Electronic Colloquium on Computational Complexity, Report No. 140 (2025)

Composing Low-Space Algorithms

Edward Pyne[*]
MIT
epyne@mit.edu

Roei Tell[†]
University of Toronto
roei@cs.toronto.edu

Abstract

Given algorithms A_1, A_2 running in logspace and linear time, there are two basic ways to compute the composition $x \to A_2(A_1(x))$. Applying naive composition gives an algorithm in linear time but linear space, while applying emulative composition (i.e. the composition of space-bounded algorithms) gives an algorithm in logarithmic space but quadratic time. Such time overheads are extremely common while designing low-space algorithms.

A natural question is whether one can enjoy the best of both worlds: for every A_1, A_2, is there a routine to compute $A_2 \circ A_1$ in linear time *and* in small space? We prove that:

1. For linear time algorithms, this is *unconditionally* impossible – any space-efficient composition must be polynomially slower than the naive algorithm.
2. Extending the unconditional lower bound in either one of many different ways (e.g., to algorithms running in large polynomial time, or to a lower bound for k-fold composition that increases as $n^{\omega_k(1)}$) would imply breakthrough algorithms for a major question in complexity theory, namely derandomization of probabilistic logspace.

The main conceptual contribution in our work is the connection between three questions: the complexity of composition, time-space tradeoffs for deterministic algorithms, and derandomization of probabilistic logspace. This connection gives additional motivation to study time-space tradeoffs, even under complexity-theoretic assumptions, and the motivation is strengthened by the fact that the tradeoffs required in our results are very relaxed (i.e., super-linear tradeoffs for deterministic algorithms).

To prove our results we develop the technical toolkit in an ongoing line of works studying pseudorandomness for low-space algorithms, and as a by-product, we improve several very recent results in this line of work. For example, we show a new "win-win pair of low-space algorithms", extending the construction of (Doron, Pyne, Tell, Williams, STOC 2025) to more general functions and eliminating a previously large polynomial runtime overhead; we reduce the task of derandomizing low-space algorithms to the seemingly tractable task of proving lower bounds for uniform Read-Once Branching Programs; and we introduce two highly efficient targeted pseudorandom generators, which use optimized versions of previous generators as building-blocks.

[*]Supported by the NSF GRFP under grant No. 2141064.

[†]Supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) Discovery Grant RGPIN-2024-04490.

ISSN 1433-8092

Contents

1 Introduction **1**
- 1.1 Our Main Results . . . 2
- 1.2 Corollaries for Pseudorandomness and Circuit Evaluation . . . 5

2 Overview of Proofs **6**
- 2.1 Lower Bounds for Composition . . . 7
- 2.2 Win-Win Algorithms and Derandomization . . . 8

3 Preliminaries **14**
- 3.1 Basic Results . . . 14
- 3.2 Read-Once Branching Programs and Nisan's PRG . . . 15
- 3.3 Pseudorandomness . . . 16

4 Algorithms and Lower Bounds for Composition **17**
- 4.1 Interpolating Naive and Emulative Composition . . . 17
- 4.2 A Lower Bound for Composing Two Algorithms . . . 19
- 4.3 Partial Evidence for General Hardness of Composition . . . 21

5 A Generator with Uniform Deterministic Logspace Reconstruction **24**
- 5.1 Arithmetic Setup . . . 25
- 5.2 The Generator . . . 26
- 5.3 The Reconstruction Procedure for L_v . . . 27
- 5.4 The Reconstruction Procedure for $L_0^{(q)}, \ldots, L_{v-1}^{(q)}$ . . . 30
- 5.5 Putting It All Together: The Reconstruction Procedure . . . 41

6 The Line and Tree Compressors **42**
- 6.1 The Line Generator and Compressor . . . 43
- 6.2 The Tree Generator and Compressor . . . 45

7 Win-Win Algorithms and Derandomization **47**
- 7.1 Win-win Pairs of Algorithms . . . 48
- 7.2 Consequences of Composition Lower Bounds . . . 50
- 7.3 Derandomization From Hardness for ROBPs . . . 53

A Proof of Theorem 5.19 **62**

B Circuit Lower Bounds from Composition Lower Bounds **63**

1 Introduction

Given two algorithms A_1 and A_2 that use very little memory, and an input $x \in \{0,1\}^n$, how can we compute the composition $A_2(A_1(x))$? For concreteness, let's assume that A_1 and A_2 run in linear time and use $O(\log n)$ bits of work space.

The naive method is to compute $y = A_1(x)$, store the output y, and then compute $A_2(y)$. This method runs in linear time, but it may require too much memory. For example, when y is of linear length, this method uses a linear amount of space rather than $O(\log n)$.

A well-known alternative method in algorithm design and complexity theory preserves the space complexity (approximately), but increases the runtime. In this method, we run A_2 and provide it "virtual access" to its input $y = A_1(x)$: whenever A_2 tries to read a bit of y, we run $A_1(x)$ from scratch to compute that bit (and discard the rest of A_1's output). This method still uses $O(\log n)$ bits of space,[1] but its running time is now quadratic instead of linear. This space-efficient yet time-inefficient method (coined "emulative composition" by Goldreich [Gol08]) is a component in nearly every low-space algorithm in theoretical computer science.

We observe that the two known methods can also be interpolated, up to lower order factors, giving an achievable curve: we can compute the composition in any time t and space $s \geq \Omega(\log n)$, as long as the time-space product satisfies $t \cdot s \geq \tilde{\Omega}(n^2)$; see Proposition 4.1 for details. Also, for certain *specific* classes of algorithms, we can do much better; for example, when A_2 is read-once over its input (see Definition 3.2), we can compose in linear time and $O(\log n)$ space.

The question motivating the current work is whether there is a better method for composing low-space algorithms, in general. In other words, we ask:

Question 1.1. What is the time and space complexity of composing low-space algorithms?

As far as we are aware, no methods better than the ones mentioned above are known. However, we are also unaware of any paper studying lower bounds for this problem (e.g., trying to prove that it is impossible to beat this curve), even under complexity-theoretic hardness assumptions.

For context, many previous works in complexity theory studied time-space tradeoffs for computing a *single function*; we survey some of these works below. The new perspective in the current work is trying to find *two functions*, each of them efficiently computable in low space by itself, such that any algorithm computing their composition must incur an additional overhead.[2]

Our contributions, in a gist. For linear time algorithms (as above), we show that overhead-free composition is impossible; that is, we prove an unconditional time-space lower bound for composing two natural algorithms. We then prove a sequence of results, showing that proving the same lower bound in various other natural settings would imply a breakthrough on a major open problem in complexity theory, namely the $\mathsf{BPL} = \mathsf{L}$ problem. For example, proving the same lower bound for algorithms running in *large polynomial time* would imply such a breakthrough.

From a technical perspective, the novelty in the latter results is that the required time-space lower bounds need only hold for *uniform deterministic algorithms*. That is, in contrast to prior work (which deduced such conclusions from time-space lower bounds for non-deterministic or probabilistic machines), the connection that we show to $\mathsf{BPL} = \mathsf{L}$ holds in a relaxed setting, which may be amenable to analysis. Furthermore, the consequences that we deduce also seem within reach.

Hence, the main conceptual contribution in our work is the new connection between three questions: the complexity of composing algorithms, time-space tradeoffs for deterministic algorithms,

[1]For a formal statement incorporating minor overheads, see Proposition 3.3.

[2]Intuitively, the interesting technical challenge here is that we want to prove a "complexity increase"; that is, that the complexity of the composition $A_2 \circ A_1$ is *higher* than the maximum of the complexity of A_1 and of A_2.

and **BPL** = **L**. This connection gives additional motivation for studying time-space tradeoffs for deterministic algorithms, even conditionally (i.e., even under complexity-theoretic assumptions), and this motivation is strengthened by the fact that the tradeoffs required in our results are very relaxed (e.g., only requiring time-space $t^{1.01}$ for t-time algorithms).

1.1 Our Main Results

Our first result is that there are two linear-time algorithms using space $O(\log n)$ whose composition *unconditionally* requires time-space $n^{1.33}$. In particular, there is no way to compose these algorithms without either increasing the space to $n^{\Omega(1)}$, or suffering a polynomial slowdown. The proof of this result is due to Ryan Williams, who generously agreed to include it in this work.

Theorem 1.2 (hardness of composing low-space algorithms [Wil25])**.** *There are two algorithms A_1 and A_2 each mapping n bits to n bits in linear time and space $O(\log n)$ such that for any constant $\varepsilon > 0$, any algorithm computing the composition $A_2(A_1(x))$ correctly on more than an ε-fraction of the inputs x has time-space product at least $n^{1.33}$.*

The lower bound in Theorem 1.2 suggests that, perhaps, there is indeed no better algorithm for composition than the two known methods (and their interpolation). Theorem 1.2 falls short of proving the foregoing statement, since the lower bound is $n^{1.33}$, rather than n^2. As far as we know, there is no reason to believe that it cannot be improved to n^2, and we pose a concrete open problem whose resolution would imply this (see Section 2.1).

Composing algorithms that run in large polynomial time. Next, we ask whether composition overheads are necessary only for linear-time algorithms, or also for algorithms running in larger time. That is, can we show that for *every polynomial* t, there are t-time logspace algorithms whose composition requires time-space $t^{1+\Omega(1)}$? The technical version of Theorem 1.2 holds for $t \approx n^{3/2}$ (and asserts a lower bound of $t^{4/3-\Omega(1)}$, see Theorem 4.4), but this t is still fixed and small.

Our second main result is that generalizing Theorem 1.2 for arbitrary large polynomials t would yield a significant breakthrough on a long-standing open problem in complexity theory: derandomization of probabilistic logspace. (That is, the problem of simulating probabilistic logspace algorithms by deterministic algorithms that incur as little simulation overhead as possible.)

Theorem 1.3 (hardness of composing low-space t-time algorithms implies derandomization)**.** *Suppose that there is $\delta > 0$ such that for every polynomial $t(n)$ and constant $\varepsilon > 0$ the following holds. There are two algorithms A_1 and A_2 running in time t and space $O(\log n)$ such that any algorithm computing $A_2(A_1(x))$ successfully on an ε-fraction of inputs $x \in \{0,1\}^n$ requires time-space product $t^{1+\delta}$.*[3] *Then* $\mathbf{BPL} \subseteq \cap_{\varepsilon>0}\mathbf{zavg}_\varepsilon\mathbf{L}$.[4]

Note that in Theorem 1.2 the lower bound holds for $\delta \approx 1/3$, whereas in Theorem 1.3, a lower bound with any $\delta > 0$ (i.e., arbitrarily small) suffices for the conclusion.

It is often conjectured that **BPL** can indeed be derandomized (i.e., that **BPL** = **L**), but proving this has remained an infamously open challenge for decades. The strongest known results assert that $\mathbf{BPL} \subseteq \mathbf{SPACE}[n^{3/2-o(1)}]$ (see [SZ99, Hoz21, Hoz22]) or that $\mathbf{BPL} \subseteq \mathbf{SC}$ (see [Nis91, Nis92, Pyn24]).

[3] In this statement we refer to the runtime of both A_1 and A_2 as a function $t = t(n)$ of the length of the input $x \in \{0,1\}^n$ to the composition. We could alternatively describe A_1 as running in time t and outputting t bits, and describe A_2 as running in linear time (in t); these choices are syntactic and do not meaningfully affect our results.

[4] For definitions of **BPL** and of **zavgL**, see Section 3. In a gist, the meaning of the statement is that every probabilistic algorithm using space $O(\log n)$ can be simulated by a *deterministic* algorithm using space $O(\log n)$, where the simulation is correct over $1-\varepsilon$ of the inputs (and $\varepsilon > 0$ is arbitrarily small).

The conclusion in Theorem 1.3 asserts derandomization of BPL on *average*, which is weaker than $\mathsf{BPL} = \mathsf{L}$, but would still be a major result (see, e.g., [DPT24, DPTW25]).

We do not have strong evidence that the hypothesis in Theorem 1.3 is true, even under standard complexity-theoretic hardness assumptions, and we believe that finding such evidence is an important open problem. In Section 4.3 we present candidate functions whose composition may be hard, along with a reduction of proving lower bounds for composing two algorithms to a potentially easier problem (of proving lower bounds for composing constantly many algorithms).

Composing more than two algorithms. We now consider another seemingly innocent generalization of Theorem 1.2, which may be more amenable to analysis. Fix arbitrary algorithms $A_1, ..., A_k$, each running in linear time and logspace. Note that we can compute the composition $A_k(A_{k-1}(...(A_1(x))))$ in time $O(k \cdot n)$ and linear space, and that emulative composition runs in logspace but requires time at least n^k. Can we compute the k-fold composition in low space (say, $O(\log n)$ or $\text{polylog}(n)$) without paying a runtime overhead of $n^{\Omega(k)}$?

For large values of $k = k(n)$, this question captures classical questions in complexity theory. For example, for $k = O(\log n)$, the classical question of whether $\mathsf{L} = \mathsf{NL}$ is equivalent to the question of whether the k-wise composition of a certain logspace algorithm (i.e., for Boolean matrix squaring) can be done in polynomial time and logspace.[5] For an even larger $k = \text{poly}(n)$, the question of whether $\mathsf{L} = \mathsf{P}$ is equivalent to the question of whether k-wise composition of any function can be computed in logspace and polynomial time (this is because the P-complete problem of Circuit Evaluation reduces to composing $\text{poly}(n)$ logspace algorithms). We elaborate more on that below.

Our third main result focuses on the case of a small $k = O(1)$. We show that a lower bound of $n^{\Omega(k)}$ for this case would yield the same breakthrough derandomization result as in the conclusion of Theorem 1.3. In fact, even a more relaxed lower bound of $n^{\Omega(\log^*(k))}$ suffices.

Theorem 1.4 (hardness of composing k low-space algorithms implies derandomization)**.** *Assume for every polynomial p and $\varepsilon > 0$ there is $k \in \mathbb{N}$ for which the following holds. There are algorithms $A_1, ..., A_k$, each running in linear time and space $O(\log n)$, such that any algorithm computing $A_k(A_{k-1}(...A_1(x)))$ successfully on an ε-fraction of inputs x with space* $\text{polylog}(n)$ *runs in time at least $p(n)$. Then $\mathsf{BPL} \subseteq \cap_{\varepsilon>0}\mathsf{zavg}_\varepsilon\mathsf{L}$.*

Even proving a lower bound as in Theorem 1.4 for composing *super-constantly many algorithms* would yield a similar, slightly weaker conclusion. (For the technical statement of this, see Theorem 2.3.) For example, if there are $k = (\log n)^{.01}$ logspace algorithms that cannot be composed in SC (i.e., in polynomial time and $\text{polylog}(n)$ space), then BPL can be simulated in deterministic space $(\log n)^{1.01}$ (where both the hardness assumption and the derandomization conclusion refer to $1 - \varepsilon$ of inputs).

However, the bright side is that this setting can be more easily related to classical questions in algorithm design and complexity theory (i.e., compared to the setting of $k = 2$ and large t), and thus we have somewhat stronger evidence that the lower bound might be true. Specifically, for the setting of $k \geq \log(n)$, two well-known complexity-theoretic conjectures – the standard assumption that $\mathsf{NC} \not\subseteq \mathsf{SC}$, and the strong assumption that Savitch's theorem [Sav70] cannot be sped-up in space $n^{o(1)}$ – imply that k-wise composition with space $\text{polylog}(n)$ requires super-polynomial time, and requires time $n^{\Omega(k^{1-\varepsilon})}$ for every $\varepsilon > 0$, respectively. See Section 4.3 for further details.

[5]This is since NL-complete problem of $s \to t$ connectivity is equivalent under logspace reductions to the problem of squaring such a matrix for $O(\log n)$ times.

Even worst-case hardness suffices. In the unconditional lower bound of Theorem 1.2 as well as in Theorems 1.3 and 1.4, the hardness was "average case", in the sense that every algorithm fails to compute the composition on at least $1-\varepsilon$ of the inputs. Finally, we show that in the setting of composing exponential-time algorithms, even a worst-case lower bound suffices for derandomization, and in fact for full (i.e., worst-case) derandomization of linear space.

Theorem 1.5 (worst-case hardness implies derandomization). *Suppose there is $\delta > 0$ such that for every sufficiently large $d \in \mathbb{N}$ there is $k \in \mathbb{N}$ for which the following holds. There are algorithms $A_1, \ldots, A_k$ so that:*

- *On input $x \in \{0,1\}^n$, letting $x_i = A_i \circ \ldots \circ A_1(x)$, we have that $A_{i+1}(x_i)$ is computable in time 2^{dn} and space $O(n)$ for every i.*
- *For every algorithm B running in time $2^{dn(1+\delta)}$ and space $\mathrm{poly}(n)$, for every sufficiently large n there is $x \in \{0,1\}^n$ such that $B(x) \neq A_k \circ \ldots A_1(x)$.*

Then $\mathsf{BPSPACE}[n] \subseteq \mathsf{SPACE}[O(n)]$.

Context: Time-space tradeoffs for computing a single function. Time-space tradeoffs for computing a *single function* (i.e., that does not necessarily arise from a composition) have been studied in complexity theory since the works of Cobham [Cob66] and of Borodin and Cook [BC82]. To contextualize our results, we compare them to these prior works.

First, the difference in our results between the setting of small polynomial time and that of large polynomial time (cf., Theorem 1.2 vs. Theorem 1.3) is analogous to the difference between the two settings when studying time-space tradeoffs for a single function. Specifically, there are many known time-space tradeoffs for functions computable in small polynomial time (e.g., quadratic tradeoffs for functions [BC82, Yes84, Abr91, RS82, Bea91, BCM13, MW19, Din20, YZ24] and super-linear tradeoffs for decision problems [BJS01, Ajt05, Ajt02, BSSV03, BV02, Pag05]), but no known tradeoffs for functions computable in large polynomial time.[6] In fact, as far as we are aware, time-space tradeoffs for functions computable in large polynomial time are not even known to follow from standard complexity-theoretic hardness assumptions.

Secondly, similarly to our conclusions in Theorems 1.3 to 1.5, previous works showed that derandomization-type conclusions would follow from time-space tradeoffs for a function computable in large time. However, crucially, the required hardness in prior works is for strong models of computation, for which far fewer lower bound techniques are known (e.g., for non-deterministic machines [Kor22, Theorem 29], or for probabilistic algorithms [BCT25, Theorem 5.3]), and accordingly, the conclusions in prior work are general and strong (e.g., $\mathsf{prBPP} = \mathsf{prP}$). In contrast, our results only rely on time-space tradeoffs for *uniform deterministic machines*, and we deduce conclusions that seem within reach (i.e., relaxed versions of $\mathsf{BPL} = \mathsf{L}$).

Our interpretation of the latter difference is optimistic. Lower bounds for deterministic machines (and the conclusions regarding $\mathsf{BPL} = \mathsf{L}$) might be tractable, and our results give additional motivation for studying the three questions that the current work connects: the complexity of composition, time-space tradeoffs for deterministic algorithms, and $\mathsf{BPL} = \mathsf{L}$.

[6]Another line of works shows polynomial time-space tradeoffs for computing NP-hard functions, such as satisfiability [Kan84, PPST83, Gra94, For97, LV99, Tou01, FLvMV05, Wil06, Wil08, Wil13, BW15], and some of these works also showed time-space tradeoffs for probabilistic machines [DvM06, Die07, MW21]. Also, a recent line of works studies time-space tradeoffs in the read-once streaming model [Raz19, DS18, GRT19, LTWY23].

1.2 Corollaries for Pseudorandomness and Circuit Evaluation

As we will explain in Section 2, to prove Theorems 1.3 to 1.5 we significantly improve the technical toolbox in an ongoing line of work studying pseudorandomness for low-space algorithms [PRZ23, DPT24, LPT24, Pyn24, DPTW25]. Hence, as a by-product of our technical contributions, we obtain significant improvement to some very recent results in this line of work.

1.2.1 Derandomization from Hardness for ROBPs

Recall that space-bounded derandomization amounts to constructing a low-space algorithm that approximates the acceptance probability of a given Read-Once Branching Program (ROBP; see Definition 3.5). Recent works reduced the latter task to proving lower bounds for uniform deterministic models of computation, where these models are weak but still stronger than ROBPs (e.g., for uniform constant-depth circuits with majority gates and oracle access to ROBPs in [DPT24], or for uniform polysize circuits with oracle access to space $\varepsilon \cdot n$ in [DPTW25]). This is an undesirable state of affairs, since the target of the reduction might be harder to analyze than the origin.

Our next result rectifies this: we show that derandomization of linear space follows from natural lower bounds for uniform ROBPs. We find this quite striking, since ROBPs are often thought of as one of the simplest computational models in complexity theory (i.e., both weak and amenable to combinatorial analysis), and indeed exponential lower bounds for ROBPs have been well-known for many decades (see, e.g., [BHST87]). The particular lower bound needed in our result differs from what is known because it refers to hardness of *compressing* an explicit string by uniform ROBPs (of size quasipolynomial in the length of the string), rather than to hardness of *computing* an explicit truth-table (by ROBPs of size that is smaller than the truth-table).

Theorem 1.6 (derandomization of ROBPs from lower bounds for ROBPs)**.** *There is $c > 1$ such that the following holds. Suppose that the following is true:*

> *There is an algorithm that gets input 1^n, runs in space $O(\log n)$, and outputs a list of n-bit strings $f_{n,1}, \ldots, f_{n,m}$ such that for every logspace-uniform family of ROBPs of width $2^{(\log n)^c}$ and length n and every large enough n, the ROBP fails to compress $f_{n,i}$ to size $\mathrm{polylog}(n)$, for some i.*
>
> *(The notion of "compressing" here means outputting a machine M of size $\mathrm{polylog}(n)$ that gets input $i \in [n]$, runs in space $\mathrm{polylog}(n)$ and time $\mathrm{poly}(n)$, and outputs $(f_n)_i$.)*

Then, $\mathbf{BPSPACE}[n] \subseteq \mathbf{SPACE}[O(n)]$.

In fact, to deduce the conclusion, we only need a logspace-computable string incompressible by a specific family of ROBPs, which is even "more uniform" than stated (e.g., the map from input x and $i \in [n]$ to the i^{th} internal state of the ROBP after reading $x_{\leq i}$ can be computed in space $O(\log n)$, which is sub-logarithmic in the ROBP size). For a formal statement, see Theorem 7.5.

Remark 1.7. An alternative way of reading the assumption in Theorem 1.6 is as referring to hardness for deterministic, low-space streaming algorithms; indeed, known lower bounds hold even for *probabilistic* streaming algorithms, whereas Theorem 1.6 only needs lower bounds for deterministic algorithms.[7] Other ways of interpreting the assumption are as asserting the existence of a refuter,

[7] Theorem 1.6 is somewhat reminiscent of a result of Chen, Tell, and Williams [CTW23], who showed that derandomization of **BPP** is equivalent to constructing a refuter for n^{ε}-space probabilistic streaming algorithms. Inspecting their proof, to derandomize **BPP** it suffices to refute algorithms that compress the string to a circuit of size n^{ε}. In comparison, in Theorem 1.6 we only need lower bounds for *deterministic* streaming algorithms (indeed, just uniform ROBPs), these algorithms run in *polylogarithmic* space rather than space n^{ε}, and the compressed version is a *low-space* machine rather than a general circuit (but we need the algorithm printing f_n to run in space $O(\log n)$).

or as an algorithm obtained using a low-space witnessing theorem from bounded arithmetic; for brevity, we omit a full explanation (for explanations see, e.g., [CJSW21, GC25]).

1.2.2 Unconditional "Win-Win" Results

A special case of composing k low-space algorithms is the problem of evaluating a logspace-uniform circuit of depth $(k \cdot \log(n))$ (this is because we can compute each sequence of $O(\log n)$ layers as a function of the previous layer in logspace, using the standard DFS-style algorithm). We use this observation to establish an "instance-wise" connection between circuit evaluation and pseudorandomness for low-space algorithms, as follows.

Recall that Doron *et al.* [DPTW25] showed a "win-win pair of algorithms": a pair of algorithms $\mathcal{A}_1, \mathcal{A}_2$ such that on *every input* x, either $\mathcal{A}_1$ simulates a BPL machine on x, or $\mathcal{A}_2$ solves $s \to t$ connectivity on x (interpreted as a graph), and both algorithms are significantly more efficient than the known algorithms for the respective task. The caveat is that we do not know which of the two algorithms works on any given input (even though at least one is guaranteed to work).

We present a significantly improved win-win pair of algorithms. Specifically, in our result $\mathcal{A}_1$ can compute the k-wise composition of any logspace computable function f, rather compute only $s \to t$ connectivity, and in addition it is faster, running in time that is near-linear in the runtime of f (the algorithm in [DPTW25] had large polynomial overheads; see Section 2.2 for details). As one appealing corollary, we prove *unconditionally* that for *every given input* x, either we can evaluate a circuit of super-logarithmic depth at x in SC (i.e., in polynomial time and polylogarithmic space), or we can derandomize a BPL machine at x.

Theorem 1.8 (win-win pair of algorithms for derandomization and circuit evaluation)**.** *For every $\varepsilon > 0$ and $R \in \mathsf{BPL}$ and $C \in \mathsf{unif}_{\mathsf{logspace}}\mathsf{NC}^{1+\varepsilon}$,*[8] *there are algorithms $\mathcal{A}_1, \mathcal{A}_2$ such that for every $x \in \{0,1\}^n$, either:*

1. *$\mathcal{A}_1(x)$ computes $C(x)$ in time $\mathrm{poly}(n)$ and space $\mathrm{polylog}(n)$.*
2. *$\mathcal{A}_2(x)$ computes $R(x)$ in space $O(\log^{1+\varepsilon} n)$.*

Moreover, both algorithms report if they fail to compute the answer, and never exceed their resource bounds.

By scaling up Theorem 1.8 to linear space and exponential time, we can also obtain a win-win result for complexity classes: either probabilistic linear space can be derandomized in deterministic space $O(n^{1+\varepsilon})$, or all languages decidable by logspace-uniform circuits of size $2^{O(n)}$ and depth $(\log n)^{1+\varepsilon}$ can be decided in time $2^{O(n)}$ and space $\mathrm{poly}(n)$. See Theorem 7.3 for details.

2 Overview of Proofs

We first discuss the lower bound for time-space efficient composition, then the derandomization results.

[8] We say a language is in $\mathsf{unif}_{\mathsf{logspace}}\mathsf{NC}^{1+\varepsilon}$ if there is a logspace-uniform sequence of circuits $\{C_n\}_{n\in\mathbb{N}}$, where C_n has size $\mathrm{poly}(n)$ and depth $O(\log^{1+\varepsilon} n)$, and $x \in \{0,1\}^n$ is in the language iff $C_n(x) = 1$.

2.1 Lower Bounds for Composition

Our lower bound for composition proceeds by taking a problem for which we have quadratic time-space lower bounds, and writing it as the composition of functions g_1, g_2 that can themselves be computed in small space and time. Thus, any function computing $g_2 \circ g_1$ must incur a nontrivial time-space overhead.

The specific problem we use is *sorting*. We are given a list of n numbers $L = (v_1, \ldots, v_n)$ in $[n^c]$, and must output $\mathsf{sort}(L)$. Near-quadratic time-space lower bounds for this problem are known even for RAM algorithms with access to advice [BC82, RS82, Bea91, MW19], and these bounds hold on average over a random input list.

We show that we can sort as $g_2(g_1(L))$, where both functions are length-preserving, and computable in logspace and time $\widetilde{O}(n^{3/2})$. Thus, a time- and space-efficient composition would violate the sorting lower bound (and then a simple padding trick implies Theorem 4.4).

Sorting in Two Levels For simplicity of presentation, here we assume the input list contains no duplicates, and $m = \sqrt{n}$ is an integer. The function g_1 iterates over $i \in [m]$, and for each i does the following. Given the quantile values d_i, d_{i+1} for the $(i \cdot m)^{th}$ and $((i+1) \cdot m)^{th}$ largest elements respectively, first output d_i. Subsequently, scan through the input and output all elements of L in the range $[d_i, d_{i+1})$ in the order we encounter them. Finally, increment i and proceed to finding the next quantile. It is easy to see that the output of $g_1(L)$ is

$$d_0, L_0, \ldots, d_{m-1}, L_{m-1}$$

and the sublists L_i satisfy $d_i \leq L_i < d_{i+1}$ but are themselves unsorted.

For elements in a polynomial-sized domain, quantile finding can be done in nearly-linear time and logspace: Specifically, we perform a binary search over the domain, where at each step we determine which half of the current sub-domain to recurse into by scanning the input list once and counting the elements in each half of the current sub-domain. We do this to find each of the m quantiles, and so the time complexity of g_1 is $m \cdot \tilde{O}(n) = \tilde{O}(n^{3/2})$.

Then the function g_2 takes the output $g_1(L)$ and sorts each sublist L_i, using the brute-force algorithm, which runs in $O(m^2) = O(n)$ time and logspace. The total runtime of g_2 is $O(n^{3/2})$.

We note that our bound for composition could be quantitatively improved by speeding up both layers. In fact, such a speed up might yield essentially the tightest possible tradeoff:

Open Problem 2.1. Are there functions g_1, g_2 computable in simultaneous logspace and nearly-linear time such that $g_2(g_1(L)) = \mathsf{sort}(L)$?

An affirmative answer to Open Problem 2.1 would imply space-efficient composition of length-preserving functions must impose a quadratic overhead on runtime, which matches the time-space product achieved by naive and emulative composition (and the interpolation between them) up to polylog factors. Moreover, a quantitative improvement over our results can even be obtained from a more relaxed algorithm, which sorts by composing four algorithms:

Open Problem 2.2. Are there functions g_1, g_2, g_3, g_4 computable in simultaneous logspace and nearly-linear time such that $g_4(g_3(g_2(g_1(L)))) = \mathsf{sort}(L)$?

It is not immediately obvious that an affirmative answer for Open Problem 2.2 would imply a lower bound for a *single composition*, but we prove that this is indeed the case. Specifically, an affirmative answer (for sorting, or for any other function for which quadratic time-space tradeoffs are known) would yield a lengh-preserving function f such that computing $f \circ f$ with polylogarithmic space requires runtime $\widetilde{\Omega}\left(n^{\sqrt{2}}\right) = \omega(n^{1.41})$ (see Proposition 4.5).

2.2 Win-Win Algorithms and Derandomization

Both Theorem 1.3 and Theorem 1.4 follow from a more general technical result. The result asserts that for every low-space computable function f and probabilistic low-space machine M there are two algorithms $\mathcal{A}_1$ and $\mathcal{A}_2$ that, on *every input* x, satisfy the following: Either $\mathcal{A}_1(x)$ computes the k-wise composition $f^{(k)}(x)$ at time and space similar to those of computing $f(x)$ once, or $\mathcal{A}_2(x)$ deterministically simulates $M(x)$ with low space overhead. We stress that at each input at least one of the algorithms is guaranteed to work, but we do not know which one of them works. Indeed, this "win-win pair of algorithms" is closely related to the main technical result of Doron *et al.* [DPTW25], and we will explain the connection to their work after the result statement.

To state the result, consider an input $x \in \{0,1\}^n$, and a function f that we will compose for k times. We want f to represent computation in time $t = t(n)$ and logspace; for convenience, we assume that f is length-preserving and computable in *linear* time and logspace, and we will first apply it to $x||0^t$ (i.e., to a padded version of x), and then to $f(x)$, $f(f(x))$, and so on.[9] Our result asserts that for every $k = k(n)$ and every input x, either $\mathcal{A}_1(x)$ computes $f^{(k)}(x||0^t)$ in time $t^{1.01}$ and polylogarithmic space (i.e., dramatically faster than emulative composition, which takes time $t^{k(n)}$), or $\mathcal{A}_2(x)$ deterministically simulates a $\mathbf{BPL}$ machine on x in space $O(k(n) \cdot \log(n))$.

Theorem 2.3 (win-win pair of algorithms for composition and derandomization)**.** *Let $R \in \mathbf{BPL}$, let $\delta > 0$ be a constant, and let $t(n)$ be a sufficiently large polynomial (depending on R and on δ). Then, for every $k : \mathbb{N} \to \mathbb{N}$ and length-preserving $f : \{0,1\}^* \to \{0,1\}^*$ computable in quasilinear time and logspace, there are algorithms $\mathcal{A}_1, \mathcal{A}_2$ such that for every $x \in \{0,1\}^n$ at least one of the following occurs:*

1. *$\mathcal{A}_1(x)$ computes $f^{(k)}(x||0^{t(n)})$ in time $t(n)^{1+\delta}$ and space $\mathrm{polylog}(n)$.*
2. *$\mathcal{A}_2(x)$ computes $R(x)$ in space $O(k(n) \cdot \log n)$.*

Moreover, both algorithms report if they fail to compute the answer, and never exceed their resource bounds.

It is instructive to compare Theorem 2.3 to the main technical result of Doron *et al.* [DPTW25]. They also presented a win-win pair of algorithms: in their case, either $\mathcal{A}_1$ solved directed s-t connectivity on the graph x in polynomial time and polylogarithmic space (i.e., in $\mathcal{SC}$), or $\mathcal{A}_2$ computed $R(x)$ in non-deterministic logspace. In comparison, Theorem 2.3 is a vast generalization and a significant quantitative improvement. First, instead of solving connectivity, $\mathcal{A}_1$ now computes a composition of an *arbitrary* function f (connectivity is a special case, since it can be solved by repeated composition of matrix squaring). Secondly, $\mathcal{A}_1$ runs in near-linear time $t^{1+\delta}$, rather than in arbitrarily large polynomial time; this is crucial for our results asserting that lower bounds for composition imply derandomization (cf., Theorem 1.3). And thirdly, $\mathcal{A}_2$ does *not* use non-determinism, but (in contrast to [DPTW25]) has a multiplicative space overhead of k.[10]

Recap: The bootstrapping system of [DPTW25]. As a starting point, let us recall the techniques in [DPTW25]. Given input x, they considered a bootstrapping system a-la [CT21a], comprised of a sequence of rows $P_1(x), ..., P_{k=\log n}(x)$ where each $P_i(x) \in \{0,1\}^n$ is the result of applying matrix squaring on the adjacency matrix of the graph x for i times.

[9]This choice is not consequential for our results. For example, instead of padding x, we could alternately define an initial function f_0 mapping n bits to $t(n)$ bits, and then iteratively compose f on $f_0(x)$.

[10]The original result of [DPTW25] does not follow from the statement of Theorem 2.3 (since the original result refers to ($k = \log n$)-wise composition and has $\mathcal{A}_2$ running in non-deterministic logspace), but it can be easily recovered using the new technical tools that underlie Theorem 2.3.

Fixing a suitable suitable reconstructive pseudorandom generator GEN, let us ask what happens if we apply GEN to each $P_i(x)$, yielding a set $S_i(x) = \mathsf{GEN}(P_i(x))$. (For simplicity, think of using the Nisan-Wigderson generator [NW94] with the string $P_i(x)$ as a "hard truth-table".) If there is $i \in [k]$ such that $S_i(x)$ fools the probabilistic machine $M(x)$, then $\mathcal{A}_2$ can derandomize $M(x)$ in non-deterministic logspace; specifically, it finds the useful i, computes $P_i(x)$ in $\mathcal{NL}$, applies the generator, and simulates $M(x)$ with the coins in $S_i(x)$ (see details below). Otherwise, the failure of GEN for all $i \in [k]$ allows them to deduce that there is a low-space machine REC that gets input $P_i(x)$ and compresses $P_i(x)$ to a polylogarithmic-size oracle circuit.[11] Since we can simulate access to $P_i(x)$ when we have access to $P_{i-1}(x)$ (because the sequence of matrix-squaring rows is downward self-reducible in logspace), the algorithm $\mathcal{A}_1(x)$ can now iteratively run REC to compress $P_1(x), P_2(x), \ldots$, and so on, each time discarding all but the previous circuit, until it prints $P_k(x)$.

To make this approach work, they needed the following properties from the generator GEN and the reconstruction algorithm REC. The generator GEN needs to be computable in logspace, and to be able to detect its failure (in order to decide if there is $i \in [k]$ such that $S_i(x)$ is pseudorandom for $M(x)$). The reconstruction REC needs to be deterministic, and to run in polylogarithmic space. For these purposes, they constructed a version of the generator of Shaltiel and Umans [SU05] with a derandomized low-space reconstruction, building on versions in prior works [PRZ23, DPT24].

Our starting point: Trying to generalize to composition of arbitrary functions. The initial observation underlying our result is that the approach above superficially seems to almost immediately generalize to a k-wise composition of arbitrary functions. Indeed, starting with an arbitrary low-space f (rather than matrix squaring) we can define $P_i(x) = f^{(i)}(x)$, and apply GEN to each $P_i(x)$. In the first case (derandomization), instead of computing $P_i(x)$ non-deterministically, we compute it using emulative composition, in space $O(k \cdot \log(n))$.

However, this idea cannot be materialized to obtain our results (or, in fact, any meaningful result for a small k) using previously known technical tools. The problem is in the second case, in which the algorithm $\mathcal{A}_1$ computes $f^{(k)}$ in time that is polynomial, but is very large; in particular, for small k, the time overhead is larger than the runtime t^k of emulative composition. This overhead comes from the large runtime of applying REC to each $P_i(x)$, and from the fact that when we have a compressed version of $f^{(i-1)}(x)$ and want to compute $\mathsf{REC}(f^{(i)}(x))$ in low space, the natural approach is to use emulative composition (i.e., whenever REC wants a bit j of $f^{(i)}(x)$, we compute f on $f^{(i-1)}(x)$ and store only the j^{th} bit), which yields a quadratic overhead on top of that.

Main technical challenge: Eliminating *all* time overheads. The core technical contribution in our work is a new generator that allows us to overcome the two challenges above. In a gist, the reconstruction algorithm for this generator is both extremely time efficient (i.e., runs in near-linear time), and is *read-once* with respect to its input f (see Definition 3.2). The read-once property will allow us to avoid the overhead of emulative composition in the algorithm $\mathcal{A}_1$ as a whole.

Theorem 2.4 (logspace generator with deterministic, near-linear time, read-once reconstruction). *There is a universal constant $c > 1$ and there are algorithms* $\mathsf{GEN}, \mathsf{REC}$ *such that for every $f \in \{0,1\}^{t=n^d}$ and* ROBP *B of size n, the following occurs.*[12]

[11]For simplicity, assume that this compression is a circuit C_i that gets input j and oracle access to $M(x,\cdot)$ and outputs $P_i(x)_j$. In their actual result, C_i is a machine of description size $\mathrm{polylog}(t)$ (where $t = |P_i(x)|$) that, on any input j, runs in time $\mathrm{poly}(t)$ and space $\mathrm{polylog}(t)$.

[12]Indeed, our generator will be pseudorandom for ROBPs. Recall that for a randomized logspace algorithm, if we fix the input x and consider the action of the algorithm over the random coins r, we obtain an ROBP $B_x : \{0,1\}^{n^c} \to \{0,1\}$ of size at most n^c, where the constant c depends on the space complexity of the randomized algorithm; and

1. *The algorithm* $\mathsf{GEN}(B, f)$ *runs in space* $O(\log n)$ *and either outputs a* $(1/10)$*-additive estimate of* $\mathbb{E}[B(\mathbf{U})]$ *or* $\perp$.

2. *If* $\mathsf{GEN}(B, f) = \perp$, *the algorithm* $\mathsf{REC}(B, f)$ *runs in time* $t \cdot n^c$ *and space* $\mathrm{polylog}(n)$, *reads* f *in read-once fashion, and outputs an oracle machine* M *of description size* $\mathrm{polylog}(n)$ *that satisfies the following. When given input* $j \in [t]$, *the machine* M *runs in space* $\mathrm{polylog}(n)$ *and time* n^c, *and satisfies* $M^B(j) = f_j$.

The construction underlying Theorem 2.4 uses classical generators as sub-components (e.g., yet-another variation on [SU05]), but it is a new construction that requires additional ideas, rather than an optimization of a known generator. We will describe it in Sections 2.2.1 and 2.2.2.

Remark 2.5. For context, recall that reconstruction procedures in classical generators are either non-uniform or probabilistic (e.g., non-uniform in [NW94, IW97, SU05, MV05], or probabilistic in [IW98, SU07, TV07, CRTY20, CLO$^+$23]). Generators with derandomized reconstruction procedures for the low-space setting have been introduced recently by Pyne, Raz, and Zhan [PRZ23] (see also [PRZ23, LPT24, DPT24], and indeed [DPTW25]), and all previous constructions use "heavy" pseudorandom tools and involve large time overheads.

Let us see how Theorem 2.4 suffices to prove Theorem 2.3. The derandomization algorithm $\mathcal{A}_2$ enumerates over $i \in [k]$, and instantiates GEN with $f = f^{(i)}(x)$. If for some i we obtain an estimate of $\mathbb{E}[B(\mathbf{U})]$, we successfully derandomize R on input x; otherwise we output $\perp$. We can compute $x \mapsto f^{(i)}(x)$ in space $O(k \cdot \log n)$ via emulative composition, and thus $\mathcal{A}_2$ runs in space $O(k \cdot \log n)$.

If $\mathcal{A}_2$ outputs $\perp$, it must be the case that for every i, $\mathsf{REC}(B, f^{(i)}(x))$ prints a machine M_i of size $\mathrm{polylog}(n)$ such that $M_i^B(j) = f^{(i)}(x)_j$. The algorithm $\mathcal{A}_1$ iteratively finds a compressed representation of $f^{(1)}(x), \dots, f^{(k)}(x)$ in time $t \cdot \mathrm{poly}(n) \le t^{1+\delta}$ and space $\mathrm{polylog}(n)$.

Specifically, assuming we have such a representation M_{i-1} for $f^{(i-1)}(x)$, we run the algorithm $\mathsf{REC}(B, f^{(i)}(x))$ to build a compressed representation M_i for $f^{(i)}(x)$. Crucially, we now exploit the fact that REC reads $f^{(i)}(x)$ in read-once fashion (and sequentially, i.e. it reads the bits in-order): We simulate the machine that computes f on input $f^{(i-1)}(x)$ in stages; whenever REC asks for the next bit of $f^{(i)}(x)$, we simulate the machine until it prints that bit, and then pause the simulation until REC asks for the subsequent bit (keeping its intermediary paused configuration in storage). Once REC halts and outputs a machine M_i representing $f^{(i)}(x)$, we delete M_{i-1} and increment i. Thus, the runtime of compressing each layer i is only $t \cdot n^c$, and so is the total runtime;[13] and since we never store more than two machines, the space complexity stays $\mathrm{polylog}(n)$.

2.2.1 The Line Generator and Compressor

We first describe a simple generic transformation that takes a reconstructive pseudorandom generator and makes the reconstruction procedure read-once over the hard string/truth-table f. We call the resulting construction the $\mathsf{Line\ Generator\ and\ Compressor}$, and denote the pair of algorithms ($\mathsf{LINEGEN}, \mathsf{LINEREC}$) (and see Section 6.1 for a formal statement).

Suppose we are given a string $f \in \{0,1\}^t$ which we want to either compress or use to derandomize B. Fix a reconstructive PRG, for example the version of the Shaltiel-Umans generator [SU05] from [DPTW25], denoting the PRG by SU and the reconstruction by RSU. The reconstruction RSU is deterministic and low-space, but requires read-many access to f.[14]

to determine the output of the machine at x it suffices to estimate the expectation of B_x up to error, say, $1/10$ (see Proposition 3.7). For simplicity of presentation in this introduction, we assume $c = 1$.

[13] Without loss of generality we may assume $k \le \sqrt{\log n}$, as otherwise the algorithm $\mathcal{A}_2$ unconditionally exists [SZ99].

[14] In this overview, we ignore the role of transforming the ROBP distinguisher to a collection of next-bit-predictors. For this key step, we adopt the same strategy as prior work [LPT24, DPTW25].

The generator. For each $i \in [t]$, denote by

$$P_i \stackrel{\text{def}}{=} f_{1\dots i}||0^{t-i}$$

the first i bits of f, padded to length t. The generator $\mathsf{LINEGEN}$ computes SU^{P_i} for every i, and outputs the collection of lists $(\mathsf{SU}^{P_i})_{i\in[t]}$, one of which is hopefully pseudorandom.[15] Since SU is logspace, $\mathsf{LINEGEN}$ can compute the output lists in logspace (using its access to f).

The reconstruction. For $i = 1, ..., n$, the reconstruction $\mathsf{LINEREC}$ iteratively builds a machine M_i such that for all $j \in [i]$

$$M_i(B, j) = (P_i)_j,$$

where each M_i is of size $\text{polylog}(n)$ and can be evaluated in time $\text{poly}(n)$ and space $\text{polylog}(n)$.

Assuming $\mathsf{LINEREC}$ has found such a machine M_{i-1}, it now wants to run RSU^{P_i} to obtain M_i. To do so it reads the next bit f_i, and since

$$P_i = f_{1..i-1}||f_i||0^{t-i} = M_{i-1}(B, 1)|| \dots ||M_{i-1}(B, i-1)||f_i||0^{t-i}$$

the reconstruction now has all of the information needed to compute P_i. (Specifically, whenever P_i is queried at location $j < i$, the reconstruction runs M_{i-1} and answers its queries using its input B, and whenever $(P_i)_i$ is queried the reconstruction answers with the stored bit f_i.) It then invokes RSU^{P_i}, and once RSU produces a machine M_i representing P_i, we can delete M_{i-1} and increment i. After t steps, $\mathsf{LINEREC}$ obtains a compressed representation $M = M_t$ for f.

Note that the reconstruction runs in time that is essentially t times the runtime of RSU (on a truth table of length t), and in space $\text{polylog}(n)$ (as it only stores at most two machines M_{i-1}, M_i at each step). Finally, note that $\mathsf{LINEREC}$ indeed reads each bit of f once, and in increasing order (see Definition 3.2 for a formal definition of this read-once model).

2.2.2 The Tree Generator and Compressor

The remaining problem is that the reconstruction algorithm $\mathsf{LINEREC}$ runs in large $\text{poly}(t)$ time. Specifically, its runtime is dominated by the final application of RSU, where we apply it on the entire truth table f of length t. Examining RSU in depth (as well as other classical reconstructive PRGs with deterministic log-space reconstruction [PRZ23,DPT24]), we do not see a way to directly optimize/tweak the underlying technical tools and reduce c to $1 + \varepsilon$.[16]

The next construction, called the $\mathsf{Tree\ Generator\ and\ Compressor}$, is again based on a generic transformation of any reconstructive PRG whose reconstruction is read-once over f into a reconstructive PRG whose reconstruction is both read-once and *time-efficient*. Specifically, the reconstruction will run in time $t \cdot n^c \ll t^{1+\delta}$, where $c > 1$ is universal and the inequality holds for a large enough polynomial t (see Theorem 6.5).

The high-level idea is as follows. Let us consider the reconstruction's perspective, trying to compress f. Imagine that we have the ability to compress any piece of information of length m into $\text{polylog}(n)$ bits, in time $\text{poly}(m)$. (Intuitively, this is a justifiable assumption, since the generator can try and use any m bits of information for derandomization, and if it fails then the reconstruction

[15]Indeed, this yields a somewhere-PRG rather than a PRG (since we only hope that one the lists will be pseudorandom). For $(\mathsf{SU}, \mathsf{RSU})$ specifically and when the distinguisher is an ROBP, the generator can also test each list SU^{P_i} for pseudorandomness, and only output a single pseudorandom list (this is possible because the reconstruction algorithm is low-space and deterministic, so the generator finds P_i on which the reconstruction fails; see [DPT24]).

[16]For example, even if we ignore the time complexity of all technical tools, as part of the low-space derandomization of the reconstruction it repeatedly enumerates over sampler outputs in a universe of size at least 2^t.

can compress these m bits.) For simplicity, let us ignore the complexity of decompression for a moment. The problem with the line compressor is that it tries to compress $m = t$ bits, which costs time $\text{poly}(t)$. So now we will try and compress only blocks of $m = n$ bits.

Recursive compression. We will use a recursive compression approach. We first compress each block of n bits of f into $\text{polylog}(n)$ bits, yielding a new string of length $(t/n) \cdot \text{polylog}(n)$ (i.e., the concatenated compressed representations of the blocks). Then we repeatedly compress the string, at each iteration i obtaining a string of length $(t/n^i) \cdot \text{polylog}(n)$. After constantly many iterations, we obtain a representation of f of size $\text{polylog}(n)$, as we wanted.

Indeed, at each compression step we only pay time complexity $\text{poly}(n)$. However, the problem (again, ignoring the complexity of decompression) is the complexity of the compression procedure as a whole. To see the issue, think of the recursive approach above as a tree over f, and note that given f, the reconstruction wants to output the compressed version that is at the root of this tree. On the one hand, the reconstruction can never store an entire intermediate tree layer. On the other hand, using emulative composition costs time that is polynomial in t (which seems like a chicken-and-egg problem, since we were trying to avoid precisely this overhead).

We avoid the time overhead by exploiting the fact that the algorithm we use to compress each n-bit block (i.e., LINEREC) is *read-once* over the block. Specifically, in contrast to general low-space algorithms, observe that emulative composition of read-once algorithms can be done in a way that is both time-efficient and space-efficient. For example, to compute $A_1(A_2(x))$ where A_1 is read-once, we simulate A_2 and store its intermediate configuration, and whenever A_1 accesses an input bit, we continue the simulation of A_2 until it produces the next bit;[17] this way, A_2 is only simulated once, and we avoid a multiplicative time overhead. Using this idea and analyzing the recursive composition carefully, this suffices to compress f in time $t \cdot \text{poly}(n)$.

Repairing the generator's complexity. The idea above suffices for efficient compression, but now the generator became space-inefficient. To see why, note that we need the generator to apply LINEGEN to the n bits corresponding to each node of the tree described above.[18] In particular, the generator needs to compute the compressed descriptions of each block in the tree.

The problem is that the compressed description of a block is the output of LINEREC on the block, and thus the generator needs to compute LINEREC on n-bit blocks. However, LINEREC uses space $\text{polylog}(n)$, which (while being low enough for the reconstruction algorithm) is prohibitively large for the generator, which we want to run in logspace (for derandomization of **BPL**).

To resolve this problem we show that the output of LINEREC can also be computed by an alternative algorithm, which runs in logspace but is not read-once over f. Indeed, the reconstruction will use the first algorithm for LINEREC, which is read-once but uses space $\text{polylog}(n)$, whereas the generator will use the second algorithm for LINEREC, which is not read-once but uses space $O(\log n)$. This alternative algorithm is essentially just the reconstruction algorithm RSU of the underlying reconstructive PRG that we apply to each node, where the non-trivial point is that this algorithm produces the same output as our first algorithm for LINEREC.

Decompression. The compressed representation (i.e., the root of the tree above) allows to compute the mapping $i \mapsto f_i$ in time $\text{poly}(t)$ and space $\text{polylog}(t)$, as follows. In our actual construction

[17]Indeed, this requires not only that A_1 accesses each input bit once, but also that it accesses them in the order at which A_2 produces them. In the current paper our notion of read-once requires this stricter condition (in fact we require that A_1 reads its input sequentially and in-order, left to right), and LINEREC satisfies this notion.

[18]Recall that the reconstruction needs the ability to compress each such block of n bits, and thus we need to assume that the ROBP distinguishes the output of LINEGEN on each block from random.

of the tree above, instead of using arbitrary n-bit blocks, we use blocks of size $n \cdot \text{polylog}(n)$, each of which contains n compressed descriptions of size $\text{polylog}(n)$. Thus, given location $i \in [t]$, we can trace a unique path from the root to a leaf representing a block of f that contains index i.

The decompression algorithm follows this path, which is of length $d = O(1)$. At each iteration it has a current compressed description of a block stored, which allows it to compute the $\text{polylog}(t)$-size portion of the block that is the compressed description of the next block (in space $\text{polylog}(t)$ and time $\text{poly}(t)$). Since at any given moment the algorithm just needs to store two compressed descriptions, the overall decompression runs in space $\text{polylog}(t)$ and time $\text{poly}(t)$.

2.2.3 Improving the Uniformity of Shaltiel-Umans

Implementing the idea outlined in Section 2.2.2 runs into a final serious issue. As explained above, the generator's space complexity is at least the space complexity of RSU, i.e. the reconstruction of the underlying PRG used at each node. Recall that the latter PRG is a version of the Shaltiel-Umans [SU05] generator, shown by Doron et. al. [DPTW25]. However, their reconstruction algorithm uses $\text{polylog}(n)$ space, rather than $O(\log n)$ space.

We improve their construction, and indeed show a version of the Shaltiel-Umans PRG with a *deterministic logspace reconstruction algorithm*. The full construction, which is presented in Section 5, is involved and relies on too many low-level technical details to survey here. Let us therefore describe the main observation that underlies our improvement, at a high level.

The algorithm in [DPTW25] produces $\text{poly}(n)$ circuits, each of size $\text{polylog}(n)$, and one of those correctly computes the hard string/truth-table $P \in \{0,1\}^n$.[19] Since these circuits do not seem evaluable in space less than $\text{polylog}(n)$, the approach in [DPTW25] was to try out all circuits.

In this work too, we still do not know if it is possible to evaluate each of the output circuits in space less than $\text{polylog}(n)$. However, the crucial observation is that for the *correct* circuit, we can verify in space $O(\log n)$ that it is indeed correct.

Being somewhat inaccurate for the sake of this high-level description, each circuit computes the mapping $i \mapsto P_i$ by iteratively computing a sequence of $k = \log(n)$ collections of bits of P, denoted $\mathcal{L}_1, ..., \mathcal{L}_k$, where each $\mathcal{L}_j$ consists of $\text{polylog}(n)$ bits $P_{j,1}, ..., P_{j,\text{polylog}(n)}$ (and $\mathcal{L}_k$ contains P_i). At each iteration, the circuit tries to compute $\mathcal{L}_{j+1}$ from $\mathcal{L}_j$, and the correct circuit will do this successfully (whereas incorrect circuits will output $\mathcal{L}'_{j+1}$ that contains an incorrect bit of P).

Thus, the verification task reduces to checking that for each $j = 1, ..., k-1$, the sub-component in the candidate circuit that computes $\mathcal{L}_{j+1}$ from $\mathcal{L}_j$ works correctly. This might mistakenly seem infeasible, because just storing $\mathcal{L}_j$ requires $\text{polylog}(n)$ bits. However, for the *correct* circuit, we do not need to store all of the bits in $\mathcal{L}_j$, because we *know* that these are the corresponding bits of P. And fortunately, the locations in P of these bits, as well as the functionality of the sub-component, can all be computed in space $O(\log n)$. Thus, for each candidate circuit, and for $j = 1, ..., k-1$, we check that the sub-component correctly produces $\mathcal{L}_{j+1}$ from $\mathcal{L}_j$, without ever needing to store more than $O(\log n)$ bits.

2.2.4 Derandomization From Lower Bounds for ROBPs

Finally, we explain how we derive derandomization from finding strings that are hard to compress by read-once branching programs (Theorem 1.6). We focus on derandomizing *linear* space. For derandomization on n-bit inputs, our hypothesized algorithm will print an incompressible string f

[19]We use the notation P rather than f to remind the reader that this PRG will be applied to each node in the construction of the Tree Generator.

of length $N = 2^{O(n)}$, in space $O(\log N) = O(n)$. For simplicity, let us consider derandomizing a single ROBP B_n of length and width $N = 2^{O(n)}$, which can be printed in space $O(n)$.[20]

For a compression ROBP $\mathcal{B}_n$, we label each state v in the final layer with a machine M_v; we say that $\mathcal{B}_n$ compresses f if the final state reached on input f is labeled with a machine that computes f. Note that if $\mathcal{B}_n$ takes inputs of length N and has size w, it can hope to successfully compress only at most a $2^{-\log(w)/N}$ fraction of inputs. We construct such an ROBP $\mathcal{B}_n$ of length N and width $2^{\text{poly}(n)}$, such that every non-compressed input can be used to derandomize B_n.

The ROBP $\mathcal{B}_n$ is obtained from the line compressor. Fixing B_n, $\mathsf{LINEREC}(B_n, f)$ attempts to compress f to size $s = \text{polylog}(N)$, and moreover does so taking a read-once pass over f. Our ROBP $\mathcal{B}_n$ essentially implements $\mathsf{LINEREC}(B_n, \cdot)$, and as $\mathsf{LINEREC}$ uses space $\text{poly}(n)$, $\mathcal{B}_n$ has size $2^{\text{poly}(n)}$. Thus, for every f that $\mathcal{B}_n$ does not compress, we have that $\mathsf{LINEGEN}(B_n, f) = \rho$, where ρ is a $(1/10)$-approximation of $\mathbb{E}[B_n]$, which suffices to derandomize L on input length n.

Furthermore, this ROBP $\mathcal{B}_n$ can be printed in space $\text{poly}(n) = O(\log |\mathcal{B}_n|)$, essentially because $\mathsf{LINEREC}$ runs in this space on inputs of length 2^n. Moreover, we can compute the map from (potentially partial) input $f_{\leq i}$ to the i^{th} state v reached in $\mathcal{B}_n(f_{\leq i})$ in space $O(n)$, essentially via the alternative logspace algorithm for $\mathsf{LINEREC}$ mentioned in Section 2.2.2.

3 Preliminaries

Throughout the paper we work in the multi-tape Turing machine model. We say a machine runs in space s if s is the space used in total across all worktapes. However, our lower bound for composition (Theorem 1.2) holds in the RAM model as well, as the time-space lower bounds it relies on hold in that model.

Definition 3.1. We say a family of circuits/ROBPs $\{C_n\}_{n\in\mathbb{N}}$ is logspace uniform if there is an algorithm that on input 1^n runs in space $O(\log |C_n|)$ and outputs C_n. We say $L \in \mathbf{unif_{logspace}SIZEDEP}[S, D]$ if L can be decided by a family of logspace-uniform circuits of size $S(n)$ and depth $D(n)$.

We define what it means for an algorithm to be read-once over an input.

Definition 3.2. We say an algorithm $\mathcal{A}(x, y)$ is read-once over the input x if the algorithm works as follows. It is given y on a standard (read-only) input tape, and x on a second (also read-only) input tape, where the head on the second input tape (for x) can never be moved left.

3.1 Basic Results

We first formally recall emulative composition. For clarity, we state it for length-preserving functions.

Proposition 3.3 ([Gol08], Lemma 5.2)**.** *Let $g_1, g_2 \colon \{0,1\}^\star \to \{0,1\}^\star$ be length-preserving functions computable in space $s_1, s_2 \in [\log n, n]$ and time t_1, t_2. Then, $g_2 \circ g_1$ can be computed in space*

$$s(n) = s_2(n) + s_1(n) + O(\log(n)) \text{ and time } O(t_1 \cdot t_2).$$

We require that we can hardwire inputs into algorithms in a space-efficient manner (in particular, the machine with this information hardwired can printed in small space).

[20]Technically, we must derandomize a set of 2^n ROBPs, one for each input $x \in \{0,1\}^n$. However, these can be represented by a single uniform ROBP of size $2^{O(n)}$; see the proof of Theorem 7.5 for details.

Lemma 3.4 (Efficiently printing TMs with "hard-wired" information)**.** *Let M be a Turing machine that runs in time $t(n) \geq n$ and space $s(n)$ on n-bit inputs. Then, there is another Turing machine that gets input 1^n and $y \in \{0,1\}^m$, where $m < n$, runs in space $O(\log m)$ and time $O(m)$, and prints a machine M' satisfying the following:*

1. *The description size of M' is $O(m)$.*
2. *When given input $x \in \{0,1\}^{n-m}$, the machine M' runs in time $\tilde{O}(t(n) \cdot m)$ and space $s(n) + O(\log n)$ and outputs $M(x \circ y)$.*[21]

Proof. First observe that given input $y \in \{0,1\}^m$, we can print in time $O(m)$ a machine $PRINT_y$ that, on an empty input, prints y to its output tape.[22]

The machine M' that we print implements the following functionality. It simulates M, while keeping track of the location of its input head; whenever the input head exceeds location $n-m$, at every time-step M' simulates $PRINT_y$ (from scratch) while keeping a counter for the number of bits $PRINT_y$ outputs, stores the output bit y_i corresponding to the current location i of the input head, and simulates the functionality of M according to y_i.

One way to design M' that implements this functionality and that has an efficiently printable representation is as follows. The machine has two sets of states, one set of constantly many states implementing the main functionality, and another set of $O(m)$ states implementing the functionality of $PRINT_y$. The first set of states are indexed by $0j$ where $j \in [O(1)]$, and the second set of states are indexed by $1j$ where $j \in [O(m)]$. The transition function may be represented as a truth-table of size $O(m)$ (the transitions between $0j$ states are a constant-sized DFA, and the transitions between $1j$ states are trivial). This representation is of total size $O(m)$, and can be printed in time $O(m)$.

The space consumption of M' is $s(n)+O(\log n)$, where the additive overhead is for counters and for keeping track of the input head location. Its runtime is $\tilde{O}(t(n) \cdot m)$, where the polylogarithmic overhead comes from the various counters (i.e., for simulating $PRINT_y$ and for keeping track of the input head location), and for simulating the functionality above using the same number of tapes as M (i.e., when we work in a multitape model with a fixed number of tapes). $\square$

3.2 Read-Once Branching Programs and Nisan's PRG

Read-once branching programs act as our derandomization target, and as a model of compressors.

Definition 3.5. An read-once branching program (ROBP) of length n and size s is a layered directed graph with $(n+1)$ layers, each with s states per layer.[23] For every state v in layer i, there are edges labeled $0, 1$ to (not necessarily distinct) states v_0, v_1 in layer $i+1$. There is an initial state v_{st} and a final state v_{acc}, and we say $B(x) = 1$ for $x \in \{0,1\}^n$ if, starting at v_{st} and following the edge labeled x_i at layer i for $i \in [n]$, we reach state v_{acc}.

We say B is a multi-output ROBP if each state v in the final layer is labeled with a string $p_v \in \{0,1\}^*$, such that $B(x) = p_v$ for the final state v reached on input x.

We need that ROBPs can be evaluated in logspace.

Fact 3.6. A read once branching program $B : \{0,1\}^n \to \{0,1\}$ of size $s \geq n$ has the property that the map $x \to B(x)$ can be computed in space $O(s)$, given access to x and B.

[21]We do not claim anything about the behavior of M when it is given input whose length is not $n - m$.

[22]Specifically, we print a machine with $O(m)$ states, where for $i \in [m]$ the i^{th} set of $O(1)$ states implements the functionality "print y_i and move the head right".

[23]We let the size denote the size of each layer, which as we always assume $s \geq n$ does not affect the asymptotics.

We recall the correspondence between randomized logspace algorithms and ROBPs:

Proposition 3.7 (Languages to ROBPs)**.** *There is $c > 1$ such that for every $R \in \mathsf{BPSPACE}[s(n)]$ for $s \geq \log n$, there is a space cs algorithm that, on input $x \in \{0,1\}^n$, outputs an ROBP $B_x : \{0,1\}^{2^{cs}} \to \{0,1\}$ of size 2^{cs} such that if $x \in R$, then $\mathbb{E}[B_x] \leq 1/3$ and if $x \notin R$, $\mathbb{E}[B_x] \geq 2/3$.*

We next recall the pseudorandom generator for ROBPs of Nisan [Nis94], which requires an explicit family of hash functions:

Fact 3.8. For every $t \in \mathbb{N}$, there exists a pairwise independent hash family $\mathcal{H}\colon \{0,1\}^t \to \{0,1\}^t$ such that $|\mathcal{H}| = 2^{2t}$, and $h \in \mathcal{H}$ (which we associate with $h \in \{0,1\}^{2t}$) can be evaluated in space $O(\log t)$.

Definition 3.9 (Nisan's PRG)**.** For $n = 2^\ell \in \mathbb{N}$, let $t = 50\ell$. For $(h_1, \ldots, h_\ell) \in \mathcal{H}_t$, define $\mathrm{NIS}_{(h_1,\ldots,h_\ell)} : \{0,1\}^t \to \{0,1\}^{t \cdot n}$ inductively as follows. Let $\mathrm{NIS}_0(r) = r_1$, and for $j \in [\ell]$

$$\mathrm{NIS}_{(h_1,\ldots,h_j)}(r) = (\mathrm{NIS}_{(h_1,\ldots,h_{j-1})}(x) \| \mathrm{NIS}_{(h_1,\ldots,h_{j-1})}(h_j(r))).$$

Note that NIS can be evaluated in space $O(\ell)$ given access to r and the hash functions.

3.3 Pseudorandomness

We recall basic definitions related to pseudorandomness. We let $\mathbf{U}_n$ denote the uniform distribution over $\{0,1\}^n$.

Definition 3.10. We say $L \in \mathsf{BPL}$ if there is a randomized Turing machine M (with one-way access to the random tape) that runs in logspace, always halts in polynomial time, and $x \in L$ (resp. $x \notin L$) if $\Pr[M(x) = 1] \geq 2/3$ (resp $\Pr[M(x) = 1] \leq 1/3$).

Definition 3.11. We say $L \in \mathsf{zavg}_\varepsilon\mathsf{L}$ if there is a deterministic Turing machine M that runs in logspace, always halts in polynomial time, and $M(x) \in \{0, 1, \perp\}$. Moreover, for every $n \in \mathbb{N}$ we have that

$$\Pr_{x \leftarrow \mathbf{U}_n}[M(x) = \mathbb{I}[x \in L]] \geq 1 - \varepsilon$$

and $M(x) = \perp$ for every case where $M(x) \neq \mathbb{I}[x \in L]$.

Note that our definition is zero error, i.e. the machine never incorrectly decides the language, only fails to return an answer.

Predictors and D2P transformations. For a distribution $\mathbf{D}$, we say the size of the distribution is its support size.

Definition 3.12. For a function $C : \{0,1\}^n \to \{0,1\}$ and a distribution $\mathbf{D}$ over $\{0,1\}^n$, we say $\mathbf{D}$ α-fools C if $|\mathbb{E}[C(\mathbf{U}_n)] - \mathbb{E}[C(\mathbf{D})]| \leq \alpha$. For a function $P : \{0,1\}^m \to \{0,1\}$ where $m < n$, we say P is a δ-predictor for $\mathbf{D}$ if

$$\Pr_{x \leftarrow \mathbf{D}}[P(x_{\leq m}) = x_{m+1}] \geq \frac{1}{2} + \delta.$$

We recall the definition of a Distinguish-to-Predict (D2P) transformation by Li et. al [LPT24], following Doron et. al. [DPT24].

Definition 3.13 ([LPT24])**.** For a circuit $C\colon \{0,1\}^n \to \{0,1\}$, we say a collection of circuits $\mathcal{P}\colon \{0,1\}^{<n} \to \{0,1\}$ is an α-distinguish to δ-predict (D2P) transformation for C if the following holds. For every distribution $\mathbf{D}$ of size at most m that does not δ-fool C, there is $P \in \mathcal{P}$ such that P is an α-predictor for $\mathbf{D}$.

4 Algorithms and Lower Bounds for Composition

In this section we show one upper bound and one lower bound on composing space-bounded algorithms. Specifically, in Section 4.1 we show an algorithm that interpolates naive composition and emulative composition (as mentioned in Section 1), and in Section 4.2 we prove Theorem 1.2. Finally, in Section 4.3 we discuss evidence for hardness of composition and a specific candidate, and prove that a lower bound on any constant number of compositions implies a lower bound on one composition.

4.1 Interpolating Naive and Emulative Composition

Consider composing two algorithms running in time t and space s. Naive composition works in time $O(t)$ and space $O(t)$, whereas emulative composition works in time $O(t^2)$ and space $O(s + \log t)$. The following algorithm generalizes both up to polylogarithmic factors, by allowing to compute the composition in any time $t' < t$ and $s' > s$, as long as $s' \cdot t' \geq \tilde{\Theta}(t^2 \cdot s)$.

The idea is to divide the computation of the inner machine $\mathcal{M}_1(x)$ into b checkpoints, and for each checkpoint (which we space evenly throughout the time-t computation) we store the working configuration of $\mathcal{M}_1$. These checkpoints take $O(bs)$ space to store. Then, when the outer machine $\mathcal{M}_2$ requests a bit j of $\mathcal{M}_1(x)$, rather than simulating $\mathcal{M}_1$ from the start, we can begin the simulation from the relevant checkpoint, from which we only need to simulate $O(t/b)$ steps to obtain $\mathcal{M}_1(x)_j$, resulting in a total runtime of $\widetilde{O}(t^2/b)$. We make this idea precise below.

Proposition 4.1. *For every pair of functions g_1, g_2 computable in simultaneous time $t(n) \geq n$ and space $s(n) \geq \log(t)$, and for any nice function $b(n) \in [1, t]$,*[24] *the composition $g_2 \circ g_1$ is computable in simultaneous time $\widetilde{O}(t^2(n)/b(n))$ and space $O(s(n)b(n))$.*

Proof. Without loss of generality we assume $t(n)/b(n)$ is an integer. Let $\mathcal{M}_1$ and $\mathcal{M}_2$ be Turing machines that compute g_1 and g_2 respectively in time t and space s. We simulate their composition with $O(1)$ additional worktapes, then appeal to the result of [HS66] to construct a final two-tape simulation with polylogarithmic time[25] overhead.

We first compute $b = b(n)$ and store it on a worktape using $O(\log t) = O(s)$ bits. For $i \in \{0, \ldots, b-1\}$, we set $c_i = ti/b$. We think of c_i as dividing the timesteps of the simulation of $\mathcal{M}_1$, and call c_i the ith checkpoint.

Phase 1: Computing checkpoints We first compute and store on two worktapes the following:

$$\vec{E} = (e_0, \ldots, e_{b-1})$$
$$\vec{C} = (C_0, \ldots, C_{b-1})$$

where $e_i \in [n]$ is the number of bits output by $\mathcal{M}_1$ up to timestep c_i, and the string $C_i \in \{0,1\}^{v = s + \lceil \log n \rceil + O(\log s) + O(1)}$ is the configuration of $\mathcal{M}_1(x)$ at timestep c_i. Here the configuration includes the state of the worktapes, the location of the input and working tape heads, and the configuration of the FSM. Note that the total space to compute and store $\vec{E}, \vec{C}$ is $O(s) + b \cdot v = O(bs)$, and this step runs in time $\widetilde{O}(t + bs) = \widetilde{O}(t)$ (note that we can assume $bs < t$, since otherwise the space consumption in our statment is $O(t)$ and the statement follows by naive composition).

[24]We say $b : \mathbb{N} \to \mathbb{N}$ is nice if it can be computed in simultaneous time $O(t)$ and space $O(s)$.

[25]Inspecting the proof, the simulation of a time-t space-s k-tape computation can be simulated as a time $O(t \log s)$, space $O(s)$ two-tape computation (i.e., by using $O(s)$ "regions/zones" rather than $O(t)$).

The time bound follows as we simulate $\mathcal{M}_1$ once (tracking the number of elapsed steps and the number of output bits), and b times in this simulation halt and record $O(s)$ bits onto the separate tapes.

Phase 2: Simulating the outer machine. Subsequently, we simulate $\mathcal{M}_2$ on a virtual input $y = g_1(x)$ (using a separate pair of worktapes for this simulation), and write the output as it is produced to the final output tape.

We track the location of the tape head of $\mathcal{M}_2$ on the virtual input tape using $O(\log n)$ bits of space. At each step of the simulation, if the input head of $\mathcal{M}_2$ is on y_j, we pause the simulation. Next, we compute the value $\textsc{blk}(j) \in [b]$ such that

$$e_{\textsc{blk}(j)} \le j < e_{\textsc{blk}(j)+1}$$

i.e. the block where $\mathcal{M}_1(x)$ computes the jth bit of output. We compute $\textsc{blk}(j)$ by moving the head on the tape holding $\vec{E}$. If the current head location is at the start of e_k and $j \ge e_k$, we move right, and otherwise move left, until we find the value of k for which $e_k \le j < e_{k+1}$, and let this value be $\textsc{blk}(j)$. Once we find $\textsc{blk}(j)$, we move the head on the tape holding $\vec{C}$ to before $C_{\textsc{blk}(j)}$, copy $C_{\textsc{blk}(j)}$ to a separate set of worktapes, and begin to simulate $\mathcal{M}_1(x)$ from configuration $C_{\textsc{blk}(j)}$. Once this simulation has produced its $j - e_{\textsc{blk}(j)}$th bit of output y_j, we halt the simulation and return control to $\mathcal{M}_2$ with the correct value y_j.

We now show the space and time are as claimed. The space follows from the analysis of the first phase, as the second phase runs in space $O(s + \log n) = O(s)$.

For time, note that in the second phase we perform at most t queries to the virtual tape holding $y = \mathcal{M}_1(x)$. Let the sequence of queried locations be $q_1, \ldots, q_t$. Since we are working in the multi-tape Turing machine model, we have that for every i,

$$|q_i - q_{i+1}| \le 1. \tag{1}$$

We claim we can compute $y_{q_1}, \ldots, y_{q_t}$ (and hence perform the overall simulation in the second phase) in time $O(t^2 s/b + ts)$. The time to produce y_{q_i} is as follows:

1. The time to compute the value $\textsc{blk}(q_i)$, given q_i.
2. The time to set up the simulation of $\mathcal{M}(x)$ from configuration $C_{\textsc{blk}(q_i)}$.
3. The time to simulate $\mathcal{M}_1$ until it produces y_{q_i}.

First, we maintain after query q_i that the the tape heads for $\vec{E}, \vec{C}$ are positioned before $e_{\textsc{blk}(q_i)}$ and $C_{\textsc{blk}(q_i)}$ respectively. Next, we must determine the location of $\textsc{blk}(q_{i+1})$. By Equation (1) we only need to examine the immediately adjacent locations $e_{\textsc{blk}(q_{i+1}\pm 1)}$. Thus the time to determine the next value $\textsc{blk}(q_{i+1})$ (and move the heads to immediately before $e_{\textsc{blk}(q_{i+1})}$ and $C_{\textsc{blk}(q_{i+1})}$) is $O(s)$ per step.

Second, with the heads in this position we can copy $C_{\textsc{blk}(q_{i+1})}$ and simulate $\mathcal{M}_1$ from this configuration for at most t/b steps in time $O(t/b)$, so the latter two steps run in time $O(s + t/b)$ per query to the virtual input tape. Thus, our total runtime is $\widetilde{O}(t(t/b) + ts)$ as claimed (and note that the logarithmic time loss comes from simulating this machine with a two-tape machine).

Finally, we claim that the second factor in the time complexity is never the dominant term and hence we can omit it. If $t^2/b \le ts$, then $t \le bs$ and the statement follows from naive composition. $\square$

4.2 A Lower Bound for Composing Two Algorithms

Following the description in Section 2, we first show how to sort n elements as a composition of two algorithms running in time $\tilde{O}(n^{3/2})$ and space $O(\log n)$, and then deduce Theorem 1.2 as a corollary.

Theorem 4.2 (Sorting as a composition of two algorithms)**.** *There are length-preserving algorithms $A_1, A_2 : \{0,1\}^* \to \{0,1\}^*$ such that:*

1. *Each of A_1, A_2 is computable in time $\widetilde{O}(n^{3/2})$ and space $O(\log n)$ on inputs of length n.*
2. *For every n-bit input, interpreted as a list $L \in [\mathrm{poly}(m)]^m$ for $m = n/O(\log n)$, we have that $A_2(A_1(L)) = \mathsf{sort}(L)$.*

We will need the ability to find quantiles in nearly linear time and logarithmic space:

Lemma 4.3. *There is an algorithm that, given a list $L \in [D]^m$ and an index $k \in [m]$, outputs the element $\mathsf{sort}(L)_k$ in time $\widetilde{O}(m \log D)$ and space $O(\log(Dm))$.*

Proof. We initiate $\textsc{low} = 1$, $\textsc{high} = D + 1$ and counters $l_{\textsc{low}} = 0, l_{\textsc{high}} = m$.[26] We maintain the following invariants:

1. $l_{\textsc{low}}$ is always equal to the number of elements of L strictly less than $\textsc{low}$, and $l_{\textsc{high}}$ is the number of elements of L strictly less than $l_{\textsc{high}}$.
2. $l_{\textsc{low}} \leq k < l_{\textsc{high}}$.

The algorithm proceeds in $d = \lceil \log D \rceil$ iterations as follows. At each step, we compute

$$M = \left\lceil \frac{\textsc{low} + \textsc{high}}{2} \right\rceil$$

and determine I_M, the number of elements in $[\textsc{low}, M)$, which can be done in time $\widetilde{O}(m \log D)$ and space $O(\log(m \log D))$ by a single pass over the input list L. If $l_{\textsc{low}} + I_M \leq k$, we set

$$\textsc{low} \leftarrow M, \qquad l_{\textsc{low}} \leftarrow l_{\textsc{low}} + I_M.$$

and otherwise set

$$\textsc{high} \leftarrow M, \qquad l_{\textsc{high}} \leftarrow l_{\textsc{high}} - I_M.$$

Finally, after d iterations we have that $\textsc{high} - \textsc{low} = 1$. Then it must be the case that $\mathsf{sort}(L)_k = \textsc{low}$, as by the second invariant the kth element in $\mathsf{sort}(L)$ is not strictly less than $\textsc{low}$ but is strictly less than $\textsc{low} + 1$. □

We can now prove the upper bound:

Proof of Theorem 4.2. The algorithms work as follows. Assume without loss of generality that $\ell = \sqrt{m}$ is an integer. Let $k_i = i \cdot \ell$ for $i \in \{0, \ldots, \ell\}$.

[26]The latter counter is only used in the analysis.

The Algorithm A_1. The algorithm A_1 maintains $d_l, d_h \in [m]$, where at stage i

$$d_l = \mathsf{sort}(L)_{k_i}, \qquad d_h = \mathsf{sort}(L)_{k_{s(i)}}.$$

where $s(i)$ is defined as the least $j > i$ where

$$\mathsf{sort}(L)_{k_j} > \mathsf{sort}(L)_{k_i}.$$

Note that if all values are unique then $s(i)$ is simply $i+1$, and the definition only serves to avoid double-printing values.

We set $i = 0$ and define $\mathsf{sort}(L)_{k_0} = 0$, and iterate over $i = 0, s(0), s(s(0)), \ldots$, where stage i works as follows. First, the algorithm computes both values by calls to Lemma 4.3 in time $\widetilde{O}(m)$ and space $O(\log m)$. Next, we determine $C_l = |\{i \in [m] : L_i = d_l\}|$, the number of occurrences of d_l in the list (which we can do by a single scan through L). We then print to the output C_l copies of d_l. Next, we scan through L, and print to the output every element $y \in L$ such that

$$d_l < y < d_h$$

in the order we encounter them in L. Finally, we set $i \leftarrow s(i)$ and proceed to the next stage. After $s(i)$ reaches $\ell + 1$ and that phase is completed (we define $\mathsf{sort}(L)_{k_{\ell+1}} = \infty$), we halt.

We argue correctness as follows. Note that the output is of the form

$$V_{u_0}, U_{u_1}, \ldots, U_{u_t}, V_{u_t}$$

where $u_0, \ldots, u_t$ is the sequence of values of i processed in the loop, U_{u_j} is the number of copies of $\mathsf{sort}(L)_{k_{u_j}}$ in the list, and every $v \in V_{u_j}$ satisfies $U_{u_{j-1}} < v < U_{u_j}$. In particular, for every i where $\mathsf{sort}(L)_{k_i} = \mathsf{sort}(L)_{k_{u_j}}$ for some $j \in [t]$, the algorithm prints $\mathsf{sort}(L)_{k_i}$ to location k_i in the output in stage j. Thus, the output can equivalently be cast as

$$L_1, d_1, \ldots, d_\ell, L_\ell$$

where $d_i \leq d_{i+1}$ for every i, and $|L_i| = \sqrt{m} - 1$ for every i, and for every $y \in L_i$ we have $d_{i-1} \leq y \leq d_i$.

Note that A_1 works in at most $\ell = \sqrt{m}$ steps, and in each step it performs computation in time $\tilde{O}(m)$ and using space $O(\log m)$. Thus, A_1 runs in time $\tilde{O}(m^{3/2})$ and space $O(\log m)$.

The Algorithm A_2. The algorithm A_2 assumes that the input list is of the partially sorted form described above, and works in ℓ stages as follows. For stage i, it prints d_i, sorts the sublist L_i using the brute-force algorithm running in time $O(\ell^2 \log(m)) = \widetilde{O}(m)$ and space $O(\log m)$, and prints $\mathsf{sort}(L_i)$, then increments i. The runtime is $\tilde{O}(m^{3/2})$ since there are ℓ stages each sorting a list of $\sqrt{m}$ elements, and correctness follows from the output guarantee of A_1. □

We now use Theorem 4.2 to prove lower bounds on the overhead of composition.

Theorem 4.4 (Sorting Lower Bound [BC82,RS82,MW19]). *For every constant $\varepsilon > 0$ the following holds. Let A be an algorithm such that on infinitely many n, with probability at least ε over $L \in [n^2]^n$ it holds that $A(L) = \mathsf{sort}(L)$. Then A runs in time-space product $\widetilde{\Omega}(n^2)$. Moreover, the time-space lower bound holds even if A gets non-uniform advice.*[27]

[27]That is, the lower bound holds even when for each input length n, the algorithm is allowed to receive an arbitrary non-uniform advice string a_n on a separate read-only advice tape.

We remark that this result holds for algorithms in the RAM model as well, and thus so does our lower bound.

Theorem 1.2 (hardness of composing low-space algorithms [Wil25])**.** *There are two algorithms A_1 and A_2 each mapping n bits to n bits in linear time and space $O(\log n)$ such that for any constant $\varepsilon > 0$, any algorithm computing the composition $A_2(A_1(x))$ correctly on more than an ε-fraction of the inputs x has time-space product at least $n^{1.33}$.*

Proof. Fix $\varepsilon > 0$. Given a string $x \in \{0,1\}^n$, we interpret x as $L||y$, where $L \in [m^2]^m$ with $m(n) \overset{\text{def}}{=} n^{2/3}/\operatorname{polylog}(n)$ for some large enough $\operatorname{polylog}(n)$, and y is the remaining bits of the input. Let A_1, A_2 be the algorithms of Theorem 4.2, where we slightly modify A_1 such that it simply ignores the suffix y, so that

$$A_2(A_1(x)) = \mathsf{sort}(L).$$

By choosing the $\operatorname{polylog}(n)$ term in the definition of m to be sufficiently large, we obtain that both algorithms run in space $O(\log n)$ and time $\widetilde{O}(m^{3/2}) = O(n)$.

Finally, we claim that any algorithm computing $x \to \mathsf{sort}(L)$ on an ε fraction of x on an infinite sequence of input lengths must have time-space product $n^{1.33}$.

Suppose for contradiction there was some algorithm A running in time-space $O(n^{1.33})$ that computes $x \to \mathsf{sort}(L)$ with probability at least ε over the input $L||y$, for an infinite sequence of input lengths n. By an averaging argument, for infinitely many n there exists a fixing y_n such that

$$\Pr_{L \sim \mathbf{U}_{[m^2]^m}}[A(L||y_n) = \mathsf{sort}(L)] \geq \varepsilon.$$

We modify A into an algorithm with advice by hardwiring this string y_n on each input length $m(n)$; this adds at most a multiplicative $\operatorname{polylog}(n)$ time overhead (from tracking where the input head of A exceeds the input L, and answering instead with the advice string). Then for an infinite sequence of input lengths m, $A_{y_n}(L) = \mathsf{sort}(L)$ with probability at least ε over L, and A runs in time-space $\widetilde{O}(n^{1.33}) \leq m^{2-.0001}$. This contradicts the time-space lower bound of $\widetilde{\Omega}(m^2)$ of Theorem 4.4. □

4.3 Partial Evidence for General Hardness of Composition

In this section we present partial evidence suggesting that generalized versions of Theorem 1.2 are true; specifically, that time-space lower bounds for composition hold also for large polynomial time, and also for k-fold composition (with a time-space overhead that grows with k).

First, we detail the connection between two prior conjectures in complexity theory and hardness of composition. Then we present a reduction of proving hardness of composing two algorithms to proving hardness of composing $O(1)$ algorithms. Finally we present a candidate hard problem for composition, and explain why it seems hard.

4.3.1 Evidence from complexity-theoretic conjectures

Consider the classical conjecture that $\mathsf{NC} \not\subseteq \mathsf{SC}$; that is, there is a problem computable by logspace-uniform circuits of polynomial size and polylogarithmic depth that is hard for polynomial-time algorithms using space $\operatorname{polylog}(n)$ (see, e.g., [Coo79], [Coo81, Section 7], [Bor77, Open Problem 2], [GHR95, Chapter 5.3]). Since evaluating a polynomial size circuit of depth $O(\log^i n)$ can be computed as a $k = \log^{i-1}(n)$-fold composition of a function computable in quasilinear time and logspace (i.e., the function that outputs the gate values in the next layer when given the gate values of current layer), a separation $\mathsf{NC}^i \not\subseteq \mathsf{SC}$ would imply that there is no polynomial time, polylogarithmic space algorithm for $\log^{k-1}(n)$-fold composition of a quasilinear time, logspace function.

Quantitatively stronger evidence comes from hypothesized time-space tradeoffs for directed connectivity. To see this, recall the classical question of whether Savitch's theorem [Sav70] for $s \to t$ connectivity in directed graphs can be improved: Savitch's algorithm runs in space $O(\log^2 n)$ and time $n^{O(\log n)}$, and an open question is whether there is an algorithm running in space $n^{.99}$ and time $n^{O(\log^{0.99} n)}$ (and see also [LV20], [Coo79], [Wig92, Section 3.1]).[28] Since $s \to t$ connectivity can be computed as a $\log(n)$-fold composition of a space $O(\log n)$, linear-time algorithm, if the algorithm cannot be improved in this way, then the time overhead for $(k = \log n)$-wise composition must be $n^{\Omega(k^{1-\varepsilon})}$ for every $\varepsilon > 0$, even if the space of the algorithm computing the composition is allowed to be $n^{0.99} \gg \mathrm{polylog}(n)$.

4.3.2 A reduction to proving hardness of composing $O(1)$ algorithms

Suppose that we want to prove the hypothesis in Theorem 1.3; that is, a time-space tradeoff of $t^{1+\delta}$ for composing an algorithm running in time t and space $\mathrm{polylog}(t)$.

We observe that a lower bound on a constant number of compositions implies a (quantitatively weaker) bound on a single composition. In particular, proving a time-space lower bound of $t^{1+\delta}$ for some $\delta > 0$ on a single composition reduces to proving a time-space lower bound of $t^{1+\varepsilon}$ for some $\varepsilon > 0$ on $(k = O(1))$-fold compositions. (Note that in the target of the reduction, we do not need a lower bound of $t^{\Omega(k)}$, but rather only a lower bound of $t^{1+\varepsilon}$.)

Proposition 4.5. *Suppose there is $\varepsilon > 0$ and $k = 2^\ell$ and a length-preserving function g computable in simultaneous time $t(n)$ and space* $\mathrm{polylog}(n)$ *such that every algorithm computing $x \to g^{(k)}(x)$ in space* $\mathrm{polylog}(n)$ *requires time $\Omega(t^{1+\varepsilon})$. Then, for some polynomial $t'(n) \geq t(n)$, there is a length-preserving function g' computable in time t' and* $\mathrm{polylog}(n)$ *space such that every algorithm computing $x \to g'(g'(x))$ in space* $\mathrm{polylog}(n)$ *requires time $\Omega((t')^{1+\delta})$, where $\delta = (1+\varepsilon)^{1/\ell} - 1 > 0$.*

Proof. Assume for contradiction that for every polynomial $t'(n) > t(n)$ and every t'-time, polylog-space computable function g', the composition $g' \circ g'$ can be computed in space $\mathrm{polylog}(n)$ and time $c \cdot (t')^{r^{1/\ell}}$, where $r = 1 + \varepsilon$ and the constant c may be arbitrarily small. We will construct a machine computing $g^{(k)}$ that violates the assumed lower bound.

The construction is iterative. We proceed in ℓ stages, where at stage i we have a machine that computes the length-preserving function g_{i-1} defined as

$$x \to g^{(2^{i-1})}(x)$$

in time t_i and space $\mathrm{polylog}(n)$, where we define t_i inductively in terms of t_{i-1}.

Setting $t_1 = t$, the base case follows from the machine that computes g. Assuming that g_{i-1} is computable in time t_i for stage i, by assumption there is a machine computing

$$x \to g^{(2^i)}(x)$$

in time $t_{i+1} = c \cdot t_i^{r^{1/\ell}}$ and $\mathrm{polylog}(n)$ space. After ℓ iterations, we obtain that $x \mapsto g^{(k)}(x)$ is computable in time

$$c^\ell \cdot t^{(r^{1/\ell})^\ell} = c^\ell \cdot t^{1+\varepsilon}$$

and space $\mathrm{polylog}(n)$, where c^ℓ is arbitrarily small; this violates the assumption.

[28] In restricted models, such a lower bound holds unconditionally [BS83, Tom82, EPA99].

Hence, for some constant $c > 0$ and some $i \in [\ell]$, the function g_{i-1}, which is length-preserving and computable in time $t_{i+1} > t$ and space $\mathrm{polylog}(n)$, cannot be composed once in time $t_{i+1}^{(1+\varepsilon)^{1/\ell}} = t_{i+1}^{1+\delta}$ and $\mathrm{polylog}(n)$ space. □

4.3.3 A candidate hard problem

Finally, we present a candidate problem for proving that there are functions computable in time t and (separately) space $O(\log t)$, such that every algorithm computing their composition must run in time $t^{1+\Omega(1)}$ for space $\mathrm{polylog}(n)$. We think of t as a large polynomial in the input length n.

The basic version. Fix a function f that on inputs of length $O(\log n)$ runs in time $t'(n) = t(n)/n$ and space $O(\log t)$, and assume that f is hard to compute in space $\mathrm{polylog}(n)$ and time less than $t'(n)$. Moreover, assume that f supports a strong direct product result: any algorithm running in space $\mathrm{polylog}(n)$ cannot batch-compute f on inputs $x_1, ..., x_{d=n^{\varepsilon}}$ in time $t'(n) \cdot d^{1-o(1)}$, even when given $\mathrm{polylog}(n)$ bits of advice that depends on $x_1, ..., x_d$. Then, the candidate hard function is

$$x_1, ..., x_{n/O(\log n)} \mapsto \mathsf{sort}\big(f(x_1), ..., f(x_{n/O(\log n)})\big) \tag{4.1}$$

Relying on Theorem 4.2, the function in Eq. (4.1) is computable as the composition of three functions computable in time $t(n)$ and space $O(\log t)$.

It is natural to suspect that computing Eq. (4.1) in space $\mathrm{polylog}(n)$ should take time $t^{1+\Omega(1)}$. This is because the time-space lower bound for sort asserts that any $\mathrm{polylog}(n)$-space algorithm must make $n^{1.99}$ queries to its input (for simplicity we ignore the difference between n and $n/O(\log n)$), and by the properties of f, answering each query takes time essentially t. The assumption that f is hard to batch-compute even given advice ensures that the prior state of the machine is not helpful for batch-computing f at any given moment in the execution.

Problem 4.6. Under plausible complexity-theoretic assumptions, prove that for some f, the function in Eq. (4.1) requires a time-space trade-off of $t^{1+\Omega(1)}$; or provide evidence to the contrary.

Intuitively, the difficult part in affirmatively resolving Problem 4.6 is ensuring that there are no "interactions" between f and sort, in the sense that the computation used for sort cannot be used to speed-up the computation of f (or vice versa).

Remark 4.7. Indeed, the candidate above is based on $\mathsf{block\ composition}$, and is thus reminiscent of the well-known KRW conjecture [KRW95]. However, in contrast to the KRW conjecture, we are focused on time-space tradeoffs for *uniform algorithms*, and we are not aware of any result indicating that hardness of our candidate would imply lower bounds for non-uniform circuits. Moreover, the KRW conjecture asserts that block-composition of *every* pair of functions is hard (for formulas); in contrast, when studying low-space algorithms as in our work, composing certain functions is easy (e.g., read-once logspace algorithms can be composed in linear time and logspace), and thus our candidate refers to the block-composition of two specific hard functions.

A more general version. Building on the same intuition, we now present a more general candidate, which may support a stronger result. Specifically, for this candidate it may be possible to prove a time-space tradeoff of $t^{2-\varepsilon}$ for a single composition.

Definition 4.8. We let the $\mathsf{machine\ composition}$ problem be defined as follows. On input

$$(M', M, x_1, \ldots, x_n) \in \{0,1\}^{\widetilde{O}(n)}$$

where $x_i \in \{0,1\}^{\text{polylog}(n)}$, and M' and M are polylog(n)-sized descriptions of Turing machines that run in time $t(n)$ and $t(n)/n$ respectively and space $O(\log n)$ on inputs of size polylog(n) (and output a single bit), the output is

$$M'(M(x_1),\ldots,M(x_n)).$$

Observe that machine composition can be written as $f \circ g$, where both f and g are computable in simultaneous space $O(\log n)$ and time t.

Problem 4.9. Under plausible complexity-theoretic assumptions, prove that the machine composition problem requires a time-space tradeoff of $t^{2-o(1)}/n$; or provide evidence to the contrary.

Generalizing the basic version, the properties we are hoping for is that computing M' in space polylog(n) requires $\approx t$ queries to its input $M(x_1),...,M(x_n)$, and that answering each query in space polylog(n) (i.e., computing M) takes time essentially t/n.

5 A Generator with Uniform Deterministic Logspace Reconstruction

In this section we design a reconstruction procedure for the Shaltiel-Umans [SU05] generator that runs in *deterministic* logspace. The algorithm is a modified version of the reconstruction procedure from [DPTW25]. In terms of presentation, we find it infeasible to point out the relevant changes without surveying the entire construction (which is complicated and involves many parameters and sub-components). Therefore we will present the entire construction – most of it identical to [DPTW25] – while marking the relevant new parts or significant changes in blue.

Theorem 5.1 (a somewhere-PRG with uniform deterministic logspace reconstruction)**.** *Let $M\colon \mathbb{N} \to \mathbb{N}$ be a logspace-computable function such that $M(N) \le N^{\varepsilon_{\mathsf{SU}}}$, where $\varepsilon_{\mathsf{SU}} > 0$ is a universal constant. Then, there exist a pair of algorithms* SU *and* RSU *that for every $f \in \{0,1\}^N$ satisfy the following.*

1. *When* SU *is given input 1^N and oracle access to f it runs in space $O(\log N)$ and prints a collection $L_1,...,L_\ell$ where each L_i is a list of* poly(N) *strings of length $M = M(N)$, where $\ell = O(\log(N)/\log(M))$.*
2. *For each $i \in [\ell]$, let $j_i \le M$, and let $P_i\colon \{0,1\}^{j_i} \to \{0,1\}$ be a $(1/M^2)$-next-bit-predictor for the uniform distribution on L_i. Then, when* RSU *gets input 1^N and oracle access to f and to $P_1,...,P_\ell$, it runs in space $O(\log N) +$ polylog(M), and prints a (deterministic) oracle circuit $C\colon \{0,1\}^{\log(N)} \to \{0,1\}$ of size* poly(M) *such that $C^{P_1,...,P_\ell}(x) = f(x)$ for all $x \in [N]$.*

Recall that the standard model of space-bounded oracle machines allows the machine to specify queries on a dedicated "write-only" tape, and then enter a "query" mode in which the oracle reads the query on the tape, erases the tape's content, and returns the answer. This model facilitates replacing the oracle by another space-bounded machine, in which case the composition can be implemented in space that is additive in the space of both machines but *not* in the length of the query (see, e.g., [Gol08, Exercise 5.7]). This is important in the specific case of RSU from Theorem 5.1, which runs in space $O(\log N)$ but makes queries that may be much longer (i.e., of length $M-1$).

In Section 5.1 we present the arithmetic setting for the generator and reconstruction, as well as a few preliminary technical lemmas. In Section 5.2 we present the generator itself. Then, in Section 5.3 and Section 5.4 we present two stand-alone parts of the reconstruction procedure, and in Section 5.5 we present the full reconstruction procedure.

5.1 Arithmetic Setup

Throughout our argument, we will denote the input by $x \in \{0,1\}^n$ (rather than $f \in \{0,1\}^N$) and the output length by m instead of M. We also assume without loss of generality that $m \geq \log(n)$ (otherwise, the generator can trivially output all m-bit strings).

5.1.1 Arithmetic setting

For input length $n \in \mathbb{N}$ and output length $m \leq n$, and for a constant $\varepsilon = \varepsilon_{\mathsf{SU}} > 0$:

- **(Field.)** Let $q = \Theta(m \cdot \log(n))^c$ be a prime power, where $c > 1$ is a sufficiently large universal constant. We consider $\mathbb{F}_q$ as an extension of a subfield $\mathbb{F}_{q_0}$ of size $q_0 = \Theta(m \cdot \log n)$; note that the extension degree is a constant $\Delta = \Theta(c)$.
- **(Degree.)** Let $d = m^\varepsilon$.
- **(Number of variables.)** Let $v = O_\varepsilon(\log(n)/\log(m))$ such that $v \geq (1/\varepsilon) \cdot \log(n/\log(q))/\log(d)$.
- **(Prediction advantage.)** Let $\rho = 1/2m^2$.

Given $x \in \{0,1\}^n$, treat it as a list of $\lfloor n/\log(q) \rfloor$ coefficients specifying a polynomial $\hat{x} \colon \mathbb{F}_q^v \to \mathbb{F}_q$ of degree d. By our lower bound on v we have $\binom{d+v}{d} \geq \lfloor n/\log(q) \rfloor$, and therefore all the coefficients specified by x are useful towards defining $\hat{x}$; in particular different x's give rise to different polynomials $\hat{x}$.[29]

Fact 5.2. There is an algorithm that gets input n, m, q, Δ satisfying the constraints above, runs in space $O(\log n)$, and outputs a representation of $\mathbb{F}_q$.

Proof. Let $q = p^r$ for a prime p. The algorithm enumerates over degree-$(r-1)$ polynomials $\mathbb{F}_p \to \mathbb{F}_p$, each of which is represented by $r \cdot \log(p) = \log(q) < O(\log n)$ bits, and tests each polynomial u for irreducibility. The latter test is also done by brute-force, enumerating over all polynomials $\mathbb{F}_p \to \mathbb{F}_p$ of degree at most $r-2$ and checking if there is one that divides u. $\square$

Fact 5.3. There is an algorithm that gets as input a representation of $\mathbb{F}_q$ and an integer $v \in \mathbb{N}$, runs in space $O(v \cdot \log(q))$, and prints a monic irreducible polynomial $u^\star \in \mathbb{F}_q[x]$ of degree $v-1$.

Proof. The algorithm works by brute-force, analogously to the proof of Fact 5.2. Each polynomial $\mathbb{F}_q \to \mathbb{F}_q$ of degree $v-1$ is represented by $O(v \cdot \log(q))$ bits. $\square$

Due to Fact 5.2 and 5.3, from now on we will assume that all of our space-bounded algorithms have access to a fixed representation of $\mathbb{F}_{q^v}$, in the form of the irreducible polynomial $u^\star \in \mathbb{F}_q[x]$ produced by the algorithm above.

5.1.2 A generator matrix in logspace

We will need an algorithm that prints powers of a generator matrix for $\mathbb{F}_q^v$ in space $O(v \cdot \log(q))$. We first define this notion, and show that several basic operations in $\mathbb{F}_{q^v}$ and in $\mathbb{F}_q^v$ can be done in small space. Then, we construct the algorithm for printing powers of a generator matrix.

Definition 5.4. We say that $A \in \mathbb{F}_q^{v \times v}$ is a $\mathsf{generator\ matrix\ for}$ $\mathbb{F}_q^v$ if $\left\{A^i \cdot \vec{s}\right\}_{i \in [q^v - 1]} = \mathbb{F}_q^v \setminus \left\{\vec{0}\right\}$ for any non-zero $\vec{s} \in \mathbb{F}_q^v$.

[29] When x is too short to specify all the coefficients of a polynomial of degree d, we consider the polynomial $\hat{x}$ obtained by padding x with zeroes to the appropriate length.

Claim 5.5. *There is an algorithm that gets input $A \in \mathbb{F}_q^{v\times v}$ and $i \in [q^v - 1]$, runs in space $O(\log(i)\cdot\log(q))$, and prints A^i.*

Proof. Consider the binary tree of depth $\lceil\log(i)\rceil$ with i leaves labeled by A and $2^{\lceil\log(i)\rceil} - i$ leaves labeled by the identity matrix, and each node labeled by the multiplication of the labels of its children. Computing each entry of the label of each node can be done in space $O(\log(v)+\log(q)) = O(\log q)$ with query access to the labels of its children. The algorithm prints each entry of the top node, and simulates the query access of each node by space-efficient composition; the space complexity is thus $O(\log(i)\cdot\log(q))$. □

Claim 5.6. *There is an algorithm that gets input $\omega \in \mathbb{F}_{q^v}$, runs in space $O(\log(v)\cdot\log(q))$, and prints the matrix $T_\omega \in \mathbb{F}_q^{v\times v}$ that represents multiplication by ω in $\mathbb{F}_q^v$.*[30]

Proof. Let $C_{u^\star} \in \mathbb{F}_q^{v\times v}$ be the companion matrix of the irreducible $u^\star$ from Fact 5.3, and recall that $C_{u^\star} = T_x$ where $x \in \mathbb{F}_q[x]/(u^\star)$ is the identity polynomial. Also recall that $C_{u^\star}$ has a very simple structure,[31] and in particular there is an algorithm that (given $u^\star$) prints $C_{u^\star}$ in space $O(\log(v)+\log(q))$.

Let $\omega = \sum_{i=0}^{v-1}\omega_i x^i$. Then, $T_\omega = \sum_{i=0}^{v-1}\omega_i C_{u^\star}^i$. Using Claim 5.5, we can print each $C_{u^\star}^i$ in space $O(\log(v)\cdot\log(q))$, and hence we can also print T_ω in such space. □

Proposition 5.7. *There is a generator matrix A for $\mathbb{F}_q^v$ and an algorithm A' such that A' gets input $i \in [q^v - 1]$, runs in space $O(\log(n))$, and prints A^i.*

Proof. The algorithm first finds a primitive element $\omega \in \mathbb{F}_{q^v}$, by brute-force. That is, it enumerates over elements of $\mathbb{F}_{q^v}$, and for each element ω' it raises it to the powers $i = 2, 3, ..., q^v - 1$ and checks whether any intermediate result is 1. This can readily be done in space $O(v\log q)$.

Now, let ω be the first primitive element encountered, and recall that $A = T_\omega$ is a generator matrix for $\mathbb{F}_q^v$. The algorithm raises ω to the power i, and then uses Claim 5.6 to compute $T_{\omega^i} = T_\omega^i = A^i$. The proposition follows, noting that $v\log q = O(\log n)$. □

5.1.3 A standard list-decodable code

Our construction will use a logspace-computable list decodable code. We do not need particularly tight parameters, and the classical construction of Sudan, Trevisan, and Vadhan [STV01] suffices for us. (We do not even rely on the locality of the decoder in their construction.)

Theorem 5.8 (a list-decodable code; see [STV01])**.** *There is a universal constant $c_{\mathsf{STV}} > 1$ and algorithm $\mathsf{Enc}_{\mathsf{STV}}$ that maps $x \in \{0,1\}^{\log(q)}$ to $\mathsf{Enc}_{\mathsf{STV}}(x) \in \{0,1\}^{\ell_q = \mathrm{poly}(\log(q),1/\rho)}$ such that the mapping yields a $(\frac{1}{2}-\rho, \bar{\rho} = (1/\rho)^{c_{\mathsf{STV}}})$-list-decodable code, $\mathsf{Enc}_{\mathsf{STV}}$ runs in space $O(\log q)$, and the list-decoder $\mathsf{Dec}_{\mathsf{STV}}$ runs in time $\mathrm{poly}(\log(q), 1/\rho)$.*

5.2 The Generator

On an input $x \in \{0,1\}^n$, G first encodes x as a polynomial $\hat{x}\colon \mathbb{F}_q^v \to \mathbb{F}_q$ of (total) degree d.

[30] That is, consider the $\mathbb{F}_q$-basis $\{1, x, x^2, ..., x^{v-1}\}$ for $\mathbb{F}_q^v$, and the corresponding bijection $\xi\colon \mathbb{F}_{q^v} \to \mathbb{F}_q^v$ (i.e., ξ maps a polynomial $\sum_{i\in\{0,...,v-1\}} a_i x^i$ to $(a_0, ..., a_{v-1})$). Then, for every $\omega, \nu \in \mathbb{F}_{q^v}$ we have that $T_\omega \cdot \xi(\nu) = \xi(\omega\cdot\nu)$.

[31] Specifically, the coefficients of $u^\star$ appear in its rightmost column, and otherwise all of the entries in the matrix are zero except for one subdiagonal whose entries are one.

The lists $L_0, ..., L_{v-1}$. We first define "q-ary lists" whose elements are vectors in $\mathbb{F}_q^m$, and then define the final output lists L_i whose elements are strings in $\{0,1\}^m$. For every $i = 0, ..., v-1$, the i^{th} q-ary list $L_i^{(q)} \subseteq \mathbb{F}_q^m$ that $G(x)$ outputs is defined as follows. For every $\vec{s} \in \mathbb{F}_q^v$, the generators includes in $L_i^{(q)}$ the m-element string

$$L_i^{(q)}(\vec{s}) = \hat{x}(A^{1 \cdot q^i} \cdot \vec{s}) \circ \hat{x}(A^{2 \cdot q^i} \cdot \vec{s}) \circ ... \circ \hat{x}(A^{m \cdot q^i} \cdot \vec{s}), \tag{5.1}$$

where A is the generator matrix given by Proposition 5.7.

Then, for every $\vec{s} \in \mathbb{F}_q^v$ and $j \in [\ell_q]$, the corresponding m-bit string in L_i is

$$L_i(\vec{s}, j) = \mathsf{Enc}_{\mathsf{STV}}\left(L_i^{(q)}(\vec{s})_1\right)_j \circ \mathsf{Enc}_{\mathsf{STV}}\left(L_i^{(q)}(\vec{s})_2\right)_j \circ ... \circ \mathsf{Enc}_{\mathsf{STV}}\left(L_i^{(q)}(\vec{s})_m\right)_j .$$

The list L_v. In addition, the generator outputs the list $L_v \subseteq \{0,1\}^m$ defined as follows. Let $\mathsf{pow}\colon \{0,1\}^{v \cdot \log(q)} \to \{0,1\}^{v \cdot \log(q)}$ be the function that parses its input $i \in \{0,1\}^{v \cdot \log(q)}$ as an integer $i \in \{0, ..., q^v - 1\}$ and outputs $\mathsf{pow}(i) = A^i \cdot \vec{1}$ (in binary representation). Then, for every $i \in \{0,1\}^{v \cdot \log(q)}$ and $z \in \{0,1\}^{v \cdot \log(q)}$ the generator outputs

$$L_v(i, z) = \left\langle \mathsf{pow}^{(m-1)}(i), z \right\rangle \circ \left\langle \mathsf{pow}^{(m-2)}(i), z \right\rangle \circ ... \circ \langle \mathsf{pow}(i), z \rangle \circ \langle i, z \rangle ,$$

where $\mathsf{pow}^{(j)}$ is the j-wise repeated composition of pow.

Complexity. Note that there are $v + 1 = O(\log(n)/\log(m))$ lists, and each list contains at most $q^{2v} \cdot \text{poly}(m) = \text{poly}(n)$ strings of m field elements.

Also note that the generator is computable in space $O(\log n)$. To see this, for each L_i with $i < v$, and for each fixed $\vec{s}$, observe that computing each output element reduces in space $O(\log n)$ to computing $A^i \cdot \vec{s}$, which can be done in space $O(\log n)$ using Proposition 5.7. For L_v, given any fixed $(z, i) \in \{0,1\}^{v \cdot \log(q)} \times \{0,1\}^{v \cdot \log(q)}$, the bottleneck is computing $\mathsf{pow}^{(j)}(i)$. To do so, observe that the output of pow is of length $v \cdot \log(q)$; hence, we can iteratively compute $\mathsf{pow}^{(j)}(i)$ by storing the output of each iteration and computing pow again.

5.3 The Reconstruction Procedure for L_v

Let $P_v\colon \{0,1\}^{i_v} \to \{0,1\}$ be the $(1/m^2)$-next-bit-predictor for L_v, where $i_v < m$.

Proposition 5.9 (efficiently printing a circuit that computes discrete log)**.** *There is an algorithm R_{dl} that on input 1^n and with oracle access to P_v runs in space $O(\log n)$ and outputs an oracle circuit C_{dl} of size $\text{poly}(m)$ such that $C_{\mathsf{dl}}^{P_v}(A^i \cdot \vec{1}) = i$ for all $i \in \{0,1\}^{v \cdot \log(q)}$.*

Proof. The algorithm R_{dl} will be the combination of three algorithms R_1, R_2, R_3 that print three corresponding circuits C_1, C_2, C_3. We first describe the three algorithm, and then explain how to combine them to get a single algorithm and a single circuit.

Consider the oracle circuit C_1 that gets $y = A^i \cdot \vec{1} = \mathsf{pow}(i) \in \{0,1\}^{v \cdot \log(q)}$ and tries to find i. The circuit gets an additional input $z \in \{0,1\}^{v \cdot \log(q)}$, computes

$$w_{y,z} = \left\langle \mathsf{pow}^{(i_v-1)}(y), z \right\rangle \circ \left\langle \mathsf{pow}^{(i_v-2)}(y), z \right\rangle \circ ... \circ \langle \mathsf{pow}(y), z \rangle \circ \langle y, z \rangle ,$$

and outputs $P_v(w_{y,z})$. The distribution obtained by uniformly choosing $i \in \{0,1\}^{v \cdot \log(q)}$ and setting $y = A^i \cdot \vec{1}$ is identical to the distribution obtained by uniformly choosing $y_0 \in \{0,1\}^{v \cdot \log(q)}$ and setting

$i = \mathsf{pow}^{(m-1-i_v)}(y_0)$ and $y = A^i \cdot \vec{1}$. With probability at least $1/2 + 1/m^2$ over (y_0, i, y) chosen from the latter distribution and over z, the predictor satisfies $P_v(w_{y,z}) = \mathsf{pow}^{(m-1-i_v)}(y_0) = i$. Since the distributions are identical, we have that

$$\Pr_{i,z}[C_1^{P_v}(A^i \cdot \vec{1}, z) = \langle i, z\rangle] \geq 1/2 + \varepsilon_1, \tag{5.2}$$

where $\varepsilon_1 = 1/m^2$. Observe that C_1 is of size $\mathrm{poly}(m, \log(n)) = \mathrm{poly}(m)$ and that it can be printed by a machine R_1 running in space $O(\log n)$. For simplicity, we will denote $C_1 = C_1^{P_v}$.

Also observe that the function computed by C_1 can be computed in space $O(\log n)$ with oracle access to P_v. To do this, given input (y, z), for each $j = i_v - 1, ..., 0$ we compute $\mathsf{pow}^{(j)}(y)$ and print it to the oracle-query tape along with z. Computing $\mathsf{pow}^{(j)}(y)$ is done by repeatedly invoking Proposition 5.7 (recalling that the output of pow is of length $v \cdot \log(q) = O(\log n)$).

The next procedure will compute the mapping $A^i \cdot \vec{1} \mapsto i$ correctly on $\varepsilon_2 = \mathrm{poly}(\varepsilon_1)$ fraction of the i's. We will use a space-efficient and randomness-efficient version of the Goldreich-Levin [GL89] decoding algorithm, given by Doron, Pyne, and Tell [DPT24] following Pyne, Raz, and Zhan [PRZ23]:

Theorem 5.10 (efficient Goldreich-Levin decoding; see [DPT24, Theorem 5.16]). *Let $k = v \cdot \log(q)$. There is an algorithm $\mathsf{Dec}_{\mathsf{GL}}$ that gets input $\varepsilon_1, \delta > 0$ and a random seed s_{GL} of length $O(k + \log(1/\delta))$, runs in space $O(\log(k/\varepsilon_1) + \mathrm{loglog}(1/\delta))$, and outputs a list of $L_{\mathsf{GL}} = O((k/\varepsilon_1^2) \cdot \log(1/\delta))$ oracle circuits $C_{s_{\mathsf{GL}},1}, ..., C_{s_{\mathsf{GL}},L_{\mathsf{GL}}}$ satisfying the following.*

- *Each $C_{s_{\mathsf{GL}},i}$ is an oracle TC^0 circuit of size $\mathrm{poly}(k/\varepsilon_1)$ with one majority gate that makes non-adaptive oracle queries.*
- *For every $i \in \{0,1\}^{v \cdot \log(q)}$ and every $C_1^{(i)} \colon \{0,1\}^{v \cdot \log(q)} \to \{0,1\}$ satisfying $\Pr_z[C_1^{(i)}(z) = \langle i, z\rangle] \geq 1/2 + \varepsilon_1/2$, we have that*

$$\Pr_{s_{\mathsf{GL}}}\left[\exists j \in [L_{\mathsf{GL}}] : \forall u \in [k],\ C_{s_{\mathsf{GL}},j}^{C_1^{(i)}}(u) = i_u\right] \geq 1 - \delta.$$

Consider the algorithm R_2 that gets input $A^i \cdot \vec{1}$, draws a random $s_{\mathsf{GL}} \in \{0,1\}^{O(k + \log(1/\delta))}$ and $j \in [L_{\mathsf{GL}}]$, executes $\mathsf{Dec}_{\mathsf{GL}}$ with values $\varepsilon_1/2$ and $\delta = 1/2$, and prints an oracle circuit C_2 that gets input $A^i \cdot \vec{v}$ and outputs the truth-table of $C_{s_{\mathsf{GL}},j}^{C_1^{(i)}}$, where $C_1^{(i)}(z) = C_1(A^i \cdot \vec{1}, z)$. By Eq. (5.2), with probability at least $\varepsilon_1/2$ over $i \in \{0,1\}^{v \cdot \log(q)}$ it holds that $\Pr_z[C_1(A^i \cdot \vec{1}, z) = \langle i, z\rangle] \geq 1/2 + \varepsilon_1/2$, and for each such i, by Theorem 5.10, with probability at least $(1-\delta)/L_{\mathsf{GL}}$ over s_{GL}, j the truth-table of $C_{s_{\mathsf{GL}},j}^{C_1^{(i)}}$ is i. Hence, $\Pr_{i,s_{\mathsf{GL}},j}[C_2^{C_1}(A^i \cdot \vec{1}) = i] \geq \frac{\varepsilon_1 \cdot (1-\delta)}{2L_{\mathsf{GL}}} = \Omega(\varepsilon_1/(m^4 \cdot \log(n)))$. It follows that with probability at least $\Omega(\varepsilon_1/(m^4 \cdot \log(n)))$ over the random choices s_{GL}, j of R_2, we have that $\Pr_i[C_2^{C_1}(A^i \cdot \vec{1}) = i] \geq \varepsilon_2 = \Omega(\varepsilon_1/(m^4 \cdot \log(n)))$. Note that C_2 is of size $\mathrm{poly}(m, \log(n)) = \mathrm{poly}(m)$, and that R_2 can print it in space $O(\log n)$ (relying on the fact that $\mathsf{Dec}_{\mathsf{GL}}$ runs in space $O(\log(v \cdot \log(q) \cdot m^2)) = O(\log n)$). Also, the number of random coins that R_2 uses is $O(k + \log(1/\delta) + \log(L_{\mathsf{GL}})) = O(v \cdot \log(q) + \log(m)) = O(\log n)$.

We now explain how to modify R_2 so that it does not use random coins. The modified algorithm enumerates over choices of coins s_{GL}, j, trying to find a choice such that

$$\Pr_i[C_2^{C_1}(A^i \cdot \vec{1}) = i] \geq \varepsilon_2.$$

For every fixed (s_{GL}, j), the modified R_2 enumerates over $i \in \{0,1\}^{v \cdot \log(q)}$, and for each i checks whether $C_2^{C_1}(A^i \cdot \vec{q}, s_{\mathsf{GL}}, j) = i$. Thus, it is just left to verify that given input s_{GL}, j, i, we can compute $C_2^{C_1}(A^i \cdot \vec{1}, s_{\mathsf{GL}}, j)$ in space $O(\log n)$.

By Proposition 5.7, we can compute each output bit of the mapping $i \mapsto A^i$ in space $O(\log n)$, and by the discussion after Eq. (5.2), we can compute C_1 in space $O(\log n)$ (with oracle access to P_v). Relying on standard composition of space-bounded algorithms, we only need to show that C_2 can be computed in space $O(\log n)$. This is true since each of the $v \cdot \log(q)$ output bits of C_2 is simply an output of $C_{s_{\mathsf{GL}},j}$, and by Theorem 5.10 we have that $C_{s_{\mathsf{GL}},j}$ is a $\mathbf{TC}^0$ circuit of size $\mathrm{poly}(v \cdot \log(q))$. We can simulate this circuit in space logarithmic in its size, with access to the description of the circuit given by the algorithm $\mathsf{Dec}_{\mathsf{GL}}$.

The next procedure will compute $A^i \cdot \vec{1} \mapsto i$ correctly on all inputs i, using the random self-reducibility of discrete log. This procedure is a space-efficient and randomness-efficient adaptation of [CLO+23, Lemma 4.6].

For any fixed $j \in [q^v - 1]$, consider the following oracle circuit $C_{3,j}$ (looking ahead, for some choices of j, the $C_{3,j}$'s will be sub-circuits in C_3). Given input $A^i \cdot \vec{1}$ and oracle access to $C_2^{C_1}$, the circuit $C_{3,j}$ computes $\vec{v} = A^j \cdot (A^i \cdot \vec{1})$, and then computes $b = C_2^{C_1}(\vec{v})$. It checks whether $A^b \cdot \vec{1} = \vec{v}$, and rejects otherwise. Now it knows that $A^b \cdot \vec{1} = A^j \cdot A^i \cdot \vec{1}$, and hence $A^{-j} \cdot A^b \cdot \vec{1} = A^i \cdot \vec{1}$, and the circuit outputs $\begin{cases} b - j & b > j \\ q^v - (b-j) & o.w. \end{cases}$. (Recall that A is a generator matrix, and hence A is invertible.) Note that $C_{3,j}$ is of size $\mathrm{polylog}(n)$ and can be printed in space $O(\log n)$. (We rely on Proposition 5.7 to hard-wire a description of A into $C_{3,j}$.)

We claim that, given a fixed j and input $A^i \cdot \vec{1}$, the output of $C_{3,j}(A^i \cdot \vec{1})$ can be computed in space $O(\log n)$ with oracle access to P_v. Computing $\vec{v}$ can be done in space $O(\log n)$ by Proposition 5.7, and computing $b \in \{0,1\}^{v \cdot \log(q)}$ can be done in such space with access to P_v by the discussion following Theorem 5.10. Computing $A^b \cdot \vec{1}$ is done via Proposition 5.7 again, and the final step amounts to checking if $b > j$ and printing an integer in $[q^v]$ (i.e., either $b - j$ or $q^v - (b - j)$).

To construct R_3 and C_3 we will need the following space-efficient sampler:

Theorem 5.11 (see, e.g. [DPT24, Theorem 3.12])**.** *For every $\varepsilon, \delta \colon \mathbb{N} \to [0,1]$ computable in space $O(\log(1/\varepsilon\delta))$, there is an algorithm* Samp *that for every $n \in \mathbb{N}$ computes a strong (ε, δ)-sampler with sample size* $\mathrm{poly}(\log(1/\delta), \varepsilon)$ *and randomness $\bar{n} = n + O(\log(1/\varepsilon\delta))$, using space $O(\bar{n})$.*

The machine R_3 uses Theorem 5.11 with output length $v \cdot \log(q)$, accuracy $\varepsilon_2/2 = 1/\mathrm{poly}(m, \log(n))$, and confidence $1/n^2$, to draw a random sample. Note that the number of random coins for Samp is $O(v \cdot \log(q) + \log(n)) = O(\log n)$, and that its sample size is $\mathrm{poly}(m, \log(n))$. Then, R_3 prints an oracle circuit C_3 that gets input $A^i \cdot \vec{1}$, computes $C_{3,j}^{C_2^{C_1}}(A^i \cdot \vec{1})$ for every output $j \in \{0,1\}^{v \cdot \log(q)}$ in the sample of Samp, and if one of the $C_{3,j}$'s printed i then C_3 prints that i.[32] By a union-bound, with probability at least $1 - 1/n$, it holds that C_3 computes $A^i \cdot \vec{1} \mapsto i$ for all $i \in \{0,1\}^{v \cdot \log(q)}$. Note that R_3 is computable in space $O(\log n)$, and that C_3 is of size $\mathrm{poly}(m, \log(n))$.

[32]Observe that for every j, the circuit $C_{3,j}$ never errs (i.e., it either aborts or outputs the correct answer), so if several $C_{3,j}$'s print answers, the answers are identical.

We explain how to modify R_3 so that it does not use random coins. The modified algorithm enumerates over choices of $O(\log n)$ coins for Samp, and for each fixed choice r, it tests whether the resulting circuit C_3 indeed computes $A^i \cdot \vec{1} \mapsto i$ for every $i \in \{0,1\}^{v \cdot \log(q)}$. Since we can enumerate over i's in space $O(\log n)$, it suffices to verify that $C_3(A^i \cdot \vec{1})$ can be computed in space $O(\log n)$.

This follows from the fact that Samp is computable in such space, that the sample size is $\mathrm{poly}(m, \log(n)) \le \mathrm{poly}(n)$ (so that we can enumerate over the sample), that for each fixed j in the sample, the sub-circuit $C_{3,j}$ is computable in such space (by the discussion before Theorem 5.11), and that we can compute $A^{i'} \cdot \vec{1}$ in space $O(\log n)$, where i' is the output of $C_{3,j}$ (by Proposition 5.7), and check whether it equals our input.

The final algorithm R_{dl} combines R_1, R_2, R_3 in a straightforward way to output an oracle circuit C_{dl} that implements $C_3^{C_2^{C_1}}$ (the oracle queries that C_{dl} makes are intended to be answered by P_v, since C_1 requires oracle access to P_v). The space complexity of R_{dl} is $O(\log n)$, and it outputs C_{dl} of size $\mathrm{poly}(m, \log(n)) = \mathrm{poly}(m)$ that, when given oracle access to P_v, correctly computes $A^i \cdot \vec{1} \mapsto i$ for all $i \in [q^v - 1]$. (This paragraph replaces the last paragraph of the original proof in [DPTW25]).

□

5.4 The Reconstruction Procedure for $L_0^{(q)}, \ldots, L_{v-1}^{(q)}$

In this section we will construct a procedure that gets oracle access to predictors for the q-ary lists $L_0^{(1)}, \ldots, L_{v-1}^{(q)}$ (in a sense that will be defined in Definition 5.16) and computes the function $y \mapsto \hat{x}(A^y \cdot \vec{1})$. The main result statement appears in Proposition 5.21.

Notation. For $i \in \{0, \ldots, q_0 - 1\}$, let w_i be the $(i+1)^{\text{th}}$ element in $\mathbb{F}_{q_0}$ in lexicographical order. For consistency, throughout the section we will use the following notation:

- i ranges in $\{0, \ldots, v\}$.
- j ranges in $[m-1]$.
- k ranges in $[q^v - 1]$.
- t is an element in $\mathbb{F}_q$, or an index in a set $[r]$.

For brevity, we will also use the following notation for elements of $\mathbb{F}_q$. Recall that $\mathbb{F}_q \equiv \mathbb{F}_{q_0}^{\Delta}$, and we will frequently parse each $t \in \mathbb{F}_q$ as a sequence consisting of one element $w_i \in \mathbb{F}_{q_0}$ and $\Delta - 1$ elements $u \in \mathbb{F}_{q_0}^{\Delta-1}$. Thus, we will frequently denote this by $t = (w_i, u) \in \mathbb{F}_q$.

5.4.1 Pseudorandom primitives

We now present a few pseudorandomness primitives that we will need for the reconstruction of the q-ary lists $L_0^{(q)}, \ldots, L_{v-1}^{(q)}$. Specifically, we will need the randomness-efficient curve sampler by Guo [Guo13], and a space-efficient averaging sampler. We first define curve samplers, state Guo's result, and then prove that Guo's sampler is computable in logspace. Then, we state the averaging sampler that our proof will use.

Throughout this section and the next, we will frequently refer to distributions over subsets S' of a set S as (ε, δ)-samplers. (Recall that the standard terminology refers to samplers as functions whose output is a subset $S' \subseteq S$, i.e., the set of sampled points.) By this terminology, we mean that for every $T \subseteq S$, with probability $1 - \delta$ over the choice of S' we have $\Pr_{s \in S'}[s \in T] \in |T|/|S| \pm \varepsilon$.

We will also frequently identify curves $C\colon \mathbb{F}_q \to \mathbb{F}_q$ with their image $C = \{C(t)\}_{t\in\mathbb{F}_q}$, and it will be clear from context which of the two interpretation of "a curve" we are referring to (i.e., a function or its image). For a matrix $A \in \mathbb{F}_q^{v\times v}$, the notation $A \cdot C$ means $\{A \cdot C(t)\}_{t\in\mathbb{F}_q}$.

Definition 5.12 (curve sampler)**.** Let $\mathsf{Samp}\colon \{0,1\}^{\bar{n}} \times \mathbb{F}_q \to \mathbb{F}_q^v$ be an (ε,δ)-sampler. We say that Samp is a degree-t curve sampler if for every fixed $z \in \{0,1\}^{\bar{n}}$, the function $\mathsf{Samp}_z(i) = \mathsf{Samp}(z,i)$ is a curve $\mathbb{F}_q \to \mathbb{F}_q^v$ of degree at most t.

Theorem 5.13 (Guo's curve sampler [Guo13])**.** *Let $\varepsilon,\delta\colon \mathbb{N} \to [0,1]$ and $q\colon \mathbb{N} \to \mathbb{N}$ be space-computable such that $q(v)$ is a prime power satisfying $q(v) \geq (v\cdot\log(1/\delta(v))/\varepsilon(v))^{\Theta(1)}$. Then, there is an algorithm that for every $v \in \mathbb{N}$ computes a strong degree-t curve (ε,δ)-sampler*

$$\mathsf{Samp}\colon \{0,1\}^{\bar{n}} \times \mathbb{F}_q \to \mathbb{F}_q^v$$

with $t = (m\cdot\log_q(1/\delta))^{O(1)}$ and $\bar{n} = O(v\cdot\log(q)+\log(1/\delta))$, using space $O(v\cdot\log(q)+\mathrm{loglog}(1/\delta)))$.

Proof. We prove that Guo's construction is computable in space $O(v \cdot \log(q) + O(\mathrm{loglog}(1/\delta)))$, and that it yields a strong sampler. Let us first bound the space-complexity. The construction is the composition of an outer sampler and an inner one, and we first analyze them separately and then analyze the composition.

Claim 5.13.1. The outer sampler in Guo's construction $\mathsf{Out}\colon \mathbb{F}_q^{O(v+\log_q(1/\delta))} \times \mathbb{F}_q^{\log(v)+1} \to \mathbb{F}_q^v$ is computable in space $O(v \cdot \log(q) + \mathrm{loglog}(1/\delta))$.

Proof. The outer sampler first transforms its source into a block source, using the condenser of [GUV09]. Given $(x, y) \in \mathbb{F}_q^{\bar{n}_{\mathsf{out}}} \times \mathbb{F}_q$ and a parameter v_i, where $\bar{n}_{\mathsf{out}} = O(v + \log_q(1/\delta))$, the condenser outputs

$$\mathsf{Cond}_{v_i}(x,y) = \big(y, f_x(y), f_x(\zeta\cdot y), ..., f_x(\zeta^{v_i-2}\cdot y)\big) \in \mathbb{F}_q^{v_i} \tag{5.3}$$

where $\zeta \in \mathbb{F}_q$ is a primitive element and $f_x(z) = \sum_{i=0}^{\bar{n}_{\mathsf{out}}-1} x_i \cdot z^i$. We can find a primitive element $\zeta \in \mathbb{F}_q$ in space $O(\log q)$ (by brute-force), raise it to the power $\leq v_i$ in space $O(\log(v_i) + \log(q))$, and compute $(x, z) \mapsto f_x(z)$ in space $O(\log(\bar{n}_{\mathsf{out}}) + \log(q))$. The transformation of the source into a block source is

$$\mathsf{Blk}(x, y_1, ..., y_s) = (\mathsf{Cond}_{v_1}(x, y_1), ..., \mathsf{Cond}_{v_s}(x, y_s)) \in \mathbb{F}_q^{4(v-1)},$$

where $s = \log(v)$, $v_i = 4v \cdot 2^{-i}$, $\sum_{i\in[s]} v_i = 4(v-1)$, and Blk is computable in space $O(\log(v) + \log(q) + \mathrm{loglog}(1/\delta))$ because Cond is computable in that space.

Now, given a block source $((a_1, b_1), ..., (a_s, b_s))$ where $(a_i, b_i) \in \mathbb{F}_q^{v_i/2} \times \mathbb{F}_q^{v_i/2}$ for all i, and seed $y'_s \in \mathbb{F}_q$, the outer extractor works as follows. For each $i = s, ..., 1$, it prints the first $v_i/2 - 1$ elements of $a_i \cdot y'_i + b_i$ and defines y_{i-1} to be the last element of $a_i \cdot y'_i + b_i$. This is computable in space $O(v \cdot \log(q))$ since the linear function in each block (i.e., $a_i \cdot y'_i + b_i$) is computable in such space, and the algorithm only needs to store a single element in $\mathbb{F}_q$ when moving from one block to the next. By composing this algorithm with Blk, we get an (ε,δ)-sampler

$$\mathsf{Out}\colon \mathbb{F}_q^{O(v+\log_q(1/\delta))} \times \mathbb{F}_q^{\log(v)+1} \to \mathbb{F}_q^v$$

computable in space $O(v \cdot \log(q) + \mathrm{loglog}(1/\delta))$, where the output length is truncated to v (i.e., we rely on the fact that $\sum_{i\in[s]} v_i/2 - 1 = 2v - 2 - \log(v) > v$). □

Claim 5.13.2. The inner sampler $\mathsf{In}\colon \mathbb{F}_q^{\mathrm{polylog}(v)+O(\log_q(1/\delta))} \times \mathbb{F}_q \to \mathbb{F}_q^{\log(v)+1}$ is computable in space $O(\log(q) \cdot \mathrm{polylog}(v))$.

Proof. Let $\ell = \log(v) + 1$. The sampler In is defined recursively, with $s' = \log(\ell) = O(\mathrm{loglog}(v))$ levels of recursion. For level i, we fix parameters $d_i = \ell/2^i$ and $t_i = \begin{cases} 16^i/4 & i < s' \\ 16^i/4 + 5 \cdot \log_q(1/\delta) & i = s' \end{cases}$, and define an algorithm

$$\mathsf{In}_i \colon \mathbb{F}_q^{4t_i \cdot d_i} \times \mathbb{F}_q^{d_i} \to \mathbb{F}_q^{\ell}.$$

At the first level In_0 just outputs its seed. At level $i \geq 1$, the algorithm In_i gets input $(x_{i,1}, x_{i,2}) \in \mathbb{F}_q^{3t_i \cdot d_i} \times \mathbb{F}_q^{t_i \cdot d_i}$ and a seed $s_i \in \mathbb{F}_q^{d_i}$, computes $(z_{i,1}, z_{i,2}, z_{i,3}) = \mathsf{Curve}_i(x_{i,1}, s_i) \in \mathbb{F}_q^{3d_i}$, and outputs $\mathsf{In}_{i-1}\left(\mathsf{Cond}_i(x_{i,2}, z_{i,1}), (z_{i,2}, z_{i,3})\right)$, where the algorithms are defined as follows.

- The algorithm $\mathsf{Curve}_i(x_{i,1}, s_i)\colon \mathbb{F}_q^{3t_i \cdot d_i} \times \mathbb{F}_q^{d_i} \to \mathbb{F}_q^{3d_i}$ parses its input $x_{i,1}$ as t_i triplets of elements $(c_{0,1}, c_{0,2}, c_{0,3}), \ldots, (c_{t_i-1,1}, c_{t_i-1,2}, c_{t_i-1,3}) \in \mathbb{F}_{q^{d_i}}^3$, parses its seed s_i as an element $y \in \mathbb{F}_{q^{d_i}}$, and computes $\left(\sum_{j=0}^{t_i-1} c_{j,1} \cdot y^j, \sum_{j=0}^{t_i-1} c_{j,2} \cdot y^j, \sum_{j=0}^{t_i-1} c_{j,3} \cdot y^j\right)$. It then parses the latter triplet as $3d_i$ elements in $\mathbb{F}_q$ and outputs these elements.

  Note that Curve_i is computable in space $O(d_i \cdot \log(q) + \log(t_i)) \leq O(\log(v) \cdot \log(q) + \mathrm{polylog}(v))$, and that its output is of length $O(d_i \cdot \log(q)) \leq O(\log(v) \cdot \log(q)))$.

- The algorithm $\mathsf{Cond}_i\colon \mathbb{F}_q^{d_i \cdot t_i} \times \mathbb{F}_q^{d_i} \to \mathbb{F}_q^{2d_i \cdot t_{i-1}}$ parses its input $x_{1,2}$ as t_i elements in $\mathbb{F}_{q^{d_i}}$ and its seed as an element $y \in \mathbb{F}_{q^{d_i}}$, and outputs $2d_i \cdot t_{i-1}$ elements defined as in Eq. (5.3).

  This algorithm works over the field of size $q' = q^{d_i}$, and is computable in space $O(\log(q') + \log(d_i \cdot t_{i-1})) \leq O(\log(v) \cdot \log(q))$. Its output length is at most $t_{i-1} \cdot d_i \cdot \log(q) = \mathrm{polylog}(v)$.

Thus, the computation of $\mathsf{In} = \mathsf{In}_{s'}$ amount to computing two strings of total length at most $\log(q) \cdot \mathrm{polylog}(v)$, and then passing them on to level $s' - 1$ (as the input and seed to $\mathsf{In}_{s'-1}$); indeed, at each level, the algorithm maps its input to two strings, and gives these strings as input to the level below. Since the strings at each level are of length at most $\log(q) \cdot \mathrm{polylog}(v)$, and the computation at each level can be done in space $O(\log(v) \cdot \log(q) + \mathrm{polylog}(v))$, the inner sampler is computable in space $O(\log(q) \cdot \mathrm{polylog}(v))$. ☐

The final sampler uses the inner sampler $\mathsf{In}\colon \mathbb{F}_q^{\bar{n}_{\mathsf{in}}} \times \mathbb{F}_q \to \mathbb{F}_q^{\log(v)+1}$ to sample from the outputs of Out, where $\bar{n}_{\mathsf{in}} = \mathrm{polylog}(v) + O(\log_q(1/\delta))$; that is, given $(x, x') \in \mathbb{F}_q^{\bar{n}_{\mathsf{out}}} \times \mathbb{F}_q^{\bar{n}_{\mathsf{in}}}$ and $y \in \mathbb{F}_q$,

$$\mathsf{Samp}((x, x'), y) = \mathsf{Out}(x, \mathsf{In}(x', y)).$$

The bound on the space complexity follows by combining Claims 5.13.1 and 5.13.2.

Strongness. Having proved that Guo's curve sampler is computable in small space, let us now prove that the construction yields a strong sampler. We only give here a proof sketch, and the full proof involves fully articulating standard techniques from extractor theory.

In [Guo13], the outer and inner samplers are in fact analyzed using the terminology of randomness extractors, and indeed, strong samplers are equivalent to strong extractors [Zuc97]. Following [RSW06, Theorem 8.2], we know that when we compose an outer and an inner extractor, for the final extractor to be strong, it suffices for the outer one to be strong (with a very minor loss in parameters, that essentially "sacrifices" the entropy in the seed).

The outer extractor Out employs the block-source extraction framework, after a block-source conversion step. One can verify that if the block-source conversion step is strong (namely, that Blk is close to a block-source even conditioned on a typical fixing of $y_1, \dots, y_s$)), and the extractor used in the block-source extraction procedure is strong, then the entire process yields a strong extractor. To argue that the block-source conversion step is strong, one can use the fact that Cond is a strong condenser (the latter fact is immediate, since it outputs the seed y). For the block-source extraction step, Out uses the "line extractor" that maps $((a,b),y)$ to $(a_1 y + b_1, \dots, a_{v_i} y + b_{v_i})$. The fact that it is strong readily follows from its analysis as a sampler (see, e.g., [Guo13, Lemma 2.3]). $\square$

Having established Theorem 5.13, the following corollary is immediate.

Corollary 5.14. *For any $\varepsilon = \mathrm{poly}(\rho)$ and $\delta = q^{-O(v)}$, there is a probabilistic algorithm that generates a curve $C\colon \mathbb{F}_q \to \mathbb{F}_q^v$, using $O(\log n)$ random coins and in space $O(\log n)$, such that $d^{\mathsf{crv}} \triangleq \deg(C) = (m \cdot \log(n))^{O(1)}$ and the resulting distribution over curves is a strong (ε, δ)-sampler.*

An averaging sampler. Recall that $\mathbb{F}_q \equiv \mathbb{F}_{q_0}^{\Delta}$. We will also need to sample a set of points $R \subseteq \mathbb{F}_{q_0}^{\Delta-1}$ space-efficiently and randomness-efficiently, which we do using the sampler from Theorem 5.11.

Corollary 5.15. *For any $\delta = q^{-O(v)}$, there is a probabilistic algorithm that generates a set $R \subseteq \mathbb{F}_{q_0}^{\Delta-1}$ of size $r = \mathrm{poly}(v, \log(q))$, using $O(\log n)$ random coins and in space $O(\log n)$, such that the distribution over R's is an (ε, δ)-sampler, for $\varepsilon = \frac{1}{(m \cdot \log(n))^2}$.*

Proof. We use Theorem 5.11 with output length $\Delta \cdot \log(q_0) < \log(q)$ and $\varepsilon = 1/(m \cdot \log(n))^2$ and $\delta = q^{-O(v)}$. The sample size is $\mathrm{poly}(1/\varepsilon, \log(1/\delta)) = \mathrm{poly}(m, \log(n))$, the required randomness is $\log(q) + O(\log(1/\varepsilon\delta)) = O(\log n)$, and the space complexity is linear in the randomness. $\square$

5.4.2 Learning a single curve

We first show an algorithm analogous to "Learn Next Curve" in [SU05]. Intuitively, the algorithm uses a predictor and a sequence of "known" points on previous curves to predict points on the next curve, and then uses a small number of "known" points on the next curve in order to error-correct its predictions. The crucial part for us is the efficiency of this algorithm (i.e., it is a space-bounded machine that outputs a small circuit), and distilling the exact properties that this algorithm needs from the distribution over the relevant curves in order to work.

Definition 5.16 (good predictors)**.** We say that $P^{(i)}\colon \mathbb{F}_q^{m-1} \to \mathbb{F}_q^{\bar{\rho}}$ is a ρ-good predictor for $L_i^{(q)}$ if $\Pr_{\vec{z} \in L_i^{(q)}}[P^{(i)}(\vec{z}_{1,\dots,m-1}) \ni \vec{z}_m] > \rho$.

Recall that $\bar{\rho}$ was defined as $\bar{\rho} = (1/\rho)^{c_{\mathsf{STV}}}$ in Theorem 5.8, and indeed we use the same parameter when considering predictors in Definition 5.16.

For simplicity of presentation, we will assume throughout this section that all predictors predict the m^{th} element; that is, for each $i \in \{0, \dots, v-1\}$, the predictor $P^{(i)}\colon \mathbb{F}_q^{j_i} \to \mathbb{F}_q^{\bar{\rho}}$ for $L_i^{(q)}$ has $j_i = m-1$. This assumption does not meaningfully affect the argument (in fact, it is a "worst-case" scenario for the reconstruction) and we make it only to reduce notational clutter.

Lemma 5.17 (derandomized learning of a single curve)**.** *There is a machine* LrnNext *that gets input 1^n and $i \in \{0, \dots, v-1\}$ and $R \subseteq \mathbb{F}_{q_0}^{\Delta-1}$ of size $r = |R|$, runs in space $O(\log n)$, and prints a circuit $C_{\mathsf{LrnNext},i}$ of size $\mathrm{poly}(q, 1/\rho)$ satisfying the following.*

1. ***Input.*** *Points* $\left\{(a_t^{(m-1)}, ..., a_t^{(1)}) \in \mathbb{F}_q^{m-1}\right\}_{t\in\mathbb{F}_q}$, *evaluations* $\{b_t \in \mathbb{F}_q\}_{t\in[r]}$, *and* $i^* \in \{0, ..., v\}$.

2. ***Output.*** *A set* $\{o_t \in \mathbb{F}_q\}_{t\in\mathbb{F}_q}$.

3. ***Functionality.*** *Consider a distribution over curves* $C\colon \mathbb{F}_q \to \mathbb{F}_q^v$ *of degree* $D = d^{\mathsf{crv}} \cdot q_0 \cdot r$ *and an independent distribution over sets* $R \subseteq \mathbb{F}_{q_0}^{\Delta-1}$ *of size* r *such that the distribution over* C *is a* $(\rho/4, q^{-20v})$*-sampler and the distribution over* R *is a* $(1/2, q^{-20v})$*-sampler. Then, for every* $(\rho/2)$*-good predictor* $P^{(i)}$*, with probability at least* $1 - q^{-10v}$ *over* C, R *it holds that:*

   When $a_t^{(j)} = \hat{x}(A^{-j\cdot q^i} \cdot C(t))$ *for all* $t \in \mathbb{F}_q$ *and* $j \in [m-1]$*, and* $b_t = \hat{x}(C(w_{i^*}, R_t))$ *for all* $t \in [r]$*, and* $C_{\mathsf{LrnNext},i}$ *is given as oracle* $P^{(i)}$*, its output satisfies* $o_t = \hat{x}(C(t))$ *for all* $t \in \mathbb{F}_q$.

Moreover, there is another machine that gets input 1^n *and* i *and* R *(same as* $\mathsf{LrnNext}$*) and in addition a curve* C *and oracle access to* $\hat{x}$ *and to* $P^{(i)}$*, runs in space* $O(\log(n) + \text{polylog}(m))$*, and decides whether or not the circuit* $C_{\mathsf{LrnNext},i}$ *that* $\mathsf{LrnNext}$ *outputs satisfies the requirement with respect to* C *and* $P^{(i)}$ *(i.e., whether when* $a_t^{(j)}$ *satisfy the hypothesis above with respect to* C *and* R*, the output of* $C_{\mathsf{LrnNext},i}$ *with oracle access to* $P^{(i)}$ *is* $\{\hat{x}(C(t))\}_{t\in\mathbb{F}_q}$*).*

Proof. Let us first describe $C_{\mathsf{LrnNext},i}$, and then prove the required properties. The circuit will use Sudan's [Sud97] list-decoding algorithm for the Reed-Solomon code:

Theorem 5.18 (list-decoding of the RS code [Sud97])**.** *Given* p *distinct pairs* $\{(x_a, y_a) \in \mathbb{F}_q \times \mathbb{F}_q\}_{a\in[p]}$*, there are at most* $2/\mu$ *degree-*d' *polynomials* g *such that* $g(x_a) = y_a$ *for at least a* μ*-fraction of the pairs, as long as* $\mu > \sqrt{2d'/p}$*. Furthermore, a list of all such polynomials can be computed in time* $\text{poly}(p, \log(q))$.

For each $t \in \mathbb{F}_q$, the circuit queries its oracle $P_j\colon \mathbb{F}_q^{m-1} \to \mathbb{F}_q^{\bar{\rho}}$ on the point $(a_t^{(m-1)}, ..., a_t^{(1)}) \in \mathbb{F}_q^{m-1}$, which yields a set $S_t \subseteq \mathbb{F}_q$ of size $|S_t| = \bar{\rho}$. It then uses Theorem 5.18 with the set $S = \cup_{t\in\mathbb{F}_q} S_t$ and with parameter values $p = \bar{\rho} \cdot q$ and $\mu = (\rho/4) \cdot \bar{\rho}$ and $d' = d \cdot D$ to obtain a list of $8/(\rho \cdot \bar{\rho})$ polynomials in time $\text{poly}(q, \rho^{-1})$. (Note that $\mu > \sqrt{2d'/p}$, since $q^{1-1/\Delta} > 32d \cdot d^{\mathsf{crv}} \cdot r \cdot \text{poly}(1/\rho)$, relying on a sufficiently large choice of constant $c > 1$ in Section 5.1.1.) If the list contains a unique polynomial $p\colon \mathbb{F}_q \to \mathbb{F}_q$ such that $p(w_{i^*}, R_t) = b_t$ for all $t \in [r]$, output $\{p(t)\}_{t\in\mathbb{F}_q}$; otherwise, fail.

Observe that there is a uniform machine that gets a first set of inputs $1^n, i, R$ and a second set of inputs $\left\{(a_t^{(m-1)}, ..., a_t^{(1)})\right\}, \{b_t\}, i^*$, and computes the value of $C_{\mathsf{LrnNext},i}$ (i.e., of the circuit corresponding to the first set of inputs) on the second set of inputs in time $t_{\mathsf{LrnNext}} = \text{poly}(q, \rho^{-1})$. Hence, this functionality is also computable by a $O(\log(t_{t_{\mathsf{LrnNext}}}))$-space-uniform circuit of size $\text{poly}(t_{\mathsf{LrnNext}})$ (i.e., by a standard simulation of machines by highly uniform circuits of quadratic size). The machine $\mathsf{LrnNext}$ gets input $1^n, i, R$, and prints the latter circuit in space $O(\log(t_{t_{\mathsf{LrnNext}}})) \leq O(\log n)$ while hard-wiring the values of i and of R to the appropriate input gates.

Analysis. We want to prove the claim about the functionality of $C_{\mathsf{LrnNext},i}$. We first argue that:

Claim 5.18.1. With probability $1 - q^{-20v}$ over the choice of C, it holds that

$$\Pr_{t\in\mathbb{F}_q}[\hat{x}(C(t)) \in S_t] \geq \rho/4.$$

Proof. Consider a uniformly chosen $\vec{z} \in L_i^{(q)}$. By the guarantee on $P^{(i)}$ we know that

$$\Pr[P^{(i)}(\vec{z}_{1,...,m-1}) \ni \vec{z}_m] \geq \rho/2.$$

However, we also know that $\vec{z}$ is distributed identically to

$$\hat{x}(A^{-(m-1)\cdot q^i}(\vec{y})) \circ \ldots \circ \hat{x}(A^{-q^i} \cdot \vec{y}) \circ \hat{x}(\vec{y})$$

for a uniformly chosen $\vec{y} \in \mathbb{F}_q^v$. (This is because A^{q^i} is invertible, and relying on the definition of $L_i^{(q)}(\vec{s})$ in Eq. (5.1) and on the fact that $\vec{z}$ is obtained via a uniform choice of $\vec{s} \in \mathbb{F}_q^v$.)

Hence, when choosing a uniformly random $\vec{y} \in \mathbb{F}_q^v$, with probability at least $\rho/2$ we have that

$$P^{(i)}(\hat{x}(A^{-(m-1)\cdot q^i}(\vec{y})), \ldots, \hat{x}(A^{-q^i} \cdot \vec{y})) \ni \hat{x}(\vec{y}). \tag{5.4}$$

Since the distribution over C is an $(\varepsilon = \mathrm{poly}(\rho), \delta = q^{-20v})$-sampler, with probability at least $1 - \delta$, the fraction of points $\vec{y} = C(t)$ on the curve such that Eq. (5.4) holds is at least $\rho/2 - \varepsilon = \rho/4$. □

By Claim 5.18.1, with probability $1 - q^{-20v}$ over the choice of C there are at least $\mu = (\rho/4) \cdot \bar{\rho}$ pairs (t, u) in $S = \cup_{t \in \mathbb{F}} \{t\} \times S_t$ such that $u = p_C(t)$, where $p_C(t) = \hat{x}(C(t))$. Also note that p_C is of degree $d' = d \cdot D$. Hence, for every C satisfying the above, the list of polynomials that the algorithm from Theorem 5.18 outputs contains p_C.

Now, condition on such a C, and consider any $p \neq p_C$ of degree $\deg(p) = d \cdot D$. Note that $d \cdot D / q_0^{\Delta-1} < 1/2$ (relying on a sufficiently large choice of $c > 1$ in the definition of q). Hence, by the Schwartz-Zippel lemma, the fraction of roots of $p - p_C$ in any set $S \subseteq \mathbb{F}_q$ of size $q_0^{\Delta-1}$ is less than 1/2. In particular,

$$\Pr_{u \in \mathbb{F}_{q_0}^{\Delta-1}} [p(w_{i^*}, u) \neq p_C(w_{i^*}, u)] > 1/2.$$

Since the distribution over $R \subseteq \mathbb{F}_{q_0}^{\Delta-1}$ is a $(1/2, q^{-20v})$-sampler, with probability $1 - q^{-20v}$ there is $t \in [r]$ such that $p(w_i, R_t) \neq p_C(w_i, R_t)$. By a union-bound over all p's that the algorithm of Theorem 5.18 outputs, with probability at least $1 - q^{-20v} \cdot \mathrm{poly}(1/\rho) \geq 1 - q^{-10v}$ over R the circuit $C_{\mathsf{LrnNext},i}$ outputs the unique p_C.[33]

[33]The bound $\mathrm{poly}(1/\rho) \leq q^{10v}$ assumes that $m \leq n^{\zeta}$ for a universal constant $\zeta > 0$. We can indeed assume this without loss of generality, otherwise Theorem 5.1 is trivial.

Let us now describe the machine for the "moreover" part. It first checks whether or not the assertion in Claim 5.18.1 holds, using its oracle access to $P^{(i)}$ and to $\hat{x}$ and the given curve C. That is, for each $t \in \mathbb{F}_q$, it computes $C(t)$, queries $\hat{x}(C(t))$, queries $P^{(i)}(t)$ to obtain S_t, and checks if $\hat{x}(C(t)) \in S_t$. When $\Pr_{t\in\mathbb{F}_q}[\hat{x}(C(t)) \in S_t] < \rho/4$, the machine rejects.

At this point, by Theorem 5.18, we know that the list-decoding algorithm used by $C_{\mathsf{LrnNext},i}$ outputs a list that contains the correct polynomial p_C. We just need to rule out the existence of another p in this list that agrees with p_C on the point-set $\{(w_{i^*}, R_t) \in \mathbb{F}_q\}_{t\in[r]}$. Since R is given to the machine as input, and we can compute $p_C(t) = \hat{x}(C(t))$ at any point $t \in \mathbb{F}_q$ (using our input C and the oracle access to $\hat{x}$), it suffices to show that we can evaluate each polynomial that the list-decoding algorithm outputs, at any given $t \in \mathbb{F}_q$.

This calls for a space-efficient implementation of the algorithm in Theorem 5.18. As proved in [CT21b], this algorithm is implementable in logspace-uniform randomized $\mathcal{NC}$ when the field is prime and the number of points p is relatively small. We need a deterministic algorithm, and to get rid of the assumptions about primality and p. We prove that:

Theorem 5.19. *Let $q, p, d\colon \mathbb{N} \to \mathbb{N}$ and $\mu\colon \mathbb{N} \to [0,1]$ be computable in logspace. There is a logspace-uniform circuit family that gets as input 1^n and a representation of a finite field $\mathbb{F}_q$ and p distinct pairs $\left\{(a_i, b_i) \in \mathbb{F}_q^2\right\}_{i\in[p]}$ such that $\mu > 2\sqrt{d/p}$, and outputs a list of at most $2/\mu$ polynomials that contains every polynomial τ of degree at most d satisfying $\Pr_{i\in[m]}[\tau(a_i) = b_i] \geq \mu$. The circuit size is $\mathrm{poly}(|\mathbb{F}_q|)$ and its depth is $\mathrm{polylog}(|\mathbb{F}_q|)$.*

Since the proof of Theorem 5.19 uses known ideas, we defer it to Appendix A. Now, the list-decoding algorithm in its statement is implementable by a circuit of depth $\mathrm{polylog}(|\mathbb{F}_q|)$ constructible in space $O(\log|\mathbb{F}_q|)$, and hence the algorithm is also computable in space $\mathrm{polylog}(|\mathbb{F}_q|)$. In particular, in our setting this is space $\mathrm{polylog}(m)$.

□

5.4.3 Learning interleaved curves

The next algorithm will construct two interleaved curves, using Corollary 5.14 and Corollary 5.15, such that one can use the $C_{\mathsf{LrnNext},i}$'s from Lemma 5.17 repeatedly to learn $q^v - 1$ "shifts" of each of these curves by A. At each step, the previous learned curve will intersect the next curve we want to learn at sufficiently many locations for the needed error-correction in Lemma 5.17. Our construction of the interleaved curves is different than that in previous works (e.g., in [SU05, CLO+23]), since we need a randomness-efficient algorithm.

Lemma 5.20 (derandomized interleaved learning)**.** *There is a randomized algorithm that gets input 1^n, uses $O(\log n)$ random coins and $O(\log n)$ space, and outputs two curves $C_1, C_2\colon \mathbb{F}_q \to \mathbb{F}_q^v$ and a set $R \subseteq \mathbb{F}_{q_0}$ such that for every collection of $(\rho/2)$-good predictors $P^{(0)}, ..., P^{(v)}$, with probability $1 - q^{-5v}$ the following holds. For every $i \in \{0, ..., v-1\}$ and $k \in [q^v - 1]$, when we give* $\mathsf{LrnNext}$ *input i and R, it prints $C_{\mathsf{LrnNext},i}$ such that:*

1. **Learning $C^{\mathsf{Nxt},1}(t) \triangleq A^{k+q^i} \cdot C_1(t)$ from $C^{\mathsf{Prv},1}(t) \triangleq A^k \cdot C_2(t)$.** *When we give $C_{\mathsf{LrnNext},i}$ the points $\left\{a_t^{(j)} = \hat{x}(A^{-j\cdot q^i} \cdot C^{\mathsf{Nxt},1}(t))\right\}_{t\in\mathbb{F}_q, j\in[m-1]}$ and evaluations $\left\{b_t = \hat{x}(C^{\mathsf{Prv},1}(w_i, R_t))\right\}_{t\in[|R|]}$ and $i^* = i$ and oracle access to $P^{(i)}$, it outputs $\left\{\hat{x}(C^{\mathsf{Nxt},1}(t))\right\}_{t\in\mathbb{F}_q}$.*

2. **Learning $C^{\mathsf{Nxt},2}(t) \triangleq A^k \cdot C_2(t)$ from $C^{\mathsf{Prv},2}(t) \triangleq A^k \cdot C_1(t)$.** *When we give $C_{\mathsf{LrnNext},i}$ the points $\left\{a_t^{(j)} = \hat{x}(A^{-j\cdot q^i} \cdot C^{\mathsf{Nxt},2}(t))\right\}_{t\in\mathbb{F}_q, j\in[m-1]}$ and evaluations $\left\{b_t = \hat{x}(C^{\mathsf{Prv},2}(w_v, R_t))\right\}_{t\in[|R|]}$ and $i^* = v$ and oracle access to $P^{(i)}$, it outputs $\left\{\hat{x}(C^{\mathsf{Nxt},2}(t))\right\}_{t\in\mathbb{F}_q}$.*

Moreover, there is another machine that gets input 1^n and oracle access to $\hat{x}$ and to a collection of $(\rho/2)$-good predictors $P^{(0)}, ..., P^{(v)}$, runs in space $O(\log n) + \mathrm{polylog}(m)$, and outputs C_1, C_2 and R such that the guarantee above holds (i.e., for every i, k, when we give LrnNext *input i and R, it prints $C_{\mathsf{LrnNext},i}$ satisfying the two items).*

Proof. Let $C\colon \mathbb{F}_q \to \mathbb{F}_q^v$ be a degree-d^{crv} curve sampled by Corollary 5.14 with $\varepsilon = \rho/12$ and $\delta = q^{-30v}$, and let $R \subseteq \mathbb{F}_{q_0}$ be a set sampled by Corollary 5.15 with $\delta = q^{-30v}$. Denote $r = |R|$. We let $C_1 = C$, and define $C_2\colon \mathbb{F}_q \to \mathbb{F}_q^v$ as the unique curve of degree $(r \cdot q_0 - 1)$ satisfying the following:

$$\forall a \in \{0, ..., q_0 - 1\}, \forall u \in R, \quad C_2(w_a, u) = S_a \cdot C_1(w_a, u), \tag{5.5}$$
$$\text{where } S_a = \begin{cases} A^{q^a} & a \in \{0, ..., v-1\} \\ \mathsf{Id} & a \in \{v, ..., q_0 - 1\} \end{cases}$$

Observe that $\max\{\deg(C_1), \deg(C_2)\} < d^{\mathsf{crv}} \cdot q_0 \cdot r = D$, and that we can sample the curves and R and print them in space $O(\log n)$ and by using $O(\log n)$ coins.

Thus, we turn to the analysis. We first claim that the two curves have sufficient "sampling" properties and "intersection sampling" properties, as follows.

Claim 5.20.1 (each curve is a sampler, marginally)**.** For any fixed $k \in [q^v - 1]$, when choosing C from Corollary 5.14 with $\varepsilon = \rho/12$ and $\delta = q^{-30v}$ and R from Corollary 5.15 with $\delta = q^{-30v}$,

1. The distribution $A^k \cdot C_1$ is an (ε, δ)-sampler.

2. The distribution $A^k \cdot C_2$ is an (ε', δ')-sampler, where $\varepsilon' = 3\varepsilon = \rho/4$ and $\delta' = 2\delta \cdot q_0 < q^{-20v}$.

Proof. First observe that C_1 is an (ε, δ)-sampler, because $C_1 = C$. Next, for any shift A^k, the curve $A^k \cdot C_1$ is also an (ε, δ)-sampler; this is since A^k is invertible, and so the mapping of the image of C_1 to the image of $A^k \cdot C_1$ is a bijection.

We prove that C_2 is a sampler relying on the fact that C is a strong sampler and on the fact that R is a sampler. Specifically, fix any choice of $C = C_1$. We first claim that for every $T \subseteq \mathbb{F}_q^v$ and every fixed $a \in \{0, ..., q_0 - 1\}$, with probability at least $1 - \delta$ over R it holds that

$$\left| \Pr_{u\in R}[C_2(w_a, u) \in T] - \Pr_{u\in\mathbb{F}_{q_0}^{\Delta-1}}[S_a \cdot C_1(w_a, u) \in T] \right| \leq \varepsilon, \tag{5.6}$$

and

$$\left| \Pr_{u\in R}[C_2(w_a, u) \in T] - \Pr_{u\in\mathbb{F}_{q_0}^{\Delta-1}}[C_2(w_a, u) \in T] \right| \leq \varepsilon. \tag{5.7}$$

Indeed, the statements in Eqs. (5.6) and (5.7) are true because R is an (ε, δ)-sampler in $\mathbb{F}_{q_0}^{\Delta-1}$ (and considering the tests $T_{w_a}(u) = \mathbf{1}[S_a \cdot C_1(w_a, u) \in T]$ and $T'_{w_a}(u) = \mathbf{1}[C_2(w_a, u) \in T]$).

It follows that for every fixed C_1 and $T \subseteq \mathbb{F}_q^v$, with probability at least $1-\delta \cdot q_0$ over R we have

$$\begin{aligned}
&\Big|\Pr_{a\in\{0,\dots,q_0-1\},u\in\mathbb{F}_{q_0}^{\Delta-1}}[C_2(w_a,u)\in T] - \Pr_{a\in\{0,\dots,q_0-1\},u\in\mathbb{F}_{q_0}^{\Delta-1}}[S_a\cdot C_1(w_a,u)\in T]\Big| \\
&= \Big|\mathop{\mathbb{E}}_{a\in\{0,\dots,q-1\}}\Big[\Pr_{u\in\mathbb{F}_{q_0}^{\Delta-1}}[C_2(w_a,u)\in T] - \Pr_{u\in\mathbb{F}_{q_0}^{\Delta-1}}[S_a\cdot C_1(w_a,u)\in T]\Big]\Big| \\
&\le \mathop{\mathbb{E}}_{a\in\{0,\dots,q-1\}}\Big[\Big|\Pr_{u\in\mathbb{F}_{q_0}^{\Delta-1}}[C_2(w_a,u)\in T] - \Pr_{u\in\mathbb{F}_{q_0}^{\Delta-1}}[S_a\cdot C_1(w_a,u)\in T]\Big|\Big] \\
&\le 2\varepsilon. \qquad (5.8)
\end{aligned}$$

Now, consider the joint distribution (C_1,R,C_2), which is obtained by choosing $C=C_1$ and R independently, and defining $C_2=C_2(C_1,R)$ as in Eq. (5.5). Assume towards a contradiction that there is $T\subseteq\mathbb{F}_q^v$ such that with probability more than $2\delta\cdot q_0$ over (C_1,R,C_2) it holds that $\Pr_{t\in\mathbb{F}_q}[C_2(t)\in T]-|T|/q^v>3\varepsilon$.[34]

Now, by Eq. (5.8), for every fixed choice of C_1, with probability $1-\delta\cdot q_0$ over R we have that

$$\Pr_{a\in\{0,\dots,q_0-1\},u\in\mathbb{F}_{q_0}^{\Delta-1}}[S_a\cdot C_1(w_a,u)\in T]-|T|/q^v>\varepsilon. \qquad (5.9)$$

We call every choice of R satisfying Eq. (5.9) good for C_1.

Next, consider the following test $T'\subseteq\mathbb{F}_q\times\mathbb{F}_q^v$. Given $(t,\vec{z})$, we parse $t=(w_a,u)\in\mathbb{F}_{q_0}\times\mathbb{F}_{q_0}^{\Delta-1}$, and define $M((w_a,u),\vec{z})\triangleq S_a\cdot\vec{z}\in\mathbb{F}_q^v$; we include $(t,\vec{z})$ in T' iff $M(t,\vec{z})\in T$. Note that when $(t,\vec{z})$ is chosen uniformly, the distribution $M(t,\vec{z})$ is uniform in $\mathbb{F}_q^v$, and hence

$$\Pr_{t,\vec{z}\in\mathbb{F}_q\times\mathbb{F}_q^v}[(t,\vec{z})\in T']=|T|/q^v.$$

On the other hand, when $(t,\vec{z})$ is uniformly chosen from the set $\{(t,C_1(t))\}_{t\in\mathbb{F}_q}$, the distribution $M(t=(w_a,u),\vec{z})$ is the uniform distribution on $\{S_a\cdot C_1(w_a,u)\}_{a\in\{0,\dots,q_0-1\},u\in\mathbb{F}_{q_0}^{\Delta-1}}$, and hence

$$\Pr_{t\in\mathbb{F}_q}\left[(t,C_1(t))\in T'\right]=\Pr_{a\in\{0,\dots,q_0-1\},u\in\mathbb{F}_{q_0}^{\Delta-1}}\left[S_a\cdot C_1(w_a,u)\in T\right].$$

Plugging the two equations above into Eq. (5.9), whenever R is good for C_1, we have that

$$\Pr_{t\in\mathbb{F}_q}\left[(t,C_1(t))\in T'\right]-\Pr_{t,\vec{z}\in\mathbb{F}_q\times\mathbb{F}_q^v}[(t,\vec{z})\in T']>\varepsilon. \qquad (5.10)$$

It follows that

$$\begin{aligned}
\Pr_{C_1}[\text{Eq. (5.10) holds}] &\ge \Pr_{C_1,R}\left[\left(\Pr_{t\in\mathbb{F}_q}[C_2(t)\in T]-|T|/q^v>3\varepsilon\right)\wedge R\text{ is good for }C_1\right] \\
&\ge 1-2\delta\cdot q_0-\delta\cdot q_0,
\end{aligned}$$

contradicting the fact that $C=C_1$ is a strong (ε,δ)-sampler. □

[34]Note that if there is T' such that $\left|\Pr_{t\in\mathbb{F}_q}[C_2(t)\in T']-|T'|/q^v\right|>3\varepsilon$ with probability more than $2\delta\cdot q_0$, then there is T such that $\Pr_{t\in\mathbb{F}_q}[C_2(t)\in T']-|T'|/q^v>3\varepsilon$ with probability more than $\delta>0$ (i.e., by taking either $T=T'$ or T as the complement of T'). Thus, to rule out the former it suffices to rule out the latter.

Claim 5.20.2 (interleaved curves agree on the points specified by R)**.** For any fixed $i \in \{0,...,v-1\}$ and any choice of C and R we have that

$$R \subseteq \left\{ u \in \mathbb{F}_{q_0}^{\Delta-1} : A^{q^i} \cdot C_1(w_i,u) = C_2(w_i,u) \right\}.$$

Moreover, for any fixed $k \in [q^v - 1]$, the claim still holds if we simultaneously replace "C_1" by "$A^k \cdot C_1$" and "C_2" by "$A^k \cdot C_2$".

Proof. The basic claim follows from the definition of C_2, and the "moreover" part follows immediately from the basic claim. □

Now, for Item (1) of our lemma, fix i and k, and let $C^{\mathsf{Nxt},1}(t) = A^{k+q^i} \cdot C_1(t)$. By Lemma 5.17, with probability at least $1 - q^{-10v}$ over $C^{\mathsf{Nxt},1}$ and R, when we give $C_{\mathsf{LrnNext},i}$ the points

$$\left\{ a_t^{(j)} = \hat{x}(A^{-j \cdot q^i} \cdot C^{\mathsf{Nxt},1}(t)) \right\}_{t \in \mathbb{F}_q, j \in [m-1]}$$

and the evaluations $\left\{ b_t = \hat{x}(C^{\mathsf{Nxt},1}(w_i,R_t)) \right\}_{t \in [|R|]}$ and $i^* = i$ and oracle $P^{(i)}$, it outputs $\left\{ C^{\mathsf{Nxt},1}(t) \right\}_{t \in \mathbb{F}_q}$, as long as the distributions over $C^{\mathsf{Nxt},1}$ and R are appropriate samplers. By Claim 5.20.1, the distribution over $C^{\mathsf{Nxt},1}$ is indeed such a sampler. By Claim 5.20.2, the curves $C^{\mathsf{Nxt},1}$ and $C^{\mathsf{Prv},1}(t) = A^k \cdot C_2(t)$ agree on the point-set $\{(w_i,R_t)\}_{t \in [r]}$, and hence the functionality of $C_{\mathsf{LrnNext},i}$ remains identical if we replace the set of b_t's above by $\left\{ b_t = \hat{x}(C^{\mathsf{Prv},1}(w_i,R_t)) \right\}_{t \in [r]}$.

For Item (2), similarly, we fix i,k, and let $C^{\mathsf{Nxt},2}(t) = A^k \cdot C_2(t)$. We use Lemma 5.17 identically to the proof above, the only difference being that now we argue about $C_{\mathsf{LrnNext},i}$ when it is given $i^* = v$ (rather than $i^* = i$). The claim follows relying on the fact that $C^{\mathsf{Nxt},2}$ is an appropriate sampler (by Claim 5.20.1) and that $C^{\mathsf{Nxt},2}$ and $C^{\mathsf{Prv},2}(t) = A^k \cdot C_1(t)$ agree on the point-set $\{(w_v,u)\}_{u \in \mathbb{F}_{q_0}}$ (by the definition of C_1 and C_2 on $\{(w_v,\cdot)\}$, in Eq. (5.5)).

The two paragraphs above established that for every fixed i,k, the statements in Items (2) and (1) hold with probability at least $1 - q^{-10v}$. By a union-bound over $i \in \{0,...,v-1\}$ and $k \in [q^v - 1]$, with probability at least $1 - q^{-5v}$ the two statements hold for every i,k.

For the "moreover" part, our machine will enumerate over choices of R, C_1, C_2, and for each choice use the machine $M_{5.17}$ from the "moreover" part of Lemma 5.17 to test the conditions with respect to $R, C^{\mathsf{Nxt},1}, C^{\mathsf{Nxt},2}$ and all i,k.

In more detail, the machine enumerates over the seeds for Corollaries 5.14 and 5.15. For each fixed pair of seeds, it enumerates over all $i \in \{0,...,v-1\}$ and $k \in [q^v - 1]$, and invokes $M_{5.17}$ twice: once with virtual access to $C = C^{\mathsf{Nxt},1} = A^{k+q^i} \cdot C_1$, and afterwards with virtual access to $C = C^{\mathsf{Nxt},2} = A^k \cdot C_2$.[a] When the machine encounters a seed-pair such that $M_{5.17}$ accepts for all i,k, it outputs the corresponding C_1, C_2, R. Note that the machine runs in space $O(\log n) + \mathrm{polylog}(m)$.

By the analysis above, for most choices of seed pairs, the two requirements in the statement of Lemma 5.20 will be satisfied for all i,k. Syntactically, the first requirement is exactly the functionality guarantee of Lemma 5.17 with $C = C^{\mathsf{Nxt},1}$ and $i^* = i$, and the second requirement is the functionality guarantee with $C = C^{\mathsf{Nxt},2}$ and $i^* = v$.[b] Hence, our machine will output C_1, C_2, R such that the two requirements hold for all i,k.

[a] Computing bits of the description of C, in both cases, is done using the fixed seed and the algorithm from Corollary 5.14. Similarly, providing virtual access to R is done using the fixed seed and Corollary 5.15.

[b] When comparing syntactically, for the first requirement we relied on the fact that $C^{Prv,1}(w_i,R_t) = A^k \cdot C_2(w_i,R_t) = A^k \cdot S_i \cdot C_1(w_i,R_t) = A^{k+q^i} \cdot C_1(w_i,R_t) = C^{Nxt,1}(w_i,R_t)$ for all $i \in [v-1]$; and for the second requirement we relied on the fact that $C_2(w_v,u) = C_1(w_v,u)$ for all u (and hence $C^{\mathsf{Nxt},2}$ and $C^{\mathsf{Prv},2}$ agree on $\{(w_v,R_t)\}_{t \in [|R|]}$).

□

5.4.4 The main reconstruction algorithm

We are now ready to state and prove the main algorithm of the current section. This algorithm uses $O(\log n)$ coins and $O(\log n)$ space, makes queries to $\hat{x}$, and with high probability prints $\vec{v} \in \mathbb{F}_q^v$ and a circuit that computes the mapping $y \mapsto \hat{x}(A^y \cdot \vec{v})$.

Proposition 5.21. *There is an algorithm that gets input 1^n and oracle access to $\hat{x}$ and to a collection of $(\rho/2)$-good predictors $\vec{P} = P^{(0)}, ..., P^{(v-1)}$, uses space $O(\log n) + \mathrm{polylog}(m)$, and prints a circuit $R_0\colon \{0, ..., q^v - 1\} \to \mathbb{F}_q$ of size $\mathrm{poly}(m, \log(n))$ such that $R_0^{\vec{P}}(y) = \hat{x}(A^y \cdot \vec{1})$ for all y.*

Proof. We first describe the circuit R_0 and then explain how to construct it. The circuit has hard-wired choices of curves C_1, C_2 and a set $R \subseteq \mathbb{F}_{q_0}^{\Delta-1}$, as well as the circuits $C_{\mathsf{LrnNext},i}$ from Lemma 5.20 with all values of $i \in \{0, ..., v-1\}$. Let $\vec{v} = C_1(1)$.

The circuit receives $y \in \{0, ..., q^v - 1\}$ and parses it in basis q as $y = \sum_{i=0}^{v-1} y_i \cdot q^i$. Denote $y^{(-1)} = 0$, and for $i \in \{0, ..., v-1\}$, let $y^{(i)} = \sum_{i'=0}^{i} y_{i'} \cdot q^{i'}$. The circuit works in v iterations, where in iteration $i \in \{0, ..., v-1\}$ it has already obtained the values

$$\left\{\hat{x}\left(A^{y^{(i-1)}+j\cdot q^i} \cdot C_b(t)\right)\right\}_{t\in\mathbb{F}_q, b\in\{1,2\}, j\in[m-1]}$$

and its goal is to compute the values of

$$\left\{\hat{x}\left(A^{y^{(i-1)}+j'\cdot q^i} \cdot C_b(t)\right)\right\}_{t\in\mathbb{F}_q, b\in\{1,2\}, j'\in\{m,...,(m-1)\cdot q\}},$$

where we denote $A^0 = \mathsf{Id}$ and $\sum_{i'=0}^{-1} y_{i'} \cdot q^{i'} = 0$.

The values $\left\{\hat{x}\left(A^j \cdot C_b(t)\right)\right\}_{t\in\mathbb{F}_q, b\in\{1,2\}, j\in[m-1]}$ will be hard-wired into R_0, so the first iteration has the values it needs to start its execution. Observe that after iteration i completes successfully, the circuit has the values that it needs for iteration $i+1$. Also note that after iteration $v-1$ the circuit has learned the value $\hat{x}\left(A^y \cdot C_1(1)\right)$, as we wanted.

Thus, it remains to describe how a single iteration i is executed. The circuit R_0 uses the circuit $C_{\mathsf{LrnNext},i}$. For $j' \in \{m, ..., (m-1)\cdot q\}$, we first use Item (1) of Lemma 5.20 with value $k = y^{(i-1)} + (j'-1)\cdot q^i$ and then use Item (2) of Lemma 5.20 with value $k = y^{(i-1)} + j' \cdot q^i$. In both cases, the circuit R_0 gives $C_{\mathsf{LrnNext},i}$ access to its oracle $P^{(i)}$.

Tedious verification, which may be skipped. To carefully verify the correct use of Lemma 5.20, for each $j' \in \{m, ..., m\cdot q\}$, denote $C^{\mathsf{Nxt},j',1}$ as $C^{\mathsf{Nxt},1}$ when we use Item (1), and denote $C^{\mathsf{Nxt},j',2}$ as $C^{\mathsf{Nxt},2}$ when we use Item (2); analogously, denote $C^{\mathsf{Prv},j',1}, C^{\mathsf{Prv},j',2}$.

Now, fix $j' \in \{m, ..., m\cdot q\}$, and recall that we enter step j' (of iteration i) having already learned $\left\{\hat{x}(A^{y^{(i-1)}+j\cdot q^i} \cdot C_b(t))\right\}_{t,b,j\in[j'-1]}$ in previous steps (or in iteration $i-1$). To reduce notational clutter, for a curve $C\colon \mathbb{F}_q \to \mathbb{F}_q^v$ we will denote $\hat{x}(C) = \{\hat{x}(C(t))\}_{t\in\mathbb{F}_q}$. We also use the shorthand notation $\vec{a}^{(j)} = (a_t^{(j)})_{t\in\mathbb{F}_q}$.

- When we use Item (1), we have $C^{\mathsf{Nxt},j',1} = A^{y^{(i-1)}+j'\cdot q^i} \cdot C_1$ and $C^{\mathsf{Prv},j',1} = A^{y^{(i-1)}+(j'-1)\cdot q^i} \cdot C_2$. The evaluations we learned going into step j' include $\left\{\vec{a}^{(j)} = \hat{x}(A^{-j\cdot q^i} \cdot C^{\mathsf{Nxt},j',1})\right\}_{j\in[m-1]}$. Also, in the previous step $j'-1$ we learned $\hat{x}(C^{\mathsf{Nxt},j'-1,2}) = \hat{x}(A^{y^{(i-1)}+(j'-1)\cdot q^i} \cdot C_2)$, so in particular we learned $\left\{b_t = \hat{x}(C^{\mathsf{Prv},j',1}(w_i, R_t))\right\}_{t\in[r]}$.

- When we use Item (2), we have $C^{\mathsf{Nxt},j',2} = A^{y^{(i-1)}+j'\cdot q^i} \cdot C_2$ and $C^{\mathsf{Prv},j',2} = A^{y^{(i-1)}+j'\cdot q^i} \cdot C_1$. The evaluations we learned going into step j' include $\left\{\vec{a}^{(j)} = \hat{x}(A^{-j\cdot q^i} \cdot C^{\mathsf{Nxt},j',2})\right\}_{j\in[m-1]}$, and in the most recent usage of Item (1) we learned $\hat{x}(C^{\mathsf{Nxt},j',1}) \supseteq \left\{b_t = \hat{x}(C^{\mathsf{Prv},j',2}(w_v, R_t))\right\}_{t\in[r]}$.

The execution of the two items yields the values $\hat{x}(A^{y^{(i-1)}+j'\cdot q^i} \cdot C_b)$ for $b \in \{1,2\}$, so we can continue to step $j'+1$.

Complexity and correctness. Note that R_0 has $O(m \cdot q)$ elements of $\mathbb{F}_q$ hard-wired into it, as well as two curves (i.e., $2q$ elements of $\mathbb{F}_q^v$), a set of size r, and v circuits of size $\mathrm{poly}(q, 1/\rho)$. For its execution, it works in v iterations, and in each iteration it simulates $C_{\mathsf{LrnNext},i}$ and stores $O(m \cdot q^2)$ values. Thus, overall, R_0 can be implemented in size $\mathrm{poly}(m, \log(n))$.

Moreover, since the functionality of R_0 (given the hard-wired information) can be implemented by a uniform machine, the following holds: There is a (uniform) Turing machine that gets as input the information that is supposed to be hard-wired into R_0 (i.e., the elements of $\mathbb{F}_q$ for the first iteration, the two interleaved curves, the sampled set of size r, and the v circuits $C_{\mathsf{LrnNext},i}$) as well as an input y, and computes the value of the corresponding circuit R_0 (i.e., the R_0 that is obtained by the given "hard-wired" information) at y in time $\mathrm{poly}(m, \log(n))$. Thus, similarly to the proof of Lemma 5.17, observe that the foregoing functionality can be computed by an $O(\log(m, \log(n)))$-space-uniform circuit family $\left\{R'_{0,n}\right\}_{n\in\mathbb{N}}$ of size $\mathrm{poly}(m, \log(n))$.

The machine M that prints R_0 simulates the machine that prints $R'_0 = R'_{0,n}$, which uses space $O(\log(m, n\log(n))) \leq O(\log(n))$, and hard-wires the needed information into the corresponding input gates of R'_0. Specifically, M obtains C_1, C_2, R using the "moreover" part of Lemma 5.20, queries $\hat{x}$ at points $\left\{A^j \cdot C_b(t)\right\}_{j\in[m-1],t\in\mathbb{F}_q,b\in\{0,1\}}$, and hard-wires all of this information into the corresponding input gates for R'_0. (Recall that M can compute powers of A in space $O(\log n)$, by Proposition 5.7, and note that M can evaluate C_b for any $b \in \{0,1\}$ and at any point t in space $O(\log n) + \mathrm{polylog}(m)$, using the "moreover" part of Lemma 5.20.) In addition, the machine M computes the descriptions of $C_{\mathsf{LrnNext},i}$ for all $i \in \{0, ..., v-1\}$, using Lemma 5.17, and hard-wires them into the corresponding input gates of R'_0. Thus, M runs in space $O(\log n) + \mathrm{polylog}(m)$, and the circuit that it prints computes the mapping $y \mapsto R_0(y)$.

□

5.5 Putting It All Together: The Reconstruction Procedure

Our goal now is to prove the reconstruction part of Theorem 5.1. That is, we show an algorithm RSU that gets input 1^n, and gets oracle access to $x \in \{0,1\}^n$ and to $(1/m^2)$-next-bit-predictors $\left\{P_i \colon \{0,1\}^{j_i} \to \{0,1\}\right\}_{i\in\{0,...,v\}}$, runs in space $O(\log n)$, and prints an oracle circuit $C \colon \{0,1\}^{\log(n)} \to \{0,1\}$ of size $\mathrm{poly}(m)$ such that $C^{P_0,...,P_v}(u) = x_u$ for all $u \in [n]$.

Discrete log. By Proposition 5.9, we can print in space $O(\log n)$ an oracle circuit C_{dl} of size $\mathrm{poly}(m)$ such that $C_{\mathsf{dl}}^{P_v}(A^y \cdot \vec{1}) = y$ for all $y \in \{0,1\}^{v\cdot\log(q)}$.

***q*-ary reconstruction.** By Proposition 5.21, given oracle access to $\hat{x}$, we can print in space $O(\log n) + \mathrm{polylog}(m)$ a circuit $R_0 \colon [q^v - 1] \to \mathbb{F}_q$ of size $\mathrm{poly}(m, \log(n))$ such that $R_0^{\vec{P}}(y) = \hat{x}(A^y\vec{1})$, when $\vec{P}$ is a sequence of $(\rho/2)$-good predictors for the $L_i^{(q)}$'s. The queries to $\hat{x}$ can be answered in space $O(\log n)$, given our oracle access to x.

List-decoding. Now, the list-decoder for $\mathsf{Enc}_{\mathsf{STV}}$ from Theorem 5.8 runs in time $\mathrm{poly}(m, \log(n))$, and hence a circuit $\mathsf{Dec}_{\mathsf{STV}}$ of such size implementing its functionality can be printed in space $O(\log(m, \log(n))) = O(\log n)$. By a standard argument (see, e.g., [SU05, Lemma 4.16], following [TZS06]), given oracle access to a $(1/m^2)$-next-bit-predictor $P_i \colon \{0,1\}^{j_i} \to \{0,1\}$ for L_i, we can compute a $(\rho/2)$-good predictor $P_i^{(q)} \colon \mathbb{F}_q^{j_i} \to \mathbb{F}_q^{\bar{\rho}}$ for $L_i^{(q)}$ as follows:

- Given $w_1, ..., w_{j_i} \in \mathbb{F}_q$, for each $k \in [\ell_q]$, compute $r_k = P_i(\mathsf{Enc}_{\mathsf{STV}}(w_1)_k, ..., \mathsf{Enc}_{\mathsf{STV}}(w_{i_j})_k)$.
- Let $r = (r_1, ..., r_{\ell_q})$, and output the list of decoded messages that $\mathsf{Dec}_{\mathsf{STV}}$ outputs when given access to the corrupt codeword r.

Observe that we can implement $P_i^{(q)}$ by an oracle circuit $C_j^{(\mathsf{Dec})}$ of size $\mathrm{poly}(m, \log(n))$ (which gets oracle access to P_i), and that this circuit can be constructed in space $O(\log n)$.

Combining the ingredients. We print an oracle circuit $C_{\hat{x}}$ that computes $\hat{x}$, as follows:

- Given $u \in \mathbb{F}_q^v$, use $C_{\mathsf{dl}}(u)$ to compute $y \in \{0,1\}^{v \cdot \log(q)} \equiv [q^v - 1]$ such that $u = A^y \cdot \vec{1}$. (Note that for every u there exists such y, since A is a generator matrix.)
- Use $R_0(y)$ to compute $\hat{x}(A^y \cdot \vec{1}) = \hat{x}$. Whenever R_0 queries one of its $(\rho/2)$-good predictors, answer using $C_j^{(\mathsf{Dec})}$ and our oracle access to the next-bit-predictors.

The size of $C_{\hat{x}}$ is at most $\mathrm{poly}(m, \log(n))$, and it can be printed in space $O(\log n)$.

The final circuit C needs to compute the mapping $u \mapsto x(u)$. Recall that u represents the coefficient of some monomial in $\hat{x} \colon \mathbb{F}_q^v \to \mathbb{F}_q$, say $y_1^{e_1} \cdot y_2^{e_2} \cdot ... \cdot y_v^{e_v}$ where $\sum_{k \in [v]} e_k \le d$. The coefficient of this monomial is determined by the evaluation of $\hat{x}$ of at most d points, and thus the circuit C invokes $C_{\hat{x}}$ for d times and outputs the corresponding linear combination.

6 The Line and Tree Compressors

Before constructing the compressors, we recall (a black box version of) the distinguish to predictor transformation for the Nisan PRG (Definition 3.9) of [DPTW25].

Theorem 6.1. *For every $n = 2^\ell$, let $t = 50\ell$. For every ROBP B of length and width n vertices, there is a function $T_B \colon (\{0,1\}^{2t})^\ell \to \{0,1\}$ such that:*

1. ***Evaluability.*** *The function T_B can be computed in space $O(\log n)$, given B.*
2. ***Usefulness.*** *For every $\vec{h} = (h_1, \ldots, h_\ell)$ such that $T_B(\vec{h}) = 1$, it holds that*
$$\left| \mathbb{E}[B] - \mathbb{E}_r[B(\mathrm{NIS}_{\vec{h}})(r)] \right| \le n^{-2}.$$
3. ***Likeliness.*** *We have $\mathbb{E}[T_B(\mathbf{U}_{\ell \cdot 2t})] \ge 1 - n^{-2}$.*
4. ***D2P.*** *There is a logspace algorithm that, given B, outputs a δ-distinguish to ρ-predict D2P transformation $(\mathsf{PRED}_1, \ldots, \mathsf{PRED}_{b=\mathrm{polylog}(n)})$ for T_B, where $\delta = 1/2$ and $\rho = \Omega(1/\log^2(n))$. Moreover, there is a logspace algorithm that, given B, $i \in [b]$, and $x \in \{0,1\}^{\le \ell \cdot 2t}$, returns $\mathsf{PRED}_i(x)$.*

6.1 The Line Generator and Compressor

We first construct the line generator and compressor, which have the property that REC acts as a read-once algorithm over the truth table f (and see further explanation in Section 2.2.1).

Theorem 6.2 (Line Generator and Compressor)**.** *There are algorithms*

$$\mathsf{LINEGEN}, \mathsf{LINEREC}^{log}, \mathsf{LINEREC}^{str}$$

that work as follows. For every ROBP B of length and width n and $f \in \{0,1\}^n$, the algorithm $\mathsf{LINEGEN}(B,f)$ runs in space $O(\log n)$, and either outputs $\perp$, or ρ such that

$$|\rho - \mathbb{E}[B]| \leq n^{-2}.$$

If $\mathsf{LINEGEN}(B,f) = \perp$, then there is a machine M of description size polylog(n)*, and M runs in space* polylog(n) *and time* poly(n) *and $M(B,j) = f_j$ for every $j \in [n]$, and the following occurs:*

1. *$\mathsf{LINEREC}^{log}$ is a space $O(\log n)$ algorithm and $\mathsf{LINEREC}^{log}(B,f) = M$.*
2. *$\mathsf{LINEREC}^{str}$ is a space* polylog(n)*, time* poly(n) *algorithm and $\mathsf{LINEREC}^{str}(B,f) = M$, and $\mathsf{LINEREC}^{str}$ is read-once over f (see Definition 3.2).*

Proof. Let $T_B : \{0,1\}^s \to \{0,1\}$ be the function of Theorem 6.1 applied to B, and recall that $s = O(\log^2 n)$.

Given $f \in \{0,1\}^n$, recall that $f_{\leq i}$ constitutes the first i bits of f. Let $P_i \in \{0,1\}^n$ be defined as

$$P_i = f_{\leq i} || 0^{n-i}$$

and let SU be the generator of Theorem 5.1 with $N = n$ and $M = s$ and $\ell = O(\log(n)/\text{loglog}(n))$. Let $L_{i,j}$ be the jth list output by SU^{P_i}, and note that these lists can be computed in space $O(\log n)$ with access to P_i (and thus also with oracle access to f).

The Generator. We first define $\mathsf{LINEGEN}$. The generator enumerates over $i \in [n]$ and for each i computes the lists $L_{i,1}, \dots, L_{i,\ell}$. If there is some element $\vec{h}_u \in L_{i,j}$ such that $T_B(\vec{h}_u) = 1$, the generator takes the first such $\vec{h}_u$ and computes

$$\rho = \mathop{\mathbb{E}}_{r}[B(\text{NIS}_{\vec{h}_u}(r))]$$

where $r \in \{0,1\}^{O(\log n)}$ is as in Theorem 6.1, and the correctness of these values ρ_u follows from Item 2 of Theorem 6.1. Otherwise, $\mathsf{LINEGEN}$ outputs $\perp$. The space complexity follows directly from that of the SU generator and from Item 1 of Theorem 6.1, and from the explicitness of the Nisan PRG Definition 3.9.

The Machines $M_1, \dots, M_n$. We define a series of machines $M_1, \dots, M_n$, each with description size, space, and runtime equal to that of M in the theorem statement, where for every i we have $M_i(B,j) = f_j$ for every $j \leq i$. We then set $M = M_n$. Note that when $\mathsf{LINEGEN}$ returns $\perp$, it must be the case that for every P_i and $j \in [\ell]$, every element $\vec{h}$ of $L_{i,j}$ satisfies $T_B(\vec{h}) = 0$, and therefore by Item 3 of Theorem 6.1

$$\left| \mathop{\mathbb{E}}_{\vec{h} \leftarrow L_{i,j}} \left[T_B(\vec{h}) \right] - \mathbb{E}[T_B(\mathbf{U})] \right| \geq 1 - o(1).$$

Hence, by Item 4 of Theorem 6.1, the D2P transform for B contains a ρ-predictor for $L_{i,j}$ for every j, where $\rho = \Omega(1/\log^2(n))$. We let $k_{i,j} \in [\text{polylog}(n)]$ denote the index of the first such predictor, and for $i \in [n]$ let

$$\vec{k_i} = (k_{i,1}, \dots, k_{i,\ell}).$$

For a fixed P_i, consider the set of predictors

$$\mathsf{PRED}_{\vec{k_i}} = \mathsf{PRED}_{k_{i,1}}, \dots, \mathsf{PRED}_{k_{i,\ell}},$$

where $\mathsf{PRED}_{k_{i,j}}$ is the $k_{i,j}$th element of the D2P transform of B. When given oracle access to P_i and to $\mathsf{PRED}_{\vec{k_i}}$,[35] the algorithm RSU outputs C_i of size $\text{polylog}(n)$ such that for every $j \le i$,

$$C_i^{\mathsf{PRED}_{\vec{k_i}}}(j) = P_j = f_j.$$

We then transform this circuit into the machine M_i as follows:

Claim 6.3. *There is a space $O(\log n)$ algorithm that, given i, C_i, and $\vec{k_i}$, prints a machine M_i of description size* $\text{polylog}(n)$. *On input $j \in [n]$, M_i runs in* $\text{poly}(n)$ *time and* $\text{polylog}(n)$ *space and $M_i(B, j) = f_j$.*

Proof. Consider the machine E that, on input

$$j, B, C_i, \vec{k_i}$$

computes $C_i(j)$, and when C makes an oracle call y to some predictor $\mathsf{PRED}_k \in \mathsf{PRED}_{\vec{k_i}}$, invokes the logspace machine of Item 4 that computes the map $(y, B, k) \to \mathsf{PRED}_k(y)$. This machine E has constant description size and runs in space $\text{polylog}(n)$ and time $\text{poly}(n)$.

Our final machine M_i is the output $M_i = M'$ of Lemma 3.4 with $M = E$ and $m = (C_i, \vec{k_i})$. Note that on input (j, B), M_i outputs $E(j, B, C_i, \vec{k_i}) = f_j$, and runs in space $\text{polylog}(n)$ and space $\text{poly}(n)$ as desired, and by Lemma 3.4 can be printed in space $O(\log |m|) = O(\log n)$. ☐

The Reconstruction Algorithms. We now show how the algorithms compute the machines M_i.

First note that we can determine $k_{i,j}$ in space $O(\log n)$ with access to B and P_i by enumerating over the predictors in the D2P transform (each of which is evaluable in logspace) and testing the advantage of each predictor on the j^{th} output list of SU^{P_i} (which is computable in logspace).

- First, $\mathsf{LINEREC}^{log}(B, f)$. $\mathsf{LINEREC}^{log}$ invokes the algorithm of Claim 6.3 on input $n, C_n, \vec{k_n}$ and prints its output. Whenever this machine queries a bit of C_n, we answer using the machine RSU, where we give RSU oracle access to $P_n = f$ and $\mathsf{PRED}_{\vec{k_n}}$. Similarly, when the machine queries a bit of $\vec{k_n}$, we compute the relevant index in logspace as above. Using emulative composition, the algorithm runs in logspace.

- Next, $\mathsf{LINEREC}^{str}(B, f)$.

  Claim 6.4. *There is an algorithm* STEP *running in* $O(\text{polylog}\, n)$ *and time* $\text{poly}(n)$ *where* $\mathsf{STEP}(B, M_{i-1}, f_i) = M_i$ *for every i.*

[35] We have that $\rho > 1/M^2$ by our choice of parameters.

Proof. Note that

$$P_i = M_{i-1}(B,1)||\dots||M_{i-1}(B,i-1)||f_i||0^{n-i}$$

and hence STEP can compute any bit of P_i in time $\mathrm{poly}(n)$ and space $\mathrm{polylog}(n)$ from its inputs. The machine then computes $\vec{k_i}$ as above, then involves RSU with oracle access to P_i and $\mathsf{PRED}_{\vec{k_i}}$. Once we obtain the machine C_i (which we can store on the worktape directly), we invoke Claim 6.3 and print the output. $\square$

We then define $\mathsf{LINEREC}^{str}$ as follows. Let $M = M_0 = \perp$ be stored in workspace, and iterate from $i = 1, \dots, n$. For each iteration, compute $M' = \mathsf{STEP}(B, M, f_i)$ and set $M = M'$. It is clear that after n steps, we have that $M = M_n$, which we can then output. The space and time are immediate from that of STEP, and the fact that $\mathsf{LINEREC}^{str}$ is read-once over f follows from the fact that in each iteration i we only read the bit f_i of f (to feed into STEP). $\square$

6.2 The Tree Generator and Compressor

We now construct the tree generator and compressor. As explained in Section 2.2.2, the reconstruction algorithm (i.e., the compressor) maintains the read-once property of the line compressor, while obtaining almost-linear runtime in terms of the length of f.

Theorem 6.5 (Tree Generator and Compressor)**.** *There are algorithms* $\mathsf{GEN}, \mathsf{REC}$ *such that for every* ROBP B *of length and width* n *and* $f \in \{0,1\}^t$*:*

- **Generator:** *The algorithm* $\mathsf{GEN}(B, f)$ *runs in space* $O(\log t)$*, and outputs either* $\perp$*, or* $\rho \in [0,1]$ *such that* $|\rho - \mathbb{E}[B]| \le n^{-2}$*.*
- **Reconstruction:** *The algorithm* $\mathsf{REC}(B, f)$ *runs in time* $t \cdot \mathrm{poly}(n)$ *and space* $\mathrm{polylog}(n)$*, and is read-once over* f *(see Definition 3.2). If* $\mathsf{GEN}(B, f) = \perp$*, then* $\mathsf{REC}(B, f)$ *outputs a machine* M *of description size* $\mathrm{polylog}(n)$ *that on input* $j \in [t]$ *runs in time* $\mathrm{poly}(n)$ *and space* $\mathrm{polylog}(n)$*, and satisfies* $M(B, j) = f_j$*.*

Proof. First note that we can assume without loss of generality that $t \le n^{\log n}$, as otherwise we can take GEN to be the Nisan PRG [Nis92] (which ignores the input f and always outputs an estimate ρ), and REC can always return $\perp$ without reading its input.

We define an n-ary tree on t leaves, where the depth is $d \overset{\text{def}}{=} \lceil \log_n(t) \rceil = O((\log t)/(\log n))$. Each node V in the tree is labeled by a bitstring $\textsc{val}(V)$ of size $v = \mathrm{polylog}(n)$, or by the symbol $\star$, as follows.

For $k \in [t]$, the kth leaf has label $f_k||0^{v-1}$. Given a non-leaf node V and $i \in [n]$, we denote by $\textsc{child}(V, i)$ the ith child of V, and define the label of V as follows: If for some child $i \in [n]$ it holds that $\textsc{val}(\textsc{child}(V, i)) = \star$, then $\textsc{val}(V) = \star$; otherwise, let

$$P_V \overset{\text{def}}{=} \textsc{val}(\textsc{child}(V,1))||\cdots||\textsc{val}(\textsc{child}(V,n))$$

be the concatenation of the values of the children. Next, let

$$\textsc{val}(V) \overset{\text{def}}{=} \begin{cases} M & \mathsf{LINEGEN}(B, P_V) = \perp \\ \star & \mathsf{LINEGEN}(B, P_V) \neq \perp \end{cases}$$

where $M = M_n$ is the machine output by $\mathsf{LINEREC}^{str}(B, P_V)$ and $\mathsf{LINEREC}^{log}(B, P_V)$ (and note that we can view B as size $n \cdot \mathrm{polylog}(n)$).

Preliminary facts. First observe that, by the definition of the tree and Theorem 6.2, to estimate $\mathbb{E}[B]$ it suffices to find a node with label $\star$ whose children are labeled by non-$\star$:

Fact 6.5.1. For every tree node V, if $\textsc{val}(V) = \star$ but $\textsc{child}(V,i) \neq \star$ for every $i \in [n]$, then $\mathsf{LINEGEN}(B, P_V)$ outputs ρ that is n^{-2}-close to $\mathbb{E}[B]$.

Next, both the generator GEN and the reconstruction REC will need to compute labels of node. We will show two methods of doing so. The first method, which will be used by GEN, runs in space $O(\log t)$ and in time t^c, where the constant c may be large.

Claim 6.5.2. There is an algorithm running in space $O(\log t)$ that, given (B, V, f), returns $\textsc{val}(V)$.

Proof. Computing $\textsc{val}(V)$ reduces to computing $\mathsf{LINEGEN}$ and $\mathsf{LINEREC}^{log}$ at input (B, P_V), where

$$P_V = \textsc{val}(\textsc{child}(V,1))||\cdots||\textsc{val}(\textsc{child}(V,n)).$$

Both $\mathsf{LINEGEN}$ and $\mathsf{LINEREC}^{log}$ run in space $O(\log n)$, and to simulate access to the input P_V we use emulative composition. Since there are d levels of composition, we can compute the map $(f,V) \to \textsc{val}(V)$ for any node V in space $O(d\log(n)) = O(\log t)$, and thus GEN has this space consumption. □

The second method to compute labels, which will be used by REC, uses higher space polylog(t) but lower time $t \cdot n^c$ for a universal constant $c > 1$. This method will only be used when assuming that $\textsc{val}(V) \neq \star$ for every V (see below), and hence in this method we will not need to compute $\mathsf{LINEGEN}$ (i.e., since we know that $\mathsf{LINEGEN}(B, P_V) = \perp$ for all V). Thus we will only need to compute the machine M labeling the node. At a high level, we will do this using $\mathsf{LINEREC}^{str}$ instead of $\mathsf{LINEREC}^{log}$, and replace the naive emulative composition in the algorithm of Claim 6.5.2 with a more careful emulative composition, which exploits the fact that $\mathsf{LINEREC}^{str}(B, P_V)$ is *read-once* over P_V to compose the d levels more time efficiently.

For a node V, we let f_V be the interval of f corresponding to the leaves of the subtree rooted at f.

Claim 6.5.3. Assume that $\textsc{val}(V) \neq \star$ for all V. Then, there is an algorithm running in time $t \cdot \text{poly}(n)$ and space polylog(n) algorithm that, given (B, V, f_V), returns $\textsc{val}(V)$, and is read-once over f_V.

Proof. When V is a leaf, we can output the label in time $O(n)$ and space $O(\log n)$ by reading f_V. For a non-leaf V, to compute $\textsc{val}(V)$ we do not need to simulate $\mathsf{LINEGEN}$ (since we know that $\mathsf{LINEGEN}(V') = \perp$ for all V'), and thus our goal is to simulate $\mathsf{LINEREC}^{str}$ at input (B, P_V).

Recall that $\mathsf{LINEREC}^{str}(B, P_V)$ is read-once over P_V, and in particular it reads all the bits of $\textsc{val}(\textsc{child}(V,i))$ in P_V before reading the first bit of $\textsc{val}(\textsc{child}(V,i+1))$ in P_V. We simulate $\mathsf{LINEREC}^{str}(B, P_V)$, and each time it queries the first bit of the label $\textsc{val}(\textsc{child}(V,i))$ of the next child $i \in [m]$, we discard the label of the previous child, recurse to compute the label of $\textsc{child}(V,i)$, store it, and continue simulating $\mathsf{LINEREC}^{str}(B, P_V)$.

We prove by induction on the level j of V (where $j = 1$ corresponds to the leaves) that this recursive algorithm runs in time and space at most

$$t_j = (2n)^j \cdot n^c, \qquad S_j = j \cdot \log^c(n)$$

respectively, for some universal constant c, and that it is read-once over f_V.

The base case of $j = 1$ is trivial. For the inductive step, recall that $\mathsf{LINEREC}^{str}$ runs in time $\text{poly}(n)$ and space $\text{polylog}(n)$. Each time it queries the bits of $\text{VAL}(\text{CHILD}(V, i))$ for some $i \in [n]$, the algorithm recurses with a node at level $j-1$, so the runtime is

$$\text{poly}(n) + n \cdot t_{j-1} = \text{poly}(n) + n \cdot (2 \cdot n)^{j-1} \cdot n^c \leq (2n)^j \cdot n^c$$

where the final step comes from choosing the universal cosntant c sufficiently large.

For space, $\mathsf{LINEREC}^{str}$ runs in space $\text{polylog}(n)$, and we additionally incur an additive space overhead of $\text{polylog}(n)$, corresponding to the size of one label (i.e., since we store one label at a time). Finally, as $\mathsf{LINEREC}$ queries $\text{VAL}(\text{CHILD}(W, i))$ in increasing order of i, we have that final algorithm is read-once over f_V. □

The Generator. The generator GEN iterates over nodes in DFS order and for each node V it computes the label $\text{VAL}(V)$ using Claim 6.5.2. For the first V such that $\text{VAL}(V) = \star$, the generator outputs $\mathsf{LINEGEN}(B, P_V)$; if no such V exists, the generator outputs $\perp$. Note that the first encountered node with label $\text{VAL}(V) = \star$ must have all children labeled with non-$\star$ (since the algorithm processes nodes in DFS order, and leaves are labeled with non-$\star$); hence, by Fact 6.5.1, when the algorithm outputs $\mathsf{LINEGEN}(B, P_V)$, this is a value ρ that is n^{-2}-close to $\mathbb{E}[B]$.

By Claim 6.5.2, computing the label at each node can be done in space $O(\log t)$. In addition, the algorithm needs to perform DFS in an m-ary tree with d levels, and doing so yields an additive space overhead of $O(d \cdot \log(m)) = O(\log t)$.

The Reconstruction. Now suppose that $\mathsf{GEN}(B, f) = \perp$, which means that $\text{VAL}(V) \neq \star$ for every node V. We first compute the label $M_R \overset{\text{def}}{=} \text{VAL}(R)$ of the root R, using Claim 6.5.3. We then show that given M_R, there is a logspace algorithm to output M.

Claim 6.6. *There is a space $O(\log n)$ algorithm that, given M_R, outputs M.*

Proof. Let E be the machine that on input

$$j, B, M_R$$

computes the path $j_1, \ldots, j_d$ from R to the jth leaf in the tree, then sets $M_{V_0} = M_R$ and iteratively computes

$$M_{V_i} = \text{VAL}(\text{CHILD}(V_{i-1}, j_i))$$

by evaluating $M_{V_{i-1}}(B, k)$ for the indices k that correspond to the j_ith child. This iterates until M' computes the label at leaf j, from which it outputs f_j. This machine E has constant description size and runs in time $\text{poly}(n)$ and space $\text{polylog}(n)$.

Our final machine M is the output of Lemma 3.4 with $M = E$ and $m = M_R$. Note that on input (j, B), M outputs $E(j, B, M_R) = f_j$, and runs in space $\text{polylog}(n)$ and space $\text{poly}(n)$ as desired, and by Lemma 3.4 can be printed in space $O(\log m) = O(\log n)$. □

The final reconstruction algorithm simply invokes this algorithm and prints its output. □

7 Win-Win Algorithms and Derandomization

In this section we use the line generator and compressor and the tree generator and compressor (from Sections 6 and 6.1) to prove the results in the introduction. In Section 7.1 we show two "win-win pairs of algorithms", which are more general technical statements of Theorems 1.8 and 2.3. Then, in Section 7.2 we prove Theorems 1.3 and 1.4 and in Section 7.3 we prove Theorem 1.6.

7.1 Win-win Pairs of Algorithms

We first prove a more general technical statement Theorem 2.3, from which our main results will follow as corollaries. The proof follows the description in Section 2.2.

Theorem 7.1 (win-win pair of algorithms for composition and derandomization)**.** *For every $R \in$* **BPL***, there is a constant $c > 0$ such that the following holds. For every $d \in \mathbb{N}$, and $k : \mathbb{N} \to \mathbb{N}$ such that $k(n)$ is computable in space $O(\log n)$, and length-preserving $g : \{0,1\}^* \to \{0,1\}^*$ computable in quasilinear time and logspace, there are algorithms $\mathcal{A}_1, \mathcal{A}_2$ such that for every $x \in \{0,1\}^n$ at least one of the following occurs:*

1. *$\mathcal{A}_1(x)$ computes $g^{(k)}(x||0^{n^d})$ in time $n^d \cdot n^c$ and space* $\mathrm{polylog}(n)$*.*
2. *$\mathcal{A}_2(x)$ computes $R(x)$ in space $O(k(n) \cdot \log n)$.*

Moreover, both algorithms report if they fail to compute the answer, and never exceed their resource bounds.

Theorem 2.3 can be obtained from Theorem 7.1 by taking $t(n) = n^d$ to be a sufficiently large polynomial as a function of R and of $\delta > 0$, in which case $n^{d+c} < t(n)^{1+\delta}$.

Proof of Theorem 7.1. For $i \in \{0, \dots, k\}$, let $x_i = g^{(i)}(k)$, and let $L = n + n^d$ be the length of each x_i. Let $\mathcal{M}$ be the quasilinear time, logspace machine that computes g. For every x, let B_x be the ROBP of size n^c of Proposition 3.7 for the language R, and note that the map $x \to B_x$ can be computed in space $O(\log n)$ (and hence time $\mathrm{poly}(n)$).

Observe that when $k \geq \sqrt{\log n}$, the algorithm $\mathcal{A}_2$ exists unconditionally [SZ99]. Hence, we may assume without loss of generality that $k < \sqrt{\log n}$.

The Algorithm $\mathcal{A}_2$. We define the algorithm as follows. Iterate over $i = 1, \dots, k$ in sequence, and run the logspace algorithm $\mathsf{GEN}(B_x, f = x_i)$ of Theorem 6.5. If some invocation returns ρ, then $\mathcal{A}_2$ returns 1 iff $\rho > 1/2$; if no iteration returns ρ, the algorithm aborts and returns $\perp$. By Proposition 3.7 and the guarantee on ρ, whenever $\mathcal{A}_2(x)$ returns a bit, it correctly computes $R(x)$. Finally, to answer the queries of GEN to x_i, the algorithm uses k levels of emulative composition, and so it runs in space $O(k \log n)$.

The Algorithm $\mathcal{A}_1$. We define the algorithm as follows. It iterates over $i = 0, \dots, k$, and in each iteration i it obtains a machine machine M_i of size $\mathrm{polylog}(n)$ such that for every $j \in [L]$, $M_i(x, B, j)$ runs in space $\mathrm{polylog}(n)$ and time $\mathrm{poly}(n)$ and

$$M_i(x, B, j) = (x_i)_j.$$

The machine M_0 simply references x (and for $j > n$ answers with 0 immediately, so it runs in the claimed time and space). Then the iterative step from i to $i+1$ works as follows.

Assuming we have such a machine M_i, we prepare to simulate (on separate worktapes) the machines $\mathsf{REC}(B, x_{i+1})$ of Theorem 6.5, and $\mathcal{M}(x_i)$. We begin to simulate REC. Recall that this simulation queries x_{i+1} in a read-once stream. When REC queries the next bit of x_{i+1}, we progress the simulation of $\mathcal{M}$ until it produces the next output bit, then *pause* the simulation of $\mathcal{M}$. During this simulation, when $\mathcal{M}$ queries x_i at position j, we evaluate $M_i(x, B, j)$ in time $\mathrm{poly}(n)$ and space $\mathrm{polylog}(n)$ and return this bit.

Once REC returns an output M_{i+1}, we first *verify* that M_{i+1} is correct. We again simulate $\mathcal{M}(x_i)$, and each time it outputs bit j of x_{i+1}, we verify that $M_{i+1}(x, B, j)$ returns the correct value (and does not exceed time or space bounds). If this test fails, we return $\perp$. Otherwise, we delete M_i and proceed to the next phase. At the end, we have a machine M_k, and we output $M_k(x,B,1),\ldots,M_k(x,B,L)$.

Observe that if $\mathcal{A}_2$ outputs $\perp$, then for all $i \in [k]$ the i^{th} iteration of $\mathcal{A}_1$ computes a machine M_i such that $M(x,B,j) = (x_i)_j$ for all j, in which case $\mathcal{A}_1$ outputs $g^{(k)}(x||0^{n^d})$. We now analyze the runtime of phase i. Simulating $\mathcal{M}(x_i)$ (which we do twice) requires $\widetilde{O}(n^d)$ time for the simulation, and each query to the input $\mathcal{M}(x_i)$ incurs a time overhead of $\mathrm{poly}(n)$, for a total time of $n^d \cdot \mathrm{poly}(n)$. Simulating REC likewise requires time $n^d \cdot \mathrm{poly}(n)$, so the total time is $k \cdot \widetilde{O}(n^d) \cdot \mathrm{poly}(n) = n^d \cdot n^c$ from choosing a large enough constant c (recall that $k < \sqrt{\log n}$). The space consumption is dominated by storing at most two machines M_i, and simulating REC, so it is $\mathrm{polylog}(n)$. $\square$

Using Theorem 7.1, we now prove Theorem 1.8, which gives a win-win pair of algorithms for circuit evaluation vs derandomization. The insight going into the proof is that circuit evaluation can be computed as a composition of low-space algorithms.

Theorem 1.8 (win-win pair of algorithms for derandomization and circuit evaluation)**.** *For every $\varepsilon > 0$ and $R \in \mathsf{BPL}$ and $C \in \mathsf{unif_{logspace}NC}^{1+\varepsilon}$,[36] there are algorithms $\mathcal{A}_1, \mathcal{A}_2$ such that for every $x \in \{0,1\}^n$, either:*

1. *$\mathcal{A}_1(x)$ computes $C(x)$ in time $\mathrm{poly}(n)$ and space $\mathrm{polylog}(n)$.*
2. *$\mathcal{A}_2(x)$ computes $R(x)$ in space $O(\log^{1+\varepsilon} n)$.*

Moreover, both algorithms report if they fail to compute the answer, and never exceed their resource bounds.

Proof. Fix an arbitrary $R \in \mathsf{BPL}$ and $C \in \mathsf{unif_{logspace}NC}^{1+\varepsilon}$ and let c be the constant of Theorem 7.1. Let C_n be the circuit of language C on inputs of size n. Choose $d \in \mathbb{N}$ such that $|C_n| \le n^{d-1}$ for every n. For an input x to C_n, let $V_i = V_i(x) \in \{0,1\}^{n^{d-1}}$ be the gate values in layer $i \cdot \log(n)$ (where if a layer is smaller than this we pad the values with zeroes). Moreover, we assume wlog that the final layer values are $V_{\ell = O(\log^{\varepsilon} n)}$, and the output gate is the first entry.

We then define g as

$$g\left(x||0^{n^d}\right) = V_1||0^p||\langle 1\rangle$$

where the encoding $\langle 1\rangle$ of the integer 1 is of length $\lceil \log(\ell)\rceil$, and the padding length $p = n^d + n - n^{d-1} - \lceil \log(\ell)\rceil$ makes g length preserving, and for $i \in [\ell - 1]$

$$g(V_i||0^p||\langle i\rangle) = V_{i+1}||0^p||\langle i+1\rangle.$$

Claim 7.2. *g can be computed in logspace.*

Proof. Since C_n is logspace uniform, there is a logspace algorithm that given 1^n and i, prints the layers of C_n from $i\log(n)$ to $(i+1)\log n$. Letting C^i be the restriction of C_n to these layers, the circuit evaluation problem $(V_i, C^i) \to V_{i+1}$ can be computed in space $O(\log n)$, essentially via the argument that $\mathsf{NC}^1 \subseteq \mathsf{L}$. $\square$

[36]We say a language is in $\mathsf{unif_{logspace}NC}^{1+\varepsilon}$ if there is a logspace-uniform sequence of circuits $\{C_n\}_{n\in\mathbb{N}}$, where C_n has size $\mathrm{poly}(n)$ and depth $O(\log^{1+\varepsilon} n)$, and $x \in \{0,1\}^n$ is in the language iff $C_n(x) = 1$.

Finally, we apply Theorem 7.1 with $k = \ell$. For every input x, either $\mathcal{A}_2(x) = R(x)$ and $\mathcal{A}_2$ runs in space $O(k \log n) = O(\log^{1+\varepsilon} n)$, or

$$\mathcal{A}_1(x) = g^{(k)}(x||0^{n^d}) = C_n(x)||z$$

for some $z \in \{0,1\}^{n^d+n-1}$ and $\mathcal{A}_1$ runs in simultaneous time $\mathrm{poly}(n)$ and space $\mathrm{polylog}(n)$, and so a trivial modification of $\mathcal{A}_1$ decides the language $C(x)$ in the same time and space (both algorithms do not exceed their resource bounds and report failure if they do not succeed). □

Scaling up Theorem 1.8 to linear space and exponential time, we obtain a win-win result referring to complexity classes (rather than to instance-wise algorithms).

Theorem 7.3 (scaled-up win-win result)**.** *For every $\varepsilon > 0$, at least one of the following hold.*

1. $\mathsf{unif}_{\mathsf{logspace}}\mathsf{SIZEDEP}[2^n, n^{1+\varepsilon}] \subseteq i.o.\mathsf{TISP}[2^{O(n)}, \mathrm{poly}(n)]$.
2. $\mathsf{BPSPACE}[n] \subseteq \mathsf{SPACE}[O(n^{1+\varepsilon})]$.

Proof. For every $R \in \mathsf{BPSPACE}[n]$, denote by $R' = \left\{x||y||0^{2^{|x|}} : |x| = |y| \wedge x \in R\right\}$ a padded version such that $R' \in \mathsf{BPL}$. For every $L \in \mathsf{unif}_{\mathsf{logspace}}\mathsf{SIZEDEP}[2^n, n^{1+\varepsilon}]$, denote by $L' = \left\{x||y||0^{2^{|x|}} : |x| = |y| \wedge y \in L\right\}$ a padded version such that $L \in \mathsf{unif}_{\mathsf{logspace}}\mathsf{NC}^{1+\varepsilon}$.

We do a case analysis, based on whether or not the following assumption is true:

> For every $R \in \mathsf{BPSPACE}[n]$ there is $L \in \mathsf{unif}_{\mathsf{logspace}}\mathsf{SIZEDEP}[2^n, n^{1+\varepsilon}]$ such that, when instantiating Theorem 1.8 with R' and L' and with the parameter value ε, for all but finitely many inputs x there exists $y \in \{0,1\}^{|x|}$ such that $\mathcal{A}_1(x||y||0^{2^{|x|}})$ fails.

If the assumption above holds, we prove that $\mathsf{BPSPACE}[n] \subseteq \mathsf{SPACE}[O(n^{1+\varepsilon})]$. Specifically, for every $R \in \mathsf{BPSPACE}[n]$, we instantiate Theorem 1.8 with R' and with the corresponding L' from the assumption. Given input $x \in \{0,1\}^n$, we enumerate over all $y \in \{0,1\}^n$, and run $\mathcal{A}_2(x||y||0^{2^n})$; for the first y such that $\mathcal{A}_2$ does not fail, we report the outcome of $\mathcal{A}_2$ (and if no such y exists, we abort). Correctness follows since, by our assumption, for all but finitely many inputs x, there will be some y such that $\mathcal{A}_1(x||y||0^{2^n})$ fails, meaning that $\mathcal{A}_2(x||y||0^{2^n})$ does not fail, and hence our algorithm outputs $R'(x||y||0^{2^{|x|}}) = R(x)$. This algorithm runs in space $O(n^{1+\varepsilon})$.

Otherwise (if the assumption above does not hold), we prove that $\mathsf{unif}_{\mathsf{logspace}}\mathsf{SIZEDEP}[2^n, n^{1+\varepsilon}] \subseteq i.o.\mathsf{TISP}[2^{O(n)}, \mathrm{poly}(n)]$. Specifically, for every $L \in \mathsf{unif}_{\mathsf{logspace}}\mathsf{SIZEDEP}[2^n, n^{1+\varepsilon}]$, we instantiate Theorem 1.8 with R' from the assumption and with L'. Given input $y \in \{0,1\}^n$, we enumerate over all $x \in \{0,1\}^n$, and run $\mathcal{A}_1(x||y||0^{2^n})$, reporting an outcome for the first x in which $\mathcal{A}_1$ does not fail (otherwise aborting). By our assumption, there are infinitely many inputs x such that for all $y \in \{0,1\}^{|x|}$, the algorithm $\mathcal{A}_1$ succeeds. For every input length n corresponding to one of those x's, our algorithm will correctly decide $L(y)$ for all $y \in \{0,1\}^n$. □

7.2 Consequences of Composition Lower Bounds

We now deduce Theorems 1.3 and 1.4 as corollaries of Theorem 7.1. The proofs amount to choosing suitable parameters and to presenting the different composed functions $A_1, A_2, \ldots$ as a single function g that will be plugged into Theorem 7.1 (this is done by syntactic manipulations).

Theorem 1.3 (hardness of composing low-space t-time algorithms implies derandomization)**.** *Suppose that there is $\delta > 0$ such that for every polynomial $t(n)$ and constant $\varepsilon > 0$ the following holds. There are two algorithms A_1 and A_2 running in time t and space $O(\log n)$ such that any algorithm computing $A_2(A_1(x))$ successfully on an ε-fraction of inputs $x \in \{0,1\}^n$ requires time-space product $t^{1+\delta}$. Then $\mathsf{BPL} \subseteq \cap_{\varepsilon>0}\mathsf{zavg}_\varepsilon\mathsf{L}$.*

Proof. Fix an arbitrary $R \in \mathsf{BPL}$ and $\varepsilon > 0$ and let c be the constant of Theorem 7.1. Choose $d \in \mathbb{N}$ sufficiently large such that $n^d \cdot n^c < n^{d(1+\delta)-1}$ and let $t(n) = n^d$. Let A_1, A_2 be the time-t, logspace algorithms guaranteed to exist per the assumption.

Defining the function g. We define g differently on inputs of the form $x||0^{n^d}$, indicating that A_1 needs to be invoked on x, and on inputs whose last bit is 1 (see below), indicating that A_2 needs to be invoked on the prefix. Specifically, we define

$$g\left(x||0^{|x|^d}\right) = A_1(x)||0^*||l||1^{|l|}$$

where $l = |A_1(x)| \in \{0,1\}^{\log t}$, and the padding 0^* is taken such that the function is length preserving (and this is possible since A_1 is computable in time $t = n^d$ and so $n + n^d \geq |A_1(x)| + \log(t) + 1$). On inputs that end with the bit 1, we define

$$g\left(y||0^p||l||1^{|l|}\right) = A_2(y)||0^{p+l+|l|+1-|A_2(y)|}.$$

where $l = |y|$ and p is a parameter indicating the length of the 0-padding. On inputs not of the forms above, g is defined trivially.

Note that g is computable in quasilinear time and logspace, by applying the algorithms A_1 and A_2. Hence, we can invoke Theorem 7.1 with $k = 2$.

Correctness. We claim that on all but an ε fraction of inputs $x \in \{0,1\}^n$, the algorithm $\mathcal{A}_2(x)$ from Theorem 7.1 computes $R(x)$ in space $O(\log n)$. If this did not occur, on an ε fraction of these x we would have that

$$\mathcal{A}_1\left(x||0^{n^d}\right) = g^{(2)}\left(x||0^{n^d}\right) = A_2(A_1(x))||0^{n^d+n-|A_2(A_1(x))|}$$

and $\mathcal{A}_1$ runs in time-space product

$$\text{polylog}(n) \cdot n^d \cdot n^c < t^{1+\delta}$$

which contradicts the assumed hardness. Thus, for every $R \in \mathsf{BPL}$ and $\varepsilon > 0$ we have a derandomization which succeeds on all but an ε-fraction of inputs, so the result follows. □

We now prove Theorem 1.4. We state the result for composing a single linear-time, logspace algorithm $k(n)$ times, where k can be superconstant. Note that composing k distinct algorithms can be modeled as composing a single algorithm k times, since we can define the algorithm A' where for $i \in \{0,1\}^{\log k}$, $A'(x||i) = A_i(x)||i+1$.

Theorem 7.4. *For every $R \in \mathsf{BPL}$ and $\varepsilon > 0$, there exists a polynomial $p(n)$ such that the following holds. Suppose there is $k : \mathbb{N} \to \mathbb{N}$ computable in space $O(\log n)$, and a linear time and logspace algorithm A such that any algorithm computing $A^{(k)}(x)$ successfully on an ε-fraction of inputs x requires time p for space* polylog(n)*. Then $R \in \mathsf{zavg}_\varepsilon\mathsf{SPACE}[O(k \cdot \log n)]$.*

Proof. Let c be the constant of Theorem 7.1 wth $R = R$. Choose $d \in \mathbb{N}$ sufficiently large such that $n^d \cdot n^c < n^{2d-n}$ and let $p(n) = n^{2d}$. Let $\langle a \rangle$ be a in binary.

We define g as

$$g\left(x||0^{n^d}\right) = x||0^*||\langle|x|\rangle||1^{|\langle|x|\rangle|}$$

where we choose the padding 0^* such that g is length preserving. Moreover, define

$$g\left(y||0^*||\langle|y|\rangle||1^{|\langle|y|\rangle|}\right) = A(y)||0^*||\langle|A(y)|\rangle||1^{|\langle|A(y)|\rangle|}.$$

where we again choose the padding to make g length preserving (and on all other inputs g is defined trivially). We can do this as

$$y_i = (A)^{(i)}(x)$$

is of length at most $O(1)^k \cdot n$ as A runs in linear time,[37] so $|x_k| + 2\log|x_k| < n^d + n$. Moreover, g can be computed in quasilinear time and logspace by applying A. Finally, we set $k = k+1$ and apply Theorem 7.1.

We claim that on all but an ε fraction of inputs $x \in \{0,1\}^n$, the algorithm $\mathcal{A}_2(x)$ computes $R(x)$ in space $O(k \cdot \log n)$. If this did not occur, on an ε fraction of these x we would have that

$$\mathcal{A}_1\left(x||0^{n^d}\right) = g^{(k+1)}\left(x||0^{n^d}\right) = A^{(k)}||0^*||l||1^{|l|}$$

and $\mathcal{A}_1$ runs in time $n^d \cdot n^c < p(n)$ and space $\mathrm{polylog}(n)$, which contradicts the assumed hardness of computing $A^{(k)}$. Thus, we have that for every $R \in \mathbf{BPL}$ and $\varepsilon > 0$ we have a derandomization which succeeds on all but an ε-fraction of inputs, so the result follows. $\square$

Finally, we prove the scaled-up result in Theorem 1.5, which only assumes a worst-case lower bound.

Theorem 1.5 (worst-case hardness implies derandomization)**.** *Suppose there is $\delta > 0$ such that for every sufficiently large $d \in \mathbb{N}$ there is $k \in \mathbb{N}$ for which the following holds. There are algorithms $A_1, \ldots, A_k$ so that:*

- *On input $x \in \{0,1\}^n$, letting $x_i = A_i \circ \ldots \circ A_1(x)$, we have that $A_{i+1}(x_i)$ is computable in time 2^{dn} and space $O(n)$ for every i.*
- *For every algorithm B running in time $2^{dn(1+\delta)}$ and space $\mathrm{poly}(n)$, for every sufficiently large n there is $x \in \{0,1\}^n$ such that $B(x) \neq A_k \circ \ldots A_1(x)$.*

Then $\mathbf{BPSPACE}[n] \subseteq \mathbf{SPACE}[O(n)]$.

Proof. For arbitrary $R \in \mathbf{BPSPACE}[n]$, let $R' \in \mathbf{BPL}$ be the padded language where for $n = |x| = |y|$,

$$R'(x||y||0^{2^n}) = R(x)$$

(and the language is defined trivially on inputs not of the foregoing form). Let c be the constant of Theorem 7.1 with $R = R'$. Let $d \in \mathbb{N}$ be sufficiently large such that $(1+\delta)d > c+1$, and let $A_1, \ldots, A_k$ be the algorithms that are guaranteed to exist per the assumption with parameter value d.

[37]We can assume $k = o(\sqrt{\log n})$ since otherwise the claimed derandomization unconditionally exists by [SZ99, Hoz21], and hence even for superconstant k the final output is of length $c^k \cdot n = n^{1+o(1)}$, so the padding is valid.

Note that for every $y \in \{0,1\}^n$, we have that $y_i = A_i \circ \ldots \circ A_1(y)$ is of length at most 2^{dn} (and WLOG we assume the length is exactly 2^{dn}). Thus, let g be the length-preserving function (defined analogously to the function g in Theorem 7.4) such that

$$g^{(k)}(x||y||0^{2^n+(2n+2^n)^d}) = A_k(\ldots(A_1(y))).$$

and note that g is computable in quasilinear time and space logarithmic in its input length.

Next, we apply Theorem 7.1 with

$$R = R', \qquad k = k, \qquad d = d.$$

Let $N \in \mathbb{N}$ be the input length above which the second bullet of the assumption holds. We claim that for every $n > N$, for every $x \in \{0,1\}^n$ there is some $y \in \{0,1\}^n$ where

$$\mathcal{A}_2(x||y||0^{2^n}) = R(x)$$

and note $\mathcal{A}_2$ runs in space $O(n)$. Given this claim, the derandomization works as follows. Given x, if $|x| \le N$ it returns a hard-coded answer and otherwise enumerates over $y \in \{0,1\}^n$ in space $O(n)$ and once $\mathcal{A}_2(x||y||0^{2^n})$ produces an output, returns.

Now assume this claim does not occur. Consider the algorithm B that, given $y \in \{0,1\}^n$, enumerates over $x \in \{0,1\}^n$ and invokes $\mathcal{A}_1(x||y||0^{2^n})$, and returns the first non-$\perp$ output. We can see that if B produces an output it returns

$$g^{(k)}(x||y||0^{2^n+(2n+2^n)^d}) = A_k(\ldots(A_1(y)))$$

and B runs in space $\text{poly}(n)$ and time

$$(2n+2^n)^d \cdot (2n+2^n)^c < 2^{(1+\delta)n}.$$

Thus, if there is $x \in \{0,1\}^{\ge N}$ where $\mathcal{A}_2(x||y||0^{2^n}) = \perp$ for every y, then $B(y)$ computes the correct output for every y, contradicting the assumption. $\square$

7.3 Derandomization From Hardness for ROBPs

We give a formal statement of Theorem 1.6, which asserts that hardness of compression implies derandomization.

Theorem 7.5 (derandomization from hardness of compression by ROBPs)**.** *There is $c > 1$ and a sequence $\{\mathcal{B}_n\}_{n\in\mathbb{N}}$ of multi-output read-once branching programs*[38] *where $\mathcal{B}_n$ has width 2^{n^c} and length 2^n such that the following holds.*

- *There is a space $O(n^c)$ algorithm sthat, on input 1^n, outputs $\mathcal{B}_n$ (i.e. $\mathcal{B}$ is logspace uniform).*
- *There is an $O(n)$-space algorithm that, on input $(1^n, r)$, outputs the state v_r reached in $\mathcal{B}_n$ on (possibly partial) input r.*[39]

Finally, suppose there is an algorithm that, given 1^n, runs in space $O(n)$ and outputs a list of strings $f_{n,1}, \ldots, f_{n,m}$ such that $\mathcal{B}(f_{n,i}) \ne f_{n,i}$ for some $i \in [m]$. Then $\mathsf{BPSPACE}[n] \subseteq \mathsf{SPACE}[O(n)]$.

Proof. First, let L_{derand} be a $\mathsf{prBPSPACE}[n]$-complete language nder $O(n)$-space reductions. For $x \in \{0,1\}^n$, let B_x be the ROBP (of size and length $S \overset{\text{def}}{=} 2^{O(n)}$) of Proposition 3.7 applied with $R = L_{derand}$ and $x = x$. Let $\{B_x\}_{x\in\{0,1\}^n}$ be all such ROBPs on input length n, and note that this set can be produced by a space $O(n)$ algorithm.

[38]Each state v in the final layer of the ROBP is labeled with a string $p_y \in \{0,1\}^{2^n}$.

[39]If the input r is of length 2^n, $\mathcal{S}$ likewise prints the label p_y.

Defining the ROBP Family. We define the multi-output ROBP $\mathcal{B}_n$ as follows. In layer $i \in [N]$, a state (M, x) corresponds to a $2^{\text{polylog}(S)} + n$ bit string, and we add a special reject state $\perp$.

Definition 7.6. We say a state (M, x) in layer i is valid if the following occurs. For some $f \in \{0,1\}^i$ we have $\mathsf{LINEGEN}(B_x, f||0^{S-i}) = \perp$, and $M = \mathsf{LINEREC}^{log}(B_x, f||0^{S-i})$. We say that (M, x) $\mathsf{represents}$ f if this occurs, and (M, x) $\mathsf{canonically\ represents}$ f if for every $y < x$, (M, y) does not represent f.

Note that if (M, x) is valid, it holds that $M(B_x, j) = f_j$ for every $j \leq i$, and hence (M, x) represents a unique f. We can verify if M is valid in small space:

Claim 7.7. *There is a space* $\text{poly}(n)$ *algorithm that determines if* (M, x) *is valid.*

Proof. Note that we can run $M(B_x, j)$ for every x, j in space $\text{poly}(n)$ and time $2^{O(n)}$ (and halt if it exceeds the resource bounds of the machine printed by $\mathsf{LINEREC}$). If the output is $f||0^{S-i}$ for some $f \in \{0,1\}^i$, we then test if $\mathsf{LINEGEN}(B_x, f||0^{S-i}) = \perp$ (where we produce f via emulative composition). If this holds, by definition (M, x) is valid and represents f. $\square$

If a state is invalid, we add $0, 1$ edges from it to $\perp$ (and we always add edges from $\perp$ to itself).

For a valid state (M, x) in layer i, let f be the (unique) string it represents. For each $b \in \{0,1\}$, if there a valid state (M', x') in layer $i + 1$ that canonically represents $f||b$, we add an edge labeled b from (M, x) to (M', x'), and otherwise add a b-edge to $\perp$.

We can determine these edge relationships very space efficiently:

Claim 7.8. *Given* $(M, x), f, b$ *where* (M, x) *represents* f*, there is a space* $O(n)$ *algorithm that returns the canonical* (M', x') *that represents* $f||b$*, and* $\perp$ *if no such state exists.*

Proof. We iterate over $x' \in \{0,1\}^n$ and find the least value such that $\mathsf{LINEGEN}(B_{x'}, f||b||0^{S-i-1}) = \perp$, and if no such x' exists then no state represents $f||b$ and we return $\perp$. Otherwise we store this x' and run $\mathsf{LINEREC}^{log}(B_{x'}, f||b||0^{S-i-1})$, and denoting the output M', by definition (M', x') canonically represents $f||b$. $\square$

Finally, for a valid state (M, x) in layer N, we let the label of (M, x) be the string f it represents. We let the start state be $(M_0, 0^n)$, where M_0 outputs 0 on every input.

We show that all strings f that do not reach a final state labeled with f must be useful for derandomization:

Claim 7.9. *For every* f *where* $\mathcal{B}_n(f) \neq f$*, there is* $j \in [N]$ *where for every* $x \in \{0,1\}^n$*,*

$$\mathsf{LINEGEN}(B_x, f_{\leq j}||0^{S-j}) = \rho_x.$$

Proof. Taking the contrapositive, let $f \in \{0,1\}^N$ be such that for every $j \leq N$, there is $y_j \in \{0,1\}^n$ such that

$$\mathsf{LINEGEN}(B_{y_j}, f_{\leq j}||0^{S-j}) = \perp .$$

Assume y_j is the lexicographically first y such that $\mathsf{LINEGEN}(B_y, f_{\leq j}||0^{S-j}) = \perp$. Then by Theorem 6.2, there is M_i where

$$M_i = \mathsf{LINEREC}^{log}(B_{y_j}, f_{\leq i}||0^{S-i}),$$

and by the definition of $\mathcal{B}_n$ there is an edge labeled f_i from (M_{i-1}, y_{i-1}) to (M_i, y_i) for every i. Thus, f will reach a valid final state with label f. $\square$

Note that for every x, $|\rho_x - \mathbb{E}[B_x]| \leq 1/10$ by Theorem 6.2.

Uniformity. The first algorithm enumerates over layers $i \in [N]$ and strings (M, x) of length $2^{\text{polylog}(S)} + n$. For each such state, we determine if (M, x) is valid in logspace via Claim 7.7, and if it is we recover the string f it represents by evaluating $M(B_x, j)$, and determine the edges to the next layer via Claim 7.8. It is clear that this procedure ultimately prints the entire ROBP, and runs in space $\text{poly}(n)$, and by padding the ROBP to a larger $2^{\text{poly}(n)}$ width we obtain that the ROBP is logspace uniform.

The second algorithm checks if for each $i \le |r|$, the prefix $r_{\le i}$ reaches a (canonical) valid state (M_i, x_i). If this ever does not occur, we return $\perp$ (as clearly then r must have traversed an edge to $\perp$). Otherwise, we return $(M_{|r|}, x_{|r|})$. Since $r||0^{S-|r|}$ is of length S and $\mathsf{LINEGEN}, \mathsf{LINEREC}^{log}$ run in space $O(\log S) = O(n)$ (and we can enumerate over B_x in space $O(n)$), $\mathcal{S}$ runs in space $O(n)$.

Derandomization From Refutation. Finally, suppose there is an $O(n)$-space algorithm $\mathcal{R}$ such that $\mathcal{R}(1^n)$ outputs $f_{n,1}, \dots, f_{n,m}$ where $\mathcal{B}_n(f_{n,i}) \ne f_{n,i}$ for some i. By Claim 7.9, for some $j \le N$, $\mathsf{LINEGEN}(B_x, (f_{n,i})_{\le j}||0^{S-j}) = \rho_x$ for every $x \in \{0,1\}^n$ where ρ_x is a $(1/10)$-additive estimate of $\mathbb{E}[B_x]$. Thus, to derandomize L_{derand} on x, the final algorithm enumerates over $i \in [m]$ and $j \in [N]$, finds the first i, j for which this holds, computes ρ_x, and checks if $\rho_x < 1/2$. The correctness of the derandomization follows from Theorem 6.2 and Claim 7.9. $\square$

Another improved reduction of derandomization to weak hardness assumptions. We also improve on a second set of results from [DPT24, DPTW25] that deduce derandomization from hardness for uniform deterministic procedures. Recall that the work of [DPT24] obtained derandomized from hardness of linear space for $\mathsf{SPACE}[O(n)]$-uniform circuits of size $2^{\varepsilon n}$, and [DPTW25] obtained the same derandomization from hardness of $\mathsf{TISP}[2^{O(n)}, \text{poly}(n)]$-uniform circuits of size $\text{poly}(n)$ (where the circuits have access to a $\mathsf{SPACE}[\varepsilon n]$ oracle). Note that the latter result obtains derandomization from hardness for exponentially smaller circuits, but at the cost of worse uniformity. We prove a result whose assumption enjoys both relaxations simultaneously:

Theorem 7.10. *There is a constant $c > 1$ such that the following holds. Suppose there exists a constant $\varepsilon > 0$ such that $\mathsf{SPACE}[n]$ is hard for $\mathsf{SPACE}[cn]$-uniform circuits of size n^c with access to $\mathsf{SPACE}[\varepsilon n]$. Then, $\mathsf{BPSPACE}[n] \subseteq \mathsf{SPACE}[O_\varepsilon(n)]$.*

The proof is identical, mutatis mutandis, to [DPTW25, Theorem 4], except that we substitute their reconstructive PRG (Theorem 5.1) with our Theorem 5.1.

Acknowledgements

The authors thank Ryan Williams for permission to include his proof of Theorem 1 in the current work, and for helpful discussions. The authors thank Dean Doron and Dhruv Rohtangi for helpful conversations, and Robert Andrews for explanations about computing GCDs of polynomials.

References

[Abr91] Karl Abrahamson. Time-space tradeoffs for algebraic problems on general sequential machines. *Journal of Computer and System Sciences*, 43(2):269–289, 1991.

[Ajt02] Miklós Ajtai. Determinism versus nondeterminism for linear time RAMs with memory restrictions. volume 65, pages 2–37. 2002. Special issue on STOC, 1999 (Atlanta, GA).

[Ajt05] Miklós Ajtai. A non-linear time lower bound for Boolean branching programs. *Theory of Computing*, 1:149–176, 2005.

[BC82] Allan Borodin and Stephen A. Cook. A time-space tradeoff for sorting on a general sequential model of computation. *SIAM J. Comput.*, 11(2):287–297, 1982.

[BCM13] Paul Beame, Raphaël Clifford, and Widad Machmouchi. Element distinctness, frequency moments, and sliding windows. In *Proc. 54th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, 2013.

[BCT25] Marshall Ball, Lijie Chen, and Roei Tell. Towards free lunch derandomization from necessary assumptions (and OWFs). In *Proc. 40th Annual IEEE Conference on Computational Complexity (CCC)*, pages Art. No. 31, 20. 2025.

[Bea91] Paul Beame. A general sequential time-space tradeoff for finding unique elements. *SIAM Journal on Computing*, 20(2):270–277, 1991.

[BHST87] László Babai, Péter Hajnal, Endre Szemerédi, and György Turán. A lower bound for read-once-only branching programs. *Journal of Computer and System Sciences*, 35(2):153–162, 1987.

[BJS01] Paul Beame, T. S. Jayram, and Michael Saks. Time-space tradeoffs for branching programs. volume 63, pages 542–572. 2001. Special issue on FOCS 98 (Palo Alto, CA).

[BKM$^+$24] Tatiana Belova, Alexander S. Kulikov, Ivan Mihajlin, Olga Ratseeva, Grigory Reznikov, and Denil Sharipov. Computations with polynomial evaluation oracle: ruling out superlinear SETH-based lower bounds. In *Proc. 35th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1834–1853, 2024.

[Bor77] Allan Borodin. On relating time and space to size and depth. *SIAM Journal on Computing*, 6(4):733–744, 1977.

[BS83] Piotr Berman and Janos Simon. Lower bounds on graph threading by probabilistic machines. In *Proc. 24th Annual Symposium on Foundations of Computer Science*, 1983.

[BSSV03] Paul Beame, Michael Saks, Xiaodong Sun, and Erik Vee. Time-space trade-off lower bounds for randomized computation of decision problems. *Journal of the ACM*, 50(2):154–195, 2003.

[BV02] Paul Beame and Erik Vee. Time-space tradeoffs, multiparty communication complexity, and nearest-neighbor problems. In *Proc. 34th Annual ACM Symposium on Theory of Computing (STOC)*, pages 688–697. ACM, New York, 2002.

[BvH82] Allan Borodin, Joachim von zur Gathen, and John Hopcroft. Fast parallel matrix and gcd computations. *Information and Control*, 52(3):241–256, 1982.

[BW15] Samuel R. Buss and Ryan Williams. Limits on alternation trading proofs for time-space lower bounds. *Computational Complexity*, 24(3):533–600, 2015.

[CJSW21] Lijie Chen, Ce Jin, Rahul Santhanam, and Ryan Williams. Constructive separations and their consequences. In *Proc. 62nd Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 646–657, 2021.

[CKMS24] Nikolai Chukhin, Alexander S. Kulikov, Ivan Mihajlin, and Arina Smirnova. Deriving nonuniform lower bounds from uniform nondeterministic lower bounds. *arXiv*, 2024.

[CLO$^+$23] Lijie Chen, Zhenjian Lu, Igor Carboni Oliveira, Hanlin Ren, and Rahul Santhanam. Polynomial-time pseudodeterministic construction of primes. *arXiv preprint arXiv:2305.15140*, 2023.

[Cob66] Alan Cobham. The recognition problem for the set of perfect squares. In *Proc. 7th Annual Symposium on Switching and Automata Theory (SWAT)*, page 78–87, 1966.

[Coo79] Stephen A. Cook. Deterministic CFL's are accepted simultaneously in polynomial time and log squared space. In *Proc. 11th Annual ACM Symposium on Theory of Computing (STOC)*. 1979.

[Coo81] Stephen A. Cook. Towards a complexity theory of synchronous parallel computation. *Enseign. Math. (2)*, 27(1-2):99–124, 1981.

[CRTY20] Lijie Chen, Ron D. Rothblum, Roei Tell, and Eylon Yogev. On exponential-time hypotheses, derandomization, and circuit lower bounds. In *Proc. 61st Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 13–23, 2020.

[CT21a] Lijie Chen and Roei Tell. Hardness vs randomness, revised: Uniform, non-black-box, and instance-wise. In *Proc. 62nd Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 125–136, 2021.

[CT21b] Lijie Chen and Roei Tell. Simple and fast derandomization from very hard functions: Eliminating randomness at almost no cost. In *Proc. 53st Annual ACM Symposium on Theory of Computing (STOC)*, pages 283–291, 2021.

[CTW23] Lijie Chen, Roei Tell, and Ryan Williams. Derandomization vs refutation: A unified framework for characterizing derandomization. In *Proc. 64 Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, 2023. To appear.

[Die07] Scott Diehl. Lower bounds for swapping arthur and merlin. In *Proc. 11th International Workshop on Randomization and Approximation Techniques in Computer Science (RANDOM)*, pages 449–463, 2007.

[Din20] Itai Dinur. Tight time-space lower bounds for finding multiple collision pairs and their applications. In *Proc. Advances in cryptology (EUROCRYPT)*, pages 405–434. [2020] ©2020.

[DPT24] Dean Doron, Edward Pyne, and Roei Tell. Opening up the distinguisher: A hardness to randomness approach for $\mathsf{BPL} = \mathsf{L}$ that uses properties of BPL. In *Proc. 56th Annual ACM Symposium on Theory of Computing (STOC)*, 2024.

[DPTW25] Dean Doron, Edward Pyne, Roei Tell, and Ryan Williams. When connectivity is hard, random walks are easy with non-determinism. In *Proc. 57th Annual ACM Symposium on Theory of Computing (STOC)*, 2025.

[DS18] Yuval Dagan and Ohad Shamir. Detecting correlations with little memory and communication. In *Proc. 31st Annual Conference on Learning Theory*. 2018.

[DvM06] Scott Diehl and Dieter van Melkebeek. Time-space lower bounds for the polynomial-time hierarchy on randomized machines. *SIAM Journal on Computing*, 36(3):563–594, 2006.

[EPA99] Jeff Edmonds, Chung Keung Poon, and Dimitris Achlioptas. Tight lower bounds for st-connectivity on the NNJAG model. *SIAM J. Comput.*, 28(6):2257–2284, 1999.

[FLvMV05] Lance Fortnow, Richard Lipton, Dieter van Melkebeek, and Anastasios Viglas. Time-space lower bounds for satisfiability. *Journal of the ACM*, 52(6):835–865, 2005.

[For97] Lance Fortnow. Nondeterministic polynomial time versus nondeterministic logarithmic space: time-space tradeoffs for satisfiability. In *Proc. 12th Annual IEEE Conference on Computational Complexity (CCC)*, pages 52–60. 1997.

[GC25] Stefan Grosser and Marco Carmosino. Student-teacher constructive separations and (un)provability in bounded arithmetic: witnessing the gap. In *Proc. 57th Annual ACM Symposium on Theory of Computing (STOC)*, pages 1341–1347, 2025.

[GHR95] Raymond Greenlaw, H. James Hoover, and Walter L. Ruzzo. *Limits to parallel computation:* P*-completeness theory.* The Clarendon Press, Oxford University Press, New York, 1995.

[GL89] Oded Goldreich and Leonid A. Levin. A hard-core predicate for all one-way functions. In *Proc. 21st Annual ACM Symposium on Theory of Computing (STOC)*, pages 25–32, 1989.

[Gol08] Oded Goldreich. *Computational Complexity: A Conceptual Perspective.* Cambridge University Press, New York, NY, USA, 2008.

[Gra94] Étienne Grandjean. Linear time algorithms and NP-complete problems. *SIAM Journal on Computing*, 23(3):573–597, 1994.

[GRT19] Sumegha Garg, Ran Raz, and Avishay Tal. Time-space lower bounds for two-pass learning. In *Proc. 34th Annual IEEE Conference on Computational Complexity (CCC)*, volume 137, pages Art. No. 22, 39. 2019.

[Guo13] Zeyu Guo. Randomness-efficient curve samplers. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques (APPROX/RANDOM)*, pages 575–590. Springer, 2013.

[GUV09] Venkatesan Guruswami, Christopher Umans, and Salil Vadhan. Unbalanced expanders and randomness extractors from Parvaresh-Vardy codes. *Journal of the ACM*, 56(4):Art. 20, 34, 2009.

[HAB02] William Hesse, Eric Allender, and David A. Mix Barrington. Uniform constant-depth threshold circuits for division and iterated multiplication. *Journal of Computer and System Sciences*, 65(4):695–716, 2002.

[Hoz21] William M. Hoza. Better pseudodistributions and derandomization for space-bounded computation. In *Proceedings of the 25th International Conference on Randomization and Computation (RANDOM)*, pages 28:1–28:23, 2021.

[Hoz22] William M. Hoza. Recent progress on derandomizing space-bounded computation. *Bull. EATCS*, 138, 2022.

[HS66] F. C. Hennie and Richard Edwin Stearns. Two-tape simulation of multitape turing machines. *J. ACM*, 13(4):533–546, 1966.

[IW97] Russell Impagliazzo and Avi Wigderson. **P** = **BPP** if **E** requires exponential circuits: derandomizing the XOR lemma. In *Proc. 29th Annual ACM Symposium on Theory of Computing (STOC)*, pages 220–229, 1997.

[IW98] Russell Impagliazzo and Avi Wigderson. Randomness vs. time: De-randomization under a uniform assumption. In *Proc. 39th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 734–743, 1998.

[Kan84] Ravindran Kannan. Towards separating nondeterminism from determinism. *Mathematical Systems Theory. An International Journal on Mathematical Computing Theory*, 17(1):29–45, 1984.

[Kor22] Oliver Korten. Derandomization from time-space tradeoffs. In *Proc. 37th Annual IEEE Conference on Computational Complexity (CCC)*, 2022.

[KRW95] Mauricio Karchmer, Ran Raz, and Avi Wigderson. Super-logarithmic depth lower bounds via the direct sum in communication complexity. *Computational Complexity*, 5(3-4):191–204, 1995.

[LPT24] Jiatu Li, Edward Pyne, and Roei Tell. Distinguishing, predicting, and certifying: On the long reach of partial notions of pseudorandomness, 2024.

[LTWY23] Xin Lyu, Avishay Tal, Hongxun Wu, and Junzhao Yang. Tight time-space lower bounds for constant-pass learning. In *Proc. 64th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 1195–1202. 2023.

[LV99] Richard J. Lipton and Anastasios Viglas. On the complexity of SAT. In *Proc. 40th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 459–464. 1999.

[LV20] Andrea Lincoln and Nikhil Vyas. Algorithms and lower bounds for cycles and walks: Small space and sparse graphs. In Thomas Vidick, editor, *11th Innovations in Theoretical Computer Science Conference, ITCS 2020, January 12-14, 2020, Seattle, Washington, USA*, volume 151 of *LIPIcs*, pages 11:1–11:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

[Mul86] Ketan Mulmuley. A fast parallel algorithm to compute the rank of a matrix over an arbitrary field. In *Proceedings of the eighteenth annual ACM symposium on Theory of computing*, pages 338–339, 1986.

[MV05] Peter Bro Miltersen and N. V. Vinodchandran. Derandomizing arthur-merlin games using hitting sets. *Comput. Complex.*, 14(3):256–279, 2005.

[MW19] Dylan M. McKay and Richard Ryan Williams. Quadratic time-space lower bounds for computing natural functions with a random oracle. In *Proc. 10th Conference on Innovations in Theoretical Computer Science (ITCS)*, volume 124, pages Art. No. 56, 20. 2019.

[MW21] Abhijit S. Mudigonda and R. Ryan Williams. Time-space lower bounds for simulating proof systems with quantum and randomized verifiers. In *Proc. 12th Conference on Innovations in Theoretical Computer Science (ITCS)*, pages Art. No. 50, 20. 2021.

[Nis91] Noam Nisan. Pseudorandom bits for constant depth circuits. *Combinatorica*, 11(1):63–70, 1991.

[Nis92] Noam Nisan. Pseudorandom generators for space-bounded computation. *Combinatorica*, 12(4):449–461, 1992.

[Nis94] Noam Nisan. $\mathsf{RL} \subseteq \mathsf{SC}$. *Computational Complexity*, 4:1–11, 1994.

[NW94] Noam Nisan and Avi Wigderson. Hardness vs. randomness. *Journal of Computer and System Sciences*, 49(2):149–167, 1994.

[Pag05] Jakob Pagter. On Ajtai's lower bound technique for R-way branching programs and the Hamming distance problem. *Chicago Journal of Theoretical Computer Science*, pages 1–16, 2005.

[PPST83] Wolfgang J. Paul, Nicholas Pippenger, Endre Szemerédi, and William T. Trotter. On determinism versus non-determinism and related problems. In *Proc. 24th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*. 1983.

[PRZ23] Edward Pyne, Ran Raz, and Wei Zhan. Certified hardness vs. randomness for log-space. In *64th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2023*, 2023.

[Pyn24] Edward Pyne. Derandomizing logspace with a small shared hard drive. In *Proc. 39th Annual IEEE Conference on Computational Complexity (CCC)*, pages 4:1–4:20, 2024.

[Raz19] Ran Raz. Fast learning requires good memory: a time-space lower bound for parity learning. *Journal of the ACM*, 66(1):Art. 3, 18, 2019.

[RS82] Stefan Reisch and Georg Schnitger. Three applications of Kolmogorov-complexity. In *Proc. 23rd Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 45–52. 1982.

[RSW06] Omer Reingold, Ronen Shaltiel, and Avi Wigderson. Extracting randomness via repeated condensing. *SIAM Journal on Computing*, 35(5):1185–1209, 2006.

[Sav70] Walter J. Savitch. Relationships between nondeterministic and deterministic tape complexities. *Journal of Computer and System Sciences*, 4:177–192, 1970.

[STV01] Madhu Sudan, Luca Trevisan, and Salil Vadhan. Pseudorandom generators without the XOR lemma. *Journal of Computer and System Sciences*, 62(2):236–266, 2001.

[SU05] Ronen Shaltiel and Christopher Umans. Simple extractors for all min-entropies and a new pseudorandom generator. *Journal of the ACM*, 52(2):172–216, 2005.

[SU07] Ronen Shaltiel and Christopher Umans. Low-end uniform hardness vs. randomness tradeoffs for AM. In *Proc. 39th Annual ACM Symposium on Theory of Computing (STOC)*, pages 430–439, 2007.

[Sud97] Madhu Sudan. Decoding of Reed Solomon codes beyond the error-correction bound. *J. Complex.*, 13(1):180–193, 1997.

[SZ99] Michael E. Saks and Shiyu Zhou. $\mathsf{BP_HSPACE}[S] \subseteq \mathsf{DSPACE}[S^{3/2}]$. *Journal of Computer and System Sciences*, 58(2):376–403, 1999.

[Tel20] Roei Tell. Karp-lipton theorems: Translating non-uniform "collapses"' into uniform "collapses". https://sites.google.com/site/roeitell/Expositions, 2020. Accessed: 2025-09-10.

[Tom82] Martin Tompa. Two familiar transitive closure algorithms which admit no polynomial time, sublinear space implementations. *SIAM Journal on Computing*, 11(1):130–137, 1982.

[Tou01] Iannis Tourlakis. Time-space tradeoffs for SAT on nonuniform machines. *Journal of Computer and System Sciences*, 63(2):268–287, 2001.

[TV07] Luca Trevisan and Salil Vadhan. Pseudorandomness and average-case complexity via uniform reductions. *Computational Complexity*, 16(4):331–364, 2007.

[TZS06] Amnon Ta-Shma, David Zuckerman, and Shmuel Safra. Extractors from Reed-Muller codes. *Journal of Computer and System Sciences*, 72(5):786–812, 2006.

[Wig92] Avi Wigderson. The complexity of graph connectivity. In *Mathematical Foundations of Computer Science (MFCS)*, volume 629 of *Lecture Notes in Computer Science*, pages 112–132. Springer, 1992.

[Wil06] Ryan Williams. Inductive time-space lower bounds for SAT and related problems. *Computational Complexity*, 15(4):433–470, 2006.

[Wil08] R. Ryan Williams. Time-space tradeoffs for counting NP solutions modulo integers. *Computational Complexity*, 17(2):179–219, 2008.

[Wil13] Ryan Williams. Alternation-trading proofs, linear programming, and lower bounds. *ACM Transactions on Computation Theory*, 5(2):Art. 6, 49, 2013.

[Wil24] Ryan Williams. The orthogonal vectors conjecture and non-uniform circuit lower bounds. In *Proc. 65th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 1372–1387. [2024] ©2024.

[Wil25] Ryan Williams. Personal communication, 2025.

[Yes84] Yaacov Yesha. Time-space tradeoffs for matrix multiplication and the discrete Fourier transform on any general sequential random-access computer. *Journal of Computer and System Sciences*, 29(2):183–197, 1984.

[YZ24] Huacheng Yu and Wei Zhan. Randomized vs. deterministic separation in time-space tradeoffs of multi-output functions. In *Proc. 15th Conference on Innovations in Theoretical Computer Science (ITCS)*, pages Art. No. 99, 15. 2024.

[Zuc97] David Zuckerman. Randomness-optimal oblivious sampling. *Random Structures & Algorithms*, 11(4):345–367, 1997.

A Proof of Theorem 5.19

We restate Theorem 5.19 and prove it. The proof amounts to implementing Sudan's list-decoding algorithm for the Reed-Solomon code using appropriate (known) tools so that the algorithm is implementable in logspace-uniform $\mathcal{NC}$.

Theorem A.1. *Let $q, p, d\colon \mathbb{N} \to \mathbb{N}$ and $\mu\colon \mathbb{N} \to [0,1]$ be computable in logspace. There is a logspace-uniform circuit family that gets as input 1^n and a representation of a finite field $\mathbb{F}_q$ and p distinct pairs $\left\{(a_i, b_i) \in \mathbb{F}_q^2\right\}_{i\in[p]}$ such that $\mu > 2\sqrt{d/p}$, and outputs a list of at most $2/\mu$ polynomials that contains every polynomial τ of degree at most d satisfying $\Pr_{i\in[m]}[\tau(a_i) = b_i] \geq \mu$. The circuit size is $\mathrm{poly}(|\mathbb{F}_q|)$ and its depth is $\mathrm{polylog}(|\mathbb{F}_q|)$.*

Proof. The algorithm is an implementation of Sudan's algorithm [Sud97], following the implementation in [CT21b, Theorem B.13]. We follow the proof of the latter, and note the necessary changes.[40] In their statement, they assume that the field is prime and that the number of points is sufficiently small, i.e. $p < |\mathbb{F}|/8d$; and they only conclude the existence of a randomized algorithm, rather than a deterministic one. (In addition, they bound the number of output polynomials by $\mathrm{poly}(|\mathbb{F}|)$, but their proof shows that every polynomial with agreement at least μ is included in the output list, and by Sudan's original result [Sud97], there are at most $2/\mu$ such polynomials.)

The algorithm has two steps: Constructing a bivariate polynomial Q, and outputting all linear factors of Q of a specific form. We mention the necessary changes in each step:

1. **Constructing Q.** This step amounts to constructing a certain homogeneous linear system with p equations and $O(p)$ variables (which can indeed be done in logspace-uniform deterministic $\mathcal{NC}$, see [CT21b, Proof of Theorem B.13]) and then finding a solution for this system.

   In [CT21b] the algorithm for solving the linear system is randomized (see Lemma B.4), where the only randomized component is an algorithm computing the rank of a $p \times O(p)$ matrix (see Lemma B.2). Computing the rank is repeated p times, for p different matrices.

   We replace the randomized algorithm for computing rank with the deterministic algorithm by Mulmuley [Mul86]. Given a matrix A, his algorithm works as follows:

   (a) Transform A into a symmetric matrix $\bar{A} = \begin{pmatrix} 0 & A \\ A^T & 0 \end{pmatrix}$, doubling the rank.

   (b) Extend $\mathbb{F}_q$ to the field of fractions $\mathbb{F}_q(x)$ where x is a transcendental variable.

   (c) Compute the characteristic polynomial $p'(t)$ of the matrix $C = X \cdot \bar{A}$, where X is the diagonal matrix such that $X_{ii} = x^{i-1}$.

   (d) The rank of $\bar{A}$ is the largest u such that t^u divides $p'(t)$ (this is proved in [Mul86]).

   Indeed, all of these operations can be done in logspace-uniform $\mathcal{NC}$. To verify this, note that the non-trivial steps are performing field arithmetic in $\mathbb{F}_q(x)$ (which reduces to polynomial multiplication and division over $\mathbb{F}_q$) and computing a determinant (to compute the characteristic polynomial p'), both of which are well-known to be implementable in logspace-uniform $\mathcal{NC}$ (see, e.g., the verification in [CT21b, Lemmas B.1 and B.2]).

   We note that in [CT21b], several statements mentioning other algorithmic components used in the proof assume that the field is prime (see Lemmas B.1 and B.3). This assumption is not needed, as long as a representation of the field is given; specifically, the aforementioned

[40]For easy comparison, note that the notation in [CT21b] refers to absolute number t of elements in the list with which τ agrees, whereas we use refer to relative agreement and use the notation $\mu = t/p$.

algorithmic components rely on the algorithms of Hesse, Allender, and Barrington [HAB02], which perform arithmetic over the integers in dlogtime-uniform $\mathcal{TC}^0$. When we have a representation of $\mathbb{F}_q$ (as a prime and an irreducible polynomial), we can use integer arithmetic to perform computation over $\mathbb{F}_q$ (when elements are represented as polynomials).

2. **Factoring Q.** In this step we find all factors of Q of the form $y - \tau(x)$, using the Hensel lifting technique. (The final output of the algorithm consists of all polynomials τ such that $y - \tau(x)$ is a factor of Q.) In [CT21b] this is performed in logspace-uniform deterministic $\mathcal{NC}$, but the implementation relies on the fact that the field is prime and that p is small. We modify their implementation to avoid this assumption.

   In more detail, the Hensel lifting step relies on the assumption that Q factors as $(y - \tau(x)) \cdot h$ where the two factors are coprime.[41] To guarantee this, we first make Q monic in y by a linear transformation of the variables, i.e. defining $Q_{\alpha,\beta}(x, y) = Q(\beta \cdot x, \beta \cdot y + \alpha \cdot x)$ for suitable $\alpha, \beta \in \mathbb{F}_q$. It is well-known that $Q_{\alpha,\beta}$ is monic for some $\alpha, \beta \in \mathbb{F}_q$, assuming that $|\mathbb{F}_q| > 4dp$,[42] and we can find α, β in logspace-uniform $\mathcal{NC}$ of size $\mathrm{poly}(|\mathbb{F}_q|)$ by brute-force. (We will justify the assumption $|\mathbb{F}_q| > 4dp$ below.) Factors of $Q_{\alpha,\beta}$ can be transformed to factors of Q in logspace-uniform $\mathcal{NC}$ by a linear transformation.

   Next, we check whether $\partial Q_{\alpha,\beta}/\partial y \neq 0$. If this is indeed the case, then compute $Q' = Q_{\alpha,\beta}/gcd(Q_{\alpha,\beta}, \frac{\partial Q_{\alpha,\beta}}{\partial y})$, which is the square-free part of $Q_{\alpha,\beta}$; we compute the GCD in logspace-uniform $\mathcal{NC}$ using the algorithm of Borodin, von zur Gathen, and Hopcroft [BvH82]. Otherwise, if $\partial Q_{\alpha,\beta}/\partial y = 0$, it must be the case that $Q_{\alpha,\beta}(x, y) = y^{c \cdot p_0}$ where c is an integer and p_0 is the field's characteristic (since $Q_{\alpha,\beta}$ is monic in y), so we obtained a full factorization of $Q_{\alpha,\beta}$.

   Now, for each $s \in \mathbb{F}_q$, let $Q'_s(x, y) = Q'(x, y + s)$. When $\mathbb{F}_q$ is sufficiently large, there exists $s \in \mathbb{F}_q$ such that $Q'_s(x, 0)$ has no repeated factors. Specifically, we need $|\mathbb{F}_q| > \mathrm{poly}(p)$, and we will justify this assumption below. For each $s \in \mathbb{F}_q$ and for each $t \in \mathbb{F}_q$ (in parallel), we run the Hensel lifting algorithm with Q'_s and with a guess t for the value of $Q'_s(x, 0)$; an implementation in logspace-uniform $\mathcal{NC}$ is spelled out in [CT21b, Proposition B.11].[43]

Let us now describe the entire algorithm, while justifying the assumptions. We will work over a sufficiently large extension of $\mathbb{F}_q$, to guarantee that the field size is at least $\mathrm{poly}(p, d)$; note that $p < |\mathbb{F}_q|^2$, and thus a constant-degree extension suffices, which can be found by brute-force. Then we construct the bivariate Q, by constructing the linear system described in [CT21b, Proof of Theorem B.13] and finding a solution for the system using [CT21b, Lemma B.4], while replacing the randomized algorithm for rank described there with the algorithm of Mulmuley described above. Next, we pre-process Q in parallel as described above; and for each $t \in \mathbb{F}_q$, we run the Hensel lifting procedure in [CT21b, Proposition B.11] to recover a univariate τ (or $\perp$). Let S be the set of τ's given by the Hensel lifting procedures in all branches; we output the subset of S in which the polynomials are over the original field $\mathbb{F}_q$. □

B Circuit Lower Bounds from Composition Lower Bounds

In this appendix we prove that lower bounds for *non-deterministically* computing the composition of two algorithms (that run in super-polynomial time) imply strong circuit lower bounds. Com-

[41] In [CT21b], to guarantee this they performed a non-standard preprocessing step that relies on their assumptions about $\mathbb{F}_q$ and p. We use a more standard preprocessing.

[42] This is because the degree of $disc_x(f)$ is at most $4dp$, since Q has individual degree at most $\sqrt{dp}$.

[43] The statement of [CT21b, Proposition B.11] assumes that the field is prime, but this is not used in the proof.

pared with our main results, the hypothesized lower bound here is much stronger – it holds even for non-deterministic machines that try to compute the composition – and the conclusion is also much stronger (i.e., strong circuit lower bounds, rather than derandomization of bounded-space machines). Indeed, known time-space lower bounds that hold even for non-deterministic machines are weaker than those known for deterministic machines (see [BJS01]).

We first state and prove a general parameterized result, and then state two corollaries obtained by instantiating the parameters to specific useful values.

Theorem B.1 (circuit lower bounds from composition lower bounds)**.** *Let $s, t \colon \mathbb{N} \to \mathbb{N}$ be time-constructible functions. Assume that there are functions A_1 and A_2 such that:*

1. *On any input $x \in \{0,1\}^n$, the function $A_1(x)$ is computable in time $t(n)$ and space $O(\log(t(n))$, and on input $A_1(x)$ the function A_2 is computable in time $t(n)$ and space $O(\log(t(n)))$.*

2. *For any non-deterministic algorithm B running in time $O(t) + \tilde{O}(s(n + O(\log t)))$ and space $\tilde{O}(s(n + \log t)) + O(\log(t))$ there exists x such that $B(x) \neq A_2(A_1(x))$.*

Then, $\mathsf{SPACE}[n]$ does not have circuits of size s.

Proof. Assume towards a contradiction that $\mathsf{SPACE}[n]$ has circuits of size s, and let A_1 and A_2 be two functions as in our hypothesis. Without loss of generality, assume that $A_1(x)$ always has length $t(|x|)$.

Define $a_1(x,i) = A_1(x)_i$ (i.e., when a_1 gets input $(x,i) \in \{0,1\}^n \times \{0,1\}^{\log(t(n))}$ it outputs the i^{th} output bit of $A_1(x)$), and note that on inputs of length $\bar{n} = n + \log(t(n))$ the function a_1 is computable in space $O(\log(t(n))) \leq O(\bar{n})$. By our assumption, a_1 has circuits of size $s(\bar{n})$.

Our algorithm for computing $A_2(A_1(x))$ first guesses a circuit C of size $s(\bar{n})$ and stores it in memory. Then it checks that for every $i \in [t]$ we have $C(x,i) = A_1(x)_i$, as follows. It runs A_1 on input x, storing a counter for the index of the current output bit, and whenever A_1 prints the next bit i, the algorithm simulates $C(x,i)$ and compares the outcomes. If this check fails, the algorithm rejects the non-deterministic guess. Note that this step can be performed in non-deterministic time $O(t(n)) + \tilde{O}(s(\bar{n}))$ and space $\tilde{O}(s(\bar{n})) + O(\log(t(n)))$.

If the algorithm did not reject, at this point it has a circuit $C'(i) = C(x,i)$ whose truth-table is $A_1(x)$. The algorithm simulates A_2 on virtual input $A_1(x)$, keeping track of the input head, and whenever A_2 accesses input bit i the algorithm simulates $C'(i)$. This step can be performed in the same time and space as the first step. □

The following two corollaries are obtained by instantiating Theorem B.1 with the parameter values noted in the statements below:

Corollary B.2 (composition lower bounds for exponential time imply lower bounds for exponential-sized circuits)**.** *If there are two functions computable in time $t(n) = 2^{O(n)}$ and space $O(\log t)$ whose composition cannot be computed in non-deterministic time $O(t)$ and space t^{ε}, for some constant $\varepsilon > 0$, then $\mathsf{SPACE}[n]$ does not have circuits of size $s(n) = 2^{\delta \cdot n}$, for some constant $\delta > 0$.*

Corollary B.3 (composition lower bounds for super-polynomial time imply lower bounds for polynomial-sized circuits)**.** *If there are two functions computable in time $t(n) = n^{\omega(1)}$ and space $O(\log t)$ whose composition cannot be computed in non-deterministic time $O(t)$ and space $t^{o(1)}$, then $\mathsf{SPACE}[n]$ does not have circuits of size $s(n) = n^c$, for any constant $c \geq 1$. Consequently (relying on the existence of a $\mathsf{SPACE}[n]$-complete problem), $\mathsf{SPACE}[n] \not\subset \mathsf{P}/\mathrm{poly}$.*

Recall that a lower bound as in the conclusion of Corollary B.3 is not currently known even for the class $\mathsf{EXP}^{\mathsf{NP}}$, let alone for PSPACE.

Context and related results. It is well-known that circuit lower bounds follow from lower bounds for uniform machines that use randomness and non-determinism (i.e., for **MA**-type machines; this follows from standard Karp-Lipton-style results, and see, e.g., [Tel20] and [CRTY20, Appendix B] for descriptions). However, there are fewer results along the lines of Theorem B.1, which deduce circuit lower bounds from lower bounds for uniform non-deterministic machines that do not use randomness (i.e., for **NP**-type machines).

The most relevant previous result is that of Korten [Kor22], who showed that if $\mathsf{NTIME}[T]$ is hard for one-tape non-deterministic machines running in time $T^{1+\varepsilon}$ and space T^{ε}, for some $T = 2^{O(n)}$, then E^{NP} is hard for circuits of essentially maximal size. In comparison, Theorem B.1 uses a stronger assumption (since the hard function arises specifically from the composition of two low-space algorithms) and the conclusion is incomparable, but refers to a circuit lower bound in a significantly smaller class (i.e., in $\mathsf{SPACE}[n]$ rather than in E^{NP}).

For additional results deducing circuit lower bounds from lower bounds for **NP**-type machines see [BKM$^+$24,CKMS24], and for a recent work that deduces circuit lower bounds from lower bounds for uniform machines that *do not use non-determinism*, see [Wil24].