Polynomial-Time Pseudodeterministic Construction of Primes

Lijie Chen
UC Berkeley
lijiechen@berkeley.edu

Zhenjian Lu
University of Oxford
zhenjian.lu@cs.ox.ac.uk

Igor C. Oliveira
University of Warwick
igor.oliveira@warwick.ac.uk

Hanlin Ren
University of Oxford
hanlin.ren@cs.ox.ac.uk

Rahul Santhanam
University of Oxford
rahul.santhanam@cs.ox.ac.uk

May 25, 2023

Abstract

A randomized algorithm for a search problem is *pseudodeterministic* if it produces a fixed canonical solution to the search problem with high probability. In their seminal work on the topic, Gat and Goldwasser [GG11] posed as their main open problem whether prime numbers can be pseudodeterministically constructed in polynomial time.

We provide a positive solution to this question in the infinitely-often regime. In more detail, we give an *unconditional* polynomial-time randomized algorithm B such that, for infinitely many values of n, $B(1^n)$ outputs a canonical n-bit prime p_n with high probability. More generally, we prove that for every dense property Q of strings that can be decided in polynomial time, there is an infinitely-often pseudodeterministic polynomial-time construction of strings satisfying Q. This improves upon a subexponential-time construction of Oliveira and Santhanam [OS17].

Our construction uses several new ideas, including a novel bootstrapping technique for pseudodeterministic constructions, and a quantitative optimization of the uniform hardness-randomness framework of Chen and Tell [CT21], using a variant of the Shaltiel–Umans generator [SU05].

Contents

1   Introduction

How hard is it to construct an n-bit prime[1]? This is a fundamental problem in number theory and in complexity theory. Under reasonable assumptions, the problem is solvable in deterministic polynomial time. In more detail, Cramér's conjecture [Cra36] in number theory asserts that the largest prime gap in any consecutive sequence of n-bit numbers is $O(n^2)$. Assuming this conjecture, we can solve the prime construction problem efficiently by testing the first $O(n^2)$ integers greater than 2^{n-1} for primality and outputting the first one, where the primality tests are done efficiently using the algorithm of Agrawal, Kayal and Saxena [AKS04]. An independent source of evidence for the efficiency of prime construction is the complexity-theoretic conjecture that $\mathsf{DTIME}(2^{O(n)})$ requires Boolean circuits of exponential size on almost all input lengths. Under this conjecture, we can use the Impagliazzo–Wigderson pseudorandom generator [IW97] to *derandomize* the simple randomized algorithm that outputs a random n-bit number, using the facts that primality testing is in polynomial time and that an $\Omega(1/n)$ fraction of n-bit numbers are prime.

However, we seem very far from either settling Cramér's conjecture or proving strong complexity lower bounds. The best upper bound we can prove on the gap between consecutive n-bit primes is $2^{(0.525+o(1))n}$ [BHP01], and no super-linear circuit lower bounds are known for $\mathsf{DTIME}(2^{O(n)})$ [LY22]. Indeed, the best unconditional result we have so far is that deterministic prime construction can be done in time $2^{(0.5+o(1))n}$ [LO87], which is very far from the polynomial-time bound we seek. The Polymath 4 project (see [TCH12]) sought to improve this upper bound using number-theoretic techniques but did not achieve an unconditional improvement.

In contrast to the situation with deterministic prime construction, it is easy to generate an n-bit prime *randomly*, as mentioned above: simply generate a random n-bit number, test it for primality in polynomial time, and output it if it is a prime. This algorithm has success probability $\Omega(1/n)$ by the Prime Number Theorem, and the success probability can be amplified to be exponentially close to 1 by repeating the process poly(n) times independently, and outputting the first of these poly(n) numbers that is verified to be prime, assuming that there is at least one.

Gat and Goldwasser [GG11] asked whether it is possible to generate primes efficiently by a randomized process, such that the output is essentially *independent* of the randomness of the algorithm. In other words, is there a polynomial-time randomized algorithm, which on input 1^n, constructs a *canonical* prime of length n with high probability? They call such an algorithm a *pseudodeterministic* algorithm, since the output of the algorithm is (almost) deterministic even though the algorithm might use random bits in its operation. Note that the randomized algorithm for prime generation we described in the previous paragraph is very far from being pseudodeterministic, as different runs of the algorithm are unlikely to produce the same prime. It is easy to see that a pseudodeterministic construction serves as an intermediate notion between a randomized construction (which is trivial for primes) and a deterministic construction (where little progress has been made so far).

[GG11] initiate a general theory of pseudodeterminism for search problems, motivated by applications in cryptography and distributed computing. Since then, there have been a number of papers on pseudodeterminism, in various contexts, such as query complexity [GGR13, GIPS21, CDM23], streaming algorithms [GGMW20, BKKS23], parallel computation [GG17, GG21], learning algorithms [OS18], Kolmogorov complexity [Oli19, LOS21], space-bounded computation [GL19], proof systems [GGH18, GGH19], number theory and computational algebra [Gro15, OS17], approximation algorithms [DPV18], and many other settings (see, *e.g.*, [BB18, Gol19, DPV21, DPWV22, WDP+22, CPW23]).

Despite all this progress, the main problem about pseudodeterminism posed in [GG11] has

---

[1]Recall that a positive integer q is an n-bit prime if q is a prime number and $2^{n-1} \leq q \leq 2^n - 1$.

remained open: Is there a pseudodeterministic polynomial-time algorithm for prime construction? They describe this problem as "the most intriguing" and "perhaps the most compelling challenge for finding a unique output".

Unlike in the case of deterministic construction, number-theoretic techniques have so far not proven useful for the pseudodeterministic construction problem for primes. Using complexity-theoretic techniques, Oliveira and Santhanam [OS17] (see also [LOS21]) showed that for any $\varepsilon > 0$, there is an algorithm that runs in time 2^{n^ε} and succeeds on infinitely many input lengths.

1.1 Our Results

In this paper, we design a significantly faster algorithm and provide an affirmative answer to the question posed by Gat and Goldwasser in the infinitely-often regime. Our main result can be stated in full generality as follows.

Theorem 1.1 (Infinitely-Often Polynomial-Time Pseudodeterministic Constructions). *Let $Q \subseteq \{0,1\}^*$ be a language with the following properties:*

(Density.) *there is a constant $\rho \geq 1$ such that for every $n \in \mathbb{N}_{\geq 1}$, $Q_n \triangleq Q \cap \{0,1\}^n$ satisfies $|Q_n| \geq n^{-\rho} \cdot 2^n$; and*

(Easiness.) *there is a deterministic polynomial-time algorithm A_Q that decides whether an input $x \in \{0,1\}^*$ belongs to Q.*

Then there exist a probabilistic polynomial-time algorithm B and a sequence $\{x_n\}_{n \in \mathbb{N}_{\geq 1}}$ of n-bit strings in Q such that the following conditions hold:

1. *On every input length $n \in \mathbb{N}_{\geq 1}$, $\Pr_B[B(1^n) \notin \{x_n, \bot\}] \leq 2^{-n}$.*

2. *On infinitely many input lengths $n \in \mathbb{N}_{\geq 1}$, $\Pr_B[B(1^n) = x_n] \geq 1 - 2^{-n}$.*

Interestingly, our construction is non-black-box, in the sense that changing the *code* of the algorithm A_Q deciding property Q affects the canonical output of the corresponding algorithm B. We will revisit this point when we discuss our techniques (see the remark at the end of Section 1.3.2).

Letting Q be the set of prime numbers and noticing that Q is both dense (by the Prime Number Theorem) and easy (by the AKS primality test [AKS04]), we immediately obtain the following corollary of Theorem 1.1.

Corollary 1.2 (Infinitely-Often Polynomial-Time Pseudodeterministic Construction of Primes). *There is a randomized polynomial-time algorithm B such that, for infinitely many values of n, $B(1^n)$ outputs a canonical n-bit prime p_n with high probability.*

Corollary 1.2 improves upon the subexponential-time infinitely-often pseudodeterministic construction of primes from [OS17] mentioned above. Note that the result for prime construction is a corollary of a far more general result about properties that are dense and easy. This is evidence of the surprising power of complexity theory when applied to a problem which seems to be about number theory (but where number-theoretic techniques have not so far been effective). The famous efficient primality testing algorithm of [AKS04] similarly applied complexity-theoretic derandomization ideas to solve a longstanding open problem in computational number theory, though their argument does require more information about primes.

For a string $w \in \{0,1\}^*$ and $t \colon \mathbb{N} \to \mathbb{N}$, we let $\mathsf{rK}^t(w)$ denote the length of the smallest randomized program that runs for at most $t(|w|)$ steps and outputs w with probability at least $2/3$.

4

(We refer to [LO22] for a formal definition and for an introduction to probabilistic notions of time-bounded Kolmogorov complexity.) By encoding the (constant-size) randomized polynomial-time algorithm B and each good input length n using $O(1) + \log n$ bits in total, the following result holds.

Corollary 1.3 (Infinitely Many Primes with Efficient Succinct Descriptions). *There is a constant $c \geq 1$ such that, for $t(n) = n^c$, the following holds. For every $m \geq 1$, there is $n > m$ and an n-bit prime p_n such that $\mathsf{rK}^t(p_n) \leq \log(n) + O(1)$.*

In other words, there are infinitely many primes that admit very short efficient descriptions. The bound in Corollary 1.3 improves upon the sub-polynomial bound on $\mathsf{rK}^{\mathrm{poly}}(p_n)$ from [LOS21].

In the next subsection, we describe at a high level the ideas in the proof of Theorem 1.1, and how they relate to previous work.

1.2 Proof Ideas

The proof of Theorem 1.1 relies on *uniform hardness-randomness tradeoffs* [IW01, TV07]. For concreteness, assume that $Q = \{Q_n\}_{n \in \mathbb{N}_{\geq 1}}$, with each $Q_n \subseteq \{0,1\}^n$ consisting of the set of n-bit prime numbers. Let A_Q be a deterministic polynomial-time algorithm that decides Q (*e.g.*, A_Q is the AKS primality test algorithm [AKS04]). Before we present our algorithm and the main ideas underlying our result, it is instructive to discuss the approach of [OS17], which provides a subexponential-time pseudodeterministic construction that succeeds on infinitely many input lengths.

Subexponential-time constructions [OS17]. We first recall how uniform hardness-randomness tradeoffs work. Given a presumed hard language L, a uniform hardness-randomness tradeoff for L states that either L is easy for probabilistic polynomial-time algorithms, or else we can build a *pseudorandom set* $G_n \subseteq \{0,1\}^n$ computable in subexponential time (thus also has subexponential size), which fools probabilistic polynomial-time algorithms on inputs of length n (for infinitely many n). In particular, Trevisan and Vadhan [TV07] give a uniform hardness-randomness tradeoff for a PSPACE-complete language L_{TV} they construct, which has certain special properties tailored to uniform hardness-randomness tradeoffs.[2]

The subexponential-time construction in [OS17] uses a *win-win* argument to derive an *unconditional* pseudodeterministic algorithm from the uniform hardness-randomness tradeoff of [TV07]. There are two cases: either $L_{\mathsf{TV}} \in \mathsf{BPP}$, or it is not. If the former is the case, then $\mathsf{PSPACE} \subseteq \mathsf{BPP}$ by the PSPACE-completeness of L_{TV}. Now, since we can in *polynomial space* test all n-bit numbers using A_Q until we find the lexicographic first prime number, we can also do it in *randomized polynomial time*, *i.e.*, there is a randomized algorithm $B(1^n)$ that runs in polynomial time and outputs the lexicographically first n-bit prime with high probability. Thus, in this case, the lexicographically first n-bit prime is the "canonical" output of the pseudodeterministic algorithm, and the algorithm works on *every* input length n.

Suppose, on the other hand, that $L_{\mathsf{TV}} \notin \mathsf{BPP}$. Using the uniform hardness-randomness tradeoff of [TV07], we have that for each $\varepsilon > 0$, there is a pseudorandom set $G = \{G_n\}$, where each $G_n \subseteq \{0,1\}^n$ is of size at most 2^{n^ε}, such that for infinitely many n, G_n fools the algorithm A_Q on inputs of length n. Since A_Q accepts an $\Omega(1/n)$ fraction of strings of length n by the Prime Number Theorem, we have that the fraction of strings in G_n that are prime is $\Omega(1/n)$ (by choosing the error parameter of the uniform hardness-randomness tradeoff to be small enough). In particular, there

---

[2]For the pseudorandomness experts, these special properties are *downward self-reducibility* and *random self-reducibility*.

5

must exist an element of G_n that is prime. Since G_n is computable in subexponential time, we can define a subexponential time *deterministic* algorithm that enumerates elements of G_n and tests each one for primality until it finds and outputs one that is prime. This algorithm is deterministic but it runs in subexponential time, and is only guaranteed to be correct for infinitely many n.

Thus, in either case, we have a pseudodeterministic algorithm for constructing primes that runs in subexponential time and works infinitely often. Note that we do not know a priori which of the two cases above holds, and therefore the argument is somewhat non-constructive. By exploiting further properties of the uniform hardness-randomness tradeoff, [OS17] manage to give an explicit construction algorithm that runs in subexponential time infinitely often.

Win-win arguments. The above argument gives a subexponential-time construction, but the win-win structure of the argument seems incapable of giving an optimal polynomial-time construction. Indeed, this is the case for many win-win arguments used in complexity theory:

- A win-win argument based on the Karp–Lipton theorem [KL80] gives that $\Sigma_2\mathsf{EXP}$ requires super-polynomial size Boolean circuits [Kan82], but seems incapable of giving truly exponential $(2^{\Omega(n)})$ Boolean circuit lower bounds.

- A win-win argument based on uniform hardness-randomness tradeoffs gives that either $\mathsf{E} \subseteq \mathsf{BPP}$ or BPP can be simulated infinitely often in deterministic subexponential time on average [IW01], but it remains unknown if such a tradeoff holds at the "high end", *i.e.*, whether it is the case that either E is in probabilistic subexponential-time or else BPP can be simulated infinitely often in deterministic polynomial time on average.

- A win-win argument based on the Easy Witness Lemma gives that if $\mathsf{NEXP} \subseteq \mathsf{SIZE}(\text{poly})$, then $\mathsf{NEXP} = \mathsf{MA}$ [IKW02], but it is unknown if any interesting uniform collapse follows from the simulation of NEXP by subexponential-size Boolean circuits.

In each of these cases, the win-win argument seems to have inherent limitations that prevent us from getting optimal lower bounds or tradeoffs. Indeed, a paper by Miltersen, Vinodchandran and Watanabe [MVW99] studies the "fractional exponential" lower bounds that seem to be the best provable using win-win arguments in the context of Boolean circuit lower bounds for exponential-time classes.[3]

Thus, in order to obtain a polynomial-time pseudodeterministic algorithm for primality, it seems that we need to go beyond win-win arguments. One natural idea is to apply uniform hardness-randomness tradeoffs *recursively*. However, this seems hard to do with the uniform hardness-randomness tradeoff of [TV07]. Their tradeoff applies only to the special language L_{TV}. If we argue based on the hardness or other properties of L_{TV}, then in the case where $L_{\mathsf{TV}} \in \mathsf{BPP}$, we get a pseudodeterministic polynomial-time algorithm for constructing primes, but in the case where $L_{\mathsf{TV}} \notin \mathsf{BPP}$, we get a subexponential-time constructible pseudorandom set, and it is unclear how to apply the uniform hardness-randomness tradeoff to the algorithm for constructing this set.

Recursive application of uniform hardness-randomness tradeoffs. One of our main ideas is to exploit very recent work on uniform hardness-randomness tradeoffs [CT21] which applies

---

[3]For example, a function $f : \mathbb{N} \to \mathbb{N}$ is *sub-half-exponential* if $f(f(n)^c)^c \leq O(2^n)$ for every constant c. (The exact definition of sub-half-exponential functions may be different in different papers.) Functions such as n^k and $2^{\log^k n}$ are sub-half-exponential, while $2^{\varepsilon n}$ and 2^{n^ε} are not. It is known that $\Sigma_2\mathsf{EXP}$ cannot be computed by $f(n)$-size circuits for every sub-half-exponential f, but it remains open to show that $\Sigma_2\mathsf{EXP}$ requires circuit complexity 2^{n^ε} for any constant $\varepsilon > 0$.

to *generic* computations, as long as they satisfy certain mild properties. These tradeoffs yield *hitting sets* rather than pseudorandom sets based on hardness — a hitting set $\mathsf{H} \subseteq \{0,1\}^M$ is a set that has non-empty intersection with every $Q_M \subseteq \{0,1\}^M$ that is dense (*i.e.*, accepts at least a $1/\mathrm{poly}(M)$ fraction of strings) and is efficiently computable. It turns out that for our application to pseudodeterministic algorithms, uniform hardness-randomness tradeoffs that yield hitting sets are sufficient.

Specifically, Chen and Tell [CT21] show that for any multi-output function $f\colon \{1^n\} \to \{0,1\}^n$ computed by uniform Boolean circuits of size $T = T(n)$ and depth $d = d(n)$, either there is a hitting set $\mathsf{H} \subseteq \{0,1\}^M$ computable in time $\mathrm{poly}(T)$, or $f(1^n)$ can be computed with high probability in time $(d+n) \cdot \mathrm{poly}(M)$ (which could be much less than T). Note that this tradeoff is applicable to *any* multi-output function f given bounds on its uniform circuit complexity.

Our key idea is that this more generic uniform hardness-randomness tradeoff can be applied *recursively*. Indeed, we apply it to multi-output functions which capture the very task we are trying to solve, *i.e.*, constructing a prime! In our base case, we use the function f which does a brute-force search over n-bit numbers and outputs the lexicographically first one which is prime. This function can be computed by uniform Boolean circuits of size $2^{O(n)}$ and depth $\mathrm{poly}(n)$, and hence we can apply the Chen–Tell tradeoff to it. We set $M = n^\beta$ for some large enough constant $\beta > 1$ in the tradeoff. If we have that $f(1^n)$ is computable with high probability in time $(d+n) \cdot \mathrm{poly}(M)$, then we are done, since this gives us a pseudodeterministic algorithm for primes at length n. If not, we have that there is a hitting set $\mathsf{H} \subseteq \{0,1\}^{n^\beta}$ computable in time $2^{O(n)}$. In particular, by iterating over the elements of H and outputting the first one that is prime, we gain over the naïve brute-force search algorithm, since we are now outputting a prime of length n^β in time $2^{O(n)}$. Now *this* new algorithm can be captured by a multi-output function with output length n^β to which we apply the Chen–Tell tradeoff again. In each recursive step, we either obtain a pseudodeterministic polynomial-time construction of primes, or we obtain a significantly faster deterministic construction of primes (of a larger input length). Intuitively, analyzing this process after $O(\log n)$ steps of recursion, we can hope to show that at least one of the steps leads to a polynomial-time pseudodeterministic algorithm at the input length considered at that step.

This doesn't quite work as stated because the Chen–Tell tradeoff uses the Nisan–Wigderson generator [NW94], which is not known to have optimal parameters for all levels of hardness.[4] Our recursive process explores essentially all possible levels of hardness for the uniform hardness-randomness tradeoff, since each recursive step corresponds to a different level of hardness. Using the original Chen–Tell tradeoff gives a *quasi-polynomial-time* pseudodeterministic construction, but in order to get a polynomial-time pseudodeterministic construction, we need to work harder.

Another crucial idea for us is to optimize the Chen–Tell tradeoff by using the Shaltiel–Umans generator [SU05] rather than the Nisan–Wigderson generator. This idea comes with its own implementation challenges, since the Shaltiel–Umans generator is not known to have a crucial learnability property that is required for the uniform hardness-randomness tradeoff. We sidestep this issue using a further win-win analysis, together with some other tricks; see Section 1.3.3 for details. This enables us to achieve an optimal polynomial-time pseudodeterministic construction on infinitely many input lengths, and thereby establish Theorem 1.1.[5] We note that the subexponential-time construction of [OS17] also only works for infinitely many input lengths, and it is still open even to get a subexponential-time construction that works on all input lengths.

---

[4]Informally speaking, given a "hard truth table" of length T, we want to construct a hitting set $\mathsf{H} \subseteq \{0,1\}^M$ in $\mathrm{poly}(T)$ time; however, the Nisan–Wigderson generator requires $2^{\Theta(\log^2 T/\log M)}$ time to construct.

[5]While we do not explore this direction in the current work, we believe that our improvement on the Chen-Tell tradeoff can be used to improve the tradeoff from [CRT22, Theorem 5.2 and Theorem 5.3], thus getting a better uniform hardness vs randomness connection in the low-end regime.

The intuitive description here does not address several subtleties that arise in the proof, such as maintaining the right uniformity and depth conditions when recursively applying the uniform hardness-randomness tradeoff. We refer to Section 1.3 for a more detailed discussion of such matters.

1.3 Technical Overview

As explained above, we consider a chain of $t = O(\log n)$ recursively defined (candidate) HSGs $H_0, H_1, \ldots, H_t$ operating over different input lengths. These HSGs are obtained from the recent construction of Chen and Tell [CT21], which we informally describe next. Recall that we use Q_M to denote the easy and dense property over inputs of length M.

The Chen–Tell [CT21] targeted HSG ("ideal version"). Let $c \geq 1$ be a large enough constant, and let $f \colon \{1^n\} \to \{0,1\}^n$ be a family of unary functions computed by (uniform) Boolean circuits of size $T = T(n)$ and depth $d = d(n)$. Then, for every $\log T \leq M \leq T$ there is a set $H \subseteq \{0,1\}^M$ computable in

$$\text{time } \widetilde{T} \triangleq T^c \text{ and depth } \widetilde{d} \triangleq d \cdot \log(T) + M^c$$

such that, if $Q_M \subseteq \{0,1\}^M$ *avoids* H, (*i.e.*, Q_M is dense but $Q_M \cap H = \varnothing$), then we can compute $f(1^n)$ with high probability in time $(d + n) \cdot M^c$.

In other words, if f admits *low-depth* circuits, we can construct a candidate HSG H over length-M inputs such that breaking the generator H allows us to compute $f(1^n)$ in time $\text{poly}(n, d, M)$. For $d, M \ll T$, this can be much faster than the original time T required to compute f.

The statement above differs from the results in [CT21] (stated for unary functions) in two important ways. First, the claimed upper bound on $\widetilde{T}$ (the running time of the HSG) is not obtained by [CT21] for all choices of M. Secondly, we have not formally specified the *uniformity* of the family of circuits computing f. While these are crucial points in [CT21] and when proving our result, for simplicity we will assume for now that this upper bound can be achieved and omit the discussion on uniformity.

Bootstrapping the win-win argument. We now review the idea discussed in Section 1.2, using notations that will be more convenient for the remainder of this technical overview. Fix an arbitrary $n \in \mathbb{N}_{\geq 1}$, and consider the corresponding property $Q_n \subseteq \{0,1\}^n$ decided by $A_Q(x)$ on inputs of length n. Our initial H_0 is trivial and set to $\{0,1\}^n$. (Intuitively, this corresponds to the first case of the [OS17] argument sketched above where $L_{\mathsf{TV}} \in \mathsf{BPP}$.) Consider now a "brute-force" algorithm $\mathsf{BF}(1^n)$ that computes the first $x \in H_0$ such that $A_Q(x) = 1$. We let $f(1^n) \triangleq \mathsf{BF}(1^n)$ in the Chen–Tell HSG. Note that $f(1^n)$ can be uniformly computed in time $T = 2^{O(n)}$ and depth $d = \text{poly}(n)$, since $A_Q(x)$ runs in polynomial time and all elements of H_0 can be tested in parallel. We set $M(n) \triangleq n^\beta$, where $\beta > 1$ is a large enough constant. Let $H_1 \subseteq \{0,1\}^M$ be the candidate HSG provided by Chen–Tell. Note that H_1 can be computed in time $\widetilde{T} = 2^{O(n)}$ and depth $\widetilde{d} = \text{poly}(n)$.

Next, we consider a win-win argument based on whether Q_M avoids H_1. If this is the case, then Chen–Tell guarantees that we can compute $f(1^n) = \mathsf{BF}(1^n) \in Q_n$ with high probability in time $(d + n) \cdot M^c = \text{poly}(n)$. In other words, we can pseudodeterministically produce a string in Q_n in polynomial time. On the other hand, if $H_1 \cap Q_M \neq \varnothing$, we now have a set H_1 of strings of length $M = n^\beta$ that contains a string in Q_M and that can be deterministically computed in time $2^{O(n)}$. That is, we are back to the former case, except that we can compute H_1 (a set containing at least

8

one M-bit prime) in time much faster than $2^{O(M)}$. Crucially, in contrast to the approach of [OS17], the Chen–Tell HSG does not limit us to the use of the special language L_{TV}, effectively allowing us to reapply the same argument (with a speedup) over a larger input length.

In the next subsection, we discuss the "bootstrapping" and its parameters in more detail and explain how it gives a polynomial-time pseudodeterministic construction, assuming we have the ideal version of [CT21] described above.

1.3.1 Infinitely-Often Pseudodeterministic Polynomial-Time Constructions

Let $n_0 \in \mathbb{N}$ be an "initial" input length, and $t = O(\log n_0)$ be a parameter. For each $1 \le i \le t$, we define the i-th input length to be $n_i \triangleq n_{i-1}^{\beta}$, for a large enough constant $\beta > 1$. Our goal is to design a pseudodeterministic algorithm for finding elements in Q that will be correct on *at least one of the input lengths* $n_0, n_1, \ldots, n_t$. On each input length n_i we will have:

1. the property Q_{n_i} that we want to hit;

2. a candidate hitting set generator $\mathsf{H}_i \subseteq \{0,1\}^{n_i}$; and

3. the brute-force algorithm $\mathsf{BF}_i : \{1^{n_i}\} \to \{0,1\}^{n_i}$, which iterates through all elements in H_i and outputs the first element that is in Q_{n_i}.

Note that BF_i is completely defined by H_i. Suppose that H_i can be computed (deterministically) in time T_i and depth d_i, then BF_i can also be computed (deterministically) in time $T_i' \triangleq T_i \cdot \mathrm{poly}(n_i)$ and depth $d_i' \triangleq d_i \cdot \mathrm{poly}(n_i)$. As discussed above, initially, $\mathsf{H}_0 \triangleq \{0,1\}^{n_0}$ is the trivial hitting set generator, $T_0 \triangleq 2^{O(n_0)}$, and $d_0 \triangleq \mathrm{poly}(n_0)$.

For each $0 \le i < t$, we let $f(1^{n_i}) \triangleq \mathsf{BF}_i$, $M \triangleq n_{i+1}$, and invoke the Chen–Tell HSG to obtain the HSG $\mathsf{H}_{i+1} \subseteq \{0,1\}^{n_{i+1}}$. Recall that Chen–Tell guarantees the following: Suppose that $Q_M = Q_{n_{i+1}}$ avoids the HSG H_{i+1}, then one can use $Q_{n_{i+1}}$ to compute $f(1^{n_i})$ with high probability in time $\mathrm{poly}(d_i', n_i, M) \le \mathrm{poly}(d_i, n_i)$, by our choice of parameters. Recall that if H_i indeed hits Q_{n_i}, then $f(1^{n_i})$ implements the brute-force algorithm and outputs the first element in $\mathsf{H}_i \cap Q_{n_i}$ (*i.e.*, a *canonical* element in Q_{n_i}). To reiterate, Chen–Tell gives us the following win-win condition:

- *either* $Q_{n_{i+1}}$ avoids H_{i+1}, in which case we obtain a probabilistic algorithm that outputs a canonical element in Q_{n_i} (thus a pseudodeterministic algorithm) in $\mathrm{poly}(d_i, n_i)$ time;

- *or* H_{i+1} hits $Q_{n_{i+1}}$, in which case we obtain a hitting set H_{i+1} that hits $Q_{n_{i+1}}$, thereby making progress on input length n_{i+1}.

The HSG H_{i+1} can be computed in time $T_{i+1} \triangleq (T_i')^c$ and depth $d_{i+1} \triangleq d_i' \cdot \log T_i' + n_{i+1}^c$. Crucially, although T_0 is exponential in n_0, it is possible to show by picking a large enough $\beta > 1$ that the sequence $\{n_i\}_{i \in \mathbb{N}}$ grows faster than the sequence $\{T_i\}_{i \in \mathbb{N}}$, and eventually when $i = t = O(\log n_0)$, it will be the case that $T_t \le \mathrm{poly}(n_t)$ and we can apply the brute-force algorithm to find the first element in H_t that is in Q_{n_t} in time polynomial in n_t.

A more precise treatment of the growth of the two sequences $\{n_i\}$ and $\{T_i\}$ are as follows. There is some absolute constant $\alpha \ge 1$ such that $T_0 \le 2^{\alpha n_0}$ and

$$T_{i+1} \le T_i^{\alpha} \text{ (for each } 0 \le i < t).$$

We set $\beta \triangleq 2\alpha$ (recall that each $n_{i+1} = n_i^{\beta}$). It follows from induction that for each $0 \le i \le t$,

$$T_{i+1} \le T_0^{\alpha^i} = 2^{\alpha^{i+1}n_0} \text{ and } n_{i+1} = n_i^{\beta} = n_0^{\beta^{i+1}} = n_0^{(2\alpha)^{i+1}}.$$

9

Since
$$\frac{\log T_t}{\log n_t} \leq \frac{\alpha^t n_0}{(2\alpha)^t \log n_0} = \frac{n_0}{2^t \log n_0},$$
it follows that when $t \approx \log(n_0/\log n_0)$, T_t will be comparable to n_t (rather than 2^{n_t}). Similarly, one can show that $d_i \leq \text{poly}(n_i)$ for every $i \leq t$.

Informal description of the algorithm and correctness. To wrap up, we arrive at the following pseudodeterministic algorithm that is correct on at least one of the input lengths $n_0, n_1, \ldots, n_t$. On input length n_i, if $i = t$, then we use $\text{poly}(T_t) \leq \text{poly}(n_t)$ time to find the first string in H_i that is also in Q_{n_i} (*i.e.*, simulate BF_i); otherwise, use $Q_{n_{i+1}}$ as a distinguisher for the Chen–Tell hitting set H_i and print the output of BF_i in $\text{poly}(n_i, d_i) \leq \text{poly}(n_i)$ time. To see that our algorithm succeeds on at least one n_i, consider the following two cases:

1. Suppose that H_t indeed hits Q_{n_t}. Then clearly, our algorithm succeeds on input length n_t.

2. On the other hand, suppose that H_t does not hit Q_{n_t}. Since our trivial HSG H_0 hits Q_{n_0}, there exists an index $0 \leq i < t$ such that H_i hits Q_{n_i} but $Q_{n_{i+1}}$ avoids H_{i+1}.

   Since $Q_{n_{i+1}}$ avoids H_{i+1}, Chen–Tell guarantees that we can speed up the computation of BF_i using $Q_{n_{i+1}}$ as an oracle. Since H_i hits Q_{n_i}, the output of BF_i is indeed a canonical element in Q_{n_i}. It follows that our algorithm succeeds on input length n_i.

This completes the sketch of the algorithm and its correctness. We note that while this exposition explains how the second bullet of Theorem 1.1 is achieved, it does not address the behavior of the algorithm on other input lengths (*i.e.*, the first bullet in the same statement). For simplicity, we omit this here and refer to the formal presentation in Section 3.[6]

While the aforementioned construction conveys the gist of our approach, there are two important issues with our presentation. Firstly, as explained before, the results of [CT21] do not achieve the *ideal parameters* of the HSG stated above. Secondly, we have only vaguely discussed the *circuit uniformity* of the function $f(1^n)$. The uniformity of f is critical for the reconstruction procedure of [CT21] to run in time comparable to the circuit depth of f. On the other hand, since our HSGs and functions f (corresponding to the algorithm BF) are recursively defined, the circuit uniformity of the [CT21] generator itself becomes another critical complexity measure in the proof.

In the next subsection, we discuss the Chen–Tell generator in more detail and explain how to obtain an improved generator construction satisfying our requirements.

1.3.2 Improving the Chen–Tell Targeted Hitting Set Generator

The uniform hardness-to-randomness framework of Chen–Tell builds on two important ingredients:[7]

1. A *layered-polynomial representation* of a shallow uniform circuit.

2. A hitting set generator with a *uniform learning reconstruction* algorithm.

---

[6] Alternatively, the guarantee from the first bullet of Theorem 1.1 can always be achieved via a general argument. We refer to [OS17, Proposition 2] for the details.

[7] Below we will focus on the high-level picture of the Chen–Tell framework without diving into too many details. Our presentation is also somewhat different from the original presentation in [CT21].

Layered-polynomial representation. We now discuss the first ingredient. Let $f\colon \{0,1\}^n \to \{0,1\}^n$ be a logspace-uniform circuit family of size $T(n)$ and depth $d(n)$.[8] Let $M\colon \mathbb{N} \to \mathbb{N}$ be the parameter for output length. Building on the doubly efficient interactive proof system by [GKR15] (and its subsequent simplification by [Gol17]), for any $z \in \{0,1\}^n$, [CT21] showed that there is a sequence of polynomials $\{P_i^z\}_{i\in[d']}$ for $d' = d \cdot \mathrm{polylog}(T)$ with the following nice properties:

- (**Arithmetic setting.**) Let $\mathbb{F}$ be a finite field of size M^c for a large universal constant $c > 1$, and let m be of order $\frac{\log T}{\log M}$. All the P_i^z map $\mathbb{F}^m$ to $\mathbb{F}$ and have total degree at most M.

- (**Base case.**) There is an algorithm Base such that, given the input $z \in \{0,1\}^n$ and $\vec{w} \in \mathbb{F}^m$, computes $P_1^z(\vec{w})$ in $\mathrm{poly}(M)$ time.

- (**Downward self-reducibility.**) There is an oracle algorithm DSR that, given input $i \in \{2,\dots,d'\}$ and $\vec{w} \in \mathbb{F}^m$, together with the oracle access to $P_{i-1}^z(\cdot)$, computes $P_i^z(\vec{w})$ in $\mathrm{poly}(M)$ time.

- (**Faithful representation.**) There is an oracle algorithm OUT that, given input $i \in [n]$ and oracle access to $P_{d'}^z$, outputs $f(z)_i$ in $\mathrm{poly}(M)$ time.

Intuitively, these polynomials form an *encoded* version of the computation of f in the sense that they admit both *downward self-reducibility* and *random self-reducibility*: every P_i^z has low degree and hence admits error correction properties; downward self-reducibility follows from definition.

We note that the proof of this result depends in a crucial way on the logspace-uniformity of the circuit family computing f. (This allows one to arithmetize a formula of bounded size that computes the direct connection language of the circuit, while also controlling the circuit uniformity of the resulting polynomials.)

Hitting set generators with a uniform learning reconstruction algorithm. The second ingredient of [CT21] is the Nisan-Wigderson generator combined with Reed-Muller codes [NW94, STV01]. The most important property of this generator is that it supports a uniform learning reconstruction algorithm. In more detail, for a polynomial $P\colon \mathbb{F}^m \to \mathbb{F}$, the generator NW^P takes $s = O\left(\frac{\log^2 T}{\log M}\right)$ bits as seed, such that there is a uniform oracle algorithm R (for "reconstruction") where the following holds. Given oracle access to both P and an oracle $D\colon \{0,1\}^M \to \{0,1\}$ that distinguishes $\mathsf{NW}^P(U_s)$ from the uniform distribution, $R^{P,D}$ runs in $\mathrm{poly}(M)$ time and with high probability outputs a polynomial-size D-oracle circuit that computes P.

Now, the hitting set $H_f(z)$ is defined as

$$H_f(z) \triangleq \bigcup_{i\in[d']} \mathsf{NW}^{P_i^z}.$$

The uniform reconstruction algorithm. One key observation here is that if a distinguisher $D\colon \{0,1\}^M \to \{0,1\}$ avoids $H_f(z)$, meaning that D accepts a large fraction of inputs from $\{0,1\}^M$ but rejects all strings in $H_f(z)$, then clearly D also distinguishes all $\mathsf{NW}^{P_i^z}(U_s)$ from the uniform distribution. Following [IW01], [CT21] then shows that there is a uniform oracle algorithm R_f that takes input $z \in \{0,1\}^n$ and any "avoider" D of $H_f(z)$ as oracle, and outputs $f(z)$ with high probability. In more detail, R_f works as follows:

---

[8]Intuitively, a circuit family is logspace-uniform if each circuit in the family can be printed by a fixed machine that runs in space that is of logarithmic order in the size of the circuits. See Section 2.3 for the precise definition of logspace-uniform circuits.

1. It is given input $z \in \{0,1\}^n$ and oracle access to an avoider $D \colon \{0,1\}^M \to \{0,1\}$ of $H_f(z)$.

2. For every $i \in \{2, \ldots, d'\}$:

   (a) The goal of the i-th step is to construct a poly(M)-size D-oracle circuit C_i that computes P_i^z.

   (b) It runs the learning reconstruction algorithm $R^{P_i^z, D}$ to obtain a poly(M)-size D-oracle circuit. To answer queries to P_i^z, we first run the algorithm DSR to convert them into queries to P_{i-1}^z. Next, when $i = 2$, we answer these queries by calling Base directly, and when $i > 2$ we answer these queries by evaluating our D-oracle circuit C_{i-1}.

3. For every $i \in [n]$, output $\mathsf{OUT}^{C_{d'}^D}(i)$.

Issue with the original Chen–Tell construction: Super-logarithmic seed length of NW.
The main issue with the construction above is that $\mathsf{NW}^{P_i^z}$ has seed length $O\left(\frac{\log^2 T}{\log M}\right)$. In particular, this means that when $\log M \leq o(\log T)$, the hitting set $H_f(z)$ has super-polynomial size, and therefore cannot be computed in poly(T) time as in the "ideal version" of [CT21] stated above.[9] Hence, to improve the computation time of $H_f(z)$ to poly(T), we need an HSG with seed length $O(\log T)$ for all possible values of M, together with a uniform learning reconstruction, when it is instantiated with polynomials. Jumping ahead, we will replace NW with the Shaltiel–Umans Hitting Set Generator [SU05], obtaining an optimized version of the Chen–Tell generator with better parameters. However, the original generator from [SU05] does not provide a uniform learning reconstruction procedure. By a clever use of the classical construction of a *cryptographic pseudorandom generator from a one-way permutation* and of another idea, we managed to modify their construction to allow a uniform learning reconstruction. See the next subsection for more details.

Controlling the circuit uniformity of the optimized Chen–Tell generator. As stressed above, in order to construct a layered-polynomial representation for f with the aforementioned parameters, it is crucial that f admits a logspace-uniform circuit family. Since we will rely on multiple applications of the generator, and each new function BF on which the result is invoked contains as a subroutine the code of the previous generator, we must *upper bound the circuit uniformity* of our optimized Chen–Tell generator. This turns out to require a delicate manipulation of all circuits involved in the proof and of the Turing machines that produce them, including the components of the Shaltiel–Umans generator. For this reason, whenever we talk about a Boolean circuit in the actual proof, we also bound the description length and space complexity of its corresponding machine. Additionally, as we manipulate a super-constant number of circuits (and their corresponding machines) in our construction, we will also consider the complexity of producing the code of a machine M_2 encoding a circuit C_2 from the code of a machine M_1 encoding a circuit C_1 (see, e.g., the "Moreover" part in the statement of Theorem 3.1). The details are quite tedious, but they are necessary for verifying the correctness and running time of our algorithm. In order to provide some intuition for it, we notice that as we move from the HSG H_i to H_{i+1}, we also increase the corresponding input length parameter from n_i to $n_{i+1} = n_i^\beta$. While there is an increase in the uniformity complexity, it remains bounded relative to the new input length. We omit the details in this proof overview.

---

[9]Indeed, if we rely on the original Chen–Tell construction to implement the bootstrapping method described above, we would only obtain a quasi-polynomial-time pseudodeterministic construction, instead of a polynomial-time one.

Non-black-box behavior. We note that the recursive application of the Chen–Tell generator is responsible for the *fully non-black-box* behavior of our pseudodeterministic construction. Indeed, since we invoke the Chen–Tell generator on each function BF (which contains the code of the algorithm A_Q deciding property Q as a subroutine), the collection of strings in the hitting set generator depends on the layered-polynomial representation that is obtained from the *code* of BF. As a consequence, our construction has the unusual feature that the canonical outputs of the algorithm B in Theorem 1.1 are affected by the code of A_Q. In other words, by using a different primality test algorithm (or by making changes to the code implementing the AKS routine), one might get a different n-bit prime!

The parameters of our hitting set generator appear in Section 3. The proof of the result is given in Section 5.

1.3.3   Modified Shaltiel–Umans Generator with Uniform Learning Reconstruction

As explained above, in order to complete the proof of Theorem 1.1 we need to design a variant of the Shaltiel–Umans generator [SU05] with a *uniform learning reconstruction* procedure.

The Shaltiel–Umans generator takes as input a low-degree polynomial $P : \mathbb{F}_p^m \to \mathbb{F}_p$ (in our case p will be a power of 2) and produces a set of binary strings (which is supposed to be a hitting set). The construction of this generator also relies on "generator matrices". A matrix $A \in \mathbb{F}_p^{m \times m}$ is a *generator matrix* if it satisfies $\{A^i \cdot \vec{1}\}_{1 \leq i < p^m} = \mathbb{F}_p^m \setminus \{\vec{0}\}$. Roughly put, the matrix A can be thought of as performing multiplication with a generator of the multiplicative group of $\mathbb{F}_{p^m}$.

Recall that a generator has a uniform learning reconstruction algorithm if the following holds. Given an algorithm D that avoids the output of the generator constructed using P, as well as P itself, we can *uniformly* and *efficiently* generate (with high probability) a D-oracle circuit that computes the polynomial P. (In other words, we can query P while producing the circuit, but the circuit itself does not have access to P.)

However, the reconstruction procedure provided by the original Shaltiel–Umans generator only guarantees the following: If the generator is constructed using P and some generator matrix A, then using an algorithm D that avoids the output of the generator, and *given the matrix A* and oracle access to P, one can obtain a (D-oracle) circuit $C : [p^m - 1] \to \mathbb{F}_p^m$ such that $C(i) = P(A^i \cdot \vec{1})$.[10] (For the precise statement, see Theorem 4.9.) That is, this reconstruction is not a uniform learning algorithm in the following sense:

1. It needs to know the matrix A (which can be viewed as non-uniform advice).

2. Given oracle access to P, it only learns a circuit that computes the mapping $i \mapsto P(A^i \cdot \vec{1})$, instead of a circuit that computes $P(\vec{x})$ on a given $\vec{x} \in \mathbb{F}_p^m$.

We now describe how to modify the Shaltiel–Umans generator to make its reconstruction a uniform learning algorithm.

For the first issue, our idea is that, instead of using a generator matrix that is obtained by brute-force search as in the original construction (we note that the reconstruction cannot afford to perform the brute-force search due to its time constraints), we will use a generator matrix that is from a small set of matrices that can be constructed *efficiently*. More specifically, using results about finding primitive roots of finite fields (*e.g.*, [Sho92]), we show that one can efficiently and deterministically construct a set S of matrices that contains at least one generator matrix. The

---

[10]In fact, the circuit only computes $P(A^i \cdot \vec{v})$ for some $\vec{v}$ output by the reconstruction algorithm. We assume $\vec{v} = \vec{1}$ here for simplicity.

advantage is that the reconstruction algorithm can still afford to compute this set S. Note that although we don't know which matrix in S is a valid generator matrix (as verifying whether a matrix is a generator matrix requires too much time), we can try all the matrices from S, and one of them will be the correct one. This allows us to obtain a list of candidate circuits, one of which computes P (provided that we can also handle the second issue, which will be discussed next). Then by selecting from the list a circuit that is sufficiently close to P (note that given oracle access to P, we can easily test whether a circuit is close to P by sampling) and by using the *self-correction* property of low-degree polynomials, we can obtain a circuit that computes P exactly.

With the above idea, we may now assume that in the reconstruction we know the generator matrix A used by the Shaltiel–Umans generator. Next, we describe how to handle the second issue. Recall that the reconstruction algorithm of the Shaltiel–Umans generator gives a circuit C such that $C(i) = P(A^i \cdot \vec{1})$, for $i \in [p^m - 1]$, and we want instead a circuit that given $\vec{x} \in \mathbb{F}_p^m$ computes $P(\vec{x})$. Now suppose given $\vec{x} \in \mathbb{F}_p^m \setminus \{\vec{0}\}$, we can also *efficiently* compute the value $i \in [p^m - 1]$ such that $A^i \cdot \vec{1} = \vec{x}$. Then we would be able to combine this with C to get a circuit E that computes P, *i.e.*, if $\vec{x} = \vec{0}$ then E outputs $P(\vec{0})$ (where the value $P(\vec{0})$ can be hardcoded); otherwise, E computes i for $\vec{x}$ as described above and then outputs $C(i)$. However, the task of finding such i given A and $\vec{x}$ is essentially the *discrete logarithm problem*, for which no efficient algorithm is known!

A classical result in cryptography is that one can construct a pseudorandom generator based on the hardness of the discrete logarithm problem (see, *e.g.*, [BM84, Yao82]). More generally, given a permutation f whose inverse admits *random self-reducibility*[11], one can construct a generator G based on f so that if there is a distinguisher D that breaks G, then it can be used to invert f via a uniform reduction. Our idea is to consider the bijection $f : [p^m - 1] \rightarrow \mathbb{F}_p^m \setminus \{\vec{0}\}$ such that for each $i \in [p^m - 1]$, $f(i) = A^i \cdot \vec{1}$ (where the random self-reducibility of f^{-1} follows easily from that of the discrete logarithm problem), and try to construct a pseudorandom generator G based on f. We then combine the output of G with that of the Shaltiel–Umans generator constructed with the polynomial P and the generator matrix A. Now if there is an algorithm D that avoids this combined generator, which means D *simultaneously* avoids both the Shaltiel–Umans generator and the generator G, then D can be used to obtain

- a circuit C such that $C(i) = P(A^i \cdot \vec{1})$ for every $i \in [p^m - 1]$, and

- a circuit C' that inverts f, *i.e.*, $C'(\vec{x})$ outputs i such that $A^i \cdot \vec{1} = \vec{x}$ for every $\vec{x} \in \mathbb{F}_p^m \setminus \{\vec{0}\}$.

Then it is easy to combine C and C' to obtain a circuit that computes P.

A careful implementation of these ideas allows us to obtain a variant of the Shaltiel–Umans generator with uniform learning reconstruction, as needed in our optimized Chen–Tell generator. We refer to Theorem 4.1 in Section 4 for more details.

This completes the sketch of the proof of Theorem 1.1.

2 Preliminaries

For a positive integer k, we use $[k]$ to denote the set $\{1, 2, \ldots, \}$. We use $\mathbb{N}$ to denote all non-negative integers and $\mathbb{N}_{\geq 1}$ to denote all positive integers.

For $x, y \in \{0, 1\}^*$, we use $x \circ y$ to denote their concatenation.[12] For a function $f : \{0, 1\}^\ell \rightarrow \{0, 1\}$

---

[11]Roughly speaking, a function has random self-reducibility if computing the function on a given instance can be efficiently reduced to computing the function for uniformly random instances.

[12]We sometimes also use $C_1 \circ C_2$ to denote the composition of two circuits, but the meaning of the symbol $\circ$ will always be clear from the context.

we use $\mathtt{tt}(f)$ to denote the 2^ℓ-length truth-table of f (*i.e.*, $\mathtt{tt}(f) = f(w_1) \circ f(w_2) \circ \ldots \circ f(w_{2^\ell})$, where $w_1, \ldots, w_{2^\ell}$ is the enumeration of all strings from $\{0,1\}^\ell$ in the lexicographical order).

Unless explicitly stated otherwise, we assume that all circuits are comprised of Boolean `NAND` gates of fan-in two. In several places in the paper we will need the following notion, which strengthens the standard notion of a time-computable function by requiring the function to be computable in logarithmic space. The depth of a circuit is defined to be the maximum length (measured by the number of edges) of any input-to-output path.

Definition 2.1 (Logspace-Computable Functions). We say that a function $T \colon \mathbb{N} \to \mathbb{N}$ is logspace-computable if there exists an algorithm that gets input 1^n, runs in space $O(\log(T(n)))$, and outputs $T(n)$.

For convenience, we consider circuit families indexed by a tuple of parameters. Specifically, a circuit family with k input parameters $\vec{\ell} = (\ell_1, \ell_2, \ldots, \ell_k) \in \mathbb{N}^k$ is defined as $\{C_{\vec{\ell}}\}_{\vec{\ell} \in \mathbb{N}^k}$, where each $C_{\vec{\ell}}$ is a circuit.

2.1 Finite Fields

Throughout this paper, we will only consider finite fields of the form $\mathrm{GF}(2^{2 \cdot 3^\lambda})$ for some $\lambda \in \mathbb{N}$ since they enjoy simple representations that will be useful for us. We say $p = 2^r$ is a *nice power* of 2, if $r = 2 \cdot 3^\lambda$ for some $\lambda \in \mathbb{N}$.

Let $\ell \in \mathbb{N}$ and $n = 2 \cdot 3^\ell$. In the following, we use $\mathbb{F}$ to denote $\mathbb{F}_{2^n}$ for convenience. We will always represent $\mathbb{F}_{2^n}$ as $\mathbb{F}_2[\mathbf{x}]/(\mathbf{x}^n + \mathbf{x}^{n/2} + 1)$.[13] That is, we identify an element of $\mathbb{F}_{2^n}$ with an $\mathbb{F}_2[\mathbf{x}]$ polynomial with degree less than n. To avoid confusion, given a polynomial $P(\mathbf{x}) \in \mathbb{F}_2[\mathbf{x}]$ with degree less than n, we will use $(P(\mathbf{x}))_{\mathbb{F}}$ to denote the unique element in $\mathbb{F}$ identified with $P(\mathbf{x})$.

Let $\kappa^{(n)}$ be the natural bijection between $\{0,1\}^n$ and $\mathbb{F} = \mathrm{GF}(2^n)$: for every $a \in \{0,1\}^n$, $\kappa^{(n)}(a) = \left(\sum_{i \in [n]} a_i \cdot \mathbf{x}^{i-1} \right)_{\mathbb{F}}$. We always use $\kappa^{(n)}$ to encode elements from $\mathbb{F}$ by Boolean strings. That is, whenever we say that an algorithm takes an input from $\mathbb{F}$, we mean it takes a string $x \in \{0,1\}^n$ and interprets it as an element of $\mathbb{F}$ via $\kappa^{(n)}$. Similarly, whenever we say that an algorithm outputs an element from $\mathbb{F}$, we mean it outputs a string $\{0,1\}^n$ encoding that element via $\kappa^{(n)}$. For simplicity, sometimes we use $(a)_{\mathbb{F}}$ to denote $\kappa^{(n)}(a)$. Also, when we say the i-th element in $\mathbb{F}$, we mean the element in $\mathbb{F}$ encoded by the i-th lexicographically smallest Boolean string in $\{0,1\}^n$.

2.2 Bounded-Space Turing Machines

Our argument is robust to specific details about the computational model, but in order to estimate the relevant bounds, we must fix a model. We use the standard model of space-bounded computation (see [Gol08, Section 5] or [AB09, Section 4]). A deterministic space-bounded Turing machine has three tapes: an input tape (that is read-only); a work tape (that is read/write) and an output tape (that is write-only and uni-directional). We assume that the machine's alphabet is $\Sigma \triangleq \{0,1\}$. The space complexity of the machine is the number of used cells on the work tape. For concreteness, we assume that the work tape contains initially only $\square$ ("blank") symbols, and that the machine writes symbols from Σ in the tape.

Throughout the paper, we will describe a space-bounded Turing machine by fixing a universal Turing machine U that has an additional read-only *program tape* such that $\mathsf{TM}(x)$ is defined to be

---

[13]$\mathbf{x}^{2 \cdot 3^\ell} + \mathbf{x}^{3^\ell} + 1 \in \mathbb{F}_2[\mathbf{x}]$ is irreducible, see [VL99, Theorem 1.1.28].

the output of U with the program tape initialized as TM.[14] Abusing the notation, we often use TM to denote both the Turing machine and a binary string description of the Turing machine. Without loss of generality, we also assume our description is *paddable* meaning that for every $\mathsf{TM} \in \{0,1\}^*$ and $k \in \mathbb{N}$, TM and $\mathsf{TM} \circ 0^k$ represent the same machine. To avoid certain technicalities, we will always assume that the space bound of a Turing machine TM is greater than its description size.

Configurations of space-bounded machines. On a fixed input $x \in \{0,1\}^n$, a space-s Turing machine TM has $2^{s'}$ possible configurations, where $s' = s'(s,n) = s + O(\log s) + \log n$. Each configuration can be described by s' bits. Here, s measures the space used by the universal Turing machine U that simulates TM on input x. In more detail, it can be described by the content of U's work tape, U's current state, and the location of U's heads, including the head on the input/program tape. (Note that a configuration does not include the content of the output tape, which does not affect the next step of the machine.)

We will need the following fact for determining the relationship between configurations of a Turing machine. Recall that a sequence $\{D_n\}_{n \geq 1}$ of size-$T(n)$ computational devices is *logspace-uniform* if there is a machine $M(1^n)$ that runs in space $O(\log T(n))$ and outputs D_n (or equivalently, decides the direct connection language of D_n).

Fact 2.2. *Given a description of Turing machine $\mathsf{TM} \in \{0,1\}^*$, a space bound $s \in \mathbb{N}$, an input $x \in \{0,1\}^n$, and two configurations $\gamma, \gamma' \in \{0,1\}^{s'}$, there is an algorithm $\mathbb{A}_{\mathsf{nxt}}$ that determines whether γ' is the next configuration obtained by running TM for one step on input x. Moreover, $\mathbb{A}_{\mathsf{nxt}}$ can be computed by a logspace-uniform $O(m^3)$-size $O(\log m)$-depth formula and by an $O(m)$-space algorithm, where m is the total number of input bits. (Here, we assume that if γ is the accepting state or the rejecting state, then the next configuration of γ is always γ itself.)*

2.3 Circuits Generated by Bounded-Space Turing Machines

In this paper we often use the following two representations of a circuit (recall that throughout this paper all circuits consist entirely of fan-in two $\mathtt{NAND}$ gates).

- (**Adjacency relation tensor.**) A circuit C of size T is given as a tensor $T_C \in \{0,1\}^{T \times T \times T}$ such that for every tuple $(u,v,w) \in [T]^3$, $T_C(u,v,w) = 1$ if and only if the gates in C indexed by v and by w feed into the gate in C indexed by u.

- (**Layered adjacency relation tensor.**) A circuit C of width T and depth d is given as a list of d tensors $T_C^{(i)} \in \{0,1\}^{T \times T \times T}$, where $i \in [d]$, such that for every layer $i \in [d]$ and tuple $(u,v,w) \in [T]^3$, $T_C^{(i)}(u,v,w) = 1$ if and only if the gates in the $(i-1)$-th layer of C indexed by v and by w feed into the gate in the i-th layer of C indexed by u.

  Here, the input gates are on the 0-th layer, and the output gates are on the d-th layer. Without loss of generality we can assume all layers have exactly T gates.

In both cases above, when evaluating C in a context, we will also specify two integers n_{in} and n_{out} to denote the number of input/output gates; see the definition of $\mathsf{Circuit}[T, s, n_{\mathsf{in}}, n_{\mathsf{out}}](\mathsf{TM})$ given below for details.

While we will mostly use the (unlayered) adjacency relation tensor representation, the layered variant will be very convenient in Section 5.1.

---

[14]The advantage of fixing a universal Turing machine is that now our Turing machine always has a constant number of states, which is helpful when bounding the number of configurations of a Turing machine of super-constant size.

We define next a more general notion of a space-uniform circuit family with input parameters. This will be useful in some situations where we need to compute explicit space bounds for uniformity and index circuits by a tuple of parameters.

Definition 2.3 (α-Space-Uniform Circuits). Let $k \in \mathbb{N}$ and $\alpha, T \colon \mathbb{N}^k \to \mathbb{N}$. We say that a circuit family with k input parameters $\{C_{\vec{\ell}}\}_{\vec{\ell} \in \mathbb{N}^k}$ of size $T = T(\vec{\ell})$ is α-space-uniform if there exists an algorithm A such that:

1. (**Decides the adjacency relation.**) The algorithm gets $\vec{\ell} \in \mathbb{N}^k$ and $(u, v, w) \in \{0,1\}^{3 \log(T)}$ as input and accepts if and only if the gates in $C_{\vec{\ell}}$ indexed by v and by w feed into the gate in $C_{\vec{\ell}}$ indexed by u. (That is, the algorithm computes the adjacency relation tensor of $C_{\vec{\ell}}$.)

2. (**Runs in $\alpha(\vec{\ell})$ space.**) For input parameters $\vec{\ell} \in \mathbb{N}^k$, the algorithm runs in space $\alpha(\vec{\ell})$.

We say $\{C_{\vec{\ell}}\}_{\vec{\ell} \in \mathbb{N}^k}$ is logspace-uniform if it is $\mu \log T$-space-uniform for some constant μ.

Circuit determined by a Turing machine through the adjacency relation tensor. We will also consider the circuit determined by a Turing machine in the non-asymptotic setting. More specifically, given a Turing machine $\mathsf{TM} \in \{0,1\}^*$, parameters $T, s, n_{\mathsf{in}}, n_{\mathsf{out}} \in \mathbb{N}$, we use $\mathsf{Circuit}[T, s, n_{\mathsf{in}}, n_{\mathsf{out}}](\mathsf{TM})$ to denote the circuit whose adjacency relation is determined by running TM with space bound s over all triples $(u, v, w) \in \{0,1\}^{3 \log T}$ with $u > v > w$. The first n_{in} out of T gates are the input gates, and the last n_{out} out of T gates are the output gates. If TM fails to halt on some triples using s bits of space, or the resulting circuit is invalid (*i.e.*, inputs are not source, or outputs are not sink), we let $\mathsf{Circuit}[T, s, n_{\mathsf{in}}, n_{\mathsf{out}}](\mathsf{TM}) = \bot$.

Given two circuits $C_1 \colon \{0,1\}^{n_1} \to \{0,1\}^{n_2}$ and $C_2 \colon \{0,1\}^{n_2} \to \{0,1\}^{n_3}$, one can compose them into a single circuit $C_2 \circ C_1 \colon \{0,1\}^{n_1} \to \{0,1\}^{n_3}$ in a natural way (*i.e.*, by identifying the outputs of C_1 with the inputs of C_2). Suppose C_1 is a circuit of size T_1 and depth d_1, and C_2 is a circuit of size T_2 and depth d_2, then $C_2 \circ C_1$ has size $T_1 + T_2$ and depth $d_1 + d_2$. Also, if C_1, C_2 are given by two Turing machines TM_1 and TM_2, we can easily generate another Turing machine TM_3 that specifies $C_2 \circ C_1$. Formally, we will pick a universal machine such that we have the following simple fact on the description length of TM_3, whose proof we omit.

Fact 2.4 (Turing Machine Description of Circuit Composition). *There is a universal constant* $c_{\mathsf{comp}} \in \mathbb{N}$ *such that the following holds. Given the descriptions of Turing machines TM_1 and TM_2, parameters*

$$\vec{\ell}_1 = (T_1, s_1, n_1, n_2), \qquad \vec{\ell}_2 = (T_2, s_2, n_2, n_3) \in \mathbb{N}^4,$$

and letting

$$C_1 = \mathsf{Circuit}[\vec{\ell}_1](\mathsf{TM}_1), \ C_2 = \mathsf{Circuit}[\vec{\ell}_2](\mathsf{TM}_2), \quad and \ \ \vec{\ell}_3 = (T_1 + T_2, 2 \cdot (s_1 + s_2) + c_{\mathsf{comp}}, n_1, n_3),$$

there is a polynomial-time algorithm $\mathbb{A}_{\mathsf{comp}}$ that given $\mathsf{TM}_1, \mathsf{TM}_2, \vec{\ell}_1, \vec{\ell}_2$ as input, outputs the description of a Turing machine TM_3 such that[15]

$$(C_2 \circ C_1) = \mathsf{Circuit}[\vec{\ell}_3](\mathsf{TM}_3) \ \ and \ \ |\mathsf{TM}_3| \leq 2 \cdot (|\mathsf{TM}_1| + |\mathsf{TM}_2| + \log n_2) + c_{\mathsf{comp}}.$$

---

[15]We note that if either $C_1 = \bot$ or $C_2 = \bot$, then there is no guarantee on $\mathbb{A}_{\mathsf{comp}}$'s behavior.

2.4 Pseudorandom Generators and Hitting Set Generators

Definition 2.5 (Avoiding and Distinguishing). Let $m, t \in \mathbb{N}$, $D \colon \{0,1\}^m \to \{0,1\}$, and $Z = (z_i)_{i \in [t]}$ be a list of strings from $\{0,1\}^m$. Let $\varepsilon \in (0,1)$. We say that D ε-*distinguishes* Z, if

$$\left| \Pr_{r \leftarrow \{0,1\}^m}[D(r) = 1] - \Pr_{i \leftarrow [t]}[D(z_i) = 1] \right| \geq \varepsilon.$$

We say that D ε-*avoids* Z, if $\Pr_{r \leftarrow \{0,1\}^m}[D(r) = 1] \geq \varepsilon$ and $D(z_i) = 0$ for every $i \in [t]$.

3 Polynomial-Time Pseudodeterministic Constructions for Dense Properties

In this section, we prove our main result, restated below for convenience.

Theorem 1.1 (Infinitely-Often Polynomial-Time Pseudodeterministic Constructions). *Let* $Q \subseteq \{0,1\}^*$ *be a language with the following properties:*

(Density.) *there is a constant* $\rho \geq 1$ *such that for every* $n \in \mathbb{N}_{\geq 1}$, $Q_n \triangleq Q \cap \{0,1\}^n$ *satisfies* $|Q_n| \geq n^{-\rho} \cdot 2^n$; *and*

(Easiness.) *there is a deterministic polynomial-time algorithm* A_Q *that decides whether an input* $x \in \{0,1\}^*$ *belongs to* Q.

Then there exist a probabilistic polynomial-time algorithm B *and a sequence* $\{x_n\}_{n \in \mathbb{N}_{\geq 1}}$ *of* n-bit *strings in* Q *such that the following conditions hold:*

1. *On every input length* $n \in \mathbb{N}_{\geq 1}$, $\Pr_B[B(1^n) \notin \{x_n, \bot\}] \leq 2^{-n}$.

2. *On infinitely many input lengths* $n \in \mathbb{N}_{\geq 1}$, $\Pr_B[B(1^n) = x_n] \geq 1 - 2^{-n}$.

We will need the following theorem, which is obtained by combining [SU05] and [CT21]. The proof is presented in Section 5.

Theorem 3.1 (Improved Chen–Tell Hitting Set Generator). *There exists a universal* $c \in \mathbb{N}_{\geq 1}$, *a deterministic algorithm* H^{ct}, *and a probabilistic oracle algorithm* R^{ct} *such that the following holds. Let* $\kappa, \rho \in \mathbb{N}$. *Let* $T, d, M, n \in \mathbb{N}$ *all be sufficiently large such that* $n \leq T$, $d \leq T$, *and* $c \cdot \log T \leq M \leq T^{1/(c\rho)}$. *Denote* $\vec{\ell} \triangleq (n, T, d, M, \kappa, \rho)$ *as the input parameters.*

For a Turing machine TM *with description size* $|\mathsf{TM}| = \kappa \cdot \log T$, *we let*

$$C_{\mathsf{TM}} \triangleq \mathsf{Circuit}[T, \kappa \cdot \log T, n, n](\mathsf{TM}).$$

Assume the circuit $C_{\mathsf{TM}} \neq \bot$ *and* C_{TM} *has depth at most* d.

(Generator.) *The generator* $\mathsf{H}^{\mathsf{ct}}_{\vec{\ell}}$ *(we write* $\mathsf{H}^{\mathsf{ct}}_{\vec{\ell}}$ *to denote that* H^{ct} *takes* $\vec{\ell}$ *as input parameters) takes the description of a Turing machine* $\mathsf{TM} \in \{0,1\}^{\kappa \log T}$ *as input, and outputs a list of* M-bit *strings. We assume that the list has exactly* $T^{(c \cdot \kappa)/2}$ *entries.*

Let $\widetilde{T} \triangleq T^{c \cdot \kappa}$ *and* $\widetilde{d} \triangleq c \cdot (d \log T + \kappa^2 \log^2 T) + M^c$. *There is a Turing machine* TM_{H} *with description length* $c \log \widetilde{T}$ *such that for*

$$C_{\mathsf{H}} \triangleq \mathsf{Circuit}\left[\widetilde{T}, c \cdot \kappa \log T, n, \left(\widetilde{T}\right)^{1/2} \cdot M\right](\mathsf{TM}_{\mathsf{H}}),$$

it holds that (1) $C_{\mathsf{H}}(1^n) = \mathsf{H}^{\mathsf{ct}}_{\vec{\ell}}(\mathsf{TM})$ and (2) C_{H} has depth $\widetilde{d}$. Moreover, there is a polynomial-time[16] algorithm $\mathbb{A}^{\mathsf{ct}}$ that on inputs $\vec{\ell}$ and $\mathsf{TM} \in \{0,1\}^{\kappa \log T}$, outputs the description of TM_{H}.

(Reconstruction.) *The reconstruction algorithm R^{ct} takes the description of a Turing machine $\mathsf{TM} \in \{0,1\}^{\kappa \log T}$ as input, receives an oracle $D\colon \{0,1\}^M \to \{0,1\}$, and satisfies the following:*

> **(Soundness.)** *For every oracle $D\colon \{0,1\}^M \to \{0,1\}$, $(\mathsf{R}^{\mathsf{ct}})^D_{\vec{\ell}}(\mathsf{TM})$ runs in time $(d+n) \cdot M^{c\rho}$ and with probability at least $1 - 2^{-M}$, its output is either $C_{\mathsf{TM}}(1^n)$ or $\perp$.*

> **(Completeness.)** *If D $(1/M^\rho)$-avoids $\mathsf{H}^{\mathsf{ct}}_{\vec{\ell}}(\mathsf{TM})$, then $(\mathsf{R}^{\mathsf{ct}})^D_{\vec{\ell}}(\mathsf{TM})$ outputs $C_{\mathsf{TM}}(1^n)$ with probability at least $1 - 2^{-M}$.*

We are now ready to prove Theorem 1.1.

Proof of Theorem 1.1. We start with some notations.

Notation. Let $n_0 \in \mathbb{N}$ be sufficiently large. We define $n_0^{(0)} = n_0$, and for every $\ell \in \mathbb{N}_{\geq 1}$,

$$n_0^{(\ell)} = 2^{2^{n_0^{(\ell-1)}}}.$$

Now, fix $\ell \in \mathbb{N}$. For simplicity of notation, in the following we will use n_i, H_i, t to denote $n_i^{(\ell)}, \mathsf{H}_i^{(\ell)}, t^{(\ell)}$, which will be defined later.

Construction of hitting sets. For some parameter t that we set later, we will define a sequence of input lengths $n_1, \ldots, n_t$, with the hope that we can construct a string in Q pseudodeterministically on at least one of the input lengths.

Let $\beta \in \mathbb{N}_{\geq 1}$ be a sufficiently large constant to be chosen later. For every $i \in [t]$, we set $n_i = (n_{i-1})^\beta$. For each $i \in \{0, \ldots, t\}$, we will construct a hitting set $\mathsf{H}_i \subseteq \{0,1\}^{n_i}$, which is computable by a logspace-uniform T_i-size d_i-depth circuit. As the base case, we set H_0 as the whole set $\{0,1\}^{n_0}$. We note that there is a logspace-uniform T_0-size d_0-depth circuit that outputs all elements in H_0, where $T_0 = 2^{2n_0}$ and $d_0 = 2n_0$.

Let $\kappa \in \mathbb{N}$ be a large enough constant to be specified later. Let c be the universal constant from Theorem 3.1.

Informal description. We will first give a somewhat informal description of the construction of the H_i, in particular, we will omit details about the uniformity of the circuits (whose analysis is rather tedious). We hope this can help the reader to gain some intuition first. Later we will carefully analyze the uniformity of the circuits for H_i.

For each $i \in [t]$, we construct H_i as follows:

1. We define BF_{i-1} as the circuit implementing the following algorithm: Enumerate every element in $\mathsf{H}_{i-1} \subseteq \{0,1\}^{n_{i-1}}$, and output the first element that is in $Q_{n_{i-1}}$; if no such element exists, then $\mathsf{BF}_{i-1}(n)$ outputs $\perp$;

   Using the assumed polynomial-time algorithm A_Q for deciding membership in Q, BF_{i-1} can be implemented by a T'_{i-1}-size d'_{i-1}-depth circuit, where

   $$T'_{i-1} = T_{i-1} \cdot \mathrm{poly}(n_{i-1}) \quad \text{and} \quad d'_{i-1} = d_{i-1} + \mathrm{poly}(n_{i-1}).$$

---

[16]In this paper, whenever we say an algorithm $\mathbb{A}$ that generates Turing machines or other succinct descriptions *runs in polynomial time*, we mean the running time is polynomial in the total number of input bits. In this case, the time bound is polynomial in the description length of $\vec{\ell}$ and TM, *i.e.*, $\mathrm{poly}(\kappa \log T)$.

2. We then set H_i as the hitting set from Theorem 3.1 constructed with the Turing machine describing the circuit BF_{i-1} and output length n_i.[17] By Theorem 3.1, H_i can be implemented by a T_i-size d_i-depth circuit, where

$$T_i = \mathrm{poly}(T'_{i-1}) \quad \text{and} \quad d_i = O(d'_{i-1} \cdot \log T'_{i-1} + \log^2 T'_{i-1}) + \mathrm{poly}(n_i).$$

(Here we are being informal, see below for a more precise description.)

Formal construction. Next we carefully detail the construction. Let $\mu \in \mathbb{N}_{\geq 1}$ be a large enough constant. First, we define a Turing machine $\mathsf{TM}_{\mathsf{H}_0}$ of description size μ that describes a T_0-size d_0-depth circuit C_{H_0} for H_0 on input 1^{n_0} in $\mu \log T_0$ space. Formally

$$\mathsf{Circuit}[T_0, \mu \cdot \log T_0, n_0, \sqrt{T_0} \cdot n_0](\mathsf{TM}_{\mathsf{H}_0}) = C_{\mathsf{H}_0}.$$

Let $\tau \in \mathbb{N}$ be a large enough constant such that the running time of A_Q on n-bit inputs is bounded by $n^{\tau/3}$.

We will make sure all H_i has exactly $\sqrt{T_i}$ elements. (This is satisfied for $i = 0$ since $T_0 = 2^{2n_0}$.)

Now, for each $i \in [t]$, we will define a Turing machine $\mathsf{TM}_{\mathsf{H}_i}$ such that

$$\mathsf{Circuit}[T_i, \mu \cdot \log T_i, n_i, \sqrt{T_i} \cdot n_i](\mathsf{TM}_{\mathsf{H}_i}) = C_{\mathsf{H}_i},$$

where C_{H_i} has depth at most d_i. We will also ensure the invariance that $|\mathsf{TM}_{\mathsf{H}_i}| \leq \mu \cdot \log T_i$. By our choice of μ, the above is satisfied when $i = 0$. The machine $\mathsf{TM}_{\mathsf{H}_i}$ is defined in two steps: In the first step we define a machine $\mathsf{TM}_{\mathsf{BF}_{i-1}}$ describing the circuit BF_{i-1}, and in the second step we plug $\mathsf{TM}_{\mathsf{BF}_{i-1}}$ in Theorem 3.1 to obtain the machine $\mathsf{TM}_{\mathsf{H}_i}$.

A Turing machine $\mathsf{TM}_{\mathsf{BF}_{i-1}}$ for BF_{i-1}. We first define a Turing machine $\mathsf{TM}_{\mathsf{BF}_{i-1}}$ such that $\mathsf{TM}_{\mathsf{BF}_{i-1}}(1^{n_{i-1}})$ outputs a circuit for the algorithm BF_{i-1}. Recall that BF_{i-1} works as follows: Enumerate every element in $\mathsf{H}_{i-1} \subseteq \{0,1\}^{n_{i-1}}$ and output the first element that is in $Q_{n_{i-1}}$; if no such element exists, then $\mathsf{BF}_{i-1}(n)$ outputs $\bot$;

Using the assumed polynomial-time algorithm A_Q for deciding membership in Q, we first construct a Turing machine $\mathsf{TM}_{\mathsf{test}}$ with description size μ such that

$$C_{\mathsf{test}} = \mathsf{Circuit}\Big[T_{i-1} \cdot (n_{i-1})^{\tau/2}, \mu \cdot \log T_{i-1}, \sqrt{T_{i-1}} \cdot n_{i-1}, n_{i-1}\Big](\mathsf{TM}_{\mathsf{test}})$$

has depth $(n_{i-1})^{\tau/2}$, takes a list of $(T_{i-1})^{1/2}$ strings from $\{0,1\}^{n_{i-1}}$, and outputs the lexicographically first one in $Q_{n_{i-1}}$ (if no such string exists, outputs $\bot$ instead).

Applying Fact 2.4 to compose $C_{\mathsf{H}_{i-1}}$ and C_{test}, we obtain the desired Turing machine $\mathsf{TM}_{\mathsf{BF}_{i-1}}$ that constructs a circuit $C_{\mathsf{BF}_{i-1}}$ computing BF_{i-1}. Noting that μ is sufficiently large, we have that $\mathsf{TM}_{\mathsf{BF}_{i-1}}$ takes

$$2 \cdot \left(\left|\mathsf{TM}_{\mathsf{H}_{i-1}}\right| + \mu + \log n_{i-1} + \log T_{i-1}\right) \leq 3\mu \cdot \log T_{i-1}$$

bits to describe and uses

$$2 \cdot (\mu \cdot \log T_{i-1} + \mu \cdot \log T_{i-1} + \log T_{i-1}) + \mu \leq 5\mu \cdot \log T_{i-1}$$

space. We now set $T'_{i-1} = T_{i-1} \cdot n^\tau_{i-1}$ and $d'_{i-1} = d_{i-1} + n^\tau_{i-1}$, and we have

$$\mathsf{Circuit}\Big[T'_{i-1}, 5\mu \cdot \log T_{i-1}, n_{i-1}, n_{i-1}\Big](\mathsf{TM}_{\mathsf{BF}_{i-1}}) = C_{\mathsf{BF}_{i-1}},$$

where $C_{\mathsf{BF}_{i-1}}$ has depth at most d'_{i-1}.

---

[17]We do not discuss how to construct the Turing machine here, the details can be found in the formal construction below.

The Turing machine $\mathsf{TM}_{\mathsf{H}_i}$ for H_i. Recall that H_i is defined as the hitting set H^{ct} of Theorem 3.1 constructed with the circuit BF_{i-1} and output length n_i in the informal argument. We now formally define H_i as the hitting set

$$\mathsf{H}^{\mathsf{ct}}_{n_{i-1}, T'_{i-1}, d'_{i-1}, n_i, \kappa, \rho}\big(\mathsf{TM}_{\mathsf{BF}_{i-1}}\big).$$

To apply Theorem 3.1, we first need to ensure that

$$5\mu \cdot \log T_{i-1} \leq \kappa \log T'_{i-1},$$

which is satisfied by setting $\kappa \geq 5\mu$. We also need to ensure that

$$n_{i-1} \leq T'_{i-1}, \quad d'_{i-1} \leq T'_{i-1}, \quad \text{and} \quad c \cdot \log T'_{i-1} \leq n_i \leq (T'_{i-1})^{1/(c\rho)}. \tag{1}$$

By Theorem 3.1, we know that

$$\mathsf{TM}_{\mathsf{H}_i} = \mathbb{A}^{\mathsf{ct}}_{n_{i-1}, T'_{i-1}, d'_{i-1}, n_i, \kappa, \rho}\big(\mathsf{TM}_{\mathsf{BF}_{i-1}}\big)$$

describes a T_i-size, d_i-depth circuit C_{H_i} such that $C_{\mathsf{H}_i}(1^{n_{i-1}})$ computes H_i. Moreover, $\mathsf{TM}_{\mathsf{H}_i}$ takes $c \cdot \kappa \cdot \log T'_{i-1} \leq \mu \cdot \log T_i$ space and $c \cdot \log T_i$ bits to describe, where

$$T_i = (T'_{i-1})^{c \cdot \kappa} \quad \text{and} \quad d_i = c \cdot (d'_{i-1} \log T'_{i-1} + \kappa^2 \cdot \log^2 T'_{i-1}) + n_i^c.$$

Formally, we have

$$C_{\mathsf{H}_i} = \mathsf{Circuit}[T_i, \mu \cdot \log T_i, n_i, \sqrt{T_i} \cdot n_i](\mathsf{TM}_{\mathsf{H}_i})$$

as desired. Our invariance on $|\mathsf{TM}_{\mathsf{H}_i}|$ is satisfied by setting $\mu > c$.

Analysis of T_i and d_i and justification of (1). We set t to be the first integer such that

$$n_{t+1} > T_t^{1/(c\rho)}.$$

In the following we first show that $t \leq \log n_0$.

We first analyze the growth of T_i and T'_i. For every $i < t$, by our choice of t, we have that $n_i < n_{i+1} \leq T_i^{1/(c\rho)} < T_i$ and hence $T'_i = T_i \cdot n_i^\tau \leq T_i^{\tau+1}$. Then, from $T_{i+1} = (T'_i)^{c \cdot \kappa}$, we have $T_{i+1} \leq T_i^{c \cdot (\tau+1) \cdot \kappa}$ and consequently $\log T_{i+1} \leq c \cdot (\tau+1) \cdot \kappa \cdot \log T_i$. Letting $\lambda = c \cdot (\tau+1) \cdot \kappa$, we have

$$\log T_i \leq \lambda^i \cdot \log T_0 = \lambda^i \cdot 2n_0$$

for every $i \leq t$.

Recall that $n_i = n_{i-1}^\beta$, we have $\log n_i = \beta^i \cdot \log n_0$. For $T_t < n_t$ to hold, we only need to ensure the following:

$$\lambda^i \cdot 2n_0 < \beta^i \cdot \log n_0$$
$$\iff 2n_0 / \log n_0 < (\beta/\lambda)^i.$$

Now we will set $\beta \geq 100\lambda$. Let $\bar{t} \leq \log n_0$ be the first integer satisfying the above. We claim that $t \leq \bar{t}$. Since otherwise $\bar{t} < t$, and we would have $n_{\bar{t}} > T_{\bar{t}}$ (which certainly implies $n_{\bar{t}+1} > T_{\bar{t}}^{1/(c\rho)}$) by our choice of $\bar{t}$. This contradicts our choice of t. Therefore, we have established that $t \leq \log n_0$.

Now we turn to analyze d_i for $i \leq t$. Note that $d_0 = 2n_0$, and for $i \geq 1$, we have

$$d_i = O\big((d_{i-1} + n_{i-1}^\tau) \cdot \log T'_{i-1} + \log^2 T'_{i-1}\big) + n_i^c.$$

We will show that for every $i < t$, $d_i \leq 2n_i^c$. Clearly this holds for $i = 0$.

Since $\log T'_{i-1} \leq \log T_{i-1} + O(\log n_{i-1}) \leq \lambda^{i-1} \cdot 2n_0 + O(\log n_{i-1}) \leq n_{i-1}$ (recall here that $n_{i-1} = (n_0)^{\beta^{i-1}}$ and $\beta = 100\lambda$), we have

$$d_i \leq O\big((n_{i-1} + n_{i-1}^\tau) \cdot n_{i-1} + n_{i-1}^2\big) + n_i^c.$$

We can set β large enough so that $d_i \leq (n_{i-1})^\beta + n_i^c \leq 2 \cdot n_i^c$. From definition, we also have $d'_i \leq 2n_i^c + n_i^\tau$ for every $i < t$.

Now we are ready to justify the conditions from (1) are satisfied for $i \in [t]$. By our choice of t and the definition of T'_{i-1}, we have $n_{i-1} \leq T_{i-1} \leq T'_{i-1}$. To see $d'_{i-1} \leq T'_{i-1}$ holds, recall that $T'_{i-1} = T_{i-1} \cdot n_{i-1}^\tau$, and we have $d'_{i-1} \leq 2n_{i-1}^c + n_{i-1}^\tau \leq T_{i-1} \cdot n_{i-1}^\tau = T'_{i-1}$ by setting $\tau > c$. We also have that $c \log T'_{i-1} = c(\log T_{i-1} + \tau \log n_{i-1}) = c(\lambda^i \cdot 2n_0 + \tau \log n_{i-1}) < n_i$ since $n_0 < (n_i)^{1/\beta}$ and $\lambda^i \leq \log_{n_0} n_i$. Finally, by our choice of t, we have $n_i \leq T_{i-1}^{1/(c\rho)} < \big(T'_{i-1}\big)^{1/(c\rho)}$.

Informal argument of the correctness. We first give a somewhat informal argument below, and then give the precise argument later.

We will argue that for every $\ell \in \mathbb{N}$, there exists an $i \in \{0, 1, \ldots, t^{(\ell)}\}$ that our polynomial-time pseudodeterministic algorithm for constructing an element from Q works on input length $n_i^{(\ell)}$.

Let $i \geq 0$ be the largest integer such that $\mathsf{H}_i \subseteq \{0,1\}^{n_i}$ is a hitting set of Q_{n_i}. (Note that such i exists, since $\mathsf{H}_0 = \{0,1\}^{n_0}$ is a hitting set of Q_{n_0}.) If $i = t$, then we can simply run BF_t to obtain an element in Q_{n_t} deterministically. Note that this takes time $\mathrm{poly}(T_t) = \mathrm{poly}(n_t)$, since by our choice of t, $T_t \leq n_t^{c \cdot \beta \cdot \rho}$.

Otherwise, we have $i < t$. In this case, we know that $Q_{n_{i+1}}$ avoids the hitting set H_{i+1} (here we use the fact that $Q_{n_{i+1}}$ accepts more than an $n_{i+1}^{-\rho}$ fraction of strings from $\{0,1\}^{n_{i+1}}$). By the reconstruction part of Theorem 3.1, there is a $\mathrm{poly}(n_{i+1}) \cdot d'_i$ randomized time algorithm that simulates BF_i with probability at least $1 - 2^{n_{i+1}}$. Since H_i is a hitting set for Q_{n_i}, this gives us a pseudodeterministic algorithm with $\mathrm{poly}(n_{i+1})$ time that finds a canonical element in Q_{n_i}. Since $n_{i+1} = \mathrm{poly}(n_i)$, our pseudodeterministic algorithm runs in polynomial time.

Formal description of the algorithm B. First, note that by our choice of t and β, it holds that $n_0^{(\ell+1)} > n_{t^{(\ell)}}^{(\ell)}$. On an input length $n \in \mathbb{N}_{\geq 1}$, our algorithm B is defined as follows:

1. Given input 1^n for $n \in \mathbb{N}_{\geq 1}$.

2. Compute the largest $\ell \in \mathbb{N}$ such that $n_0^{(\ell)} \leq n$, then compute the largest i such that $n_i^{(\ell)} \leq n$. Output $\perp$ and abort immediately if $n_i^{(\ell)} \neq n$. From now on we use n_i, T_i, d_i, etc. to denote $n_i^{(\ell)}, T_i^{(\ell)}, d_i^{(\ell)}$, etc.

3. For every $j \in \{0, 1, \ldots, i\}$, compute $T_j, T'_j, d_j, d'_j, \mathsf{TM}_{\mathsf{H}_j}, \mathsf{TM}_{\mathsf{BF}_j}$. There are two cases:

   - Case I: $n_{i+1} \leq T_i^{1/(c\rho)}$: In this case, we have that $i < t$. Run

     $$\big(\mathsf{R}^{\mathsf{ct}}\big)_{n_i, T'_i, d'_i, n_{i+1}, \kappa, \rho}^{Q_{n_{i+1}}}(\mathsf{TM}_{\mathsf{BF}_i})$$

     and set z_n be its output.

   - Case II: $n_{i+1} > T_i^{1/(c\rho)}$: In this case, we have that $t \leq i$. Compute t first (recall that t is the first integer such that $n_{t+1} > T_t^{1/(c\rho)}$). Output $\perp$ and abort immediately if $i > t$. Otherwise, construct C_{BF_i} from $\mathsf{TM}_{\mathsf{BF}_i}$ and set $z_n = C_{\mathsf{BF}_i}(1^n)$.

22

4. Output z_n if $A_Q(z_n) = 1$ and $\perp$ otherwise.

From our choice of parameters and Theorem 3.1, the algorithm B runs in $\mathrm{poly}(n)$ time.

Analysis of the algorithm B. Finally we show that the algorithm B satisfies our requirements. We call an input length $n \in \mathbb{N}_{\geq 1}$ *valid* if there exist $\ell \in \mathbb{N}$ and $i \in \{0, \dots, t^{(\ell)}\}$ such that $n = n_i^{(\ell)}$, and we call n *invalid* otherwise.[18] For every $n \in \mathbb{N}_{\geq 1}$, let y_n be the lexicographically first element in Q_n.

For every invalid $n \in \mathbb{N}_{\geq 1}$, we simply set $x_n = y_n$. For every valid $n \in \mathbb{N}_{\geq 1}$, we set x_n as follows:

$$x_n = \begin{cases} C_{\mathsf{BF}_i}(1^{n_i}), & \text{if } C_{\mathsf{BF}_i}(1^{n_i}) \in Q_{n_i}, \\ y_n, & \text{if otherwise.} \end{cases}$$

We first observe that for all invalid $n \in \mathbb{N}_{\geq 1}$, it holds that $B(1^n) = \perp$ with probability 1. Now we are ready to show that for every $n \in \mathbb{N}_{\geq 1}$, $\Pr_B[B(1^n) \notin \{x_n, \perp\}] \leq 2^{-n}$. Clearly we only need to consider valid n.

Fix a valid $n \in \mathbb{N}_{\geq 1}$. From the soundness of the reconstruction part of Theorem 3.1, it follows that $z_n \in \{C_{\mathsf{BF}_i}(1^n), \perp\}$ with probability at least $1 - 2^{-n}$ (if $i = t$, then $z_n = C_{\mathsf{BF}_i}(1^n)$ with probability 1). If $C_{\mathsf{BF}_i}(1^{n_i}) \in Q_{n_i}$, then $x_n = C_{\mathsf{BF}_i}(1^{n_i})$ and $z_n \in \{x_n, \perp\}$ with high probability; otherwise we have $z_n = \perp$. In both cases the soundness of B holds.

Next, we show that for infinitely many $n \in \mathbb{N}_{\geq 1}$, we have $\Pr_B[B(1^n) = x_n] \geq 1 - 2^{-n}$. Following the informal argument, for every $\ell \in \mathbb{N}$, let $i \geq 0$ be the largest integer such that $\mathsf{H}_i \subseteq \{0, 1\}^{n_i^{(\ell)}}$ is a hitting set of $Q_{n_i^{(\ell)}}$. Letting $n = n_i^{(\ell)}$, we will show that $B(1^n)$ outputs x_n with probability at least $1 - 2^{-n}$, which would finish the proof.

If $i = t$, since H_i is a hitting set for Q_n, it follows that $z_n = C_{\mathsf{BF}_i}(1^n) \in Q_n$, and we have $B(1^n) = x_n$ with probability 1. If $i < t$, we know that $Q_{n_{i+1}}$ $(1/n_{i+1}^\rho)$-avoids the hitting set H_{i+1}. By the completeness of the reconstruction part of Theorem 3.1, we have that $z_n = (\mathsf{R}^{\mathsf{ct}})^{Q_{n_{i+1}}}_{n_i, T_i', d_i', n_{i+1}, \kappa, \rho}(\mathsf{TM}_{\mathsf{BF}_i})$ equals $C_{\mathsf{BF}_i}(1^n)$ with probability at least $1 - 2^{-n}$. Moreover, in this case, since H_i is a hitting set of Q_n, we know $z_n \in Q_n$ and $z_n = x_n$, which completes the proof. □

Let B be the algorithm given by Theorem 1.1. We note that, by using 1 bit of advice to encode if a given input length n satisfies $\Pr_B[B(1^n) = x_n] \geq 1 - 2^{-n}$, we can obtain an efficient algorithm that outputs a canonical answer with high probability (*i.e.*, satisfies the promise of a pseudodeterministic algorithm) *on all input lengths* and is correct on infinitely many of them. We state the result below as it might be useful in future work.

Corollary 3.2 (Pseudodeterministic Polynomial-Time Construction with 1 Bit of Advice that Succeeds Infinitely Often)**.** *Let Q be a dense and easy language. There exist a polynomial-time probabilistic algorithm A and a sequence of advice bits $\{\alpha_i \in \{0, 1\}\}_{i \in \mathbb{N}_{\geq 1}}$ such that*

- *for all $n \in \mathbb{N}_{\geq 1}$, $A(1^n, \alpha_n)$ outputs a canonical $x_n \in \{0, 1\}^n$ with probability at least $1 - 2^{-n}$, and*

- *for infinitely many $n \in \mathbb{N}_{\geq 1}$, $x_n \in Q \cap \{0, 1\}^n$.*

---

[18]By our choice of parameters, such pair (ℓ, i) is unique for a valid n.

4 Modified Shaltiel–Umans Generator with Uniform Learning Reconstruction

In order to prove Theorem 3.1, we will need the following result.

Theorem 4.1 (A HSG with Uniform Learning Reconstruction)**.** *There exist an algorithm* H *and a probabilistic oracle algorithm* $\mathsf{R}^{(-)}$ *such that the following holds. Let* p *be a nice power of* 2, m *be a power of* 3, $\Delta, M \in \mathbb{N}$ *with* $p > \Delta^2 m^7 M^9$, *and let* $\vec{\ell} \triangleq (p, m, M, \Delta)$ *be the input parameters.*

- *The generator* $\mathsf{H}_{\vec{\ell}}$ *takes as input a polynomial* $P \colon \mathbb{F}_p^m \to \mathbb{F}_p$ *with total degree at most* Δ, *specified as a list of* p^m *evaluations of* P *on all points from* $\mathbb{F}_p^m$ *in the lexicographic order, and outputs a set of strings in* $\{0,1\}^M$. *Moreover,* $\mathsf{H}_{\vec{\ell}}$ *can be implemented by a logspace-uniform circuit of size* $\mathrm{poly}(p^m)$ *and depth* $\mathrm{poly}(\log p, m, M)$.

- *The reconstruction algorithm* $\mathsf{R}_{\vec{\ell}}^{D,P}$, *where* $D \colon \{0,1\}^M \to \{0,1\}$ *is any function that* $(1/M)$- *avoids* $\mathsf{H}_{\vec{\ell}}(P)$, *runs in time* $\mathrm{poly}(p, m)$ *and outputs, with probability at least* $1 - 1/p^m$, *a* D-*oracle circuit that computes* P.

The rest of this section is dedicated to the proof of Theorem 4.1.

4.1 Technical Tools

4.1.1 Error-Correcting Codes

Theorem 4.2 (List-Decoding Reed–Solomon Codes [Sud97])**.** *Let* b, a, *and* d *be integers such that* $a > \sqrt{2d \cdot b}$. *Given* b *distinct pairs* (x_i, y_i) *in a field* $\mathbb{F}$, *there are at most* $2 \cdot b/a$ *polynomials* g *of degree* d *such that* $g(x_i) = y_i$ *for at least* a *pairs. Furthermore, a list of all such polynomials can be computed in time* $\mathrm{poly}(b, \log |\mathbb{F}|)$.

In particular, if $a = \alpha \cdot b$ for some $0 < \alpha \leq 1$, provided that $\alpha > \sqrt{2d/b}$, there are at most $O(1/\alpha)$ degree-d polynomials that agree with an α-fraction of the b pairs.

4.1.2 Generator Matrices

Definition 4.3 (Generator Matrices)**.** Let p be a power of 2 and $m \in \mathbb{N}$. We say that $A \in \mathbb{F}_p^{m \times m}$ is a *generator matrix* for $\mathbb{F}_p^m$ if A is invertible, $A^{p^m-1} = I$, and $\{A^i \cdot \vec{v}\}_{1 \leq i < p^m} = \mathbb{F}_p^m \setminus \{\vec{0}\}$ for any nonzero $\vec{v} \in \mathbb{F}_p^m$.[19]

Theorem 4.4 ([Sho92])**.** *Let* $n \in \mathbb{N}$. *Given any irreducible polynomial* f *of degree* n *over* $\mathbb{F}_2$, *one can deterministically construct in time* $\mathrm{poly}(n)$ *a set* S_n *that contains at least one primitive root of the multiplicative group of* $\mathbb{F}_2[\mathbf{x}]/(f)$.

We need the following lemma to deterministically construct generator matrices. Note that it is unclear how to deterministically construct a single generator matrix. Instead, we reduce the task of constructing such matrices to the task of constructing primitive roots of $\mathbb{F}_{p^m}$. Then, we invoke Theorem 4.4 to construct a *set* of matrices that contains at least one generator matrix. It turns out that this set of matrices suffices for our purposes.

Lemma 4.5. *Let* p *be a nice power of* 2 *and* m *be a power of* 3. *One can deterministically construct in time* $\mathrm{poly}(\log p, m)$ *a set of matrices in* $\mathbb{F}_p^{m \times m}$ *that contains at least one generator matrix for* $\mathbb{F}_p^m$.

---

[19]In fact, it is not hard to see that the third condition implies the first two. We include those two conditions in this definition as they will be useful later.

Proof Sketch. Let $p = 2^{2 \cdot 3^\alpha}$ and $m = 3^\beta$, where $\alpha, \beta \in \mathbb{N}$. First recall that

$$\mathbb{F}_{p^m} = \frac{\mathbb{F}_2[\mathbf{x}]}{\left(\mathbf{x}^{2 \cdot 3^{\alpha+\beta}} + \mathbf{x}^{3^{\alpha+\beta}} + 1\right)} \quad \text{and} \quad \mathbb{F}_p = \frac{\mathbb{F}_2[\mathbf{y}]}{\left(\mathbf{y}^{2 \cdot 3^\alpha} + \mathbf{y}^{3^\alpha} + 1\right)}.$$

We view the field $\mathbb{F}_{p^m}$ as an m-dimensional vector space over $\mathbb{F}_p$ with a basis $(1, \mathbf{x}, \mathbf{x}^2, \ldots, \mathbf{x}^{3^\beta - 1})$, using the following (bijective) mapping. Let $v \in \mathbb{F}_{p^m}$, then we can write

$$v \triangleq \sum_{t=0}^{2 \cdot 3^{\alpha+\beta} - 1} \hat{v}_t \cdot \mathbf{x}^t \qquad \text{(where } \hat{v}_t \in \mathbb{F}_2)$$

$$= \sum_{i=0}^{2 \cdot 3^\alpha - 1} \sum_{j=0}^{3^\beta - 1} \hat{v}_{i,j} \cdot \mathbf{x}^{i \cdot 3^\beta + j}$$

$$= \sum_{j=0}^{3^\beta - 1} \mathbf{x}^j \cdot \left(\sum_{i=0}^{2 \cdot 3^\alpha - 1} \hat{v}_{i,j} \cdot \mathbf{x}^{i \cdot 3^\beta} \right).$$

By mapping $\mathbf{x}^{3^\beta}$ to $\mathbf{y}$, we get that for every $j \in [3^\beta - 1]$,

$$\sum_{i=0}^{2 \cdot 3^\alpha - 1} \hat{v}_{i,j} \cdot \mathbf{x}^{i \cdot 3^\beta} = \sum_{i=0}^{2 \cdot 3^\alpha - 1} \hat{v}_{i,j} \cdot \mathbf{y}^i,$$

which represents an element in $\mathbb{F}_p$, so v corresponds under the mapping to an element in the vector space $\mathbb{F}_p^m$ over $\mathbb{F}_p$.

Next, analogously to [SU05, Lemma 4.4], we observe that:

1. multiplication by a fixed element $g \in \mathbb{F}_{p^m}$ within the field corresponds to a linear transformation $A_g \in \mathbb{F}_p^{m \times m}$ within the vector space $\mathbb{F}_p^m$ (with respect to the above map and its inverse);

2. $A_g \in \mathbb{F}_p^{m \times m}$ can be obtained in time $\mathrm{poly}(\log p, m)$ given $g \in \mathbb{F}_{p^m}$;

3. if g is a primitive root of $\mathbb{F}_{p^m}$, then A_g is a generator matrix for $\mathbb{F}_p^m$.

The lemma now follows from these observations and Theorem 4.4. $\qquad \square$

4.1.3 Random Self-Reducibility for Discrete Logarithm

Lemma 4.6. *There is a probabilistic polynomial-time oracle algorithm* $\mathsf{DLCorr}^{(-)}$ *such that the following holds. Let p be a power of 2, $m \in \mathbb{N}$, $\varepsilon > 0$, A be a generator matrix for $\mathbb{F}_p^m$, and let g be any probabilistic procedure that satisfies*

$$\Pr_{\vec{v} \leftarrow \mathbb{F}_p^m \setminus \{\vec{0}\}, \, g} \left[g(\vec{v}) \text{ outputs } i \in [p^m - 1] \text{ such that } A^i \cdot \vec{1} = \vec{v} \right] \geq \varepsilon.$$

Then for every $\vec{u} \in \mathbb{F}_p^m \setminus \{\vec{0}\}$, $\mathsf{DLCorr}^g(p, m, 1^{\lceil 1/\varepsilon \rceil}, A, \vec{u})$ outputs $\ell \in [p^m - 1]$ such that $A^\ell \cdot \vec{1} = \vec{u}$ with probability at least $2/3$.

25

Proof Sketch. The algorithm is an adaptation of the worst-case to average-case reduction for the discrete logarithm problem. Given $\vec{u} \in \mathbb{F}_p^m \setminus \{\vec{0}\}$, we pick a random $j \in [p^m - 1]$ and set $\vec{v} \triangleq A^j \cdot \vec{u}$. Let $i \triangleq g(\vec{v})$. Since $\vec{v}$ is uniformly distributed, with probability at least ε we have $A^i \cdot \vec{1} = \vec{v}$. We check if this is the case in polynomial time (note that we can compute A^i in polynomial time by repeated squaring). Suppose this is indeed the case, then $A^i \cdot \vec{1} = \vec{v} = A^j \cdot \vec{u}$. Recall that A is invertible. If $i > j$, we output $\ell \triangleq i - j$. If $i = j$, we have $\vec{u} = \vec{1}$. In this case, we output $\ell \triangleq p^m - 1$. Finally, if $j > i$, we output $\ell \triangleq t - (j - i)$.

By sampling $O(1/\varepsilon)$ many values of j, with probability at least $2/3$, there is at least one invocation $i \triangleq g(\vec{v})$ such that $A^i \cdot \vec{1} = \vec{v}$ indeed holds. Therefore, the success probability of our algorithm is at least $2/3$. $\qquad\square$

4.1.4 Pseudorandom Generators from One-Way Permutations

Theorem 4.7 ([BM84, Yao82, GL89]). *There exist a deterministic oracle algorithm $\mathsf{CryptoG}^{(-)}$ and a probabilistic oracle algorithm $\mathsf{Invert}^{(-)}$ such that the following holds. Let $s, M \in \mathbb{N}$ be the input parameters, and let $f \colon \{0,1\}^s \to \{0,1\}^s$ be a permutation.*

1. *$\mathsf{CryptoG}_{s,M}^f$ outputs a set of 2^{2s} M-bit strings. Moreover, $\mathsf{CryptoG}_{s,M}^f$ can be implemented by a logspace-uniform f-oracle circuit of size $\mathrm{poly}(2^s, M)$ and depth $\mathrm{poly}(s, M)$.*

2. *$\mathsf{Invert}_{s,M}^{(-)}$ takes $x \in \{0,1\}^s$ as input and runs in $\mathrm{poly}(s, M)$ time. For any function $D \colon \{0,1\}^M \to \{0,1\}$ that ε-distinguishes $\mathsf{CryptoG}_{s,M}^f$ from $\{0,1\}^M$, we have*

$$\Pr_{x \leftarrow \{0,1\}^s}\left[\mathsf{Invert}_{s,M}^{f,D}(x) = f^{-1}(x)\right] \geq \frac{\varepsilon}{\mathrm{poly}(M)}.$$

Proof Sketch. The generator $\mathsf{CryptoG}^{(-)}$ follows from the well-known construction of pseudorandom generators from one-way permutations using the Goldreich–Levin Theorem [GL89]. More specifically,

$$\mathsf{CryptoG}_{s,M}^f \triangleq \bigcup_{x,r \in \{0,1\}^s} \left(\langle x, r \rangle, \langle f(x), r \rangle, \langle f(f(x)), r \rangle, \dots, \left\langle f^{(M-1)}(x), r \right\rangle \right),$$

where $\langle \cdot \rangle$ denotes the inner product mod 2 function and $f^{(i)}$ denotes the composition of f with itself i times.

The "inverting" algorithm $\mathsf{Invert}^{(-)}$ and its correctness rely on standard techniques in pseudorandomness such as the hybrid argument, Yao's theorem on the equivalence between pseudorandomness and unpredictability [Yao82], and the Goldreich–Levin decoding algorithm [GL89]. (See *e.g.*, [AB09, Section 9.3].)

Finally, to see that $\mathsf{CryptoG}_{s,M}^f$ can be implemented by a logspace-uniform f-oracle circuit of size $\mathrm{poly}(2^s, M)$ and depth $\mathrm{poly}(M)$, we first note that there is a Turing machine that given $s, M \in \mathbb{N}$ and $x, r \in \{0,1\}^s$, computes the M-bit string $\langle x, r \rangle, \langle f(x), r \rangle, \langle f(f(x)), r \rangle, \dots, \left\langle f^{M-1}(x), r \right\rangle$ in $\mathrm{poly}(s, M)$ time using f as an oracle. Then by the fact that any time-t Turing machine can be simulated by a logspace-uniform circuit of size $O(t^2)$, computing a single M-bit string in $\mathsf{CryptoG}_{s,M}^f$ can be done using a logspace-uniform f-oracle circuit of size $\mathrm{poly}(s, M)$. The desired conclusion follows from the observation that we can compute these 2^{2s} M-bit strings in parallel. $\qquad\square$

4.1.5 Self-Correction for Polynomials

Theorem 4.8 (A Self-Corrector for Polynomials, cf. [GS92, Sud95])**.** *There is a probabilistic oracle algorithm* PCorr$^{(-)}$ *such that the following holds. Let p be a power of 2, $m, \Delta \in \mathbb{N}$ such that $\Delta < p/3$. Let $g \colon \mathbb{F}_p^m \to \mathbb{F}_p$ be such that there exists a polynomial P of total degree at most Δ for which*

$$\Pr_{\vec{x} \leftarrow \mathbb{F}_p^m} [g(\vec{x}) \neq P(\vec{x})] \leq 1/4.$$

Then for all $\vec{x} \in \mathbb{F}_p^m$, PCorr$^g(p, m, \Delta, \vec{x})$ *runs in time* $\mathrm{poly}(\Delta, \log p, m)$ *and outputs $P(\vec{x})$ with probability at least $2/3$.*

4.2 The Shaltiel–Umans Generator

We state a version of the hitting set generator constructed by Shaltiel and Umans [SU05] that will be convenient for our purposes.

Theorem 4.9 (Implicit in [SU05])**.** *There exist a deterministic algorithm* HSU *and a probabilistic oracle algorithm* RSU$^{(-)}$ *such that the following holds. Let p be a power of 2, $m, M, \Delta \in \mathbb{N}$ with $p > \Delta^2 m^7 M^9$, $\vec{\ell} \triangleq (p, m, M, \Delta)$ be the input parameters, and let*

- *$P \colon \mathbb{F}_p^m \to \mathbb{F}_p$ be a polynomial with total degree at most Δ, given as a list of p^m evaluations of P on all points from $\mathbb{F}_p^m$ in lexicographic order, and*

- *A be a generator matrix for $\mathbb{F}_p^m$.*

Then

1. *The generator* HSU$_{\vec{\ell}}(P, A)$ *outputs a set of strings in $\{0, 1\}^M$. Moreover,* HSU$_{\vec{\ell}}$ *can be implemented by a logspace-uniform circuit of size $\mathrm{poly}(p^m)$ and depth $\mathrm{poly}(\log p, m)$.*

2. *The reconstruction algorithm* RSU$_{\vec{\ell}}^{D,P}(A)$*, where $D \colon \{0,1\}^M \to \{0,1\}$ is any function that $(1/M)$-avoids* HSU$_{\vec{\ell}}(P, A)$*, runs in $\mathrm{poly}(p, m)$ time and outputs, with probability at least $1 - 1/p^{2m}$, a vector $\vec{v} \in \mathbb{F}_p^m \setminus \{\vec{0}\}$ and a D-oracle circuit $C \colon [p^m - 1] \to \mathbb{F}_p$ such that*

$$C(i) = P(A^i \cdot \vec{v}) \text{ for every } i \in [p^m - 1].$$

The statement of Theorem 4.9 and the HSG result of [SU05] differ in two aspects:

- First, we use a *polynomial* instead of a Boolean function to construct the HSG, which fits more naturally into the framework of Chen–Tell [CT21] (see also Section 5).

- Second, we explicitly calculated a circuit depth upper bound for computing the HSG, which is not stated in [SU05].

Nevertheless, Theorem 4.9 easily follows from the arguments in [SU05]. For completeness, we review the construction of [SU05] and present a proof sketch of Theorem 4.9 in this subsection.

The generator. We first construct m candidate "p-ary PRGs" $G_{p\text{-ary}}^{(0)}, G_{p\text{-ary}}^{(1)}, \cdots, G_{p\text{-ary}}^{(m-1)} : \mathbb{F}_p^m \to \mathbb{F}_p^M$; note that the inputs and outputs of these "p-ary PRGs" are elements in $\mathbb{F}_p$. In particular:

$$G_{p\text{-ary}}^{(j)}(\vec{x}) = \Big(P(A^{p^j \cdot 1} \vec{x}), P(A^{p^j \cdot 2} \vec{x}), \cdots, P(A^{p^j \cdot M} \vec{x}) \Big).$$

Then we convert each p-ary PRG into a (usual binary) PRG by invoking [SU05, Lemma 5.6]. More precisely, for each $0 \le j < m$, we interpret $G_{p\text{-ary}}^{(j)}$ as a PRG that takes a binary seed of length $m \log p$ and outputs M elements in $\{0,1\}^{\log p}$, using the canonical bijection $\kappa^{(\log p)}$ between $\mathbb{F}_p$ and $\{0,1\}^{\log p}$. Then, for $G_{p\text{-ary}}^{(j)} : \{0,1\}^{m \log p} \to (\{0,1\}^{\log p})^M$, given seeds $x \in \{0,1\}^{m \log p}$ and $r \in \{0,1\}^{\log p}$, we define

$$G^{(j)}(x,r) = \Big(\langle G_{p\text{-ary}}^{(j)}(x)_1, r \rangle, \langle G_{p\text{-ary}}^{(j)}(x)_2, r \rangle, \ldots, \langle G_{p\text{-ary}}^{(j)}(x)_M, r \rangle \Big).$$

Here, $\langle \cdot \rangle$ denotes the inner product mod 2 function. In other words, we combine $G_{p\text{-ary}}^{(j)}$ with the *Hadamard code* to obtain a Boolean PRG $G^{(j)} : \{0,1\}^{m \log p + \log p} \to \{0,1\}^M$.

Finally, our HSG will be the union of all PRGs $G^{(j)}$. That is, our algorithm $\mathsf{HSU}_{\vec{\ell}}(P, A)$ enumerates every $0 \le j < m$, $x \in \{0,1\}^{m \log p}$, $r \in \{0,1\}^{\log p}$, and prints the string $G^{(j)}(x,r)$.

To see that $\mathsf{HSU}_{\vec{\ell}}$ can be computed by a logspace-uniform low-depth circuit, we argue that given appropriate indexes j and i, the i-th bit of $G^{(j)}(x,r)$ can be computed by a logspace-uniform low-depth circuit. The bit we want to compute is

$$G^{(j)}(x,r)_i = \langle G_{p\text{-ary}}^{(j)}(x)_i, r \rangle = \langle P(A^{p^j \cdot i} \vec{x}), r \rangle,$$

where $\vec{x}$ is the vector in $\mathbb{F}_p^m$ encoded by x. By repeated squaring, we can output a (logspace-uniform) circuit of size and depth $\mathrm{poly}(\log p, m)$ that computes $A^{p^j \cdot i}$. Multiplying $A^{p^j \cdot i}$ with $\vec{x}$, indexing (*i.e.*, finding the $(A^{p^j \cdot i} \vec{x})$-th entry of P), and computing Boolean inner product have logspace-uniform circuits of size $\mathrm{poly}(M, p^m) = \mathrm{poly}(p^m)$ and depth $\mathrm{poly}(m, \log p, \log M) = \mathrm{poly}(\log p, m)$. Since we need to output $m \cdot p^{m+1}$ strings of length M and each output bit can be computed by a logspace-uniform circuit of size $\mathrm{poly}(p^m)$ and depth $\mathrm{poly}(\log p, m)$, the complexity upper bounds for computing $\mathsf{HSU}_{\vec{\ell}}$ follows.

Now we consider the reconstruction algorithm. Suppose there is an adversary $D : \{0,1\}^M \to \{0,1\}$ that $(1/M)$-avoids $\mathsf{HSU}_{\vec{\ell}}(P, A)$. It follows that D $(1/M)$-distinguishes every binary PRG $G^{(j)}$.

From distinguishers to next-element predictors. For each $0 \le j < m$, we use D to build a "next-element predictor" $D^{(j)}$ for $G_{p\text{-ary}}^{(j)}$. Since D $(1/M)$-distinguishes $G^{(j)}$, it can be used to build a next-*bit* predictor $D_{\mathsf{bit}}^{(j)}$ such that

$$\Pr_{i \leftarrow [M], x \leftarrow \{0,1\}^{m \log p}, r \leftarrow \{0,1\}^{\log p}} \Big[D_{\mathsf{bit}}^{(j)} \Big(G^{(j)}(x,r)_1, \ldots, G^{(j)}(x,r)_{i-1} \Big) = G^{(j)}(x,r)_i \Big] \ge 1/2 + 1/M^2.$$

Therefore, with probability $\ge 1/2M^2$ over $i \leftarrow [M]$ and $x \leftarrow \{0,1\}^{m \log p}$, the probability over $r \leftarrow \{0,1\}^{\log p}$ that $D_{\mathsf{bit}}^{(j)}$ predicts the i-th bit of $G^{(j)}(x,r)$ given its first $i-1$ bits correctly is at least $1/2 + 1/2M^2$. In this case, using the list-decoding algorithm for Hadamard code [GL89], we can find a list of $O(M^4)$ elements that contains $G_{p\text{-ary}}^{(j)}(x)_i$. (In fact, the trivial list-decoding algorithm suffices here, since it runs in time $\mathrm{poly}(p)$.) We call this procedure the *next-element predictor* $D^{(j)}$; it takes as input

$$u_{M-1} \triangleq P(A^{-(M-1)p^j} \vec{x}), u_{M-2} \triangleq P(A^{-(M-2)p^j} \vec{x}), \ldots, u_1 \triangleq P(A^{-p^j} \vec{x}),$$

28

where $\vec{x} \leftarrow \mathbb{F}_p^m$ is a random vector. It randomly selects $i \leftarrow [M]$, invokes $D_{\mathsf{bit}}^{(j)}$ and the list-decoding algorithm for the Hadamard code, and outputs a list of $O(M^4)$ elements. With probability $\Omega(1/M^2)$ over $\vec{x} \leftarrow \mathbb{F}_p^m$ and the internal randomness of $D_{\mathsf{bit}}^{(j)}$, this list will contain $P(\vec{x})$.

We repeat $D^{(j)}$ for $O(m \log p)$ times and fix its internal randomness, so that in what follows we can assume $D^{(j)}$ is deterministic. With probability at least $1 - 1/(10p^{2m})$, for every $0 \le j < m$, $D^{(j)}$ will be correct in the following sense: For some $\rho \triangleq 1/\Theta(M^2 m \log p)$, $D^{(j)}$ outputs ρ^{-2} elements, and

$$\Pr_{\vec{x} \leftarrow \mathbb{F}_p^m} \left[P(\vec{x}) \in D^{(j)}(u_{M-1}, u_{M-2}, \ldots, u_1) \right] > \rho.$$

Learn Next Curve. We will use the following notation from [SU05]. Let $r \triangleq O(m \log p)$ be a parameter denoting the number of reference points, and $v \triangleq (m+1)r - 1$ denotes the degree of curves.[20] A *curve* is a polynomial $C : \mathbb{F}_p \to \mathbb{F}_p^m$ with degree v. (That is, each coordinate of C is a univariate polynomial of degree v over $\mathbb{F}_p$.) Recall that $A \in \mathbb{F}_p^{m \times m}$ is the generator matrix. We use AC to denote the curve where for each $t \in \mathbb{F}_p$, $AC(t) = A \cdot C(t)$ (note that AC is still a degree-v polynomial). We also use $P(C)$ to denote the function such that for every $t \in \mathbb{F}_p$, $P(C)(t) = P(C(t))$; the *evaluation table* of $P(C)$ is the length-p vector where for every $t \in \mathbb{F}_p$, the t-th entry of the vector is $P(C(t))$.

Now, we recall the implementation of an important subroutine called LEARN NEXT CURVE as defined in [SU05, Section 5.5]. LEARN NEXT CURVE takes as input a next curve $C : \mathbb{F}_p \to \mathbb{F}_p^m$, a set of reference points $R \subseteq \mathbb{F}_p$ of size r, a stride $0 \le j < m$, as well as input evaluations; the input evaluations consist of two parts, namely the evaluation tables of $P(A^{-ip^j}C)$ for every $1 \le i < M$ and the values of $P(C(t))$ for every $t \in R$. The intended output evaluations consist of the evaluation table of $P(C)$.

In particular, LEARN NEXT CURVE starts by obtaining a set of ρ^{-2} values

$$S_t \triangleq D^{(j)}\Big(P(A^{-(M-1)p^j}C(t)), P(A^{-(M-2)p^j}C(t)), \ldots, P(A^{-p^j}C(t)) \Big)$$

for each $t \in \mathbb{F}_p$. Then it calls the algorithm from Theorem 4.2 on the pairs $\{(t, e)\}_{t \in \mathbb{F}_p, e \in S_t}$ to obtain the list of all polynomials Q such that $Q(t) \in S_t$ for many coordinates t. (This takes $\mathrm{poly}(p\rho^{-2}, \log p) \le \mathrm{poly}(p, m)$ time.) If this list contains a unique polynomial Q such that $Q(t) = P(C(t))$ for every $t \in R$, then we output this polynomial; otherwise we output $\bot$. It is clear that LEARN NEXT CURVE runs in $\mathrm{poly}(p, m)$ time.

We say LEARN NEXT CURVE *succeeds* (on next curve, reference points, and stride), if whenever the input evaluations are the intended values, the output evaluations are also the intended values. Let

$$\varepsilon_{\mathrm{LNC}} \triangleq O(v\rho^{-1}/p)^{v/2} + (8\rho^{-3})(v \deg(P)/p)^r.$$

It is proven in [SU05, Lemma 5.12] that, assuming $p > 32 \deg(P)v/\rho^4$, if the next curve and reference points are chosen uniformly at random, LEARN NEXT CURVE succeeds with probability $1 - \varepsilon_{\mathrm{LNC}}$. Since $\deg(P) = \Delta$, $\rho^{-1} = \Theta(M^2 m \log p)$, $v = O(m^2 \log p)$, and $p > \Delta^2 m^7 M^9$, it is indeed the case that $p > 32 \deg(P)v/\rho^4$. Also note that

$$\varepsilon_{\mathrm{LNC}} \le O(\rho^3/32 \deg(P))^{v/2} + (8\rho^{-3})(\rho^4/32)^r \le (1/2)^{r-1} \ll 1/(10p^{4m}).$$

A first attempt for the reconstruction algorithm would be the following. Let $i \in [p^m - 1]$, and suppose that we want to compute $P(A^i\vec{1})$. We write i in p-ary as $i = \sum_{j=0}^{m-1} i_j p^j$ (where each

---

[20]The parameter v is set in the proof of [SU05, Lemma 5.14].

$i_j \in \{0, 1, \ldots, p-1\}$). Recall that for each next curve C and stride j, given the evaluation tables of $P(A^{-kp^j}C)$ for every $1 \leq k < M$, we can learn the evaluation table of $P(C)$ in one invocation of LEARN NEXT CURVE. Therefore, we proceed in m rounds, where for each $0 \leq l < m$, the l-th round performs the following computation:

- Let $i' \triangleq \sum_{j=0}^{l-1} i_j p^j$. Suppose that at the beginning of the l-th round, we already know the evaluation tables of $P(A^{kp^l+i'}C)$ for each $1 \leq k < M$. (For $l = 0$, these values can be hardcoded as advice; for $l \geq 1$, they should be obtained from the previous round.) We invoke LEARN NEXT CURVE $M(p-1)$ times with stride l to obtain the evaluation tables of $P(A^{kp^l+i'}C)$ for each $1 \leq k < M \cdot p$. The l-th round ends here; note that we have obtained the evaluation tables required in the $(l+1)$-th round (namely $P(A^{kp^{l+1}+i_l p^l+i'}C)$ for every $1 \leq k < M$).

However, there is one issue with this approach: to learn a curve C, we also need to provide LEARN NEXT CURVE with the evaluations of some reference points on C. To resolve this issue, [SU05] introduced an *interleaved learning* procedure that involves two curves C_1 and C_2. These two curves possess nice intersection properties that for certain choices of k and l, $A^k C_1$ and $A^l C_2$ intersect on at least r points. This enables us for example to learn the evaluation table of $P(A^l C_2)$ whenever we know the evaluation table of $P(A^k C_1)$, by using the evaluations of $P(A^k C_1)$ at reference points R, where R is the intersection of $A^k C_1$ and $A^l C_2$.

Interleaved learning. In what follows, we use $[C_1 \cap C_2]$ to denote the set $\{t \in \mathbb{F}_p : C_1(t) = C_2(t)\}$. We say two curves C_1 and C_2 are *good* if they satisfy the following properties:

- $C_1(1) \neq \vec{0}$;

- for all $1 \leq i < p^m$ and all $0 \leq j < m$, $[A^{i+p^j}C_1 \cap A^i C_2]$ and $[A^i C_1 \cap A^i C_2]$ are of size $\geq r$;

- for all $1 \leq i < p^m$ and all $0 \leq j < m$, LEARN NEXT CURVE succeeds given next curve $A^{i+p^j}C_1$, reference points $[A^{i+p^j}C_1 \cap A^i C_2]$, and stride j; and

- for all $1 \leq i < p^m$ and all $0 \leq j < m$, LEARN NEXT CURVE succeeds given next curve $A^i C_2$, reference points $[A^i C_1 \cap A^i C_2]$, and stride j.

By [SU05, Lemma 5.14], there is a poly(v, p)-time randomized algorithm that, with probability $1 - 2mp^m \cdot \varepsilon_{\mathrm{LNC}} \geq 1 - 1/(10p^{2m})$, outputs two curves C_1 and C_2 that are good.

The basic step in the reconstruction algorithm is called *interleaved learning* in [SU05]. This step has the following guarantee: For a stride j, given the correct evaluation tables of $P(A^{i-kp^j}C_1)$ and $P(A^{i-kp^j}C_2)$ for every $1 \leq k < M$, we can compute the correct evaluation tables of $P(A^i C_1)$ and $P(A^i C_2)$. In particular, *interleaved learning* consists of the following two steps:

- first, we invoke LEARN NEXT CURVE with next curve $A^i C_1$, reference points $[A^{i-p^j}C_2 \cap A^i C_1]$, and stride j;

- then, we invoke LEARN NEXT CURVE with next curve $A^i C_2$, reference points $[A^i C_1 \cap A^i C_2]$, and stride j.

Note that we assume that all invocations of LEARN NEXT CURVE succeed, as this happens with high probability $(1 - 1/(10p^{2m}))$.

The reconstruction algorithm. Recall that our reconstruction algorithm needs to output two elements: a vector $\vec{v} \in \mathbb{F}_p^m \setminus \{\vec{0}\}$ and a D-oracle circuit $C : [p^m - 1] \to \mathbb{F}_p$ such that $C(i) = P(A^i \cdot \vec{v})$ for every $i \in [p^m - 1]$.

We first compute the curves C_1 and C_2 that are good with probability $1 - 1/(10p^{2m})$. Our reconstruction algorithm will be correct provided that C_1 and C_2 are good (and that we fixed good internal randomness of our next-element predictors $D^{(j)}$); this happens with probability $\geq 1 - 1/(10p^{2m}) - 1/(10p^{2m}) \geq 1 - 1/p^{2m}$. The vector we output will be $\vec{v} \triangleq C_1(1)$ (which is non-zero if C_1 and C_2 are good). It remains to output a circuit C such that for every $i \in [p^m - 1]$, $C(i) = P(A^i \cdot \vec{v})$.

Given an integer i, our circuit C first writes i in p-ary as $i = \sum_{j=0}^{m-1} i_j p^j$. Then, it proceeds in m rounds, where for each $0 \leq l < m$, the l-th round performs the following:

- Let $i' \triangleq \sum_{j=0}^{l-1} i_j p^j$. Suppose that at the beginning of the l-th round, we already know the evaluation tables of $P(A^{kp^l + i'} C_1)$ and $P(A^{kp^l + i'} C_2)$ for each $1 \leq k < M$. We perform interleaved learning $M(p-1)$ times with stride l to obtain the evaluation tables of $P(A^{kp^l + i'} C_1)$ and $P(A^{kp^l + i'} C_2)$ for each $1 \leq k < M \cdot p$. The l-th round ends here; note that we have obtained the evaluation tables required to perform the $(l+1)$-th round (namely, $P(A^{kp^{l+1} + i_l p^l + i'} C_1)$ and $P(A^{kp^{l+1} + i_l p^l + i'} C_2)$ for every $1 \leq k < M$).

Finally, after the $(m-1)$-th round, we have obtained the evaluation table of $P(A^i C_1)$, and we can simply output $P(A^i C_1(1)) = P(A^i \vec{v})$ as the answer.

Note that the interleaved learning procedure needs to invoke the next-element predictor, therefore our circuit C will be a D-oracle circuit. Also, at the beginning of the first (0-th) round, we need the evaluation tables of $P(A^k C_1)$ and $P(A^k C_2)$ for each $0 \leq k < M$. Our reconstruction algorithm can simply query the polynomial P to obtain these values and hardcode them into our circuit C. It is clear that our reconstruction algorithm runs in $\mathrm{poly}(p, m)$ time and succeeds with probability $\geq 1 - 1/p^{2m}$.

4.3 Modified Shaltiel–Umans Generator: Proof of Theorem 4.1

In this subsection, we prove Theorem 4.1, which is restated below.

Theorem 4.1 (A HSG with Uniform Learning Reconstruction)**.** *There exist an algorithm* H *and a probabilistic oracle algorithm* $\mathsf{R}^{(-)}$ *such that the following holds. Let* p *be a nice power of 2,* m *be a power of 3,* $\Delta, M \in \mathbb{N}$ *with* $p > \Delta^2 m^7 M^9$, *and let* $\vec{\ell} \triangleq (p, m, M, \Delta)$ *be the input parameters.*

- *The generator* $\mathsf{H}_{\vec{\ell}}$ *takes as input a polynomial* $P : \mathbb{F}_p^m \to \mathbb{F}_p$ *with total degree at most* Δ, *specified as a list of* p^m *evaluations of* P *on all points from* $\mathbb{F}_p^m$ *in the lexicographic order, and outputs a set of strings in* $\{0,1\}^M$. *Moreover,* $\mathsf{H}_{\vec{\ell}}$ *can be implemented by a logspace-uniform circuit of size* $\mathrm{poly}(p^m)$ *and depth* $\mathrm{poly}(\log p, m, M)$.

- *The reconstruction algorithm* $\mathsf{R}_{\vec{\ell}}^{D,P}$, *where* $D : \{0,1\}^M \to \{0,1\}$ *is any function that* $(1/M)$-*avoids* $\mathsf{H}_{\vec{\ell}}(P)$, *runs in time* $\mathrm{poly}(p, m)$ *and outputs, with probability at least* $1 - 1/p^m$, *a* D-*oracle circuit that computes* P.

Proof. One difference between our generator and the Shaltiel–Umans generator (Theorem 4.9) is that the reconstruction procedure in the latter only learns a circuit C_0 that computes the mapping $i \mapsto P(A^i \cdot \vec{v})$ (for some $\vec{v}$ output by the reconstruction procedure), where A is the generator matrix used in the Shaltiel–Umans construction, instead of a circuit that computes P itself. Let us assume for simplicity that the circuit C_0 computes $i \mapsto P(A^i \cdot \vec{1})$. Note that if given $\vec{x} \in \mathbb{F}_p^m \setminus \{\vec{0}\}$ (which

is the input on which we intend to evaluate P), we could *efficiently* compute the value $i \in [p^m - 1]$ such that $A^i \cdot \vec{1} = \vec{x}$, then we would be able to combine this with the circuit C_0 to compute P (roughly speaking, by first computing i and then outputting $C_0(i)$). However, there are two issues with this approach:

1. First, we do not know the generator matrix A, as we need our reconstruction algorithm to be uniform and thus we cannot hardcode A.

2. Second, the task of finding such i given $\vec{x}$ and A is essentially the *discrete logarithm problem*, for which no efficient algorithm is known.

To handle the first issue, we will construct our generator by using the Shaltiel–Umans construction based on a generator matrix that is from a small set S given by Lemma 4.5. Then in the reconstruction, we will try all the matrices from S, which can be generated efficiently, to obtain a list of candidate circuits. We then select from the list a circuit that is sufficiently close to P and use a self-corrector to compute P everywhere. For the second issue, we first observe that the mapping $f \colon i \mapsto A^i \cdot \vec{1}$ is a *permutation*. Treating f as a "cryptographic one-way permutation" and invoking Theorem 4.7, we can construct a "cryptographic pseudorandom generator", which has a uniform reconstruction algorithm. We can then combine the output of this "cryptographic pseudorandom generator" with that of the Shaltiel–Umans generator so that if there is an algorithm D that avoids this combined generator, then D can also be used to invert f efficiently! Details follow.

The construction of H.   For a matrix $A \in \mathbb{F}_p^{m \times m}$, let $f_A \colon [p^m - 1] \cup \{0\} \to \mathbb{F}_p^m$ be such that

$$f_A(i) \triangleq \begin{cases} 0^n & \text{if } i = 0 \\ A^i \cdot \vec{1} & \text{if } 1 \leq i < p^m. \end{cases}$$

We will also view f as a function mapping s bits to s bits, where $s \triangleq m \cdot \log p$. Also note that if A is a generator matrix for $\mathbb{F}_p^m$, then f_A is a permutation.

Let HSU be the generator from Theorem 4.9 and $\mathsf{CryptoG}^{(-)}$ be the generator from Theorem 4.7. Also, let $S \subseteq \mathbb{F}_p^{m \times m}$ be the set of matrices constructed using Lemma 4.5. We define

$$\mathsf{H}_{\vec{\ell}}(P) \triangleq \bigcup_{A \in S} \left(\mathsf{HSU}_{\vec{\ell}}(P, A) \bigcup \mathsf{CryptoG}_{s,M}^{f_A} \right).$$

The complexity of H.   We argue that $\mathsf{H}_{\vec{\ell}}$ can be implemented by a logspace-uniform circuit of size $\mathrm{poly}(p^m)$ and depth $\mathrm{poly}(\log p, m, M)$.

First note that given A, f_A can be computed in $\mathrm{poly}(\log p, m)$ time. Then again by the fact that every time-t Turing machine can be simulated by a logspace-uniform circuit of size $O(t^2)$, f_A can be computed by a logspace-uniform circuit of size $\mathrm{poly}(\log p, m)$. This means given A, $\mathsf{CryptoG}_{s,M}^{f_A}$, which by Theorem 4.7 has a logspace-uniform f_A-oracle circuit of size $\mathrm{poly}(2^s, M)$ and depth $\mathrm{poly}(s, M)$, can be implemented by a logspace-uniform circuit of size $\mathrm{poly}(p^m)$ and depth $\mathrm{poly}(\log p, m, M)$, where we have used that $s = m \cdot \log p$ and $M \leq p^{1/9}$. Also, by Theorem 4.9, $\mathsf{HSU}_{\vec{\ell}}$ has a logspace-uniform circuit of size $\mathrm{poly}(p^m)$ and depth $\mathrm{poly}(\log p, m, M)$. To compute $\mathsf{H}_{\vec{\ell}}(P)$, we just need to compute $\mathsf{HSU}_{\vec{\ell}}(P, A)$ and $\mathsf{CryptoG}_{s,M}^{f_A}$ for all $A \in S$ in parallel, where S can also be computed in time $\mathrm{poly}(\log p, m)$ and hence has logspace-uniform circuit of size $\mathrm{poly}(\log p, m)$. This yields a logspace-uniform circuit of size $\mathrm{poly}(p^m)$ and depth $\mathrm{poly}(\log p, m, M)$ computing $\mathsf{H}_{\vec{\ell}}$.

The reconstruction. Given oracle access to the polynomial P and a function D that $(1/M)$-avoids $\mathsf{H}_{\vec{\ell}}(P)$, we want to output a D-circuit that computes P. We do this in two stages. In the first stage, we obtain a list of candidate circuits, one for each $A \in S$, that (with high probability) contains at least one circuit that computes P. In the second stage, we will select, from the list of candidate circuits, one that is sufficiently close to P and combine it with a self-corrector to obtain a circuit that computes P on all inputs.

We now describe the first stage. Let A^* be the lexicographically first matrix in S that is a generator matrix for $\mathbb{F}_p^m$, and consider the two sets

$$\mathsf{HSU}_{\vec{\ell}}(P, A^*) \quad \text{and} \quad \mathsf{CryptoG}_{s,M}^{f_{A^*}},$$

which are subsets of $\mathsf{H}_{\vec{\ell}}(P)$. Since D avoids $\mathsf{H}_{\vec{\ell}}$, it also avoids *both* $\mathsf{HSU}_{\vec{\ell}}(P, A^*)$ and $\mathsf{CryptoG}_{s,M}^{f_{A^*}}$.

Assume for a moment that we are given the matrix A^*. We will construct a circuit C_{A^*} as follows. Let $\mathsf{RSU}^{(-)}$ and $\mathsf{Invert}^{(-)}$ be the oracle algorithms from Theorem 4.9 and Theorem 4.7 respectively. We first run $\mathsf{RSU}_{\vec{\ell}}^{D,P}(A^*)$ to obtain a D-oracle circuit C'_{A^*} and some $\vec{v} \in \mathbb{F}_p^m \setminus \{\vec{0}\}$. By the property of $\mathsf{RSU}^{(-)}$ (item 2 of Theorem 4.9) and the fact that D avoids $\mathsf{HSU}_{\vec{\ell}}(P, A^*)$, we get that, with probability at least $1 - 1/p^{2m}$, for every $i \in [p^m - 1]$,

$$C'_{A^*}(i) = P((A^*)^i \cdot \vec{v}). \tag{2}$$

Similarly, by the property of $\mathsf{Invert}^{(-)}$ (item 2 of Theorem 4.7) and the fact that D avoids $\mathsf{CryptoG}_{s,M}^{f_{A^*}}$, we get that

$$\Pr_{x \leftarrow \{0,1\}^s}\left[\mathsf{Invert}_{s,M}^{f_{A^*},D}(x) = f_{A^*}^{-1}(x)\right] \geq \frac{1}{\mathrm{poly}(M)}.$$

By combining

$$g \triangleq \mathsf{Invert}_{s,M}^{f_{A^*},D}$$

with the algorithm $\mathsf{DLCorr}^{(-)}$ from Lemma 4.6, we get that for every $\vec{x} \in \mathbb{F}_p^m$, with probability at least $2/3$ over the internal randomness of DLCorr^g,

$$\mathsf{DLCorr}^g\left(p, m, 1^{\mathrm{poly}(M)}, A^*, \vec{x}\right) = f_{A^*}^{-1}(\vec{x}).$$

By using standard error reduction techniques (to reduce the error from $2/3$ to $1/(10p^{2m})$) and by fixing the internal randomness (that hopefully works correctly for all p^m inputs), we can obtain, in time $\mathrm{poly}(p, m)$ and with probability at least $1 - 1/(10p^m)$, a D-oracle circuit C''_{A^*} such that for every $\vec{x} \in \mathbb{F}_p^m$,

$$C''_{A^*}(\vec{x}) = f_{A^*}^{-1}(\vec{x}). \tag{3}$$

That is, given $\vec{x} \in \mathbb{F}_p^m \setminus \{\vec{0}\}$, $C''_{A^*}(\vec{x})$ outputs $i \in [p^m - 1]$ such that $(A^*)^i \cdot \vec{1} = \vec{x}$. This is almost what we need except that we want the circuit to output i such that $(A^*)^i \cdot \vec{v} = \vec{x}$. We further construct such a circuit C'''_{A^*} as follow. Given $\vec{x} \in \mathbb{F}_p^m$, we first compute

$$j \triangleq C''_{A^*}(\vec{v}) \quad \text{and} \quad k \triangleq C''_{A^*}(\vec{x}).$$

That is, $\vec{v} = (A^*)^j \cdot \vec{1}$ and $\vec{x} = (A^*)^k \cdot \vec{1}$. We then output i depending on the values of j and k. On the one hand, if $j < k$, we let $i \triangleq k - j$. Then

$$(A^*)^i \cdot \vec{v} = (A^*)^{k-j} \cdot (A^*)^j \cdot \vec{1} = (A^*)^k \cdot \vec{1} = \vec{x}.$$

On the other hand, if $k \leq j$, we let $i \triangleq p^m - 1 - (j - k)$, which yields

$$(A^*)^i \cdot \vec{v} = (A^*)^{p^m - 1 - j + k} \cdot (A^*)^j \cdot \vec{1} = I \cdot (A^*)^k \cdot \vec{1} = \vec{x}.$$

Now we have a circuit C'''_{A^*} that given $\vec{x} \in \mathbb{F}_p^m \setminus \{\vec{0}\}$, outputs $i \in [p^m - 1]$ such that $(A^*)^i \cdot \vec{v} = \vec{x}$ and a circuit C'_{A^*} that given $i \in [p^m - 1]$, computes $P((A^*)^i \cdot \vec{v})$. We then construct the circuit

$$C_{A^*}(\vec{x}) \triangleq \begin{cases} P(\vec{0}) & \text{if } \vec{x} = \vec{0} \\ C'_{A^*}(C'''_{A^*}(\vec{x})) & \text{if } \vec{x} \in \mathbb{F}_p^m \setminus \{\vec{0}\}. \end{cases}$$

Note that we can hardwire the value of $P(\vec{0})$. Also notice that if both Equations 2 and 3 are true (which happens with probability at least $1 - 1/(9p^m)$) we will get that for all $\vec{x} \in \mathbb{F}_p^m$,

$$C_{A^*}(\vec{x}) = P(\vec{x}).$$

However, we don't know the matrix A^*. Instead, we will run the above procedure for each $A \in S$ to obtain a list $\mathcal{C} \triangleq \{C_A\}_{A \in S}$ of candidate circuits C_A. Then, with probability at least $1 - 1/(9p^m)$, $\mathcal{C}$ contains at least one circuit (in particular, C_{A^*}) that computes the polynomial P.

Given the list of candidate circuits $\mathcal{C}$, we now describe the second stage. First of all, given a circuit $C_A \in \mathcal{C}$, we want to check if C_A is sufficiently close to P.

Claim 4.10. *There is a randomized algorithm* IsClose *that, given a circuit* $B : \mathbb{F}_p^m \to \mathbb{F}_p$, $\delta \in (0, 1]$, *and oracle access to the polynomial* P, *runs in time* $\text{poly}(|B|) \cdot \log(1/\delta)$ *such that*

- *if* $\Pr_{\vec{x}}[B(\vec{x}) = P(\vec{x})] = 1$, *the algorithm accepts with probability* 1, *and*

- *if* $\Pr_{\vec{x}}[B(\vec{x}) = P(\vec{x})] \leq 3/4$, *the algorithm rejects with probability at least* $1 - \delta$.

Proof of Claim 4.10. The algorithm picks $3 \log(1/\delta)$ points uniformly at random from $\mathbb{F}_p^m$ and checks if B and P agree on all those points. If so, the algorithm accepts; otherwise it rejects. Note that if $\Pr_{\vec{x}}[B(\vec{x}) = P(\vec{x})] \leq 3/4$, then the probability that it accepts is at most $(3/4)^{3\log(1/\delta)} < \delta$. $\diamond$

For each $C_A \in \mathcal{C}$, we run $\text{IsClose}^P(C_A, \delta \triangleq 1/(4|\mathcal{C}|p^m))$ and pick the first one that the algorithm accepts. By the fact that $\mathcal{C}$ contains at least one circuit that computes P and by the property of the algorithm IsClose (Claim 4.10), with probability at least $1 - 1/(4p^m)$, we will obtain some D-oracle circuit C_{close} such that

$$\Pr_{\vec{x} \leftarrow \mathbb{F}_p^m}[C_{\text{close}}(\vec{x}) = P(\vec{x})] > 3/4. \tag{4}$$

Conditioned on Equation 4, by combining C_{close} with the self-corrector $\text{PCorr}^{(-)}$ from Theorem 4.8, we get that for every $\vec{x} \in \mathbb{F}_p^m$, $\text{PCorr}^{C_{\text{close}}}(p, m, \Delta, \vec{x}) = P(\vec{x})$ with probability at least $2/3$ (over the internal randomness of $\text{PCorr}^{C_{\text{close}}}$). Again, by using standard error reduction techniques and by picking a randomness uniformly at random, we can obtain in time $\text{poly}(p, m)$, with probability at least $1 - 1/(4p^m)$, a D-oracle circuit C that computes P.

By a union bound, the above procedure gives, with probability at least $1 - 1/p^m$, a D-oracle circuit that computes the polynomial P.

Finally, it is easy to verify that the running time is $\text{poly}(p, m)$. $\square$

5   Improved Chen–Tell Targeted Hitting Set Generator

In this section, we prove Theorem 3.1, showing how to build a reconstructive hitting set generator from any uniform low-depth circuit.

5.1 Layered-Polynomial Representation

The first step is to "arithmetize" our low-depth circuit into a *layered-polynomial representation*. Roughly speaking, given a (uniform) circuit C of depth d and size T, we will produce a table of size $d' \times T'$ where $d' \approx d$ and $T' = \text{poly}(T)$, such that the following key properties hold:

(Low-degree.) Each row is the "truth table" of a low-degree polynomial (thus admits self-correction properties).

(Faithful representation.) Given oracle access to the d'-th row, we can compute the output of $C(1^n)$ quickly.

(Downward self-reducibility.) For each $2 \leq i \leq d'$, given oracle access to the $(i-1)$-th polynomial, we can quickly compute the output of the i-th polynomial on a given input. Moreover, the entries of the first row (corresponding to $i = 1$) can be computed quickly.

Later, we will use these properties of the layered-polynomial representation to compile them into a reconstructive HSG.

We now formally describe our layered-polynomial representation, which can be proved by modifying the construction in [CT21]. In the following, letting p be a power of 2, and $f \colon \mathbb{F}_p^\ell \to \mathbb{F}_p$, we use $\text{tt}(f)$ to denote the length-$(p^\ell \cdot \log p)$ Boolean string that consists of p^ℓ blocks, where the i-th block corresponds to the Boolean encoding of the i-th element in $\mathbb{F}_p$.

Theorem 5.1 (Layered-Polynomial Representation). *There exist universal constants $c, c', \beta > 1$ such that the following holds. Let $\kappa \in \mathbb{N}$ and let $T, d, n, h, p \in \mathbb{N}$ all be sufficiently large such that (1) $d \leq T$ and $n \leq T$, and (2) h, p are both nice powers of 2 and $\log T \leq h < p \leq h^{27} \leq T$. (Recall that p is a nice power of 2 if $p = 2^{2 \cdot 3^\lambda}$ for some $\lambda \in \mathbb{N}$.)*

Let $\vec{\ell} \triangleq (\kappa, T, d, n, h, p)$ be the input parameters, and let $\mathbb{F} \triangleq \mathbb{F}_p$. For a Turing machine TM with description size $|\mathsf{TM}| = \kappa \cdot \log T$, let

$$C_{\mathsf{TM}} \triangleq \mathsf{Circuit}[T, \kappa \cdot \log T, n, n](\mathsf{TM}).$$

Assuming $C_{\mathsf{TM}} \neq \perp$ and C_{TM} has depth at most d, there are $d' \triangleq c\kappa \cdot \log^2 T \cdot (d + \kappa^2 \log T)$ polynomials $\left(P_i^{\vec{\ell}, \mathsf{TM}}\right)_{i \in [d']}$ such that the following hold (below we write $P_i^{\vec{\ell}, \mathsf{TM}}$ as P_i for simplicity):

1. **(Arithmetic setting.)** *Let $H \subset \mathbb{F}$ be the first h elements of $\mathbb{F}$, and let m be the smallest power of 3 such that $h^m \geq T^{\beta\kappa}$. Each polynomial is from $\mathbb{F}^{3m}$ to $\mathbb{F}$ and has total degree at most $\Delta = c \cdot h \cdot \log^3(T)$.*

2. **(Faithful representation.)** *Fix an injective function $\mathsf{id} \colon [n] \to H^m$ in an arbitrary but canonical way.[21] For every $i \in [n]$, $(C_{\mathsf{TM}}(1^n))_i = P_{d'}(\mathsf{id}(i), 0^{2m})$.*

3. **(Complexity of the polynomials.)** *Let $T_{\mathsf{poly}} \triangleq T^{c \cdot \kappa}$ and $d_{\mathsf{poly}} \triangleq c \cdot (d \log T + \kappa^2 \log^2 T)$. There is a Turing machine $\mathsf{TM}_{\mathsf{poly}}$ of description length $\log T_{\mathsf{poly}}$ such that for*

$$C_{\mathsf{poly}} \triangleq \mathsf{Circuit}\left[T_{\mathsf{poly}}, \log T_{\mathsf{poly}}, \log d', |\mathbb{F}|^{3m} \cdot \log |\mathbb{F}|\right](\mathsf{TM}_{\mathsf{poly}}),$$

   it holds that (1) for every $i \in [d']$ $C_{\mathsf{poly}}(i) = \text{tt}(P_i)$ and (2) C_{poly} has depth d_{poly}.

   Moreover, there is a polynomial-time algorithm $\mathbb{A}_{\vec{\ell}}^{\mathsf{poly}}$ that takes $\mathsf{TM} \in \{0, 1\}^{\kappa \log T}$ as input, and outputs the description of $\mathsf{TM}_{\mathsf{poly}}$.

---

[21] For simplicity, we will ignore the complexity of computing id and its inverse since it is negligible.

4. **(Downward self-reducibility.)** *There is a* $\max(n, h) \cdot h^{c'}$*-time algorithm* Base *that takes inputs* $\vec{\ell}$, $\mathsf{TM} \in \{0, 1\}^{\kappa \cdot \log T}$, *and* $\vec{w} \in \mathbb{F}^{3m}$, *outputs* $P_1(\vec{w})$.

Also, there is an $h^{c'}$-time oracle algorithm DSR that takes inputs $\vec{\ell}$, $\mathsf{TM} \in \{0, 1\}^{\kappa \cdot \log T}$, $i \in \{2, \ldots, d'\}$, and $\vec{w} \in \mathbb{F}^{3m}$, and oracle access to a polynomial $\widetilde{P} \colon \mathbb{F}^{3m} \to \mathbb{F}$, such that when it is given P_{i-1} as the oracle, it outputs $P_i(\vec{w})$.

Proof. Recall that we use $\vec{\ell}$ to denote the input parameters (κ, T, d, n, h, p). We will follow the proof of [CT21, Proposition 4.7], which in turn follows [Gol17] (see also [Gol18]). In the following, we will simply use C to denote the (low-depth) circuit $C_{\mathsf{TM}} = \mathsf{Circuit}[T, \kappa \cdot \log T, n, n](\mathsf{TM})$ for notational convenience, but we stress that C depends on both $\vec{\ell}$ and TM (and so does the later circuits constructed from C).

5.1.1 Construction of a Highly Uniform Circuit D

We first construct a circuit D that has better uniformity and preserves the functionality of C, i.e., $D(1^n) = C(1^n)$. Given input 1^n, D first computes a description of $C = \mathsf{Circuit}[T, \kappa \cdot \log T, n, n](\mathsf{TM})$ (represented as a $T \times T \times T$ tensor) and then computes the Eval function $\langle \langle C \rangle, n, d \rangle \mapsto C(1^n)$. Let $s \triangleq \kappa \cdot \log T$ and $s' \triangleq O(s + \log(3 \log T))$ be such that each configuration of TM on $3 \log T$-bit inputs can be described by s' bits.

The circuit D is constructed by composing the following three subcircuits. Let $\mu \in \mathbb{N}$ be a sufficiently large universal constant. We will describe and analyze their complexities (or state the complexity bounds proved in [CT21, Gol17]).

1. **(Computing the adjacency matrices for configurations.)** The first circuit $D^{(1)}$ takes n bits as input (which are supposed to be 1^n), outputs a list of T^3 matrices from $\{0, 1\}^{2^{s'} \times 2^{s'}}$, such that the (u, v, w)-th matrix[22] $M^{(u,v,w)}$ satisfies the following condition: for every $\gamma, \gamma' \in \{0, 1\}^{s'}$, $M^{(u,v,w)}[\gamma, \gamma'] = 1$ if and only if $\mathbb{A}_{\mathsf{nxt}}(\mathsf{TM}, s, (u, v, w), \gamma, \gamma')$ (*i.e.*, γ' is the configuration obtained by running TM for one step on configuration γ and input (u, v, w) with space bound s). Recall we assumed that if γ is the accepting or the rejecting configuration, then its next configuration is γ itself.

   Complexity of $D^{(1)}$. $D^{(1)}$ can be implemented by a projection (*i.e.*, depth $d_{D^{(1)}} = 2$ and size $T_{D^{(1)}} = T^3 \cdot 2^{2s'}$).[23] Moreover, from Fact 2.2, given $\vec{\ell}$ and TM, in polynomial time we can compute a Turing machine $\mathsf{TM}_{D^{(1)}} \in \{0, 1\}^{(\kappa+\mu) \cdot \log T}$ such that

   $$\mathsf{Circuit}\Big[T_{D^{(1)}}, s_{D^{(1)}}, n, T^3 \cdot 2^{2s'}\Big](\mathsf{TM}_{D^{(1)}}) = D^{(1)},$$

   where $s_{D^{(1)}} = \mu \cdot s'$.

2. **(Computing the adjacency relation tensor of C via matrix multiplication.)** The second circuit $D^{(2)}$ takes a list of T^3 matrices from $\{0, 1\}^{2^{s'} \times 2^{s'}}$ as input, and outputs a tensor from $\{0, 1\}^{T \times T \times T}$ followed by the encoding of a pair (n, d).

   In more detail, given the output of $D^{(1)}(1^n)$, for every $(u, v, w) \in [T]^3$, it determines whether $\mathsf{TM}(u, v, w) = 1$ by computing $(M^{(u,v,w)})^{2^{s'}}$, which can be done by repeated squaring s' times. This gives the adjacent relation tensor of C.

---

[22] We use $(u, v, w) \in [T]^3$ to denote the integer $(u-1)T^2 + (v-1)T + w \in [T^3]$.

[23] Note that we can implement projections and restrictions of input bits to 0 and 1 using two layers of NAND gates.

Complexity of $D^{(2)}$. $D^{(2)}$ can be implemented by a circuit of depth $d_{D^{(2)}} = \mu \cdot (s')^2$ and size $T_{D^{(2)}} = T^3 \cdot 2^{\mu s'}$. Moreover, from [CT21, Gol17] (note that $D^{(2)}$ does not depend on TM), given $\vec{\ell}$, in polynomial time we can compute a Turing machine $\mathsf{TM}_{D^{(2)}} \in \{0,1\}^{\mu \cdot \log T}$ such that

$$\mathsf{Circuit}\Big[T_{D^{(2)}}, s_{D^{(2)}}, T^3 \cdot 2^{2s'}, T^3 + |(n,d)|\Big](\mathsf{TM}_{D^{(2)}}) = D^{(2)},$$

where $s_{D^{(2)}} = \mu \cdot s'$.

3. (**Computing Eval.**) The final circuit $D^{(3)}$ takes $\langle \langle C \rangle, n, d \rangle$ as input, and outputs $\mathsf{Eval}(\langle C \rangle, n, d)$.

Complexity of $D^{(3)}$. $D^{(3)}$ can be implemented by a circuit of depth $d_{D^{(3)}} = \mu \cdot d \cdot \log T$ and size $T_{D^{(3)}} = T^\mu$. Moreover, from [CT21, Gol17] (note that $D^{(3)}$ does not depend on TM), given $\vec{\ell}$, in polynomial-time we can compute a Turing machine $\mathsf{TM}_{D^{(3)}} \in \{0,1\}^{\mu \cdot \log T}$ such that

$$\mathsf{Circuit}\big[T_{D^{(3)}}, s_{D^{(3)}}, T^3 + |(n,d)|, n\big](\mathsf{TM}_{D^{(3)}}) = D^{(3)},$$

where $s_{D^{(3)}} = \mu \cdot s'$.

Formally, we have

$$D = D^{(3)} \circ D^{(2)} \circ D^{(1)}.$$

Let $\beta \in \mathbb{N}$ be a sufficiently large constant such that $\beta \geq 100\mu$. The following claim summarizes the required properties of D for us.

Claim 5.2. *The following properties about the circuit D are true.*

1. *The depth of D is $d_D = \beta \cdot (\kappa^2 \cdot \log^2 T + d \cdot \log T)$ and the width of D is $T'_D = T^{\beta\kappa}$.*

2. *The layered adjacency relation function $\Phi' \colon [d_D] \times \{0,1\}^{3\log(T'_D)} \to \{0,1\}$ of D can be decided by a formula of depth $O(\log\log T)$ and size $O(\log^3 T)$. Moreover, there is an algorithm $\mathbb{A}_{\Phi'}$ that given $\vec{\ell}$ and TM as input, outputs the formula above in $O(\kappa \log T)$ space.*

3. *There is a Turing machine $\mathsf{TM}_D \in \{0,1\}^{\beta\kappa \log T}$ such that*

$$\mathsf{Circuit}[T_D, s_D, n, n](\mathsf{TM}_D) = D,\text{[24]}$$

where $T_D = T'_D \cdot (d_D + 1)$ and $s_D = \beta\kappa \log T$. Moreover, given $\vec{\ell}$ and TM as input, the description of TM_D can be computed in polynomial time.

Proof of Claim 5.2. By construction, the size of D is bounded by $\mathrm{poly}(T) \cdot 2^{O(s')} \leq T^{O(\kappa)}$ (recall that $s' = O(s + \log(3\log T))$ and $s = \kappa \log T$), and its depth is bounded by $O(s^2 + d \cdot \log T)$. The first bullet then follows directly from the fact that β is sufficiently large.

Recall that the $D^{(1)}$ part of D has depth $d_{D^{(1)}} = 2$. To see the complexity of computing $\Phi'(i, -, -, -)$ for $i > 2$, we note that the layers corresponding to $D^{(2)}$ and $D^{(3)}$ *do not* depend on TM. Hence the complexity of computing their layered adjacent relation function follows directly from [CT21, Claim 4.7.1].[25] Also, the complexity of computing $\Phi'(i, -, -, -)$ for $i \in \{1, 2\}$ follows

---

[24]Note that $\mathsf{Circuit}$ generates the unlayered version of D of size $T'_D \cdot (d_D + 1)$, Without loss of generality we can assume the first T' gates are on the first layer, the next T' gates are on the second layer, and so on.

[25]We note that although [CT21, Gol17] only claims a $\mathrm{polylog}(T)$ bound on the formula size, the formula is indeed very simple and its size and depth can be easily bounded by $O(\log^3 T)$ and $O(\log\log T)$, respectively; see [Gol17, Page 8-9] for details.

directly from Fact 2.2. To see the moreover part, again, the case for $i > 2$ follows from [CT21, Claim 4.7.1], and the case for $i \in \{1, 2\}$ follows from Fact 2.2.[26]

Finally, to obtain the algorithm that computes TM_D, we simply apply the composition $\mathbb{A}_{\mathsf{comp}}$ (from Fact 2.4) twice to compose the circuits $D^{(1)}, D^{(2)}, D^{(3)}$ in order and add some dummy gates to the circuit. The space bound and the description size bound can also be verified easily. $\diamond$

5.1.2 Arithmetization of D

The construction of the polynomials and their corresponding algorithms can then be done in the same way as in [CT21]. We only state the necessary changes to establish our theorem.

Note that $|\mathbb{F}|^m = p^m \leq \mathrm{poly}(h^{27m}) \leq T^{O(\beta\kappa)} \leq T^{O(\kappa)}$ (β is a universal constant), from our assumption that $p \leq h^{27}$ and our choice of m.

First, we need an arithmetization of each $\Phi_i' \triangleq \Phi'(i, -, -, -)$.

Claim 5.3. *For $i \in [d_D]$ there exists a polynomial $\hat{\Phi}_i \colon \mathbb{F}^{3m} \to \mathbb{F}$ that satisfies the following:*

1. *For every $(\vec{w}, \vec{u}, \vec{v}) = z_1, ..., z_{3m} \in H^{3m}$ we have that $\hat{\Phi}_i(\vec{w}, \vec{u}, \vec{v}) = 1$ if gate $\vec{w}$ in the ith layer of D is fed by gates $\vec{u}$ and $\vec{v}$ in the $(i-1)^{th}$ layer of D, and $\hat{\Phi}_i(\vec{w}, \vec{u}, \vec{v}) = 0$ otherwise.*

2. *The degree of $\hat{\Phi}_i$ is at most $O(h \cdot \log^3 T)$. Moreover, there exists an algorithm that on input $\vec{\ell}, \mathsf{TM}, i, \vec{w}, \vec{u}, \vec{v}$, computes $\hat{\Phi}_i(\vec{w}, \vec{u}, \vec{v})$ in $\mathrm{poly}(h)$ time.*

3. *For a universal constant $c_1 > 1$, there exists a circuit $C_{\hat{\Phi}}$ of size $T_{\hat{\Phi}} \triangleq T^{c_1\kappa}$ and depth $c_1\kappa \cdot \log T$ that on input $i \in [d_D]$ outputs $\mathtt{tt}(\hat{\Phi}_i) \in \mathbb{F}^{|\mathbb{F}|^{3m}}$ (represented as a Boolean string). Moreover, there is a polynomial-time algorithm $\mathbb{A}_{\hat{\Phi}}$ that takes $\vec{\ell}$ and $\mathsf{TM} \in \{0,1\}^{\kappa\log T}$ as input, and outputs the description of a Turing machine $\mathsf{TM}_{\hat{\Phi}} \in \{0,1\}^{c_1\kappa\log T}$ such that*

$$C_{\hat{\Phi}} = \mathsf{Circuit}\big[T_{\hat{\Phi}}, c_1 \cdot \kappa\log T, \log(d_D+1), |\mathbb{F}|^{3m}\log|\mathbb{F}|\big](\mathsf{TM}_{\hat{\Phi}}).$$

Proof Sketch of Claim 5.3. We first define $\hat{\Phi}_i$ and then establish each item separately.

Construction of $\hat{\Phi}_i$. Let $F_{\Phi'}$ be the $O(\log\log T)$-depth $O(\log^3 T)$-size formula computing $\Phi' \colon [d_D] \times \{0,1\}^{3 \cdot \log T_D'} \to \{0,1\}$ from Claim 5.2. For every $i \in [d_D]$, let F_i be the restriction of $F_{\Phi'}$ by fixing the first input to be i. Then, we arithmetize F_i by replacing every NAND gate in F_i by an arithmetic gate $\widetilde{\mathsf{NAND}} \colon \mathbb{F}^2 \to \mathbb{F}$ computing $\widetilde{\mathsf{NAND}}(u, v) \triangleq 1 - uv$. Denote the new formula (which is now an *arithmetic* formula) by $\hat{F}_i$.

For each j, let $\pi_j \colon H \to \{0,1\}$ be the mapping that maps $z \in H$ to the j-th bit of the encoding of z. Note that since H consists of the smallest h elements in $\mathbb{F}$, we know that $\pi(z) = (\pi_1(z), \ldots, \pi_{\log h}(z))$ is a bijection between H and $\{0,1\}^{\log h}$.[27]

For each $j \in [\log h]$, let $\hat{\pi}_j \colon \mathbb{F} \to \mathbb{F}$ be the unique degree-$(h-1)$ extension of π_j to $\mathbb{F}$, that can be computed via standard interpolation via logspace-uniform circuits of $O(\log(h \cdot \log|\mathbb{F}|)) = O(\log T)$ depth and $\mathrm{polylog}(T)$ size [HAB02, HV06] (see [CT21, Claim 4.7.2] for the details). We also let $\hat{\pi}(z) = (\hat{\pi}_1(z), \ldots, \hat{\pi}_{\log h}(z))$. Then, we set

$$\hat{\Phi}_i(z_1, \ldots, z_{3m}) \triangleq \hat{F}_i(\hat{\pi}(z_1), \hat{\pi}(z_2), \ldots, \hat{\pi}(z_{3m})).$$

---

[26] Strictly speaking we need to combine the formulas for two cases to obtain a single formula for Φ'. The overhead of doing so is negligible so we omit this discussion here.

[27] More specifically, by our specific encoding of H as strings from $\{0,1\}^{\log|\mathbb{F}|}$, $\pi(z)$ is simply the first $\log h$ bits of the encoding of z, hence it can be computed by a projection.

We also use $F_{\hat{\Phi}_i}$ to denote the *arithmetic* formula on the right side above that computes the formula $\hat{\Phi}_i$.

From the construction above, the first two items of the claim can be proved identically as [CT21, Claim 4.7.2]. It remains to establish the third item.

Construction of $C_{\hat{\Phi}}$. We hardwire the description of $F_{\Phi'}$ into $C_{\hat{\Phi}}$. The circuit $C_{\hat{\Phi}}$ takes $i \in [d_D]$ as input, performs the above computation to obtain a description of the arithmetic formula $F_{\hat{\Phi}_i}$, and then outputs the truth table of $F_{\hat{\Phi}_i}$ by evaluating it on all vectors in $\mathbb{F}^{3m}$.

In more detail, computing the description of $F_{\hat{\Phi}_i}$ from the description of $F_{\Phi'}$ and i can be done in $O(\log T)$ depth and $\mathrm{polylog}(T)$ size. $C_{\hat{\Phi}}$ then evaluates $F_{\hat{\Phi}_i}$ on all vectors from $\mathbb{F}^m$, which can be done in $\mathrm{poly}(|\mathbb{F}|^m)$ size and $O(\log(|\mathbb{F}|^m))$ depth. The third bullet (except for the moreover part) then follows by setting c_1 to be sufficiently large and recalling that $|\mathbb{F}|^m \leq T^{O(\beta\kappa)}$.

Establishing the uniformity. Finally, we establish the moreover part of the third bullet. Let $\mu_{\hat{\Phi}} \in \mathbb{N}$ be a sufficiently large universal constant that depends on the space complexity of the algorithm $\mathbb{A}_{\Phi'}$ from Claim 5.2.

Our algorithm $\mathbb{A}_{\hat{\Phi}}$ works as follows:

1. We first construct a Turing machine $\mathsf{TM}_{[1]}$ with $\vec{\ell}$ and TM hardwired. $\mathsf{TM}_{[1]}$ corresponds to a circuit $C_{[1]}$ that takes $i \in [d_D]$ as input and outputs i together with the description of $F_{\Phi'}$.[28] $C_{[1]}$ has depth $d_{[1]} = O(1)$ and size $T_{[1]} = \mathrm{polylog}(T)$. Let $s_{[1]} = \mu_{\hat{\Phi}} \cdot \kappa \log T$, we have

$$C_{[1]} = \mathsf{Circuit}\big[T_{[1]}, s_{[1]}, \log d_D, \log d_D + |F_{\Phi'}|\big]\big(\mathsf{TM}_{[1]}\big)$$

and $\mathsf{TM}_{[1]}$ has description size at most $|\mathsf{TM}| + \mu_{\hat{\Phi}} \cdot \log T = (\kappa + \mu_{\hat{\Phi}}) \cdot \log T$.

Here, we crucially use the fact that the algorithm $\mathbb{A}_{\Phi'}$ from Claim 5.2 runs in $O(\kappa \log T)$ space (and $\mu_{\hat{\Phi}}$ is sufficiently large).

2. Then we construct a Turing machine $\mathsf{TM}_{[2]}$ that corresponds to a circuit $C_{[2]}$ that takes $i \in [d_D]$ together with the description of $F_{\Phi'}$ as input and outputs $\mathtt{tt}(\hat{\Phi}_i)$. By the discussion above, from $\vec{\ell}$ we can compute a Turing machine $\mathsf{TM}_{[2]} \in \{0,1\}^{\mu_{\hat{\Phi}}\kappa \log T}$ such that for $T_{[2]} = \mathrm{poly}(|\mathbb{F}|^m) \leq T^{\mu_{\hat{\Phi}}\kappa}$, $d_{[2]} = O(\log(|\mathbb{F}|^m)) \leq \mu_{\hat{\Phi}}\kappa \cdot \log T$, $s_{[2]} = \mu_{\hat{\Phi}}\kappa \log T$, we have

$$C_{[2]} = \mathsf{Circuit}\big[T_{[2]}, s_{[2]}, \log d_D + |F_{\Phi'}|, |\mathbb{F}|^{3m}\big]\big(\mathsf{TM}_{[2]}\big) \, ,$$

and $C_{[2]}$ has depth $d_{[2]}$.

3. Finally, $\mathbb{A}_{\hat{\Phi}}$ composes $C_{[1]}$ and $C_{[2]}$ by applying Fact 2.4 and outputs the obtained Turing machine as $\mathsf{TM}_{\hat{\Phi}}$. Setting c_1 sufficiently large completes the proof. $\diamond$

Then we define the following polynomials, according to [CT21, Definition 4.6].

Input polynomial. Let $\alpha_0 \colon H^m \to \{0,1\}$ represent the string $1^n 0^{h^m - n}$, and let $\hat{\alpha}_0 \colon \mathbb{F}^m \to \mathbb{F}$ be the "arithmetization" of α_0, defined by

$$\hat{\alpha}_0(\vec{w}) = \sum_{\vec{z} \in H^{m'} \times \{0\}^{m-m'}} \delta_{\vec{z}}(\vec{w}) \cdot \alpha_0(\vec{z}).$$

Here, $m' \leq m$ is the minimal integer such that $h^{m'} \geq n$, and $\delta_{\vec{z}}$ is Kronecker's delta function (*i.e.*, $\delta_{\vec{z}}(\vec{w}) = \prod_{j \in [m]} \prod_{a \in H \setminus \{z_j\}} \frac{w_j - a}{z_j - a}$).

---

[28]Precisely, $\mathsf{TM}_{[1]}$ simulates $\mathbb{A}_{\Phi'}$ on input $\vec{\ell}$ and TM to construct a projection that maps i to $(i, F_{\Phi'})$.

Layer polynomials. For each $i \in [d_D]$, let $\alpha_i \colon H^m \to \{0,1\}$ represent the values of the gates at the ith layer of D in the computation of $D(1^n)$ (with zeroes in locations that do not index valid gates). Recall that we consider circuits consisting of NAND gates, where for $a, b \in \{0,1\}$ we have $\mathsf{NAND}(a,b) = 1 - a \cdot b$. We define $\hat{\alpha}_i \colon \mathbb{F}^m \to \mathbb{F}$ as

$$\hat{\alpha}_i(\vec{w}) = \sum_{\vec{u},\vec{v} \in H^m} \hat{\Phi}_i(\vec{w}, \vec{u}, \vec{v}) \cdot (1 - \hat{\alpha}_{i-1}(\vec{u}) \cdot \hat{\alpha}_{i-1}(\vec{v})) . \tag{5}$$

Note that $\hat{\alpha}_i$ is the "arithmetization" of α_i in the sense that for every $\vec{w} \in H^m$, $\alpha_i(\vec{w}) = \hat{\alpha}_i(\vec{w})$.

Sumcheck polynomials. For each $i \in [d_D]$, let $\hat{\alpha}_{i,0} \colon \mathbb{F}^{3m} \to \mathbb{F}$ be the polynomial

$$\hat{\alpha}_{i,0}(\vec{w}, \sigma_1, ..., \sigma_{2m}) = \hat{\Phi}_i(\vec{w}, \sigma_{1,...,m}, \sigma_{m+1,...,2m}) \cdot (1 - \hat{\alpha}_{i-1}(\sigma_{1,...,m}) \cdot \hat{\alpha}_{i-1}(\sigma_{m+1,...,2m})) . \tag{6}$$

For every $j \in [2m]$, let $\hat{\alpha}_{i,j} \colon \mathbb{F}^{3m-j} \to \mathbb{F}$ be the polynomial

$$\hat{\alpha}_{i,j}(\vec{w}, \sigma_1, ..., \sigma_{2m-j}) =$$
$$\sum_{\sigma_{2m-j+1},...,\sigma_{2m} \in H} \hat{\Phi}_i(\vec{w}, \sigma_{1,...,m}, \sigma_{m+1,...,2m}) \cdot (1 - \hat{\alpha}_{i-1}(\sigma_{1,...,m}) \cdot \hat{\alpha}_{i-1}(\sigma_{m+1,...,2m})) , \tag{7}$$

where $\sigma_{k,...,k+r} = \sigma_k, \sigma_{k+1}, ..., \sigma_{k+r}$. It is easy to check that $\hat{\alpha}_{i,2m} = \hat{\alpha}_i$.

We are now ready to define the sequence $(P_i)_{i \in [d']} = \left(P_i^{\vec{\ell},\mathsf{TM}} \right)_{i \in [d']}$. We set $d' \triangleq (2m+1) \cdot d_D + 1$ and

$$(P_i)_{i \in [d']} = (\hat{\alpha}_0, \hat{\alpha}_{1,0}, \ldots, \hat{\alpha}_{1,2m}, \hat{\alpha}_{2,0}, \ldots, \hat{\alpha}_{2,2m}, \ldots, \hat{\alpha}_{d_D,0}, \ldots, \hat{\alpha}_{d_D,2m}).$$

For those $\hat{\alpha}_{i,j}$ (and $\hat{\alpha}_0$) that take less than $3m$ variables, we add some dummy variables at the end to make all polynomials taking exactly $3m$ variables.

From the definitions of m and d_D, we have $m \leq 3 \cdot \beta\kappa \cdot \log T + 1$ and $d_D = \beta \cdot (\kappa^2 \cdot \log^2 T + d \cdot \log T)$. Hence, we have $d' = (2m+1) \cdot d_D + 1 \leq c\kappa \cdot \log^2 T \cdot (d + \kappa^2 \log T)$ as desired.[29]

Below we verify the desired properties of the sequence $(P_i)_{i \in [d']}$.

Arithmetic setting, faithful representation, and downward self-reducibility. First, the degree bounds of all these polynomials follow directly from their definitions and from the degree bound on $\hat{\Phi}_i$ (from Claim 5.3). The faithful representation property also follows directly from the definition of α_{d_D} and $\hat{\alpha}_{d_D,2m} = \hat{\alpha}_{d_D}$. Finally, the downward self-reducibility of the polynomials follows from the complexity of computing $\hat{\Phi}_i$ (from Claim 5.3) and the definitions of these polynomials, similarly to the proof of [CT21, Proposition 4.7].

5.1.3 Complexity of the Polynomials

Now we verify the complexity of computing these polynomials. The argument below is straightforward but tedious. We first give a high-level overview.

---

[29]We can add identical polynomials at the end to make d' exactly $c\kappa \cdot \log^2 T \cdot (d + \kappa^2 \log T)$ as in the theorem statement.

High-level overview of the construction. To construct the circuit C_{poly} that maps $i' \in [d']$ to $\text{tt}(P_i)$, we will construct three subcircuits C_α, $C_{\text{tt-}\hat{\Phi}}$, and C_{arith} such that:

1. C_α maps $i' \in [d']$ to $(\text{tt}(\alpha_{i-1}), i, j)$. Here, if $i' \geq 2$, then $i \in \{1, \ldots, d_D\}$ and $j \in \{0, 1, \ldots, 2m\}$ satisfies that $P_{i'} = \hat{\alpha}_{i,j}$ and $\text{tt}(\alpha_{i-1}) \in \{0, 1\}^{h^m}$ denotes the values of the gates at the i-th layer of D. If $i' = 1$, then we consider $i = j = 0$ and C_α outputs $(\text{tt}(\alpha_0), 0, 0)$.

2. $C_{\text{tt-}\hat{\Phi}}$ maps $(\text{tt}(\alpha_{i-1}), i, j)$ to $(\text{tt}(\alpha_{i-1}), i, j, \text{tt}(\hat{\Phi}_i))$.

3. C_{arith} maps $(\text{tt}(\alpha_{i-1}), i, j, \text{tt}(\hat{\Phi}_i))$ to $\text{tt}(\hat{\alpha}_{i,j}) \in \mathbb{F}^{|\mathbb{F}|^{3m}}$.

C_{poly} is then simply $C_{\text{arith}} \circ C_{\text{tt-}\hat{\Phi}} \circ C_\alpha$. To compute the Turing machine TM_{poly} that corresponds to C_{poly}, we construct the Turing machines TM_α, $\text{TM}_{\text{tt-}\hat{\Phi}}$, and TM_{arith} corresponding to the three circuit above, and compose them using Fact 2.4.

Construction of C_α and TM_α. First, we construct a circuit C_α that takes as input $i' \in [d']$ and outputs $(\text{tt}(\alpha_{i-1}), i, j)$. To construct C_α, we first compute i and j from i' using basic arithmetic, and then truncate D up to its i-th layer. It is easy to see that given the Turing machine TM_D that specifies the circuit D, in polynomial-time we can construct a Turing machine $\text{TM}_\alpha \in \{0, 1\}^{|\text{TM}_D|+\mu}$ such that (in what follows, we write $|\langle i, j \rangle| = \log(d_D + 1) + \log(2m + 1)$ for convenience):

$$\text{Circuit}[T_\alpha, s_\alpha, \log(d'), h^m + |\langle i, j \rangle|](\text{TM}_\alpha) = C_\alpha,$$

where $T_\alpha = \mu \cdot T_D$, $s_\alpha = 2s_D$. Moreover, C_α has depth at most $d_\alpha = 2 \cdot d_D$.

Construction of $C_{\text{tt-}\hat{\Phi}}$ and $\text{TM}_{\text{tt-}\hat{\Phi}}$. Let c_1 be the universal constant from Claim 5.3. Next we construct a circuit $C_{\text{tt-}\hat{\Phi}}$ that on input $(\text{tt}(\alpha_{i-1}), i, j)$, outputs $(\text{tt}(\alpha_{i-1}), i, j, \text{tt}(\hat{\Phi}_i))$. It is straightforward to obtain this circuit from the circuit $C_{\hat{\Phi}}$ constructed in Claim 5.3. In other words, given $\vec{\ell}$ and $\text{TM}_{\hat{\Phi}} \in \{0, 1\}^{c_1 \kappa \log T}$ as input (where $\text{TM}_{\hat{\Phi}}$ is the Turing machine that generates $C_{\hat{\Phi}}$ as defined in Claim 5.3), we can compute a Turing machine $\text{TM}_{\text{tt-}\hat{\Phi}} \in \{0, 1\}^{2 \cdot c_1 \kappa \log T}$ such that

$$\text{Circuit}[T_{\text{tt-}\hat{\Phi}}, s_{\text{tt-}\hat{\Phi}}, h^m + |\langle i, j \rangle|, h^m + |\langle i, j \rangle| + |\mathbb{F}|^{3m} \log |\mathbb{F}|](\text{TM}_{\text{tt-}\hat{\Phi}}) = C_{\text{tt-}\hat{\Phi}},$$

where $T_{\text{tt-}\hat{\Phi}} = T^{2c_1\kappa}$, $s_{\text{tt-}\hat{\Phi}} = 2c_1 \kappa \log T$. Moreover, $C_{\text{tt-}\hat{\Phi}}$ has depth $d_{\text{tt-}\hat{\Phi}} = 2c_1 \kappa \log T$.

Construction of C_{arith} and TM_{arith}. We construct a circuit C_{arith} that maps $(\text{tt}(\alpha_{i-1}), i, j, \text{tt}(\hat{\Phi}_i))$ to $\text{tt}(\hat{\alpha}_{i,j}) \in \mathbb{F}^{|\mathbb{F}|^{3m}}$. (Recall that throughout this proof we view $\hat{\alpha}_{i,j}$ as a $3m$-variable polynomial by adding dummy variables at the end.) Suppose that $i \geq 1$ (the base case $i = j = 0$ can be handled similarly). If $j = 0$, C_{arith} computes $\text{tt}(\hat{\alpha}_{i,0})$ using Equation 6, otherwise ($j \geq 1$) C_{arith} computes $\text{tt}(\hat{\alpha}_{i,j})$ using Equation 7. (Note that both Equation 6 and Equation 7 only depend on the values of $\hat{\alpha}_{i-1}$ over H^m, which is exactly $\text{tt}(\alpha_{i-1})$ due to our arithmetization.) Since arithmetic operations over $\mathbb{F}$ (including iterated addition, multiplication, and inverse) are in logspace-uniform NC^1 [HAB02, HV06],[30] it follows that C_{arith} is of $T_{\text{arith}} \triangleq \text{poly}(|\mathbb{F}|^m)$ size and $d_{\text{arith}} \triangleq O(m \log |\mathbb{F}|)$ depth. Moreover, C_{arith} does not depend on TM, and we can compute a Turing machine TM^{arith} from $\vec{\ell}$ in time polylog(T) such that

$$\text{Circuit}[T_{\text{arith}}, s_{\text{arith}}, h^m + |\langle i, j \rangle| + |\mathbb{F}|^{3m} \log |\mathbb{F}|, |\mathbb{F}|^{3m} \log |\mathbb{F}|](\text{TM}^{\text{arith}}) = C_{\text{arith}},$$

where $s_{\text{arith}} = \mu \cdot \beta \kappa \log T$.

Composing TM_α, $\text{TM}_{\text{tt-}\hat{\Phi}}$, and TM_{arith} by applying Fact 2.4 twice gives the desired Turing machine TM_{poly} that computes the desired circuit C_{poly}.

---

[30]It is in fact in logtime-uniform TC^0, but here we only need it to be in logspace-uniform NC^1.

Complexity of C_{poly} and $\mathsf{TM}_{\mathsf{poly}}$. Finally, we verify that $\mathsf{TM}_{\mathsf{poly}}$ and C_{poly} satisfy our requirements. First, from the discussions above, we can bound the size of C_{poly} by $T_\alpha + T_{\mathsf{tt}\text{-}\hat{\Phi}} + T_{\mathsf{arith}} \leq T_{\mathsf{poly}} = 2^{c \cdot \kappa \log T}$ by picking a sufficiently large c. Note that $m \log |\mathbb{F}| = \log(p^m) \leq O(\kappa \log T)$. The depth of C_{poly} can be bounded by $d_{\mathsf{poly}} = d_\alpha + d_{\mathsf{tt}\text{-}\hat{\Phi}} + d_{\mathsf{arith}} \leq c \cdot (\kappa^2 \cdot \log^2 T + d \cdot \log T)$ as desired.

From Fact 2.4, we have that

$$|\mathsf{TM}_{\mathsf{poly}}| \leq 100 \cdot \left(|\mathsf{TM}_\alpha| + |\mathsf{TM}_{\mathsf{tt}\text{-}\hat{\Phi}}| + |\mathsf{TM}_{\mathsf{arith}}| + \log(|\mathbb{F}|^{3m})\right) \leq c \cdot \kappa \log T = \log T_{\mathsf{poly}}$$

by setting c sufficiently large. The space complexity of $\mathsf{TM}_{\mathsf{poly}}$ can be bounded by

$$100 \cdot \left(s_\alpha + s_{\mathsf{tt}\text{-}\hat{\Phi}} + s_{\mathsf{arith}}\right) \leq c \cdot \kappa \log T = \log T_{\mathsf{poly}}$$

as well. This completes the proof. $\qquad\square$

5.2 Improved Chen–Tell Generator: Proof of Theorem 3.1

Now we are ready to prove Theorem 3.1 by plugging every polynomial from Theorem 5.1 into our modified Shaltiel–Umans generator (Theorem 4.1).

Proof of Theorem 3.1. We first observe that we can assume $\rho = 1$ without loss of generality. To see how the general case follows from the case that $\rho = 1$, letting $M' = M^\rho$, we can simply define $\mathsf{H}^{\mathsf{ct}}_{n,T,d,M,\kappa,\rho}$ as the set of strings obtained by truncating every string from $\mathsf{H}^{\mathsf{ct}}_{n,T,d,M',\kappa,1}$ to their first M bits. The reconstruction algorithm $\mathsf{R}^{\mathsf{ct}}_{n,T,d,M,\kappa,\rho}$ can then be obtained by slightly modifying $\mathsf{R}^{\mathsf{ct}}_{n,T,d,M',\kappa,1}$.

Let

$$\vec{\ell}_{\mathsf{ct}} = (n, T, d, M, \kappa, 1)$$

be the input parameters from the theorem statement and c be a sufficiently large universal constant. From the assumption, we have $n \leq T, d \leq T$, and $c \cdot \log T \leq M \leq T^{1/c}$. Let

$$C_{\mathsf{TM}} = \mathsf{Circuit}[T, \kappa \cdot \log T, n, n](\mathsf{TM}).$$

The layered-polynomial representation. Let c_0, c_0', β be the universal constants from Theorem 5.1. Let h be the *smallest* nice power of 2 such that $h \geq M$, $p \triangleq h^{27}$, m be the smallest power of 3 such that $h^m \geq T^{\beta\kappa}$, and $\mathbb{F} = \mathbb{F}_p$. Note that p is also a nice power of 2 and $h \leq M^3$.

We will invoke Theorem 5.1 with input parameters

$$\vec{\ell}_{\mathsf{poly}} = (\kappa, T, d, n, h, p).$$

Note that from their definitions and our assumption $M \geq c \log T$, we have $\log T \leq h < p \leq h^{27} \leq M^{81} \leq T$ (assuming $c \geq 81$ is large enough), meaning that the requirements on the input parameters of Theorem 5.1 are satisfied.

We first apply Theorem 5.1 with input parameters $\vec{\ell}_{\mathsf{poly}}$ and Turing machine TM to obtain $d' = c_0\kappa \cdot \log^2 T \cdot (d + \kappa^2 \log T)$ polynomials $(P_i)_{i \in [d']} = \left(P_i^{\vec{\ell},\mathsf{TM}}\right)_{i \in [d']}$.

Hitting set H^{ct}. Let $\mathsf{H}^{\mathsf{layer}}$ and $\mathsf{R}^{\mathsf{layer}}$ denote the H and R algorithms from Theorem 4.1, respectively.[31] Let $\Delta = c_0 h \log^3(T)$,

$$\vec{\ell}_{\mathsf{layer}} = (p, 3m, M, \Delta)$$

---

[31] The superscript layer highlights the fact that they are applied to each layer of the polynomial representation of the circuit.

be the input parameters when applying Theorem 4.1. We can verify that $p > \Delta^2(3m)^7 M^9$, *i.e.*, the requirement on the input parameters of Theorem 4.1 is satisfied.

We then define $\mathsf{H}^{\mathsf{ct}}_{\vec{\ell}_{\mathsf{ct}}}(\mathsf{TM})$ as the union of $\mathsf{H}^{\mathsf{layer}}_{\vec{\ell}_{\mathsf{layer}}}(P_i)$ for every $i \in [d']$. Next we analyze the complexity of computing $\mathsf{H}^{\mathsf{ct}}_{\vec{\ell}_{\mathsf{ct}}}(\mathsf{TM})$. First, from Theorem 5.1, letting $T_{\mathsf{poly}} = T^{c_0 \cdot \kappa}$ and $d_{\mathsf{poly}} = c_0 \cdot (d \log T + \kappa^2 \log^2 T)$, there is a polynomial-time algorithm $\mathbb{A}^{\mathsf{poly}}_{\vec{\ell}}$ that takes $\mathsf{TM} \in \{0,1\}^{\kappa \log T}$ as input, and outputs a description of Turing machine $\mathsf{TM}_{\mathsf{poly}} \in \{0,1\}^{\log T_{\mathsf{poly}}}$ such that for

$$C_{\mathsf{poly}} = \mathsf{Circuit}\big[T_{\mathsf{poly}}, \log T_{\mathsf{poly}}, \log d', |\mathbb{F}|^{3m} \cdot \log |\mathbb{F}|\big](\mathsf{TM}_{\mathsf{poly}})$$

it holds that (1) for every $i \in [d']$ $C_{\mathsf{poly}}(i) = \mathtt{tt}(P_i)$ and (2) C_{poly} has depth d_{poly}.

Second, from Theorem 4.1, there is a logspace-uniform circuit family with input parameters $\vec{\ell}_{\mathsf{layer}}$, size $\mathrm{poly}(p^m)$, and depth $\mathrm{poly}(\log p, m, M)$ such that for every $i \in [d']$, it outputs $\mathsf{H}^{\mathsf{layer}}_{\vec{\ell}_{\mathsf{layer}}}(P_i)$ when taking $\mathtt{tt}(P_i)$ as input. Note that $\mathrm{poly}(p^m) \le T^{O(\beta\kappa)}$ and $\mathrm{poly}(\log p, m, M) \le \mathrm{poly}(M)$. Applying Fact 2.4 to compose the machines above and enumerating over all $i \in [d']$,[32] we obtain the desired circuit C_{H} (note that c is sufficiently large).

Reconstruction R^{ct}. For every $i \in \{2, \ldots, d'\}$, the reconstruction algorithm R^{ct} attempts to construct a $\mathrm{poly}(p, m, \log(Md'))$-size D-oracle circuit E_i that computes P_i. A formal description of R^{ct} is as follows:

- We start with the circuit $E_1(\vec{x}) = \mathsf{Base}(\vec{\ell}, \mathsf{TM}, \vec{x})$ that computes the polynomial P_1.

- For every $i \in \{2, \ldots, d'\}$:

  1. We first construct a procedure $\widetilde{P}_i$ computing P_i using the D-oracle circuit E^D_{i-1} for P_{i-1} and the downward self-reducibility for P_i. In particular, on input $\vec{x} \in \mathbb{F}^{3m}$, let

     $$\widetilde{P}_i(\vec{x}) \triangleq \mathsf{DSR}^{E^D_{i-1}}(\vec{\ell}, \mathsf{TM}, i, \vec{x}).$$

  2. Run $\big(\mathsf{R}^{\mathsf{layer}}\big)^{D, \widetilde{P}_i}_{\vec{\ell}_{\mathsf{layer}}}$ which outputs a D-oracle circuit $\widetilde{E}^D_i$ in $\mathrm{poly}(p, m, M)$ time.

  3. Let $t \triangleq c_1 \cdot m \cdot \log p$ for a sufficiently large constant $c_1 > 1$. Take t i.i.d. samples $\vec{x}_1, \ldots, \vec{x}_t$ from $\mathbb{F}^{3m}$. Check that for every $j \in [t]$, $\widetilde{E}^D_i(\vec{x}_j) = \widetilde{P}_i(\vec{x}_j)$. If any condition does not hold, the algorithm outputs $\perp$ and aborts immediately.

  4. Let E_i be a D-oracle circuit constructed as follows:
     (a) Draw $t = \Theta(m \log p)$ i.i.d. samples of random strings $r_1, \ldots, r_t$ used by PCorr. (Recall that PCorr is the self-corrector for low-degree polynomials in Theorem 4.8.)
     (b) Set $E_i(\vec{x}) = \mathsf{MAJ}_{k \in [t]} \mathsf{PCorr}^{\widetilde{E}_i}(p, 3m, \Delta, \vec{x}; r_k)$ for all $\vec{x} \in \mathbb{F}^{3m}$.

- For every $j \in [n]$, output $E^D_{d'}(\mathsf{id}(j), 0^{2m})$.

For ease of notation, for every $i' \in \{2, \ldots, d'\}$, we use $\tau_{i'}$ to denote the randomness used when running the algorithm above with $i = i'$, and we use $\tau_{\le i}$ to denote $\tau_1, \ldots, \tau_i$. Also, if E_i is not constructed by the algorithm (meaning that the algorithm aborts before constructing E_i), we set $E_i = \perp$.

---

[32]Enumerating all $i \in [d']$ only adds a $O(\log d')$ additive overhead in depth and a $O(d')$ multiplicative blowup in size, which are negligible.

From Theorem 4.8, Theorem 4.1, and Theorem 5.1, the running time of the algorithm above can be bounded by

$$\mathrm{poly}(p, m, h, \log(Md')) \cdot (d' + n) \leq \mathrm{poly}(M) \cdot (d' + n) \leq \mathrm{poly}(M) \cdot (d + n).$$

The last inequality follows from the fact that $M \geq \log T$ and hence $d' = c_0 \kappa \cdot \log^2 T \cdot (d + \kappa^2 \log T) \leq \mathrm{poly}(M) \cdot d$. Now we establish the soundness and completeness of the reconstruction. We show the following claim.

Claim 5.4. *Fix* $D \colon \{0,1\}^M \to \{0,1\}$. *For every* $i \in \{2, \ldots, d'\}$, *for every fixed* $\tau_{\leq i-1}$, *if* E_{i-1}^D *computes* P_{i-1} *or* $i = 2$,[33] *then with probability at least* $1 - 1/p^m$ *over* τ_i *the following holds:*

- **(Soundness.)** *If* $E_i \neq \bot$, *then* E_i^D *computes* P_i.

- **(Completeness.)** *If* D $(1/M)$-*avoids* $\mathsf{H}^{\mathsf{layer}}_{\vec{\ell}_{\mathsf{layer}}}(P_i)$, *then* E_i^D *computes* P_i.

Before establishing the claim, we show it implies the completeness and soundness of the reconstruction. To see the soundness, note that by induction over all $i \in \{2, \ldots, d'\}$, with probability at least $1 - d'/p^m > 9/10$, it holds that if $E_{d'} \neq \bot$, then $E_{d'}$ computes $P_{d'}$, meaning the reconstruction outputs the correct output $C_{\mathsf{TM}}(1^n)$. To see the completeness, note that an oracle $D \colon \{0,1\}^M \to \{0,1\}$ that $(1/M)$-avoids $\mathsf{H}^{\mathsf{ct}}_{\vec{\ell}}(\mathsf{TM})$ also $(1/M)$-avoids $\mathsf{H}^{\mathsf{layer}}_{\vec{\ell}_{\mathsf{layer}}}(P_i)$ for every $i \in [d']$. Hence, by induction over $i \in \{2, \ldots, d'\}$, with probability at least $1 - d'/p^m > 9/10$, it holds that E_i computes P_i for every $i \in \{2, \ldots, d'\}$. Thus the reconstruction will output $C_{\mathsf{TM}}(1^n)$. The success probability $9/10$ can be amplified to $1 - 2^{-M}$ by running the reconstruction algorithm $O(M)$ times independently and outputting the answer that occurs most frequently.

Finally, we prove the claim.

Proof of Claim 5.4. We first establish the soundness. From the assumption that E_{i-1}^D computes P_{i-1} or $i = 2$ and the downward self-reducibility property of Theorem 5.1, it follows that $\widetilde{P}_i$ computes P_i. Therefore, $E_i \neq \bot$ means that $\widetilde{E}_i$ has passed the test in Step 3, meaning that with probability at least $1 - p^{-4m}$ over the randomness in Step 3, it holds that $\widetilde{E}_i$ agrees P_i on at least $3/4$ fraction of inputs from $\mathbb{F}^{3m}$. This then means that with probability at least $1 - p^{-3m}$ over the randomness in Step 4(a), we have E_i^D computes P_i.

The completeness follows immediately from Theorem 4.1. (Here $\widetilde{E}_i^D$ already computes P_i with probability at least $1 - 1/p^m$.) ◇

This completes the proof of Theorem 3.1. □

Acknowledgments

---

[33]Note that $\tau_{\leq i-1}$ determines E_{i-1}.

References

[AB09]   Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach.* Cambridge University Press, 2009. 15, 26

[AKS04]   Manindra Agrawal, Neeraj Kayal, and Nitin Saxena. PRIMES is in P. *Annals of Mathematics*, 160(2):781–793, 2004. `doi:10.4007/annals.2004.160.781`. 3, 4, 5

[BB18]   Ben Berger and Zvika Brakerski. Zero-knowledge protocols for search problems. In *International Conference on Security and Cryptography for Networks* (SCN), pages 292–309, 2018. `doi:10.1007/978-3-319-98113-0\_16`. 3

[BHP01]   R. C. Baker, G. Harman, and J. Pintz. The difference between consecutive primes. II. *Proc. London Math. Soc. (3)*, 83(3):532–562, 2001. `doi:10.1112/plms/83.3.532`. 3

[BKKS23]   Vladimir Braverman, Robert Krauthgamer, Aditya Krishnan, and Shay Sapir. Lower bounds for pseudo-deterministic counting in a stream. *CoRR*, abs/2303.16287, 2023. `doi:10.48550/arXiv.2303.16287`. 3

[BM84]   Manuel Blum and Silvio Micali. How to generate cryptographically strong sequences of pseudo-random bits. *SIAM J. Comput.*, 13(4):850–864, 1984. `doi:10.1137/0213053`. 14, 26

[CDM23]   Arkadev Chattopadhyay, Yogesh Dahiya, and Meena Mahajan. Query complexity of search problems. *Electronic Colloquium on Computational Complexity* (ECCC), 2023. URL: `https://eccc.weizmann.ac.il/report/2023/039/`. 3

[CPW23]   Suvradip Chakraborty, Manoj Prabhakaran, and Daniel Wichs. A map of witness maps: New definitions and connections. Cryptology ePrint Archive, Paper 2023/343, 2023. URL: `https://eprint.iacr.org/2023/343`. 3

[Cra36]   Harald Cramér. On the order of magnitude of the difference between consecutive prime numbers. *Acta Arithmetica*, 2:23–46, 1936. 3

[CRT22]   Lijie Chen, Ron D. Rothblum, and Roei Tell. Unstructured hardness to average-case randomness. In *63rd IEEE Annual Symposium on Foundations of Computer Science, FOCS 2022, Denver, CO, USA, October 31 - November 3, 2022*, pages 429–437. IEEE, 2022. `doi:10.1109/FOCS54457.2022.00048`. 7

[CT21]   Lijie Chen and Roei Tell. Hardness vs randomness, revised: Uniform, non-black-box, and instance-wise. In *IEEE Symposium on Foundations of Computer Science* (FOCS), pages 125–136, 2021. `doi:10.1109/FOCS52979.2021.00021`. 1, 6, 7, 8, 9, 10, 11, 12, 18, 27, 35, 36, 37, 38, 39, 40

[DPV18]   Peter Dixon, Aduri Pavan, and N. V. Vinodchandran. On pseudodeterministic approximation algorithms. In *Symposium on Mathematical Foundations of Computer Science* (MFCS), pages 61:1–61:11, 2018. `doi:10.4230/LIPIcs.MFCS.2018.61`. 3

[DPV21]   Peter Dixon, Aduri Pavan, and N. V. Vinodchandran. Complete problems for multi-pseudodeterministic computations. In *Innovations in Theoretical Computer Science* (ITCS), 2021. `doi:10.4230/LIPIcs.ITCS.2021.66`. 3

[DPWV22]   Peter Dixon, Aduri Pavan, Jason Vander Woude, and N. V. Vinodchandran. Pseudodeterminism: promises and lowerbounds. In *ACM Symposium on Theory of Computing* (STOC), pages 1552–1565, 2022. `doi:10.1145/3519935.3520043`. 3

[GG11]   Eran Gat and Shafi Goldwasser. Probabilistic search algorithms with unique answers and their cryptographic applications. *Electronic Colloquium on Computational Complexity* (ECCC), 18:136, 2011. URL: `https://eccc.weizmann.ac.il/report/2011/136/`. 1, 3

[GG17]   Shafi Goldwasser and Ofer Grossman. Bipartite perfect matching in pseudo-deterministic NC. In *International Colloquium on Automata, Languages, and Programming* (ICALP), pages 87:1–87:13, 2017. `doi:10.4230/LIPIcs.ICALP.2017.87`. 3

[GG21]   Sumanta Ghosh and Rohit Gurjar. Matroid intersection: A pseudo-deterministic parallel reduction from search to weighted-decision. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques* (APPROX/RANDOM), pages 41:1–41:16, 2021. `doi:10.4230/LIPIcs.APPROX/RANDOM.2021.41`. 3

[GGH18]   Shafi Goldwasser, Ofer Grossman, and Dhiraj Holden. Pseudo-deterministic proofs. In *Innovations in Theoretical Computer Science,* (ITCS), pages 17:1–17:18, 2018. `doi:10.4230/LIPIcs.ITCS.2018.17`. 3

[GGH19]   Michel X. Goemans, Shafi Goldwasser, and Dhiraj Holden. Doubly-efficient pseudo-deterministic proofs. *Electronic Colloquium on Computational Complexity* (ECCC), 26:135, 2019. URL: `https://eccc.weizmann.ac.il/report/2019/135`. 3

[GGMW20]   Shafi Goldwasser, Ofer Grossman, Sidhanth Mohanty, and David P. Woodruff. Pseudo-deterministic streaming. In *Innovations in Theoretical Computer Science* (ITCS), pages 79:1–79:25, 2020. `doi:10.4230/LIPIcs.ITCS.2020.79`. 3

[GGR13]   Oded Goldreich, Shafi Goldwasser, and Dana Ron. On the possibilities and limitations of pseudodeterministic algorithms. In *Innovations in Theoretical Computer Science* (ITCS), pages 127–138, 2013. `doi:10.1145/2422436.2422453`. 3

[GIPS21]   Shafi Goldwasser, Russell Impagliazzo, Toniann Pitassi, and Rahul Santhanam. On the pseudo-deterministic query complexity of NP search problems. In *Computational Complexity Conference* (CCC), pages 36:1–36:22, 2021. `doi:10.4230/LIPIcs.CCC.2021.36`. 3

[GKR15]   Shafi Goldwasser, Yael Tauman Kalai, and Guy N. Rothblum. Delegating computation: interactive proofs for muggles. *Journal of the ACM*, 62(4):27:1–27:64, 2015. `doi:10.1145/2699436`. 11

[GL89]   Oded Goldreich and Leonid A. Levin. A hard-core predicate for all one-way functions. In *ACM Symposium on Theory of Computing* (STOC), pages 25–32, 1989. `doi:10.1145/73007.73010`. 26, 28

[GL19]   Ofer Grossman and Yang P. Liu. Reproducibility and pseudo-determinism in Log-Space. In *Symposium on Discrete Algorithms* (SODA), pages 606–620, 2019. `doi:10.1137/1.9781611975482.38`. 3

[Gol08]   Oded Goldreich. *Computational complexity: a conceptual perspective*. Cambridge University Press, 2008. `doi:10.1017/CBO9780511804106`. 15

[Gol17]   Oded Goldreich. On the doubly-efficient interactive proof systems of GKR. *Electronic Colloquium on Computational Complexity* (ECCC), 24:101, 2017. URL: `https://eccc.weizmann.ac.il/report/2017/101`. 11, 36, 37

[Gol18]   Oded Goldreich. On doubly-efficient interactive proof systems. *Foundations and Trends® in Theoretical Computer Science*, 13(3):158–246, 2018. `doi:10.1561/0400000084`. 36

[Gol19]   Oded Goldreich. Multi-pseudodeterministic algorithms. *Electronic Colloquium on Computational Complexity* (ECCC), 26:12, 2019. URL: `https://eccc.weizmann.ac.il/report/2019/012`. 3

[Gro15]   Ofer Grossman. Finding primitive roots pseudo-deterministically. *Electronic Colloquium on Computational Complexity* (ECCC), 22:207, 2015. URL: `https://eccc.weizmann.ac.il/report/2015/207`. 3

[GS92]   Peter Gemmell and Madhu Sudan. Highly resilient correctors for polynomials. *Inf. Process. Lett.*, 43(4):169–174, 1992. `doi:10.1016/0020-0190(92)90195-2`. 27

[HAB02]   William Hesse, Eric Allender, and David A. Mix Barrington. Uniform constant-depth threshold circuits for division and iterated multiplication. *J. Comput. Syst. Sci.*, 65(4):695–716, 2002. `doi:10.1016/S0022-0000(02)00025-9`. 38, 41

[HV06]   Alexander Healy and Emanuele Viola. Constant-depth circuits for arithmetic in finite fields of characteristic two. In *Symposium on Theoretical Aspects of Computer Science* (STACS), volume 3884 of *Lecture Notes in Computer Science*, pages 672–683. Springer, 2006. `doi:10.1007/11672142\_55`. 38, 41

[IKW02]   Russell Impagliazzo, Valentine Kabanets, and Avi Wigderson. In search of an easy witness: exponential time vs. probabilistic polynomial time. *J. Comput. Syst. Sci.*, 65(4):672–694, 2002. `doi:10.1016/S0022-0000(02)00024-7`. 6

[IW97]   Russell Impagliazzo and Avi Wigderson. P = BPP if E requires exponential circuits: Derandomizing the XOR lemma. In *ACM Symposium on Theory of Computing* (STOC), pages 220–229. ACM, 1997. `doi:10.1145/258533.258590`. 3

[IW01]   Russell Impagliazzo and Avi Wigderson. Randomness vs time: Derandomization under a uniform assumption. *Journal of Computer and System Sciences*, 63(4):672–688, 2001. `doi:10.1006/jcss.2001.1780`. 5, 6, 11

[Kan82]   Ravi Kannan. Circuit-size lower bounds and non-reducibility to sparse sets. *Inf. Control.*, 55(1-3):40–56, 1982. `doi:10.1016/S0019-9958(82)90382-5`. 6

[KL80]   Richard M. Karp and Richard J. Lipton. Some connections between nonuniform and uniform complexity classes. In *ACM Symposium on Theory of Computing* (STOC), pages 302–309, 1980. `doi:10.1145/800141.804678`. 6

[LO87]   J. C. Lagarias and Andrew M. Odlyzko. Computing $\pi(x)$: An analytic method. *J. Algorithms*, 8(2):173–191, 1987. `doi:10.1016/0196-6774(87)90037-X`. 3

[LO22]   Zhenjian Lu and Igor C. Oliveira. Theory and applications of probabilistic Kolmogorov complexity. *Bull. EATCS*, 137, 2022. URL: `http://bulletin.eatcs.org/index.php/beatcs/article/view/700`. 5

[LOS21]   Zhenjian Lu, Igor C. Oliveira, and Rahul Santhanam. Pseudodeterministic algorithms and the structure of probabilistic time. In *ACM Symposium on Theory of Computing* (STOC), pages 303–316, 2021. `doi:10.1145/3406325.3451085`. 3, 4, 5

[LY22]   Jiatu Li and Tianqi Yang. $3.1n - o(n)$ circuit lower bounds for explicit functions. In *STOC*, pages 1180–1193. ACM, 2022. `doi:10.1145/3519935.3519976`. 3

[MVW99]   Peter Bro Miltersen, N. V. Vinodchandran, and Osamu Watanabe. Super-polynomial versus half-exponential circuit size in the exponential hierarchy. In *International Computing and Combinatorics Conference* (COCOON), volume 1627 of *Lecture Notes in Computer Science*, pages 210–220. Springer, 1999. `doi:10.1007/3-540-48686-0\_21`. 6

[NW94]   Noam Nisan and Avi Wigderson. Hardness vs randomness. *Journal of Computer and System Sciences*, 49(2):149–167, 1994. `doi:10.1016/S0022-0000(05)80043-1`. 7, 11

[Oli19]   Igor C. Oliveira. Randomness and intractability in Kolmogorov complexity. In *International Colloquium on Automata, Languages, and Programming* (ICALP), pages 32:1–32:14, 2019. `doi:10.4230/LIPIcs.ICALP.2019.32`. 3

[OS17]   Igor C. Oliveira and Rahul Santhanam. Pseudodeterministic constructions in subexponential time. In *Symposium on Theory of Computing* (STOC), pages 665–677, 2017. `doi:10.1145/3055399.3055500`. 1, 3, 4, 5, 6, 7, 8, 9, 10

[OS18]   Igor C. Oliveira and Rahul Santhanam. Pseudo-derandomizing learning and approximation. In *International Conference on Randomization and Computation* (RANDOM), pages 55:1–55:19, 2018. `doi:10.4230/LIPIcs.APPROX-RANDOM.2018.55`. 3

[Sho92]   Victor Shoup. Searching for primitive roots in finite fields. *Mathematics of Computation*, 58(197):369–380, January 1992. `doi:10.1090/S0025-5718-1992-1106981-9`. 13, 24

[STV01]   Madhu Sudan, Luca Trevisan, and Salil P. Vadhan. Pseudorandom generators without the XOR lemma. *J. Comput. Syst. Sci.*, 62(2):236–266, 2001. `doi:10.1006/jcss.2000.1730`. 11

[SU05]   Ronen Shaltiel and Christopher Umans. Simple extractors for all min-entropies and a new pseudorandom generator. *J. ACM*, 52(2):172–216, 2005. `doi:10.1145/1059513.1059516`. 1, 7, 12, 13, 18, 25, 27, 28, 29, 30

[Sud95]   Madhu Sudan. *Efficient Checking of Polynomials and Proofs anf the Hardness of Approximation Problems*, volume 1001 of *Lecture Notes in Computer Science*. Springer, 1995. `doi:10.1007/3-540-60615-7`. 27

[Sud97]   Madhu Sudan. Decoding of Reed Solomon codes beyond the error-correction bound. *J. Complex.*, 13(1):180–193, 1997. `doi:10.1006/jcom.1997.0439`. 24

[TCH12]   Terence Tao, Ernest Croot, III, and Harald Helfgott. Deterministic methods to find primes. *Math. Comp.*, 81(278):1233–1246, 2012. `doi:10.1090/S0025-5718-2011-02542-1`. 3

[TV07]   Luca Trevisan and Salil P. Vadhan. Pseudorandomness and average-case complexity via uniform reductions. *Computational Complexity*, 16(4):331–364, 2007. `doi:10.1007/s00037-007-0233-x`. 5, 6

[VL99]   Jacobus Hendricus Van Lint. *Introduction to coding theory*, volume 86. Springer-Verlag Berlin Heidelberg, 1999. 15

[WDP+22]   Jason Vander Woude, Peter Dixon, A. Pavan, Jamie Radcliffe, and N. V. Vinodchandran. The geometry of rounding. *Electronic Colloquium on Computational Complexity* (ECCC), TR22-160, 2022. URL: `https://eccc.weizmann.ac.il/report/2022/160`. 3

[Yao82]   Andrew Chi-Chih Yao. Theory and applications of trapdoor functions (extended abstract). In *IEEE Symposium on Foundations of Computer Science* (FOCS), pages 80–91, 1982. `doi:10.1109/SFCS.1982.45`. 14, 26