Electronic Colloquium on Computational Complexity, Report No. 176 (2026)

Parallel Kolmogorov Complexity with Connections to Learning Theory and Cryptography

Marco Carmosino
IBM Research
mlc@ibm.com

Mandar Juvekar
Boston University
mandarj@bu.edu

September 4, 2026

Abstract

We introduce a new family of *parallel* time-bounded Kolmogorov complexity measures (KP^t). A string w has low KP^t complexity if any individual bit of w can be decompressed from a "short" description using "few" parallel processors within at most t steps.

The definition of KP^t uses Parallel Random Access Machines (PRAMs) instead of Turing Machines. Consequently, KP^t is closely related to constant-depth circuit complexity. By varying the PRAM instruction set, we define a variant of KP^t for each standard constant-depth circuit complexity class: $\mathsf{AC}^0, \mathsf{AC}^0[p], \mathsf{ACC}^0$, and TC^0. We show that for each variant, the KP^t complexity of a string is polynomially related to the minimum size of a circuit in the corresponding circuit class that computes the indexing function for the string. This is directly analogous to the close relationship between the time-bounded Kolmogorov complexity KT and circuit size due to Allender et al. (SIAM J. Comp. 2006).

We show that some recently-discovered connections between time-bounded Kolmogorov complexity, learning theory, and cryptography "scale down" to analogous implications for KP.

1. If approximating KP^t is easy on average, then functions computed by constant-depth circuits are learnable from random examples over the uniform distribution. On the other hand, if functions computed by constant-depth circuits are learnable from random examples, then KP^t is easy on average to approximate.
2. If approximating KP^t is hard on average, then constant-depth circuits compute secure pseudorandom functions. On the other hand, if constant-depth circuits compute secure pseudorandom functions, then $\mathsf{KP}^{t'}$ is hard on average to approximate for some $t' = O(\log n)$.

Complementing these implications, we observe that the natural proofs of lower bounds against constant-depth circuit complexity classes AC^0 and $\mathsf{AC}^0[p]$ imply approximation algorithms for the corresponding variants of KP^t. This approximation guarantee is insufficient to obtain new learning algorithms for $\mathsf{AC}^0[p]$, but does show that KP^t is strictly easier to approximate than standard time-bounded Kolmogorov complexity (under cryptographic assumptions).

ISSN 1433-8092

Contents

1 Introduction **1**
- 1.1 The Measures KP . . . 3
- 1.2 Connections to Learning . . . 6
- 1.3 Connections to Constant-Depth Pseudorandom Functions . . . 7
- 1.4 Algorithms for KP-vs-Random$_{\mathsf{MOD}p}$[polylog] . . . 9
- 1.5 Related Work . . . 9
- 1.6 Future Directions . . . 10

2 Preliminaries **11**
- 2.1 Circuit Model . . . 11
- 2.2 Learning and the Direct-Product Generator . . . 12
- 2.3 Average-Case Complexity . . . 13
- 2.4 The Nisan–Wigderson Generator . . . 13
- 2.5 Cryptography . . . 14

3 (Parallel) Random Access Machines **15**
- 3.1 Finite Oracles . . . 15
- 3.2 Random Access Machines . . . 15
- 3.3 Parallel Random Access Machines . . . 15

4 Resource-Bounded Kolmogorov Complexity on $\mathcal{B}$-PRAMs **18**
- 4.1 Basic Properties of KP . . . 20
- 4.2 Computational Problems Based on KP$_h$ . . . 21

5 KP$_h$ and AC^0[h] Complexity Are Polynomially Related **21**
- 5.1 Simulating circuits with PRAMs . . . 21
- 5.2 Simulating PRAMs with circuits . . . 23
- 5.3 A Nontrivial Algorithm for KP-vs-Random$_{\mathsf{MOD}p}$ . . . 28

6 Connections to Learning **29**
- 6.1 Algorithms for KP Imply Learning . . . 29
- 6.2 Learning Implies Algorithms for KP . . . 32

7 Connections to Constant-Depth Cryptography **34**
- 7.1 Average-Case* Hardness . . . 34
- 7.2 A PRF from Hardness of KP-vs-Random . . . 37
- 7.3 Hardness of KP-vs-Random from Constant-Depth PRFs . . . 41

A Proof of Lemma 2.5 **50**

1 Introduction

The *time-bounded Kolmogorov complexity* of a string w, denoted $\mathsf{K}^t(w)$, is the description length of the smallest Turing Machine that prints w within at most t steps. Strings of low time-bounded Kolmogorov complexity must contain "realizable patterns" that can be generated by a "short" program using "few" steps of computation, so $\mathsf{K}^t(w)$ measures the time-bounded *compressibility* of w. The complexity of computing K^t and similar compressibility measures has been studied since the Cybernetics Program in 1960s Russia [Tra84]. In decades since, researchers have uncovered many deep connections between the complexity of computing K^t and fundamental open questions about learning theory, cryptography, and computational complexity (surveyed by [All17, All20, MP25]).

In general, *hardness* of approximating K^t implies the existence of secure cryptography, while *easiness* of approximating K^t implies the feasibility of accurate learning. Such implications hold for a wide variety of compressibility measures, cryptographic primitives, and learning tasks. For example, recent breakthroughs characterized both the existence of *polynomial-time one-way functions* and feasibility of *average-case universal inductive inference* by the complexity of approximating K^t on average [LP20, HN23]. Some results scale down to weaker models of computation: the existence of *one-way functions in* NC^0 is characterized by the complexity of approximating a variant of K^t [RS21].

However, many important open problems still lack a characterization in terms of resource-bounded Kolmogorov complexity, including the learnability of *constant-depth circuits* and existence of *parallel pseudorandom functions* (PRFs). Such problems are particularly interesting because they constitute the research frontier between learning and cryptography: as the complexity of target concepts and cryptographic primitives scales down, questions about learnability and cryptanalysis become more tractable. To better understand the resulting landscape in terms of compressibility measures, we:

1. Introduce a new family of parallel resource-bounded Kolmogorov complexity measures (KP^t) with fine-grained connections to constant-depth circuit complexity. Intuitively, $\mathsf{KP}^t(w)$ is the minimum number $|M| + p$ such that a *Parallel Random Access Machine* M can print any individual bit of w within t steps and using at most p processors (§1.1).

2. Show that each measure $\mathsf{KP}^t(w)$ is polynomially related to the minimum size of a circuit in the corresponding circuit class that, given i, computes the ith bit of w (Theorem 1.6).

3. Prove implications between the complexity of approximating KP^t, the feasibility of learning constant-depth circuit classes (§1.2), and the existence of constant-depth PRFs (§1.3) by scaling down prior results.

4. Observe that, for the KP^t measures corresponding to $\mathsf{AC}^0[p]$ circuits, an algorithm deciding if $\mathsf{KP}^t(w) \overset{?}{\leq} 2^{\log^\varepsilon |w|}$ follows from the natural proofs of circuit lower bounds for $\mathsf{AC}^0[p]$ (§1.4). This approximation guarantee is insufficient to obtain new learning algorithms for $\mathsf{AC}^0[p]$, but does show that (under cryptographic assumptions) KP^t is strictly easier to approximate than standard time-bounded Kolmogorov complexity.

The main open problems are to design better approximation algorithms for variants of KP^t complexity and so obtain new learning algorithms, or give formal evidence for the hardness of approximating KP^t and so obtain constant-depth PRFs that are secure under standard assumptions. Before describing our contributions in detail, we discuss the analogous implications from prior work.

Approximating K^t is easy on average $\implies$ efficient learner for poly-size circuits. Valiant's *Probably Approximately Correct* (PAC) setting models the task of learning to classify according to an unknown element of a fixed *concept class*—an infinite collection of Boolean functions [Val84]. A PAC-learner for the concept class $\mathcal{C}$ must, for *any* unknown concept $f \in \mathcal{C}$ and *any* unknown distribution $\mathcal{D}$ over the inputs to f, output with high probability a hypothesis h that has high agreement with f over fresh samples from $\mathcal{D}$. The learner is given independent and identically distributed (iid) examples labeled $(x, f(x))$ where x is drawn from $\mathcal{D}$. This stringent learning requirement demands a worst-case guarantee about both the unknown concept and distribution. Accordingly, efficient PAC-learners are only known for simple concept classes, such as linear threshold functions, decision lists, and parity functions [ST17].

But if approximating K^t is easy, then an even more demanding learning task can be solved. Solomonoff's *Inductive Inference* models sequential prediction: given a sequence of symbols $x_1, \ldots, x_m$ generated by an unknown randomized Turing machine M, the task is to sample from a distribution over symbols that is close to the output distribution of M conditioned on having generated $x_1, \ldots, x_m$. Recently, Hirahara and Nanashima showed that *time-bounded* formulations of Inductive Inference can be efficiently reduced to approximating K^t [HN23, HN26]. The reductions of Hirahara and Nanashima rely on an intricate analysis of K^t complexity that may not scale down to handle KP^t. So our connections between KP^t complexity and learning adapt an earlier result with a simpler analysis.

Goldberg and Kabanets showed that if approximating K^t is easy, then there is an efficient PAC-learner for polynomial-sized Boolean circuits that works over any *efficiently samplable* distribution on inputs [GK23]. This would constitute a best-case scenario for PAC-learning algorithms: polynomial-sized Boolean circuits can express any "feasible concept", and the assumption that "natural observations come from feasibly samplable distributions" is strikingly minimal. Goldberg and Kabanets' result strengthened known connections between compressibility measures and PAC-learning tasks because they reduced PAC-learning from *random examples* to approximating K^t. Previously, only a reduction from PAC-learning with *membership queries* to approximating minimum circuit size was known [CIKK16].

Their reduction uses a generalization of Yao's connection between pseudorandomness and (un)predictability [Yao82] and relies on K^t to distinguish between random and compressible strings. However, it is unlikely that this reduction will yield new learning algorithms directly, because K^t is hard to approximate if computationally secure cryptography is at all possible (i.e., if one-way functions exist [IL89]). Indeed, Goldberg and Kabanets remarked that "it is difficult to get meaningful circuit class restrictions when talking about [time-bounded Kolmogorov Complexity], and utilize the known circuit lower bound proofs for these restricted circuit classes in order to derive a learning algorithm." Our new family of Kolmogorov complexity measures directly addresses this issue.

Approximating K^t is hard on average $\implies$ secure PRFs in poly-time. In a breakthrough result, Liu and Pass showed that average-case hardness of approximating K^t is *equivalent* to the existence of (polynomial-time computable) cryptographic one-way functions [LP20]. Extensions of that work showed that by varying the parameters in the hardness assumption one can characterize one-way functions with quasipolynomial or subexponential security [LP21]. Each of these constructions implicitly yields constructions of PRFs based on the hardness of K^t. Indeed, one can apply well-known cryptographic transformations [HILL99, GGM86] to transform any one-way function into a PRF with proportional security. In further work, Liu and Pass gave a *direct* construction of a PRF from the hardness of approximating K^t [LP24a]. Their construction employs the complexity-theoretic pseudorandom generator (PRG) of Nisan and Wigderson [NW94]. This direct construction is particularly useful to us. The generic transformation from one-way functions to PRFs has super-

constant depth overhead, and thus is not a viable way of obtaining constant-depth PRFs from the hardness of our measures KP^t. On the other hand, the Nisan–Wigderson PRG *is* computable in several constant-depth circuit classes, allowing us to adapt Liu and Pass's direct construction to a constant-depth setting.

There has been prior work on translating the Liu–Pass results to other variants of Kolmogorov complexity. Notably, Ren and Santhanam [RS21] show that average-case hardness of a "time-penalized" Kolmogorov complexity KT [ABK+06] is equivalent to one-way functions in NC^0. It is not clear whether hardness of KT also implies PRFs in NC^0 since the transformation from one-way functions to PRFs does not necessarily preserve computability in NC^0, and to the best of our knowledge there are no direct constructions of PRFs from hardness of KT.

1.1 The Measures KP

Let $h : \{0,1\}^* \to \{0,1\}$ be any "reasonable" symmetric function. We focus for the remainder of this introduction on the KP-complexity measure associated with $\mathsf{AC}^0[h]$, denoted KP_h. Recall that $\mathsf{AC}^0[h]$ is the class of functions computable by constant-depth, polynomial-size circuits with NOT gates and unbounded-fan-in AND, OR, and h gates.

The measure KP_h combines elements of "time-bounded" K^t and "time-penalized" KT, defined informally below for the sake of comparison and discussion.

Definition 1.1 (Time-bounded Kolmogorov complexity on TMs; informal)**.** We say a Turing Machine (TM) M *computes* a string w *within* t *steps* if, for all inputs $i \in [|w|]$, running M on i outputs w_i in at most t steps of computation. Write $|M|$ to denote the description length of a TM.

- $\mathsf{K}^t(w)$ is the minimum number $|M|$ such that a TM M computes w within t steps.
- $\mathsf{KT}(w)$ is the minimum number $|M| + t$ such that a TM M computes w within t steps.

Notice that K^t fixes a *hard cap* t on the runtime of M in advance while KT pays an *additive penalty* for the runtime of M. Our first basic requirement for KP is that it connects to constant-depth circuit classes just as KT connects to standard Boolean circuits.

Theorem 1.2 ([ABK+06])**.** $\mathsf{KT}(x)$ *is polynomially related to the minimum size of a circuit* C *such that for every* $i \in [|x|]$, $C(i) = x_i$.

This relationship between KT and circuit size follows from the fact that Turing machines and circuits are equivalent up to polynomial complexity, that is, there are efficient simulations of (1) circuits by Turing Machines and (2) Turing Machines by circuits.[1] The circuit model in this equivalence necessarily has unrestricted depth, as otherwise simulation in the latter direction would not be possible. Intuitively, KP_h should be defined using a computational machine model that is similarly equivalent to the *depth-restricted* $\mathsf{AC}^0[h]$ model. To the best of our knowledge, no such model was available in the literature before this work.

However, connections between AC^0 circuits and constant-time Parallel Random Access Machines (PRAMs) had been thoroughly investigated by Stockmeyer and Vishkin; these models are indeed equivalent up to polynomial factors [SV84]. A PRAM is specified by a fixed finite *program,* which is a sequence of assembly-like instructions. PRAM execution starts with an input bitstring x and a specified number of processors q. Each processor is itself a Random Access Machine (RAM). The processors execute the program synchronously (but may branch to different locations in the code) and communicate via special instructions that address a shared global memory. There are several

[1] This elides issues of uniformity because we only need equivalence up to computation of fixed strings.

flavors of PRAMs, chiefly varying how conflicts are resolved when more than one processor attempts to write to the same location in global memory. The simulation of Stockmeyer and Vishkin uses the conflict resolution mechanisms of their specific PRAM model to simulate AC^0 circuits using PRAMs; the access controls for global memory can be used to simulate parallel computation of AND and OR gates with unbounded fan-in available to AC^0 circuits.

To make the global coordination capabilities of PRAMs more explicit and modular, we define *h-PRAMs*, which generalize the PRAM model considered by Stockmeyer and Vishkin with additional *global logic* instructions. During execution of a global logic instruction, each processor may contribute a single bit as input to the AND, OR, or h function. The result of evaluating that logic function on these input bits is then stored in a particular location of global memory (Section 3).

Fixing an h-PRAM M, number of processors q, and input w, denote by $M_q(w)$ the results of running M on input w with q processors. The h-PRAM M *computes a string x within t steps using q processors* if, for all inputs $i \in [|x|]$, running $M_q(i)$ prints x_i in at most t steps of computation. With these models in hand, we give the following informal definition of KP_h.

Definition 1.3 (Kolmogorov complexity on h-PRAMs; informal, see Definition 4.2)**.** $\mathsf{KP}^t_h(x)$ is the minimum number $|M| + q$ such that an h-PRAM M computes x within t steps using q processors.

The description length of a h-PRAM M includes any strings "hardcoded" into M and the sequence-coding overhead required to represent them (see Lemma 3.1 and subsequent discussion). Note the features shared by KP_h and other time-bounded Kolmogorov complexity measures. Like KT, which charges additively for the number of *time steps* used to compute x, KP_h charges additively for the number of *processors* used. Like K^t, which enforces a hard cap on the runtime of a *serial* machine, KP_h enforces a hard cap on the runtime of a *parallel* machine. The additive penalty for processors allows us to relate KP_h complexity to circuit *size*, whereas the runtime cap allows us to impose a *depth* bound on the circuits. The following two simulations yield a close relationship between KP_h and $\mathsf{AC}^0[h]$ circuit complexity.

Theorem 1.4 (Simulation of $\mathsf{AC}^0[h]$ by h-PRAMs; informal, see Theorem 5.1)**.** *Every $\mathsf{AC}^0[h]$ circuit A can be efficiently simulated by an h-PRAM M, with running time linear in the depth of A and number of processors equal to the number of wires in A. The description length of M is $O(|A| \log |A|)$—asymptotically equivalent to the number of bits used by a reasonable encoding of the circuit A.*

Proof sketch. Fix an $\mathsf{AC}^0[h]$ circuit A. We describe a h-PRAM M that simulates A. Intuitively, each processor of M is responsible for a single wire of the circuit A. Every gate g of A is mapped to a single location C_g in global memory, intended to hold the evaluation of g. Computation proceeds layer by layer. Consider all the wires $\langle *, g\rangle$ leading into gate g. Each processor corresponding to a wire (g_k, g) for $k \in [\mathsf{fanin}(g)]$ first executes a "read" instruction to fetch an already-evaluated bit b_k from C_{g_k} or from the input string. If g is not a NOT gate, then all these processors execute a global logic operation on their local bit b_k with target global memory location C_g. The logical operation requested is exactly the function computed by gate g, which must be among {AND, OR, h} because A is an $\mathsf{AC}^0[h]$ circuit. If g is a NOT gate, then the (single) processor corresponding to the gate's input wire simply writes $1 - b_k$ to the target cell C_g. This computation stores the evaluation of gate g on input x at location C_g of global memory, using only a constant number of steps. Observe that an entire layer of the circuit A can be evaluated in parallel using this simple idea. While any given layer is being computed, the processors corresponding to wires in other layers are placed in a "waiting" state.

One caveat in the above description is that it requires each processor to have a different program. This is not allowed in our PRAM model, which has a single fixed program that all processors execute.

To get around this limitation we arrange the code for each processor in a "jump table" which has a dedicated block of code for each processor. At the beginning of execution each processor uses its unique *processor ID* to compute the location of its block in the jump table and then branches to it using a jump instruction. □

Theorem 1.5 (Simulation of h-PRAMs by $\mathsf{AC}^0[h]$; informal, see Theorem 5.4)**.** *Every h-PRAM M can be efficiently simulated by an $\mathsf{AC}^0[h]$ circuit A. The depth of A is linear in the time complexity of M and the number of wires in A is a polynomial in the description length, number of steps, and number of processors used by M.*

Proof sketch. We follow the construction of Stockmeyer and Vishkin [SV84] very closely, departing only where it is necessary to support our global logic instructions and hardcoded data.

The key step in supporting global logic instructions is to recognize that because their construction works with so-called "Concurrent-Read Concurrent-Write" (CRCW) PRAMs, constant-depth circuitry for aggregating bits located on different processors is already implicit in their simulation. Concretely, CRCW PRAMs resolve conflicting writes to global memory locations by (1) assigning each processor a unique processor identifier (PID) number and (2) declaring that the processor with minimum PID wins every write conflict, which determines the contents of the contested location in global memory at each step of computation. Simulating this by circuits necessarily requires simulating simple operations on inputs coming from multiple processors. We essentially "hijack" this circuitry to *aggregate* bits from multiple processors that want to write to the same location instead of *ignoring* the processors that do not have the minimum PID. Using this intuition, we are able to handle the global logic instructions.

We support hardcoded data by encoding the data in DNFs. Since one can memorize n bits of data by a DNF with $\mathsf{poly}(n)$ wires, this only introduces an overhead that is polynomial in the amount of data hardcoded in the h-PRAM. The description length of an h-PRAM is at least the amount of data hardcoded, and so the overhead in the number of wires is polynomial in the description length of the h-PRAM. □

For a string $w \in \{0,1\}^*$, we define the $\mathsf{AC}^0_d[h]$ *circuit complexity of* w, denoted $\mathsf{AC}^0_d[h]\text{-}\mathsf{CC}(w)$ as the minimum size of a depth-d $\mathsf{AC}^0[h]$ circuit C such that for each $i \in [|w|]$, $C(i) = w_i$. Combining the above simulations we obtain the following KP_h analog of the relationship between KT and circuit complexity.

Theorem 1.6 (KP_h captures $\mathsf{AC}^0[h]$-complexity; informal)**.** *The KP_h-complexity of a string w is polynomially related to the $\mathsf{AC}^0[h]$ circuit complexity of w, up to constant-factor overhead in the time bound. More precisely, there exist universal constants $A, B > 0$ such that for all $d \in \mathbb{N}$ and $w \in \{0,1\}^*$,*

$$\widetilde{\Omega}(\mathsf{KP}_h^{Ad}(w)) \leq \mathsf{AC}^0_d[h]\text{-}\mathsf{CC}(w) \leq \mathsf{poly}(\mathsf{KP}_h^{Bd}(w)).$$

With KP_h defined, we introduce the following distinguishing problem.

Definition 1.7 (KP vs. Random problem)**.** Consider any $t \in \mathbb{N}$ and $s : \mathbb{N} \to \mathbb{N}$. In the $\mathsf{KP}\text{-vs-}\mathsf{Random}^t_h[s]$ problem the solver is given as input a string w and must (a) output YES if $\mathsf{KP}^t_h(w) \leq s(|w|)$, and (b) output NO with probability at least $2/3$ over a uniformly random w.

By adapting several previous results for time-bounded Kolmogorov complexity to our measure KP_h we show that the complexity of $\mathsf{KP}\text{-vs-}\mathsf{Random}_h$ is extensively connected to the learnability of $\mathsf{AC}^0[h]$ and existence of cryptographic primitives in $\mathsf{AC}^0[h]$. We also show that for the special case where h is the "mod p" function for some prime p, existing circuit lower bounds against $\mathsf{AC}^0[p]$ imply

t	s	Implication	Reference
$\forall$ constant t	$\forall\, s(n) \leq 2^{(\log n)^{1/(100t)}}$	$\mathsf{KP\text{-}vs\text{-}Random}^t_{\mathsf{MOD}p}[s]$ is easy	Theorem 5.7
$\exists\, t = O(d)$	$\exists\, s(n) = n - o(n)$	If $\mathsf{KP\text{-}vs\text{-}Random}^t_h[s]$ is easy, then depth-d $\mathsf{AC}^0[h]$ is learnable.	Theorem 6.2
$\forall$ constant t	$\forall\, s(n) \leq n^\varepsilon$	If depth-$O(t)$ $\mathsf{AC}^0[h]$ is learnable, then $\mathsf{KP\text{-}vs\text{-}Random}^t_h[s]$ is easy.	Theorem 6.5
$\forall$ constant t	$\forall\, s(n) < n$	If $\mathsf{KP\text{-}vs\text{-}Random}^t_h[s]$ is hard on average, then there are PRFs in $\mathsf{AC}^0[h]$.	Theorem 7.11 and its corollaries
$\exists\, t = O(\log n)$	$\forall\, s(n) = n - O(\log n)$	If there are PRFs in$'$ $\mathsf{AC}^0[h]$, then $\mathsf{KP\text{-}vs\text{-}Random}^t_h[s]$ is hard on average.	Theorem 7.18

Table 1: A summary of our results. The ε in line 3 is a constant that depends on properties of the learner. The security properties and complexity of the PRF in line 4 depend on s. The notion of computability in $\mathsf{AC}^0[h]$ for PRFs used in the last line is stronger than the notion used in the second-to-last line.

algorithms for $\mathsf{KP\text{-}vs\text{-}Random}_h$ for some thresholds s. Table 1 contains a summary of these results. The following subsections describe them in more detail before we discuss related work and open problems.

1.2 Connections to Learning

As mentioned before, Goldberg and Kabanets used the pseudorandomness versus learning paradigm and the connection between KT and general circuit complexity to reduce PAC-learning $\mathsf{P/poly}$ to deciding KT on average. Roughly speaking, their reduction uses (a generalization of) the following connection between pseudorandomness and learning. Given a function f, let

$$\mathsf{DP}^f(x_1, \ldots, x_m) = x_1 \| \cdots \| x_m \| f(x_1) \| \cdots \| f(x_m),$$

where $\|$ denotes concatenation. We call DP^f the m-wise *direct product generator* for f. It was known from prior work that if one can distinguish, for any $f \in \mathcal{C}$, the string $\mathsf{DP}^f(x)$ where x is a uniformly random string, then $\mathcal{C}$ is PAC-learnable [Vad17].[2] Goldberg and Kabanets leverage the connection between KT and circuit size to show that the KT complexity of $\mathsf{DP}^f(x)$ is low when f can be computed by feasible circuits. On the other hand, with overwhelming probability, random strings have near-maximal KT. Using this gap, they show that certain types of average-case algorithms ("randomized errorless heuristics") for KT imply a distinguisher, which then implies learning algorithms.

To show that PAC-learning $\mathsf{AC}^0[h]$ reduces to computing KP_h, we follow a similar program to that of Goldberg and Kabanets. We prove the following result, connecting the existence of efficient algorithms for $\mathsf{KP\text{-}vs\text{-}Random}_h$ to PAC-learning $\mathsf{AC}^0[h]$. We further observe that randomized errorless heuristics for KP (in the same sense as in [GK23]) imply algorithms for $\mathsf{KP\text{-}vs\text{-}Random}_h$ satisfying the premise of Theorem 1.8 and thus also yield learning algorithms for $\mathsf{AC}^0[h]$.

[2] Goldberg and Kabanets actually use a generalization of this result due to Kothari and Livni [KL18], which allows them to prove stronger forms of learnability. That generalization requires distinguishing the direct product generator where some fraction of the $f(x_i)$ bits are corrupted. We discuss their ideas in the special case of the direct product generator to simplify presentation.

Theorem 1.8 (Learning from KP; informal, see Theorem 6.2)**.** *If* $\mathsf{KP}\text{-vs-Random}_h^{O(d)}[n - o(n)]$ *admits a (possibly randomized) polynomial-time algorithm, then depth-*d $\mathsf{AC}^0[h]$ *can be learned from random examples over the uniform distribution in (randomized) polynomial time.*

Our proof, like that of Goldberg and Kabanets, goes by way of constructing a distinguisher for the direct product generator. We do so by showing that the KP_h complexity of a *shuffled* version of the direct product generator (one where evaluations appear before examples) is low, while the KP_h complexity of a random string is near-maximal. Note that we need to consider the shuffled generator because the indexing arithmetic required to compute the unshuffled $\mathsf{DP}^f(x)$ is too costly for $\mathsf{AC}^0[h]$ circuits and h-PRAMs for the distinguisher to be good enough (for many natural choices of h such as the parity function).

In the other direction, we show that if depth-d $\mathsf{AC}^0[h]$ is PAC-learnable, then $\mathsf{KP}\text{-vs-Random}_h^{O(d)}[s]$ must be easy for $s(n) = n^\varepsilon$ where $\varepsilon > 0$ is a constant depending on properties of the PAC-learner.

Theorem 1.9 (Algorithms for KP from learning; informal, see Theorem 6.5)**.** *Suppose depth-*d $\mathsf{AC}^0[h]$ *is PAC-learnable. Then there exists a constant* $\varepsilon > 0$ *that depends on the running time of the learner such that* $\mathsf{KP}\text{-vs-Random}_h^{O(d)}[n^\varepsilon]$ *admits a polynomial-time algorithm.*

Theorem 1.9 shows that approximating KP is not "overkill" for PAC-learning $\mathsf{AC}^0[h]$ from random examples. At least for some s which is considerably lower than the sufficient threshold given by Theorem 1.8, tractability of $\mathsf{KP}\text{-vs-Random}_h^t[s]$ is *necessary*. On the other hand, cryptographic impossibility for K^t and KT extends to approximation with this lower threshold. Accordingly, it is more plausible (under cryptographic assumptions) that a reduction from PAC-learning to approximating KP complexity could yield new learning algorithms. Theorem 1.9 also implies that, under cryptographic assumptions, to obtain learning algorithms using KP (if at all possible) one must use properties of KP that do not also hold for K^t or KT. It is natural to ask if the gap between complexity thresholds s in $\mathsf{KP}\text{-vs-Random}_h^t[s]$ given by Theorem 1.8 and Theorem 1.9 can be tightened; we defer this question to future work.

The proof of Theorem 1.9 crucially uses the simulation of h-PRAMs by $\mathsf{AC}^0[h]$ circuits. Suppose there exists a learner for $\mathsf{AC}^0[h]$. To test whether a string w has low KP_h complexity, we simulate the learner on the function that maps $i \in [|w|]$ to w_i, estimate the accuracy of the hypothesis it produces, and accept if it is sufficiently high. If w has low KP_h complexity then, by Theorem 1.6, w also has low $\mathsf{AC}^0[h]$ circuit complexity. This means the learner can learn the indexing function for w with high probability, and so our algorithm accepts with high probability. On the other hand, notice that if the learner is efficient, the hypothesis produced by it has low description length. A simple counting argument shows that random strings cannot be approximated to high accuracy by hypotheses with low description length.

1.3 Connections to Constant-Depth Pseudorandom Functions

A pseudorandom function (PRF) is a function $F_\lambda : \{0,1\}^{k(\lambda)} \times \{0,1\}^{\ell(\lambda)} \to \{0,1\}$ such that a $\mathsf{poly}(\lambda)$-time adversary cannot distinguish, given oracle access, between (a) $F_\lambda(z, \cdot)$ where $z \in \{0,1\}^{k(\lambda)}$ is uniformly random and (b) a random function $f : \{0,1\}^{\ell(\lambda)} \to \{0,1\}$. We call $k(\lambda)$ the *key length* of the PRF and $\{0,1\}^{\ell(\lambda)}$ the *input domain*. We scale down the direct construction of PRFs from hardness of K^t due to Liu and Pass [LP24a] to show that average-case hardness of $\mathsf{KP}\text{-vs-Random}_h$ implies pseudorandom functions computable in $\mathsf{AC}^0[h]$, as long as the circuit class $\mathsf{AC}^0[h]$ is powerful enough to compute the parity function.

Theorem 1.10 (PRF from Hardness of KP; informal, see Theorem 7.11)**.** *For every reasonable* h *such that* $\oplus \in \mathsf{AC}^0[h]$ *and* $s(n) < n$*, if* $\mathsf{KP}\text{-vs-Random}_h^t[s]$ *is hard on average*, then there exists a*

PRF computable by constant-depth, polynomial-size $\mathsf{AC}^0[h]$ *circuits. The key length, input domain, and security properties depend on* s *and parameters in the hardness assumption.*

As a concrete illustration, we state the following quantitative instantiation of the theorem.

Corollary 1.11 (Concrete PRF from Hardness of KP; informal, see Corollary 7.13)**.** *For every reasonable* h *such that* $\oplus \in \mathsf{AC}^0[h]$*, if* $\mathsf{KP}\text{-vs-Random}^t_h[2^{O(\log^\varepsilon n)}]$ *is hard on average*, then there is a PRF computable by constant-depth,* $\mathsf{poly}(\lambda)$*-size circuits with key length* $O(\lambda^{1+C})$ *and input domain* $\{0,1\}^{\Omega(\log^{C'}\lambda)}$ *for some constants* $C, C' > 0$*.*

The star in "hard on average*" refers to the fact that we use the average-case hardness notion (with the same name) of Liu and Pass [LP21]. Roughly speaking, $\mathsf{KP}\text{-vs-Random}^t_h[s]$ is α-hard on average* if there is no efficient algorithm that both (a) accepts a random n-bit string w such that $\mathsf{KP}^t_h(w) \le s(|w|)$ with probability at least $1-\alpha(n^*)$, and (b) rejects a uniformly random n-bit string with probability at least $1-\alpha(n^*)$. Here, n^* denotes the logarithm of the number of n-bit strings w with $\mathsf{KP}^t_h(w) \le s(n)$. This notion of average-case hardness is useful as it is meaningfully two-sided even for very sparse languages such as the set of strings with low KP_h complexity. Indeed, for average-case hardness where the error probability is proportional to the input length n, if the number of YES instances of length n is, say, polynomial in n, then a $1/\mathsf{poly}(n)$-good algorithm could not make *any* mistakes on YES instances.

In particular, we get that hardness of $\mathsf{KP}\text{-vs-Random}^t_\oplus[2^{O(\log^\varepsilon n)}]$ implies a PRF with key length $O(\lambda^{1+C})$ and input domain $\{0,1\}^{\Omega(\log^{C'}\lambda)}$ computable in $\mathsf{AC}^0[\oplus]$. This adds to the growing list of candidate PRF constructions in $\mathsf{AC}^0[\oplus]$ based on various hardness assumptions [ABG+14, BCG+21, FLY22].

We note that unlike several candidate constructions, our construction yields a *strong* PRF, that is, a PRF secure against adversaries with *oracle* access to the function as opposed to just adversaries who are given random samples. One might wonder whether this contradicts the results of [CIKK16], which give efficient learning algorithms for $\mathsf{AC}^0[\oplus]$ given membership queries (i.e., oracle access). The reason the security of the PRF, call it F, given by Corollary 1.11 cannot be broken by a straightforward application of [CIKK16] amounts to its input domain size. The learning algorithm of [CIKK16] works as long as the function being learned can be computed by $\mathsf{AC}^0[\oplus]$ circuits with size polynomial in *the number of input bits to the function*. To use the learner to break the security of F, one would have to learn the function $F(z,\cdot) : \{0,1\}^{\mathsf{polylog}(\lambda)} \to \{0,1\}$ for a random key z. But notice that the only guarantee on the circuit size of $F(z,\cdot)$ that we have is that it is computable by $\mathsf{poly}(\lambda)$-size $\mathsf{AC}^0[\oplus]$ circuits—superpolynomial in the number of input bits it takes. Despite the limited input domain, the PRF is cryptographically "useful:" it takes in $\mathsf{poly}(\lambda)$ bits of entropy as a key, and produces $2^{\mathsf{polylog}(\lambda)} \gg \mathsf{poly}(\lambda)$ bits that are computationally indistinguishable from random.

As mentioned before, the proof of Theorem 1.10 closely follows [LP24a]. The key technical step in [LP24a] is constructing a "weak family of PRGs:" a set of (sufficiently) length-expanding functions $\{g_z\}_{z\in\mathcal{I}}$ such that $g_z(y)$ with uniformly random z and y is indistinguishable from a uniformly random string. A weak family of PRGs can be generically transformed into a PRF as long as each g_z is "locally computable," i.e., each bit $g_z(y)_i$ is efficiently computable. The family of PRGs is constructed so that $g_z(y)$ interprets the key z as the description of a Turing machine M and evaluates the Nisan–Wigderson generator $\mathsf{NW}^{M(\cdot)}(y)$. Hardness of K^t and the reconstruction property of the Nisan–Wigderson generator imply the required security of the family, while the definition of the Nisan–Wigderson generator ensures local computability.

We adapt the proof by replacing Turing machines with h-PRAMs and K^t with KP^t in the construction, and show that the analysis goes through with these modifications. Doing so requires deriving counting and average-case* complexity results for KP analogous to those known for K^t. Our

simulation of h-PRAMs by $\mathsf{AC}^0[h]$ circuits, the fact that the Nisan–Wigderson generator can be computed efficiently in $\mathsf{AC}^0[\oplus]$ [CIKK16], and a simple analysis of the transformation from weak families of PRGs to PRFs, allow us to conclude that the resulting PRF is computable in $\mathsf{AC}^0[h]$.

We also make progress in the other direction. We prove that the existence of PRFs in $\mathsf{AC}^0[h]$ implies hardness of $\mathsf{KP\text{-}vs\text{-}Random}_h$ with a logarithmic (as opposed to constant) time bound.

Theorem 1.12 (Hardness of KP from PRFs; informal, see Theorem 7.18)**.** *If there exists a PRF computable′ by polynomial-size* $\mathsf{AC}^0[h]$ *circuits, then* $\mathsf{KP\text{-}vs\text{-}Random}_h^{O(\log n)}[n - O(\log n)]$ *is hard on average*.*

"Computable′" refers to the fact that we require a stronger computability requirement here than in Theorem 1.10. We conjecture that the PRF constructed in Theorem 1.10 in fact also satisfies this stronger computability requirement, though we leave proving that to future work (see Remark 7.7). The other limitation of Theorem 1.12 is that it only gets hardness for KP-vs-Random with a logarithmic time bound. This limitation arises since we do not know how to implement certain primitives used in the proof of the theorem using constant-depth circuits (see Remark 7.15).

The proof of Theorem 1.12 uses techniques from [RS21]. Ren and Santhanam show that if there are one-way functions computable in $\oplus\mathsf{L}$, then there exist *entropy-preserving PRGs* computable by log-depth circuits. These entropy-preserving PRGs are constructed using *randomized encodings* [AIK06], which are the primitives we do now know how to build with constant-depth circuits. We are able to adapt calculations from [RS21] to show that if $\mathsf{KP\text{-}vs\text{-}Random}_h^{O(\log n)}[n - O(\log n)]$ is easy on average*, then the security of the aforementioned PRGs can be broken. Finally, we observe that PRFs computable′ in $\mathsf{AC}^0[h]$ imply one-way functions computable in $\oplus\mathsf{L}$ (so long as h is sufficiently reasonable), completing the proof.

1.4 Algorithms for $\mathsf{KP\text{-}vs\text{-}Random}_{\mathsf{MOD}_p}[\mathrm{polylog}]$

The breakthrough works of Razborov [Raz87] and Smolensky [Smo93] showed that the majority function cannot be computed by constant-depth, polynomial-size $\mathsf{AC}^0[p]$ circuits. Razborov and Rudich [RR97] showed that those lower bounds are *natural*, that is, they imply efficient algorithms (called "natural properties") that can distinguish between truth tables of functions computable by small $\mathsf{AC}^0[p]$ circuits and those of random functions. We observe that, combined with our simulation of h-PRAMs by $\mathsf{AC}^0[h]$ circuits, these natural properties imply that $\mathsf{KP\text{-}vs\text{-}Random}_{\mathsf{MOD}_p}$ is unconditionally easy with certain thresholds.

Theorem 1.13 (Algorithms for $\mathsf{KP\text{-}vs\text{-}Random}_{\mathsf{MOD}_p}$)**.** *For all constants* t *and* $s(n) \leq 2^{(\log n)^{1/(100t)}}$, $\mathsf{KP\text{-}vs\text{-}Random}^t_{\mathsf{MOD}_p}[s]$ *admits a polynomial-time algorithm.*

1.5 Related Work

Previous approaches to scaling down resource-bounded Kolmogorov complexity. *Boolean formulas* are circuits with wiring restrictions: fan-in two and fan-out one. Allender et al. defined *time-penalized* Kolmogorov complexity with respect to *alternating Turing machines,* denoted KF. They proved that KF is polynomially related to Boolean formula size by giving efficient simulations of (1) Boolean formulas by alternating Turing machines and (2) alternating Turing machines by Boolean formulas [AKRR11, Proposition 6].

The complexity class NC^1 is defined by restricting Boolean circuits to $O(\log n)$ depth and fan-in two (fan-out is unrestricted). NC^1 is equivalent to polynomial-sized Boolean formulas. Therefore, there is also efficient mutual simulation between NC^1 circuits and alternating Turing machines. Ren

and Santhanam defined *time-bounded* Kolmogorov complexity with respect to *alternating Turing machines,* denoted NC^1-K^t. They proved that NC^1-K^t is hard on average to approximate if and only if there is a one-way function computable in $\mathsf{DLOGTIME}$ [RS21, Theorem 42].

Just like KP, the above measures vary resource-bounded Kolmogorov complexity by replacing Turing machines with a different model of computation. But alternating Turing machines appear unable to "scale down" and capture constant-depth circuit complexity in this setting. Indeed, Allender et. al. comment that "it is natural to wonder if it is possible (and useful) to define even more restrictive notions of Kolmogorov complexity, in order to capture even more limited models of computation" [AKRR11, Section 8]. Our results about KP show that it is both possible (via the relationship between PRAMs and circuits) and useful (because of the connections to learning and cryptography) to define very restrictive notions of Kolmogorov complexity.

Characterizing OWFs in NC^0 using time-penalized Kolmogorov complexity. Ren and Santhanam show (quite surprisingly) that average-case hardness of approximating KT is equivalent to one-way functions (OWFs) computable by constant-depth circuits with *bounded* fan-in or NC^0 [RS21]. They apply the *randomized encodings* of Applebaum, Ishai, and Kushilevitz [AIK06] to do so, previously used to show that there are one-way functions in NC^0 if and only if there are one-way functions in logspace. They further leverage randomized encodings to exhibit reductions between approximating KT and other complexity measures on average, including NC^1-K^t [RS21, Theorems 6–8]. Thus, the results of Ren and Santhanam use randomized encodings to characterize OWFs and obtain close relationships between distinct Kolmogorov complexity measures.

Our results about KP^t concern pseudorandom functions instead of one-way functions, and obtain a linear relationship between t and circuit depth instead of reductions between approximating distinct members of the KP family. Furthermore, on a technical level, it is unclear if randomized encodings can be as useful in our setting as in that of Ren and Santhanam.

Other variants of Kolmogorov complexity. As resource-bounded Kolmogorov complexity becomes yet more pervasive, many natural variations are defined and analyzed. Much recent work explores compressibility with respect to *stronger* computational models, such as probabilistic, nondeterministic, and interactive machines [LO22, HLO26, BLMP23]. This is quite different from the present work introducing KP, because we make the underlying computational model apparently *weaker.* However, these lines of investigation are entirely compatible. In particular, we hope that future work will explore similar extensions of KP complexity.

1.6 Future Directions

We have shown that KP can be substituted for K^t in simple predictor-to-distinguisher reductions. Do more complicated reductions and statements also scale down from K^t to KP in a natural way? For example, can Inductive Inference for PRAMs be reduced to approximating KP? One step towards answering this question would be to determine if an analog of the conditional chain rule for K^t ([HN26, Lemma 3.1]) holds for KP.

More broadly, we can classify proofs according to which properties of K^t they use. KP shares two basic properties with K^t, and indeed with standard Kolmogorov complexity: the KP-complexity of every string can be upper bounded by memorization (Proposition 4.3) and most strings are KP-incompressible (Proposition 4.4). Are there widely useful properties of K^t that are unknown, unlikely, or simply false for KP? Alternatively, do the apparent differences between these compressibility measures arise entirely from their computational tractability (or lack thereof)?

Another concrete question is whether the gaps between the forward and backward implications related to learning (Theorems 1.8 and 1.9) and cryptography (Theorems 1.10 and 1.12) can be closed to obtain equivalences between easiness/hardness of $\mathsf{KP\text{-}vs\text{-}Random}_h$ and learning/cryptography in $\mathsf{AC}^0[h]$. In the case of learning, the main gap is in the threshold parameters: to obtain learning algorithms for $\mathsf{AC}^0[h]$ our results require an algorithm distinguishing strings with KP_h complexity $n-o(n)$ from random strings, whereas an arbitrary PAC-learner for $\mathsf{AC}^0[h]$ only implies a distinguisher with threshold n^{ε}. We note that the parameter ε can be increased by reducing the overhead in simulating PRAMs by circuits, though we do not believe such optimizations can get us all the way to $n - o(n)$. In our cryptography results, the main gap is in the time bound for which we are able to prove hardness. We note that to obtain hardness with a constant time bound, it suffices to implement randomized encodings in $\mathsf{AC}^0[h]$.

2 Preliminaries

We let $\mathbb{N} = \{1, 2, \ldots\}$ be the set of natural numbers. We denote by $[n]$ the set $\{1, 2, \ldots, n\}$. We use $\|$ for string concatenation. Logarithms in this paper are base-2 by default. For $a \in \mathbb{N}$ write $\mathrm{lsb}(a) \in \{0,1\}$ for the least significant bit of the binary expansion of a.

For a finite set S, $\mathcal{U}[S]$ denotes the uniform distribution over S. For $n \in \mathbb{N}$, $\mathcal{U}_n$ denotes the uniform distribution over $\{0,1\}^n$. If D is a distribution, we write "$x \sim D$" to mean "x is drawn according to D." For a set S, we write "$x \sim S$" for "$x \sim \mathcal{U}[S]$;" consequently, "$x \sim \{0,1\}^n$" is equivalent to "$x \sim \mathcal{U}_n$."

2.1 Circuit Model

Definition 2.1 (Circuit)**.** Fix a set ("basis") of Boolean functions $\mathcal{B}$. A *circuit C over $\mathcal{B}$* is a directed acyclic graph. Each node ("gate") of C is labeled by a variable from $\{x_i\}_{i\in[n]}$ or an element of $\mathcal{B}$. Exactly one gate of C is labeled as the *output*.

The *size* of the circuit, denoted $|C|$, is the number of edges ("wires") in C. The *depth* of a circuit is the length of the longest path from a variable to the output gate.

We denote by $\mathcal{B}^0$ the standard AC^0 basis of unbounded-fan-in AND, unbounded-fan-in OR, and NOT functions: $\mathcal{B}^0 = \{\wedge, \vee, \neg\}$. Since all bases considered in this paper will be supersets of $\mathcal{B}^0$, henceforth we use the term *basis* to mean a set of functions that necessarily contains $\mathcal{B}^0$ as a subset. Furthermore, we insist that all functions in a basis other than $\neg$ must be symmetric functions with unbounded fan-in.

Definition 2.2 (Constant-Depth Circuit Complexity of Strings)**.** For every $d \in \mathbb{N}$, the AC^0_d *circuit complexity of a string x* is defined by

$$\mathsf{AC}^0_d\text{-}\mathsf{CC}(x) = \min_C\{\ |C|\ :\ C \text{ is a depth-}d \text{ circuit over } \mathcal{B}^0 \text{ such that } \forall i \in [|x|]\ C(i) = x_i\ \}.$$

Let $h : \{0,1\}^* \to \{0,1\}$ be a symmetric Boolean function. The $\mathsf{AC}^0_d[h]$ *circuit complexity of x* is defined by

$$\mathsf{AC}^0_d[h]\text{-}\mathsf{CC}(x) = \min_C\{\ |C|\ :\ C \text{ is a depth-}d \text{ circuit over } \mathcal{B}^0 \cup \{h\} \text{ and } \forall i \in [|x|]\ C(i) = x_i\ \}.$$

Let $w_1, w_2, \ldots, w_{2^n} \in \{0,1\}^n$ be the list of n-bit strings in lexicographic order. Given an arbitrary Boolean function $f : \{0,1\}^n \to \{0,1\}$, the truth table of f is $tt(f) = f(w_1)\|f(w_2)\|\ldots\|f(w_{2^n})$. For any time-constructible function $s : \mathbb{N} \to \mathbb{N}$, denote by $\mathsf{AC}^0_d\text{-}\mathsf{SIZE}[s(n)]$ the set of all sequences of

Boolean functions $\{f_n : \{0,1\}^n \to \{0,1\}\}_{n\in\mathbb{N}}$ such that $\mathsf{AC}^0_d\text{-}\mathsf{CC}(tt(f_n)) = O(s(n))$. Then define $\mathsf{AC}^0_d = \bigcup_{c\in\mathbb{N}} \mathsf{AC}^0_d\text{-}\mathsf{SIZE}[n^c]$ and $\mathsf{AC}^0 = \bigcup_{d\in\mathbb{N}} \mathsf{AC}^0_d$. Classes $\mathsf{AC}^0_d[h]\text{-}\mathsf{SIZE}[s(n)]$, $\mathsf{AC}^0_d[h]$, and $\mathsf{AC}^0[h]$ are defined analogously. Of interest to us are the special cases $\mathsf{AC}^0[\oplus]$ and $\mathsf{TC}^0 = \mathsf{AC}^0[\mathsf{MAJ}]$ where MAJ is the majority function.

We define by $\mathsf{MOD}p : \{0,1\}^* \to \{0,1\}$ the function that outputs 0 if the Hamming weight of its input is divisible by p and 1 otherwise. We use the standard notation $\mathsf{AC}^0[p]$ as shorthand for $\mathsf{AC}^0[\mathsf{MOD}p]$. Note that $\oplus = \mathsf{MOD2}$, and so $\mathsf{AC}^0[\oplus] = \mathsf{AC}^0[2]$.

We will often refer to circuits over $\mathcal{B}^0$ as "AC^0 circuits," circuits over $\mathcal{B}^0 \cup \{h\}$ as "$\mathsf{AC}^0[h]$ circuits," and so on.

2.2 Learning and the Direct-Product Generator

An *example oracle* for a Boolean function f over a distribution D, denoted $\mathsf{Ex}_D(f)$, is an oracle that when queried returns a pair $(x, f(x))$ where x is drawn (independent of previous outputs) according to D. We write $\mathsf{Ex}(f)$ to mean $\mathsf{Ex}_D(f)$ where D is the uniform distribution over the domain of f. We say that a circuit C is ε-close to f over D if $\Pr_{x\sim D}[C(x) \neq f(x)] \leq \varepsilon$.

Definition 2.3 (Realizable PAC Learning [Val84])**.** A (possibly randomized) algorithm A *PAC-learns a class of functions $\mathcal{C}$ over a distribution D to error ε and confidence $1-\delta$* if for every $f : \{0,1\}^n \to \{0,1\}$ in $\mathcal{C}$

$$\Pr_{A,\mathsf{Ex}_D(f)}[A^{\mathsf{Ex}_D(f)}(1^n) \text{ outputs a circuit that is } \varepsilon(n)\text{-close to } f \text{ over } D] \geq 1-\delta(n).$$

We say $\mathcal{C}$ is *PAC-learnable over D* to error ε and confidence $1-\delta$ if it admits such an algorithm A.

We will show in Section 6 that computing KP is sufficient for learning constant-depth circuit classes. Our algorithms will use properties of the direct product generator (see, e.g., [Hir20]).

Definition 2.4 (Direct Product Generator)**.** Let $f : \{0,1\}^n \to \{0,1\}$ be a function. The *m-wise direct product generator for f*, $\mathsf{DP}^f : (\{0,1\}^n)^m \to \{0,1\}^{nm+m}$, is defined as

$$\mathsf{DP}^f(x^1, \ldots, x^m) = x^1 \| \cdots \| x^m \| f(x^1) \| \cdots \| f(x^m).$$

Yao [Yao82] showed a connection between the pseudorandomness of the direct product generator and the complexity of computing f. The following lemma, which is implicit in work by Vadhan [Vad17] and whose proof is sketched in a paper by Ilango, Loff, and Oliveira [ILO20, Appendix A], uses ideas from Yao's result and is at the heart of our results. We include a full proof in Appendix A for completeness.

Lemma 2.5 ([Vad17, ILO20])**.** *Let $\mathcal{C}$ be a class of circuits, and let $c \geq 1$. Suppose there exists $m(n) \leq \mathsf{poly}(n)$ and a polynomial-time algorithm B such that for all $n \in \mathbb{N}$ and $f \in \mathcal{C}\text{-}\mathsf{SIZE}[n^c]$,*

$$\Pr_{x\sim(\{0,1\}^n)^{m(n)}}[B(1^n, \mathsf{DP}^f(x)) = 1] - \Pr_{r\sim\{0,1\}^{nm(n)+m(n)}}[B(1^n, r) = 1] \geq \frac{1}{10}.$$

Then there exists a $\mathsf{poly}(n)$-time algorithm A that PAC-learns $\mathcal{C}\text{-}\mathsf{SIZE}[n^c]$ over the uniform distribution to error $\varepsilon = 1/2 - 1/O(m)$ and confidence $1 - 1/\mathsf{poly}(n)$.

2.3 Average-Case Complexity

A *distributional problem* is a pair $(L, \mathcal{D})$ where $L \subseteq \{0,1\}^*$ is a language and $\mathcal{D} = \{D_n\}_{n\in\mathbb{N}}$ is a family of distributions. A distribution family that will be important for us is the family of *parameterized uniform distributions* $\mathcal{U} = \{\mathcal{U}_{n,t_1,\ldots,t_k}\}_{n,t_1,\ldots,t_k\in\mathbb{N}}$ where $\mathcal{U}_{n,t_1,\ldots,t_k} = (\mathcal{U}_n, 1^{t_1}, \ldots, 1^{t_k})$.

Definition 2.6 (Randomized Errorless Heuristics [BT06])**.** A *t-time randomized errorless heuristic* for a distributional problem (L, D) is an algorithm A such that, for every $n \in \mathbb{N}$, $x \in \mathrm{supp}(D_n)$, and $\delta > 0$,

1. $A(x; n, \delta)$ runs in time $t(n, 1/\delta)$;
2. $\Pr_A[A(x; n, \delta) \notin \{L(x), \perp\}] \leq \frac{1}{10}$; and
3. $\Pr_{y\sim D_n}[\Pr_A[A(y; n, \delta) = \perp] \geq 1/10] \leq \delta(n)$.

We denote by AvgBPP the class of all distributional problems (L, D) that admit polynomial-time randomized errorless heuristics.

Intuitively, a randomized errorless heuristic can "detect" its errors and output the failure symbol $\perp$, except with some probability over its internal randomness. The term "errorless" refers to the property that the probability of outputting an erroneous answer is independent of the input distribution.

2.4 The Nisan–Wigderson Generator

In Section 7 we build PRFs based on the average-case hardness of KP. The PRF construction uses the Nisan–Wigderson generator, which we give a high-level overview of here.

Definition 2.7 (Nisan–Wigderson Generator [NW94])**.** Let $\mathcal{I} = (I_1, \ldots, I_m)$ be a sequence of subsets of $[d]$. Let $|I_i| = \ell$ for every i, and let $f : \{0,1\}^\ell \to \{0,1\}$ be a function. The *Nisan–Wigderson generator for f with design $\mathcal{I}$* is the function $\mathsf{NW}^f_{\mathcal{I}} : \{0,1\}^d \to \{0,1\}^m$ given by

$$\mathsf{NW}^f_{\mathcal{I}}(y) = f(y_{I_1}) \| f(y_{I_2}) \| \cdots \| f(y_{I_m}).$$

Here y_{I_j} with $I_j = \{i_1, i_2, \ldots, i_\ell\}$ denotes the string $y_{i_1} y_{i_2} \ldots y_{i_\ell}$.

The sequences of sets $\mathcal{I}$ that are useful to us are the so-called "combinatorial designs."

Definition 2.8 (Combinatorial Design [NW94])**.** A sequence of sets $\mathcal{I} = (I_1, \ldots, I_m)$ is said to be a *(d, ℓ, k)-design with size m* if for every i, $I_i \subseteq [d]$ and $|I_i| = \ell$, and for every i, j with $i \neq j$, $|I_i \cap I_j| \leq k$.

Lemma 2.9 ([NW94])**.** *Let $d(\ell)$ be ℓ times the smallest power of 2 that is greater than or equal to ℓ. There is a deterministic algorithm $\mathsf{NWDesign}$ that, given input $\ell, \kappa, i \in \mathbb{N}$ with $\kappa < \ell$ and $i \in [2^\kappa]$, outputs a subset $I_i^{\ell,\kappa} \subseteq [d(\ell)]$ such that for every valid ℓ and κ, $(I_1^{\ell,\kappa}, I_2^{\ell,\kappa}, \ldots, I_{2^\kappa}^{\ell,\kappa})$ is a $(d(\ell), \ell, \kappa)$-design. The algorithm runs in time $\widetilde{O}(\ell^2)$.*

Carmosino, Impagliazzo, Kabanets, and Kolokolova [CIKK16] showed that the designs generated by $\mathsf{NWDesign}$ can in fact be generated efficiently in $\mathsf{AC}^0[\oplus]$.

Lemma 2.10 ([CIKK16])**.** *There is a universal constant $d_{\mathrm{NW}} \geq 1$ such that for every $\ell, \kappa, i \in \mathbb{N}$ with $\kappa < \ell$ and $i \in [2^\kappa]$, the function $\mathsf{MX}_{\mathrm{NW}} : [2^\kappa] \times \{0,1\}^{d(\ell)} \to \{0,1\}^\ell$ defined by $\mathsf{MX}_{\mathrm{NW}}(i, z) = z_{I_i}$ where $I_i = \mathsf{NWDesign}(\ell, \kappa, i)$ can be computed by an $\mathsf{AC}^0[\oplus]$ circuit with size $\mathsf{poly}(\ell)$ and depth d_{NW}.*

We will use the following version of the Nisan–Wigderson reconstruction theorem.

Theorem 2.11 (Implicit in [NW94, IW97], explicit in [LP22])**.** *There is a probabilistic polynomial-time algorithm* NWRecon *with the following input/output behavior.*

1. *Input: the truth table of $f : \{0,1\}^\ell \to \{0,1\}$, a (d, ℓ, κ)-design $\mathcal{I}$ with size m, and a distinguishing gap parameter $1^{\varepsilon^{-1}}$.*
2. *Oracle: a (possibly probabilistic) machine D such that*
$$\left| \Pr_{y \sim \{0,1\}^d}[D(\mathsf{NW}^f_{\mathcal{I}}(y)) = 1] - \Pr_{r \sim \{0,1\}^m}[D(r) = 1] \right| \geq \varepsilon.$$
3. *Output: a deterministic Turing machine M with description length at most $m2^\kappa + m + d + O(\log d)$ such that*
$$\Pr_{i \sim \{0,1\}^\ell}[M(\mathcal{I}, i) = f(i)] \geq \frac{1}{2} + \frac{\varepsilon}{2m}.$$

2.5 Cryptography

Definition 2.12 (One-Way Functions)**.** A polynomial-time computable function $f : \{0,1\}^* \to \{0,1\}^*$ is a *one-way function* (OWF) if for all probabilistic polynomial-time algorithms A, for all sufficiently large λ,
$$\Pr_{x \sim \{0,1\}^\lambda}[f(A(1^\lambda, f(x))) = f(x)] \leq \mathsf{negl}(\lambda).$$

We say f is computable in $\oplus\mathsf{L}$ if it can be computed by an $O(\log n)$-space parity Turing machine.

Definition 2.13 (Weak Family of PRGs)**.** A family of polynomial-time computable functions $g_\lambda = \{g_j : \{0,1\}^{d(\lambda)} \to \{0,1\}^{m(\lambda)}\}_{j \in S_\lambda}$ is an *α-weak family of (T, ε)-pseudorandom generators* (PRGs) if for every $T(\lambda)$-time probabilistic algorithm D, for all sufficiently large λ,
$$\Pr_{j \sim S_\lambda}\left[\left| \Pr_{y \sim \{0,1\}^{d(\lambda)}, D}[D(1^\lambda, g_j(y)) = 1] - \Pr_{r \sim \{0,1\}^{m(\lambda)}, D}[D(r) = 1] \right| \leq \varepsilon(\lambda) \right] \geq \alpha(\lambda).$$

Definition 2.14 (Pseudorandom Functions)**.** A polynomial-time computable function $F_\lambda : \{0,1\}^{k(\lambda)} \times \{0,1\}^{\ell(\lambda)} \to \{0,1\}$ is a *(T, ε)-pseudorandom function* (PRF) if it is $\mathsf{poly}(\lambda)$-time computable and for every $T(\lambda)$-time probabilistic algorithm D, for sufficiently large λ,
$$\left| \Pr_{z \sim \{0,1\}^{k(\lambda)}}[D^{F_\lambda(z,\cdot)}(1^\lambda) = 1] - \Pr_{\substack{f : \{0,1\}^{\ell(\lambda)} \to \{0,1\} \\ \text{uniformly random}}}[D^{f(\cdot)}(1^\lambda) = 1] \right| \leq \varepsilon(\lambda).$$

We call $k(\lambda)$ the key length and $\{0,1\}^{\ell(\lambda)}$ the input domain of the PRF.

There are two natural ways of defining the circuit complexity of a PRF in restricted classes such as $\mathsf{AC}^0[h]$. The first is to define it as the maximum over all key sequences $\{s_\lambda \in \{0,1\}^{k(\lambda)}\}_{\lambda \in \mathbb{N}}$ of the circuit complexity of the function family $\{F(s_\lambda, \cdot)\}_{\lambda \in \mathbb{N}}$. This notion appears to be the standard in the literature (see, e.g., [BR17, ABG+14, BCG+21, FLY22]), and thus is the definition we use. A second, perhaps even more natural, way is to simply define the circuit complexity of the PRF as being the circuit complexity of the function $F(\cdot, \cdot)$. We call this variant the *joint* circuit complexity of F.

Definition 2.15 (Circuit Complexity of PRFs)**.** Let $\mathcal{C}$ be a class of circuits. A PRF $F_\lambda : \{0,1\}^{k(\lambda)} \times \{0,1\}^{\ell(\lambda)} \to \{0,1\}$ is *computable by $p(\lambda)$-size $\mathcal{C}$-circuits* if for every sequence $\{s_\lambda \in \{0,1\}^{k(\lambda)}\}_{\lambda\in\mathbb{N}}$ the functions $\{F_\lambda(s_\lambda, \cdot)\}_{\lambda\in\mathbb{N}}$ are computable by $\mathcal{C}$-circuits of size $p(\lambda)$.

F is *jointly computable by $p(\lambda)$-size $\mathcal{C}$-circuits* if $\{F_\lambda(\cdot,\cdot)\}_{\lambda\in\mathbb{N}}$ is computable by $p(\lambda)$-size $\mathcal{C}$-circuits.

We say that F is (jointly) computable by $p(\lambda)$-size *uniform* $\mathcal{C}$-circuits if there is a logspace Turing machine that, when given input $(1^\lambda, s_\lambda)$ (or just 1^λ in the joint case) prints a $p(\lambda)$-size $\mathcal{C}$-circuit (jointly) computing F.

3 (Parallel) Random Access Machines

3.1 Finite Oracles

Our RAMs and PRAMs will be allowed to access finite oracles, which we describe before our machine models. A *finite oracle* $\mathcal{O}$ is a finite sequence of finite bitstrings $\mathcal{O} = \mathcal{O}_1, \ldots, \mathcal{O}_k$. The strings $\mathcal{O}_i$ may be of different sizes, including zero (the empty string). When defined, we write $\mathcal{O}_{i,j}$ to refer to the jth bit of $\mathcal{O}_i$. The number of strings in $\mathcal{O}$ is called the *length* of $\mathcal{O}$ and denoted $\mathsf{len}(\mathcal{O}) = k$. The number of bits stored in $\mathcal{O}$ is called the *size* of $\mathcal{O}$ and denoted $\mathsf{size}(O) = \sum_{i\in[k]} |\mathcal{O}_i|$. Concatenating together all strings in $\mathcal{O}$ is called *flattening* the oracle: $\mathsf{flat}(\mathcal{O}) = \mathcal{O}_1 \| \ldots \| \mathcal{O}_k$. Sometimes it is natural to think of $\mathcal{O}$ as a Boolean matrix (possibly with some entries undefined). In these cases, we will call i the "row number" and j the "column number" of bit $\mathcal{O}_{i,j}$. We denote by $\mathcal{O}^*$ the space of all finite oracles.

3.2 Random Access Machines

A *Random Access Machine* (RAM) M is specified by a finite *program*. The program consists of a sequence of instructions $I_1, I_2, \ldots, I_s$ drawn from the instruction set detailed in Figure 1. RAMs are equipped with an infinite sequence of *registers* $R_1, R_2, \ldots$, each containing a natural number or the *blank* symbol $\perp$. The input to a RAM is given as a finite, read-only bitstring, and is accessed via a dedicated instruction that can read one bit per step of computation. The *state* of a RAM consists of the contents of the registers and the *instruction counter* $\alpha \in [s]$. To execute a RAM M on input x with k finite oracles $\mathcal{O}^1, \ldots, \mathcal{O}^k$, we set all registers to $\perp$ and initialize $\alpha = 1$, and then repeatedly update the state as per the instruction I_α and the semantics given in Figure 1. M halts when I_α is a `HALT Ri` instruction. Its output, denoted $M^{\mathcal{O}^1,\ldots,\mathcal{O}^k}(x) \in \{0,1\}$, is the least significant bit (lsb) of the ith register R_i if it contains a number and 0 if it contains $\perp$.

3.3 Parallel Random Access Machines

Fix a basis $\mathcal{B}$ of symmetric Boolean functions. A *Parallel Random Access Machine over $\mathcal{B}$* ($\mathcal{B}$-PRAM) is specified by a finite program, just like a RAM. The program consists of a sequence of instructions $I_1, I_2, \ldots, I_s$ drawn from an expanded instruction set consisting of the basic RAM instruction set (Figure 1) and the additional instructions detailed in Figure 2. The input is presented to a $\mathcal{B}$-PRAM the same way it is presented to a RAM: as a finite sequence of bits accessed via a dedicated instruction.

The key difference between RAMs and $\mathcal{B}$-PRAMs is execution semantics and state. A $\mathcal{B}$-PRAM M may be executed using any number of *processors* q. Each processor is a RAM over the expanded instruction set, equipped with its own private registers and instruction counter, running the program

Let s be the number of instructions in a RAM program. We specify instructions with $i, j, k \in \mathbb{N}$ and $\ell \in [s]$. As a mnemonic, assignment flows right to left. To make arithmetic operations and comparisons total, the blank symbol $\perp$ is an absorbing element (e.g., $\forall x \in \mathbb{N}, \perp + x = \perp$) smaller than every number.

Memory. Every Memory instruction increments the instruction counter α.

1. `Ri := j` // $R_i \leftarrow j$ // Load register i with a constant "hardcoded" in the program.
2. `Ri := Rj` // $R_i \leftarrow R_j$ // Copy the value of register j into register i.
3. `Reg[Ri] := Rj` // $R_{R_i} \leftarrow R_j$ // Indirect write, treating the value in register i as an "address."
4. `Ri := Reg[Rj]` // $R_i \leftarrow R_{R_j}$ // Indirect read, treating the value in register j as an "address."
5. `Ri := Oracle1D $o $j` // $R_i \leftarrow \mathsf{flat}(\mathcal{O}^{\$o})_{\$j}$ if $\$j \leq \mathsf{size}(\mathcal{O}^{\$o})$, $\perp$ otherwise.
6. `Ri := Oracle2D $o $j $k` // $R_i \leftarrow \mathcal{O}^{\$o}_{\$j,\$k}$ if $\$j \leq \mathsf{len}(\mathcal{O}^{\$o})$ and $\$k \leq |\mathcal{O}^{\$o}_{\$j}|$, $\perp$ otherwise.

Arithmetic & Logic (ALU). Every ALU instruction increments the instruction counter α.

1. `Ri := Rj + Rk` // $R_i \leftarrow R_j + R_k$
2. `Ri := Rj - Rk` // $R_i \leftarrow R_j \ominus R_k$, where $a \ominus b = \max(a - b, 0)$ denotes *proper subtraction.*
3. `Ri := Rj >> Rk` // $R_i \leftarrow \lfloor R_j / 2^{R_k} \rfloor$.
4. `Ri := Rj %2 Rk` // $R_i \leftarrow$ the remainder $R_j \bmod 2^{R_k}$.

Control Flow.

1. `if Ri = Rj JMP Rk` // If $R_i = R_j$ then $\alpha \leftarrow R_k$. Otherwise, $\alpha \leftarrow \alpha + 1$. // Jump on equality.
2. `if Ri != Rj JMP Rk` // If $R_i \neq R_j$ then $\alpha \leftarrow R_k$. Otherwise, $\alpha \leftarrow \alpha + 1$. // Jump on inequality.
3. `if Ri > Rj JMP Rk` // If $R_i > R_j$ then $\alpha \leftarrow R_k$. Otherwise, $\alpha \leftarrow \alpha + 1$. // Jump on greater-than.
4. `if Ri < Rj JMP Rk` // If $R_i < R_j$ then $\alpha \leftarrow R_k$. Otherwise, $\alpha \leftarrow \alpha + 1$. // Jump on less-than.

Input & Output. Every I/O instruction increments the instruction counter α. For each execution, the *input* is a finite sequence of bits $x = x_1, \ldots, x_n$.

1. `Ri := In[Rj]` // $R_i \leftarrow \begin{cases} x_{R_j} & \text{if } R_j \leq n \\ \perp & \text{otherwise.} \end{cases}$ // Indirect read a bit of input into register i.
2. `LEN Ri` // $R_i \leftarrow n$ // Get the length of the input.
3. `HALT Ri` // Halt execution and output $\mathrm{lsb}(R_i)$ if $R_i \in \mathbb{N}$ or 0 if $R_i = \perp$.

Figure 1: Basic RAM Instruction Set

Parallel Computation & Memory Access. For display, suppose q processors are executing and denote by $p \in [q]$ the PID of the current processor in the description of each instruction. Let $\mathcal{B}$ be the basis. Every instruction here increments the instruction counters $\alpha_1, \ldots \alpha_q$.

1. `Ri := PID` // $R_i^p \leftarrow p$ where $p \in [q]$ is the PID of the processor executing this instruction.
2. `Ri := Cell[Rj]` // $R_i^p \leftarrow C_{R_j^p}$ // Read R_j^pth cell of global memory into register i of processor p.
3. `Cell[Ri] := Rj` // $C_{R_i^p} \leftarrow \text{lsb}(R_j^p)$ // Global write. Lowest PID wins write conflicts.
4. `Cell[Ri] :=` *OP* `$w` // $C_{R_i^p} \leftarrow OP(y)$ where $OP \in \mathcal{B} \setminus \{\neg\}$ and y is formed by concatenating the least significant bit of register `$w` from each processor executing this instruction with target cell R_i^p.

   Formally, let $p_1, \ldots, p_v$ be the IDs of processors executing an *OP* instruction with target cell C_k. Let $w_1, \ldots, w_v$ be the registers named by `$w` on each such processor, respectively. Concatenate to form $y = \text{lsb}(R_{w_1}^{p_1}) \| \ldots \| \text{lsb}(R_{w_v}^{p_v})$ and assign $C_k \leftarrow OP(y)$. All functions in $\mathcal{B} \setminus \{\neg\}$ are symmetric and have unbounded fan-in, so the order of concatenation does not matter.
5. `Cell[Ri] := ROM1 $o $idx $bit` // $C_{R_i^p} \leftarrow \begin{cases} \mathsf{flat}(\mathcal{O}^{\$\mathtt{o}})_\ell & \text{if } \ell \leq \mathsf{size}(\mathcal{O}^{\$\mathtt{o}}) \\ \perp & \text{otherwise} \end{cases}$

   where the number ℓ is formed by placing lsb(`$bit`) in the `$idx`th bit of the binary expansion of ℓ, from all processors executing the `ROM1` instruction with target cell R_i^p.
6. `Cell[Ri] := ROM2 $o $dim $idx $bit` // $C_{R_i^p} \leftarrow \begin{cases} \mathcal{O}^{\$\mathtt{o}}_{k,\ell} & \text{if } k \leq \mathsf{len}(\mathcal{O}^{\$\mathtt{o}}) \text{ and } \ell \leq |\mathcal{O}^{\$\mathtt{o}}_k| \\ \perp & \text{otherwise} \end{cases}$

   where the number k is formed from all processors executing `ROM2 $o 0 * *` by placing lsb(`$bit`) in the `$idx`th bit of the binary expansion of k, and the number ℓ is formed by doing the same for processors executing `ROM2 $o 1 * *`.
7. `NOP` // No operation; helps with synchronization.

Conflict Resolution. Conflicts arise when processors disagree about simultaneous instructions that target the same global memory cell C_k. To determine the outcome,

1. Let p be the minimum processor ID targeting cell C_k. Select instruction $I = I_{\alpha_p}$ executed by p.
2. If I is a Write (Instruction 3), then $C_k \leftarrow \text{lsb}(R_j^p)$ and computation proceeds.
3. If I is an *OP* (Instruction 4), then proceed as described above and simply ignore any processors attempting to execute a different instruction with target C_k.
4. If I is a `ROM1` or `ROM2` (Instructions 5 or 6), conflicts may arise in formation of the indices ℓ and/or k. In all cases the processor with lowest PID sets the value of the conflicted indexing bit.

Figure 2: Instruction set expansion for PRAMs.

$I_1, I_2, \ldots, I_s$.[3] Each processor has a unique and static *processor ID* (PID) accessible via a dedicated instruction. To coordinate distinct processors, $\mathcal{B}$-PRAMs have an infinite global memory of *cells* denoted $C_1, C_2, \ldots$, each containing a single bit or $\perp$. The *global operations* (Instructions 4-6 of Figure 2) make cells active computational elements: they aggregate information from many processors to determine what bit is stored after each step of computation.

The state of a $\mathcal{B}$-PRAM running with q processors consists of the contents of the registers of all processors, $R_1^1, R_2^1, \ldots, R_1^2, R_2^2, \ldots, R_1^q, R_2^q, \ldots$, the instruction counters of all the processors, $\alpha_1, \alpha_2, \ldots, \alpha_q$, and the contents $C_1, C_2, \ldots$ of all global memory cells. To execute a $\mathcal{B}$-PRAM M on an input x with q processors and k finite oracles $\mathcal{O}^1, \ldots, \mathcal{O}^k$, we begin by setting all registers of all processors to $\perp$, all global cells to $\perp$, and all q instruction counters to 1. Then, in lockstep, we update local registers, global cells, and instruction counters as per the semantics in Figures 1 and 2 until at least one processor executes a `HALT Ri` instruction. Let p be the minimum PID that executed a `HALT`. The output $M_q^{\mathcal{O}^1,\ldots,\mathcal{O}^k}(x) \in \{0,1\}$ is $\mathrm{lsb}(R_i^p)$, the least significant bit of register i on processor p, if R_i^p contains a number and 0 if it contains $\perp$.

A $\mathcal{B}$-PRAM execution runs in time t if the number of instructions executed by each processor is t (since processors work in lockstep, all processors execute the same number of steps). Algorithms written for $\mathcal{B}$-PRAMs require both time and processor upper bounds. Fixing $t, q : \mathbb{N} \to \mathbb{N}$, the language $L \subseteq \{0,1\}^*$ is decided by a $\mathcal{B}$-PRAM using t time and q processors if for every input $x \in \{0,1\}^*$, $M_{q(|x|)}(x) = L(x)$ in at most $t(|x|)$ steps. Since this is the only PRAM model considered in this paper, we may refer to $\mathcal{B}$-PRAMs as simply "PRAMs" where the basis is clear from context.

The *description length* of a $\mathcal{B}$-PRAM M is the number of bits required to represent M using a fixed binary encoding, denoted $|M|$. We fix a concrete encoding and description length bounds via Lemma 3.1 below. Any binary encoding with comparable overhead would be suitable for all applications in this work. Note that the encoding of RAMs by Cook & Reckhow is *not* suitable, because it represents constants in unary [CR73, Table IV].

Lemma 3.1 (Efficient Encoding for $\mathcal{B}$-PRAMs)**.** *There is an absolute constant k depending only on $\mathcal{B}$ such that every $\mathcal{B}$-PRAM M has encoding length $|M| = sk\log(\max\{c, r\})$, where*

1. *s is the number of instructions in M,*
2. *c is the largest numerical constant value appearing in an instruction of M, and*
3. *r is the number of distinct register names appearing in the instructions of M.*

Very frequently, we will need to talk about the combined description length of a $\mathcal{B}$-PRAM and a finite list of finite oracles. We do so by fixing a *canonical pairing function* $\langle\cdot\rangle$ that has the property that for all objects $a_1, a_2, \ldots, a_k$ with well-defined binary encodings, $|\langle a_1, a_2, \ldots, a_k\rangle| = \sum_{i=1}^k |a_i| + \sum_{j=2}^k O(\log |a_j|)$. Such encodings can be constructed by, for example, concatenating the lengths of the encodings of all but one of the objects in bit-doubled format followed by the individual encodings. Note that a non-empty finite oracle $\mathcal{O}$ can be encoded with $|\mathcal{O}| + O(\log \mathsf{size}(\mathcal{O}))$ bits.

4 Resource-Bounded Kolmogorov Complexity on $\mathcal{B}$-PRAMs

The PRAM model naturally induces a notion of Kolmogorov complexity which we define formally in this section.

[3] Note that all processors in a $\mathcal{B}$-PRAM execute the *same fixed program*, though they each have their own instruction counter and thus may branch to different parts of the program. This is in contrast to the "nonuniform PRAM" model from the 1980s (see, e.g., [SV84]), where every processor is allowed its own program and the program itself may depend on the input length.

Definition 4.1 (Evaluation function for PRAMs)**.** Given a $\mathcal{B}$-PRAM M encoded as a bitstring according to Lemma 3.1, an input $x \in \{0,1\}^*$, a time bound $t \in \mathbb{N}$, a number of processors $q \in \mathbb{N}$, and a (possibly empty) sequence of finite oracles $\mathcal{O}^1, \ldots, \mathcal{O}^k$, define the *evaluation function* for $\mathcal{B}$-PRAMs as

$$\mathcal{E}_{\mathcal{B}}(M, x, t, q, \mathcal{O}^1, \ldots, \mathcal{O}^k) = \begin{cases} M_q^{\mathcal{O}^1,\ldots,\mathcal{O}^k}(x) & \text{if } M_q^{\mathcal{O}^1,\ldots,\mathcal{O}^k}(x) \text{ halts within } t \text{ steps} \\ \perp & \text{otherwise.} \end{cases}$$

If the input M is not a valid encoding of a $\mathcal{B}$-PRAM, $\mathcal{E}_{\mathcal{B}}$ outputs 0 regardless of the other arguments. The symbol $\mathcal{E}$ will be used as a shorthand for $\mathcal{E}_{\mathcal{B}^0}$.

In standard definitions of Kolmogorov complexity, a universal machine serves to both (1) fix an encoding and (2) impose semantics. Recall that, given a universal Turing machine U and a time parameter t, the t-time-bounded Kolmogorov complexity of a string x, $\mathsf{K}^t(x)$, is the minimum length of a "description" d such that for all $i \in [|x|]$, $U(d, i)$ halts and outputs x_i in $t(|d|)$ steps. Both the exact format of strings d and the mapping of indices to bits are determined by fixing a "sufficiently reasonable and efficient" universal machine U.

Constructions of "reasonable and efficient" universal *PRAMs* are not often given, in stark contrast to the wide range of thoroughly studied universal TMs and RAMs available in the literature. Therefore, we accomplish jobs (1) and (2) without a UPRAM by (1) fixing a concrete encoding of $\mathcal{B}$-PRAMs (Lemma 3.1) and (2) precisely specifying abstract machine semantics (Figures 1 and 2). This is directly analogous to circuit complexity, where it is standard to define encoding and evaluation of circuits separately and concretely.

Definition 4.2 (Kolmogorov Complexity on PRAMs)**.** For every basis $\mathcal{B}$, $w \in \{0,1\}^*$, and $t \in \mathbb{N}$,

$$\mathsf{KP}^t_{\mathcal{B}}(w) = \min_{\substack{M \in \{0,1\}^*,\ q \in \mathbb{N} \\ k \in \mathbb{N} \cup \{0\},\ \mathcal{O}^1,\ldots,\mathcal{O}^k \in \mathcal{O}^*}} \{|\langle M, \mathcal{O}^1, \mathcal{O}^2, \ldots, \mathcal{O}^k \rangle| + q \ : \ \forall i \in [|w|]\ \mathcal{E}_{\mathcal{B}}(M, i, t, q, \mathcal{O}^1, \ldots, \mathcal{O}^k) = w_i\}.$$

Write KP^t as shorthand for $\mathsf{KP}^t_{\mathcal{B}^0}$. Given an unbounded-fan-in symmetric Boolean function h we use KP^t_h to denote $\mathsf{KP}^t_{\mathcal{B}^0 \cup \{h\}}$.

In Section 7 we use KP with *super-constant* time bounds. The measure KP^t where t is a function of $|w|$ is defined in the obvious way.

The measures $\mathsf{KP}_{\mathcal{B}}$ share features with both time-bounded Kolmogorov complexity K^{t} and "time-penalized" Kolmogorov complexity KT. Like KT, which charges additively for the number of *time steps* used to locally compute x, $\mathsf{KP}_{\mathcal{B}}$ charges additively for the number of *processors* used. Like K^{t}, which enforces a hard cap on the runtime of a *serial* machine, $\mathsf{KP}_{\mathcal{B}}$ enforces a hard cap on the runtime of a *parallel* machine.

We remark that this formulation of KP_h is entirely motivated by connections to $\mathsf{AC}^0[h]$-circuit complexity. One could just as well imagine other parallel Kolmogorov complexities defined using PRAMs that clip the number of processors, charge additively for time (maybe even logarithmically, like in Levin's "Kt" complexity), or both.

Before investigating the properties of KP, we define some notation that will be useful later. We will often refer to a tuple $(M, \mathcal{O}^1, \ldots, \mathcal{O}^k)$ where M is a $\mathcal{B}$-PRAM and the $\mathcal{O}^i$s are finite oracles together as a *machine* Π. We use $\Pi^t_q(i)$ as shorthand for $\mathcal{E}_{\mathcal{B}}(M, i, t, q, \mathcal{O}^1, \ldots, \mathcal{O}^k)$ and use $|\Pi|$ to denote $|\langle M, \mathcal{O}^1, \ldots, \mathcal{O}^k \rangle|$. For any $s \in \mathbb{N}$, we define $\mathcal{X}_{\mathcal{B},s}$ as the set of all tuples (Π, q) such that

1. $\Pi = (M, \mathcal{O}^1, \ldots, \mathcal{O}^k)$ where M is a $\mathcal{B}$-PRAM and $\mathcal{O}^1, \ldots, \mathcal{O}^k$ are finite oracles;

```
0. Input: i ∈ {0,1}*. Oracle: O^1 containing w.
1. R1 := PID
2. R2 := In[R1]
3. Cell[1] := ROM1 1 R1 R2
4. R1 := Cell[1]
5. HALT R1
```

Figure 3: PRAM that memorizes a string w.

2. $q \in \mathbb{N}$;
3. $|\Pi| + q \leq s$.

Therefore for all x, $\mathsf{KP}^t_{\mathcal{B}}(x) \leq s$ if and only if there exists $(\Pi, q) \in \mathcal{X}_{\mathcal{B},s}$ such that for all $i \in [|x|]$, $\Pi^t_q(i) = x_i$. We drop $\mathcal{B}$ and simply write $\mathcal{X}_s$ where the basis is clear from context.

4.1 Basic Properties of KP

KP shares several basic properties with standard variants of Kolmogorov complexity, derived below.

Proposition 4.3 (Memorization)**.** *For all bases $\mathcal{B}$, $w \in \{0,1\}^*$, and $t \geq 5$, $\mathsf{KP}^t_{\mathcal{B}}(w) \leq |w| + O(\log |w|)$.*

Proof. Fix any $w \in \{0,1\}^*$. Let M be the PRAM given in Figure 3, and let $\mathcal{O}^1$ be a finite oracle containing w. It is easy to see that for all $i \in [|w|]$, $M^{\mathcal{O}^1}_{\lceil \log(|w|) \rceil}(i) = w_i$. Furthermore, the PRAM has constant program length, oracle size $|w|$, and always runs in 5 time steps. The KP upper bound follows. □

Proposition 4.4 (Most strings are KP-incompressible)**.** *For all bases $\mathcal{B}$ and $t \in \mathbb{N}$,*

$$\Pr_{w \sim \{0,1\}^n}[\mathsf{KP}^t_{\mathcal{B}}(w) \leq n - 2\log n] \leq \frac{1}{n}.$$

Proof. Let $s(n) = n - 2\log n$. We bound the size of the set $\mathcal{X}_{\mathcal{B},s(n)}$. For every $q \in [s(n)]$, the number of machines Π such that $(\Pi, q) \in \mathcal{X}_{\mathcal{B},s(n)}$ is exactly the number of machines Π with length at most $s(n) - q$, which is less than $2^{s(n)-q+1}$. Thus

$$|\mathcal{X}_{\mathcal{B},s(n)}| \leq \sum_{q=1}^{s(n)} 2^{s(n)-q+1} = 2^{s(n)+1} - 1.$$

Fix $t \in \mathbb{N}$. For each $w \in \{0,1\}^n$, $\mathsf{KP}^t_{\mathcal{B}}(w) \leq s(n)$ if and only if there exists $(\Pi, q) \in \mathcal{X}_{\mathcal{B},s(n)}$ such that for every $i \in [n]$, $\Pi^t_q(i) = w_i$. Note that for a fixed (Π, q) there is exactly one string w such that $\Pi^t_q(i) = w_i$ for all i. Thus the number of n-bit strings w with $\mathsf{KP}^t_{\mathcal{B}}(w) \leq s(n)$ is at most $|\mathcal{X}_{\mathcal{B},s(n)}| \leq 2^{s(n)+1}$. It follows that the probability that a random string $w \sim \{0,1\}^n$ has $\mathsf{KP}^t_{\mathcal{B}}(w) \leq s(n)$ is $2^{s(n)+1}/2^n = 2^{n-2\log n+1-n} \leq 1/n$. □

4.2 Computational Problems Based on KP_h

The measure $\mathsf{KP}_{\mathcal{B}}$ in turn induces the following decision problems, analogous to the well-studied $\mathsf{MK^tP}$ and MKTP problems [Sip83, ABK+06].

Definition 4.5. For every basis $\mathcal{B}$,

$$\mathsf{MKPP}_{\mathcal{B}} = \{(w, 1^t, 1^s) \mid w \in \{0,1\}^*,\ \mathsf{KP}^t_{\mathcal{B}}(w) \leq s\}.$$

For a fixed $t \in \mathbb{N}$ and $s : \mathbb{N} \to \mathbb{N}$, we define by $\mathsf{MKPP}^t_{\mathcal{B}}[s]$ the "slice"

$$\mathsf{MKPP}^t_{\mathcal{B}}[s] = \{w \in \{0,1\}^* \mid \mathsf{KP}^t_{\mathcal{B}}(w) \leq s(|w|)\}.$$

Write MKPP as shorthand for $\mathsf{MKPP}_{\mathcal{B}^0}$. Given any symmetric Boolean function h, we use MKPP_h to denote $\mathsf{MKPP}_{\mathcal{B}^0 \cup \{h\}}$.

We also define the following distinguishing problem.

Definition 4.6. The $\mathsf{KP\text{-}vs\text{-}Random}^t_{\mathcal{B}}[s]$ problem is defined as follows.

- The input is a string $w \in \{0,1\}^*$.
- The output should be YES if $\mathsf{KP}^t_{\mathcal{B}}(w) \leq s(|w|)$.
- The output should be NO if w is a uniformly random $|w|$-bit string.

As with MKPP, we use $\mathsf{KP\text{-}vs\text{-}Random}^t_h[s]$ as shorthand for $\mathsf{KP\text{-}vs\text{-}Random}^t_{\mathcal{B}^0 \cup \{h\}}$. An algorithm A solves the $\mathsf{KP\text{-}vs\text{-}Random}^t_{\mathcal{B}}[s]$ problem with error δ if for all but finitely many input lengths n, (a) if $w \in \{0,1\}^n$ has $\mathsf{KP}^t_{\mathcal{B}}(w) \leq s(n)$, $\Pr_A[A(w) \text{ accepts}] \geq 1 - \delta(n)$, and (b) $\Pr_{w \sim \{0,1\}^n, A}[A(w) \text{ accepts}] < \delta(n)$. For $s(n) < n - 2\log n$, the $\mathsf{KP\text{-}vs\text{-}Random}^t_{\mathcal{B}}[s]$ is non-degenerate (i.e., the YES and NO cases do not overlap with non-negligible probability) due to Proposition 4.4. We may sometimes require the input length and threshold to depend on some other underlying parameter. In such cases we will explicitly specify the dependence by saying A solves $\mathsf{KP\text{-}vs\text{-}Random}^t_{\mathcal{B}}[s(n)]$ on input length $\ell(n)$.

5 KP_h and $\mathsf{AC}^0[h]$ Complexity Are Polynomially Related

5.1 Simulating circuits with PRAMs

We now show that every circuit A over $\mathcal{B}$ can be efficiently simulated by a $\mathcal{B}$-PRAM M, with running time linear in the depth of A and number of processors equal to the number of wires in A. This simulation is optimal (up to constant factors) in description length: we obtain $|M| = O(|A| \log |A|)$, asymptotically equivalent to the number of bits used by a reasonable encoding of A.

Theorem 5.1. *For all symmetric Boolean functions h there exists a constant $\alpha \in (0,1)$ such that for all $t \in \mathbb{N}$ and $x \in \{0,1\}^*$, $\mathsf{KP}^t_h(x) = \widetilde{O}(\mathsf{AC}^0_{\alpha t}[h]\text{-}\mathsf{CC}(x))$. The $\widetilde{O}$ suppresses a single* log *factor.*

Proof. Fix an arbitrary symmetric Boolean function $h : \{0,1\}^* \to \{0,1\}$ and let $\mathcal{B} = \{\wedge, \vee, \neg, h\}$. Let $\alpha < 1$ be a rational constant to be determined later. Fix a string $x \in \{0,1\}^*$ and a natural number t. Suppose A is a minimal depth-αt circuit over $\mathcal{B}$ computing x. Let w be the number of wires in A, so that $\mathsf{AC}^0_{\alpha t}[h]\text{-}\mathsf{CC}(x) = w$. We describe a $\mathcal{B}$-PRAM M that computes x: for all $i \in [|x|]$, the output $M_w(i) = x_i$ within t time steps. Our machine will satisfy $|\langle M \rangle| = O(w \log w)$, establishing the claimed upper bound on $\mathsf{KP}^t_h(x)$.

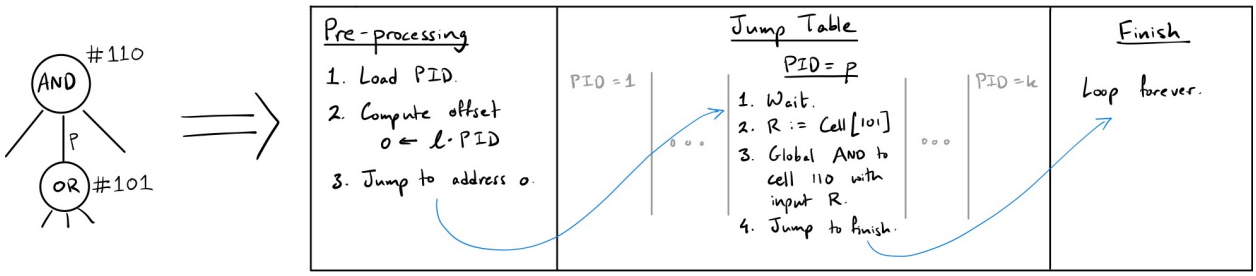

Figure 4: The life cycle of a processor simulating a wire of a circuit.

Intuitively, each processor of M is responsible for a single wire of the circuit A. Every gate g of A corresponds to a single cell C_g in global memory, intended to hold the evaluation of g. Computation proceeds layer by layer. Consider all the wires $\langle *, g\rangle$ leading into gate g. Each processor corresponding to a wire (g_k, g) for $k \in [\mathsf{fanin}(g)]$ first executes a "read" instruction to fetch a bit b_k from the input or from C_{g_k}. Then, all these processors execute a global $\mathsf{type}(g)$-operation on their local bit b_k with target cell C_g. This computation aggregates all the bits b_k using the function computed by gate g in A and stores the result in global memory, at cell C_g. Observe that an entire layer of the circuit A can be evaluated in parallel using this simple idea. While any given layer is being computed, the processors corresponding to wires in other layers are in a "waiting" state.

More specifically, the program for M consists of three sections that appear in the following order: a "pre-processing" section, a "finished" section that implements an infinite loop, and a "jump table" with w "blocks," one for each processor.[4] A processor's execution begins in the pre-processing step, where it loads a register with its PID and jumps to the block dedicated to it. We will ensure that all blocks in the jump table have a constant number of instructions (say ℓ), so this can be done by computing the address of the jump table entry as a hardcoded constant offset plus $\ell \cdot \texttt{PID}$ and then jumping to it. Note that $\ell \cdot \texttt{PID}$ can be computed in constant time by repeated addition since ℓ is a constant.

The code block for PID k is responsible for simulating information flow through the kth wire of A. When the blocks for all wires in a given layer are executed in lockstep, the outputs of all gates in that layer are computed and written to the appropriate global cells. Suppose the kth internal wire of A is at the d_kth layer from the bottom, reads from gate g_k, and feeds into gate g'_k. The code block for PID k does the following.

1. Wait for $c_{\text{wait}} \cdot d_k$ time steps by incrementing a counter register, where c_{wait} is a constant that will be decided later in this proof. The number $c_{\text{wait}} \cdot d_k$ is hardcoded in the program, and its encoding length $\log(c_{\text{wait}} d_k)$ is crudely upper bounded by $O(\log w)$.

2. Read cell C_{g_k} into a local register R. Because the number of reachable gates in a circuit is at most the number of wires in the circuit, the input cell number can be hardcoded using $O(\log w)$ bits.

3. If g'_k is a $\mathcal{B}$-gate of type $OP \neq \neg$, execute a global OP targeting cell $C_{g'_k}$ with the operand R. Then execute a `NOP` to ensure this block is synchronized with the case where $OP = \neg$. By the same reasoning as in the previous step, the address of the target cell can be hardcoded using $O(\log w)$ bits.

[4]The finished section is placed before the jump table to ensure that all hardcoded offsets for jump instructions are of constant size.

4. If g'_k is a $\neg$ gate, compute $1 - R$ and write it to cell $C_{g'_k}$.

5. Jump to the finished section. Note that since the finished section was placed before the jump table, the address of the finished state is a constant.

Note that steps 2–5 of the block have a total running time that is a constant number of time steps independent of k. We set c_{wait} to be this constant. For the output wire, we replace steps 3–5 by a `HALT` instruction.

It is easy to see that on any input $i \in \{0,1\}^{\log |x|}$, $M_w(i)$ simulates $A(i)$ layer by layer to compute x_i. Also, note that there is a constant c_0 independent of A such that each layer of the circuit is computed in c_0 time-steps. Thus the running time of M is at most $c_0 \cdot \alpha t$. Setting $\alpha = 1/c_0$, the running time of M is at most t. The description length of any given jump table block is $O(\log w)$ bits. The pre-processing and finished sections have constant length. Since there are w jump table blocks, the total length of the program is $O(1) + O(w \log w) = O(w \log w)$. The KP upper bound follows since the number of processors used is w.

□

5.2 Simulating PRAMs with circuits

Conversely to Theorem 5.1, the execution of an arbitrary PRAM M with oracles $\mathcal{O}^1, \ldots, \mathcal{O}^k$ and p processors for t time steps can be simulated by depth-$O(t)$ circuits with size $\mathsf{poly}(|\langle M, \mathcal{O}^1, \ldots, \mathcal{O}^k \rangle| + p)$. Like the simulation of circuits by PRAMs, this construction generalizes to PRAMs over more extensive bases, although in this case we can only generically handle bases consisting of functions that are "paddable" in a certain sense. We note that this limitation does not preclude modifying the construction to handle non-paddable functions. Indeed, as we show in Corollary 5.5, a simple modification of the construction allows us to handle the (non-paddable) majority function.

Definition 5.2 (Bit-paddability)**.** A Boolean function $f : \{0,1\}^* \to \{0,1\}$ is *bit-paddable* if there exists a bit b such that for every $x \in \{0,1\}^*$, $f(x) = f(x\|b)$.

The AND, OR, MOD_p, Parity, and fixed threshold functions are all bit-paddable. The majority function is not. Our construction will use simple *oracle gadgets* to memorize oracles and PRAM code.

Lemma 5.3. *Let $\mathcal{O}$ be a finite oracle and define the functions $\pi_1^{\mathcal{O}}, \eta_1^{\mathcal{O}} : \mathbb{N} \to \{0,1\}$ and $\pi_2^{\mathcal{O}}, \eta_2^{\mathcal{O}} : \mathbb{N} \times \mathbb{N} \to \{0,1\}$ as follows.*

$$\pi_2^{\mathcal{O}}(i,j) = \begin{cases} \mathcal{O}_{i,j} & \text{if } \mathcal{O}_{i,j} \text{ is defined, i.e., } i \leq \mathsf{len}(\mathcal{O}) \text{ and } j \leq |\mathcal{O}_i| \\ 0 & \text{otherwise.} \end{cases}$$

$$\eta_2^{\mathcal{O}}(i,j) = \begin{cases} 0 & \text{if } \mathcal{O}_{i,j} \text{ is defined, i.e., } i \leq \mathsf{len}(\mathcal{O}) \text{ and } j \leq |\mathcal{O}_i| \\ 1 & \text{otherwise.} \end{cases}$$

$$\pi_1^{\mathcal{O}}(i) = \begin{cases} \mathsf{flat}(\mathcal{O})_i & \text{if } \mathsf{flat}(\mathcal{O})_i \text{ is defined, i.e., } i \leq \mathsf{size}(\mathcal{O}) \\ 0 & \text{otherwise.} \end{cases}$$

$$\eta_1^{\mathcal{O}}(i) = \begin{cases} 0 & \text{if } \mathsf{flat}(\mathcal{O})_i \text{ is defined, i.e., } i \leq \mathsf{size}(\mathcal{O}) \\ 1 & \text{otherwise.} \end{cases}$$

Each of these functions is implementable by a constant-depth, $\mathsf{poly}(\mathsf{size}(\mathcal{O}))$-size AC^0 circuit.

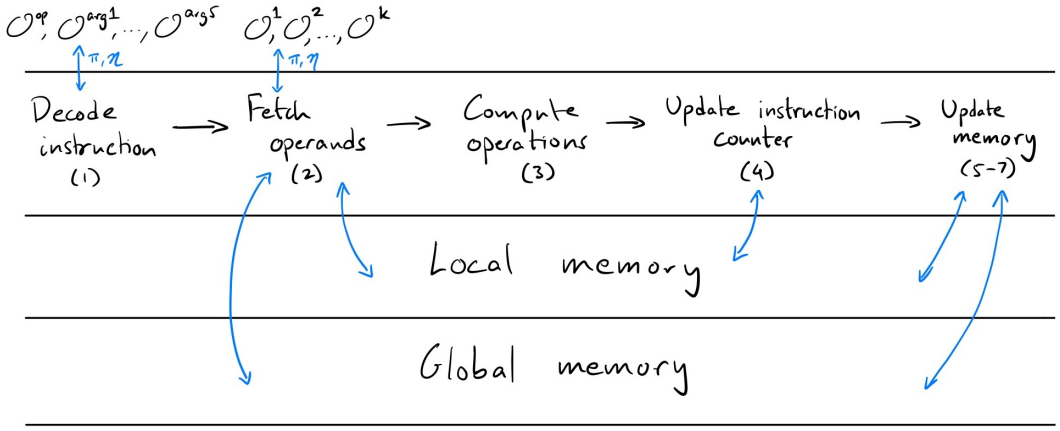

Figure 5: The phases of simulating one step of PRAM computation by constant-depth circuits.

Proof. We describe how to implement π_2 and η_2. The circuits for π_1 and η_1 are obtained via simple modifications. Recall that integer comparison is in AC^0, i.e., there is a constant-depth, $\mathsf{poly}(n)$-size circuit $C_{\leq}$ that, given two n-bit natural numbers a and b, outputs 1 if and only if $a \leq b$. Recall also that integer equality is in AC^0, i.e., there is a constant-depth, $\mathsf{poly}(n)$-size circuit $C_{=}$ that, given n-bit strings a and b, outputs 1 if and only if $a = b$. (See, e.g., [Vol99] for proofs of these facts.)

Let $\mathcal{O}$ be an arbitrary finite oracle with $\mathsf{len}(\mathcal{O}) = m$. For $k \in [m]$, let $a_k = |\mathcal{O}_k|$. Notice that for all (i, j) where $i \leq m$ is encoded as a $\lceil \log m \rceil$-bit integer and j is a $\lceil \max_{k \in [m]} \log a_k \rceil$-bit integer, the expression $C_1(i, j) = \bigvee_{k \in [m]} [(i = k) \wedge (j \leq a_k)]$ evaluates to 1 if and only if $j \leq a_i$. Since the values a_k can be hardcoded and equality and comparison are in AC^0, there is a constant-depth circuit of size $\mathsf{poly}(\log m + \max_{k \in [m]} \log a_k) = \mathsf{polylog}(\mathsf{size}(\mathcal{O}))$ computing C_1. Notice that $\eta_2^{\mathcal{O}}(i, j) = \neg[(i \leq \mathsf{len}(\mathcal{O})) \wedge C_1(i, j)]$. It follows that $\eta_2^{\mathcal{O}}$ can be implemented with a constant-depth, $\mathsf{polylog}(\mathsf{size}(\mathcal{O}))$-size circuit.

We now turn our attention to $\pi_2^{\mathcal{O}}$. First, note that the "otherwise" case in the definition of $\pi_2^{\mathcal{O}}(i, j)$ occurs if and only if $\eta_2^{\mathcal{O}}(i, j) = 1$. The oracle $\mathcal{O}$ can be memorized as follows. Define

$$C_{\mathcal{O}}(i, j) = \bigvee_{k \in [m], \ell \in [a_k]} [(i = k) \wedge (j = \ell) \wedge \mathcal{O}_{k,\ell}].$$

It is easy to see that when (i, j) is within bounds for $\mathcal{O}$, $C_{\mathcal{O}}(i, j) = \mathcal{O}_{i,j}$. There are at most $m \cdot (\max_{k \in [m]} a_k) \leq \mathsf{size}(\mathcal{O})^2$ terms in the top-level OR gate, and each term can be implemented by a constant-depth, $\mathsf{polylog}(\mathsf{size}(\mathcal{O}))$-size circuit since equality is in AC^0. Thus $C_{\mathcal{O}}$ can be implemented by a constant-depth, $\mathsf{poly}(\mathsf{size}(\mathcal{O}))$-size AC^0 circuit. The circuit upper bound for $\pi_2^{\mathcal{O}}$ follows from the observation that $\pi_2^{\mathcal{O}}(i, j) = (\eta_2^{\mathcal{O}}(i, j) \wedge 0) \vee (\neg \eta_2^{\mathcal{O}}(i, j) \wedge C_{\mathcal{O}}(i, j))$. □

The proof of the following theorem constructs circuits with $\pi^{\mathcal{O}}$ and $\eta^{\mathcal{O}}$ and invokes Lemma 5.3 to replace those gates with constant-depth sub-circuits. In that proof, we will refer to the first input to π_2 and η_2 gates (sub-circuits) as the "row" input and the second input as the "column" input.

Theorem 5.4. *For all bit-paddable, symmetric Boolean functions $h : \{0,1\}^* \to \{0,1\}$ there is a constant $\beta \geq 1$ such that for all constants t, $\mathsf{AC}^0_{\beta t}[h]\text{-}\mathsf{CC}(x) \leq \mathsf{poly}(\mathsf{KP}^t_h(x))$.*[5]

[5]The theorem also holds for super-constant t, though in that case the upper bound obtained is $t \cdot \mathsf{poly}(\mathsf{KP}^t_h(x))$.

Proof. We follow the construction of Stockmeyer and Vishkin [SV84] closely, departing where it is necessary to support our global operations. Let $\mathcal{B} = \{\wedge, \vee, \neg, h\}$. Fix a $\mathcal{B}$-PRAM M, a list of finite oracles $\mathcal{O}^1, \mathcal{O}^2, \ldots, \mathcal{O}^k$, a number p, and a constant time bound t. Let $s = |\langle M, \mathcal{O}^1, \ldots, \mathcal{O}^k \rangle| + p$.

We describe a depth-$O(t)$ $\mathsf{AC}^0[h]$ circuit of size polynomial in s for some constant $\beta \geq 1$ that outputs $M_p^{\mathcal{O}^1,\ldots,\mathcal{O}^k}(x)$ whenever that computation halts in t time steps. The theorem statement follows immediately from the definition of KP. Furthermore, we describe our circuit as if it had access to $\pi_1^{\mathcal{O}}$, $\pi_2^{\mathcal{O}}$, $\eta_1^{\mathcal{O}}$, and $\eta_2^{\mathcal{O}}$ gates for several oracles $\mathcal{O}$; this is sufficient as one can simply replace π and η gates by the poly-size, constant-depth gadgets given by Lemma 5.3.

We begin by scanning the code for all registers that are explicitly named and renumbering them starting from 1, so that the largest register name is also the total number of distinct registers explicitly used by M. Note that the maximum number of arguments for any PRAM instruction is five. We use the code of M to create five finite oracles: $\mathcal{O}^{\text{op}}$, consisting of the names of the instructions in the program in order, and $\mathcal{O}^{\text{arg1}}, \ldots, \mathcal{O}^{\text{arg5}}$, consisting of the names of the arguments to those instructions. That is, $\mathcal{O}_i^{\text{op}}$ contains the name of the ith instruction in M's program, $\mathcal{O}_i^{\text{arg1}}$ contains the name of the first operand to the ith instruction, and so on. For instructions that have fewer than five arguments, the empty string appears in the appropriate sequence(s). Our circuit will use π and η gates for these oracles as well as the oracles $\mathcal{O}^1, \ldots, \mathcal{O}^k$.

The constructed circuit has p distinct columns of hardware, once for each processor, split into t constant-depth layers, one for each step of computation on the PRAM. There is a "global column" responsible for controlling access to global memory cells. It is easy to see that the number of global cells used by the PRAM during its execution is at most $t \cdot p$, with addresses no more than $2^{O(t+|M|)}$ (the largest value one can access using our arithmetic operations and hardcoded constants). While we do not know which cells these are *a priori*, we will allocate them dynamically. For each of the possible $t \cdot p$ cells, the global column contains $O(t + |M|)$ wires for the cell's "address," and two wires for its "value" (used to encode 0, 1, or $\perp$). The address is set to 0 if the cell wires are "unallocated." For each time step in the execution, the column is connected to circuitry in each processor column to read values from global memory and write values to global memory. The input is handled as global memory, but without the circuitry to write and with the addresses set to special canonical values. Output is handled as global memory with a special address which has circuitry to write but not to read. Additionally, there is a "halt flag" wire that will be set to 1 after the PRAM halts.

We focus now on a single layer of a single processor column, describing how one step of computation is accomplished. The input to this subcircuit is (1) the instruction counter, (2) the contents of local memory, and (3) the contents of global memory. The layer is computed in seven phases.

1. Decode Instructions. This phase maps the program counter to a one-hot indicator of which instruction should be executed at this step. Let ℓ_M be the maximum bit length of any argument in the code of M, taken across both explicitly named registers (after renaming) and constants that appear in the program. Since there are a constant number of types of instructions, each type of instruction can be identified with a constant number of bits m. We call these bits "opcodes." Place m $\pi_2^{\mathcal{O}^{\text{op}}}$ gates and ℓ_M $\pi_2^{\mathcal{O}^{\text{arg}i}}$ and $\eta_2^{\mathcal{O}^{\text{arg}i}}$ gates for each $i \in [5]$ in parallel (since m is a constant and ℓ_M cannot be larger than the size of the code for M, the number of π and η gates used here is definitely polynomial in s). Feed the program counter into the "row" input of each gate. Feed the hardcoded numbers $1, 2, \ldots, m$ into the "column" inputs of the m $\pi_2^{\mathcal{O}^{\text{op}}}$ gates respectively, and $1, 2, \ldots, \ell_M$ into the "column" inputs of the ℓ_M $\pi_2^{\mathcal{O}^{\text{arg}i}}$ and $\eta_2^{\mathcal{O}^{\text{arg}i}}$ gates, for each i. We will encode instructions in $\mathcal{O}^{\text{op}}$ so that every opcode has exactly m bits, which prevents any out-of-range indices in this step. When executed, this construction computes a complete instruction from the program, outputting $m + 10\ell_M$ bits: m bits containing an opcode, and ℓ_M pairs of bits for each operand consisting of

a "flag" bit (from the η gates) indicating whether that bit is in range and a value bit (from the π gates) containing the corresponding bit of the operand when defined.

Conjunctions of constant width suffice to create a one-hot vector of bits indicating which operation is to be executed in this time-step.

2. Fetch Operands. This phase looks at the $10\ell_M$ bits corresponding to operands that come from phase 1 and the contents of local and global memory, and outputs the values for the operands to be used during execution. These values may be hardcoded constants, contents of a register, or contents of a cell. By insisting on an appropriate bit encoding for operands in $\mathcal{O}^{\text{arg1}}, \ldots, \mathcal{O}^{\text{arg5}}$, we can ensure that it is sufficient to look at the first two bits of an operand to decide whether the operand is a register, a cell, or a hardcoded constant.

For each of the five operands and each register, we compute in parallel and in constant depth whether the operand is equal to the name of the register (one can overcome the issue of variable operand lengths via careful naming of registers and usage of the flag bits produced in phase 1). Using a bitwise AND between the equality bit produced and the value stored in the register, we obtain for each register a set of wires that are (a) all set to 0 if the register is not the one being referred to, and (b) set to the value of the register if it is being used.

Similarly, for each global cell we compute equality with the address wires corresponding to the global cell and compute the AND between the result and the value in the cell to obtain a set of wires as in the case of registers.

Finally, we obtain a set of wires that is (a) set to 0 if the operand is not a constant, and (b) set to the value of the constant if it is. The number of wires to use here can be precomputed by looking at the largest constant in the program.

Taking a bitwise OR of these sets of wires gives us the value of the operand to be used.

3. Compute and Compare Values. There are a total of 8 arithmetic/logic and control flow instructions. Each arithmetic/logic operation is binary, each control flow instruction makes a binary comparison. Furthermore, these comparisons and operations can be implemented by AC^0 circuits. Arrange all these circuits in parallel to compute on the first two operands output by the previous phase. The output of this phase contains the results of all possible local binary operations/comparisons implicit in the RAM instruction set.

4. Update Instruction Counter. If the current instruction is a conditional jump, then we check the condition computed in the previous phase and overwrite the instruction counter with one of the operands computed at phase 2 if appropriate. Otherwise, we increment the instruction counter. All these steps are clearly in AC^0. The output of this phase is the value of the instruction counter at step $t + 1$.

5. Determine Updates to Memory. This is identical to steps 6–8 of [SV84]. The output of this phase is the following:

- Based on the instruction being executed, the *result* of the current instruction if applicable or a flag indicating *no result* if not.
- An address of local or global memory where the result ought to be written. If the instruction is a `HALT` instruction this is set to a special predetermined value.
- A flag indicating if it is local or global memory that this processor wishes to update.

- A bit vector indicating which type of update to global memory is desired: write or some global operation allowed by the instruction set.

6. Update Local Memory. This is identical to step 9 of [SV84]—no write conflicts can occur in the local registers.

7. Update Global Memory. We describe how each type of global write operation can be implemented by constant-depth circuits. The final circuit has a copy of these circuits for every global cell and every type of operation and uses simple constant-depth circuitry to decide whether the global cell is being written to and if so which operation needs to be performed, and appropriately overwrites the cell wire if necessary.

If the update to global memory is triggered by a simple write, this is identical to step 10 of [SV84]. Write conflicts are resolved by computing a unique "write permission bit" by disjuncting over all processors that have smaller PID than the present processor, for each processor: if any of these processors has the "wish to update global memory" bit set and the same target address, then the processor with larger PID gets a zero permission indicator bit. In this way, *only the minimum PID processor* will be allowed to write to the address in question, and circuitry from the local memory update can be reused after this conflict-resolution subcircuit. [SV84] also shows how to dynamically assign global memory to addresses; we use that circuitry as is.

If the update is triggered by a `HALT` instruction, we check if the halt flag is already set. If it is, then no action is performed. Otherwise, we perform a simple write to the output wires and set the halt flag to 1.

Now suppose the update to global memory is triggered by a non-`ROM` global operation. By design, some set of processors P are simultaneously executing a global operation with target address `addr`. This is easy to detect from the output of the previous phase. Since we consider only symmetric Boolean functions for global operations, write conflicts are meaningless. In this case, conflicts are with respect to *which operation should be executed*—each processor has declared a type of operation that it wants applied to the target cell, computed at the previous phase. In the case of such a conflict we simply reuse the write-conflict resolution strategy above to select the minimum PID that would like to execute a global operation with target address `addr`, and write the result of that operation to the `addr` location of global memory. To execute the computation and subsequent write, we place one copy of each global operation gate (each global operation is in the basis $\mathcal{B}$ and thus we have unbounded-fan-in gates for it) and feed it one input wire from *every* processor. We set the input wire from a given processor to contain (a) the "result" field from phase 5 if the processor is involved in the global operation, or (b) the "padding" bit for the operation if the processor is not involved. It is easy to see that these input wires can be computed in parallel and by constant-depth circuitry.

Finally, if the global write is triggered by a `ROM1` or `ROM2` instruction to oracle $\mathcal{O}^i$, we use $\pi_1^{\mathcal{O}^i}$ and $\eta_1^{\mathcal{O}^i}$ or $\pi_2^{\mathcal{O}^i}$ and $\eta_2^{\mathcal{O}^i}$ gates respectively where each input to the gate is computed by looking at the row/column and index operands and using them to select the right input bits. Looking at the output of the η gate, we write either the bit value coming from the π gate or a $\perp$ to the global cell.

The output of this final phase is an updated bit-vector representing the state of local and global memory at the beginning of step $t + 1$.

Chaining t copies of phases 1–7 in each processor column and putting all the columns together, we get a circuit that simulates the PRAM step-by-step. The input is wired into the input part of the global memory and the output is the output wire.

It is easy to see that each phase is computed by a constant-depth, $\mathsf{poly}(s)$-size $\mathsf{AC}^0[h]$ circuit, as every operation performed is a constant-depth, $\mathsf{poly}(s)$-size operation. Let β be the total (constant) depth of the subcircuit chaining all seven phases together for a single time step. The depth of the final circuit then is exactly βt as claimed. Since the circuit contains p copies of each time-step in parallel and there are t time-steps, the total size of the final circuit is $t \cdot p \cdot \mathsf{poly}(s)$ which is polynomial in s since t is constant and $p \leq s$. Note that if t were super-constant, the proof would go through as-is, except the circuit-size upper bound would become $t \cdot \mathsf{poly}(s)$ instead of just $\mathsf{poly}(s)$. □

Corollary 5.5. *There is a constant $\kappa \geq 1$ such that $\mathsf{TC}^0_{\kappa t}\text{-}\mathsf{CC}(x) \leq \mathsf{poly}(\mathsf{KP}^t_{\mathsf{MAJ}}(x))$.*

Proof. The only part of the construction in the proof of Theorem 5.4 that does not work if MAJ is part of the basis is phase 7. To compute a global operation f, the construction computes an f gate with one input from each processor: the bit to be considered for processors that are participating, and a "padding" bit for processors that are not participating in the operation. Since MAJ does not admit a padding bit, that process does not work. However, we can overcome that by, instead of feeding the MAJ gate one wire per processor, feeding it *two* wires per processor. If a processor is participating in the global operation with a bit b, we set these wires to $b\|b$. If it is not participating, we set these wires to 01. It is easy to see that the majority function applied to the input formed by concatenating all these wires is equal to the majority function applied to just the participating processors' bits. The corollary follows. □

5.3 A Nontrivial Algorithm for $\mathsf{KP}\text{-}\mathsf{vs}\text{-}\mathsf{Random}_{\mathsf{MOD}p}$

Razborov and Rudich [RR97] showed that the proofs due to Razborov [Raz87] and Smolensky [Smo93] that the majority function cannot be computed by polynomial-size $\mathsf{AC}^0[p]$ circuits are *natural*, that is, they are witnessed by efficient algorithms (called "natural properties") that can distinguish low-complexity truth tables from truth tables of random functions. Theorem 5.4 shows that strings with low $\mathsf{KP}_{\mathsf{MOD}p}$ can be thought of as truth tables of low-complexity $\mathsf{AC}^0[p]$ circuits. Thus, one can straightforwardly combine Theorem 5.4 with the natural properties of Razborov and Rudich to obtain algorithms for $\mathsf{KP}\text{-}\mathsf{vs}\text{-}\mathsf{Random}_{\mathrm{MOD}p}$.

Theorem 5.6 (Natural Properties against $\mathsf{AC}^0[p]$ [RR97])**.** *For every prime p, constant depth parameter d, and constant $\delta \in (0,1)$ there is a polynomial-time algorithm $M_{p,d,\delta}$ such that the following are true.*

1. *(Largeness) For every sufficiently large n,*

$$\Pr_{\substack{f:\{0,1\}^n \to \{0,1\} \\ \textit{uniformly random}}} [M_{p,d,\delta}(tt(f)) = 0] \geq 1 - \delta.^{6}$$

2. *(Usefulness) There are constants $C, n_0 > 0$ such that for all $n \geq n_0$ and $f_n : \{0,1\}^n \to \{0,1\}$, if f_n can be computed by an $\mathsf{AC}^0_d[p]$ circuit of size at most $2^{Cn^{1/(2d)}}$ then*

$$M_{p,d,\delta}(tt(f_n)) = 1.$$

[6] Razborov and Rudich's original definition of natural properties only considered largeness $2^{-O(n)}$. [CIKK16, Lemma 2.7] shows that largeness for any natural property can be amplified to $1/2$. It is easy to see that if applied to a natural property that already has constant largeness, their amplification procedure yields a natural property with largeness $1 - \delta$ for any choice of $\delta \in (0,1)$.

Theorem 5.7. *For every prime p there exist constants $C', \beta > 0$ such that for every constant t and $s(n) \leq 2^{C' \log^{1/(2\beta t)} n}$, $\mathsf{KP\text{-}vs\text{-}Random}^t_{\mathsf{MOD}_p}[s]$ can be solved in polynomial time with error at most δ for every constant $\delta > 0$. In particular, $\mathsf{KP\text{-}vs\text{-}Random}^t_{\mathsf{MOD}_p}[\mathsf{polylog}(n)]$ can be solved in polynomial time with error at most δ for every constant $\delta > 0$.*

Proof. Fix a prime p. Let C' be a constant to be set later. Let g and β be the polynomial and constant given by Theorem 5.4 so that for all t and x, $\mathsf{AC}^0_{\beta t}[p]\text{-}\mathsf{CC}(x) \leq g(\mathsf{KP}^t_{\mathsf{MOD}_p}(x))$. Pick an arbitrary constant time bound t and $s(n) \leq 2^{C' \log^{1/(2\beta t)} n}$. Let $\delta > 0$ be any constant. Let $M = M_{p,\beta t,\delta}$ be the polynomial-time algorithm given by Theorem 5.6. We claim that M is an algorithm for $\mathsf{KP\text{-}vs\text{-}Random}^t_{\mathsf{MOD}_p}[s]$ with error at most δ on sufficiently large input lengths.

Suppose $x \in \{0,1\}^n$ is a YES instance, i.e., $\mathsf{KP}^t_{\mathsf{MOD}_p}(x) \leq s(n)$. Then by Theorem 5.4, there is an $\mathsf{AC}^0_{\beta t}[p]$ circuit of size $g(s(n))$ that computes f_x. Note that $g(s(n)) \leq g(2^{C' \log^{1/(2\beta t)} n}) \leq 2^{O(1) \cdot C' \log^{1/(2\beta t)} n}$. Setting C' to a sufficiently small constant thus ensures $g(s(n)) \leq 2^{C \log^{1/(2d)} n}$ where C is the constant from the usefulness condition of Theorem 5.6. Since f_x is a function on $\log n$ bits, this means that as long as n is sufficiently large, $M(tt(f_x)) = 1$. On the other hand, if $x \sim \{0,1\}^n$ is uniformly random, then the largeness property implies that as long as n is sufficiently large, M rejects with probability at least $1 - \delta$. □

It is easy to see that if subexponentially-secure one-way functions (and thus subexponentially-secure pseudorandom functions) exist, then the analogously defined "$\mathsf{K^{poly}vsRandom}[\mathsf{polylog}(n)]$" and "$\mathsf{KTvsRandom}[\mathsf{polylog}(n)]$" problems cannot be solved to constant error in polynomial time. In fact, Liu and Pass [LP21, LP24b] show that if subexponentially-secure one-way functions exist, then $\mathsf{K^{poly}vsRandom}[\mathsf{polylog}(n)]$ is hard even on average (as opposed to just worst-case over YES instances in our definition). Thus, Theorem 5.7 shows that under the assumption that subexponentially-secure one-way functions exist, distinguishing low-$\mathsf{KP}_{\mathsf{MOD}_p}$ strings from random is provably easier than distinguishing low-$\mathsf{K^{poly}}$ or -KT strings from random, at least when the threshold is $\mathsf{polylog}(n)$.

6 Connections to Learning

In this section we show that easiness of $\mathsf{KP\text{-}vs\text{-}Random}_h$ implies learning algorithms for $\mathsf{AC}^0[h]$ and, conversely, that learning algorithms for $\mathsf{AC}^0[h]$ imply algorithms for $\mathsf{KP\text{-}vs\text{-}Random}_h$ (albeit with a lower threshold).

6.1 Algorithms for KP Imply Learning

We show that easiness of $\mathsf{KP\text{-}vs\text{-}Random}$ implies learning algorithms by exhibiting distinguishers for DP^f for each $f \in \mathsf{AC}^0[h]$ and appealing to Lemma 2.5. This approach is inspired by the work of Goldberg and Kabanets [GK23].

To construct our distinguisher, we show that a shuffled version of the direct product generator, $\widetilde{\mathsf{DP}}^f$, produces outputs with non-trivially low KP complexity. Random strings, on the contrary, have near-maximal KP complexity by Proposition 4.4. We need to consider a shuffled version of DP^f because the indexing arithmetic required to compute $\mathsf{DP}(x^1, x^2, \ldots, x^m)$ itself incurs too much overhead on constant-time PRAMs for the distinguisher we construct to be effective. Our shuffling simply places the evaluations $f(x^1), \ldots, f(x^m)$ before the strings $x^1, \ldots, x^m$.

Lemma 6.1. *For every $f : \{0,1\}^n \to \{0,1\}$ and $x \in (\{0,1\}^n)^m$, define $\widetilde{\mathsf{DP}}^f(x)$ as*

$$\widetilde{\mathsf{DP}}^f(x) = f(x^1) \| \cdots \| f(x^m) \| x^1 \| \cdots \| x^m.$$

1. Input: $i \in [nm+m]$. Oracle: $\mathcal{O}^1$ such that $\mathcal{O}^1_{a,b} = x^a_b$.
2. Use the simulation of AC^0 circuits from Theorem 5.1 to check whether $i \leq m$, with m hardcoded in the program.
3. If $i > m$, then use the simulation of AC^0 circuits from Theorem 5.1 to simulate a circuit for integer subtraction. As a result, for $j \in [\log nm]$, the processor with PID j should contain the jth least significant bit in the binary representation of $i - m$.
4. Use a `ROM1` instruction to compute $\mathsf{flat}(\mathcal{O}^1)_{i-m}$ to a global cell. Then read the value and output it.
5. If $i \leq m$, then use read instructions and conditioning on PIDs to load the $\log m$ least significant bits of the input i into the first $\log m$ processors. That is, for $j \in [\log m]$, the processor with PID j should read the jth least significant bit of i.
6. At the same time, designate n groups of $\log n$ processors each, and load each of them with the binary representations of the numbers $1, 2, \ldots, n$. Processor number $j \in [\log n]$ within group $k \in [n]$ should get the jth least significant bit in the binary representation of the number k. This can be done via simple arithmetic and conditioning on PIDs.
7. Use `ROM2` instructions (in parallel) to copy $x^i_1, x^i_2, \ldots, x^i_n$ to n global cells.
8. Run the $\mathcal{B}$-PRAM to compute f given by Theorem 5.1, replacing `INPUT` instructions with global read instructions to the cells containing $x^i_1, \ldots, x^i_n$.

Figure 6: PRAM computing $\widetilde{\mathsf{DP}}^f(x)$.

For all $c \geq 1$ and symmetric Boolean functions h, there exists $t = O(d)$ such that for all $d \in \mathbb{N}$, $f \in \mathsf{AC}^0_d[h]\text{-}\mathsf{SIZE}[n^c]$, and $x \in (\{0,1\}^n)^m$, $\mathsf{KP}^t_h(\widetilde{\mathsf{DP}}^f(x)) \leq nm + \widetilde{O}(n^c)$.

Proof. Fix any $d \in \mathbb{N}$, $c \geq 1$, h, $f \in \mathsf{AC}^0_d\text{-}\mathsf{SIZE}[n^c]$, and $x \in (\{0,1\}^n)^m$. Set $\mathcal{B} = \mathcal{B}^0 \cup \{h\}$. Let $\mathcal{O}^1$ be a finite oracle containing m strings of length n, with $\mathcal{O}^1_{a,b} = x^a_b$.

We claim that the $\mathcal{B}$-PRAM M in Figure 6 is such that for some $q = \widetilde{O}(n^c)$, for all $i \in [nm+m]$, $M^{\mathcal{O}^1}_q(i) = \widetilde{\mathsf{DP}}^f(x)_i$. Step 2 checks whether the output should be a bit from x (which happens when $i > m$) or $f(x^i)$ (which happens when $i \leq m$). In the first case, the desired output is $(x^1 \| \cdots \| x^m)_{i-m}$ which, by definition of flat, is $\mathsf{flat}(\mathcal{O}^1)_{i-m}$. Steps 3 and 4 thus handle this case by computing $i - m$ and using a `ROM1` instruction to obtain $\mathsf{flat}(\mathcal{O}^1)_{i-m}$. In the latter case, the desired output is $f(x^i)$. Steps 5–7 use simple PRAM operations to load the string x^i into global cells. Step 8 uses the simulation from Theorem 5.1 to simulate the optimal $\mathsf{AC}^0_d[h]$ circuit for f.

Step 2 runs in constant time and uses $O(\mathsf{polylog}(m))$ processors because m is an $O(\log m)$-bit number, and integer comparison is in AC^0. Similarly, step 3 runs in constant time and uses $O(\mathsf{polylog}(nm))$ processors because integer subtraction is in AC^0. Step 4 runs in constant time and uses the processors from step 3. Steps 5–7 run in constant time and require $O(n \log n + \log m)$ processors. Finally, step 8 can be implemented in time $O(d)$ with $\widetilde{O}(n^c)$ processors by Theorem 5.1. Thus the running time of the PRAM is $O(d)$.

Steps 1–7 have program length $O(\log n + \log m)$ since all hardcoded constants (global cell addresses, shifts, etc.) are at most $\max(n, m)$ in value. By Theorem 5.1, step 8 has program length $\widetilde{O}(n^c)$. Thus $|M| = O(\log n + \log m) + \widetilde{O}(n^c) = \widetilde{O}(n^c)$. One might worry that combining these two "snippets" of code might incur overhead, as one needs to shift any hardcoded jump targets in the code for f given by Theorem 5.1 to account for the additional instructions that are now prepended. However, this is not a problem as steps 1–7 only contain a constant number of instructions, which

means that hardcoded targets in the code for f need only be shifted by an additive constant. As a result, the overhead incurred in combining them is at most a constant factor.

Since $|\mathcal{O}^1| = nm$, we have $|\langle M, \mathcal{O}^1\rangle| = |M| + |\mathcal{O}^1| + O(\log \mathsf{size}(\mathcal{O})) = \widetilde{O}(n^c) + nm + O(\log nm) = nm + \widetilde{O}(n^c)$. □

We are ready to prove our main theorem on learning.

Theorem 6.2. *For every $c \geq 1$ and $d \in \mathbb{N}$ there exist $s(n) = n^{c+2} + \widetilde{O}(n^c)$, and $t = O(d)$ such that if there is a (randomized) polynomial-time solver for the $\mathsf{KP\text{-}vs\text{-}Random}^t_h[s(n)]$ problem on input length $\ell(n) = n^{c+2} + n^{c+1}$ with error $1/3$, then $\mathsf{AC}^0_d[h]\text{-}\mathsf{SIZE}[n^c]$ can be PAC-learned over the uniform distribution in (randomized) polynomial time to error $1/2 - 1/O(n^{c+1})$ and confidence $1 - 1/\mathsf{poly}(n)$. Furthermore, if $\oplus \in \mathsf{AC}^0_d[h]\text{-}\mathsf{SIZE}[n^c]$, then the learner's error can be reduced to an arbitrary $1/\mathsf{poly}(n)$.*

Proof. Fix $c \geq 1$ and $d \in \mathbb{N}$. Define $m = n^{c+1}$. Let $\ell(n) = n^{c+2} + n^{c+1} = nm(n) + m(n)$ and let $s(n)$ be the $nm(n) + \widetilde{O}(n^c)$ upper bound from Lemma 6.1. Note that due to our setting of m, $s(n) = n^{c+2} + \widetilde{O}(n^c)$ as in the theorem statement. Finally, let $t = O(d)$ be the time bound in the simulation of depth-d $\mathsf{AC}^0[h]$ circuits given by Theorem 5.1. By Lemma 2.5, it suffices to show that a $1/3$-error solver for $\mathsf{KP\text{-}vs\text{-}Random}^t_h[s(n)]$ on input length $\ell(n)$ implies a distinguisher B for the m-fold direct product generator.

Note that since $m(n) = \widetilde{\omega}(n^c)$, we have $s(n) \ll \ell(n) - 2\log \ell(n)$ and thus the problem $\mathsf{KP\text{-}vs\text{-}Random}^t_h[s(n)]$ on input length $\ell(n)$ is non-degenerate. Let A solve $\mathsf{KP\text{-}vs\text{-}Random}^t_h[s(n)]$ on input length $\ell(n)$ with error $1/3$. Our distinguisher B works as follows. Given input $(1^n, y)$ with $y \in \{0,1\}^{nm(n)+m(n)}$,

1. Break y into strings $x^1, \ldots, x^{m(n)} \in \{0,1\}^n$ and $r \in \{0,1\}^{m(n)}$.
2. Compute the string $z = r \| x^1 \| \cdots \| x^{m(n)}$.
3. Output $A(z)$.

Since A runs in polynomial time, so does B. Suppose $y = \mathsf{DP}^f(x)$ for some $f \in \mathsf{AC}^0_d[h]\text{-}\mathsf{SIZE}[n^c]$. Then for all $a \in [m(n)]$, $r_a = f(x^a)$, and hence $z = \widetilde{\mathsf{DP}}^f(x)$. By Lemma 6.1, $\mathsf{KP}^t_h(z) \leq nm(n) + \widetilde{O}(n^c)$. It follows by correctness of A that $A(z) = 1$ with probability at least $2/3$. On the other hand, if y is a uniformly random string then so is z. Thus, also by correctness of A, $\Pr_{y \sim \{0,1\}^\ell}[B(1^n, y) = 1] = \Pr_{z \sim \{0,1\}^\ell}[A(z) = 1] \leq 1/3$.

It follows that for every $f \in \mathsf{AC}^0_d[h]\text{-}\mathsf{SIZE}[n^c]$,

$$\Pr_{x \sim \{0,1\}^{nm(n)}}[B(1^n, \mathsf{DP}^f(x)) = 1] - \Pr_{r \sim \{0,1\}^{nm(n)+m(n)}}[B(1^n, r) = 1] \geq \frac{1}{3} \geq \frac{1}{10}$$

as desired. The "furthermore" part follows due to the boosting result of Boneh and Lipton [BL93]. □

We observe that the condition $(\mathsf{MKPP}_h, \mathcal{U}) \in \mathsf{AvgBPP}$ implies easiness of $\mathsf{KP\text{-}vs\text{-}Random}^t_h[s]$ with the necessary setting of parameters, and is thus also sufficient for learning $\mathsf{AC}^0_d[h]$.

Theorem 6.3. *For all symmetric functions h, if $(\mathsf{MKPP}_h, \mathcal{U}) \in \mathsf{AvgBPP}$ then for all $d \in \mathbb{N}$, $c \geq 1$, the concept class $\mathsf{AC}^0_d[h]\text{-}\mathsf{SIZE}[n^c]$ can be PAC-learned over the uniform distribution in randomized polynomial time to error $\varepsilon(n) = 1/2 - 1/\mathsf{poly}(n)$ and confidence $1 - 1/\mathsf{poly}(n)$ for sufficiently large n. Furthermore, if $\oplus \in \mathsf{AC}^0_d[h]\text{-}\mathsf{SIZE}[n^c]$, then the learner's error can be reduced to an arbitrary $1/\mathsf{poly}(n)$.*

Proof. Let A be a randomized errorless heuristic for $(\mathsf{MKPP}_h, \mathcal{U})$ with failure probability δ set to $1/n$. By Theorem 6.2 there exist $m(n) = \mathsf{poly}(n)$, $s(n) = nm + \widetilde{O}(n^c)$, and $t = O(d)$ such that it suffices to show that $\mathsf{KP\text{-}vs\text{-}Random}^t_h[s(n)]$ on input length $\ell = nm(n) + m(n)$ can be solved in polynomial time to error $1/3$. Let A' be the algorithm that, given a string $x \in \{0,1\}^\ell$, runs $A(x, 1^t, 1^{s(n)})$ and outputs 1 iff the result is 1 or $\perp$. Since A runs in polynomial time, so does A'. We show that A' solves $\mathsf{KP\text{-}vs\text{-}Random}$ with the required parameters.

Suppose x is a YES instance of $\mathsf{KP\text{-}vs\text{-}Random}$, i.e., $\mathsf{KP}^t_h(x) \leq s(n)$. Then it follows from the definition of errorless heuristics and MKPP_h that $A(x, 1^t, 1^{s(n)}) \in \{1, \perp\}$ except with probability $1/10$. Hence A' outputs 1 except with probability $1/10$.

On the other hand,

$$\Pr_{x\sim\{0,1\}^\ell}[A'(x) = 1] = \Pr_{x\sim\{0,1\}^\ell}[A(x, 1^t, 1^{s(n)}) = 1] + \Pr_{x\sim\{0,1\}^\ell}[A(x, 1^t, 1^{s(n)}) = \perp].$$

Since A is an errorless heuristic for $(\mathsf{MKPP}_h, \mathcal{U})$,

$$\begin{aligned}\Pr_{x\sim\{0,1\}^\ell}[A(x, 1^t, 1^{s(n)}) = \perp] &\leq \Pr_{x\sim\{0,1\}^\ell}\left[\Pr[A(x, 1^t, 1^{s(n)}) = \perp] \geq 1/10\right] + \frac{1}{10} \\ &\leq \frac{1}{n} + \frac{1}{10}.\end{aligned}$$

Also,

$$\begin{aligned}\Pr_{x\sim\{0,1\}^\ell}[A(x, 1^t, 1^{s(n)}) = 1] \leq &\Pr_{x\sim\{0,1\}^\ell}[(x, 1^t, 1^{s(n)}) \in \mathsf{MKPP}_h] \\ &+ \Pr_{x\sim\{0,1\}^\ell}[A(x, 1^t, 1^{s(n)}) \notin \{0, \perp\} \mid (x, 1^t, 1^{s(n)}) \notin \mathsf{MKPP}_h].\end{aligned}$$

By Proposition 4.4, the first term is at most $1/n$. The second term is at most $1/10$ since A is an errorless heuristic for MKPP. Putting everything together we get that $\Pr_{x\sim\{0,1\}^\ell}[A'(x) = 1] \leq 2/10 + 2/n < 1/3$. □

6.2 Learning Implies Algorithms for KP

In the other direction, we show that strong learning algorithms for $\mathsf{AC}^0[h]$ imply algorithms for the $\mathsf{KP\text{-}vs\text{-}Random}_h$ problem. To do so, we will need the following lemma that shows that Turing machines with short descriptions cannot approximate random strings.

Lemma 6.4. *For every $\delta > 0$, $\ell \in \mathbb{N}$, and n, with probability at least*

$$1 - 2^{-\frac{1}{2}n(1-2\delta)^2 \log e + \ell}$$

over a uniformly random string $x \sim \{0,1\}^n$, there is no Turing machine M with description length[7] less than ℓ such that

$$\Pr_{i\sim[n]}[M(i) = x_i] \geq 1 - \delta.$$

Proof. We show that for any fixed Turing machine M,

$$\Pr_{x\sim\{0,1\}^n}\left[\Pr_{i\sim[n]}[M(i) = x_i] \geq 1 - \delta\right] \leq e^{-\frac{n(1-2\delta)^2}{2}}.$$

[7] "Description length" here is defined with respect to some fixed reasonable universal Turing machine.

The statement in the lemma then follows from a union bound over the 2^ℓ Turing machines with description length less than ℓ. To prove the inequality above, fix a machine M and let $z_i^{(x)}$ be a random variable that is 1 if $M(i) = x_i$ and 0 otherwise. Then

$$\Pr_{i\sim[n]}[M(i) = x_i] = \frac{1}{n}\sum_{i=1}^{n} z_i^{(x)},$$

and so

$$\Pr_{x\sim\{0,1\}^n}\left[\Pr_{i\sim[n]}[M(i) = x_i] \geq 1-\delta\right] = \Pr_{x\sim\{0,1\}^n}\left[\sum_{i=1}^{n} z_i^{(x)} \geq n(1-\delta)\right].$$

Note that since M is a fixed deterministic machine, when $x \sim \{0,1\}^n$, the $z_i^{(x)}$ are independent Bernoulli(1/2) variables. Thus the probability above is simply the probability that the sum of n independent Bernoilli(1/2) random variables is at least $n(1-\delta)$. The desired inequality follows by a standard Hoeffding bound. □

Theorem 6.5. *For every bit-paddable h there exist constants $\gamma \geq 1$ and $\beta \geq 1$ such that for all constants $c, c' > 0$, if there is a $T(n)$-time PAC-learner for $\mathsf{AC}^0_{\beta t}[h]$-$\mathsf{SIZE}[n^c]$ over the uniform distribution with error $1/\mathsf{poly}(n)$ and confidence $1-\delta$ outputting hypothesis circuits of size at most $n^{c'}$, then for every $\log^{c/\gamma} n \leq s(n) = O(n^{0.99c/(\gamma c')})$, KP-vs-$\mathsf{Random}^t_h[s]$ can be solved in time $T(O(n^{1/c'})) + \mathsf{poly}(n)$ with correctness probability $1 - \delta - \mathsf{negl}(n)$.*[8]

Proof. Fix h and let p and $\beta \geq 1$ be the polynomial and constant from the simulation of PRAMs by circuits given by Theorem 5.4. Let $\gamma \geq 1$ be a constant such that $p(n) \leq n^\gamma$. Let A be a strong PAC learner for $\mathsf{AC}^0_{\beta t}[h]$-$\mathsf{SIZE}[n^c]$ that outputs hypothesis circuits of size at most $n^{c'}$ (where n is the number of input bits to the function being learned) and has confidence $1-\delta$. Let q be a polynomial such that the learner has error at most $1/q(n)$.

Let $\log^{c/\gamma} n \leq s(n) = O(n^{0.99c/(\gamma c')})$ be arbitrary. We show that the following algorithm, when given $x \in \{0,1\}^n$, solves the KP-vs-$\mathsf{Random}^t_h[s]$ problem.

1. Compute $m = s(n)^{\gamma/c}$.
2. Define $g : \{0,1\}^m \to \{0,1\}$ as $g(y) = x_{y_1\|y_2\|\cdots\|y_{\log n}}$. This function is well-defined since $s(n) \geq \log^{c/\gamma} n$, which ensures $m \geq \log n$.
3. Simulate running A with oracle $\mathsf{Ex}(g)$ to obtain a circuit C.
4. Estimate $\Pr_{y\sim\{0,1\}^m}[C(y) \neq g(y)]$ to additive error $1/q(m)$. This can be done in $\mathsf{poly}(n)$ time with probability $1 - 2^{-n^{\Omega(1)}}$ by picking polynomially many random inputs y, computing the fraction of the sample that is incorrectly computed by C, and using a Chernoff bound to bound the probability that the estimate is $1/q(m)$ away from the true value.
5. Output YES if the estimate is at most $2/q(m)$ and NO otherwise.

Step 3 of the algorithm runs in time $T(m) = T(s(n)^{\gamma/c}) = T(O(n^{1/c}))$. All other steps run in time polynomial in n. The bound on the running time follows.

Suppose $\mathsf{KP}^t_h(x) \leq s(n)$. Then, by the simulation in Theorem 5.1, the function $f_x : \{0,1\}^{\log n} \to \{0,1\}$ with $f_x(i) = x_i$ is computable by an $\mathsf{AC}^0_{\beta t}[h]$ circuit of size at most $p(s(n)) \leq s(n)^\gamma = m^c$. It

[8] Note that the circuits output by the PAC learner need not themselves be $\mathsf{AC}^0[h]$ circuits. The result holds for every choice of finite computable basis for the hypothesis circuits.

follows that g can be computed by a circuit of size m^c, namely the circuit that simply runs the circuit for f_x on the first $\log n$ input wires and ignores the remaining input wires. Thus by correctness of the PAC learner, the circuit output by $A^{\mathsf{Ex}(g)}$ is $1/q(m)$-close to g except with probability δ. Since the estimation in step 4 succeeds except with probability $2^{-n^{\Omega(1)}} = \mathsf{negl}(n)$, the algorithm outputs YES with probability at least $1 - \delta - \mathsf{negl}(n)$.

Now suppose x is a random n-bit string and let $f_x : \{0,1\}^{\log n} \to \{0,1\}$ be the function such that $f_x(i) = x_i$. If the estimation in step 4 succeeds, the algorithm only outputs YES if A outputs a circuit C of size $m^{c'}$ such that $\Pr_{y\sim\{0,1\}^m}[C(y) \neq g(y)] \leq 3/q(m)$. It suffices to show that such a circuit C only exists with probability $\mathsf{negl}(n)$ since the estimation succeeds except with probability $\mathsf{negl}(n)$. Suppose there is a circuit C of size $m^{c'}$ such that $\Pr_{y\sim\{0,1\}^m}[C(y) \neq g(y)] \leq 3/q(m)$. For every $i \in \{0,1\}^{\log n}$ and $z \in \{0,1\}^{m-\log n}$, $g(i\|z) = f_x(i)$. By averaging there must exist some $z^* \in \{0,1\}^{m-\log n}$ such that $\Pr_{i\sim\{0,1\}^{\log n}}[C(i\|z^*) \neq f_x(i)] \leq 3/q(m)$. Thus there is a circuit $\widetilde{C} : \{0,1\}^{\log n} \to \{0,1\}$ of size at most $m^{c'}$ such that $\Pr_{i\sim\{0,1\}^{\log n}}[\widetilde{C}(i) \neq f_x(i)] \leq 3/q(m)$, namely the circuit $\widetilde{C}(i) = C(i\|z^*)$. Since one can describe $\widetilde{C}$ with $O(m^{c'}\log m)$ bits, there is a Turing machine M with description size $O(m^{c'}\log m)$ such that $\Pr_{i\sim\{0,1\}^{\log n}}[M(i) = f_x(i)] \geq 1 - 3/q(m)$. By Lemma 6.4, the probability that such a machine exists for a random x is at most

$$\begin{aligned}\exp_2\left(-\frac{\log e}{2}n\left(1-\frac{3}{q(m)}\right)^2 + O(m^{c'}\log m)\right) &= \exp_2\left(-\Omega(n) + O(s(n)^{\gamma c'/c}\log n)\right)\\ &= \exp_2\left(-\Omega(n) + O(n^{0.99}\log n)\right)\\ &= \mathsf{negl}(n).\end{aligned}$$

□

Theorem 6.5 says that for $\mathsf{AC}^0_d[h]$ functions of bounded polynomial size to be strongly PAC-learnable by a polynomial-time algorithm, it is *necessary* for n-bit strings with $\mathsf{KP}_h^{O(d)}$ less than n^ε for some $\varepsilon > 0$ to be distinguishable from random strings. This gives evidence that computing KP is not "overkill" for learning constant-depth circuits. Furthermore, note that if one-way functions exist, then such distinguishability is *false* for $\mathsf{K}^{\mathsf{poly}}$ and KT. In fact, if quasipolynomially-secure one-way functions exist, then the corresponding problem for $\mathsf{K}^{\mathsf{poly}}$ is even hard on average [LP21, LP24b]. Thus Theorem 6.5 also shows (under mild cryptographic assumptions) that any technique that uses KP_h to learn $\mathsf{AC}^0[h]$ must leverage properties of KP that do not hold also for $\mathsf{K}^{\mathsf{poly}}$ and KT.

7 Connections to Constant-Depth Cryptography

In this section we prove that average-case hardness (under a suitable definition) of $\mathsf{KP\text{-}vs\text{-}Random}_h$ implies PRFs computable by constant-depth $\mathsf{AC}^0[h]$ circuits with size polynomial in the security parameter. Conversely, we show that PRFs *jointly* computable by *uniform* polynomial-size $\mathsf{AC}^0[h]$ circuits imply average-case hardness (under the same definition) of $\mathsf{KP\text{-}vs\text{-}Random}_h$ with a *logarithmic* time bound. The forward direction closely follows a construction of strong PRFs from hardness of K^t due to Liu and Pass [LP24a]. The other direction follows the work of Ren and Santhanam [RS21] which shows that cryptography in NC^0 is equivalent to hardness of KT. We begin by describing our notion of average-case hardness.

7.1 Average-Case* Hardness

The average-case hardness notion we use is based on a definition of average-case hardness with respect to two-sided heuristics due to Liu and Pass [LP21]. The reason we use that definition is that

it allows for meaningful two-sided errors even for *sparse* languages such as the set of all low-KP strings.

Definition 7.1 (Normalized input length)**.** The *normalized input length* of a language $L \subseteq \{0,1\}^*$ is defined as $n^*_L(n) = \log |L \cap \{0,1\}^n|$. We define $n^*_{t,h,s} = n^*_{\mathsf{MKPP}^t_h[s]}$. We omit t, h, and s and simply write n^* where they are clear from context.

Proposition 7.2. *There is a constant $c > 0$ such that for all $t \geq 5$, symmetric Boolean functions h, and $s(n) < n$,*

$$2^{s(n) - c\log(s(n))} \leq |\mathsf{MKPP}^t_h[s] \cap \{0,1\}^n| \leq 2^{s(n)+1}.$$

Thus $s(n) - c\log(s(n)) \leq n^* \leq s(n) + 1$.

Proof. Fix $t \geq 5$, h, and $s(n)$. The upper bound was already proved in the proof of Proposition 4.4, where we showed that for any basis $\mathcal{B}$ (and hence for the specific basis $\mathcal{B}^0 \cup \{h\}$) the number of strings $x \in \{0,1\}^n$ such that $\mathsf{KP}^t_{\mathcal{B}}(x) \leq s(n)$ is at most $2^{s(n)+1}$.

We now prove the lower bound. Let $m = s(n) - c\log(s(n))$ where c is a constant to be set later. For every $y \in \{0,1\}^m$, define $x^{(y)} \in \{0,1\}^n$ by $x^{(y)}_i = y_j$ where j is the number formed by the first $\log m$ bits of i. We claim that for every y, $\mathsf{KP}^t_h(x^{(y)}) \leq s(n)$. By Proposition 4.3 there exists $(\Pi, q) \in \mathcal{X}_{m+O(\log m)}$ such that for all $j \in [m]$, $\Pi^t_q(j) = y_j$. Let Π' be a machine that simply runs Π on the first $\log m$ bits of its input. Note that Π' has the same description length as Π. By definition of $x^{(y)}$, for all $i \in [n]$, $\Pi'^t_q(i) = x^{(y)}_i$. Thus $\mathsf{KP}^t_h(x^{(y)}) \leq m + O(\log m) = s(n) - c\log(s(n)) + O(\log(s(n))) \leq s(n)$ as long as the constant c is sufficiently large. The claim is thus true. The lower bound follows since all the $x^{(y)}$s are distinct, and hence we have exhibited $2^m = 2^{s(n) - c\log(s(n))}$ distinct elements of $\mathsf{MKPP}^t_h[s] \cap \{0,1\}^n$. □

Definition 7.3 (Average-case* Hardness)**.** Let $\mathcal{D} = \{D_n\}_{n\in\mathbb{N}}$ be an ensemble of distributions where each D_n is supported over $\mathsf{MKPP}^t_h[s] \cap \{0,1\}^n$. We say that $\mathsf{KP\text{-}vs\text{-}Random}^t_h[s]$ is (T,α)-*hard on average* with respect to* $\mathcal{D}$ if for all $T(n)$-time algorithms A and for all but finitely many input lengths n, either

$$\Pr_{x\sim D_n}[A(x) = 1] < 1 - \alpha(n^*), \text{ or}$$
$$\Pr_{x\sim\{0,1\}^n}[A(x) = 0] < 1 - \alpha(n^*).$$

We say that $\mathsf{KP\text{-}vs\text{-}Random}^t_h[s]$ is (T,α)-hard on average* with respect to the *uniform distribution* if D_n is the uniform distribution over $\mathsf{MKPP}^t_h[s] \cap \{0,1\}^n$.

That the error probability is allowed to depend on the normalized input length instead of simply n ensures that this notion is meaningfully two-sided. For example, if the error probability depended on n, then a $1/n^c$-error algorithm would not be able to make *any* mistakes on YES instances when the number of such instances is less than n^c, making the notion essentially one-sided. The problem $\mathsf{KP\text{-}vs\text{-}Random}^t_h[s]$ has a number of YES instances less than n^c when we consider thresholds $s(n) = \mathsf{polylog}(n)$.

Definition 7.4 (Universal distribution)**.** The *s-universal distribution*, $\mathcal{D}^{\text{s-univ}} = \{D^{\text{s-univ}}_n\}_{n\in\mathbb{N}}$, is the distribution sampled as follows.

1. Sample a uniformly random pair $(\Pi, q) \in \mathcal{X}_{s(n)}$.[9]

[9] One can sample uniformly from $\mathcal{X}_{s(n)}$ as long as our encoding of PRAMs as binary strings is reasonable.

2. Let $x = \Pi_q^t(1) \| \Pi_q^t(2) \| \cdots \| \Pi_q^t(n)$.

3. Output x.

Lemma 7.5. *Let c be the constant from Proposition 7.2. For all $t \geq 5$, h, and $s(n) < n$, if $\mathsf{KP\text{-}vs\text{-}Random}_h^t[s]$ is (T, α)-hard on average* with respect to the uniform distribution, then $\mathsf{KP\text{-}vs\text{-}Random}_h^t[s]$ is (T, α')-hard on average* with respect to the s-universal distribution where $\alpha'(n^*) = \frac{\alpha(n^*)}{2s(n)^c}$. If $\alpha(n^*) \geq 1/\mathsf{poly}(n^*)$ then $\alpha'(n^*) \geq 1/\mathsf{poly}(n^*)$ (for a different choice of polynomial).*

Proof. Fix t, h, and s. Let $\alpha'(n^*) = \frac{\alpha(n^*)}{2s(n)^c}$ and suppose $\mathsf{KP\text{-}vs\text{-}Random}_h^t[s]$ is not (T, α')-hard on average* with respect to the s-universal distribution. Then, by definition of average-case* hardness, there is a T-time algorithm A such that for infinitely many n,

$$\Pr_{x \sim D_n^{s\text{-univ}}}[A(x) = 1] \geq 1 - \alpha'(n^*), \text{ and}$$
$$\Pr_{x \sim \{0,1\}^n}[A(x) = 0] \geq 1 - \alpha'(n^*).$$

We show that on these input lengths, $\Pr_{x \sim D_n}[A(x) \neq 1] \leq \alpha(n^*)$ where D_n is the uniform distribution over $\mathsf{MKPP}_h^t[s] \cap \{0,1\}^n$. By definition and since $\alpha' < \alpha$, this means that $\mathsf{KP\text{-}vs\text{-}Random}_h^t[s]$ is not (T, α)-hard on average* with respect to the uniform distribution, proving the contrapositive of the lemma statement.

As mentioned in the proof of Proposition 7.2, there are at most $2^{s(n)+1}$ pairs (Π, q) in $\mathcal{X}_{s(n)}$, and furthermore every $x \in \mathsf{MKPP}_h^t[s] \cap \{0,1\}^n$ is computed in time t by at least one of these pairs. It follows that for every fixed $x \in \mathsf{MKPP}_h^t[s] \cap \{0,1\}^n$, when one picks (Π, q) at random from $\mathcal{X}_{s(n)}$, the pair computes x in time t with probability at least $1/2^{s(n)+1}$. That is, for every $x \in \mathsf{MKPP}_h^t[s] \cap \{0,1\}^n$,

$$\Pr_{x' \sim D_n^{s\text{-univ}}}[x' = x] \geq \frac{1}{2^{s(n)+1}}.$$

By Proposition 7.2, $|\mathsf{MKPP}_h^t[s] \cap \{0,1\}^n| \geq 2^{s(n) - c\log(s(n))}$. It follows that for every $x \in \mathsf{MKPP}_h^t[s] \cap \{0,1\}^n$,

$$\Pr_{x' \sim D_n}[x' = x] \leq \frac{1}{2^{s(n) - c\log(s(n))}}.$$

Hence,

$$\begin{aligned}
\alpha'(n^*) \geq \Pr_{x \sim D_n^{s\text{-univ}}}[A(x) \neq 1] &= \sum_{x : \mathsf{KP}_h^t(x) \leq s(n)} \Pr_{x' \sim D_n^{s\text{-univ}}}[x' = x] \cdot \Pr_A[A(x) \neq 1] \\
&\geq \sum_{x : \mathsf{KP}_h^t(x) \leq s(n)} \frac{1}{2^{s(n)+1}} \cdot \Pr_A[A(x) \neq 1] \\
&= \frac{1}{2^{1 + c\log(s(n))}} \sum_{x : \mathsf{KP}_h^t(x) \leq s(n)} \frac{1}{2^{s(n) - c\log(s(n))}} \cdot \Pr_A[A(x) \neq 1] \\
&\geq \frac{1}{2^{1 + c\log(s(n))}} \sum_{x : \mathsf{KP}_h^t(x) \leq s(n)} \Pr_{x' \sim D_n}[x' = x] \cdot \Pr_A[A(x) \neq 1] \\
&= \frac{1}{2^{1 + c\log(s(n))}} \Pr_{x \sim D_n}[A(x) \neq 1],
\end{aligned}$$

and so $\Pr_{x\sim D_n}[A(x) \neq 1] \leq \alpha'(n^*) \cdot 2s(n)^c = \alpha(n^*)$.

Finally, suppose $\alpha(n^*) \geq \frac{1}{\gamma(n^*)^\beta}$. Note that by Proposition 7.2, $0.99s(n) \leq s(n) - c\log(s(n)) \leq n^*$ and so $s(n) \leq \frac{n^*}{0.99}$. So $\alpha'(n^*) = \frac{\alpha(n^*)}{2s(n)^c} \geq \frac{0.99^c}{2\gamma(n^*)^\beta \cdot (n^*)^c} = 1/\mathsf{poly}(n^*)$. □

7.2 A PRF from Hardness of KP-vs-Random

We now prove that for every bit-paddable function h such that $\oplus \in \mathsf{AC}^0[h]$, if $\mathsf{KP\text{-}vs\text{-}Random}_h$ is hard on average* even for n^3-time adversaries then there exists a strong PRF computable by constant-depth $\mathsf{AC}^0[h]$ circuits with size polynomial in the security parameter. We construct our PRF assuming $\mathsf{KP\text{-}vs\text{-}Random}^t_h[s]$ is hard on average* over the s-universal distribution. Lemma 7.5 shows that hardness over the uniform distribution is also sufficient for the existence of such PRFs.

We begin by outlining the parameters that go into the construction.

1. Let $t \geq 5$ be an arbitrary constant and h be a bit-paddable function such that $\oplus \in \mathsf{AC}^0[h]$.
2. We assume that there exists a constant $\beta > 0$ such that $\mathsf{KP\text{-}vs\text{-}Random}^t_h[s]$ is α-hard on average* with respect to the s-universal distribution with $s(n) < n$ and $\alpha(n^*) = \frac{1}{O((n^*)^\beta)}$. We assume that s is monotonically increasing and thus has a well-defined inverse s^{-1}.
3. The security parameter for our construction is λ, and all other parameters depend on it.
4. Our analysis will use a distinguisher for our PRF to break the hardness assumption. The input length at which we will break the assumption is $n(\lambda) = s^{-1}(\lambda)$.
5. The input domain of our PRF is $\{0,1\}^{\ell(\lambda)}$ where $\ell(\lambda) = \frac{1}{8}\log(n(\lambda))$.
6. The construction has a parallel repetition parameter $\gamma(\lambda) = O(\frac{\log(n(\lambda))}{\alpha(\lambda)})$.
7. Let $\ell'(\lambda) = 8\ell(\lambda) = \log(n(\lambda))$. The construction uses the Nisan–Wigderson generator instantiated with a $(d(\lambda), \ell'(\lambda), \ell(\lambda))$-design $\mathcal{I}$ of size $m(\lambda) = 2^{\ell(\lambda)}$ where $d(\lambda)$ is $\ell'(\lambda)$ times the least power of 2 greater than $\ell'(\lambda)$. Note that $d(\lambda) \approx \ell'(\lambda)^2$.
8. The key space of the PRF is $(\mathcal{X}_\lambda \times \{0,1\}^{d(\lambda)})^{\gamma(\lambda)}$. Note that each key has a unique encoding (under a reasonable encoding scheme) in roughly $(\lambda + d(\lambda)) \cdot \gamma(\lambda)$ bits, and thus the key space can equivalently be thought of as $\{0,1\}^{(\lambda+d(\lambda))\cdot\gamma(\lambda)}$.

For simplicity of exposition we assume that all the parameters are integers, and that $n(\lambda)$ is a power of 2. The analysis works even when this is not the case if one takes floors and ceilings appropriately. The PRF is constructed as follows.

1. For every $(\Pi, q) \in \mathcal{X}_\lambda$, define $g_{\Pi,q} : \{0,1\}^{d(\lambda)} \to \{0,1\}^{m(\lambda)}$ as
$$g_{\Pi,q}(y) = \mathsf{NW}^{\Pi^t_q(\cdot)}_{\mathcal{I}}(y).$$
2. A key for the PRF is a tuple $k = ((\Pi_1, q_1, y_1), \ldots, (\Pi_{\gamma(\lambda)}, q_{\gamma(\lambda)}, y_{\gamma(\lambda)}))$ where for every j, $(\Pi_j, q_j) \in \mathcal{X}_\lambda$ and $y_j \in \{0,1\}^{d(\lambda)}$.
3. The PRF is the following function $F : (\mathcal{X}_\lambda \times \{0,1\}^{d(\lambda)})^{\gamma(\lambda)} \times \{0,1\}^{\ell(\lambda)} \to \{0,1\}$.
$$F(((\Pi_1, q_1, y_1), \ldots, (\Pi_\gamma, q_\gamma, y_\gamma)), i) = \bigoplus_{j=1}^{\gamma(\lambda)} g_{\Pi_j,q_j}(y_j)[i].$$

Lemma 7.6. *If $\oplus \in \mathsf{AC}^0[h]$, $s(n) \geq \log^\delta n$ for some $\delta > 0$, and $\alpha(\lambda) \geq 1/\mathsf{poly}(\lambda)$, then for every key $z = ((\Pi_1, q_1, y_1), \ldots, (\Pi_\gamma, q_\gamma, y_\gamma))$, the function $F(z, \cdot)$ is computable by a constant-depth $\mathsf{AC}^0[h]$ circuit of size $\mathsf{poly}(\lambda)$. The depth depends on the constant time bound t.*

Proof. Since $s(n) \geq \log^\delta n$, $s^{-1}(\lambda) \leq 2^{\log^{1/\delta} \lambda}$. So $\gamma(\lambda) = O(\frac{s^{-1}(\lambda)}{\alpha(\lambda)}) = O(\mathsf{poly}(\lambda) \cdot \log^{1/\delta} \lambda) = \mathsf{poly}(\lambda)$. Combined with the fact that $\oplus \in \mathsf{AC}^0[h]$, this means one can implement $\bigoplus_{j=1}^{\gamma(\lambda)}$ with $\mathsf{poly}(\lambda)$ wires and constant depth. So it suffices to show that for every $(\Pi, q) \in \mathcal{X}_\lambda$ and $y \in \{0,1\}^{d(\lambda)}$, the function $i \mapsto g_{\Pi,q}(y)[i]$ can be implemented by a $\mathsf{poly}(\lambda)$-size $\mathsf{AC}^0[h]$ circuit.

By Lemma 2.10 there is a $\mathsf{poly}(\ell'(\lambda))$-size, constant-depth $\mathsf{AC}^0[\oplus]$ circuit computing the function $\mathsf{MX}_{\mathrm{NW}} : \{0,1\}^{\ell(\lambda)} \times \{0,1\}^{d(\lambda)} \to \{0,1\}^{\ell'(\lambda)}$ defined as $\mathsf{MX}_{\mathrm{NW}}(i, y) = y_{I_i}$. Since $\oplus \in \mathsf{AC}^0[h]$, this circuit can be transformed into a $\mathsf{poly}(\ell'(\lambda))$-size, constant-depth $\mathsf{AC}^0[h]$ circuit computing $\mathsf{MX}_{\mathrm{NW}}$. By Theorem 5.4 there is a $O(t)$-depth $\mathsf{AC}^0[h]$ circuit of size $\mathsf{poly}(\lambda)$ computing $\Pi_q^t(\cdot)$. It is easy to see from the definition of the Nisan–Wigderson generator that $g_{\Pi,q}(y)[i] = \Pi_q^t(\mathsf{MX}_{\mathrm{NW}}(i, y))$. Thus the function $i \mapsto g_{\Pi,q}(y)[i]$ can be computed by an $\mathsf{AC}^0[h]$ circuit of size $\mathsf{poly}(\ell'(\lambda)) + \mathsf{poly}(\lambda)$ and constant depth. Since $s^{-1}(\lambda) \leq 2^{\log^{1/\delta} \lambda}$ and $\ell'(\lambda) = \log(s^{-1}(\lambda))$, the circuit size is at most $\mathsf{poly}(\lambda)$. □

Remark 7.7. *It is easy to see that if there exists a sufficiently efficient universal PRAM, then our construction is in fact* jointly *computable by constant-depth, $\mathsf{poly}(\lambda)$-size* uniform *$\mathsf{AC}^0[h]$ circuits. We believe that such a universal PRAM does exist, though we leave constructing it for future work.*

As mentioned before, our construction (and its analysis) closely follows a construction of Liu and Pass [LP24a]. Their construction is essentially the same as the one here, except for the $g_{\Pi,q}$s: Liu and Pass need to define their gs based on Turing machines, whereas we define ours based on PRAMs. In their analysis, they show that if the gs form a weak family of PRGs, then F is a PRF. The proof of that statement does not depend on the definition of the gs, and so we use it here without proof.

Lemma 7.8 ([LP24a])**.** *If $\alpha(\lambda) = 1/O(\lambda^\beta)$ for some $\beta > 0$ and $\{g_{\Pi,q}\}_{(\Pi,q)\in\mathcal{X}_\lambda}$ is an $\alpha(\lambda)$-weak family of $(2^{2\ell(\lambda)}, 2^{-\ell(\lambda)})$-PRGs (resp. secure against nonuniform adversaries), then there exists a constant $\delta > 0$ such that F is a $(s^{-1}(\lambda)^\delta, \frac{1}{s^{-1}(\lambda)^\delta})$-PRF (resp. secure against nonuniform adversaries).*

We also need the following lemma showing that random strings cannot be approximated by randomized Turing machines with low description length (in a different regime of parameters than our Lemma 6.4).

Lemma 7.9 ([LP24a])**.** *For all constants $\varepsilon, \delta > 0$ with $\delta < 1 - 2\varepsilon$, for all sufficiently large n, the probability over a random string $x \sim \{0,1\}^n$ that there exists a probabilistic Turing machine M of description length at most n^δ (that terminates on every random tape) such that*

$$\Pr_{i\sim[n]}[M(i) = x_i] \geq \frac{1}{2} + \frac{1}{n^\varepsilon}$$

is at most

$$n^\varepsilon \cdot 2^{-\frac{n^{1-2\varepsilon}}{2} + n^\delta + 2}.$$

Lemma 7.10. *Let $\alpha(\lambda) = \frac{1}{c'\lambda^\beta}$ for some $c', \beta > 0$. Assume $\mathsf{KP\text{-}vs\text{-}Random}_h^t[s]$ is (resp. nonuniformly) (n^3, α)-hard on average* with respect to the s-universal distribution. Then $\{g_{\Pi,q}\}_{(\Pi,q)\in\mathcal{X}_\lambda}$ is a $(\alpha(\lambda)/4)$-weak family of $(2^{2\ell(\lambda)}, 2^{-\ell(\lambda)})$-PRGs (resp. secure against nonuniform adversaries).*

Proof. We will prove the lemma in the uniform case, making note of how one can modify it to obtain the nonuniform case. For convenience, let $T(\lambda) = 2^{2\ell(\lambda)}$ and $\varepsilon(\lambda) = 2^{-\ell(\lambda)}$. Suppose $\{g_{\Pi,q}\}_{(\Pi,q)\in\mathcal{X}_\lambda}$ is not an $\frac{\alpha(\lambda)}{4}$-weak family of (T,ε)-PRGs. Then there is a distinguisher D such that for infinitely many λ, for a $1-\frac{\alpha(\lambda)}{4}$ fraction of keys $(\Pi,q)\in\mathcal{X}_\lambda$,

$$\left|\Pr_{y\sim\{0,1\}^{d(\lambda)}}[D(1^\lambda, g_{\Pi,q}(y))=1] - \Pr_{r\sim\{0,1\}^{m(\lambda)}}[D(1^\lambda, r)=1]\right| \geq \varepsilon(\lambda). \tag{1}$$

Let A be the following algorithm that we will show solves $\mathsf{KP\text{-}vs\text{-}Random}^t_h[s]$ on average* over the s-universal distribution on the infinitely many input lengths $n(\lambda)$.

1. Input: $x\in\{0,1\}^{n(\lambda)}$.
2. View x as the truth table of a function $f_x:\{0,1\}^{\log(n(\lambda))}\to\{0,1\}$.[10]
3. Estimate the expectations of the following random variables such that the estimate is at most $\varepsilon(\lambda)/8$ away from the true expectation with probability at least $1-\alpha(\lambda)/4$. This can be done by drawing $O(\frac{1}{\varepsilon(\lambda)^2}\cdot\log\frac{1}{\alpha(\lambda)})$ random samples and computing the mean over the sample. The error and correctness probability bounds hold due to standard Hoeffding bounds.
   (a) ρ_x, the random variable $D(1^\lambda, \mathsf{NW}^{f_x}_{\mathcal{I}}(\mathcal{U}_{d(\lambda)}))$.
   (b) θ, the random variable $D(1^\lambda, \mathcal{U}_{m(\lambda)})$.
4. Let ρ^*_x and θ^* be the estimates obtained in the previous step. Accept if $|\rho^*_x-\theta^*|\geq\varepsilon(\lambda)/2$.

Let us analyze the running time of A. For an arbitrary y, one can compute $\mathsf{NW}^{f_x}_{\mathcal{I}}(y)$ by (a) computing the design $\mathcal{I}=\{I_1,\ldots,I_{m(\lambda)}\}$, and (b) for each $i\in[m(\lambda)]$, computing $f_x(y_{I_i})$. By Lemma 2.9, one can compute a single I_i in time $\widetilde{O}(\ell'(\lambda)^2)$. Given I_i, computing $f_x(y_{I_i})$ takes time at most $O(|x|) = O(n(\lambda))$. Since there are $2^{\ell(\lambda)}$ designs, this can all be done in time $2^{\ell(\lambda)}\cdot(O(n(\lambda))+\widetilde{O}(\ell(\lambda)^2)) \leq O(n(\lambda)^2)$. This means that drawing a single sample from ρ_x, which requires drawing a random string y, evaluating $\mathsf{NW}^{f_x}_{\mathcal{I}}(y)$, and then running D on it, can be implemented in time $T(\lambda)+O(n(\lambda)^2)$. Hence the estimate ρ^*_x, which requires drawing a number of such samples, can be computed in time $(T(\lambda)+O(n(\lambda)^2))\cdot O(\frac{1}{\varepsilon(\lambda)^2}\cdot\log\frac{1}{\alpha(\lambda)}) \leq O(n(\lambda)^3)$. The time required to draw samples from θ is strictly smaller, and so the total running time is also $O(n(\lambda)^3)$. We now show that A is in fact a good algorithm on input lengths $n(\lambda)$.

Fix any x such that $\mathsf{KP}^t_h(x)\leq s(n)$. By definition, there is some $(\Pi,q)\in\mathcal{X}_{s(n)}$ such that $f_x(\cdot)\equiv\Pi^t_q(\cdot)$. This means

$$\begin{aligned}
\mathbb{E}[\rho_x] &= \Pr_{y\sim\{0,1\}^{d(\lambda)}}[D(1^\lambda, \mathsf{NW}^{f_x}_{\mathcal{I}}(y))=1] \\
&= \Pr_{y\sim\{0,1\}^{d(\lambda)}}[D(1^\lambda, \mathsf{NW}^{\Pi^t_q(\cdot)}_{\mathcal{I}}(y))=1] \\
&= \Pr_{y\sim\{0,1\}^{d(\lambda)}}[D(1^\lambda, g_{\Pi,q}(y))=1].
\end{aligned}$$

On the other hand, $\mathbb{E}[\theta] = \Pr_{r\sim\{0,1\}^{m(\lambda)}}[D(1^\lambda,r)=1]$. It follows by definition of the s-universal distribution that the probability, over $x\sim D^{s\text{-univ}}_n$ that $|\mathbb{E}[\rho_x]-\mathbb{E}[\theta]|\geq\varepsilon(\lambda)$ is equal to the probability,

[10] As mentioned before, we assume that n is a power of 2. If it is not (and ℓ and ℓ' are redefined with floors), one can simply carry out the same analysis with x redefined as the first $2^{\ell'(\lambda)}$ bits of x.

over a random key $(\Pi, q) \sim \mathcal{X}_\lambda$ that Eq. 1 holds. By assumption, that probability is at least $1-\alpha(\lambda)/4$. As noted in the description of the algorithm, by a standard Hoeffding bound, except with probability $\alpha(\lambda)/4$, the estimates ρ_x^* and θ^* are additively within $\varepsilon(\lambda)/8$ of the true expectations of ρ_x and θ respectively. It follows by a union bound that the probability that $|\rho_x^* - \theta^*| < \varepsilon(\lambda)/2$ is at most $\alpha(\lambda)/2$. For sufficiently large n,

$$\frac{\alpha(\lambda)}{2} = \frac{1}{2c's(n)^\beta} \leq \frac{1}{2c'(n^*-1)^\beta} \leq \frac{1}{c'(n^*)^\beta} = \alpha(n^*),$$

and so for sufficiently large n the algorithm A accepts $x \sim D_n^{s\text{-univ}}$ with probability at least $1-\alpha(n^*)$.

Let us now consider the No case, where x is drawn uniformly at random. Say an input $x \in \{0,1\}^n$ is *bad* if $|\mathbb{E}[\rho_x] - \mathbb{E}[\theta]| \geq \varepsilon(\lambda)/4$. By the Hoeffding bound mentioned in the algorithm description, if an input is not bad, then the estimates ρ^* and θ^* will be within $\varepsilon(\lambda)/2$ of each other with probability at least $1 - \alpha(\lambda)/4 \geq 1 - \alpha(n^*)/2$. So it suffices to show that the probability that a random x is bad is at most $\alpha(n^*)/2$. We show that this is true due to the reconstruction property of the Nisan–Wigderson generator.

Fix any bad string x. By Theorem 2.11, the algorithm $\mathsf{NWRecon}^{D(1^\lambda,\cdot)}$ will output with high probability an oracle Turing machine M such that $\Pr_{i\in[n]}[M^{D(1^\lambda,\cdot)}(\mathcal{I}, i) = x_i] \geq \frac{1}{2} + \frac{\varepsilon(\lambda)}{8m(\lambda)}$. Furthermore, the theorem guarantees that M has description length at most $m2^\ell + m + d + O(\log d)$. One can transform M into an oracle-free machine M' such that $M'(i) = M^{D(1^\lambda,\cdot)}(\mathcal{I}, i)$ by hardwiring D and storing the parameters for the combinatorial designs. Since D is uniform it can be hardwired with $O(1)$ bits (if D were nonuniform this would be $T(\lambda)$ bits). Storing the parameters takes $O(\log d + \log \lambda)$ bits. Thus, by our setting of parameters, M' has description length at most

$$m2^\ell + m + d + O(\log d + \log \lambda) = O(2^{\ell(\lambda)}).$$

(In the nonuniform case this would be $O(2^{2\ell(\lambda)})$.) By Lemma 7.9, a random x cannot be approximated by such a machine with high probability. Quantitatively, for sufficiently large λ, the probability that $x \sim \{0,1\}^n$ is bad is at most

$$\begin{aligned}
\frac{8m}{\varepsilon} \exp_2\left(-\frac{n}{2} \cdot \frac{\varepsilon^2}{64m^2} + O(2^\ell) + 2\right) &= 8 \cdot 2^{2\ell} \cdot \exp_2\left(-\frac{2^{4\ell}}{128} + O(2^\ell) + 2\right) \\
&= 2^{-\Omega(2^{4\ell})} \\
&\ll \frac{1}{2c'2^{8\beta\ell}} \\
&= \frac{1}{2c'n^\beta} \\
&\leq \frac{1}{2c'(n^*)^\beta} = \frac{\alpha(n^*)}{2}.
\end{aligned}$$

(Note that the calculation goes through even in the nonuniform case, where the $O(2^\ell)$ in the exponent is replaced by $O(2^{2\ell})$.) □

Theorem 7.11. *Let $t \geq 5$ be a constant, h be a bit-paddable function such that $\oplus \in \mathsf{AC}^0[h]$, and $s(n) < n$ be arbitrary. If $\mathsf{KP\text{-}vs\text{-}Random}_h^t[s]$ is (resp. nonuniformly) (n^3, α)-hard on average* with respect to the s-universal distribution for some $\alpha(n^*) = 1/O((n^*)^\beta)$, then there is a constant $\delta > 0$ such that there exists a $(s^{-1}(\lambda)^\delta, \frac{1}{s^{-1}(\lambda)^\delta})$-PRF (resp. secure against nonuniform adversaries) computable by constant-depth, $\mathsf{poly}(\lambda)$-size $\mathsf{AC}^0[h]$ circuits.*

Proof. By Lemma 7.10 the hardness assumption implies that the gs form an $(\alpha(\lambda)/4)$-weak family of $(2^{2\ell(\lambda)}, 2^{-\ell(\lambda)})$-PRGs. Lemma 7.8 then implies that there exists a constant $\delta > 0$ such that F is a $((s^{-1}(\lambda))^{\delta}, \frac{1}{(s^{-1}(\lambda))^{\delta}})$-PRF. The circuit depth and size bound holds due to Lemma 7.6. □

Using the fact that one can transfer hardness with respect to the uniform distribution to hardness with respect to the universal distribution, we get the following theorem.

Corollary 7.12. *Let $t \geq 5$ be a constant, h be a bit-paddable function such that $\oplus \in \mathsf{AC}^0[h]$, and $s(n) < n$ be arbitrary. If $\mathsf{KP\text{-}vs\text{-}Random}^t_h[s]$ is (resp. nonuniformly) (n^3, α)-hard on average* with respect to the uniform distribution for some $\alpha(n^*) = 1/O((n^*)^{\beta})$, then there is a constant $\delta > 0$ such that there exists a $((s^{-1}(\lambda))^{\delta}, \frac{1}{(s^{-1}(\lambda))^{\delta}})$-PRF (resp. secure against nonuniform adversaries) computable by constant-depth, $\mathsf{poly}(\lambda)$-size $\mathsf{AC}^0[h]$ circuits.*

Proof. Let $\alpha(n^*) = 1/O((n^*)^{\beta})$ for some $\beta > 0$. By Lemma 7.5, if $\mathsf{KP\text{-}vs\text{-}Random}^t_h[s]$ is (n^3, α)-hard on average* with respect to the uniform distribution, then it is (n^3, α')-hard on average* with respect to the s-universal distribution, where $\alpha'(n^*) = 1/O((n^*)^{\beta+c})$. The premises of Theorem 7.11 are thus met, and the conclusion follows. □

We obtain the following corollary by plugging in a specific value for s.

Corollary 7.13. *Let $t \geq 5$ be a constant, h be a bit-paddable function such that $\oplus \in \mathsf{AC}^0[h]$. Let $s(n) = 2^{O(\log^{1/d} n)}$ and $\alpha(n^*) = \frac{1}{O((n^*)^{\beta})}$. If $\mathsf{KP\text{-}vs\text{-}Random}^t_h[s]$ is (n^3, α)-hard on average* with respect to the uniform distribution then there is a strong PRF computable by constant-depth, $\mathsf{poly}(\lambda)$-size $\mathsf{AC}^0[h]$ circuits with key length $\widetilde{O}(\lambda^{1+\beta+c})$ and input domain $\{0,1\}^{\Omega(\log^d \lambda)}$.*

Proof. Since $s(n) = 2^{O(\log^{1/d} n)}$, $s^{-1}(\lambda) = 2^{\Omega(\log^d \lambda)}$. Corollary 7.12 then shows that under the hardness assumption, there is a constant $\delta > 0$ such that F constructed above is a $(2^{\Omega(\log^d \lambda)}, 2^{-\Omega(\log^d \lambda)})$-PRF. Moreover, the construction of F implicit in the proof uses the "α" obtained by transferring hardness over the uniform distribution to hardness over the universal distribution, namely $\alpha'(\lambda) = 1/O(\lambda^{\beta+c})$. Thus, inspecting the construction, we see that the key length is $(\lambda + d(\lambda)) \cdot \gamma(\lambda) \approx (\lambda + \ell(\lambda)^2) \cdot \log(n(\lambda)) \cdot \lambda^{\beta+c} = (\lambda + \log^2(n(\lambda))) \cdot \log(n(\lambda)) \cdot \lambda^{\beta+c} = \widetilde{O}(\lambda^{1+\beta+c})$. The input domain is $\{0,1\}^{\ell(\lambda)}$. We have $\ell(\lambda) = \frac{1}{8}\log(n(\lambda)) = \Omega(\log^d \lambda)$, and hence the input domain is as claimed. □

7.3 Hardness of KP-vs-Random from Constant-Depth PRFs

A natural question to ask is whether hardness of $\mathsf{KP\text{-}vs\text{-}Random}_h$ is *necessary* for PRFs in $\mathsf{AC}^0[h]$. We make progress on this question by showing that for all "reasonable" (symmetric, logspace-computable) h, if there is a PRF satisfying stronger computability requirements than those considered above, then there exists $\alpha(n^*) = 1/\mathsf{poly}(n^*)$ such that for every $T(n) = \mathsf{poly}(n)$, $\mathsf{KP\text{-}vs\text{-}Random}^{O(\log n)}_h[n - O(\log n)]$ is (T, α)-hard on average*.

We use the following lemma which follows from Theorem 25, Lemma 37, and the proof of Theorem 36 in [RS21].

Lemma 7.14 (implicit in [RS21])**.** *Suppose that there is a one-way function in uniform $\oplus\mathsf{L}$. Then there is a constant $c_{\mathrm{RS}} > 0$ such that, for every constant $\gamma > 0$, there is a family of functions $\{G_r : \{0,1\}^r \to \{0,1\}^{r+\lceil \gamma \log r \rceil}\}_{r \in \mathbb{N}}$ and subsets $\{E_r \subseteq \{0,1\}^r\}_{r \in \mathbb{N}}$ with the following properties.*

1. *For some constant $b_\gamma > 0$, every polynomial-time distinguisher has advantage less than r^{-b_γ} in distinguishing $G_r(\mathcal{U}(E_r))$ from $\mathcal{U}_{r+\lceil \gamma \log r \rceil}$.*

2. *For some constant* $d_\gamma > 0$,
$$H(G_r(\mathcal{U}(E_r))) \geq r - d_\gamma \log r,$$
where H *denotes Shannon entropy.*

3. *Given* r*, an output coordinate* $i \in [r + \lceil \gamma \log r \rceil]$*, and random access to* $z \in \{0,1\}^r$*, the bit* $G_r(z)_i$ *is computable in time* $c_{\mathrm{RS}} \log r$*.*

Remark 7.15. *The only bottleneck in getting hardness of constant-time* KP_h *as opposed to super-constant-time hardness is item 3 in Lemma 7.14. If* G_r *could be implemented in a way such that* $i \mapsto G_r(z)_i$ *is computable by a constant-time PRAM (or, equivalently, by constant-depth circuits), then we would be able to show that if there are PRFs jointly computable in uniform* $\mathsf{AC}^0[h]$ *then* $\mathsf{KP}\text{-vs-Random}^t_h[n - O(\log n)]$ *is hard on average* for some constant* t*. To modify Ren and Santhanam's construction of* G_r *to be constant-depth one would have to implement the underlying* randomized encodings *[AIK06] in constant-depth. Such an implementation is plausible but not known.*

Lemma 7.16. *Let* $G_r : \{0,1\}^r \to \{0,1\}^{r+O(\log r)}$ *be such that given* r *and random access to* $z \in \{0,1\}^r$*, the bit* $G_r(z)_i$ *is computable in time* $c \log r$*. There are constants* $c_0, a > 0$*, depending only on* c*, such that for* $\tau(n) := c_0 \log n$*, every* $z \in \{0,1\}^r$ *satisfies*
$$\mathsf{KP}^\tau_h(G_r(z)) \leq r + a \log r.$$

Proof. By assumption there is a Turing machine M such that $M^z(r,i) = G_r(z)[i]$ in time $c \log r$. A (single-processor) PRAM that stores r and z as finite oracles and simply simulates the execution of $M^z(r,\cdot)$ computes the function $i \mapsto G_r(z)[i]$. It is easy to see that our RAM instruction set can simulate deterministic oracle TMs with constant-factor overhead in running time, so the PRAM runs in time $c_0 \log r$ for some sufficiently large constant c_0 that depends on c. A description of the PRAM consists of (a) $O(1)$ bits to describe M, (b) r bits for the finite oracle containing z, and (c) $\log r$ bits for the finite oracle containing r. Thus the total description length is $r + O(\log r)$ as claimed. □

Lemma 7.17. *Let* $s(n) < n$ *be a threshold and let* τ *be the runtime bound from Lemma 7.16. Let* Y_n *be a distribution on* $\{0,1\}^n$ *such that for all sufficiently large* n*,*

1. $\mathrm{supp}(Y_n) \subseteq \mathsf{MKPP}^\tau_h[s] \cap \{0,1\}^n$*,*
2. *there exists* $q(n) \leq n$ *such that* Y_n *can be sampled by applying some deterministic function to a uniformly random string from a subset of* $\{0,1\}^{q(n)}$*,*
3. $H(Y_n) \geq q(n) - d \log(q(n))$ *for some constant* $d > 0$*, and*
4. $|n^* - q(n)| \leq a \log(q(n))$ *for some constant* $a > 0$*.*

Then there is a constant $C > 0$ *such that the following holds. If an algorithm* A *satisfies*
$$\Pr_{\substack{x \sim \mathsf{MKPP}^\tau_h[s] \cap \{0,1\}^n, \\ A}}[A(x) = 1] \geq 1 - (n^*)^{-C} \text{ and} \tag{2}$$
$$\Pr_{x \sim \{0,1\}^n, A}[A(x) = 1] \leq (n^*)^{-C}, \tag{3}$$
then A *distinguishes* Y_n *from* $\mathcal{U}_n$ *with advantage* $\Omega(1/\log(q(n)))$*.*

Proof. Define $q' = H(Y_n)$ and $p_y = \Pr[Y_n = y]$. Let

$$\mathsf{Good} = \{y \in \{0,1\}^n : 0 < p_y \le 2^{1-q'}\}.$$

Since Y_n can be sampled by postprocessing a uniform distribution on a subset of $\{0,1\}^{q(n)}$, for all y, if $p_y \neq 0$ then $p_y \ge 2^{-q(n)}$, and so for all y in the support of Y_n, $\log 1/p_y \le q(n)$. Let $\eta = \Pr_{y\sim Y_n}[y \in \mathsf{Good}]$. For $y \notin \mathsf{Good}$ we have $\log(1/p_y) < q' - 1$. By definition of Shannon entropy,

$$q' = \mathop{\mathbb{E}}_{y\sim Y_n}\left[\log \frac{1}{p_y}\right] \le \eta \cdot q(n) + (1-\eta)\cdot(q'-1).$$

Rearranging,

$$\eta \ge \frac{1}{q(n) - q' + 1} \ge \frac{1}{d\log(q(n)) + 1}.$$

By definition of normalized input length, there are 2^{n^*} strings in $\mathsf{MKPP}^\tau_h[s] \cap \{0,1\}^n$, so Eq. 2 implies that $\sum_{y\in\mathsf{MKPP}^\tau_h[s]\cap\{0,1\}^n}(1 - \Pr_A[A(y)=1]) \le 2^{n^*}\cdot(n^*)^{-C}$. Thus

$$\begin{aligned}
\mathop{\mathbb{E}}_{y\sim Y_n}\left[\left(1 - \Pr_A[A(y)=1]\right)\cdot \mathbb{1}[y\in\mathsf{Good}]\right] &= \sum_{y\in\mathsf{Good}} p_y \cdot \left(1 - \Pr_A[A(y)=1]\right)\\
&\le 2^{1-q'}\sum_{y\in\mathsf{Good}}\left(1 - \Pr_A[A(y)=1]\right)\\
&\le 2^{1-q'}\sum_{y\in\mathsf{MKPP}^\tau_h[s]\cap\{0,1\}^n}\left(1 - \Pr_A[A(y)=1]\right)\\
&\le 2^{1-q'}\cdot 2^{n^*}\cdot(n^*)^{-C}\\
&= 2^{1+n^*-q'}\cdot(n^*)^{-C}.
\end{aligned}$$

By items 3 and 4 of the lemma statement, $n^* - q' \le (a+d)\log(q(n))$. Also, item 4 implies $n^* = \Theta(q(n))$. Thus, picking C to be large enough, we get that for sufficiently large n,

$$\mathop{\mathbb{E}}_{y\sim Y_n}\left[\left(1 - \Pr_A[A(y)=1]\right)\cdot \mathbb{1}[y\in\mathsf{Good}]\right] \le q(n)^{-2}.$$

Hence,

$$\begin{aligned}
\Pr_{y\sim Y_n, A}[A(y)=1] &= \mathop{\mathbb{E}}_{y\sim Y_n}\Pr_A[A(y)=1]\\
&\ge \mathop{\mathbb{E}}_{y\sim Y_n}\left(\Pr_A[A(y)=1]\cdot\mathbb{1}[y\in\mathsf{Good}]\right)\\
&= \mathop{\mathbb{E}}_{y\sim Y_n}[\mathbb{1}[y\in\mathsf{Good}]] - \mathop{\mathbb{E}}_{y\sim Y_n}\left[\left(1-\Pr_A[A(y)=1]\right)\cdot\mathbb{1}[y\in\mathsf{Good}]\right]\\
&\ge \eta - q(n)^{-2}\\
&\ge \frac{1}{d\log(q(n))+1} - q(n)^{-2}.
\end{aligned}$$

By Eq. 3 the acceptance probability of A on a uniformly random string is at most $(n^*)^{-C} = 1/\mathsf{poly}(q(n))$, and so the distinguishing advantage is $\frac{1}{d\log(q(n))+1} - \frac{1}{\mathsf{poly}(q(n))} = \Omega(1/\log(q(n)))$. □

Theorem 7.18. *Let h be a symmetric, logspace-computable function. Suppose there is a PRF $F_\lambda : \{0,1\}^{k(\lambda)} \times \{0,1\}^{\ell(\lambda)} \to \{0,1\}$ such that*

1. *$2^{\ell(\lambda)} \geq 2k(\lambda)$ for sufficiently large λ and $k(\lambda) = \mathsf{poly}(\lambda)$,*
2. *F is jointly computable by constant-depth, $\mathsf{poly}(\lambda)$-size uniform $\mathsf{AC}^0[h]$ circuits, and*
3. *F is a (T, ε)-PRF with $T(\lambda) = k(\lambda)^{\omega(1)}$ and $\varepsilon(\lambda) = k(\lambda)^{-\omega(1)}$.*

Then there is a constant $c_0 > 0$ such that for every $\Delta > 2$, $\tau(n) = c_0 \log n$, and $s(n) = n - \Delta \log n$, there is a constant C such that for all constants $\kappa > 0$, $\mathsf{KP\text{-}vs\text{-}Random}^\tau_h[s]$ is $(n^\kappa, (n^*)^{-C})$-hard on average.*

Proof. We begin by proving the following claim.

Claim 7.19. *Under the premises of the theorem, there exists a one-way function in $\oplus\mathsf{L}$.*

Proof. Let $x_1, x_2, \ldots, x_{2k(\lambda)} \in \{0,1\}^{\ell(\lambda)}$ be an arbitrary canonical sequence of strings (for example, the lexicographically first $2k(\lambda)$ strings). Define

$$H_\lambda(z) = F_\lambda(z, x_1) \| F_\lambda(z, x_2) \| \cdots \| F_\lambda(z, x_{2k(\lambda)}).$$

It follows from the facts that h is logspace computable and F is computable in uniform $\mathsf{AC}^0[h]$ that the function H is computable in $\oplus\mathsf{L}$. Suppose A is a polynomial-time probabilistic algorithm such that $\Pr_{z \sim \{0,1\}^{k(\lambda)}}[H_\lambda(A(1^\lambda, H_\lambda(z))) = H_\lambda(z)] \geq 1/\mathsf{poly}(\lambda)$. Let A' be the algorithm that, given oracle access to a function f and input 1^λ, does the following.

1. Query $f(x_1), \ldots, f(x_{2k(\lambda)})$ and set $y = f(x_1) \| \cdots \| f(x_{2k(\lambda)})$.
2. Compute $z' \leftarrow A(1^\lambda, y)$.
3. Accept if $H_\lambda(z') = y$, reject otherwise.

Since A runs in polynomial time, so does A'. If $f(\cdot) = F_\lambda(z, \cdot)$, then $y = H_\lambda(z)$. Thus if $f(\cdot) = F_\lambda(z, \cdot)$ for a random $z \sim \{0,1\}^{k(\lambda)}$, then A' runs A on the image of a random input to H_λ, and so A' accepts with probability $1/\mathsf{poly}(\lambda)$. On the other hand, if f is a random function then y is a uniformly random string in $\{0,1\}^{2k(\lambda)}$. Since H_λ has a domain of size $2^{k(\lambda)}$ this means that the probability that y has an inverse under H_λ is at most $2^{-k(\lambda)}$. It follows that the probability that A' accepts when the oracle is a random function is at most $2^{-k(\lambda)}$. Hence A' has advantage $1/\mathsf{poly}(\lambda)$ in distinguishing the PRF from random, contradicting the PRF's security. □

Lemma 7.14 thus gives a constant c_{RS}. Let c_0 and a be the constants given by Lemma 7.16 when $c = c_{\mathrm{RS}}$. Set $\tau(n) = c_0 \log n$. Let $\Delta > 2$ be an arbitrary constant and set $\gamma = \Delta + a + 2$. By Lemma 7.14 there is a family of functions $\{G_r : \{0,1\}^r \to \{0,1\}^{r + \lceil \gamma \log r \rceil}\}_{r \in \mathbb{N}}$ and a sequence of sets $\{E_r \subseteq \{0,1\}^r\}_{r \in \mathbb{N}}$ such that for some constants $b_\gamma, d_\gamma > 0$, (a) $H(G_r(\mathcal{U}[E_r])) \geq r - d_\gamma \log r$ and (b) every probabilistic polynomial-time algorithm has advantage at most r^{-b_γ} in distinguishing $G_r(\mathcal{U}[E_r])$ from $\mathcal{U}_{r + \lceil \gamma \log r \rceil}$. Finally, by Lemma 7.16, for every $z \in \{0,1\}^r$, $\mathsf{KP}^\tau_h(G_r(z)) \leq r + a \log r$.

Note that the output lengths $r + \lceil \gamma \log r \rceil$ might skip some lengths. Since average-case* hardness is defined almost everywhere, we need to modify the family G such that the output lengths cover all (but finitely many) lengths. To that end, for $n \in \mathbb{N}$, let $r(n)$ be the largest number such that $r + \lceil \gamma \log r \rceil \leq n$, $g(n) = n - r(n) - \lceil \gamma \log(r(n)) \rceil$, and $q(n) = r(n) + g(n)$. Note that for large n, $g(n) \in \{0,1\}$. Define $\widehat{G}_n : \{0,1\}^{q(n)} \to \{0,1\}^n$ by

$$\widehat{G}_n(z, u) = G_{r(n)}(z) \| u \text{ for } z \in \{0,1\}^{r(n)},\ u \in \{0,1\}^{g(n)}.$$

Also define $\widehat{E}_n = E_{r(n)} \times \{0,1\}^{g(n)}$. It is easy to see that the following hold due to the corresponding facts for G and the fact that $\widehat{G}$ is obtained from G by adding (at most) a uniformly random bit.

1. For all probabilistic polynomial-time algorithms D and sufficiently large n,
$$\left| \Pr_{z' \sim \widehat{E}_n}[D(\widehat{G}_n(z')) = 1] - \Pr_{y \sim \{0,1\}^n}[D(y) = 1] \right| < r(n)^{-b_\gamma}.$$
2. $H(\widehat{G}_n(\mathcal{U}[\widehat{E}_n])) \geq q(n) - O(\log(q(n)))$.
3. For all $z' \in \{0,1\}^{q(n)}$, $\mathsf{KP}^\tau_h(\widehat{G}_n(z')) \leq q(n) + a\log(q(n)) + O(1)$.

Let $s(n) = n - \Delta \log n$. We apply Lemma 7.17 with $Y_n = \widehat{G}_n(\mathcal{U}[\widehat{E}_n])$ and the parameter q above. We explain here why the premises of the lemma are satisfied. Premise 1 holds due to property 3. Premise 2 holds since Y_n is obtained by applying the deterministic function $\widehat{G}_n$ to the uniform distribution over $\widehat{E}_n \subseteq \{0,1\}^{q(n)}$. Premise 3 holds because of property 2. From the definitions of r, g, and q, we get $n = q(n) + \lceil \gamma \log(r(n)) \rceil$ and $\log n = O(\log(q(n)))$. We thus have $\log(s(n)) \leq \log n = O(\log(q(n)))$. The upper bound in Proposition 7.2 gives

$$\begin{aligned} n^* \leq s(n) + 1 &= q(n) + \lceil \gamma \log(r(n)) \rceil - \Delta \log n \\ &\leq q(n) + \lceil \gamma \log(q(n)) \rceil = q(n) + O(\log(q(n))), \end{aligned}$$

whilst the lower bound gives

$$\begin{aligned} n^* &\geq s(n) - O(\log(s(n))) \\ &= q(n) + \lceil \gamma \log(r(n)) \rceil - \Delta \log n - O(\log(q(n))) \\ &= q(n) - O(\log(q(n))). \end{aligned}$$

Hence $|n^* - q(n)| = O(\log(q(n)))$, which means premise 4 holds. Let C be the constant one gets by applying Lemma 7.17.

Assume, seeking a contradiction, that there is some $\kappa > 0$ such that $\mathsf{KP\text{-}vs\text{-}Random}^\tau_h[s]$ is not $(n^\kappa, (n^*)^{-C})$-hard on average*. Then, by definition of average-case* hardness, there is a probabilistic n^κ-time algorithm such that for infinitely many input lengths n,

$$\begin{aligned} \Pr_{\substack{x \sim \mathsf{MKPP}^\tau_h[s] \cap \{0,1\}^n, \\ A}}[A(x) = 1] &\geq 1 - (n^*)^{-C} \text{ and} \\ \Pr_{x \sim \{0,1\}^n, A}[A(x) = 1] &\leq (n^*)^{-C}. \end{aligned}$$

By the conclusion of Lemma 7.17, A can distinguish Y_n from $\mathcal{U}_n$, which contradicts property 1 of $\widehat{G}$. □

As an example, we apply the theorem to the PRF parameters achieved by Corollary 7.13.

Corollary 7.20. *Suppose h is a symmetric, logspace-computable function. Let τ be the time bound from Lemma 7.16. If there is a $(2^{\Omega(\log^2 \lambda)}, 2^{-\Omega(\log^2 \lambda)})$-PRF $F_\lambda : \{0,1\}^{\mathsf{poly}(\lambda)} \times \{0,1\}^{\log^2 \lambda} \to \{0,1\}$, then for all $\Delta > 2$ there exists a constant $C > 0$ such that for all $\kappa > 0$, $\mathsf{KP\text{-}vs\text{-}Random}^\tau_h[n - \Delta \log n]$ is $(n^\kappa, (n^*)^{-C})$-hard on average*.*

References

[ABG+14] Adi Akavia, Andrej Bogdanov, Siyao Guo, Akshay Kamath, and Alon Rosen. Candidate weak pseudorandom functions in $AC^0 \circ MOD_2$. In Moni Naor, editor, *Innovations in Theoretical Computer Science, ITCS'14, Princeton, NJ, USA, January 12-14, 2014*, pages 251–260. ACM, 2014.

[ABK+06] Eric Allender, Harry Buhrman, Michal Koucký, Dieter van Melkebeek, and Detlef Ronneburger. Power from random strings. *SIAM J. Comput.*, 35(6):1467–1493, 2006.

[AIK06] Benny Applebaum, Yuval Ishai, and Eyal Kushilevitz. Cryptography in NC^0. *SIAM J. Comput.*, 36(4):845–888, 2006.

[AKRR11] Eric Allender, Michal Koucký, Detlef Ronneburger, and Sambuddha Roy. The pervasive reach of resource-bounded Kolmogorov complexity in computational complexity theory. *J. Comput. Syst. Sci.*, 77(1):14–40, 2011.

[All17] Eric Allender. The complexity of complexity. In Adam R. Day, Michael R. Fellows, Noam Greenberg, Bakhadyr Khoussainov, Alexander G. Melnikov, and Frances A. Rosamond, editors, *Computability and Complexity - Essays Dedicated to Rodney G. Downey on the Occasion of His 60th Birthday*, volume 10010 of *Lecture Notes in Computer Science*, pages 79–94. Springer, 2017.

[All20] Eric Allender. The new complexity landscape around circuit minimization. In Alberto Leporati, Carlos Martín-Vide, Dana Shapira, and Claudio Zandron, editors, *Language and Automata Theory and Applications - 14th International Conference, LATA 2020, Milan, Italy, March 4-6, 2020, Proceedings*, volume 12038 of *Lecture Notes in Computer Science*, pages 3–16. Springer, 2020.

[BCG+21] Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, and Peter Scholl. Low-complexity weak pseudorandom functions in AC0[MOD2]. In Tal Malkin and Chris Peikert, editors, *Advances in Cryptology - CRYPTO 2021 - 41st Annual International Cryptology Conference, CRYPTO 2021, Virtual Event, August 16-20, 2021, Proceedings, Part IV*, volume 12828 of *Lecture Notes in Computer Science*, pages 487–516. Springer, 2021.

[BL93] Dan Boneh and Richard J. Lipton. Amplification of weak learning under the uniform distribution. In Lenny Pitt, editor, *Proceedings of the Sixth Annual ACM Conference on Computational Learning Theory, COLT 1993, Santa Cruz, CA, USA, July 26-28, 1993*, pages 347–351. ACM, 1993.

[BLMP23] Marshall Ball, Yanyi Liu, Noam Mazor, and Rafael Pass. Kolmogorov comes to cryptomania: On interactive Kolmogorov complexity and key-agreement. In *64th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2023, Santa Cruz, CA, USA, November 6-9, 2023*, pages 458–483. IEEE, 2023.

[BR17] Andrej Bogdanov and Alon Rosen. Pseudorandom functions: Three decades later. In Yehuda Lindell, editor, *Tutorials on the Foundations of Cryptography*, pages 79–158. Springer International Publishing, 2017.

[BT06] Andrej Bogdanov and Luca Trevisan. Average-case complexity. *Found. Trends Theor. Comput. Sci.*, 2(1), 2006.

[CIKK16] Marco L. Carmosino, Russell Impagliazzo, Valentine Kabanets, and Antonina Kolokolova. Learning algorithms from natural proofs. In Ran Raz, editor, *31st Conference on Computational Complexity, CCC 2016, Tokyo, Japan, May 29 - June 1, 2016*, volume 50 of *LIPIcs*, pages 10:1–10:24. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016.

[CR73] Stephen A. Cook and Robert A. Reckhow. Time bounded random access machines. *J. Comput. Syst. Sci.*, 7(4):354–375, 1973.

[FLY22] Zhiyuan Fan, Jiatu Li, and Tianqi Yang. The exact complexity of pseudorandom functions and the black-box natural proof barrier for bootstrapping results in computational complexity. In Stefano Leonardi and Anupam Gupta, editors, *STOC '22: 54th Annual ACM SIGACT Symposium on Theory of Computing, Rome, Italy, June 20 - 24, 2022*, pages 962–975. ACM, 2022.

[GGM86] Oded Goldreich, Shafi Goldwasser, and Silvio Micali. How to construct random functions. *J. ACM*, 33(4):792–807, 1986.

[GK23] Halley Goldberg and Valentine Kabanets. Improved learning from Kolmogorov complexity. In Amnon Ta-Shma, editor, *38th Computational Complexity Conference, CCC 2023, Warwick, UK, July 17-20, 2023*, volume 264 of *LIPIcs*, pages 12:1–12:29. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023.

[HILL99] Johan Håstad, Russell Impagliazzo, Leonid A. Levin, and Michael Luby. A pseudorandom generator from any one-way function. *SIAM J. Comput.*, 28(4):1364–1396, 1999.

[Hir20] Shuichi Hirahara. Characterizing average-case complexity of PH by worst-case meta-complexity. In Sandy Irani, editor, *61st IEEE Annual Symposium on Foundations of Computer Science, FOCS 2020, Durham, NC, USA, November 16-19, 2020*, pages 50–60. IEEE, 2020.

[HLO26] Jinqiao Hu, Zhenjian Lu, and Igor C. Oliveira. Hardness of computing nondeterministic Kolmogorov complexity. In Dana Moshkovitz, editor, *41st Computational Complexity Conference, CCC 2026, Lisbon, Portugal, August 3-6, 2026*, volume 383 of *LIPIcs*, pages 9:1–9:50. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2026.

[HN23] Shuichi Hirahara and Mikito Nanashima. Learning in pessiland via inductive inference. In *64th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2023, Santa Cruz, CA, USA, November 6-9, 2023*, pages 447–457. IEEE, 2023.

[HN26] Shuichi Hirahara and Mikito Nanashima. Complexity-theoretic universal inductive inference. In Aditya Bhaskara and Artur Czumaj, editors, *Proceedings of the 58th Annual ACM Symposium on Theory of Computing, STOC 2026, Salt Lake City, UT, USA, June 22-26, 2026*, pages 377–385. ACM, 2026.

[IL89] Russell Impagliazzo and Michael Luby. One-way functions are essential for complexity based cryptography (extended abstract). In *30th Annual Symposium on Foundations of Computer Science, Research Triangle Park, North Carolina, USA, 30 October - 1 November 1989*, pages 230–235. IEEE Computer Society, 1989.

[ILO20] Rahul Ilango, Bruno Loff, and Igor C. Oliveira. NP-hardness of circuit minimization for multi-output functions. In Shubhangi Saraf, editor, *35th Computational Complexity Conference, CCC 2020, July 28-31, 2020, Saarbrücken, Germany (Virtual Conference)*,

volume 169 of *LIPIcs*, pages 22:1–22:36. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

[IW97] Russell Impagliazzo and Avi Wigderson. $P = BPP$ if E requires exponential circuits: Derandomizing the XOR lemma. In Frank Thomson Leighton and Peter W. Shor, editors, *Proceedings of the Twenty-Ninth Annual ACM Symposium on the Theory of Computing, El Paso, Texas, USA, May 4-6, 1997*, pages 220–229. ACM, 1997.

[KL18] Pravesh K. Kothari and Roi Livni. Improper learning by refuting. In Anna R. Karlin, editor, *9th Innovations in Theoretical Computer Science Conference, ITCS 2018, Cambridge, MA, USA, January 11-14, 2018*, volume 94 of *LIPIcs*, pages 55:1–55:10. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018.

[KV94] Michael J. Kearns and Umesh V. Vazirani. *An Introduction to Computational Learning Theory*. MIT Press, 1994.

[LO22] Zhenjian Lu and Igor C. Oliveira. Theory and applications of probabilistic Kolmogorov complexity. *Bull. EATCS*, 137, 2022.

[LP20] Yanyi Liu and Rafael Pass. On one-way functions and Kolmogorov complexity. In Sandy Irani, editor, *61st IEEE Annual Symposium on Foundations of Computer Science, FOCS 2020, Durham, NC, USA, November 16-19, 2020*, pages 1243–1254. IEEE, 2020.

[LP21] Yanyi Liu and Rafael Pass. Cryptography from sublinear-time average-case hardness of time-bounded Kolmogorov complexity. In Samir Khuller and Virginia Vassilevska Williams, editors, *STOC '21: 53rd Annual ACM SIGACT Symposium on Theory of Computing, Virtual Event, Italy, June 21-25, 2021*, pages 722–735. ACM, 2021.

[LP22] Yanyi Liu and Rafael Pass. Characterizing derandomization through hardness of Levin-Kolmogorov complexity. In Shachar Lovett, editor, *37th Computational Complexity Conference, CCC 2022, Philadelphia, PA, USA, July 20-23, 2022*, volume 234 of *LIPIcs*, pages 35:1–35:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022.

[LP24a] Yanyi Liu and Rafael Pass. A direct PRF construction from Kolmogorov complexity. In Marc Joye and Gregor Leander, editors, *Advances in Cryptology - EUROCRYPT 2024 - 43rd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zurich, Switzerland, May 26-30, 2024, Proceedings, Part IV*, volume 14654 of *Lecture Notes in Computer Science*, pages 375–406. Springer, 2024.

[LP24b] Yanyi Liu and Rafael Pass. On one-way functions, the worst-case hardness of time-bounded Kolmogorov complexity, and computational depth. In Elette Boyle and Mohammad Mahmoody, editors, *Theory of Cryptography - 22nd International Conference, TCC 2024, Milan, Italy, December 2-6, 2024, Proceedings, Part I*, volume 15364 of *Lecture Notes in Computer Science*, pages 222–252. Springer, 2024.

[MP25] Noam Mazor and Rafael Pass. Guest column: On cryptography and meta-complexity. *SIGACT News*, 56(2):63–101, 2025.

[NW94] Noam Nisan and Avi Wigderson. Hardness vs randomness. *J. Comput. Syst. Sci.*, 49(2):149–167, 1994.

[Raz87] Alexander A. Razborov. Lower bounds on the size of bounded depth circuits over a complete basis with logical addition. *Mathematical Notes of the Academy of Sciences of the USSR*, 41(4):333–338, 1987.

[RR97] Alexander A. Razborov and Steven Rudich. Natural proofs. *J. Comput. Syst. Sci.*, 55(1):24–35, 1997.

[RS21] Hanlin Ren and Rahul Santhanam. Hardness of KT characterizes parallel cryptography. In Valentine Kabanets, editor, *36th Computational Complexity Conference, CCC 2021, Toronto, Ontario, Canada (Virtual Conference), July 20-23, 2021*, volume 200 of *LIPIcs*, pages 35:1–35:58. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

[SB14] Shai Shalev-Shwartz and Shai Ben-David. *Understanding Machine Learning - From Theory to Algorithms*. Cambridge University Press, 2014.

[Sip83] Michael Sipser. A complexity theoretic approach to randomness. In David S. Johnson, Ronald Fagin, Michael L. Fredman, David Harel, Richard M. Karp, Nancy A. Lynch, Christos H. Papadimitriou, Ronald L. Rivest, Walter L. Ruzzo, and Joel I. Seiferas, editors, *Proceedings of the 15th Annual ACM Symposium on Theory of Computing, 25-27 April, 1983, Boston, Massachusetts, USA*, pages 330–335. ACM, 1983.

[Smo93] Roman Smolensky. On representations by low-degree polynomials. In *34th Annual Symposium on Foundations of Computer Science, Palo Alto, California, USA, November 3-5, 1993*, pages 130–138. IEEE Computer Society, 1993.

[ST17] Rocco A. Servedio and Li-Yang Tan. What circuit classes can be learned with non-trivial savings? In Christos H. Papadimitriou, editor, *8th Innovations in Theoretical Computer Science Conference, ITCS 2017, Berkeley, CA, USA, January 9-11, 2017*, volume 67 of *LIPIcs*, pages 30:1–30:21. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017.

[SV84] Larry J. Stockmeyer and Uzi Vishkin. Simulation of parallel random access machines by circuits. *SIAM J. Comput.*, 13(2):409–422, 1984.

[Tra84] Boris A. Trakhtenbrot. A survey of Russian approaches to perebor (brute-force searches) algorithms. *IEEE Ann. Hist. Comput.*, 6(4):384–400, 1984.

[Vad17] Salil P. Vadhan. On learning vs. refutation. In Satyen Kale and Ohad Shamir, editors, *Proceedings of the 30th Conference on Learning Theory, COLT 2017, Amsterdam, The Netherlands, 7-10 July 2017*, volume 65 of *Proceedings of Machine Learning Research*, pages 1835–1848. PMLR, 2017.

[Val84] Leslie G. Valiant. A theory of the learnable. *Commun. ACM*, 27(11):1134–1142, 1984.

[Vol99] Heribert Vollmer. *Introduction to Circuit Complexity - A Uniform Approach*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 1999.

[Yao82] Andrew Chi-Chih Yao. Theory and applications of trapdoor functions (extended abstract). In *23rd Annual Symposium on Foundations of Computer Science, Chicago, Illinois, USA, 3-5 November 1982*, pages 80–91. IEEE Computer Society, 1982.

A Proof of Lemma 2.5

Fix $c \geq 1$ and let m and B be as in the premise of the lemma statement. Fix an input length n and $f : \{0,1\}^n \to \{0,1\}$, $f \in \mathcal{C}\text{-}\mathsf{SIZE}[n^c]$. For $i \in \{0, 1, \ldots, m(n)\}$ let $\mathcal{D}_i$ be the distribution sampled as follows:

1. Sample $x^1, \ldots, x^{m(n)} \sim \mathcal{U}_n$ independently.
2. For $j \in [m(n)]$, set b^j to $f(x^j)$ if $j \leq i$ and to an independent random bit otherwise.
3. Output $\vec{x} \| \vec{b}$.

Note that $\mathcal{D}_{m(n)} = \mathsf{DP}^f(\mathcal{U}_{nm(n)})$ and $\mathcal{D}_0 = \mathcal{U}_{nm(n)+m(n)}$. Thus

$$\Pr_{w \sim \mathcal{D}_{m(n)}}[B(1^n, w) = 1] - \Pr_{w \sim \mathcal{D}_0}[B(1^n, w) = 1] \geq \frac{1}{10}.$$

Note that the left-hand side is equal to $\sum_{i=1}^{m(n)} (\Pr_{w \sim \mathcal{D}_i}[B(1^n, w) = 1] - \Pr_{w \sim \mathcal{D}_{i-1}}[B(1^n, w) = 1])$. Thus there must be some $i^* \in [m(n)]$ such that

$$\Pr_{w \sim \mathcal{D}_{i^*}}[B(1^n, w) = 1] - \Pr_{w \sim \mathcal{D}_{i^*-1}}[B(1^n, w) = 1] \geq \frac{1}{10m(n)}.$$

For $i \in [m(n)]$, let A_i be the following algorithm with i hardcoded that takes as input a string $y \in \{0,1\}^n$ and has access to $\mathsf{Ex}(f)$.

1. Draw $i-1$ samples $(x^1, f(x^1)), \ldots, (x^{i-1}, f(x^{i-1}))$ from $\mathsf{Ex}(f)$.
2. Sample $x^{i+1}, \ldots, x^{m(n)} \sim \mathcal{U}_n$ independently.
3. Sample independent uniformly random bits $b^i, b^{i+1}, \ldots, b^{m(n)}$.
4. Run $B(1^n, x^1, \ldots, x^{i-1}, y, x^{i+1}, \ldots, x^{m(n)}, f(x^1), \ldots, f(x^{i-1}), b^i, \ldots, b^{m(n)})$.
5. If B outputs 1, return b^i. Else, return $1 - b^i$.

We claim that when $y \sim \mathcal{U}_n$, A_{i^*} outputs $f(y)$ with probability at least $1/2 + 1/O(m(n))$. To see this, define

$$p_{\text{true}} = \Pr_{\substack{x^1,\ldots,x^{m(n)} \sim \mathcal{U}_n, \\ b^{i^*+1},\ldots,b^{m(n)} \sim \mathcal{U}_1}}[B(1^n, x^1, \ldots, x^{m(n)}, f(x^1), \ldots, f(x^{i^*-1}), f(x^{i^*}), b^{i^*+1}, \ldots, b^{m(n)}) = 1],$$

$$p_{\text{flip}} = \Pr_{\substack{x^1,\ldots,x^{m(n)} \sim \mathcal{U}_n, \\ b^{i^*+1},\ldots,b^{m(n)} \sim \mathcal{U}_1}}[B(1^n, x^1, \ldots, x^{m(n)}, f(x^1), \ldots, f(x^{i^*-1}), 1 - f(x^{i^*}), b^{i^*+1}, \ldots, b^{m(n)}) = 1], \text{ and}$$

$$p_{\text{rand}} = \Pr_{\substack{x^1,\ldots,x^{m(n)} \sim \mathcal{U}_n, \\ b^{i^*},\ldots,b^{m(n)} \sim \mathcal{U}_1}}[B(1^n, x^1, \ldots, x^{m(n)}, f(x^1), \ldots, f(x^{i^*-1}), b^{i^*}, b^{i^*+1}, \ldots, b^{m(n)}) = 1].$$

The probability that A_{i^*} correctly computes $f(y)$ on a random y is the probability that either $b^{i^*} = f(y)$ and B outputs 1 or $b^{i^*} = 1 - f(y)$ and B outputs 0. Thus

$$\Pr_{y \sim \mathcal{U}_n, \mathsf{Ex}, A_{i^*}}[A_{i^*}^{\mathsf{Ex}(f)}(y) = f(y)] = \frac{1}{2} p_{\text{true}} + \frac{1}{2}(1 - p_{\text{flip}}) = \frac{1}{2} + \frac{1}{2}(p_{\text{true}} - p_{\text{flip}}).$$

Note that $\Pr_{w\sim\mathcal{D}_{i^*-1}}[B(1^n,w)=1]=p_{\text{rand}}=\frac{1}{2}(p_{\text{true}}+p_{\text{flip}})$ and $\Pr_{w\sim\mathcal{D}_{i^*}}[B(1^n,w)=1]=p_{\text{true}}$. So

$$\frac{1}{10m(n)}\leq \Pr_{w\sim D_{i^*}}[B(1^n,w)=1]-\Pr_{w\sim D_{i^*-1}}[B(1^n,w)=1]=\frac{1}{2}(p_{\text{true}}-p_{\text{flip}}).$$

Thus $\Pr_{y\sim\mathcal{U}_n,\mathsf{Ex},A_{i^*}}[A_{i^*}(y)=f(y)]\geq 1/2+1/10m(n)$. Note that the sample complexity of A_{i^*} is $i^*\leq m$. Let $A_{i^*}^{x^1,\ldots,x^m,r}$ be the algorithm A_{i^*} with its samples hardcoded to $(x^1,f(x^1)),\ldots,(x^m,f(x^m))$ and with internal randomness set to r. It follows from an averaging argument that

$$\Pr_{x^1,\ldots,x^m,r}\left[\Pr_{y\sim\mathcal{U}_n}[A_{i^*}^{x^1,\ldots,x^m,r}(y)=f(y)]\geq\frac{1}{2}+\frac{1}{20m(n)}\right]\geq\frac{1}{20m(n)}.$$

Let $\widetilde{A}$ be the algorithm that randomly guesses $j\in[m(n)]$, draws $m(n)$ samples $x^1,\ldots,x^{m(n)}$ from its example oracle, samples randomness r, and then outputs the circuit for $A_j^{x^1,\ldots,x^{m(n)},r}$. It follows that with probability $1/20m(n)^2$ this circuit is $(1/2-1/O(m))$-close to f. The PAC-learner A as in the theorem statement is obtained by applying standard confidence boosting techniques (see, e.g., [KV94, SB14]).

□

ECCC ISSN 1433-8092
https://eccc.weizmann.ac.il