

Targeted Pseudorandom Generators, Simulation Advice Generators, and Derandomizing Logspace

William M. Hoza* Chris Umans†

April 11, 2017

Abstract

Assume that for every derandomization result for logspace algorithms, there is a pseudorandom generator strong enough to nearly recover the derandomization by iterating over all seeds and taking a majority vote. We prove under a precise version of this assumption that $\mathbf{BPL} \subseteq \bigcap_{\alpha>0} \mathbf{DSPACE}(\log^{1+\alpha} n)$.

We strengthen the theorem to an equivalence by considering two generalizations of the concept of a pseudorandom generator against logspace. A *targeted pseudorandom generator* against logspace takes as input a short uniform random seed *and* a finite automaton; it outputs a long bitstring that looks random to that particular automaton. A *simulation advice generator* for logspace stretches a small uniform random seed into a long advice string; the requirement is that there is some logspace algorithm that, given a finite automaton and this advice string, simulates the automaton reading a long uniform random input. We prove that

$$\bigcap_{\alpha>0} \mathbf{promise-BPSPACE}(\log^{1+\alpha} n) = \bigcap_{\alpha>0} \mathbf{promise-DSPACE}(\log^{1+\alpha} n)$$

if and only if for every targeted pseudorandom generator against logspace, there is a simulation advice generator for logspace with similar parameters.

Finally, we observe that in a certain *uniform* setting (namely, if we only worry about sequences of automata that can be generated in logspace), targeted pseudorandom generators against logspace *can* be transformed into simulation advice generators with similar parameters.

1 Introduction

1.1 Derandomization vs. pseudorandom generators

The *derandomization* program of complexity theory consists of trying to deterministically simulate whole classes of randomized algorithms without significant loss in efficiency. For example, we would like to prove that $\mathbf{P} = \mathbf{BPP}$, $\mathbf{NP} = \mathbf{AM}$, and $\mathbf{L} = \mathbf{BPL}$. The main strategy for derandomization is to design an efficient *pseudorandom generator*. A natural question is whether this strategy is without loss of generality. That is, does derandomization always imply a pseudorandom generator that is strong enough to recover that very same derandomization? This question appears to have

*Department of Computer Science, University of Texas at Austin, whoza@utexas.edu. This research was mostly conducted while the author was an undergraduate student at the California Institute of Technology.

†Department of Computing and Mathematical Sciences, California Institute of Technology, umans@cms.caltech.edu

first been investigated by Fortnow [For01], who gave an oracle separation between pseudorandom generators and derandomization in the \mathbf{P} vs. \mathbf{BPP} setting.

Nevertheless, for both \mathbf{NP} vs. \mathbf{AM} and \mathbf{P} vs. \mathbf{BPP} , there are indeed known constructions of pseudorandom generators from derandomization assumptions. Most such constructions come from the *hardness vs. randomness* paradigm. The idea is to show that derandomization assumptions imply *hardness* results (such as circuit lower bounds). There is a large body of literature [Yao82, BM84, NW94, IW97, IW98, HILL99, KvM02, Uma03] showing how, in turn, to construct pseudorandom generators from hardness. Typically, the constructed pseudorandom generator is not strong enough to recover the original derandomization assumption (e.g. [IKW02, KI04, AGHK11, KvMS12, Wil13]) but some results are known that establish exact equivalence between certain sorts of derandomizations and certain sorts of pseudorandom generators (see [AvM12]). Goldreich has followed another approach [Gol11a, Gol11b] to construct pseudorandom generators from derandomization assumptions in the \mathbf{BPP} setting. His approach does not directly involve establishing hardness results on the way; instead, he shows how to derandomize the standard nonconstructive existence proof for pseudorandom generators by a reduction to decision problems.

The subject of this paper is \mathbf{L} vs. \mathbf{BPL} . In this setting, there are no known constructions of pseudorandom generators from generic derandomization assumptions. Further, the question of whether derandomization is equivalent to pseudorandom generators is especially well-motivated in this setting, because nontrivial derandomizations and pseudorandom generators have been unconditionally constructed – and there is a significant *gap*. Iterating over all seeds of the best known pseudorandom generator, by Nisan [Nis92b], merely proves that $\mathbf{BPL} \subseteq \mathbf{DSPACE}(\log^2 n)$ (which can also be proven by recursive matrix exponentiation). But the best known derandomization, the celebrated Saks-Zhou theorem [SZ99], states that $\mathbf{BPL} \subseteq \mathbf{DSPACE}(\log^{3/2} n)$.

In this work, we show that (informally) *if* for every derandomization of logspace algorithms, there is a pseudorandom generator strong enough to nearly recover the derandomization by iterating over all seeds, then $\mathbf{BPL} \subseteq \bigcap_{\alpha > 0} \mathbf{DSPACE}(\log^{1+\alpha} n)$. So establishing the *equivalence* of derandomization and pseudorandom generators *would itself* yield a strong derandomization of \mathbf{BPL} .

Our result can be viewed pessimistically as showing that it will be challenging to establish equivalence of derandomization and pseudorandom generators in the \mathbf{BPL} setting. But it can also be viewed optimistically as giving a *road map* for proving that $\mathbf{BPL} \subseteq \bigcap_{\alpha > 0} \mathbf{DSPACE}(\log^{1+\alpha} n)$. From this second viewpoint, our result should be compared to other known results that give interesting sufficient conditions for derandomizing logspace:

- Klivans and van Melkebeek showed [KvM02] that if some language in $\mathbf{DSPACE}(n)$ requires branching programs of size $2^{\Omega(n)}$, then there is a pseudorandom generator strong enough to prove $\mathbf{L} = \mathbf{BPL}$. While interesting, this result does not seem to provide a viable road map for derandomizing logspace, because the strong hardness assumption seems to be far beyond current understanding.
- Reingold, Trevisan, and Vadhan showed [RTV06] that if there is an efficient pseudorandom walk generator for *regular* digraphs, then $\mathbf{L} = \mathbf{RL}$. This result *can* be reasonably thought of as giving a road map for derandomizing logspace; the result is particularly tantalizing because in the same work, they actually *did* construct a pseudorandom walk generator for *consistently labeled* regular digraphs. Alas, in the decade since these results were announced, nobody has been able to close the gap.

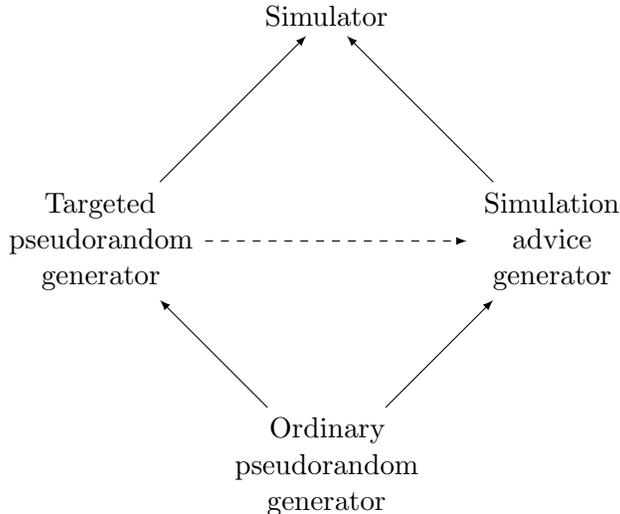


Figure 1: The four types of derandomization that we consider. A solid arrow from A to B indicates that a derandomization of type A trivially implies a derandomization of type B . Our main result is that the implication indicated by the dashed arrow is equivalent to the statement that $\bigcap_{\alpha>0} \mathbf{promise-BPSPACE}(\log^{1+\alpha} n) = \bigcap_{\alpha>0} \mathbf{promise-DSPACE}(\log^{1+\alpha} n)$.

We view our result as promising, considering that there are already established techniques for proving equivalence of derandomization and pseudorandom generators. We consider it conceivable that those techniques can be “ported” to the \mathbf{L} vs. \mathbf{BPL} setting. The previously mentioned result of [KvM02] may be a first step in that direction. To put it another way, for decades, researchers have been trying to design strong pseudorandom generators for \mathbf{BPL} ; our result shows that researchers can feel free to *make derandomization assumptions* while trying to design those pseudorandom generators, which could make the task significantly easier.

1.2 Four types of derandomization

In fact, our main result is considerably stronger than what we have said so far. To explicate our main result, it is useful to distinguish between *four* types of derandomizations of logspace. (See Figure 1.) First, the most generic type of derandomization is a *simulator* for logspace. This is an algorithm that takes as input a finite automaton Q , a start state q , and a short uniform seed x ; it outputs a state $\text{Sim}(Q, q, x)$ whose distribution is close to the distribution of final states that Q would be in were it to read a long uniform random string. (Finite automata provide a simple nonuniform model of space-bounded computation; each state of a w -state automaton corresponds to a configuration of a $(\log w)$ -space Turing machine.)

The second type of derandomization, which should be familiar, is a *pseudorandom generator* against logspace. A pseudorandom generator has two key features that distinguish it from a generic simulator:

- **Input.** The pseudorandom generator does not get to see the “source code” of the algorithm being simulated, i.e. it does not get (Q, q) as part of its input.

- Output. The pseudorandom generator produces a long string for the automaton to read, whereas a simulator merely produces the final state of the automaton.

The third and fourth types of derandomization that we will consider generalize the concept of a pseudorandom generator by relaxing these two features respectively. The third type of derandomization, a *targeted pseudorandom generator*, gets as input a finite automaton Q , a start state q , and a short uniform seed x ; it outputs a long bitstring $\text{Gen}(Q, q, x)$ that looks random to that particular automaton Q when it starts in that particular state q . (Goldreich [Gol11b] coined the term “targeted pseudorandom generator” in the context of \mathbf{P} vs. \mathbf{BPP} , where the generator gets a Boolean circuit as its auxiliary input. In the \mathbf{L} vs. \mathbf{BPL} setting, targeted pseudorandom generators have been studied before; see e.g. [Nis92a, RR99].) The fourth type of derandomization, a *simulation advice generator*, stretches a short uniform seed x into a long advice string $\text{Gen}(x)$; the requirement is that there is a deterministic logspace algorithm S such that $\text{Sim}(Q, q, x) \stackrel{\text{def}}{=} S(Q, q, \text{Gen}(x))$ is a simulator for logspace. To the best of our knowledge, we are the first to study simulation advice generators.

Our main result is that

$$\bigcap_{\alpha > 0} \mathbf{promise-BSPACE}(\log^{1+\alpha} n) = \bigcap_{\alpha > 0} \mathbf{promise-DSPACE}(\log^{1+\alpha} n) \quad (1)$$

if and only if for every targeted pseudorandom generator against logspace, there is a simulation advice generator with similar parameters. (The precise statement is in Section 2.) Here, $\mathbf{promise-BSPACE}(s(n))$ is the set of promise problems decidable by probabilistic space- $s(n)$ Turing machines that always halt and that have error probability at most $1/3$; $\mathbf{promise-DSPACE}(s(n))$ is its deterministic analog.

Additionally, in Section 7, we observe that targeted pseudorandom generators against logspace *can* be transformed into simulation advice generators for logspace if we move to the *uniform setting*, i.e. we only worry about sequences of automata that can be generated in logspace. This is almost immediate from the definitions, but it illustrates how much easier it is to construct simulation advice generators than it is to construct pseudorandom generators.

1.3 Proof techniques

One direction of our main result is easy. Under the assumption that Equation 1 holds, simulation advice generators are uninteresting objects that can be constructed for trivial reasons. The main content of the theorem is the reverse direction.

The proof of the harder direction is by extending the techniques of Saks and Zhou [SZ99]. The way Saks and Zhou originally presented their result is that they used specific properties of Nisan’s pseudorandom generator [Nis92b] to design a space-efficient algorithm for approximate matrix exponentiation by reusing parts of the seed. Later, Armoni [Arm98] constructed a pseudorandom generator that is better than Nisan’s for fooling low-randomness algorithms, and using Zuckerman’s oblivious sampler [Zuc97], he adapted the Saks-Zhou algorithm to use his generator instead of Nisan’s, giving a better derandomization of such algorithms.

In Section 4, we show that with Armoni’s ideas, the Saks-Zhou construction can instead be formulated as a *transformation on simulators*. Roughly: Starting from a simulator that uses an s -bit seed to simulate m_0 steps of a w -state automaton, given a parameter m , the Saks-Zhou-Armoni (SZA) transformation produces a new simulator that uses an $O\left(s + \frac{(\log m)(\log w)}{\log m_0}\right)$ -bit seed

to simulate m steps of a w -state automaton. We consider this reformulation to be interesting in its own right, as it clarifies the power of Saks-Zhou rounding.

A simple, tempting idea is to start with a weak simulator and apply the SZA transformation t times for some large constant t . In iteration i , choose $m = 2^{\log^{i/t} w}$. Then we end up with a simulator with $m = w$ (large enough to simulate randomized space-bounded algorithms), and the seed length is only $O(\log^{1+1/t} w)$. But unfortunately, the space complexity blows up with each application of the SZA transformation.

Because of the recursive structure of the SZA transformation, the blowup can be avoided as long as the SZA transformation is only applied to simulators obtained from simulation advice generators. So to prove the harder direction of our main result, we cycle between three transformations:

1. Our assumption, which transforms a targeted pseudorandom generator into a simulation advice generator. (This “transformation” is not necessarily effective.)
2. The SZA transformation, which we now think of as transforming a simulation advice generator into a simulator.
3. A simple transformation based on the method of conditional probabilities, which transforms a simulator into a targeted pseudorandom generator.

The SZA transformation substantially increases the number of steps being simulated. For each of the three transformations, we incur only mild degradation in the seed length, space complexity, etc. Hence, overall, each cycle significantly increases the output length of our targeted pseudorandom generator without degrading the other parameters too much. By iterating the cycle a large constant number of times, we end up with a generator strong enough to collapse $\bigcap_{\alpha>0} \mathbf{promise-BPSPACE}(\log^{1+\alpha} n)$ to $\bigcap_{\alpha>0} \mathbf{promise-DSPACE}(\log^{1+\alpha} n)$.

2 Formal statement of main result

Let $[w]$ denote the set $\{1, 2, \dots, w\}$. Let U_n denote the uniform distribution on $\{0, 1\}^n$. For two probability distributions μ, μ' on the same measurable space, write $\mu \sim_\varepsilon \mu'$ to mean that the total variation distance between μ and μ' is at most ε .

Definition 1. If \mathbf{A} is a set of functions $\{0, 1\}^m \rightarrow [w]$, we say that a function $\text{Sim} : \mathbf{A} \times \{0, 1\}^s \rightarrow [w]$ is an ε -*simulator* for \mathbf{A} if for every $f \in \mathbf{A}$, we have $\text{Sim}(f, U_s) \sim_\varepsilon f(U_m)$.

Definition 2. If \mathbf{A} is a set of functions $\{0, 1\}^m \rightarrow [w]$, we say that a function $\text{Gen} : \mathbf{A} \times \{0, 1\}^s \rightarrow \{0, 1\}^m$ is a *targeted ε -pseudorandom generator* against \mathbf{A} if the function $\text{Sim}(f, x) \stackrel{\text{def}}{=} f(\text{Gen}(f, x))$ is an ε -simulator for \mathbf{A} .

The standard definition of a pseudorandom generator is the special case where $\text{Gen}(f, x)$ does not depend on f .

Definition 3. A (w, d) -*automaton* is a function $Q : [w] \times \{0, 1\}^d \rightarrow [w]$. If Q_1 is a (w, d_1) -automaton and Q_2 is a (w, d_2) -automaton, then $Q_2 Q_1$ is the $(w, d_1 + d_2)$ -automaton defined by

$$(Q_2 Q_1)(q; x, y) = Q_2(Q_1(q; x); y).$$

Let $\mathbf{Q}_{w,d}^m$ be the set of all functions $\{0, 1\}^{md} \rightarrow [w]$ of the form $x \mapsto Q^m(q; x)$ where Q is a (w, d) -automaton.

Parameter	Interpretation
w	Number of states in the automaton
d	Number of bits the automaton reads in each step
m	Number of steps the automaton takes
ε	Simulation error, in total variation distance
s	Seed length
a	Number of advice bits

Figure 2: A summary of the parameters of the targeted pseudorandom generators, simulation advice generators, and simulators that we study. Each family of generators/simulators is indexed by w , and the other parameters are functions of w .

In words, $\mathbf{Q}_{w,d}^m$ is the set of functions computed by letting a (w, d) -automaton run for m steps and observing its final state. An element of $\mathbf{Q}_{w,d}^m$ can be specified by a pair (Q, q) , and this is how it will be presented to simulators and targeted pseudorandom generators in our theorem statements.

Definition 4. Suppose that for each w , $\text{Gen}_w : \{0, 1\}^s \rightarrow \{0, 1\}^a$ is a function, and $\mathbf{A}_w \subseteq \mathbf{Q}_{w,d}^m$, where s, a, d, m are functions of w . We say that Gen_w is¹ an ε -simulation advice generator for \mathbf{A}_w if there is some deterministic logspace algorithm S such that the function $\text{Sim}(Q, q, x) \stackrel{\text{def}}{=} S(Q, q, \text{Gen}_w(x))$ is an ε -simulator for \mathbf{A}_w .

Note that S 's space bound is logarithmic in terms of its input length, i.e. it may use $O(d + \log w + \log a)$ bits of space. It is desirable for m to be *big* and s, a, ε to be *small*. E.g. as long as $a \leq \text{poly}(w, 2^d)$, it contributes nothing to the asymptotic space complexity of S . To explicate the definition, we give several examples of where simulation advice generators might come from:

1. Any (standard, non-targeted) ε -pseudorandom generator Gen_w against $\mathbf{Q}_{w,d}^m$ is also an ε -simulation advice generator for $\mathbf{Q}_{w,d}^m$. The associated algorithm $S(Q, q, y)$ computes $Q^m(q; y)$ where y is the output of Gen_w . This can be done in logspace by storing the current state of Q and the current d -bit chunk of y .
2. Suppose there is some logspace ε -simulator for $\mathbf{Q}_{w,d}^m$ with seed length s . Then the identity function on $\{0, 1\}^s$ is an ε -simulation advice generator for $\mathbf{Q}_{w,d}^m$. (So under the assumption that **promise-L** = **promise-BPL**, simulation advice generators are only interesting for extreme values of parameters.)
3. Suppose Gen_w is a targeted ε -pseudorandom generator against $\mathbf{Q}_{w,d}^m$ of the form $\text{Gen}_w(Q, q, x) = G(\text{Compress}(Q, q, x), x)$, where Compress is computable in $O(d + \log w)$ space and outputs b bits. Let $\text{Gen}'_w(x)$ be x concatenated with the truth table T of $G(\cdot, x)$. Then Gen'_w is an ε -simulation advice generator for $\mathbf{Q}_{w,d}^m$ with output length $a = s + m2^b$. The associated algorithm $S(Q, q, x, T)$ computes $c = \text{Compress}(Q, q, x)$, referring to its advice tape for access to x . Then, S looks up the value $y = G(c, x)$ in the T portion of its advice tape and computes $Q^m(q; y)$.

¹Strictly speaking, this is a property of the family $\{\text{Gen}_w\}$, not of the individual function. There should be just one S for the whole family, and ε is a function of w .

4. Suppose Sim is an ε -simulator for $\mathbf{Q}_{w,d}^m$ that perhaps uses much more than logspace, but that, each time it reads from Q or q , first *erases* all but $O(d + \log w)$ bits. If c is a configuration of $\text{Sim}(Q, q, x)$ in which Sim just read from Q or q , then let $f(c, x)$ be the configuration that $\text{Sim}(Q, q, x)$ will next be in when it is about to read from Q or q . Let $\text{Gen}_w(x)$ be the truth table of $f(\cdot, x)$. Then Gen_w is an ε -simulation advice generator for $\mathbf{Q}_{w,d}^m$ with output length $a \leq \text{poly}(w, 2^d)$. The associated algorithm $\text{S}(Q, q, \text{Gen}_w(x))$ simulates $\text{Sim}(Q, q, x)$. To update the simulation's configuration, S alternates between reading a bit from (Q, q) and using its advice tape.

Suppose $\{\text{F}_w\}$ is a family where F_w is a simulator for, a simulation advice generator for, or a targeted pseudorandom generator against $\mathbf{Q}_{w,d}^m$, with seed length $s(w)$. For convenience, we will say that the family is *efficiently computable* if $s(w)$ is space constructible and given (w, X) , $\text{F}_w(X)$ can be computed in deterministic space $O(s(w))$. We will often speak of an individual function F_w being efficiently computable when the family is clear.

We now formally state our main result. In Condition 2, η, σ, μ are the parameters of the targeted pseudorandom generator. The last parameter γ quantifies the extent to which the derandomization degrades when the targeted pseudorandom generator is replaced with a simulation advice generator.

Theorem 1. *The following are equivalent.*

1.

$$\bigcap_{\alpha > 0} \mathbf{promise-BPSPACE}(\log^{1+\alpha} n) = \bigcap_{\alpha > 0} \mathbf{promise-DSPACE}(\log^{1+\alpha} n).$$

2. *For any constant $\mu \in [0, 1]$, for any sufficiently small constants $\sigma > \eta > 0$, and for any constant $\gamma > 0$, the following holds. Suppose there is a family $\{\text{Gen}_w\}$, where Gen_w is an efficiently computable targeted ε -pseudorandom generator against $\mathbf{Q}_{w,1}^m$ with seed length s , satisfying*

$$s \leq O(\log^{1+\sigma} w), \quad \log(1/\varepsilon) = \log^{1+\eta} w, \quad \log m \geq \log^\mu w.$$

Then there is another family $\{\text{Gen}'_w\}$, where Gen'_w is an efficiently computable ε' -simulation advice generator for $\mathbf{Q}_{w,1}^{m'}$ with seed length s' and output length a' , satisfying

$$s' \leq O(\log^{1+\sigma+\gamma} w), \quad \log(1/\varepsilon') = \log^{1+\eta-\gamma} w, \quad \log m' \geq \log^{\mu-\gamma} w, \quad \log a' \leq O(\log^{1+\eta+\gamma} w).$$

3 The implicit oracle model

Toward proving Theorem 1, we introduce a model of space-bounded oracle algorithms that seemingly does not appear in the literature. Our new oracle model (the “implicit oracle model”) gives a convenient framework for expressing the SZA result as a transformation on simulators and clarifies the effect on the simulator's space complexity when the SZA transformation is iterated.

The implicit oracle model is similar to Wilson's oracle stack model [Wil88], and it is appropriate for the situation where the algorithm doesn't have room to write down the entire query string, but it is ready to provide the oracle with random access to the query string (possibly by making more oracle queries.)

Definition 5. Fix a set $A \subseteq \{0, 1\}^*$. Giving an algorithm *implicit oracle access* to A allows the algorithm to interface with an oracle in the following ways:

- The algorithm can *invoke* the oracle, which passes control to the oracle.
- The oracle can *read position* $j \in \mathbb{N}$ by giving j to the algorithm. This passes control back to the algorithm. We associate this read with the most recent unresolved invocation.
- The algorithm can give the oracle a *query value* $b \in \{0, 1, \perp\}$. This passes control back to the oracle and *resolves* the most recent unresolved read.
- The oracle can give the algorithm a boolean *answer value*. This passes control back to the algorithm and *resolves* the most recent unresolved invocation.

The oracle is guaranteed to behave as follows: Fix any $x \in \{0, 1\}^*$. Suppose that for some invocation, when the oracle reads position j , the algorithm specifies value x_j (where we interpret $x_j = \perp$ for $j > |x|$.) Then the oracle will make finitely many reads and give the answer value corresponding to whether $x \in A$, and every read will be of a position $j \leq |x| + 1$.

We extend the definition by saying that we give an algorithm implicit oracle access to a *function* $f : \{0, 1\}^* \rightarrow \{0, 1\}^*$ to mean that we give the algorithm implicit oracle access to the set $A = \{(x, b, 0) : |f(x)| \leq b\} \cup \{(x, b, 1) : f(x)_b = 1\}$.

Wilson’s oracle stack model is equivalent to the implicit oracle model with the additional restriction that the oracle is guaranteed to read its input from left to right.

Ultimately, we will only use the implicit oracle model in intermediate steps of our proof; for our final algorithm, we will “plug in” actual algorithms in place of the oracle. The next lemma says what happens to space complexity when this actual algorithm is plugged in.

Lemma 1. *Suppose $\text{Gen} : \{0, 1\}^s \rightarrow \{0, 1\}^a$ is an efficiently computable ε -simulation advice generator for $\mathbf{Q}_{w,d}^m$, and let Sim be the corresponding simulator. Suppose Alg is an implicit oracle algorithm and x is an input such that during the execution of $\text{Alg}^{\text{Sim}}(w, x)$, Alg uses s' bits of space, and at any moment, there are at most u unresolved oracle invocations, and there are at most v unresolved reads of seeds. Then $\text{Alg}^{\text{Sim}}(w, x)$ can be computed (by a non-oracle algorithm) in space $O(s' + s \cdot (v + 1) + u \cdot (d + \log w + \log a))$.*

Proof. Recall that Sim is of the form $\text{Sim}(Q, q, x) = \text{S}(Q, q, \text{Gen}(x))$. Naturally, just simulate Alg , replacing its oracle queries with computations of Sim . The space needed is s' for the computation of Alg , plus $O(d + \log w + \log a)$ for each unresolved execution of S , plus $O(s)$ for each unresolved execution of Gen . The number of unresolved executions of S is precisely u . The number of unresolved executions of Gen is at most $v + 1$, because while an instance of Sim is in the process of computing Gen , that instance never queries the (Q, q) portion of its input. \square

4 The SZA transformation

Formulating the Saks-Zhou construction as a transformation on simulators is not technically challenging. A (w, d) -automaton with fail state² is a $(w + 1, d)$ -automaton such that $Q(w + 1; y) = w + 1$

²This is equivalent to the definition of a “finite state machine of type (w, d) ” in [SZ99] or that of a “ (w, d) -automaton” in [CCvM06].

for all y . (We think of $w + 1$ as the “fail state”.) Let $\tilde{\mathbf{Q}}_{w,d}^m$ be the set of all functions of the form $x \mapsto Q^m(q; x)$ where Q is a (w, d) -automaton with fail state. When we give an algorithm (implicit) oracle access to an ε -simulator for $\tilde{\mathbf{Q}}_{w,d}^m$ with seed length s , it is understood that the algorithm can query for the parameters w, d, m, ε, s as well as interacting with the oracle in the usual way.

Theorem 2. *There is a constant $c \in \mathbb{N}$ and a deterministic implicit oracle algorithm **SZA** with the following properties. Pick $w \in \mathbb{N}, \varepsilon > 0$ and let $d = \lceil c \log(w/\varepsilon) \rceil$. Suppose **Sim** is an ε -simulator for $\tilde{\mathbf{Q}}_{w,d}^{m_0}$ with seed length $s \leq m_0 \leq w$. Then*

1. *For any $m \in \mathbb{N}$, there is some $m' \geq m$ such that $\mathbf{SZA}_m^{\text{Sim}}$ is a $(12m\varepsilon)$ -simulator for $\tilde{\mathbf{Q}}_{w,d}^{m'}$. (Here m is an input to **SZA**; we write it as a subscript merely to separate it from the usual simulator inputs.)*
2. *At any moment in the execution of $\mathbf{SZA}_m^{\text{Sim}}$, there are at most $u \stackrel{\text{def}}{=} \lceil (\log m)/(\log m_0) \rceil$ unresolved oracle invocations, and there is at most one unresolved read of the seed of **Sim**.*
3. *The seed length and space complexity of $\mathbf{SZA}_m^{\text{Sim}}$ are both $O(s + u \log(w/\varepsilon))$.*

To illustrate the theorem statement, we demonstrate how to recover the original Saks-Zhou result of [SZ99]. Let $\text{Gen} : \{0, 1\}^s \rightarrow \{0, 1\}^{m_0}$ be the (non-targeted) efficiently computable ε -pseudorandom generator against $\tilde{\mathbf{Q}}_{w,d}^{m_0}$ of [INW94, Theorem 3] with $m_0 = 2^{\sqrt{\log w}}$, $\varepsilon = 1/(6 \cdot 12w)$, and $s \leq O(\log^{3/2} w)$. Let **Sim** be the corresponding simulator. Then $\mathbf{SZA}_w^{\text{Sim}}$ is a $(1/6)$ -simulator for $\tilde{\mathbf{Q}}_{w,d}^{m'}$ for some $m' \geq w$, and hence it can be used to simulate **BPL** (by ensuring that all transitions from the halting configurations are self loops.) The parameter u is $O(\sqrt{\log w})$, and hence the seed length and space usage of $\mathbf{SZA}_w^{\text{Sim}}$ are both $O(\log^{3/2} w)$. By Lemma 1, the space needed to simulate $\mathbf{SZA}_w^{\text{Sim}}$ by a non-oracle algorithm is $O(\log^{3/2} w)$. Iterating over all seeds proves $\mathbf{BPL} \subseteq \mathbf{DSPACE}(\log^{3/2} n)$, since the number of configurations of a logspace Turing machine on a length n input is $w \leq \text{poly}(n)$.

The rest of this section is the proof of Theorem 2. All of the ideas in the proof are already present in [SZ99] and [Arm98]. Our main contributions in this section are the *formulation and statement* of Theorem 2, which enable us to derive the consequence expressed in Theorem 1.

4.1 Randomness efficient samplers

The first step to proving Theorem 2 is an observation by Armoni [Arm98]. Let **NisGen** denote Nisan’s generator. Saks and Zhou used a special feature of **NisGen**. The special feature is that the seed can be split into two parts x, z with $z \leq O(\log(w/\varepsilon))$ such that for any particular automaton Q , for most values of x , $\text{NisGen}(x, \cdot)$ is a good pseudorandom generator for Q . (Namely, we can let x be the sequence of hash functions and z be the input to those hash functions.) Armoni observed that *any* pseudorandom generator can be made to have this feature just by precomposing with an *averaging sampler*. We give here the appropriate notion of averaging samplers for $[w]$ -valued functions:

Definition 6. Fix $\text{Samp} : \{0, 1\}^\ell \times \{0, 1\}^d \rightarrow \{0, 1\}^s$. For a function $f : \{0, 1\}^s \rightarrow [w]$, we say that a string $x \in \{0, 1\}^\ell$ is δ -good for f if $f(\text{Samp}(x, U_d)) \sim_\delta f(U_s)$. We say that **Samp** is an *averaging* (δ, γ) -sampler for $[w]$ -valued functions if for every $f : \{0, 1\}^s \rightarrow [w]$,

$$\Pr_{x \sim U_\ell} [x \text{ is } \delta\text{-good for } f] \geq 1 - \gamma.$$

We need a space-efficient averaging sampler with good parameters. Armoni used Zuckerman’s averaging sampler [Zuc97], but Zuckerman’s sampler breaks down for extremely small values of δ . Therefore, to get a slightly more general result, we use the GUV extractor [GUV09], or rather a space-optimized version by Kane, Nelson, and Woodruff [KNW08]. It is standard that extractors are good samplers; the following lemma expresses the parameters achieved by the space-optimized GUV extractor when it is viewed as a sampler for $[w]$ -valued functions:

Lemma 2. *For all $s, w \in \mathbb{N}$ and all $\delta, \gamma > 0$, there is an averaging (δ, γ) -sampler for $[w]$ -valued functions $\text{Samp} : \{0, 1\}^\ell \times \{0, 1\}^d \rightarrow \{0, 1\}^s$ with*

$$\ell \leq O(s) + \log(w/\gamma)$$

and

$$d \leq O(\log s + \log w + \log(1/\delta) + \log \log(1/\gamma)),$$

where $\text{Samp}(x, y)$ can be computed in $O(s + \log(w/\gamma))$ space.

Proof. Let $\ell = 2s + 1 + \log(w/\gamma)$. By [KNW08, Theorem A.14], there is a $(2s, 2\delta/w)$ -extractor $\text{Samp} : \{0, 1\}^\ell \times \{0, 1\}^d \rightarrow \{0, 1\}^s$ with $d \leq O(\log \ell + \log(w/\delta))$, which is

$$O(\log s + \log w + \log(1/\delta) + \log \log(1/\gamma))$$

as claimed, such that $\text{Samp}(x, y)$ can be computed in $O(\ell + \log(w/\delta))$ space, which is $O(s + \log(w/\delta))$ space as claimed.

All that remains is to prove correctness. Fix $f : \{0, 1\}^s \rightarrow [w]$. Say $x \in \{0, 1\}^\ell$ is *good for f with respect to $z \in [w]$* if

$$|\Pr[f(\text{Samp}(x, U_d)) = z] - \Pr[f(U_s) = z]| \leq 2\delta/w.$$

By [Zuc97, Proposition 2.7] (or rather its proof), for each $z \in [w]$,

$$\Pr_{x \sim U_\ell} [x \text{ is good for } f \text{ with respect to } z] \geq 1 - 2^{-\log(w/\gamma)} = 1 - \gamma/w.$$

Therefore, by the union bound over the w different values of z , the probability that a uniform random x is good for f with respect to *every* $z \in [w]$ simultaneously is at least $1 - \gamma$. For such an x , the ℓ_1 distance between $f(\text{Samp}(x, U_d))$ and $f(U_s)$ is at most 2δ . Total variation distance is half ℓ_1 distance, so such an x is δ -good for f , completing the proof. \square

4.2 The snap operation

At the heart of the SZA transformation is a randomized rounding operation that we will call **Snap**. This operation slightly perturbs a given automaton with fail state. The basic feature of this perturbation is that if $Q \approx Q'$, then with high probability, $\text{Snap}(Q) = \text{Snap}(Q')$. This phenomenon (which we will make rigorous in Lemma 5) is reminiscent of “snapping to a grid”, hence the name.

A *substochastic d -matrix* is a square matrix M filled with nonnegative multiples of 2^{-d} such that for every q , $\sum_r M_{qr} \leq 1$. A (w, d) -automaton with fail state Q has a *transition probability matrix* $\mathcal{M}(Q)$, a $w \times w$ substochastic d -matrix defined by

$$\mathcal{M}(Q)_{qr} = \Pr_{z \in \{0, 1\}^d} [Q(q; z) = r].$$

Conversely, from a $w \times w$ substochastic d -matrix M , we define a *canonical automaton with fail state* $\mathcal{Q}(M)$ by identifying $\{0, 1\}^d$ with $[2^d]$ and setting

$$\mathcal{Q}(M)(q; z) = \begin{cases} \text{the smallest } r \text{ such that } z2^{-d} \leq \sum_{r'=1}^r M_{qr'} & \text{if such an } r \text{ exists} \\ w + 1 & \text{otherwise.} \end{cases}$$

Definition 7. For $p \in [0, 1]$ and $\Delta \in \mathbb{N}$, define $\lfloor p \rfloor_\Delta = \lfloor 2^\Delta p \rfloor 2^{-\Delta}$, i.e. p truncated to Δ bits after the radix point. Define $\text{Snap} : [0, 1] \times \{0, 1\}^* \rightarrow [0, 1]$ by

$$\text{Snap}(p, y) = \lfloor \max\{0, p - (0.y) \cdot 2^{-|y|}\} \rfloor_{|y|},$$

where $0.y$ represents a number in $[0, 1]$ in binary.³ Extend the definition to operate on matrices componentwise: $\text{Snap}(M, y)_{qr} = \text{Snap}(M_{qr}, y)$. Further extend Snap to operate on automata with fail states by the rule $\text{Snap}(Q, y) = \mathcal{Q}(\text{Snap}(\mathcal{M}(Q), y))$. (The second argument to Snap should be thought of as random bits.)

Let $\|\cdot\|$ denote the *matrix norm*, i.e. the maximum sum of absolute entries of any row. Define a metric on automata with fail states with the same number of states by setting $\rho(Q, Q') = \|\mathcal{M}(Q) - \mathcal{M}(Q')\|$. The following lemma relates this metric to total variation distance.

Lemma 3. *Suppose Q is a (w, d) -automaton with fail state and Q' is a (w, d') -automaton with fail state. Let δ be the maximum, over all $q \in [w + 1]$, of the total variation distance between $Q(q; U_d)$ and $Q'(q; U_{d'})$. Then $\frac{1}{2}\rho(Q, Q') \leq \delta \leq \rho(Q, Q')$.*

Proof. For each $q, r \in [w + 1]$, let $\rho_{qr} = \Pr[Q(q; U_d) = r] - \Pr[Q'(q; U_{d'}) = r]$. Then $\rho(Q, Q') = \max_{q \in [w]} \sum_{r \in [w]} |\rho_{qr}|$. Since total variation distance is half L_1 distance, $\delta = \frac{1}{2} \max_{q \in [w+1]} \sum_{r \in [w+1]} |\rho_{qr}|$. This immediately shows that $\frac{1}{2}\rho(Q, Q') \leq \delta$. For the second inequality, let q be such that $\delta = \frac{1}{2} \sum_{r \in [w+1]} |\rho_{qr}|$. Since Q and Q' are both automata with fail states, q can be chosen to not be $w + 1$, and hence $\rho(Q, Q') \geq \sum_{r \in [w]} |\rho_{qr}| = 2\delta - |\rho_{q, w+1}|$. Since $\sum_r \rho_{qr} = 0$, $|\rho_{q, w+1}| \leq \rho(Q, Q')$, so $\rho(Q, Q') \geq 2\delta - \rho(Q, Q')$. Rearranging completes the proof. \square

Lemma 4. *For any (w, d) -automaton with fail state Q and any $y \in \{0, 1\}^\Delta$, $\rho(Q, \text{Snap}(Q, y)) \leq w2^{-\Delta+1}$.*

Proof. The snap operation perturbs each entry of the $w \times w$ matrix by at most $2^{-\Delta+1}$. \square

Lemma 5. *Fix a (w, d) -automaton with fail state Q and let $Y \sim U_\Delta$. Then*

$$\Pr[\exists Q' \text{ such that } \rho(Q, Q') \leq 2^{-2\Delta} \text{ and yet } \text{Snap}(Q, Y) \neq \text{Snap}(Q', Y)] \leq w^2 2^{-\Delta+1}.$$

Proof. Let E_{qr} be the bad event that there exists p such that $|\mathcal{M}_{qr} - p| \leq 2^{-2\Delta}$ and yet $\text{Snap}(\mathcal{M}(Q)_{qr}, Y) \neq \text{Snap}(p, Y)$. For E_{qr} to occur, there must be some x a multiple of $2^{-\Delta}$ such that $\mathcal{M}(Q)_{qr} - (0.Y) \cdot 2^{-\Delta}$ is in $[x - 2^{-2\Delta}, x + 2^{-2\Delta})$. There are only two values of Y that can make this happen, so $\Pr[E_{qr}] \leq 2^{-\Delta+1}$. The union bound completes the proof, since $\|M\| \geq \max_{q,r} |M_{qr}|$. \square

³In the notation of [SZ99] and [Arm98], $\text{Snap}(p, y) = \lfloor \Sigma_{(0.y)2^{-|y|}}(p) \rfloor_{|y|}$. In the notation of [CCvM06], $\text{Snap}(p, y) = \mathcal{R}_{y, |y|}(p)$.

4.3 The construction

Recall that w is the number of states (excluding the fail state), ε is the error of Sim , and s is the seed length of Sim . Let $\Delta = \lceil \log(w^2/\varepsilon) \rceil$, let $\delta = 2^{-2\Delta-1}$, and let $\gamma = 2\varepsilon/w$. Let $\text{Samp} : \{0, 1\}^\ell \times \{0, 1\}^d \rightarrow \{0, 1\}^s$ be the averaging (δ, γ) -sampler for $[w]$ -valued functions of Lemma 2. (This defines the constant c ; note that Lemma 2 ensures $d \leq O(\log(w/\varepsilon))$, since the theorem statement assumes $s \leq w$.)

We now define a randomized approximate automaton powering operation $\widehat{\text{Pow}}$. For a (w, d) -automaton with fail state Q and a string $x \in \{0, 1\}^\ell$, we define a (w, d) -automaton with fail state $\widehat{\text{Pow}}(Q, x)$ by the formula

$$\widehat{\text{Pow}}(Q, x)(q; z) = \text{Sim}(Q, q, \text{Samp}(x, z)).$$

Recall that m_0 is the number of steps simulated by Sim , and note that for any Q , for most x , $\widehat{\text{Pow}}(Q, x) \approx Q^{m_0}$. The idea of the SZA transformation is to alternately apply $\widehat{\text{Pow}}$ and Snap . The Snap operation allows us to reuse the randomness of the $\widehat{\text{Pow}}$ operation from one application to the next, thereby saving random bits.

Let Q_0 be the (w, d) -automaton with fail state that is given to SZA as input. Recall that $u = \lceil (\log m)/(\log m_0) \rceil$, where m is the number of steps of Q_0 that SZA is trying to simulate. For a sequence $y = (y_1, \dots, y_u) \in \{0, 1\}^{\Delta u}$ and a string $x \in \{0, 1\}^\ell$, we define a sequence of (w, d) -automata with fail states $\widehat{Q}_0[x, y], \dots, \widehat{Q}_u[x, y]$ by starting with $\widehat{Q}_0[x, y] = Q_0$ and setting

$$\widehat{Q}_{i+1}[x, y] = \text{Snap}(\widehat{\text{Pow}}(\widehat{Q}_i[x, y], x), y_{i+1}).$$

(For $i \geq 1$, Q_i is naturally thought of as a (w, Δ) -automaton with fail state, but since $\Delta \leq d$, we can think of it as reading d bits for each transition and ignoring all but the first Δ of them.) Finally, for seed values $x \in \{0, 1\}^\ell, y \in \{0, 1\}^{\Delta u}, z \in \{0, 1\}^d$, we set

$$\text{SZA}_m^{\text{Sim}}(Q_0, q, x, y, z) := \widehat{Q}_u[x, y](q; z).$$

4.4 Correctness

The bulk of the correctness proof consists of justifying the fact that we use the same x value for each application of $\widehat{\text{Pow}}$ in the definition of \widehat{Q}_i . To do this, we define a *deterministic* approximate powering operation Pow . For a (w, d) -automaton with fail state Q , define a (w, s) -automaton with fail state $\text{Pow}(Q)$ by

$$\text{Pow}(Q)(q; z) = \text{Sim}(Q, q, z).$$

Note that $\text{Pow}(Q) \approx Q^{m_0}$. For a sequence $y = (y_1, \dots, y_u) \in \{0, 1\}^{\Delta u}$, define (just for the analysis) another sequence of (w, d) -automata with fail states $Q_0[y], \dots, Q_u[y]$ by starting with $Q_0[y] = Q_0$ and setting

$$Q_{i+1}[y] = \text{Snap}(\text{Pow}(Q_i[y]), y_{i+1}).$$

We first verify that these automata Q_i (always) provide good approximations for the true powers of Q_0 :

Lemma 6. *For any y , $\rho(Q_u[y], Q_0^{m_0^u}) \leq 8m\varepsilon$.*

Proof. We show by induction on i that

$$\rho(Q_i[y], Q_0^{m_0^i}) \leq \frac{m_0^i - 1}{m_0 - 1} \cdot (2\varepsilon + w2^{-\Delta+1}).$$

In the base case $i = 0$, this is immediate. For the inductive step, by the triangle inequality,

$$\rho(Q_{i+1}[y], Q_0^{m_0^{i+1}}) \leq \rho(Q_{i+1}[y], \text{Pow}(Q_i[y])) + \rho(\text{Pow}(Q_i[y]), Q_i[y]^{m_0}) + \rho(Q_i[y]^{m_0}, Q_0^{m_0^{i+1}}).$$

The first term is at most $w2^{-\Delta+1}$ by Lemma 4. The second term is at most 2ε by the simulator guarantee and Lemma 3. The third term is at most $m_0\rho(Q_i[y], Q_0^{m_0^i})$ by [SZ99, Proposition 2.3]. Therefore, by the inductive assumption,

$$\begin{aligned} \rho(Q_{i+1}[y], Q_0^{m_0^{i+1}}) &\leq w2^{-\Delta+1} + 2\varepsilon + m_0 \cdot \frac{m_0^i - 1}{m_0 - 1} \cdot (2\varepsilon + w2^{-\Delta+1}) \\ &= \frac{m_0^{i+1} - 1}{m_0 - 1} \cdot (2\varepsilon + w2^{-\Delta+1}). \end{aligned}$$

That completes the induction. Finally, we plug in $i = u$:

$$\rho(Q_u[y], Q_0^{m_0^u}) \leq \frac{m_0^u - 1}{m_0 - 1} (2\varepsilon + 2w2^{-\Delta}) \leq 2m \cdot (2\varepsilon + 2\varepsilon). \quad \square$$

Now, we show that the **Snap** operation ensures that with high probability, \widehat{Q}_i and Q_i are exactly equal, despite their different definitions:

Lemma 7. *Let $X \sim U_\ell, Y_1 \sim U_\Delta, \dots, Y_u \sim U_\Delta$ all be independent. Then*

$$\Pr[\text{there is some } i \leq u \text{ such that } \widehat{Q}_i[X, Y] \neq Q_i[Y]] \leq 4m\varepsilon.$$

Proof. By the sampling property, Lemma 3, and a union bound over the w different start states, for each $i \in \{0, \dots, u-1\}$,

$$\Pr[\rho(\text{Pow}(Q_i[Y]), \widehat{\text{Pow}}(Q_i[Y], X)) > 2\delta] \leq w\gamma = 2\varepsilon. \quad (2)$$

(Imagine picking Y first and then taking a probability over the randomness of X alone.) Now, $2\delta = 2^{-2\Delta}$, and by Lemma 5,

$$\Pr \left[\begin{array}{l} \exists Q' \text{ such that } \rho(\text{Pow}(Q_i[Y]), Q') \leq 2^{-2\Delta} \\ \text{and } \text{Snap}(\text{Pow}(Q_i[Y]), Y_{i+1}) \neq \text{Snap}(Q', Y_{i+1}) \end{array} \right] \leq w^2 2^{-\Delta+1} \quad (3)$$

$$\leq 2\varepsilon. \quad (4)$$

By the union bound over the u different values of i , the probability that *any* of these bad events occur is at most $u(2\varepsilon + 2\varepsilon) \leq 4m\varepsilon$. So to prove the lemma, assume that *none* of these bad events occur. In this case, we show by induction that $\widehat{Q}_i[X, Y] = Q_i[Y]$ for every $0 \leq i \leq u$. The base case $i = 0$ holds by definition. For the inductive step, assume $\widehat{Q}_i[X, Y] = Q_i[Y]$. Then because we assumed that the bad event of Equation 2 did not occur, $\rho(\widehat{\text{Pow}}(\widehat{Q}_i[X, Y], X), \text{Pow}(\widehat{Q}_i[Y])) \leq 2^{-2\Delta}$. And hence because we assumed that the bad event of Equation 3 also did not occur, we may conclude that

$$\text{Snap}(\widehat{\text{Pow}}(\widehat{Q}_i[X, Y], X), Y_{i+1}) = \text{Snap}(\text{Pow}(Q_i[Y]), Y_{i+1}).$$

By definition, this implies that $\widehat{Q}_{i+1}[X, Y] = Q_{i+1}[Y]$. □

We have shown that Q_1, Q_2, \dots, Q_u provide good approximations of true powers of Q_0 , and with high probability, $\widehat{Q}_i = Q_i$ for every i . It immediately follows that a random transition of \widehat{Q}_u gives a similar distribution as m_0^u random transitions of Q_0 :

Proof of correctness of SZA. Lemmas 6 and 7 imply that

$$\Pr[\rho(\widehat{Q}_u[X, Y], Q_0^{m_0^u}) \leq 8m\varepsilon] \geq 1 - 4m\varepsilon.$$

By Lemma 3, if x and y are such that $\rho(\widehat{Q}_u[x, y], Q_0^{m_0^u}) \leq 8m\varepsilon$, then $\widehat{Q}_u[x, y](q; Z) \sim_{8m\varepsilon} Q_0^{m_0^u}(q; U_{dm_0^u})$. An averaging argument completes the proof. \square

4.5 Efficiency

The seed length of SZA is $\ell + u\Delta + d$, which is $O(s + u \log(w/\varepsilon))$. We argue that SZA can be implemented to run in $O(s + u \log(w/\varepsilon))$ space through mutual recursion involving two subroutines. The first subroutine, given i, r, z' , computes $\widehat{Q}_i[x, y](r; z')$:

1. If $i = 0$, just consult the input directly. Otherwise:
2. Use the second subroutine to obtain each required entry of $\mathcal{M}(\widehat{\text{Pow}}(\widehat{Q}_{i-1}[x, y], x))$. Apply the definition of \widehat{Q}_i directly.

The space used by this subroutine is only $O(\log(w/\varepsilon))$ plus the space required for computing each matrix entry. The second subroutine, given i, r, v , computes $\mathcal{M}(\widehat{\text{Pow}}(\widehat{Q}_i[x, y], x))_{rv}$:

1. Initialize $\xi = 0$. For all $z' \in \{0, 1\}^d$:
 - (a) Use the oracle to compute $\widehat{\text{Pow}}(\widehat{Q}_i[x, y], x)(r; z')$. If it gives v , set $\xi := \xi + 2^{-d}$. When the oracle makes reads to its automaton/start state inputs, use the first subroutine to compute the necessary values of $\widehat{Q}_i[x, y]$. When the oracle makes reads to its seed inputs, (re)compute $\text{Samp}(x, z')$ to obtain the appropriate bit.
2. Output ξ .

This subroutine's space usage can get up to $O(s + \log(w/\varepsilon))$ for computing the sampler, but before each recursive call, it erases all but $O(\log(w/\varepsilon))$ bits. By induction, this shows that the total space usage of each of these two subroutines (including now the space used for recursive calls) is $O(s + (i + 1) \log(w/\varepsilon))$. It follows that the space used by SZA is $O(s + u \log(w/\varepsilon))$, since it just requires a call to the first subroutine with $i = u$.

In this implementation, the maximum number of unresolved oracle invocations at any time is indeed u , and there is indeed at most one unresolved read of a seed. This completes the proof of Theorem 2. \square

5 Transforming simulators into targeted PRGs

Recall from Section 1.3 that to prove the harder direction of our main result, we require three transformations: an assumed transformation of targeted pseudorandom generators into simulation advice generators, the SZA transformation, and a transformation of simulators into targeted pseudorandom generators. In this section, we construct the last transformation.

We state our transformation in terms of the Ladner-Lynch (LL) oracle model [LL76]. This model is simpler than the implicit oracle model of Section 3. An LL-model oracle algorithm has a single write-only oracle tape. When the algorithm makes a query, the contents of the oracle tape are erased, and the answer to the query is stored in the algorithm's state. Symbols written on the oracle tape do not count toward the algorithm's space complexity. For a non-Boolean oracle $f : \{0, 1\}^* \rightarrow \{0, 1\}^*$, the oracle algorithm is required to specify an index i along with the query string x ; the oracle responds with $f(x)_i$. We emphasize that as with the SZA transformation, this oracle model is only used to cleanly express the transformation; ultimately, we will plug in actual algorithms in place of the oracle.

Lemma 8. *There exists a deterministic LL-model oracle algorithm G such that if Sim is an ε -simulator for $\mathbf{Q}_{wm,d}^m$ with seed length s , then:*

1. G^{Sim} is a targeted $(2mw^2\varepsilon)$ -pseudorandom generator against $\mathbf{Q}_{w,d}^m$.
2. G^{Sim} has seed length s and space complexity $O(s + d + \log w + \log m)$.

To prove Lemma 8, we use Sim to choose a final state, and then we use Sim to “reverse engineer” a string that brings Q to that final state. This reverse engineering process is a straightforward application of the method of conditional probabilities.

Proof. Given (Q, q, x) :

1. Let Q' be the (wm, d) -automaton formed by adding dummy states to Q . Use the oracle to set $R := \text{Sim}(Q', q, x)$.
2. Initialize $v = q$. For $i = 0$ to $m - 1$:
 - (a) For each $z \in \{0, 1\}^d$, let $v_z = Q(v, z)$.
 - (b) Let Q' be a (wm, d) -automaton that simulates $m - i$ steps of Q , with v'_z being the start state corresponding to v_z and R' being the end state corresponding to R .
 - (c) Compute the $z \in \{0, 1\}^d$ that maximizes $\#\{x' : \text{Sim}(Q', v'_z, x') = R'\}$, breaking ties arbitrarily.
 - (d) Print z and set $v := v_z$.

Clearly, G outputs dm bits and uses $O(s + d + \log w + \log m)$ space. Proof of correctness: For $t, r \in [w]$ and $i \in \{0, \dots, m - 1\}$, let $p_{t,r}[i] = \Pr[Q^{m-i}(t; U_{m-i}) = r]$. We show by induction on i that at the beginning of iteration i of the loop on line 2, $p_{v,R}[i] \geq p_{q,R}[0] - 2i\varepsilon$. Base case: At the beginning of iteration $i = 0$, $v = q$. Inductive step: Consider the execution of iteration i of the loop. By the simulator guarantee, there is some $z \in \{0, 1\}^d$ such that $\#\{x' : \text{Sim}(Q', v'_z, x') = R'\} \geq (p_{v,R}[i] - \varepsilon)2^s$. Therefore, G chooses a z that also satisfies that inequality. Therefore, applying the simulator guarantee again, $p_{v_z,R}[i + 1] \geq p_{v,R}[i] - 2\varepsilon$. This completes the induction.

Now, let $X \sim U_s$, and let $Y = G^{\text{Sim}}(Q, q, X)$. Fix an arbitrary state $r \in [w]$; we will show that $\Pr[Q^m(q; Y) = r]$ is close to $\Pr[Q^m(q; U_{dm}) = r]$. Say r is *typical* if $p_{q,r}[0] \geq 2m\varepsilon$. For the first case, suppose r is typical. By the fact that we proved by induction, $\Pr[Q^m(q; Y) = R \mid R \text{ is typical}] = 1$. Therefore,

$$\begin{aligned} \Pr[Q^m(q; Y) = r] &= \Pr[Q^m(q; Y) = r \mid R = r] \cdot \Pr[R = r] + \Pr[Q^m(q; Y) = r \mid R \neq r] \cdot \Pr[R \neq r] \\ &= \Pr[R = r] + \Pr[Q^m(q; Y) = r \mid R \neq r] \cdot \Pr[R \neq r]. \end{aligned}$$

This expression is *lower* bounded by $\Pr[R = r]$, which is lower bounded by $\Pr[Q^m(q; U_{dm}) = r] - \varepsilon$ by the simulator guarantee. On the other hand, the expression is *upper* bounded by $\Pr[R = r] + \Pr[R \text{ is atypical}]$, which is upper bounded by $\Pr[Q^m(q; U_{dm}) = r] + \varepsilon + 2mw\varepsilon$ by the simulator guarantee, the definition of typicality, and the union bound.

For the second case, suppose r is atypical. Then $Q^m(q; Y) = r$ implies that R is atypical, which happens with probability at most $2mw\varepsilon + \varepsilon$ by the definition of typicality and the simulator guarantee.

Therefore, in either case, $\Pr[Q^m(q; Y) = r]$ is within $\pm(2mw + 1)\varepsilon$ of $\Pr[Q^m(q; U_{dm}) = r]$. Statistical distance is half L_1 distance, so the error of \mathbf{G}^{Sim} is at most $\frac{1}{2}w(2mw + 1)\varepsilon \leq 2mw^2\varepsilon$. \square

6 Proof of Theorem 1

6.1 Composing the transformations

In this section, we compose the transformation of Condition 2 of Theorem 1, the SZA transformation, and the transformation of Section 5. (In the overview of Section 1.3, this corresponds to the composition of steps 1, 2, and 3.) The composition is a transformation on targeted pseudorandom generators:

Lemma 9. *Assume Condition 2 of Theorem 1 is true. Fix a constant $\beta > 0$, sufficiently small constants $\sigma > \eta > \gamma > 0$, and a constant $\mu \in (\gamma, 1 - \beta]$. Suppose there is a family $\{\text{Gen}_w\}$, where Gen_w is an efficiently computable targeted ε -pseudorandom generator against $\mathbf{Q}_{w,1}^m$ with seed length s satisfying*

$$s \leq O(\log^{1+\sigma} w), \quad \log(1/\varepsilon) = \log^{1+\eta} w, \quad \log m \geq \log^\mu w.$$

Then there is another family $\{\text{Gen}'_w\}$, where Gen'_w is an efficiently computable targeted ε' -pseudorandom generator against $\mathbf{Q}_{w,1}^m$ with seed length s' satisfying

$$s' \leq O(\log^{1+\max\{\sigma,\beta\}+4\eta} w), \quad \log(1/\varepsilon') \geq \Omega(\log^{1+\eta-\gamma} w), \quad \log m' \geq \log^{\mu+\beta} w.$$

All the hard work of proving Lemma 9 has already been done in Sections 4 and 5; conceptually, the proof is simply by composing. Some technicalities complicate matters slightly. First, we need two little lemmas to deal with the fact that $d > 1$ in Theorem 2, to deal with the fact that Theorem 2 is phrased in terms of automata with fail states, and to deal with the relationship between w and m in Lemma 8.

Lemma 10. *Suppose Gen is an ε -simulation advice generator for $\mathbf{Q}_{(w+1)2^d,1}^m$. Then Gen is also an ε -simulation advice generator for $\tilde{\mathbf{Q}}_{w,d}^{\lfloor m/d \rfloor}$.*

Proof. Let S be the logspace algorithm such that $S(Q, q, \text{Gen}(x))$ is an ε -simulator for $\mathbf{Q}_{(w+1)2^d,1}^m$. Let a be the output length of Gen . For a (w, d) -automaton with fail state Q , a start state $q \in [w+1]$, and a string $y \in \{0, 1\}^a$, let $S'(Q, q, y)$ behave as follows:

1. Let Q' be the $((w+1)2^d, 1)$ -automaton that simulates Q . (One step of Q is simulated by d steps of Q' ; the state space of Q' is $[w+1] \times \{0, 1\}^{<d}$.) Let q' be the start state of Q' corresponding to q .

2. Let $r' = S(Q', q', y)$.
3. Return the state $r \in [w + 1]$ that corresponds to r' .

The maps $(Q, q, y) \mapsto (Q', q', y)$ and $r' \mapsto r$ are computable in logspace, so S' can be implemented to run in logspace. Clearly, $S'(Q, q, \text{Gen}(x))$ is an ε -simulator for $\tilde{\mathbf{Q}}_{w,d}^{\lfloor m/d \rfloor}$. \square

Lemma 11. *There exists a deterministic LL-model oracle algorithm \mathbf{R} with the following properties. Pick $m \leq m'$ and $d \leq d'$. Suppose Sim is an ε -simulator for $\tilde{\mathbf{Q}}_{wm(m+1),d'}^{m'}$ with seed length s . Then $\mathbf{R}_{m,d}^{\text{Sim}}$ is an ε -simulator for $\mathbf{Q}_{wm,d}^m$ with seed length s . (Here m, d are inputs to \mathbf{R} ; we write them as subscripts merely to separate them from the usual simulator inputs.) Further, $\mathbf{R}_{m,d}^{\text{Sim}}$ only uses space $O(d' + \log w + \log m)$.*

Proof. Given (Q, q, x) and oracle access to Sim :

1. Let Q' be a $(wm(m+1), d')$ -automaton with fail state on state space $[wm] \times [m+1]$ (plus a fail state) defined by

$$Q'((q, t); y) = \begin{cases} (Q(q; y \upharpoonright_d), t + 1) & \text{if } t \leq m \\ (q, t) & \text{if } t = m + 1. \end{cases}$$

Here $y \upharpoonright_d$ denotes the first d bits of y .

2. Output the first coordinate of $\text{Sim}(Q', (q, 1), x)$.

The first coordinate of $(Q')^{m'}((q, 1); U_{m'd'})$ is distributed identically to $Q^m(q; U_{md})$, and applying a deterministic function (such as “the first coordinate of”) can only make distributions closer, so this algorithm is correct. Clearly, Q' can be computed from Q in space $O(d' + \log w + \log m)$. \square

Now we are ready to prove Lemma 9; the proof mainly consists in verifying parameters.

Proof of Lemma 9. Using Condition 2 of Theorem 1, transform the family $\{\text{Gen}_w\}$ into a family $\{\text{AdvGen}_w\}$ of simulation advice generators. For each w , let Sim_w be the simulator induced by $\text{AdvGen}_{(w+1)2^d}$ using Lemma 10, where $d = \lceil c[\log^{1+\eta-\gamma}(w) + \log(w)] \rceil$ and c is the constant in Theorem 2. Define

$$\text{Sim}'_w = \text{SZA}_{m'}^{\text{Sim}_w} \quad \text{where } \log m' = \lceil \log^{\mu+\beta} w \rceil.$$

Define

$$\text{Sim}''_w = \mathbf{R}_{m',1}^{\text{Sim}'_{wm'(m'+1)}},$$

where \mathbf{R} is the algorithm of Lemma 11. Finally, define

$$\text{Gen}'_w = \mathbf{G}^{\text{Sim}''_w},$$

where \mathbf{G} is the algorithm of Lemma 8.

Now that we have constructed Gen'_w , we show that our construction worked. Since $\log^{1+\eta-\gamma} w$ is monotone increasing, $\text{Gen}'_{(w+1)2^d}$ can be thought of as having error ε_0 where $\log(1/\varepsilon_0) = \log^{1+\eta-\gamma} w$. Therefore, Sim_w is an ε_0 -simulator for $\tilde{\mathbf{Q}}_{w,d}^{m_0}$, where $\log m_0 \geq \Omega(\log^{\mu-\gamma}(w) - \log(d)) = \Omega(\log^{\mu-\gamma} w)$. Observe that the chosen d value is exactly $\lceil c \log(w/\varepsilon_0) \rceil$. Therefore, by Theorem 2, Sim'_w is a $(12w\varepsilon_0)$ -simulator for $\tilde{\mathbf{Q}}_{w,d}^{m_1}$ for some $m_1 \geq m'$. Again using monotonicity, we can think of

$\text{Sim}'_{wm'(m'+1)}$ as having the same error. By Lemma 11, this implies that Sim''_w is a $(12w\varepsilon_0)$ -simulator for $\mathbf{Q}^{m'}_{wm',1}$, and hence Gen'_w is a targeted ε' -psedorandom generator against $\mathbf{Q}^{m'}_{w,1}$, where $\varepsilon' = 24m'w^3\varepsilon_0$, and hence $\log(1/\varepsilon') \geq \Omega(\log^{1+\eta-\gamma} w)$ as desired.

The seed length of Sim_w is $s_0 \leq O(\log^{1+\sigma+\gamma}((w+1)2^d))$, which is $O(\log^{(1+\sigma+\gamma)(1+\eta-\gamma)} w)$. Since $1 + \sigma + \eta + \gamma + \sigma\eta + \gamma\eta < 1 + \sigma + 4\eta$, we have $s_0 \leq O(\log^{1+\sigma+4\eta} w)$. The parameter u of Theorem 2 is bounded by

$$u \leq O\left(\frac{\log^{\mu+\beta} w}{\log^{\mu-\gamma} w}\right) = O(\log^{\beta+\gamma} w).$$

Therefore, the seed length of Sim'_w is $O(s_0 + u \log(w/\varepsilon_0))$, which is $O(\log^{1+\sigma+4\eta}(w) + \log^{1+\beta+\eta}(w))$, which is $O(\log^{1+\max\{\sigma,\beta\}+4\eta} w)$. Thus the seed length of Sim''_w is $O(\log^{1+\max\{\sigma,\beta\}+4\eta} \text{poly}(w))$, which is $O(\log^{1+\max\{\sigma,\beta\}+4\eta} w)$. Hence the seed length of Gen'_w is the same.

The output length a of AdvGen_{w2^d} satisfies $\log a \leq O(\log^{1+\eta+\gamma}((w+1)2^d))$, which is $O(\log^{(1+\eta+\gamma)(1+\eta-\gamma)} w)$. Since $(1 + \eta)^2 \leq 1 + 3\eta$, we have $a \leq O(\log^{1+3\eta} w)$. Therefore, by Lemma 1 and Theorem 2, the space complexity of Sim'_w is $O(s_0 + u[d + \log w + \log a])$, which is $O(\log^{1+\sigma+4\eta}(w) + \log^{1+\beta+3\eta+\gamma} w)$, which is $O(\log^{1+\max\{\sigma,\beta\}+4\eta} w)$. Therefore, by Lemma 11, the space complexity of Sim''_w satisfies the same bound, and hence so does that of Gen'_w . \square

6.2 Iterating the composition

In this section, we prove the (2 \implies 1) direction of Theorem 1, i.e. we give a strong derandomization under the assumption that targeted pseudorandom generators can be transformed into simulation advice generators. The proof follows the idea outlined in Section 1.3: we repeatedly apply the composition transformation of the last section t times for an arbitrarily large constant t . Each application substantially increases the output length of our targeted pseudorandom generator while the other parameters degrade negligibly, so we end up with an efficiently computable targeted pseudorandom generator with output length w and seed length $O(\log^{1+O(1/t)} w)$:

Lemma 12. *Assume Condition 2 of Theorem 1 is true. Fix $\alpha > 0$. There is a family $\{\text{Gen}_w\}$, where Gen_w is an efficiently computable targeted $(1/6)$ -pseudorandom generator against $\mathbf{Q}^{m_i}_{w,1}$ with seed length $O(\log^{1+\alpha} w)$ where $m_i \geq w$.*

Proof. Let $t = \lceil 2/\alpha \rceil$, $\beta = 1/t$, $\eta = \alpha/(8t)$, and $\gamma = \eta/(3t)$. We show by induction that for $1 \leq i \leq t$, there is a family $\{\text{Gen}_w\}$, where Gen_w is an efficiently computable targeted ε_i -pseudorandom generator against $\mathbf{Q}^{m_i}_{w,1}$ with seed length s_i satisfying

$$s_i \leq O(\log^{1+\beta+4i\eta} w), \quad \log(1/\varepsilon_i) = \log^{1+\eta-2i\gamma} w, \quad \log m_i \geq \log^{i\beta} w.$$

For the base case $i = 1$, use the generator of [INW94, Theorem 3]. For the chosen output length and error, the seed length is $O((\log^{1+\eta-2\gamma} w)(\log^\beta w))$. For the inductive step, suppose we have constructed family i . Apply Lemma 9 to this family, using the chosen β, γ values. (By the choice of γ , $\eta - 2i\gamma > 0$.) The parameters of the resulting family are all correct except that the error merely satisfies $\log(1/\varepsilon') \geq \Omega(\log^{1+\eta-(2i+1)\gamma} w)$; for sufficiently large w , this is at least $\log^{1+\eta-2(i+1)\gamma} w$, so modifying finitely many elements of the family gives family $i + 1$.

That completes the induction. To prove the lemma, use family t . The output length is at least w as desired, and the error is subconstant as desired. The space complexity and seed length are $\log^{1+\beta+4t\eta} w$. By the choices of β, η, γ , $1 + \beta + 4t\eta \leq 1 + \alpha$ as desired (as long as t is sufficiently large.) \square

Proof of the (2 \implies 1) direction of Theorem 1. Fix some promise problem

$$A \in \bigcap_{\beta > 0} \mathbf{promise-BPSPACE}(\log^{1+\beta} n)$$

and a constant $\alpha > 0$. Let M be a probabilistic space- $O(\log^{1+\alpha} n)$ Turing machine that decides A with error $1/6$. Without loss of generality, assume that M has unique accept/reject configurations. On input $x \in \{0, 1\}^n$:

1. Let Q be a $(w, 1)$ -automaton corresponding to the execution of $M(x)$: each state of Q specifies tape contents and a read head location of M , and the transitions of Q correspond to M reading a single random bit. Let the transitions from the accept/reject configurations be self-loops.
2. Use the generator of Lemma 12 (with the chosen α value) to deterministically simulate Q by iterating over all seeds and taking a majority vote. Accept or reject accordingly.

The value w satisfies $\log w \leq O(\log^{1+\alpha} n)$, and Q can be produced from x in deterministic space $O(\log^{1+\alpha} n)$. The space needed for the simulation is $O(\log^{1+\alpha} w)$, which is $O(\log^{(1+\alpha)^2} n)$. Therefore, the composition algorithm deterministically decides A in space $O(\log^{1+2\alpha+\alpha^2} n)$. Since α was arbitrary and $\lim_{\alpha \rightarrow 0} (1 + 2\alpha + \alpha^2) = 1$, this shows that $A \in \bigcap_{\alpha > 0} \mathbf{promise-DSPACE}(\log^{1+\alpha} n)$. \square

6.3 Transforming targeted PRGs into advice generators, assuming derandomization

In this section, we finally prove the easier half of Theorem 1, i.e. we prove that targeted pseudorandom generators can be transformed to simulation advice generators under strong derandomization assumptions. This is essentially immediate from the definitions: under strong derandomization assumptions, no advice is needed to simulate automata, so the identity function (padded appropriately) is trivially a simulation advice generator.

Lemma 13. *If $\mathbf{promise-BPL} \subseteq \bigcap_{\alpha > 0} \mathbf{promise-DSPACE}(\log^{1+\alpha} n)$, then for any $\eta, \gamma > 0$, there is a family $\{\mathbf{Gen}_w\}$, where \mathbf{Gen}_w is an efficiently computable ε -simulation advice generator for $\mathbf{Q}_{w,1}^w$ with seed length s satisfying*

$$s \leq O(\log^{1+\eta} w) \quad \log(1/\varepsilon) \geq \Omega(\log^{1+\eta} w) \quad \log a \leq O(\log^{1+\eta+\gamma} w).$$

Remark 1. For the purpose of proving Theorem 1, Lemma 13 only needed to conclude with Condition 2 of the theorem, i.e. a *transformation* from targeted pseudorandom generators to simulation advice generators. But it turns out that under the derandomization assumption of Lemma 13, we can just construct a simulation advice generator “from scratch.”

Remark 2. The derandomization premise of Lemma 13 may seem weaker than the derandomization statement in Theorem 1 (since it is about $\mathbf{promise-BPL}$ instead of $\bigcap_{\alpha > 0} \mathbf{promise-BPSPACE}(\log^{1+\alpha} n)$). This would again make Lemma 13 stronger than necessary. But the two derandomization statements are actually equivalent by a padding argument.

Proof of Lemma 13. Let B be the following promise problem:

- Input: A $(w, 1)$ -automaton Q , states $q, r \in [w]$, a positive integer $t < 2^{\lceil \log^{1+\eta} w \rceil}$, and padding to make the input length $2^{\lceil \log^{1+\eta} w \rceil}$.
- Yes instances: $\Pr[Q^w(q; U_w) = r] \geq (t + 1)/2^{\lceil \log^{1+\eta} w \rceil}$
- No instances: $\Pr[Q^w(q; U_w) = r] \leq (t - 1)/2^{\lceil \log^{1+\eta} w \rceil}$.

Then $B \in \mathbf{promise-BPL}$. Proof: Simulate w steps of Q from start state q a total of v times, using fresh randomness each time, where

$$v \geq \frac{\ln 6}{2} 2^{2^{\lceil \log^{1+\eta} w \rceil}}.$$

Count how many end up in state r , and accept if and only if the fraction is at least $t/2^{\lceil \log^{1+\eta} w \rceil}$. The space required by this algorithm is $O(\log^{1+\eta} w)$, which is logarithmic in terms of the input length. By Hoeffding's inequality, this algorithm succeeds with probability at least $\frac{2}{3}$.

Therefore, by the premise of the lemma, $B \in \mathbf{promise-DSPACE}(\log^{1+\gamma/(1+\eta)} n)$, i.e. B can be decided in deterministic space

$$O(\log^{(1+\eta)(1+\gamma/(1+\eta))} w) = O(\log^{1+\eta+\gamma} w).$$

Let $\text{Gen}_w : \{0, 1\}^{\lceil \log^{1+\eta} w \rceil} \rightarrow \{0, 1\}^{2^{\lceil \log^{1+\eta} w \rceil}}$ be the identity function padded with zeroes. When the algorithm S is given (Q, q, x) with $x \in \{0, 1\}^{\lceil \log^{1+\eta} w \rceil}$ (i.e. discarding the padding), it behaves as follows:

1. Interpret x as an integer in $\{0, \dots, 2^{\lceil \log^{1+\eta} w \rceil} - 1\}$. Initialize $t = 0$.
2. For each $r \in [w]$:
 - (a) Find the largest Δt such that $(Q, q, r, \Delta t)$ is accepted by the deterministic algorithm that decides B (when the input is padded appropriately.)
 - (b) Set $t := t + \Delta t$.
 - (c) If $t \geq x$, output r .

The space usage of S is $O(\log^{1+\eta+\gamma} w)$ as it should be. Now we analyze the error. The probability of outputting a particular $r \in [w]$ when x is chosen uniformly at random is precisely $\Delta t/2^{\lceil \log^{1+\eta} w \rceil}$, where Δt is the largest value such that $(Q, q, r, \Delta t)$ is accepted by the algorithm that decides B (when padded appropriately.) By the definition of B , this probability is within $2^{-\log^{1+\eta} w}$ of $\Pr[Q^w(q; U_w) = r]$. Total variation distance is half L_1 distance, so the error of the simulator is at most $\varepsilon = \frac{1}{2} w 2^{-\log^{1+\eta} w}$. Hence $\log(1/\varepsilon) \geq \Omega(\log^{1+\eta} w)$. \square

7 Transforming targeted PRGs into advice generators in the uniform setting

The proof of our main result (Theorem 1) is complete; this section can be considered “optional reading”. In this section, we give an unconditional proof of a uniform statement analogous to Condition 2 in Theorem 1. Namely, we show that targeted pseudorandom generators can be

transformed into simulation advice generators, as long as we only worry about correctness with respect to sequences of automata that can be generated in logspace.

One might hope that this would lead to an unconditional derandomization of **BPL** that is only guaranteed to work for easily-generated inputs. Unfortunately, we are not able to prove such a result: when trying to simulate an easily-generated automaton Q using the SZA transformation, the approximate powers of Q that arise are not so easily generated.

Definition 8. Suppose $((Q_1, q_1), (Q_2, q_2), \dots)$ is a sequence where Q_w is a $(w, 1)$ -automaton and $q_w \in [w]$. We say that the sequence is *uniform* if there is some deterministic algorithm that, given w , produces (Q_w, q_w) in space $O(\log w)$.

Definition 9. We say that Gen_w is⁴ a *targeted ε -pseudorandom generator against $\mathbf{Q}_{w,1}^m$ in the uniform setting* if Gen_w is a targeted ε -pseudorandom generator against $\mathbf{A}_w \subseteq \mathbf{Q}_{w,1}^m$ such that for every uniform sequence $((Q_1, q_1), (Q_2, q_1), \dots)$, for all sufficiently large w , the element of $\mathbf{Q}_{w,1}^m$ specified by (Q_w, q_w) is an element of \mathbf{A}_w . We similarly define what it means for Gen_w to be an *ε -simulation advice generator for $\mathbf{Q}_{w,1}^m$ in the uniform setting*.

Proposition 1. *For any constant $\mu \in [0, 1]$ and for any constants $\sigma > \eta > 0$, the following holds. Suppose there is a family $\{\text{Gen}_w\}$, where Gen_w is an efficiently computable targeted ε -pseudorandom generator against $\mathbf{Q}_{w,1}^m$ in the uniform setting with seed length s satisfying*

$$s \leq O(\log^{1+\sigma} w), \quad \log(1/\varepsilon) = \log^{1+\eta} w, \quad \log m \geq \log^\mu w.$$

Then there is another family $\{\text{Gen}'_w\}$, where Gen'_w is an efficiently computable ε -simulation advice generator for $\mathbf{Q}_{w,1}^m$ in the uniform setting with seed length s and output length $a' \leq \text{poly}(w)$.

The proof of Proposition 1 is simple: the simulation advice is just a list of pseudorandom strings for particular (Q, q) pairs. The length of the list is small, but $\omega(1)$, and constructed in such a way that for any uniform sequence $((Q_1, q_1), (Q_2, q_2), \dots)$, for sufficiently large w , the advice includes a pseudorandom string for (Q_w, q_w) .

Proof. Gen'_w behaves as follows, given seed x :

1. For all programs P of length at most $\log w$ that on input w have an explicit self-imposed $O(\log w)$ space bound:
 - (a) Run $P(w)$. If it produces a pair (Q, q) where Q is a $(w, 1)$ -automaton and $q \in [w]$, then print $(Q, q, \text{Gen}(Q, q, x))$.

This generator clearly uses space $O(\log^{1+\sigma} w)$, has seed length s , and has output length $a \leq \text{poly}(w)$. The corresponding algorithm S behaves as follows, given (Q, q, y) where y is the output of Gen'_w :

1. If, for some z , the triple (Q, q, z) appears in y , then output $Q^{|z|}(q; z)$. Otherwise output 1.

This algorithm clearly runs in logspace. We will show that it is an ε -simulator for $\mathbf{Q}_{w,1}^m$ in the uniform setting. Indeed, suppose $((Q_1, q_1), (Q_2, q_2), \dots)$ is uniform via some program P . Then for all sufficiently large w , Gen_w works against (Q_w, q_w) . Furthermore, when $w \geq 2^{|P|}$, the algorithm for Gen'_w will consider P , and hence its output will include the triple $(Q_w, q_w, \text{Gen}_w(Q_w, q_w, x))$. Therefore, for such w , the simulator will give an output that is ε -close to $Q_w^m(q_w; U_m)$. \square

⁴Again, strictly speaking, this is a property of a *family* $\{\text{Gen}_w\}$, not an individual generator.

8 Acknowledgments

The first author is supported by the National Science Foundation Graduate Research Fellowship under Grant No. DGE-1610403. The second author is supported by National Science Foundation Grant No. CCF-1423544 and by a Simons Investigator grant.

References

- [AGHK11] B. Aydinloğlu, D. Gutfreund, J. M. Hitchcock, and A. Kawachi. Derandomizing Arthur-Merlin games and approximate counting implies exponential-size lower bounds. *Computational Complexity*, 20(2):329–366, June 2011.
- [Arm98] R. Armoni. On the derandomization of space-bounded computations. In *Randomization and Approximation Techniques in Computer Science*, pages 47–59. Springer, 1998.
- [AvM12] B. Aydinloğlu and D. van Melkebeek. Nondeterministic circuit lower bounds from mildly de-randomizing Arthur-Merlin games. In *Proceedings of the 27th Annual Conference on Computational Complexity, CCC '12*, pages 269–279. IEEE, 2012.
- [BM84] M. Blum and S. Micali. How to generate cryptographically strong sequences of pseudo-random bits. *SIAM Journal on Computing*, 13(4):850–864, 1984.
- [CCvM06] J. Cai, V. T. Chakaravathy, and D. van Melkebeek. Time-space tradeoff in derandomizing probabilistic logspace. *Theory of Computing Systems*, 39(1):189–208, 2006.
- [For01] L. Fortnow. Comparing notions of full derandomization. In *Proceedings of the 16th Annual Conference on Computational Complexity, CCC '01*, pages 28–34. IEEE, 2001.
- [Gol11a] O. Goldreich. In a world of $\mathbf{P} = \mathbf{BPP}$. In *Studies in Complexity and Cryptography. Miscellanea on the Interplay between Randomness and Computation*, pages 191–232. Springer, 2011.
- [Gol11b] O. Goldreich. Two comments on targeted canonical derandomizers. In *Electronic Colloquium on Computational Complexity (ECCC)*, volume 18, page 11, 2011.
- [GUV09] V. Guruswami, C. Umans, and S. Vadhan. Unbalanced expanders and randomness extractors from Parvaresh-Vardy codes. *Journal of the ACM (JACM)*, 56(4):20, 2009.
- [HILL99] J. Håstad, R. Impagliazzo, L. A. Levin, and M. Luby. A pseudorandom generator from any one-way function. *SIAM Journal on Computing*, 28(4):1364–1396, 1999.
- [IKW02] R. Impagliazzo, V. Kabanets, and A. Wigderson. In search of an easy witness: Exponential time vs. probabilistic polynomial time. *Journal of Computer and System Sciences*, 65(4):672–694, 2002.
- [INW94] R. Impagliazzo, N. Nisan, and A. Wigderson. Pseudorandomness for network algorithms. In *Proceedings of the 26th Annual Symposium on Theory of Computing, STOC '94*, pages 356–364. ACM, 1994.

- [IW97] R. Impagliazzo and A. Wigderson. **P = BPP** if **E** requires exponential circuits: Derandomizing the XOR lemma. In *Proceedings of the 29th Annual Symposium on Theory of Computing*, STOC '97, pages 220–229. ACM, 1997.
- [IW98] R. Impagliazzo and A. Wigderson. Randomness vs. time: De-randomization under a uniform assumption. In *Proceedings of the 39th Annual Symposium on Foundations of Computer Science*, FOCS '98, pages 734–743. IEEE, 1998.
- [KI04] V. Kabanets and R. Impagliazzo. Derandomizing polynomial identity tests means proving circuit lower bounds. *Computational Complexity*, 13(1-2):1–46, 2004.
- [KNW08] D. M. Kane, J. Nelson, and D. P. Woodruff. Revisiting norm estimation in data streams. *arXiv preprint arXiv:0811.3648*, 2008.
- [KvM02] A. R. Klivans and D. van Melkebeek. Graph nonisomorphism has subexponential size proofs unless the polynomial-time hierarchy collapses. *SIAM Journal on Computing*, 31(5):1501–1526, 2002.
- [KvMS12] J. Kinne, D. van Melkebeek, and R. Shaltiel. Pseudorandom generators, typically-correct derandomization, and circuit lower bounds. *Computational Complexity*, 21(1):3–61, 2012.
- [LL76] R. E. Ladner and N. A. Lynch. Relativization of questions about log space computability. *Mathematical Systems Theory*, 10(1):19–32, 1976.
- [Nis92a] N. Nisan. **RL** \subseteq **SC**. In *Proceedings of the 24th Annual ACM Symposium on Theory of Computing*, STOC '92, pages 619–623. ACM, 1992.
- [Nis92b] N. Nisan. Pseudorandom generators for space-bounded computation. *Combinatorica*, 12(4):449–461, 1992.
- [NW94] N. Nisan and A. Wigderson. Hardness vs randomness. *Journal of Computer and System Sciences*, 49(2):149–167, 1994.
- [RR99] R. Raz and O. Reingold. On recycling the randomness of states in space bounded computation. In *Proceedings of the 31st Annual ACM Symposium on Theory of Computing*, STOC '99, pages 159–168. ACM, 1999.
- [RTV06] O. Reingold, L. Trevisan, and S. Vadhan. Pseudorandom walks on regular digraphs and the RL vs. L problem. In *Proceedings of the 38th Annual Symposium on Theory of Computing*, STOC '06, pages 457–466. ACM, 2006.
- [SZ99] M. Saks and S. Zhou. **BP_HSPACE(S) \subseteq DSPACE(S^{3/2})**. *Journal of Computer and System Sciences*, 58(2):376–403, 1999.
- [Uma03] C. Umans. Pseudo-random generators for all hardnesses. *Journal of Computer and System Sciences*, 67(2):419–440, 2003.
- [Wil88] C. B. Wilson. A measure of relativized space which is faithful with respect to depth. *Journal of Computer and System Sciences*, 36(3):303–312, 1988.

- [Wil13] Ryan Williams. Improving exhaustive search implies superpolynomial lower bounds. *SIAM Journal on Computing*, 42(3):1218–1244, 2013.
- [Yao82] A. C. Yao. Theory and application of trapdoor functions. In *Proceedings of the 23rd Annual Symposium on Foundations of Computer Science*, FOCS '82, pages 80–91. IEEE, 1982.
- [Zuc97] D. Zuckerman. Randomness-optimal oblivious sampling. *Random Structures and Algorithms*, 11(4):345–367, 1997.