas we seek an approximator of $f(x)$ that only gets x.

The key observation is that each of the "missing" values $f(\pi(x)_{S_1}), ..., f(\pi(x)_{S_j})$ depend only on a relatively small number of the bits of x. This fact is due to the hypothesis that $|S_t \cap S_{j+1}| \leq \gamma \cdot m(k)$ for $t = 1, ..., j$, which means that $\pi(x)_{S_t}$ is an $m(k)$-bit long string in which $m_t \stackrel{\text{def}}{=} |S_t \cap S_{j+1}|$ bits are projected from x and the rest are projected from the *fixed* string s'. Thus, given x, the value $f(\pi(x)_{S_t})$ can be computed by a (trivial) circuit of size $\widetilde{O}(2^{m_t})$; that is, by a circuit implementing a look-up table on m_t bits. Using all these circuits (together with C''), we will obtain the desired approximator of f. Details follow.

We obtain the desired circuit, denoted C, that T-approximates f as follows. The circuit C depends on the index j and the string s' that are fixed as in the foregoing analysis. Recall that C incorporates ($\widetilde{O}(2^{\gamma\cdot|x|})$-size) circuits for computing $x \mapsto$

$f(\pi(x)_{S_t})$, for $t = 1, ..., j$. On input $x \in \{0,1\}^{m(k)}$, the circuit C computes the values $f(\pi(x)_{S_1}), ..., f(\pi(x)_{S_j})$, invokes C'' on input x and these values, and outputs the answer as a guess for $f(x)$. That is,

$$C(x) = C''(x, f(\pi(x)_{S_1}), ..., f(\pi(x)_{S_j})) = C'(\pi(x), f(\pi(x)_{S_1}), ..., f(\pi(x)_{S_j})).$$

By the foregoing analysis, $\mathsf{Pr}_x[C(x) = f(x)] \geq p' > \frac{1}{2} + \frac{1}{7\ell(k)}$, which is lower-bounded by $\frac{1}{2} + \frac{1}{T(m(k))}$, because $T(m(k)) = 2^{\varepsilon m(k)} = 2^{\varepsilon \beta k} \gg 2^{2\alpha k} \gg 7\ell(k)$, where the first inequality is due to $\varepsilon > 2\alpha/\beta$ and second inequality is due to $\ell(k) = 2^{\alpha k}$. The size of C is upper-bounded by $\ell(k)^2 + \ell(k) \cdot \widetilde{O}(2^{\gamma \cdot m(k)}) \ll \widetilde{O}(\ell(k)^2 \cdot 2^{\gamma \cdot m(k)}) = \widetilde{O}(2^{2\alpha \cdot (m(k)/\beta) + \gamma \cdot m(k)}) \ll T(m(k))$, where the last inequality is due to $T(m(k)) = 2^{\varepsilon m(k)} \gg \widetilde{O}(2^{(2\alpha/\beta) \cdot m(k) + \gamma \cdot m(k)})$ (which in turn uses $\varepsilon > (2\alpha/\beta) + \gamma$). Thus, we derived a contradiction to the hypothesis that f is T-inapproximable. This completes the proof of Theorem 3.5.

3.2.3 Construction 3.4 as a general framework

The Nisan–Wigderson Construction (i.e., Construction 3.4) is actually a general framework, which can be instantiated in various ways. Some of these instantiations, which are based on an abstraction of the construction as well as of its analysis, are briefly reviewed next.

We first note that the generator described in Construction 3.4 consists of a generic algorithmic scheme that can be instantiated with any predicate f. Furthermore, this algorithmic scheme, denoted G, is actually an *oracle machine* that makes (non-adaptive) queries to the function f, and thus the combination (of G and f) may be written as G^f. Likewise, the proof of pseudorandomness of G^f (i.e., the bulk of the proof of Theorem 3.5) is actually a general scheme that, for every f, yields a (non-uniform) oracle-aided circuit C that approximates f by using an oracle call to any distinguisher for G^f (i.e., C uses the distinguisher as a black-box). The circuit C does depends on f (but in a restricted way). Specifically, C contains look-up tables for computing functions obtained from f by fixing some of the input bits (i.e., look-up tables for the functions $f(\pi(\cdot)_{S_t})$'s). The foregoing abstractions facilitate the presentation of the following instantiations of the general framework underlying Construction 3.4

Derandomization of constant-depth circuits. In this case we instantiate Construction 3.4 using the `parity` function in the role of the inapproximable predicate f, noting that `parity` is indeed inapproximable by "small" constant-depth circuits. With an adequate setting of parameters we obtain pseudorandom generators with stretch $\ell(k) = \exp(k^{1/O(1)})$ that fool "small" constant-depth circuits (see [40]). The analysis of this construction proceeds very much like the proof of Theorem 3.5. One important observation is that incorporating the (straightforward) circuits that compute $f(\pi(x)_{S_t})$ into the distinguishing circuit only increases its depth by two levels. Specifically, the circuit C uses depth-two circuits that compute the values $f(\pi(x)_{S_t})$'s, and then obtains a prediction of $f(x)$ by using these values in its (single) invocation of the (given) distinguisher.

The resulting pseudorandom generator, which use a seed of polylogarithmic length (equiv., $\ell(k) = \exp(k^{1/O(1)})$), can be used for derandomizing $\mathcal{RAC}^0$ (i.e., random $\mathcal{AC}^0$), analogously to Theorem 3.3. Thus, we can *deterministically* approximate, in quasi-polynomial-time and up-to an additive error, the fraction of inputs that satisfy a given (constant-depth) circuit. Specifically, for any constant d, given a depth-d circuit C, we can deterministically approximate the fraction of the inputs that satisfy C (i.e., cause C to evaluate to 1) to within any *additive constant error*[7] in time $\exp((\log |C|)^{O(d)})$. Providing a deterministic polynomial-time approximation, even in the case $d = 2$ (i.e., CNF/DNF formulae) is an open problem.

Derandomization of probabilistic proof systems. A different (and more surprising) instantiation of Construction 3.4 utilizes predicates that are inapproximable by small *circuits having oracle access to* $\mathcal{NP}$. The result is a pseudorandom generator robust against two-move public-coin interactive proofs (which are as powerful as constant-round interactive proofs). The key observation is that the analysis of Construction 3.4 provides a black-box procedure for approximating the underlying predicate when given oracle access to a distinguisher (and this procedure is valid also in case the distinguisher is a non-deterministic machine). Thus, under suitably strong (and yet plausible) assumptions, constant-round interactive proofs collapse to $\mathcal{NP}$. We note that a stronger result, which deviates from the foregoing framework, has been subsequently obtained (cf. [37]).

Construction of randomness extractors. An even more radical instantiation of Construction 3.4 was used to obtain explicit constructions of randomness extractors (see [50]). In this case, the predicate f is viewed as (an error correcting encoding of) a somewhat random function, and the construction makes sense because it refers to f in a black-box manner. In the analysis we rely on the fact that f can be approximated by combining relatively little information (regarding f) with (black-box access to) a distinguisher for G^f. For further details see either [53] or [50] (or [19, Apdx. D.4]).

3.3 Reflections Regarding Derandomization

Part 1 of Theorem 3.6 is often summarized by saying that (under some reasonable assumptions) *randomness is useless*. We believe that this interpretation is wrong even within the restricted context of traditional complexity classes, and is bluntly wrong if taken outside of the latter context. Let us elaborate.

[7] We mention that in the special case of approximating the number of satisfying assignment of a DNF formula, *relative error* approximations can be obtained by employing a deterministic reduction of relative error approximation to additive constant error approximation (see [16, Apdx. B.1.1] or [19, §6.2.2.1]). Thus, using a pseudorandom generator that fools DNF formulae, we can deterministically obtain a relative (rather than additive) error approximation to the number of satisfying assignment in a given DNF formula.

Taking a closer look at the proof of Theorem 3.3 (which underlies Theorem 3.6), we note that a randomized algorithm A of time-complexity t is emulated by a deterministic algorithm A' of time complexity $t' = \text{poly}(t)$. Further noting that $A' = A_G$ invokes A (as well as the canonical derandomizer G) for $\Omega(t)$ times (because $\ell(k) = O(2^k)$ implies $2^k = \Omega(t)$), we infer that $t' = \Omega(t^2)$ must hold. Thus, derandomization via (Part 1 of) Theorem 3.6 is not really for free.

More importantly, we note that derandomization is not possible in various distributed settings, when both parties may protect their conflicting interests by employing randomization. Notable examples include most cryptographic primitives (e.g., encryption) as well as most types of probabilistic proof systems (e.g., PCP). Additional settings where randomness makes a difference (either between impossibility and possibility or between formidable and affordable cost) include distributed computing (see [6]), communication complexity (see [31]), parallel architectures (see [32]), sampling (see, e.g., [19, Apdx. D.3]), and property testing (see, e.g., [19, Sec. 10.1.2]).

Chapter 4

Space-Bounded Distinguishers

In the previous two chapters we have considered generators that output sequences that look random to any efficient procedures, where the latter were modeled by time-bounded computations. Specifically, in Chapter 2 we considered indistinguishability by polynomial-time procedures. A finer classification of time-bounded procedures is obtained by considering their *space-complexity*; that is, restricting the space-complexity of time-bounded computations. This restriction leads to the notion of pseudorandom generators that fool space-bounded distinguishers. Interestingly, in contrast to the notions of pseudorandom generators that were considered in Chapters 2 and 3, the existence of pseudorandom generators that fool space-bounded distinguishers can be established without relying on computational assumptions.

Prerequisites: Technically speaking, the current chapter is self-contained, but various definitional choices are justified by reference to the standard definitions of space-bounded randomized algorithms. Thus, a review of that model (as provided in, e.g., [19, Sec. 6.1.5]) is recommended as conceptual background for the current chapter.

4.1 Definitional Issues

Our main motivation for considering space-bounded distinguishers is to develop a notion of pseudorandomness that is adequate for space-bounded randomized algorithms. That is, such algorithms should essentially maintain their behavior when their source of internal coin tosses is replaced by a source of pseudorandom bits (which may be generated based on a much shorter random seed). We thus start by recalling and reviewing the natural notion of space-bounded randomized algorithms.

Unfortunately, natural notions of space-bounded computations are quite subtle, especially when non-determinism or randomization are concerned (see [19, Sec. 5.3] and [19, Sec. 6.1.5], respectively). Two major definitional issues regarding randomized space-bounded computations are the need for imposing explicit *time bounds* and the type of *access to the random tape*.

1. Time bounds: The question is whether or not the space-bounded machines are restricted to time-complexity that is at most exponential in their space-complexity.[1] Recall that such an upper-bound follows automatically in the deterministic case, and can be assumed without loss of generality in the non-deterministic case, *but it does not necessarily hold in the randomized case*. Furthermore, failing to restrict the time-complexity of randomized space-bounded machines makes them unnatural and unintentionally too strong (e.g., capable of emulating non-deterministic computations with no overhead in term of space-complexity).

   Seeking a natural model of randomized space-bounded algorithms, we postulate that their time-complexity must be at most exponential in their space-complexity.

2. Access to the random tape: Recall that randomized algorithms may be modeled as machines that are provided with the necessary randomness via a special random-tape. The question is whether the space-bounded machine has uni-directional or bi-directional (i.e., unrestricted) access to its random-tape. (Allowing bi-directional access means that the randomness is recorded "for free"; that is, without being accounted for in the space-bound.)

   Recall that uni-directional access to the random-tape corresponds to the natural model of an on-line randomized machine, which determines its moves based on its internal coin tosses (and thus cannot record its past coin tosses "for free"). Thus, we consider uni-directional access.[2]

Hence, we focus on randomized space-bounded computation that have time-complexity that is at most exponential in their space-complexity and access their random-tape in a uni-directional manner.

When seeking a notion of pseudorandomness that is adequate for the foregoing notion of randomized space-bounded computations, we note that the corresponding distinguisher is obtained by fixing the main input of the computation and *viewing the contents of the random-tape of the computation as the only input of the distinguisher*. Thus, in accordance with the foregoing notion of randomized space-bounded computation, *we consider space-bounded distinguishers that have a*

[1] Alternatively, one can ask whether these machines must always halt or only halt with probability approaching 1. It can be shown that the only way to ensure "absolute halting" is to have time-complexity that is at most exponential in the space-complexity. (In the current discussion as well as throughout this chapter, we assume that the space-complexity is at least logarithmic.)

[2] We note that the fact that we restrict our attention to uni-directional access is instrumental in obtaining space-robust generators without making intractability assumptions. Analogous generators for bi-directional space-bounded computations would imply hardness results of a breakthrough nature in the area.

uni-directional access to the input sequence that they examine. Let us consider the type of algorithms that arise.

We consider *space-bounded algorithms that have a uni-directional access to their input.* At each step, based on the contents of its temporary storage, such an algorithm may either read the next input bit or stay at the current location on the input, where in either case the algorithm may modify its temporary storage. To simplify our analysis of such algorithms, we consider a corresponding *non-uniform model* in which, at each step, the algorithm reads the next input bit and update its temporary storage according to an arbitrary function applied to the previous contents of that storage (and to the new bit). Note that we have strengthened the model by allowing arbitrary (updating) functions, which can be implemented by (non-uniform) circuits having size that is exponential in the space-bound, rather than using (updating) functions that can be (uniformly) computed in time that is exponential in the space-bound. This strengthening is motivated by the fact that the known constructions of pseudorandom generators remain valid also when the space-bounded distinguishers are non-uniform and by the fact that non-uniform distinguishers arise anyhow in derandomization.

The computation of the foregoing non-uniform space-bounded algorithms (or automata)[3] can be represented by directed layered graphs, where the vertices in each layer correspond to possible contents of the temporary storage and transition between neighboring layers corresponds to a step of the computation. Foreseeing the application of this model for the description of potential distinguishers, we parameterize these layered graphs based on the index, denoted k, of the relevant ensembles (e.g., $\{G(U_k)\}_{k\in\mathbb{N}}$ and $\{U_{\ell(k)}\}_{k\in\mathbb{N}}$). That is, we present both the input length, denoted $\ell = \ell(k)$, and the space-bound, denoted $s(k)$, as functions of the parameter k. Thus, we define a **non-uniform automaton of space** $s:\mathbb{N}\rightarrow\mathbb{N}$ (and depth $\ell:\mathbb{N}\rightarrow\mathbb{N}$) as a family, $\{D_k\}_{k\in\mathbb{N}}$, of directed layered graphs with labeled edges such that the following conditions hold:

- The digraph D_k consists of $\ell(k)+1$ layers, each containing at most $2^{s(k)}$ vertices. The first layer contains a single vertex, which is the digraph's (single) source (i.e., a vertex with no incoming edges), and the last layer contains all the digraph's sinks (i.e., vertices with no outgoing edges).

- The only directed edges in D_k are between adjacent layers, going from layer i to layer $i+1$, for $i \leq \ell(k)$. These edges are labeled such that each (non-sink) vertex of D_k has two (possibly parallel) outgoing directed edges, one labeled 0 and the other labeled 1.

The result of the computation of such an automaton, on an input of adequate length

[3]We use the term automaton (rather than algorithm or machine) in order to remind the reader that this computing device reads its input in a uni-directional manner. Alternative terms that may be used are "real-time" or "on-line" machines. We prefer not using the term "on-line" machine in order to keep a clear distinction from randomized (on-line) algorithms that have free access to their input (and on-line access to a source of randomness). Indeed, the automata consider here arise from the latter algorithms by fixing their primary input and considering the random source as their (only) input. We also note that the automata considered here are a special case of Ordered Binary Decision Diagrams (OBDDs; see [57]).

(i.e., length ℓ where D_k has $\ell+1$ layers), is defined as the vertex (in last layer) reached when following the sequence of edges that are labeled by the corresponding bits of the input. That is, on input $x = x_1 \cdots x_\ell$, in the $i^{\rm th}$ step (for $i = 1, ..., \ell$) we move from the current vertex (which resides in the $i^{\rm th}$ layer) to one of its neighbors (which resides in the $i+1^{\rm st}$ layer) by following the outgoing edge labeled x_i. Using a fixed partition of the vertices of the last layer, this defines a natural notion of a decision (by D_k); that is, we write $D_k(x) = 1$ if on input x the automaton D_k reached a vertex that belongs to the first part of the aforementioned partition.

Definition 4.1 (indistinguishability by space-bounded automata):

- *For a non-uniform automaton, $\{D_k\}_{k\in\mathbb{N}}$, and two probability ensembles, $\{X_k\}_{k\in\mathbb{N}}$ and $\{Y_k\}_{k\in\mathbb{N}}$, the function $d:\mathbb{N}\to[0,1]$ defined as*

$$d(k) \stackrel{\rm def}{=} |\Pr[D_k(X_k) = 1] - \Pr[D_k(Y_k) = 1]|$$

  is called the distinguishability-gap *of $\{D_k\}$ between the two ensembles.*

- *Let $s : \mathbb{N}\to\mathbb{N}$ and $\varepsilon : \mathbb{N} \to [0,1]$. A probability ensemble, $\{X_k\}_{k\in\mathbb{N}}$, is called (s,ε)-*pseudorandom *if for any non-uniform automaton of space $s(\cdot)$, the distinguishability-gap of the automaton between $\{X_k\}_{k\in\mathbb{N}}$ and the corresponding uniform ensemble (i.e., $\{U_{|X_k|}\}_{k\in\mathbb{N}}$) is at most $\varepsilon(\cdot)$.*

- *A deterministic algorithm G of stretch function ℓ is called an (s,ε)-*pseudorandom generator *if the ensemble $\{G(U_k)\}_{k\in\mathbb{N}}$ is (s,ε)-pseudorandom. That is, every non-uniform automaton of space $s(\cdot)$ has a distinguishing-gap of at most $\varepsilon(\cdot)$ between $\{G(U_k)\}_{k\in\mathbb{N}}$ and $\{U_{\ell(k)}\}_{k\in\mathbb{N}}$.*

Thus, when using a random seed of length k, an (s,ε)-pseudorandom generator outputs a sequence of length $\ell(k)$ that looks random to observers having space $s(k)$. Note that $s(k) \leq k$ is a necessary condition for the existence of $(s, 0.5)$-pseudorandom generators, because a non-uniform automaton of space $s(k) > k$ can recognize the image of a generator (which contains at most 2^k strings of length $\ell(k) > k$). More generally, there is a trade-off between $k - s(k)$ and the stretch ℓ of (s,ε)-pseudorandom generators (i.e., $\ell(k) = O(\varepsilon(k) \cdot k \cdot 2^{k-s(k)})$ for $\varepsilon(k) \leq 1/2$).

Note: We stated the space-bound of the potential distinguisher (as well as the stretch function) in terms of the seed-length, denoted k, of the generator. In contrast, other sources present a parameterization in terms of the space-bound of the potential distinguisher, denoted m. The translation is obtained by using $m = s(k)$, and we shall provide it subsequent to the main statements of Theorems 4.2 and 4.3.

4.2 Two Constructions

In contrast to the case of pseudorandom generators that fool time-bounded distinguishers, pseudorandom generators that fool space-bounded distinguishers can be constructed without relying on any computational assumption. The following two theorems exhibit two rather extreme cases of a general trade-off between the space-bound of the potential distinguisher and the stretch function of the generator.[4] We stress that both theorems fall short of providing parameters as achieved by a non-constructive argument, but they refer to relatively efficient constructions. We start with an attempt to maximize the stretch.

Theorem 4.2 (stretch exponential in the space-bound for $s(k) = \sqrt{k}$): *For every space constructible function $s:\mathbb{N}\to\mathbb{N}$, there exists an $(s, 2^{-s})$-pseudorandom generator of stretch function $\ell(k) = \min(2^{k/O(s(k))}, 2^{s(k)})$. Furthermore, the generator works in space that is linear in the length of the seed, and in time that is linear in the stretch function.*

In other words, for every $t \le m$, we have a generator that takes a random seed of length $k = O(t \cdot m)$ and produce a sequence of length 2^t that looks random to any (non-uniform) automaton of space m (up to a distinguishing-gap of 2^{-m}). In particular, using a random seed of length $k = O(m^2)$, one can produce a sequence of length 2^m that looks random to any (non-uniform) automaton of space m. Thus, *one may replace random sequences used by any space-bounded computation, by sequences that are efficiently generated from random seeds of length quadratic in the space bound.* The common instantiation of the latter assertion is for log-space algorithms. In Sec. 4.2.2, we apply Theorem 4.2 (and its underlying ideas) for the derandomization of space-complexity classes such as $\mathcal{BPL}$ (i.e., the log-space analogue of $\mathcal{BPP}$). Theorem 4.2 itself is proved in Sec. 4.2.1.

We now turn to the case where one wishes to maximize the space-bound of potential distinguishers. We warn that Theorem 4.3 only guarantees a subexponential distinguishing gap (rather than the exponential distinguishing gap guaranteed in Theorem 4.2). This warning is voiced because failing to recall this limitation has led to errors in the past.

Theorem 4.3 (polynomial stretch and linear space-bound): *For any polynomial p and for some $s(k) = k/O(1)$, there exists an $(s, 2^{-\sqrt{s}})$-pseudorandom generator of stretch function p. Furthermore, the generator works in linear-space and polynomial-time* (both stated in terms of the length of the seed).

In other words, we have a generator that takes a random seed of length $k = O(m)$ and produce a sequence of length $\text{poly}(m)$ that looks random to any (non-uniform) automaton of space m. Thus, one may *convert any randomized computation utilizing polynomial-time and linear-space into a functionally equivalent randomized computation of similar time and space complexities that uses only a linear number of coin tosses.*

4.2.1 Sketches of the proofs of Theorems 4.2 and 4.3

In both cases, we start the proof by considering a generic space-bounded distinguisher and show that the input distribution that this distinguisher examines can

[4] These two results have been "interpolated" in [5]: There exists a parameterized family of "space fooling" pseudorandom generators that includes both results as extreme special cases.

be modified (from the uniform distribution into a pseudorandom one) without having the distinguisher notice the difference. This modification (or rather a sequence of modifications) yields a construction of a pseudorandom generator, which is only spelled-out at the end of the argument.

Sketch of the proof of Theorem 4.2 (see details in [41])

The main technical tool used in this proof is the "mixing property" of pairwise independent hash functions (see, e.g., [19, Apdx. D.2]). A family of functions H_n, which map $\{0,1\}^n$ to itself, is called mixing if for every pair of subsets $A, B \subseteq \{0,1\}^n$ for all but very few (i.e., $\exp(-\Omega(n))$ fraction) of the functions $h \in H_n$, it holds that

$$\mathsf{Pr}[U_n \in A \wedge h(U_n) \in B] \approx \frac{|A|}{2^n} \cdot \frac{|B|}{2^n} \tag{4.1}$$

where the approximation is up to an additive term of $\exp(-\Omega(n))$.

We may assume, without loss of generality, that $s(k) = \Omega(\sqrt{k})$, and thus $\ell(k) \leq 2^{s(k)}$ holds. For any $s(k)$-space distinguisher D_k as in Definition 4.1, we consider an auxiliary "distinguisher" D'_k that is obtained by "contracting" every block of $n \stackrel{\text{def}}{=} \Theta(s(k))$ consecutive layers in D_k, yielding a directed layered graph with $\ell' \stackrel{\text{def}}{=} \ell(k)/n < 2^{s(k)}$ layers (and $2^{s(k)}$ vertices in each layer). Specifically,

- each vertex in D'_k has 2^n (possibly parallel) directed edges going to various vertices of the next level; and
- each such edge is labeled by an n-bit long string such that the directed edge (u, v) labeled $\sigma_1\sigma_2\cdots\sigma_n$ in D'_k replaces the n-edge directed path between u and v in D_k that consists of edges labeled $\sigma_1, \sigma_2, ..., \sigma_n$.

The graph D'_k simulates D_k in the obvious manner; that is, the computation of D'_k on an input of length $\ell(k) = \ell' \cdot n$ is defined by breaking the input into consecutive substrings of length n and following the path of edges that are labeled by the corresponding n-bit long substrings.

The key observation is that D'_k cannot distinguish between a random $\ell' \cdot n$-bit long input (i.e., $U_{\ell' \cdot n} \equiv U_n^{(1)} U_n^{(2)} \cdots U_n^{(\ell')}$) and a "pseudorandom" input of the form $U_n^{(1)} h(U_n^{(1)}) U_n^{(2)} h(U_n^{(2)}) \cdots U_n^{(\ell'/2)} h(U_n^{(\ell'/2)})$, where $h \in H_n$ is a (suitably fixed) hash function. To prove this claim, we consider an arbitrary pair of neighboring vertices, u and v (in layers i and $i+1$, respectively), and denote by $L_{u,v} \subseteq \{0,1\}^n$ the set of the labels of the edges going from u to v. Similarly, for a vertex w at layer $i+2$, we let $L'_{v,w}$ denote the set of the labels of the edges going from v to w. By Eq. (4.1), for all but very few of the functions $h \in H_n$, it holds that

$$\mathsf{Pr}[U_n \in L_{u,v} \wedge h(U_n) \in L'_{v,w}] \approx \mathsf{Pr}[U_n \in L_{u,v}] \cdot \mathsf{Pr}[U_n \in L'_{v,w}]\,, \tag{4.2}$$

where "very few" and $\approx$ are as in Eq. (4.1). Thus, for all but $\exp(-\Omega(n))$ fraction of the choices of $h \in H_n$, *replacing the coins in the second transition* (i.e., the transition from layer $i+1$ to layer $i+2$) *with the value of* h *applied to the outcomes of the coins used in the first transition* (i.e., the transition from layer i to $i+1$), *approximately maintains the probability that D'_k moves from u to w via v.* Using a union bound (on all triples (u,v,w) as in the foregoing), we note that, for all but $2^{3s(k)} \cdot \ell' \cdot \exp(-\Omega(n))$ fraction of the choices of $h \in H_n$, the foregoing replacement approximately maintains the probability that D'_k moves through any specific two-edge path of D'_k.

Using $\ell' < 2^{s(k)}$ and a suitable choice of $n = \Theta(s(k))$, it holds that $2^{3s(k)} \cdot \ell' \cdot \exp(-\Omega(n)) < \exp(-\Omega(n))$, and thus all but "few" functions $h \in H_n$ are good for approximating all these transition probabilities. (We stress that the same h can be used in all these approximations.) Thus, *at the cost of extra $|h|$ random bits, we can reduce the number of true random coins used in transitions on D'_k by a factor of two, without significantly affecting the final decision of D'_k* (where again we use the fact that $\ell' \cdot \exp(-\Omega(n)) < \exp(-\Omega(n))$, which implies that the approximation errors do not accumulate to too much). In other words, at the cost of extra $|h|$ random bits, we can effectively contract the distinguisher to half its length while approximately maintaining the probability that the distinguisher accepts a random input. That is, fixing a good h (i.e., one that provides a good approximation to the transition probability over all $2^{3s(k)} \cdot \ell'$ two-edge paths), we can replace the two-edge paths in D'_k by edges in a new distinguisher D''_k (which depends on h) such that an edge (u,w) labeled $r \in \{0,1\}^n$ appears in D''_k if and only if, for some v, the path (u,v,w) appears in D'_k with the first edge (i.e., (u,v)) labeled r and the second edge (i.e., (v,w)) labeled $h(r)$. Needless to say, the crucial point is that $\mathsf{Pr}[D''_k(U_{(\ell'/2)\cdot n})=1]$ approximates $\mathsf{Pr}[D'_k(U_{\ell' \cdot n})=1]$.

The forgoing process can be applied to D''_k resulting in a distinguisher D'''_k of half the length, and so on. Each time we contract the current distinguisher by a factor of two, and do so by randomly selecting (and fixing) a new hash function. Thus, repeating the process for a logarithmic (in the depth of D'_k) number of times we obtain a distinguisher that only examines n bits, at which point we stop. In total, we have used $t \stackrel{\text{def}}{=} \log_2(\ell'/n) < \log_2 \ell(k)$ random hash functions. This means that we can generate a (pseudorandom) sequence that fools the original D_k by using a seed of length $n + t \cdot \log_2 |H_n|$. Using $n = \Theta(s(k))$ and an adequate family H_n (which, in particular, satisfies $|H_n| = 2^{O(n)}$), we obtain the desired $(s, 2^{-s})$-pseudorandom generator, which indeed uses a seed of length $O(s(k) \cdot \log_2 \ell(k)) = k$.

Rough sketch of the proof of Theorem 4.3 (see details in [44])

The main technical tool used in this proof is a suitable randomness extractor (see, e.g., [50] or [19, Apdx. D.4]), which is indeed a much more powerful tool than hashing functions. The basic idea is that when the distinguisher D_k is at some "distant" layer, say at layer $t = \Omega(s(k))$, it typically "knows" little about the random choices that led it there. That is, D_k has only $s(k)$ bits of memory, which leaves out $t - s(k)$ bits of "uncertainty" (or randomness) regarding the previous moves. Thus, much of the randomness that led D_k to its current state may be "re-used" (or "recycled"). To re-use these bits we need to extract *almost* uniform distribution on strings of sufficient length out of the aforementioned distribution (over $\{0,1\}^t$) that has entropy[5] at least $t - s(k)$. Furthermore, such an extraction requires some additional truly random bits, yet relatively few such bits. In particular, using $k' = \Omega(\log t)$ bits towards this end, the extracted bits are $\exp(-\Omega(k'))$ away from uniform.

The gain from the aforementioned recycling is significant if recycling is repeated sufficiently many times. Towards this end, we break the k-bit long seed into two parts, denoted $r' \in \{0,1\}^{k/2}$ and $(r_1, ..., r_{3\sqrt{k}})$, where $|r_i| = \sqrt{k}/6$, and set $n = k/3$. Intuitively, r' will be used for determining the first n steps, and it will be re-used (or recycled) together with r_i for determining the steps $i \cdot n + 1$ through $(i+1) \cdot n$. Looking at layer $i \cdot n$, we consider the information regarding r' that is "known" to D_k (when reaching a specific vertex at layer $i \cdot n$). Typically, the conditional distribution of r', given that we reached a specific vertex at layer $i \cdot n$, has (min-)entropy greater than $0.99 \cdot ((k/2) - s(k))$. Using r_i (as a seed of an extractor applied to r'), we can extract $0.9 \cdot ((k/2) - s(k) - o(k)) > k/3 = n$ bits that are almost-random (i.e., $2^{-\Omega(\sqrt{k})}$-close to U_n) with respect to D_k, and use these bits for determining the next n steps. Hence, using k random bits, we produce a sequence of length $(1+3\sqrt{k}) \cdot n > k^{3/2}$ that fools automata of space bound, say, $s(k) = k/10$. Specifically, using an extractor of the form $\mathrm{Ext} : \{0,1\}^{\sqrt{k}/6} \times \{0,1\}^{k/2} \to \{0,1\}^{k/3}$, we map the seed $(r', r_1, ..., r_{3\sqrt{k}})$ to the output sequence $(r', \mathrm{Ext}(r_1, r'), ..., \mathrm{Ext}(r_{3\sqrt{k}}, r'))$. Thus, *we obtained an $(s, 2^{-\Omega(\sqrt{s})})$-pseudorandom generator of stretch function $\ell(k) = k^{3/2}$.*

In order to obtain an arbitrary polynomial stretch rather than a specific polynomial stretch (i.e., $\ell(k) = k^{3/2}$), we iteratively compose generators as above with themselves (for a constant number of times). The basic composition combines an (s_1, ε_1)-pseudorandom generator of stretch function ℓ_1, denoted G_1, with an (s_2, ε_2)-pseudorandom generator of stretch function ℓ_2, denoted G_2. On input $s \in \{0,1\}^k$, the resulting generator first computes $G_1(s)$, parses $G_1(s)$ into t consecutive k'-bit long blocks, where $k' = s_1(k)/2$ and $t = \ell_1(k)/k'$, and applies G_2 to each block (outputting the concatenation of the t results). This generator, denoted G, has stretch $\ell(k) = t \cdot \ell_2(k')$, and for $s_1(k) = \Theta(k)$ we have $\ell(k) = \ell_1(k) \cdot \ell_2(\Omega(k))/O(k)$. The pseudorandom of G can be established via a hybrid argument (which refers to the intermediate hybrid distribution $G_2(U_{k'}^{(1)}) \cdots G_2(U_{k'}^{(t)})$ and uses the fact that the second step in the computation of G can be performed by a non-uniform automaton of space $s_1/2$).

4.2.2 Derandomization of space-complexity classes

As a direct application of Theorem 4.2, we obtain that $\mathcal{BPL} \subseteq \mathrm{DSPACE}(\log^2)$, where $\mathcal{BPL}$ denotes the log-space analogue of $\mathcal{BPP}$. (Recall that $\mathcal{NL} \subseteq \mathrm{DSPACE}(\log^2)$, but it is not known whether or not $\mathcal{BPL} \subseteq \mathcal{NL}$.)[6] A stronger derandomization result can be obtained by a finer analysis of the proof of Theorem 4.2.

[5]Actually, a stronger technical condition needs and can be imposed on the latter distribution. Specifically, with overwhelmingly high probability, at layer t, automaton D_k is at a vertex that can be reached in more than $2^{0.99 \cdot (t - s(k))}$ different ways. In this case, the distribution representing a random walk that reaches this vertex has min-entropy greater than $0.99 \cdot (t - s(k))$.

[6]Indeed, the log-space analogue of $\mathcal{RP}$, denoted $\mathcal{RL}$, is contained in $\mathcal{NL} \subseteq \mathrm{DSPACE}(\log^2)$, and thus the fact that Theorem 4.2 implies $\mathcal{RL} \subseteq \mathrm{DSPACE}(\log^2)$ is of no interest.

Theorem 4.4 *$\mathcal{BPL} \subseteq \mathcal{SC}$, where $\mathcal{SC}$ denotes the class of decision problems that can be solved by deterministic algorithms that run in polynomial-time and polylogarithmic-space.*

Thus, $\mathcal{BPL}$ (and in particular $\mathcal{RL} \subseteq \mathcal{BPL}$) is placed in a class not known to contain $\mathcal{NL}$. Another such result was subsequently obtained in [49]: Randomized log-space can be simulated in deterministic space $o(\log^2)$; specifically, in space $\log^{3/2}$. We mention that the archetypical problem of $\mathcal{RL}$ has been recently proved to be in $\mathcal{L}$ (see [47]).

Sketch of the proof of Theorem 4.4 (see details in [42])

We are going to use the generator construction provided in the proof of Theorem 4.2, but show that the main part of the seed (i.e., the sequence of hash functions) can be fixed (depending on the distinguisher at hand). Furthermore, this fixing can be performed in polylogarithmic space and polynomial-time. Specifically, wishing to derandomize a specific log-space computation (which refers to a specific input), we first obtain the corresponding distinguisher, denoted D'_k, that represents this computation (as a function of the outcomes of the internal coin tosses of the log-space algorithm). The key observation is that the question of whether or not a specific hash function $h \in H_n$ is good for a specific D'_k can be determined in space that is linear in $n = |h|/2$ and logarithmic in the size of D'_k. Indeed, the time-complexity of this decision procedure is exponential in its space-complexity. It follows that we can find a good $h \in H_n$, for a given D'_k, within these complexities (by scanning through all possible $h \in H_n$). Once a good h is found, we can also construct the corresponding graph D''_k (in which edges represent two-edge paths in D'_k), again within the same complexity. Actually, it will be more instructive to note that we can determine a step (i.e., an edge-traversal) in D''_k by making two steps (edge-traversals) in D'_k. This will allow to fix a hash function for D''_k, and so on. Details follow.

The main claim is that the entire process of finding a sequence of $t \stackrel{\text{def}}{=} \log_2 \ell'(k)$ good hash functions can be performed in space $t \cdot O(n + \log |D_k|) = O(n + \log |D_k|)^2$ and time $\text{poly}(2^n \cdot |D_k|)$; that is, the time-complexity is sub-exponential in the space-complexity (i.e., the time-complexity is significantly smaller than than the generic bound of $\exp(O(n + \log |D_k|)^2))$. Starting with $D_k^{(1)} = D'_k$, we find a good (for $D_k^{(1)}$) hashing function $h^{(1)} \in H_n$, which defines $D_k^{(2)} = D''_k$. Having found (and stored) $h^{(1)}, ..., h^{(i)} \in H_n$, which determine $D_k^{(i+1)}$, we find a good hashing function $h^{(i+1)} \in H_n$ for $D_k^{(i+1)}$ by emulating pairs of edge-traversals on $D_k^{(i+1)}$. Indeed, a key point is that we do *not* construct the sequence of graphs $D_k^{(2)}, ..., D_k^{(i+1)}$, but rather emulate an edge-traversal in $D_k^{(i+1)}$ by making 2^i edge-traversals in D'_k, using $h^{(1)}, ..., h^{(i)}$: The (edge-traversal) move $\alpha \in \{0,1\}^n$ starting at vertex v of $D_k^{(i+1)}$ translates to a sequence of 2^i moves starting at vertex v of D'_k, where the moves are determined by the 2^i-long sequence (of n-bit strings)

$$\overline{h}^{(0^i)}(\alpha), \overline{h}^{(0^{i-2}01)}(\alpha), \overline{h}^{(0^{i-2}10)}(\alpha), \overline{h}^{(0^{i-2}11)}(\alpha), ..., \overline{h}^{(1^i)}(\alpha),$$

where $\overline{h}^{(\sigma_i \cdots \sigma_1)}$ is the function obtained by the composition of a subsequence of the functions $h^{(i)}, ..., h^{(1)}$ determined by $\sigma_i \cdots \sigma_1$. Specifically, $\overline{h}^{(\sigma_i \cdots \sigma_1)}$ equals $h^{(i_{t'})} \circ \cdots \circ h^{(i_2)} \circ h^{(i_1)}$, where $i_1 < i_2 < \cdots < i_{t'}$ and $\{i_j : j=1, ..., t'\} = \{j : \sigma_j = 1\}$.

Recall that the ability to perform edge-traversals on $D_k^{(i+1)}$ allows to determine whether a specific function $h \in H_n$ is good for $D_k^{(i+1)}$. This is done by considering all the relevant triples (u, v, w) in $D_k^{(i+1)}$, computing for each such (u, v, w) the three quantities (i.e., probabilities) appearing in Eq. (4.2), and deciding accordingly. Trying all possible $h \in H_n$, we find a function (to be denoted $h^{(i+1)}$) that is good for $D_k^{(i+1)}$. This is done while using an additional storage of $s' = O(n + \log |D'_k|)$ (on top of the storage used to record $h^{(1)}, ..., h^{(i)}$), and in time that is exponential in s'. Thus, given D'_k, *we find a good sequence of hash functions,* $h^{(1)}, ..., h^{(t)}$*, in time exponential in* s' *and while using space* $s' + t \cdot \log_2 |H_n| = O(t \cdot s')$. Such a sequence of functions allows us to emulate edge-traversals on $D_k^{(t+1)}$, which in turn allows to (deterministically) approximate the probability that D'_k accepts a random input (i.e., the probability that, starting at the single source vertex of the first layer, automaton D'_k reaches some accepting vertex at the last layer). This approximation is obtained by computing the corresponding probability in $D_k^{(t+1)}$ by traversing all 2^n edges.

To summarize, given D'_k, we can (deterministically) approximate the probability that D'_k accepts a random input in $O(t \cdot s')$-space and $\exp(O(s' + n))$-time, where $s' = O(n + \log |D'_k|)$ and $t < \log_2 |D'_k|$. Recalling that $n = \Theta(\log |D'_k|)$, this means $O(\log |D'_k|)^2$-space and $\mathrm{poly}(|D'_k|)$-time. We comment that the approximation can be made accurate up to an additive term of $1/\mathrm{poly}(|D'_k|)$, but an additive term of $1/6$ suffices here.

Chapter 5

Special Purpose Generators

The pseudorandom generators considered so far were aimed at decreasing the amount of randomness utilized by any algorithm of certain time and/or space complexity (or even fully derandomizing the corresponding complexity class). For example, we considered the derandomization of classes such as $\mathcal{BPP}$ and $\mathcal{BPL}$. In the current chapter our goal is less ambitious. We only seek to derandomize (or decrease the randomness of) specific algorithms or rather classes of algorithms that use their random bits in certain (restricted) ways. For example, the algorithm's correctness may only require that its sequence of coin-tosses (or "blocks" in such a sequence) are pairwise-independent. Indeed, the restrictions that we shall consider here have a concrete and "structural" form, rather than the abstract complexity theoretic forms considered in previous chapters.

The aforementioned restrictions induce corresponding classes of very restricted distinguishers, which in particular are much weaker than the classes of distinguishers considered in previous chapters. These very restricted types of distinguishers induce correspondingly weak types of pseudorandom generators (which produce sequences that fool these distinguishers). Still, such generators have many applications (both in complexity theory and in the design of algorithms).

We start with the simplest of these generators: the pairwise-independence generator, and its generalization to t-wise independence for any $t \geq 2$. Such generators *perfectly* fool any distinguisher that only observe t locations in the output sequence. This leads naturally to almost pairwise (or t-wise) independence generators, which also fool such distinguishers (albeit non-perfectly). The latter generators are implied by a stronger class of generators, which is of independent interest: the small-bias generators. Small-bias generators fool any linear test (i.e., any distinguisher that merely considers the XOR of some fixed locations in the input sequence). We finally turn to the Expander Random Walk Generator: this generator produces a sequence of strings that hit any dense subset of strings with probability that is close to the hitting probability of a truly random sequence.[1]

[1] Related notions such as samplers, dispersers, and extractors are not treated here (although they were treated in [16, Sec. 3.6] and [19, Apdx. D.3&D.4]).

Comment regarding our parameterization: To maintain consistency with prior chapters, we continue to present the generators in terms of the seed length, denoted k. Since this is not the common presentation for most results presented in the sequel, we provide (in footnotes) the common presentation in which the seed length is determined as a function of other parameters.

5.1 Pairwise-Independence Generators

Pairwise (resp., t-wise) independence generators fool tests that inspect only two (resp., t) elements in the output sequence of the generator. Such local tests are indeed very restricted, yet they arise naturally in many settings. For example, such a test corresponds to a probabilistic analysis (of a procedure) that only relies on the pairwise independence of certain choices made by the procedure. We also mention that, in some natural range of parameters, pairwise independent sampling is as good as sampling by totally independent sample points (see, e.g., [19, Apdx. D.1.2.4]).

A t-wise independence generator of block-length $b:\mathbb{N}\to\mathbb{N}$ (and stretch function ℓ) is a relatively efficient deterministic algorithm (e.g., one that works in time polynomial in the output length) that expands a k-bit long random seed into a sequence of $\ell(k)/b(k)$ blocks, each of length $b(k)$, such that any t blocks are uniformly and independently distributed in $\{0,1\}^{t\cdot b(k)}$. That is, denoting the i^{th} block of the generator's output (on seed s) by $G(s)_i$, we require that for every $i_1 < i_2 < \cdots < i_t$ (in $[\ell(k)/b(k)]$) it holds that

$$G(U_k)_{i_1}, G(U_k)_{i_2}, ..., G(U_k)_{i_t} \equiv U_{t\cdot b(k)}. \tag{5.1}$$

We note that this condition holds even if the inspected t blocks are selected adaptively. In case $t = 2$, we call the generator pairwise independent.

5.1.1 Constructions

In the first construction, we refer to $\mathrm{GF}(2^{b(k)})$, the finite field of $2^{b(k)}$ elements, and associate its elements with $\{0,1\}^{b(k)}$.

Proposition 5.1 (t-wise independence generator):[2] *Let t be a fixed integer and $b,\ell,\ell':\mathbb{N}\to\mathbb{N}$ such that $b(k) = k/t$, $\ell'(k) = \ell(k)/b(k) > t$ and $\ell'(k) \leq 2^{b(k)}$. Let $\alpha_1, ..., \alpha_{\ell'(k)}$ be fixed distinct elements of the field $\mathrm{GF}(2^{b(k)})$. For $s_0, s_1, ..., s_{t-1} \in \{0,1\}^{b(k)}$, let*

$$G(s_0, s_1, ..., s_{t-1}) \stackrel{\text{def}}{=} \left(\sum_{j=0}^{t-1} s_j\alpha_1^j \,,\, \sum_{j=0}^{t-1} s_j\alpha_2^j \,, ..., \sum_{j=0}^{t-1} s_j\alpha_{\ell'(k)}^j \right) \tag{5.2}$$

where the arithmetic is that of $\mathrm{GF}(2^{b(k)})$. Then, G is a t-wise independence generator of block-length b and stretch ℓ.

[2] In the common presentation of this t-wise independence generator, the length of the seed is determined as a function of the desired block-length and stretch. That is, given the parameters b and $\ell' \leq 2^b$, the seed length is set to $t \cdot b$.

That is, given a seed that consists of t elements of $\mathrm{GF}(2^{b(k)})$, the generator outputs a sequence of $\ell'(k)$ such elements. The proof of Proposition 5.1 is based on the observation that, for any fixed $v_0, v_1, ..., v_{t-1}$, the condition $\{G(s_0, s_1, ..., s_{t-1})_{i_j} = v_j\}_{j=1}^{t}$ constitutes a system of t linear equations over $\mathrm{GF}(2^{b(k)})$ (in the variables $s_0, s_1, ..., s_{t-1}$) such that the equations are linearly-independent. (Thus, linear independence of certain expressions yields statistical independence of the corresponding random variables.)

A somewhat tedious comment. We warn that Eq. (5.2) does not provide a fully explicit construction (of a generator). What is missing is an explicit representation of $\mathrm{GF}(2^{b(k)})$, which requires an irreducible polynomial of degree $b(k)$ over $\mathrm{GF}(2)$. For specific values of $b(k)$, a good representation does exist: e.g., for $d \stackrel{\text{def}}{=} b(k) = 2 \cdot 3^e$ (with e being an integer), the polynomial $x^d + x^{d/2} + 1$ is irreducible over $\mathrm{GF}(2)$.

We note that a construction analogous to Eq. (5.2) works for every finite field (e.g., a finite field of any prime cardinality), but the problem of providing an explicit representation of such a field remains non-trivial also in other cases (e.g., consider the problem of finding a prime number of size approximately $2^{b(k)}$). The latter fact is the main motivation for considering the following alternative construction for the case of $t = 2$.

The following construction uses (random) affine transformations (as possible seeds). In fact, better performance (i.e., shorter seed length) is obtained by using affine transformations affected by Toeplitz matrices. A Toeplitz matrix is a matrix with all diagonals being homogeneous (see Figure 5.1); that is, $T = (t_{i,j})$ is a Toeplitz matrix if $t_{i,j} = t_{i+1,j+1}$ for all i, j. Note that a Toeplitz matrix is determined by its first row and first column (i.e., the values of $t_{1,j}$'s and $t_{i,1}$'s).

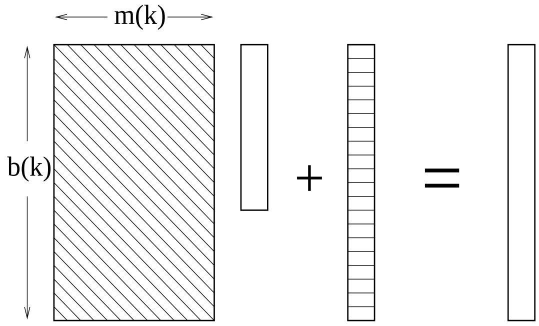

Figure 5.1: An affine transformation affected by a Toeplitz matrix.

Proposition 5.2 (alternative pairwise independence generator, see Figure 5.1):[3] *Let $b, \ell, \ell', m : \mathbb{N} \to \mathbb{N}$ such that $\ell'(k) = \ell(k)/b(k)$ and $m(k) = \lceil \log_2 \ell'(k) \rceil = k -$*

[3]In the common presentation of this pairwise independence generator, the length of the seed is determined as a function of the desired block-length and stretch. That is, given the parameters b and ℓ', the seed length is set to $2b + \lceil \log_2 \ell' \rceil - 1$.

$2b(k)+1$. *Associate* $\{0,1\}^n$ *with the* n*-dimensional vector space over* GF(2)*, and let* $v_1, ..., v_{\ell'(k)}$ *be fixed distinct vectors in the* $m(k)$*-dimensional vector space over* GF(2)*. For* $s \in \{0,1\}^{b(k)+m(k)-1}$ *and* $r \in \{0,1\}^{b(k)}$*, let*

$$G(s,r) \stackrel{\rm def}{=} (T_s v_1 + r \, , \, T_s v_2 + r \, , ..., \, T_s v_{\ell'(k)} + r) \tag{5.3}$$

where T_s *is an* $b(k)$*-by-*$m(k)$ *Toeplitz matrix specified by the string* s*. Then,* G *is a pairwise independence generator of block-length* b *and stretch* ℓ*.*

That is, given a seed that represents an affine transformation defined by an $b(k)$-by-$m(k)$ Toeplitz matrix and a $b(k)$-dimensional vector, the generator outputs a sequence of $\ell'(k) \leq 2^{m(k)}$ strings, each of length $b(k)$. Note that $k = 2b(k) + m(k) - 1$, and that the stretching property requires $\ell'(k) > k/b(k)$. The proof of Proposition 5.2 is also based on the observation that linear independence of certain expressions yields statistical independence of the corresponding random variables: here $\{G(s,r)_{i_j} = v_j\}_{j=1}^2$ is a system of $2b(k)$ linear equations over GF(2) (in Boolean variables representing the bits of s and r) such that the equations are linearly-independent. We mention that a construction analogous to Eq. (5.3) works for every finite field.

A stronger notion of efficient generation. Ignoring the issue of finding a representation for a large finite field, both the foregoing constructions are efficient in the sense that the generator's output can be produced in time that is polynomial in its length. Actually, the aforementioned constructions satisfy a stronger notion of efficient generation, which is useful in several applications. Specifically, there exists a polynomial-time algorithm that given a seed, $s \in \{0,1\}^k$, and a block location $i \in [\ell'(k)]$ (in binary), outputs the $i^{\rm th}$ block of the corresponding output (i.e., the $i^{\rm th}$ block of $G(s)$). Note that, in the case of the first construction (captured by Eq. (5.2)), this stronger notion depends on the ability to find a representation of GF$(2^{b(k)})$ in poly(k)-time.[4] Recall that this is possible in the case that $b(k)$ is of the form $2 \cdot 3^e$.

5.1.2 Applications (a brief review)

Pairwise independence generators do suffice for a variety of applications (cf., [58]). Many of these applications are based on the fact that "Laws of Large Numbers" hold for sequences of trials that are pairwise independent (rather than totally independent). This fact stems from the application of Chebyshev's Inequality, and is the basis of the (rather generic) application to ("pairwise independent") sampling. As a concrete example, we mention the derandomization of a fast parallel algorithm for the Maximal Independent Set problem (as presented in [38, Sec. 12.3]).[5]

[4] For the basic notion of efficiency, it suffices to find a representation of GF$(2^{b(k)})$ in poly$(\ell(k))$-time, which can be done by an exhaustive search in the case that $b(k) = O(\log \ell(k))$.

[5] The core of this algorithm is picking each vertex with probability that is inversely proportional to the vertex's degree. The analysis only requires that these choices be pairwise independent. Furthermore, these choices can be (approximately) implemented by uniformly selecting values in a sufficiently large set.

In general, whenever the analysis of a randomized algorithm only relies on the hypothesis that some objects are distributed in pairwise independent manner, we may replace its random choices by a sequence of choices that is generated by a pairwise independence generator. Thus, pairwise independence generators suffice for fooling distinguishers that are derived from some natural and interesting randomized algorithms.

Referring to Eq. (5.2), we remark that, for any constant $t \geq 2$, the cost of derandomization (i.e., going over all 2^k possible seeds) is exponential in the block-length (because $b(k) = k/t$). On the other hand, the number of blocks is at most exponential in the block-length (because $\ell'(k) \leq 2^{b(k)}$), and so if a larger number of blocks is needed, then we can artificially increase the block-length in order to accommodate this (i.e., set $b(k) = \log_2 \ell'(k)$). Thus, the cost of derandomization is polynomial in $\max(\ell'(k), 2^{b'(k)})$, where $\ell'(k)$ denotes the desired number of blocks and $b'(k)$ the desired block-length. (In other words, $\ell'(k)$ denotes the desired number of random choices, and $2^{b'(k)}$ represents the size of the domain of each of these choices.) It follows that *whenever the analysis of a randomized algorithm can be based on a constant amount of independence* between feasibly-many random choices, each taken within a domain of feasible size, *then a feasible derandomization is possible.*

5.2 Small-Bias Generators

As stated in Sec. 5.1.2, $O(1)$-wise independence generators allow for the efficient derandomization of any efficient randomized algorithm the analysis of which is only based on a *constant amount of independence* between the bits of its random-tape. This restriction is due to the fact that t-wise independence generators of stretch ℓ require a seed of length $\Omega(t \cdot \log \ell)$. Trying to go beyond constant-independence in such derandomizations (while using seeds of length that is logarithmic in the length of the pseudorandom sequence) was the original motivation of the notion of small-bias generators. Specifically, as we shall see in Sec. 5.2.2, small-bias generators yield meaningful approximations of t-wise independence sequences (based on logarithmic-length seeds).

While the aforementioned type of derandomizations remains an important application of small-bias generators, the latter are of independent interest and have found numerous other applications. In particular, small-bias generators fool "global tests" that examine the entire output sequence and not merely a fixed number of positions in it (as in the case of limited independence generators). Specifically, a small-bias generator produces a sequence of bits that fools any linear test (i.e., a test that computes a fixed linear combination of the bits).

For $\varepsilon : \mathbb{N} \to [0, 1]$, an ε-bias generator with stretch function ℓ is a relatively efficient deterministic algorithm (e.g., working in $\text{poly}(\ell(k))$ time) that expands a k-bit long random seed into a sequence of $\ell(k)$ bits such that for any fixed non-empty set $S \subseteq \{1, ..., \ell(k)\}$ the bias of the output sequence over S is at most $\varepsilon(k)$. The bias of a sequence of n (possibly dependent) Boolean random variables

$\zeta_1, ..., \zeta_n \in \{0,1\}$ over a set $S \subseteq \{1,..,n\}$ is defined as

$$2 \cdot \left| \mathsf{Pr}[\oplus_{i \in S} \zeta_i = 1] - \frac{1}{2} \right| = |\mathsf{Pr}[\oplus_{i \in S} \zeta_i = 1] - \mathsf{Pr}[\oplus_{i \in S} \zeta_i = 0]|. \tag{5.4}$$

The factor of 2 was introduced so to make these biases correspond to the Fourier coefficients of the distribution (viewed as a function from $\{0,1\}^n$ to the reals).[6]

5.2.1 Constructions

Relatively efficient small-bias generators with exponential stretch and exponentially vanishing bias are known.

Theorem 5.3 (small-bias generators):[7] *For some universal constant $c > 0$, let $\ell : \mathbb{N} \to \mathbb{N}$ and $\varepsilon : \mathbb{N} \to [0,1]$ such that $\ell(k) \le \varepsilon(k) \cdot \exp(k/c)$. Then, there exists an ε-bias generator with stretch function ℓ operating in time that is polynomial in the length of its output.*

In particular, we may have $\ell(k) = \exp(k/2c)$ and $\varepsilon(k) = \exp(-k/2c)$. Three simple constructions of small-bias generators that satisfy Theorem 5.3 are known (see [3]). One of these constructions is based on Linear Feedback Shift Registers (LFSRs), where the seed of the generator is used to determine both the "feedback rule" and the "start sequence" of the LFSR. Specifically, a feedback rule of a t-long LFSR is an irreducible polynomial of degree t over GF(2), denoted $f(x) = x^t + \sum_{j=0}^{t-1} f_j x^j$ where $f_0 = 1$, and the (ℓ-bit long) sequence produced by the corresponding LFSR based on the start sequence $s_0 s_1 \cdots s_{t-1} \in \{0,1\}^t$ is defined as $r_0 r_1 \cdots r_{\ell-1}$, where

$$r_i = \begin{cases} s_i & \text{if } i \in \{0,1,...,t-1\} \\ \sum_{j=0}^{t-1} f_j \cdot r_{i-t+j} & \text{if } i \in \{t, t+1, ..., \ell-1\} \end{cases} \tag{5.6}$$

(see Figure 5.2). As stated previously, in the corresponding small-bias generator the k-bit long seed is used for selecting an *almost* uniformly distributed feedback rule f (i.e., a random irreducible polynomial of degree $t = k/2$) and a uniformly distributed start sequence s (i.e., a random t-bit string).[8] The corresponding $\ell(k)$-bit long output $r = r_0 r_1 \cdots r_{\ell(k)-1}$ is computed as in Eq. (5.6).

[6] To see the correspondence replace $\{0,1\}$ by $\{\pm 1\}$, and substitute XOR by multiplication. The bias with respect to a set S is thus written as

$$\left| \mathsf{Pr}\left[\prod_{i \in S} \zeta_i = +1\right] - \mathsf{Pr}\left[\prod_{i \in S} \zeta_i = -1\right] \right| = \left| \mathsf{E}\left[\prod_{i \in S} \zeta_i\right] \right|, \tag{5.5}$$

which is merely the (absolute value of the) Fourier coefficient corresponding to S.

[7] In the common presentation of this generator, the length of the seed is determined as a function of the desired bias and stretch. That is, given the parameters ε and ℓ, the seed length is set to $c \cdot \log(\ell/\varepsilon)$. We comment that using [3] the constant c is merely 2 (i.e., $k \approx 2\log_2(\ell/\varepsilon)$), whereas using [39] $k \approx \log_2 \ell + 4\log_2(1/\varepsilon)$.

[8] Note that an implementation of this generator requires an algorithm for selecting an almost random irreducible polynomial of degree $t = \Omega(k)$. A simple algorithm proceeds by enumerating all irreducible polynomials of degree t, and selecting one of them at random. This algorithm can be implemented (using t random bits) in $\exp(t)$-time, which is $\text{poly}(\ell(k))$ if $\ell(k) = \exp(\Omega(k))$. A $\text{poly}(t)$-time algorithm that uses $O(t)$ random bits is described in [3, Sec. 8].

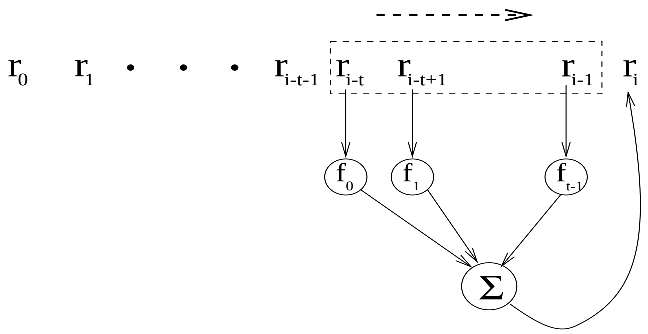

Figure 5.2: The LFSR small-bias generator (for $t = k/2$).

A stronger notion of efficient generation. As in Section 5.1.1, we note that the aforementioned constructions satisfy a stronger notion of efficient generation, which is useful in several applications. That is, there exists a polynomial-time algorithm that given a k-bit long seed and a bit location $i \in [\ell(k)]$ (in binary), outputs the i^{th} bit of the corresponding output.

5.2.2 Applications (a brief review)

An archetypical application of small-bias generators is for producing short and random "fingerprints" (or "digests") of strings such that equality/inequality among strings is (probabilistically) reflected in equality/inequality between their corresponding fingerprints. The key observation is that checking whether or not $x = y$ is probabilistically reducible to checking whether the inner product modulo 2 of x and r equals the inner product modulo 2 of y and r, *where r is produced by a small-bias generator G*. Thus, the pair (s, v), where s is a random seed to G and v equals the inner product modulo 2 of z and $G(s)$, serves as the randomized fingerprint of the string z. One advantage of this reduction is that only few bits (i.e., the seed of the generator and the result of the inner product) needs to be "communicated between x and y" in order to enable the checking. A related advantage is the low randomness complexity of this reduction, which uses $|s|$ rather than $|G(s)|$ random bits, where $|s|$ may be $O(\log |G(s)|)$. This low (i.e., logarithmic) randomness-complexity underlies the application of small-bias generators to the construction of PCP systems and amplifying reductions of gap problems regarding the satisfiability of systems of equations (see, e.g., [19, Exer. 10.6]).

Small-bias generators have been used in a variety of areas (e.g., inapproximation, structural complexity, and applied cryptography; see references in [16, Sec 3.6.2]). In addition, as shown next, small-bias generators seem an important tool in the design of various types of "pseudorandom" objects.

Approximate independence generators. As hinted at the beginning of this section, small-bias is related to approximate versions of limited independence.[9] Actually, even a restricted type of ε-bias (in which only subsets of size $t(k)$ are required to have bias upper-bounded by ε) implies that any $t(k)$ bits in the said sequence are $2^{t(k)/2} \cdot \varepsilon(k)$-close to $U_{t(k)}$, where here we refer to the variation distance (i.e., Norm-1 distance) between the two distributions. (The max-norm of the difference is bounded by $\varepsilon(k)$.)[10] Combining Theorem 5.3 and the foregoing upper-bound, we obtain *generators with exponential stretch* (i.e., $\ell(k) = \exp(\Omega(k))$) that produce *sequences that are approximately $\Omega(k)$-wise independent in the sense that any $t(k) = \Omega(k)$ bits in them are $2^{-\Omega(k)}$-close to $U_{t(k)}$*. Thus, whenever the analysis of a randomized algorithm can be based on a logarithmic amount of (almost) independence between feasibly-many binary random choices, a feasible derandomization is possible (by using an adequate generator of logarithmic seed length).[11]

Extensions to non-binary choices were considered in various works (see references in [16, Sec 3.6.2]). Some of these works also consider the related problem of constructing small "discrepancy sets" for geometric and combinatorial rectangles.

t-universal set generators. Using the aforementioned upper-bound on the max-norm (of the deviation from uniform of any t locations), any ε-bias generator yields a *t-universal set generator*, provided that $\varepsilon < 2^{-t}$. The latter generator outputs sequences such that in every subsequence of length t all possible 2^t patterns occur (i.e., each for at least one possible seed). Such generators have many applications.

5.2.3 Generalization

In this section, we outline a generalization of the treatment of small-bias generators to the generation of sequences over an arbitrary finite field. Focusing on the case of a field of prime characteristic, denoted $\mathrm{GF}(p)$, we first define an adequate notion of bias. Generalizing Eq. (5.5) (in Footnote 6), we define the bias of a sequence of n (possibly dependent) random variables $\zeta_1, ..., \zeta_n \in \mathrm{GF}(p)$ with respect to the linear combination $(c_1, ..., c_n) \in \mathrm{GF}(p)^n$ as $\left\| \mathsf{E}\left[\omega^{\sum_{i=1}^n c_i\zeta_i}\right]\right\|$, where ω denotes the p^{th} (complex) root of unity (i.e., $\omega = -1$ if $p = 2$). We mention that upper-bounds on the biases of $\zeta_1, ..., \zeta_n$ (with respect to any non-zero linear combinations) yield upper-bounds on the distance of $\sum_{i=1}^n c_i\zeta_i$ from the uniform distribution over $\mathrm{GF}(p)$.

[9] We warn that, unlike in the case of perfect independence, here we refer only to the distribution on fixed bit locations.

[10] Both bounds are derived from the Norm2 bound on the difference vector (i.e., the difference between the two probability vectors).

[11] Furthermore, relying on the linearity of the construction presented in Proposition 5.1, we can obtain *generators with double-exponential stretch* (i.e., $\ell(k) = \exp(2^{\Omega(k)})$) *that are approximately $t(k)$-independent* (in the foregoing sense). That is, we may obtain generators with stretch $\ell(k) = 2^{2^{\Omega(k)}}$ producing bit sequences in which any $t(k) = \Omega(k)$ positions have variation distance at most $\varepsilon(k) = 2^{-\Omega(k)}$ from uniform; in other words, such generators may have seed-length $k = O(t(k) + \log(1/\varepsilon(k)) + \log\log \ell(k))$. In the corresponding result for the max-norm distance, it suffices to have $k = O(\log(t(k)/\varepsilon(k)) + \log\log \ell(k))$.

We say that $S \subseteq \mathrm{GF}(p)^n$ is an ε-bias probability space if a uniformly selected sequence in S has bias at most ε with respect to any non-zero linear combination over $\mathrm{GF}(p)$. (Whenever such a space is efficiently constructible, it yields a corresponding ε-biased generator.) We mention that the LFSR construction, outlined in Sec. 5.2.1, generalizes to $\mathrm{GF}(p)$ and yields an ε-bias probability space of size (at most) p^{2e}, where $e = \lceil \log_p(n/\varepsilon) \rceil$. Such constructions can be used in applications that generalize those in Sec. 5.2.2.

5.3 Random Walks on Expanders

In this section we review generators that produce a sequence of values by taking a random walk on a large graph that has a small degree but an adequate "mixing" property. Such a graph is called an expander, and by taking a random walk (of length ℓ') on it we generate a sequence of ℓ' values over its vertex set, while using a random seed of length $b+(\ell'-1)\cdot\log_2 d$, where 2^b denotes the number of vertices in the graph and d denotes its degree. This seed length should be compared against the $\ell' \cdot b$ random bits required for generating a sequence of ℓ' independent samples from $\{0,1\}^b$ (or taking a random walk on a clique of size 2^b). Interestingly, as we shall see, the pseudorandom sequence (generated by the said random walk on an expander) *behaves similarly to a truly random sequence with respect to hitting any dense subset of* $\{0,1\}^b$. Let us start by defining this property (or rather by defining the corresponding hitting problem).

Definition 5.4 (the hitting problem): *A sequence of* (possibly dependent) *random variables, denoted* $(X_1, ..., X_{\ell'})$*, over* $\{0,1\}^b$ *is* (ε, δ)-hitting *if for any* (target) *set* $T \subseteq \{0,1\}^b$ *of cardinality at least* $\varepsilon \cdot 2^b$*, with probability at least* $1-\delta$*, at least one of these variables hits* T*; that is,* $\Pr[\exists i \text{ s.t. } X_i \in T] \geq 1-\delta$.

Clearly, a truly random sequence of length ℓ' over $\{0,1\}^b$ is (ε, δ)-hitting for $\delta = (1-\varepsilon)^{\ell'}$. The aforementioned "expander random walk generator" (to be described next) achieves similar behavior. Specifically, for arbitrary small $c > 0$ (which depends on the degree and the mixing property of the expander), the generator's output is (ε, δ)-hitting for $\delta = (1-(1-c)\cdot\varepsilon)^{\ell'}$. To describe this generator, we need to discuss expanders.

5.3.1 Background: expanders and random walks on them

By expander graphs (or expanders) of degree d and eigenvalue bound $\lambda < d$, we actually mean an infinite family of d-regular graphs, $\{G_N\}_{N \in \mathbb{S}}$ ($\mathbb{S} \subseteq \mathbb{N}$), such that G_N is a d-regular graph over N vertices and the absolute value of all eigenvalues, save the biggest one, of the adjacency matrix of G_N is upper-bounded by λ. For simplicity, we shall assume that the vertex set of G_N is $[N]$ (although in some constructions a somewhat more redundant representation is more convenient). We will refer to such a family as to a (d, λ)-expander (for $\mathbb{S}$). This technical definition is related to the aforementioned notion of "mixing" (which refers to the rate at which a random walk starting at a fixed vertex reaches uniform distribution over the graph's vertices).

We are interested in explicit constructions of such graphs, by which we mean that there exists a polynomial-time algorithm that on input N (in binary), a vertex v in G_N and an index $i \in \{1, ..., d\}$, returns the i^{th} neighbor of v. (We also require that the set $\mathbb{S}$ for which G_N's exist is sufficiently "tractable" – say that given any $n \in \mathbb{N}$ one may efficiently find an $s \in \mathbb{S}$ such that $n \leq s < 2n$.) Several explicit constructions of expanders are known (cf., e.g., [36, 35, 48]). Below, we rely on the fact that for every $\overline{\lambda} > 0$, there exist d and an explicit construction of a $(d, \overline{\lambda} \cdot d)$-expander over $\{2^b : b \in \mathbb{N}\}$.[12] The relevant (to us) fact about expanders is stated next.

Theorem 5.5 (Expander Random Walk Theorem): *Let $G = (V, E)$ be an expander graph of degree d and eigenvalue bound λ. Consider taking a random walk on G by uniformly selecting a start vertex and taking $\ell' - 1$ additional random steps such that at each step the walk uniformly selects an edge incident at the current vertex and traverses it. Then, for any $W \subseteq V$ and $\rho \stackrel{\text{def}}{=} |W|/|V|$, the probability that such a random walk stays in W is at most*

$$\rho \cdot \left(\rho + (1 - \rho) \cdot \frac{\lambda}{d}\right)^{\ell' - 1} \tag{5.7}$$

Thus, a random walk on an expander is "pseudorandom" with respect to the hitting property (i.e., when we consider hitting the set $V \setminus W$ and use $\varepsilon = 1 - \rho$); that is, a set of density ε is hit with probability at least $1 - \delta$, where $\delta = (1 - \varepsilon) \cdot (1 - \varepsilon + (\lambda/d) \cdot \varepsilon)^{\ell' - 1} < (1 - (1 - (\lambda/d)) \cdot \varepsilon)^{\ell'}$. A proof of Theorem 5.5 is given in [28], while a proof of an upper-bound that is weaker than Eq. (5.7) is outlined next.

A weak version of the Expander Random Walk Theorem: Using notations as in Theorem 5.5, we claim that the probability that a random walk of length ℓ' stays in W is at most $(\rho + (\lambda/d)^2)^{\ell'/2}$. In fact, we make a more general claim that refers to the probability that a random walk of length ℓ' intersects $W_0 \times W_1 \times \cdots \times W_{\ell'-1}$. The claimed upper-bound is

$$\sqrt{\rho_0} \cdot \prod_{i=1}^{\ell'-1} \sqrt{\rho_i + (\lambda/d)^2}, \tag{5.8}$$

where $\rho_i \stackrel{\text{def}}{=} |W_i|/|V|$. In order to prove Eq. (5.8), we view the random walk as the evolution of a corresponding probability vector under suitable transformations. The transformations correspond to taking a random step in the graph and to passing through a "sieve" that keeps only the entries that correspond to the current set W_i. The key observation is that the first transformation shrinks the component that is orthogonal to the uniform distribution, whereas the second transformation shrinks the component that is in the direction of the uniform distribution. For further details, see [19, Apdx. E.2.1.3].

[12] This can be obtained with $d = \text{poly}(1/\overline{\lambda})$. In fact $d = O(1/\overline{\lambda}^2)$, which is optimal, can be obtained too, albeit with graphs of sizes that are only approximately powers of two.

5.3.2 The generator

Using Theorem 5.5 and an explicit $(2^t, \overline{\lambda} \cdot 2^t)$-expander, we obtain a generator that produces sequences that are (ε, δ)-hitting for δ that is almost optimal.

Proposition 5.6 (The Expander Random Walk Generator):[13] *For every constant $\overline{\lambda} > 0$, consider an explicit construction of $(2^t, \overline{\lambda} \cdot 2^t)$-expanders for $\{2^n : n \in \mathbb{N}\}$, where $t \in \mathbb{N}$ is a sufficiently large constant. For $v \in [2^n] \equiv \{0,1\}^n$ and $i \in [2^t] \equiv \{0,1\}^t$, denote by $\Gamma_i(v)$ the vertex of the corresponding 2^n-vertex graph that is reached from vertex v when following its i^{th} edge. For $b, \ell' : \mathbb{N} \to \mathbb{N}$ such that $k = b(k) + (\ell'(k) - 1) \cdot t < \ell'(k) \cdot b(k)$, and for $v_0 \in \{0,1\}^{b(k)}$ and $i_1, ..., i_{\ell'(k)-1} \in [2^t]$, let*

$$G(v_0, i_1, ..., i_{\ell'(k)-1}) \stackrel{\text{def}}{=} (v_0, v_1, ..., v_{\ell'(k)-1}), \tag{5.9}$$

where $v_j = \Gamma_{i_j}(v_{j-1})$. Then, G has stretch $\ell(k) = \ell'(k) \cdot b(k)$, and $G(U_k)$ is (ε, δ)-hitting for any $\varepsilon > 0$ and $\delta = (1 - (1 - \overline{\lambda}) \cdot \varepsilon)^{\ell'(k)}$.

The stretch of G is maximized at $b(k) \approx k/2$ (and $\ell'(k) = k/2t$), but maximizing the stretch is not necessarily the goal in all applications. In many applications, the parameters n, ε and δ are given, and the goal is to derive a generator that produces (ε, δ)-hitting sequences over $\{0,1\}^n$ while minimizing both the length of the sequence and the amount of randomness used by the generator (i.e., the seed length). Indeed, Proposition 5.6 suggests using sequences of length $\ell' \approx \varepsilon^{-1} \log_2(1/\delta)$ that are generated based on a random seed of length $n + O(\ell')$.

Expander random-walk generators have been used in a variety of areas (e.g., PCP and inapproximability (see [8, Sec. 11.1]), cryptography (see [17, Sec. 2.6]), and the design of various types of "pseudorandom" objects.

[13] In the common presentation of this generator, the length of the seed is determined as a function of the desired block-length and stretch. That is, given the parameters b and ℓ', the seed length is set to $b + (\ell' - 1) \cdot t$.

Notes

Figure 5.3 depicts some of the notions of pseudorandom generators discussed in this primer. We highlight a key distinction between the case of general-purpose pseudorandom generators (treated in Chapter 2) and the other cases (cf, e.g., Chapters 3 and 4): in the former case the distinguisher is more complex than the generator, whereas in the latter cases the generator is more complex than the distinguisher. Specifically, a general-purpose generator runs in (some *fixed*) polynomial-time and needs to withstand *any* probabilistic polynomial-time distinguisher. In fact, some of the proofs presented in Chapter 2 utilize the fact that the distinguisher can invoke the generator on seeds of its choice. In contrast, the Nisan-Wigderson Generator, analyzed in Theorem 3.5, runs more time than the distinguishers that it tries to fool, and the proof relies on this fact in an essential manner. Similarly, the space-complexity of the space-resilient generators presented in Chapter 4 is higher than the space-bound of the distinguishers that they fool.

TYPE	distinguisher's resources	generator's resources	stretch (i.e., $\ell(k)$)	comments
gen.-purpose	$p(k)$-time, $\forall$ poly. p	poly(k)-time	poly(k)	Assumes OW
canon. derand.	$2^{k/O(1)}$-time	$2^{O(k)}$-time	$2^{k/O(1)}$	Assumes EvC
space-bounded robustness	$s(k)$-space, $s(k) < k$	$O(k)$-space	$2^{k/O(s(k))}$	runs in time poly$(k) \cdot \ell(k)$
	$k/O(1)$-space	$O(k)$-space	poly(k)	
t-wise indepen.	inspect t positions	poly$(k) \cdot \ell(k)$-time	$2^{k/O(t)}$	(e.g., pairwise)
small bias	linear tests	poly$(k) \cdot \ell(k)$-time	$2^{k/O(1)} \cdot \varepsilon(k)$	
expander random walk	"hitting"	poly$(k) \cdot \ell(k)$-time	$\ell'(k) \cdot b(k)$	
	$(0.5, 2^{-\ell'(k)/O(1)})$-hitting for $\{0,1\}^{b(k)}$, with $\ell'(k) = ((k - b(k))/O(1)) + 1$.			

By OW we denote the assumption that one-way functions exists, and by EvC we denote the assumption that the class $\mathcal{E}$ has (almost-everywhere) exponential circuit complexity.

Figure 5.3: Pseudorandom generators at a glance.

(The following historical notes do not mention several technical contributions that played an important role in the development of the area. For further details, the reader is referred to [16, Chap. 3]. In fact, the current text is a revision of [16, Chap. 3], providing significantly more details for the main topics, and omitting relatively secondary material.)

The general paradigm of pseudorandom generators. Our presentation, which views vastly different notions of pseudorandom generators as incarnations of a general paradigm, has emerged mostly in retrospect. We note that, while the historical study of the various notions was mostly unrelated at a technical level, the case of general-purpose pseudorandom generators served as a source of inspiration to most of the other cases. In particular, the concept of computational indistinguishability, the connection between hardness and pseudorandomness, and the equivalence between pseudorandomness and unpredictability, appeared first in the context of general-purpose pseudorandom generators (and inspired the development of "generators for derandomization" and "generators for space bounded machines"). Indeed, the study of the special-purpose generators (see Chapter 5) was unrelated to all of these.

General-purpose pseudorandom generators. The concept of *computational indistinguishability*, which underlies the entire computational approach to randomness, was suggested by Goldwasser and Micali [24] in the context of defining secure encryption schemes. Indeed, computational indistinguishability plays a key role in cryptography (see [17, 18]). The general formulation of computational indistinguishability is due to Yao [59]. Using the hybrid technique of [24], Yao also observed that defining pseudorandom generators as producing sequences that are computationally indistinguishable from the corresponding uniform distribution is equivalent to defining such generators as producing unpredictable sequences. The latter definition originates in the earlier work of Blum and Micali [9].

Blum and Micali [9] pioneered the rigorous study of pseudorandom generators and, in particular, the construction of pseudorandom generators based on some simple intractability assumption. In particular, they constructed pseudorandom generators assuming the intractability of the Discrete Logarithm Problem (over prime fields). Their work also introduces basic paradigms that were used in all subsequent improvements (cf., e.g., [59, 25]). We refer to the transformation of computational difficulty into pseudorandomness, the use of hard-core predicates (also defined in [9]), and the iteration paradigm (cf. Eq. (2.9)).

Theorem 2.14 (by which pseudorandom generators exist if and only if one-way functions exist) is due to Håstad, Impagliazzo, Levin and Luby [25], building on the hard-core predicate of [22] (see Theorem 2.11). Unfortunately, the current proof of Theorem 2.14 is very complicated and unfit for presentation in this primer. Presenting a simpler and tighter (cf. Sec. 2.7) proof is indeed an important research project.

Pseudorandom functions were defined and first constructed by Goldreich, Goldwasser and Micali [20]. We also mention (and advocate) the study of a general theory of pseudorandom objects initiated in [21]. Finally, we mention that a more detailed treatment of general-purpose pseudorandom generators is provided in [17, Chap. 3].

Derandomization of time-complexity classes. As observed by Yao [59], a non-uniformly strong notion of pseudorandom generators yields non-trivial derandomization of time-complexity classes. A key observation of Nisan [40, 43] is that whenever a pseudorandom generator is used in this way, it suffices to require that the generator runs in time that is exponential in its seed length, and so the generator may have running-time greater than the distinguisher (representing the algorithm to be derandomized). This observation motivates the definition of canonical derandomizers as well as the construction of Nisan and Wigderson [40, 43], which is the basis for further improvements culminating in [26]. Part 1 of Theorem 3.6 (i.e., the so-called "high end" derandomization of $\mathcal{BPP}$) is due to Impagliazzo and Wigderson [26], whereas Part 2 (the "low end") is from [43].

The Nisan–Wigderson Generator [43] was subsequently used in several ways transcending its original presentation. We mention its application towards fooling non-deterministic machines (and thus derandomizing constant-round interactive proof systems) and to the construction of randomness extractors (see [53] as well as [50]).

In contrast to the aforementioned derandomization results, which place $\mathcal{BPP}$ in some worst-case deterministic complexity class based on some non-uniform (worst-case) assumption, we now mention a result that places $\mathcal{BPP}$ in an average-case deterministic complexity class based on a uniform-complexity (worst-case) assumption. We refer specifically to a theorem, which is due to Impagliazzo and Wigderson [27] (but is not presented in the main text), that asserts the following: *if* $\mathcal{BPP}$ *is not contained in* $\mathcal{EXP}$ (almost everywhere) *then* $\mathcal{BPP}$ *has deterministic subexponential time algorithms that are correct on all typical cases* (i.e., with respect to any polynomial-time sampleable distribution).

Pseudorandomness with respect to space-bounded distinguishers. As stated in the first paper on the subject of "space-resilient pseudorandom generators" [1],[1] this research direction was inspired by the derandomization result obtained via the use of general-purpose pseudorandom generators. The latter result (necessarily) depends on intractability assumptions, and so the objective was identifying natural classes of algorithms for which derandomization is possible without relying on intractability assumptions (but rather by relying on intractability results that are known for the corresponding classes of distinguishers). This objective was achieved before for the case of constant-depth (randomized) circuits [40], but space-bounded (randomized) algorithms offer a more appealing class that refers to natural algorithms. Fundamentally different constructions of space-resilient pseudorandom generators were given in several works, but are superseded by the two incomparable results mentioned in Section 4.2: Theorem 4.2 (a.k.a Nisan's Generator [41]) and Theorem 4.3 (a.k.a the Nisan–Zuckerman Generator [44]). These two results have been "interpolated" in [5]. Theorem 4.4 ($\mathcal{BPL} \subseteq \mathcal{SC}$) was proved by Nisan [42].

Special Purpose Generators. The various generators presented in Chapter 5 were not inspired by any of the other types of pseudorandom generator (nor even by the generic notion of pseudorandomness). Pairwise-independence generator were

[1] Interestingly, this paper is more frequently cited for the Expander Random Walk technique, which it has introduced.

explicitly suggested in [12] (and are implicit in [10]). The generalization to t-wise independence (for $t \geq 2$) is due to [2]. Small-bias generators were first defined and constructed by Naor and Naor [39], and three simple constructions were subsequently given in [3]. The Expander Random Walk Generator was suggested by Ajtai, Komlos, and Szemerédi [1], who discovered that random walks on expander graphs provide a good approximation to repeated independent attempts to hit any fixed subset of sufficient density (within the vertex set). The analysis of the hitting property of such walks was subsequently improved, culminating in the bound cited in Theorem 5.5, which is taken from [28, Cor. 6.1].

We mention that an alternative treatment of pseudorandomness, which puts more emphasis on the relation between various techniques, is provided in [55].

Bibliography

[1] M. Ajtai, J. Komlos, E. Szemerédi. Deterministic Simulation in LogSpace. In *19th ACM Symposium on the Theory of Computing*, pages 132–140, 1987.

[2] N. Alon, L. Babai and A. Itai. A fast and Simple Randomized Algorithm for the Maximal Independent Set Problem. *J. of Algorithms*, Vol. 7, pages 567–583, 1986.

[3] N. Alon, O. Goldreich, J. Håstad, R. Peralta. Simple Constructions of Almost k-wise Independent Random Variables. *Journal of Random Structures and Algorithms*, Vol. 3, No. 3, pages 289–304, 1992. Preliminary version in *31st FOCS*, 1990.

[4] N. Alon and J.H. Spencer. *The Probabilistic Method*. John Wiley & Sons, Inc., 1992. Second edition, 2000.

[5] R. Armoni. On the derandomization of space-bounded computations. In the proceedings of *Random98*, Springer-Verlag, Lecture Notes in Computer Science (Vol. 1518), pages 49–57, 1998.

[6] H. Attiya and J. Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics*. McGraw-Hill, 1998.

[7] L. Babai, L. Fortnow, N. Nisan and A. Wigderson. BPP has Subexponential Time Simulations unless EXPTIME has Publishable Proofs. *Complexity Theory*, Vol. 3, pages 307–318, 1993.

[8] M. Bellare, O. Goldreich and M. Sudan. Free Bits, PCPs and Non-Approximability – Towards Tight Results. *SIAM Journal on Computing*, Vol. 27, No. 3, pages 804–915, 1998. Extended abstract in *36th FOCS*, 1995.

[9] M. Blum and S. Micali. How to Generate Cryptographically Strong Sequences of Pseudo-Random Bits. *SIAM Journal on Computing*, Vol. 13, pages 850–864, 1984. Preliminary version in *23rd FOCS*, 1982.

[10] L. Carter and M. Wegman. Universal Hash Functions. *Journal of Computer and System Science*, Vol. 18, 1979, pages 143–154.

[11] G.J. Chaitin. On the Length of Programs for Computing Finite Binary Sequences. *Journal of the ACM*, Vol. 13, pages 547–570, 1966.

[12] B. Chor and O. Goldreich. On the Power of Two–Point Based Sampling. *Jour. of Complexity*, Vol 5, 1989, pages 96–106. Preliminary version dates 1985.

[13] T.M. Cover and G.A. Thomas. *Elements of Information Theory.* John Wiley & Sons, Inc., New-York, 1991.

[14] O. Gaber and Z. Galil. Explicit Constructions of Linear Size Superconcentrators. *Journal of Computer and System Science*, Vol. 22, pages 407–420, 1981.

[15] O. Goldreich. A Note on Computational Indistinguishability. *Information Processing Letters*, Vol. 34, pages 277–281, May 1990.

[16] O. Goldreich. *Modern Cryptography, Probabilistic Proofs and Pseudorandomness.* Algorithms and Combinatorics series (Vol. 17), Springer, 1999.

[17] O. Goldreich. *Foundation of Cryptography: Basic Tools.* Cambridge University Press, 2001.

[18] O. Goldreich. *Foundation of Cryptography: Basic Applications.* Cambridge University Press, 2004.

[19] O. Goldreich. *Computational Complexity: A Conceptual Perspective.* Cambridge University Press, 2008.

[20] O. Goldreich, S. Goldwasser, and S. Micali. How to Construct Random Functions. *Journal of the ACM*, Vol. 33, No. 4, pages 792–807, 1986.

[21] O. Goldreich, S. Goldwasser, and A. Nussboim. On the Implementation of Huge Random Objects. In *44th IEEE Symposium on Foundations of Computer Science*, pages 68–79, 2003.

[22] O. Goldreich and L.A. Levin. Hard-core Predicates for any One-Way Function. In *21st ACM Symposium on the Theory of Computing*, pages 25–32, 1989.

[23] O. Goldreich and B. Meyer. Computational Indistinguishability – Algorithms vs. Circuits. *Theoretical Computer Science*, Vol. 191, pages 215–218, 1998. Preliminary version by Meyer in *Structure in Complexity Theory*, 1994.

[24] S. Goldwasser and S. Micali. Probabilistic Encryption. *Journal of Computer and System Science*, Vol. 28, No. 2, pages 270–299, 1984. Preliminary version in *14th STOC*, 1982.

[25] J. Håstad, R. Impagliazzo, L.A. Levin and M. Luby. A Pseudorandom Generator from any One-way Function. *SIAM Journal on Computing*, Volume 28, Number 4, pages 1364–1396, 1999. Preliminary versions by Impagliazzo *et. al.* in *21st STOC* (1989) and Håstad in *22nd STOC* (1990).

[26] R. Impagliazzo and A. Wigderson. P=BPP if E requires exponential circuits: Derandomizing the XOR Lemma. In *29th ACM Symposium on the Theory of Computing*, pages 220–229, 1997.

[27] R. Impagliazzo and A. Wigderson. Randomness vs Time: Derandomization under a Uniform Assumption. *Journal of Computer and System Science*, Vol. 63 (4), pages 672-688, 2001.

[28] N. Kahale. Eigenvalues and Expansion of Regular Graphs. *Journal of the ACM*, Vol. 42 (5), pages 1091–1106, September 1995.

[29] D.E. Knuth. *The Art of Computer Programming*, Vol. 2 (*Seminumerical Algorithms*). Addison-Wesley Publishing Company, Inc., 1969 (first edition) and 1981 (second edition).

[30] A. Kolmogorov. Three Approaches to the Concept of "The Amount Of Information". *Probl. of Inform. Transm.*, Vol. 1/1, 1965.

[31] E. Kushilevitz and N. Nisan. *Communication Complexity*. Cambridge University Press, 1996.

[32] F.T. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes.* Morgan Kaufmann Publishers, San Mateo, CA, 1992.

[33] L.A. Levin. Randomness Conservation Inequalities: Information and Independence in Mathematical Theories. *Information and Control*, Vol. 61, pages 15–37, 1984.

[34] M. Li and P. Vitanyi. *An Introduction to Kolmogorov Complexity and its Applications.* Springer Verlag, August 1993.

[35] A. Lubotzky, R. Phillips, and P. Sarnak. Ramanujan Graphs. *Combinatorica*, Vol. 8, pages 261–277, 1988.

[36] G.A. Margulis. Explicit Construction of Concentrators. *Prob. Per. Infor.*, Vol. 9 (4), pages 71–80, 1973 (in Russian). English translation in *Problems of Infor. Trans.*, pages 325–332, 1975.

[37] P.B. Miltersen and N.V. Vinodchandran. Derandomizing Arthur-Merlin Games using Hitting Sets. *Computational Complexity*, Vol. 14 (3), pages 256–279, 2005. Preliminary version in *40th FOCS*, 1999.

[38] R. Motwani and P. Raghavan. *Randomized Algorithms.* Cambridge University Press, 1995.

[39] J. Naor and M. Naor. Small-bias Probability Spaces: Efficient Constructions and Applications. *SIAM Journal on Computing*, Vol 22, 1993, pages 838–856. Preliminary version in *22nd STOC*, 1990.

[40] N. Nisan. Pseudorandom bits for constant depth circuits. *Combinatorica*, Vol. 11 (1), pages 63–70, 1991.

[41] N. Nisan. Pseudorandom Generators for Space Bounded Computation. *Combinatorica*, Vol. 12 (4), pages 449–461, 1992. Preliminary version in *22nd STOC*, 1990.

[42] N. Nisan. $\mathcal{RL} \subseteq \mathcal{SC}$. *Computational Complexity*, Vol. 4, pages 1-11, 1994. Preliminary version in *24th STOC*, 1992.

[43] N. Nisan and A. Wigderson. Hardness vs Randomness. *Journal of Computer and System Science*, Vol. 49, No. 2, pages 149–167, 1994. Preliminary version in *29th FOCS*, 1988.

[44] N. Nisan and D. Zuckerman. Randomness is Linear in Space. *Journal of Computer and System Science*, Vol. 52 (1), pages 43–52, 1996. Preliminary version in *25th STOC*, 1993.

[45] N. Pippenger and M.J. Fischer. Relations among complexity measures. *Journal of the ACM*, Vol. 26 (2), pages 361–381, 1979.

[46] A.R. Razborov and S. Rudich. Natural Proofs. *Journal of Computer and System Science*, Vol. 55 (1), pages 24–35, 1997. Preliminary version in *26th STOC*, 1994.

[47] O. Reingold. Undirected ST-Connectivity in Log-Space. In *37th ACM Symposium on the Theory of Computing*, pages 376–385, 2005.

[48] O. Reingold, S. Vadhan, and A. Wigderson. Entropy Waves, the Zig-Zag Graph Product, and New Constant-Degree Expanders and Extractors. *Annals of Mathematics*, Vol. 155 (1), pages 157–187, 2001. Preliminary version in *41st FOCS*, pages 3–13, 2000.

[49] M. Saks and S. Zhou. $\mathrm{BP_HSPACE}(S) \subseteq \mathrm{DSPACE}(S^{3/2})$. *Journal of Computer and System Science*, Vol. 58 (2), pages 376–403, 1999. Preliminary version in *36th FOCS*, 1995.

[50] R. Shaltiel. Recent Developments in Explicit Constructions of Extractors. In *Current Trends in Theoretical Computer Science: The Challenge of the New Century, Vol 1: Algorithms and Complexity*, World scietific, 2004. (Editors: G. Paun, G. Rozenberg and A. Salomaa.) Preliminary version in *Bulletin of the EATCS 77*, pages 67–95, 2002.

[51] C.E. Shannon. A mathematical theory of communication. *Bell Sys. Tech. Jour.*, Vol. 27, pages 623–656, 1948.

[52] R.J. Solomonoff. A Formal Theory of Inductive Inference. *Information and Control*, Vol. 7/1, pages 1–22, 1964.

[53] L. Trevisan. Extractors and Pseudorandom Generators. *Journal of the ACM*, Vol. 48 (4), pages 860–879, 2001. Preliminary version in *31st STOC*, 1999.

[54] C. Umans. Pseudo-random generators for all hardness. *Journal of Computer and System Science*, Vol. 67 (2), pages 419–440, 2003.

[55] S. Vadhan. *Lecture Notes for CS 225: Pseudorandomness*, Spring 2007. Available from `http://www.eecs.harvard.edu/~salil`.

[56] L.G. Valiant. A theory of the learnable. *CACM*, Vol. 27/11, pages 1134–1142, 1984.

[57] I. Wegener. *Branching Programs and Binary Decision Diagrams – Theory and Applications*. SIAM Monographs on Discrete Mathematics and Applications, 2000.

[58] A. Wigderson. The amazing power of pairwise independence. In *26th ACM Symposium on the Theory of Computing*, pages 645–647, 1994.

[59] A.C. Yao. Theory and Application of Trapdoor Functions. In *23rd IEEE Symposium on Foundations of Computer Science*, pages 80–91, 1982.