Proofs of Proximity for Context-Free Languages and Read-Once Branching Programs*

Oded Goldreich
Weizmann Institute of Science
oded.goldreich@weizmann.ac.il

Tom Gur
Weizmann Institute of Science
tom.gur@weizmann.ac.il

Ron D. Rothblum
Weizmann Institute of Science
ron.rothblum@weizmann.ac.il

February 16, 2015

Abstract

Proofs of proximity are probabilistic proof systems in which the verifier only queries a *sub-linear* number of input bits, and soundness only means that, with high probability, the input is close to an accepting input. In their minimal form, called *Merlin-Arthur proofs of proximity ($\mathcal{MAP}$)*, the verifier receives, in addition to query access to the input, also free access to an explicitly given short (sub-linear) proof. A more general notion is that of an *interactive proof of proximity ($\mathcal{IPP}$)*, in which the verifier is allowed to interact with an all-powerful, yet untrusted, prover. $\mathcal{MAP}$s and $\mathcal{IPP}$s may be thought of as the $\mathcal{NP}$ and $\mathcal{IP}$ analogues of property testing, respectively.

In this work we construct proofs of proximity for two natural classes of properties: (1) context-free languages, and (2) languages accepted by small read-once branching programs. Our main results are:

1. $\mathcal{MAP}$s for these two classes, in which, for inputs of length n, both the verifier's query complexity and the length of the $\mathcal{MAP}$ proof are $\widetilde{O}(\sqrt{n})$.
2. $\mathcal{IPP}$s for the same two classes with constant query complexity, poly-logarithmic communication complexity, and logarithmically many rounds of interaction.

*This research was partially supported by the Israel Science Foundation (grant No. 671/13).

Contents

1 Introduction

The field of property testing, initiated by Rubinfeld and Sudan [RS96] and Goldreich, Goldwasser and Ron [GGR98], studies a computational model that consists of probabilistic algorithms, called *testers*, that need to decide whether a given object has a certain global property or is far (say, in Hamming distance) from all objects that have the property, based only on a local view of the object.

A line of work [EKR04, BSGH+06, DR06, RVW13, GR15, FGL14, KR14] has considered the question of designing *proof systems* within the property testing model. The minimal type of such a proof system, which was recently studied by Gur and Rothblum [GR15], augments the property testing framework by replacing the tester with a *verifier* that receives, in addition to oracle access to the input, also free access to an explicitly given short (i.e., sub-linear length) proof. The guarantee is that for inputs that have the property there exists a proof that makes the verifier accept with high probability, whereas, for inputs that are far from the property, the verifier will reject *every* alleged proof with high probability. These proof systems can be thought of as the $\mathcal{NP}$ (or more accurately $\mathcal{MA}$) analogue of *property testing*, and are called *Merlin-Arthur proofs of proximity ($\mathcal{MAP}$)*.[1]

A more general notion was considered by Rothblum, Vadhan and Wigderson [RVW13] (prior to [GR15]). Their proof system, which can be thought of as the $\mathcal{IP}$ analogue of property testing, consists of an all powerful (but untrusted) prover who interacts with a verifier that only has oracle access to the input x. The prover tries to convince the verifier that x has a particular property Π. Here, the guarantee is that for inputs in Π, there exists a prover strategy that will make the verifier accept with high probability, whereas for inputs that are far from Π, the verifier will reject with high probability no matter what prover strategy is employed. The latter proof systems are known as *interactive proofs of proximity ($\mathcal{IPP}$s)*.[2]

The focus of this paper is identifying natural classes of properties that are known to be hard to test, but become easy to *verify* using the power of a proof ($\mathcal{MAP}$) or interaction with a prover ($\mathcal{IPP}$).

1.1 Our Results

One well-known class of properties that is hard to test is the class of *context-free languages*. Alon *et al.* [AKNS00] showed that there exists a context-free language that requires $\Omega\left(\sqrt{n}\right)$ queries to test (where here and throughout this work, n denotes the size of the input) and a context-free language that requires $\Omega(n)$ queries to test with *one-sided error*. Furthermore, there are no known (non-trivial) testers for general context-free languages.

Another interesting class is the class of languages that are accepted by small *read-once branching programs (ROBPs)*. Newman [New02] showed that the set of strings accepted by any small width ROBP can be efficiently tested.[3] More specifically, Newman showed that width w ROBPs can be tested using $(2^w/\varepsilon)^{O(w)}$ queries, where ε is the proximity parameter. Bollig [Bol05] showed that Newman's result cannot be extended to polynomial-sized ROBPs, by exhibiting an $O(n^2)$-sized

[1] A related notion is that of a *probabilistically checkable proof of proximity ($\mathcal{PCPP}$)* [BSGH+06, DR06]. $\mathcal{PCPP}$s differ from $\mathcal{MAP}$s in that the verifier is only given *query* (i.e., oracle) access to the proof, whereas in $\mathcal{MAP}$s, the verifier has free (*explicit*) access to the proof. Hence, $\mathcal{PCPP}$s are a $\mathcal{PCP}$ analogue of property testing.

[2] Indeed, $\mathcal{MAP}$s can be thought of as a restricted case of $\mathcal{IPP}$s, in which the interaction is limited to a single message sent from the prover to the verifier.

[3] The result in [New02] is stated only for *oblivious* ROBPs but in [Bol05, Section 1.3] it is stated that Newman's result holds also for general *non-oblivious* ROBPs.

ROBP that requires $\Omega(\sqrt{n})$ queries to test. No (non-trivial) testers for general ROBPs are known for width $\Omega(\sqrt{\log n})$.

In this work we consider the question of constructing *efficient* $\mathcal{MAP}$s and $\mathcal{IPP}$s for these two classes.[4] Here, by "efficient", we mean that *both* the *query complexity* (i.e., the number of queries performed by the verifier to the input) and the *proof complexity* (i.e., the length of the $\mathcal{MAP}$ proof) or *communication complexity* (i.e., the amount of communication with the $\mathcal{IPP}$ prover) are small and, in particular, sub-linear[5].

Our first pair of results are efficient $\mathcal{MAP}$s for context-free languages and for ROBPs. These $\mathcal{MAP}$s offer a multiplicative trade-off between the query and proof complexities. Here and throughout this work, $n \in \mathbb{N}$ specifies the length of the main input and $\varepsilon \in (0,1)$ denotes the proximity parameter.

Theorem 1.1. *For every context-free language $\mathcal{L}$ and every $k = k(n)$ such that $2 \leq k \leq n$, there exists an $\mathcal{MAP}$ for $\mathcal{L}$ that uses a proof of length $O(k \cdot \log n)$ and has query complexity $O\left(\frac{n}{k} \cdot \varepsilon^{-1}\right)$. Furthermore, the $\mathcal{MAP}$ has one-sided error.*

Theorem 1.2. *If a language $\mathcal{L}$ is recognized by a size $s = s(n)$ ROBP, then for every $k = k(n)$ such that $2 \leq k \leq n$, there exists an $\mathcal{MAP}$ for $\mathcal{L}$ that uses a proof of length $O(k \cdot \log s)$ and has query complexity $O\left(\frac{n}{k} \cdot \varepsilon^{-1}\right)$. Furthermore, the $\mathcal{MAP}$ has one-sided error.*

Hence, by setting $k = \sqrt{n}$, every context-free language and every language accepted by an ROBP of size at most $2^{\mathsf{polylog}(n)}$, has an $\mathcal{MAP}$ in which both the proof and query complexity are $\widetilde{O}\left(\sqrt{n}\right)$ (w.r.t. constant proximity parameter).

Next, we ask whether the query and proof complexity in Theorems 1.1 and 1.2 can be significantly reduced by allowing more extensive *interaction* between the verifier and the prover (i.e., arbitrary interactive communication rather than just a fixed non-interactive proof). Very relevant to this question is a recent result of [RVW13] by which, loosely speaking, every language in $\mathcal{NC}$ (which contains all context-free languages [Ruz81] and languages accepted by small ROBPs[6]) has an $\mathcal{IPP}$ with $\widetilde{O}(\sqrt{n})$ query and communication complexities. While the [RVW13] result is more general, for context-free languages and ROBPs it achieves roughly the same query and communication complexities as the $\mathcal{MAP}$s in Theorems 1.1 and 1.2, but uses much more interaction (i.e., at least logarithmically many rounds of interaction compared to just a single message in our $\mathcal{MAP}$s).

Using cryptographic assumptions[7], Kalai and Rothblum [KR14] recently showed that there exists a language in $\mathcal{NC}_1$ for which every $\mathcal{IPP}$ requires that either the query or communication complexity be $\Omega(\sqrt{n})$. Hence, we cannot hope to improve the [RVW13] result in general. Still, for the special case of context-free languages and ROBPs, we show that we can actually extend the $\mathcal{MAP}$ protocols in Theorems 1.1 and 1.2 into highly efficient $\mathcal{IPP}$s with only *poly-logarithmic* complexity (using a sub-logarithmic number of rounds). More generally, our $\mathcal{IPP}$s offer a trade-off between the number of rounds of interaction and the query and communication complexities.

[4] To see that these two classes do not contain each other, observe that the language $\{0^i 1^j 2^i 3^j : i, j \geq 1\}$, which is *not* a context-free language [HMU06, Example 7.20], has a $\mathsf{poly}(n)$-width ROBP (which simply counts the number of repeated occurrences of 0, 1, 2 and 3). On the other hand, Kriegal and Waack [KW88] showed that every ROBP for the Dyck_2 language, which is a context-free language, has size $2^{\Omega(n)}$.

[5] As pointed out in [GR15], if we do not restrict the length of the proof, then *every* property Π can be verified trivially using only a constant amount of queries, by considering an $\mathcal{MAP}$ proof that contains a full description of the input.

[6] See Appendix B for a discussion on why languages accepted by ROBPs can be computed in small depth.

[7] A sufficient assumption for [KR14] is the existence of (length-doubling) PRGs that can be computed in $\mathcal{NC}_1$ and whose output cannot be distinguished from random by circuits of size $2^{o(n)}$.

Theorem 1.3. *For every context-free language $\mathcal{L}$, every $k = k(n) \geq 2$ and $r = r(n) \geq 1$ such that $k^r \leq n$, there exists an r-round $\mathcal{IPP}$ for $\mathcal{L}$ with communication complexity $O((rk \log n) \cdot \varepsilon^{-1})$ and query complexity $O(\frac{n}{k^r} \cdot \varepsilon^{-1})$. Furthermore, the $\mathcal{IPP}$ is public-coin and has one-sided error.*

Theorem 1.4. *If a language $\mathcal{L}$ is recognized by a size $s = s(n)$ ROBP, then for every $k = k(n) \geq 2$ and $r = r(n) \geq 1$ such that $k^r \leq n$, there exists an r-round $\mathcal{IPP}$ for $\mathcal{L}$ with communication complexity $O((rk \log s) \cdot \varepsilon^{-1})$ and query complexity $O\left(\frac{n}{k^r} \cdot \varepsilon^{-1}\right)$. Furthermore, the $\mathcal{IPP}$ is public-coin and has one-sided error.*

(Interestingly, and in contrast to Theorems 1.1 and 1.2, here the communication complexity also depends on the proximity parameter ε.) In particular, by setting $k = \log n$ and $r = \frac{\log n}{\log \log n}$, we obtain $\mathcal{IPP}$s for context-free languages and size $2^{\mathsf{polylog}(n)}$ ROBPs, with a sub-logarithmic number of rounds, constant query complexity, and poly-logarithmic communication complexity (w.r.t. constant proximity parameter).

A Remark on Computational Complexity. Following the property testing literature, we view the query complexity and the proof complexity (resp., communication complexity) as the primary resources of an $\mathcal{MAP}$ (resp., $\mathcal{IPP}$). Still, the running time of the verifier and of the prover are also important resources. The proofs/provers in our $\mathcal{MAP}$s and $\mathcal{IPP}$s are indeed efficient; that is, polynomial in the main input x (and in the case of ROBPs also in the size of the ROBP).

As for our verifiers, those in Theorems 1.1 and 1.3 run in polynomial time (i.e., $\mathsf{poly}(|x|)$ time) rather than in *sub-linear* time as one might hope. However, by increasing the round complexity in Theorem 1.3 by a poly-logarithmic factor, we can obtain an $\mathcal{IPP}$ with sub-linear time verification. Constructing an $\mathcal{MAP}$ for context-free languages with sub-linear time verification remains an interesting open question. The verifiers in Theorems 1.2 and 1.4 run in *sub-linear time* if they are given a *suitable* (natural) representation of the ROBP.[8] See the technical sections (specifically Remark 3.2 and Remark 4.6) for further details.

Improved Results for Specific Languages. The paradigm used for the general results in Theorems 1.1-1.4 can be extended to yield better results for specific languages. A notable class of languages for which we obtain such an improvement is the class of languages of balanced parentheses expressions (a.k.a the Dyck languages), which are context-free languages, for which Parnas *et al.* [PRR01] showed a lower bound of $\widetilde{\Omega}(n^{1/11})$ for ordinary testers. Using special properties of the Dyck languages, we can improve on the general result in Theorem 1.1 in this special case and obtain a somewhat more efficient $\mathcal{MAP}$ for the Dyck languages. See details in Section 4.3.

1.2 Proof Overview

The proofs of Theorems 1.1 and 1.2 (i.e., the $\mathcal{MAP}$ results) will follow (roughly) as special cases of the proofs of Theorems 1.3 and 1.4 (i.e., the $\mathcal{IPP}$ results), respectively. Hence, in this overview we focus on the proofs of Theorems 1.3 and 1.4, while explaining how to derive Theorems 1.1 and 1.2 as special cases.

[8] Indeed, the running time of the verifier crucially relies on the specific representation of the ROBP. We remark that there are other natural representations of ROBPs than the one we use, and for some of these representations obtaining sub-linear running time may not be feasible.

The proofs of Theorems 1.3 and 1.4 share a common theme: For $\mathcal{L}$ that is either a context-free language or is accepted by a ROBP, we show that every input $x \in \mathcal{L}$ can be broken-down into k sub-problems (related to $\mathcal{L}$) such that the following holds:

1. On the one hand, if $x \in \mathcal{L}$, then there exists (1) a partition of $[n]$ into sets $S_1, \ldots, S_k$ (each of size roughly n/k); and (2) languages $\mathcal{L}_1, \ldots, \mathcal{L}_k$ such that both (1) and (2) have a concise representation, and, for every $i \in [k]$, the projection of x on S_i, denoted $x[S_i]$, is in the language $\mathcal{L}_i$. Furthermore, if $\mathcal{L}$ is a context-free language (resp., accepted by an ROBP), then the languages $\mathcal{L}_1, \ldots, \mathcal{L}_k$ are all "variants" of context-free languages[9] (resp., accepted by ROBPs).

2. On the other hand, if x is "far" from $\mathcal{L}$, then for every concise representation of a partition $S_1, \ldots, S_k$ of $[n]$ and languages $\mathcal{L}_1, \ldots, \mathcal{L}_k$ (of the type used in 1), for an average $i \in [k]$, it holds that $x[S_i]$ is proportionally "far" from $\mathcal{L}_i$.

By design, the partition $S_1, \ldots, S_k$ as well as the corresponding languages $\mathcal{L}_1, \ldots, \mathcal{L}_k$ depend on the entire input x, and so the verifier (who only has query access to x) cannot generate them by itself. Instead, the concise representation of $S_1, \ldots, S_k$ and $\mathcal{L}_1, \ldots, \mathcal{L}_k$ will be specified by the prover (as a single message in the case of an $\mathcal{IPP}$, or as the entire proof string in the case of an $\mathcal{MAP}$).

Given the latter, we construct an $\mathcal{MAP}$ as follows. The $\mathcal{MAP}$ verifier selects at random a small subset $I \subseteq [k]$ and, for every $i \in I$, reads *all* of $x[S_i]$ (which is of length roughly n/k) and checks that $x[S_i] \in \mathcal{L}_i$. Indeed, by the two foregoing conditions, if $x \in \mathcal{L}$, then $x[S_i] \in \mathcal{L}_i$ for every $i \in [k]$, whereas if x is "far" from $\mathcal{L}$, then, by an averaging argument, for many $i \in [k]$, it holds that $x[S_i]$ is proportionally "far" from $\mathcal{L}_i$ (and in particular $x[S_i] \notin \mathcal{L}_i$), and the verifier will reject.

A natural approach for extending the foregoing $\mathcal{MAP}$ to an $\mathcal{IPP}$ is to have the verifier send the set I (where I is chosen at random as in the $\mathcal{MAP}$) to the prover, and then *recursively* run $|I|$ $\mathcal{IPP}$ protocols to check that $x[S_i]$ is close to $\mathcal{L}_i$, for every $i \in I$. In each recursive call the input shrinks by (roughly) a factor of k. After the recursion reaches depth r, where r is a predetermined bound on the number of rounds, the verifier can simply read its entire current input (of length $O(n/k^r)$) and decide whether to accept or reject.

The foregoing approach indeed works, but because there is more than one recursive call in each round, the complexity of the resulting $\mathcal{IPP}$ depends *exponentially* on the number of rounds r. Instead, we use a more economical approach, which avoids the exponential dependence on r, based on the notion of a *proximity oblivious tester* [GR11]. Recall that a proximity oblivious tester for a property Π is a tester that does not receive the proximity parameter ε as input and is only required to reject inputs that are ε-far from Π with probability proportional to ε (rather than probability $2/3$). To present a more economical recursion, the $\mathcal{IPP}$ that we design is similarly "proximity oblivious". The idea is to have the verifier select at random only a single index $i \in [k]$, send i to the prover, and then have the two parties recursively run an $\mathcal{IPP}$ protocol for verifying that $x[S_i]$ is close to $\mathcal{L}_i$. Indeed, if $x \in \mathcal{L}$ then $x[S_i] \in \mathcal{L}_i$, whereas if x is ε-far from $\mathcal{L}$, then, since i was chosen at random, on the average $x[S_i]$ is ε-far from $\mathcal{L}_i$, and therefore, by inductive reasoning, the

[9] If $\mathcal{L}$ is a context-free language, then the languages $\mathcal{L}_1, \ldots, \mathcal{L}_k$ will be variants of context-free languages, which we call "partial derivation languages". However, if $\mathcal{L}$ is accepted by an ROBP, then the languages $\mathcal{L}_1, \ldots, \mathcal{L}_k$ are also accepted by (different) ROBPs.

verifier will reject with probability ε. To obtain constant soundness we can just repeat[10] the entire proximity oblivious protocol $O(1/\varepsilon)$ times in parallel.

This concludes the high-level description of our $\mathcal{MAP}$s and $\mathcal{IPP}$s. Of course, the way in which the partition is generated is quite different in the case of context-free languages and in the case of ROBP, and different technical problems arise in each case. In the following subsections we discuss the specific details. In Section 1.2.1 we give an overview of how to partition read-once branching programs. Partitioning context-free languages is more involved, and so, in Section 1.2.2, as a warm-up, we first consider partitioning into *two parts* (i.e., $k = 2$). Then, in Section 1.2.3 we show how to extend the technique to *multiple parts* (i.e., general $k \geq 2$).

1.2.1 Partitioning ROBPs

Recall that a branching program on n variables is a directed acyclic graph with a unique source vertex with in-degree 0 and (possibly) multiple sink vertices with out-degree 0. Each sink vertex is labeled with either 0 (i.e., *reject*) or 1 (i.e., *accept*). Each non-sink vertex is labeled by an index $i \in [n]$ and has exactly 2 outgoing edges, which are labeled by 0 and 1. The output of the branching program B on input $x \in \{0,1\}^n$, denoted $B(x)$, is computed in a natural way by starting at the source vertex and taking a walk such that at a vertex labeled by $i \in [n]$, we traverse the outgoing edge labeled by x_i. Once a sink is reached, we output its label. The branching program is *read-once* (ROBP for short) if along every path from source to sink, every index ($i \in [n]$) appears at most once. The *size* of a branching program B, denoted $|B|$, is the number of vertices in it.

For any fixed ROBP B, we construct an $\mathcal{IPP}$ (and an $\mathcal{MAP}$, which is a special case of the $\mathcal{IPP}$) for the language accepted by B, denoted $\mathcal{L}_B \stackrel{\text{def}}{=} \{x \in \{0,1\}^n : B(x) = 1\}$. In this overview, we make a simplifying assumption that B is both *layered* and *ordered* (a.k.a., an *ordered binary decision diagram* or OBDD). That is, we assume that the vertices of B are partitioned into $n+1$ layers such that, for every $i \in [n]$, edges only go from layer i to layer $i+1$; and vertices in layer i are labeled by the index i (i.e., the ROBP reads its input "in order").

The key idea, which enables the $\mathcal{IPP}$ verifier to generate the aforementioned partition $S_1, \ldots, S_k$ (together with the corresponding languages), is to have the prover specify k evenly-spaced vertices along the accepting path corresponding to the input $x \in \mathcal{L}_B$. More specifically, observe that x induces a path $\varphi_0 \to \varphi_1 \to \cdots \to \varphi_n$ from the start vertex φ_0 to some accepting sink φ_n. The prover sends to the verifier a subsequence of this walk, specifically the subsequence $\varphi_{n/k}, \ldots, \varphi_{i \cdot n/k}, \ldots, \varphi_n$.

Given the subsequence, we can reduce the problem of verifying that there exists a path of length n from φ_0 to φ_n to verifying that there exists a path of length n/k between each pair of consecutive vertices in the sequence $\varphi_0, \varphi_{n/k}, \ldots, \varphi_{i \cdot n/k}, \ldots, \varphi_n$. In other words, for every $i \in [k]$ we consider the ROBP B_i that consists only of layers $(i-1) \cdot n/k$ up to $i \cdot n/k$ of B, with the starting state $\varphi_{(i-1) \cdot n/k}$ and the (only) accepting state $\varphi_{i \cdot n/k}$. Verifying that $x \in \mathcal{L}_B$ can be reduced to verifying that $x[S_i] \in \mathcal{L}_{B_i}$, for every $i \in [k]$, where $S_i \subseteq [n]$ is the set of coordinates of x that are read by B_i and $\mathcal{L}_{B_i} \stackrel{\text{def}}{=} \{z \in \{0,1\}^{n/k} : B_i(z) = 1\}$. Moreover, since $S_1, \ldots, S_k$ is a partition of $[n]$, if x is ε-far from $\mathcal{L}_B$, then $x[S_i]$ is ε-far from $\mathcal{L}_{B_i}$, for an average $i \in [k]$. Hence, we can follow the high-level outline that was suggested in Section 1.2; that is, the $\mathcal{IPP}$ verifier selects $i \in [k]$ at random, sends i to the prover, and then the two parties recursively run an $\mathcal{IPP}$ protocol to verify that $x[S_i]$ is close to the $\mathcal{L}_{B_i}$.

[10] As expected, parallel repetition reduces the soundness error of $\mathcal{IPP}$s at an exponential rate. See Appendix A for details.

The foregoing intuition almost works but there is a subtle problem: What if the message sent by a *cheating* prover is such that $\mathcal{L}_{B_{i^*}}$ is empty, for some $i^* \in [k]$. This corresponds to a situation in which the branching program B contains no path from $\varphi_{(i^*-1)\cdot n/k}$ to $\varphi_{i^*\cdot n/k}$. In such case, with high probability (i.e., if the verifier chooses i such that $i \neq i^*$) the verifier, as described so far, will not notice this fact and may accept inputs that are far from $\mathcal{L}_B$.

We overcome this difficulty by observing that when the verifier interacts with the honest prover, it holds that $x[S_i] \in \mathcal{L}_{B_i}$ for every $i \in [k]$, and therefore $\mathcal{L}_{B_i} \neq \emptyset$. Hence, we can have the verifier explicitly check that $\mathcal{L}_{B_i} \neq \emptyset$ for *every* $i \in [k]$ (i.e., that there exists *some* input that leads from $\varphi_{(i-1)\cdot n/k}$ to $\varphi_{i\cdot n/k}$ in B). This check requires direct and full access to the branching program B (which is fixed) but does *not* require any queries to the input x, and so we can perform it for *every*[11] $i \in [k]$.

Given this additional check, we can show that the foregoing $\mathcal{IPP}$ works. To do so, we argue by induction on the number of rounds that if the input x is ε-far from $\mathcal{L}$ then the verifier rejects with probability at least ε. Indeed, if x is ε-far from $\mathcal{L}_B$, then in the first round we have that:

$$\begin{aligned}\Pr\left[\text{Verifier for } \mathcal{L}_B \text{ rejects } x\right] &= \mathop{\mathbf{E}}_i\left[\Pr\left[\text{Verifier for } \mathcal{L}_{B_i} \text{ rejects } x[S_i]\right]\right] \\ &\geq \mathop{\mathbf{E}}_i\left[\varepsilon_i\right] \\ &\geq \varepsilon,\end{aligned}$$

where ε_i denotes the relative distance of $x[S_i]$ from $\mathcal{L}_{B_i}$, for every $i \in [k]$, and the first inequality follows from the induction hypothesis.

We remark that when dealing with general ROBPs, rather than OBDDs, there are several additional technical difficulties. In particular, since B is not layered, we have to modify our definition of B_i (which previously consisted of layers $(i-1)\cdot n/k$ to $i \cdot n/k$ of B). A natural approach is to define B_i to consist of all paths (in B) of length n/k starting at $\varphi_{(i-1)\cdot n/k}$.[12] The difficulty is that B_i may depend on many, possibly even all, of the bits of x (since different paths may look at different bits), rather than just n/k bits (as was the case for OBDDs). Hence, the input does not necessarily shrink in the recursive step. Nevertheless, we resolve this issue by showing that the *effective length* of the input, which is the number of bits that need to be read in order to determine whether the ROBP accepts, does shrink, and this suffices to make progress in the recursion. For further details, see Section 3.

1.2.2 Partitioning Context-Free Languages into Two Parts

Recall that a *context-free grammar* is a tuple $G = (V, \Sigma, R, A_{\mathsf{start}})$, where $V = \{A_1, A_2, \dots\}$ denotes a (finite) set of variables, $\Sigma = \{\sigma_1, \sigma_2, \dots\}$ denotes a (finite) set of terminal symbols (i.e., the alphabet), R is a set of production rules (e.g., rules of the form $A_7 \to \sigma_5 A_3 A_9 \sigma_8 A_2$) and $A_{\mathsf{start}} \in V$ denotes a special "start" variable. We say that a string $\alpha \in (\Sigma \cup V)^*$ is *derived* from a variable A_j, denoted by $A_j \stackrel{*}{\Rightarrow} \alpha$, if α can be obtained from A_j by iteratively applying production rules in R. Each such derivation can be described by a *derivation tree*, which is a rooted, directed, ordered, and labeled tree (with edges oriented away from the root), where the root is labeled by A_j, the leaves

[11] However, this check does increase the running time of the verifier (which we view as a secondary resource) to $\mathsf{poly}(|B|)$. This computation can be minimized by using a pre-processing step in which we compute a $|B| \times |B|$-sized table whose $(v,u)^{\text{th}}$ entry says whether the vertices v and u are connected in B.

[12] The actual definition of B_i that we use is different. See Section 3 (in particular Footnote 18).

are labeled by the sybmols of α (in order), and the children of each vertex in the tree correspond to an application of a production rule in G. The language $\mathcal{L} \subseteq \Sigma^*$ *generated* by G consists of all strings that can be derived from A_{start} using the production rules in R.

Let $\mathcal{L}$ be a context-free language and let $G = (V, \Sigma, R, A_{\mathsf{start}})$ be the context-free grammar that generates $\mathcal{L}$. In this section we show how to partition $x \in \mathcal{L}$ into two parts. Next, in Section 1.2.3, we show how to extend this technique to multiple parts.

For $x \in \mathcal{L}$ (i.e., $A_{\mathsf{start}} \stackrel{*}{\Rightarrow} x$), there exists a derivation tree T corresponding to the derivation $A_{\mathsf{start}} \stackrel{*}{\Rightarrow} x$. For simplicity, let us assume that T is a *binary* tree. The root of T is labeled by A_{start} and the leaves are labeled, in order, by $x_1, \ldots, x_n$, where $n \stackrel{\text{def}}{=} |x|$. Recall that the Lewis-Stearns-Harmanis Lemma [LSH65] states that every binary tree on n leaves has a subtree[13] with a number of leaves between $n/3$ and $2n/3$. Applying this lemma to T, we can find such a subtree T' of T. Observe that T' induces a partition of $[n]$ into two parts $S_1, S_2 \subseteq [n]$, where S_1 (which is actually an interval) contains all the leaves of T that belong to T' and $S_2 \stackrel{\text{def}}{=} [n] \setminus S_1$ contains all other leaves. The $\mathcal{IPP}$ prover finds T' and sends S_1 and A_1 to the verifier, where A_1 is the label of the root of T'. Since S_1 is an interval, the latter requires only $O(\log n)$ communication.

Given (S_1, A_1), the verifier can construct the partition and the corresponding languages, where the partition is simply (S_1, S_2) and the languages are

$$\mathcal{L}_1 \stackrel{\text{def}}{=} \left\{ w \in \Sigma^{|S_1|} \,:\, A_1 \stackrel{*}{\Rightarrow} w \right\}$$

and

$$\mathcal{L}_2 \stackrel{\text{def}}{=} \left\{ w \in \Sigma^{|S_2|} \,:\, A_2 \stackrel{*}{\Rightarrow} w[1, \ldots, s-1] \circ A_1 \circ w[s, \ldots, |S_2|] \right\},$$

where $A_2 \stackrel{\text{def}}{=} A_{\mathsf{start}}$ and $s \in [n]$ is the starting position of the interval S_1 in $[n]$.

Note that $\mathcal{L}_2$ is not quite a context-free language (although $\mathcal{L}_1$ is). Rather, $\mathcal{L}_2$ consists of strings that correspond to *partial derivations* (i.e., derivation processes that end before all symbols are terminals) starting from A_{start} that produce strings that have the variable A_1 in their s^{th} coordinate. We refer to such languages, which we view as generalization of context-free languages, as *partial derivation languages*, and for the recursion to go through, we actually design the original protocol to handle not only context-free languages but also partial derivation languages.

Observe that if $x \in \mathcal{L}$, then clearly $x[S_1] \in \mathcal{L}_1$ and $x[S_2] \in \mathcal{L}_2$. On the other hand, suppose that $x[S_1]$ is ε_1-close to a string $z_1 \in \mathcal{L}_1$ and $x[S_2]$ is ε_2-close to a string $z_2 \in \mathcal{L}_2$. If we choose $i \in \{1, 2\}$ at random, such that $\Pr[i = 1] = |S_1|/n$ and $\Pr[i = 2] = |S_2|/n$, then x is $\mathbf{E}_i[\varepsilon_i]$-close to the string $z = z_2[1, \ldots, s-1] \circ z_1 \circ z_2[s, |S_2|]$. Since $A_1 \stackrel{*}{\Rightarrow} z_1$ and $A_{\mathsf{start}} \stackrel{*}{\Rightarrow} z_2[1, \ldots, s-1] \circ A_1 \circ z_2[s, \ldots, |S_2|]$ (because $z_1 \in \mathcal{L}_1$ and $z_2 \in \mathcal{L}_2$), we deduce that $A_{\mathsf{start}} \stackrel{*}{\Rightarrow} z$, and therefore $z \in \mathcal{L}$. Hence, x is $\mathbf{E}_i[\varepsilon_i]$-close to $\mathcal{L}$.

Given the above, we can design an $\mathcal{IPP}$ for $\mathcal{L}$ similarly to the $\mathcal{IPP}$ for ROBP that was described in Section 1.2.1. Specifically, given (S_1, A_1), the verifier chooses at random $i \in \{1, 2\}$ according to the distribution above, sends i to the prover, and both parties run the protocol recursively, with respect to the language $\mathcal{L}_i$ and the input $x[S_i]$.

[13] Here and throughout this work, by a subtree, we mean a node of the tree together with *all* of its descendants, see also Section 2.3.

1.2.3 Partitioning Context-Free Languages into Multiple Parts

The first step in partitioning context-free languages into *multiple* parts is a generalization of the Lewis-Stearns-Hartmanis lemma that shows that, for every desired parameter $t \in [n]$, every (constant degree) tree T with n leaves has a subtree with roughly t leaves. The precise statement of the lemma and its proof are given in Lemma 2.5 below.

Using Lemma 2.5, we can partition an input $x \in \mathcal{L}$ into k parts of (roughly) the same size in the following way. As before, we construct a derivation tree T corresponding to the derivation $A_{\mathsf{start}} \overset{*}{\Rightarrow} x$. However, this time we use Lemma 2.5 to find a subtree T_1 with roughly n/k leaves. The coordinates of the leaves of T_1 constitute the first part of the partition (denoted by S_1). To find the second subtree, we remove the entire subtree T_1 from T, *except for its root*. We obtain a new tree T' with (roughly) $n - \frac{n}{k}$ leaves, where one of the leaves of T' is labeled by a variable rather than a terminal. By applying Lemma 2.5 again on the new tree T', we can find a subtree T_2 of T' with roughly n/k leaves. The second part (denoted by S_2) of our partition will consist of the coordinates of all the leaves of T_2 that are labeled by terminals (i.e., are also leaves of the original tree T). We stress that S_2 may not be an interval (but rather two intervals separated by S_1).

We proceed similarly, where in each iteration we remove the subtree that was found in the previous iteration (except for its root) and find a new subtree T_i of T with roughly n/k leaves. The subtrees $T_1, T_2, \ldots, T_k$ induce a partition of $[n]$ where the i^{th} part, denoted S_i (of size roughly n/k), consists of all leaves of T_i that are labeled by terminals (i.e., are leaves of the original tree T) but do not belong to $S_1 \cup \cdots \cup S_{i-1}$.

While the representation of a general partition of $[n]$ into k parts requires $n \cdot \log_2(k)$ bits, we show that the partition $S_1, \ldots, S_\ell$ actually has a concise representation. Indeed, each subtree T_i induces an interval $I_i \subseteq [n]$, which contains all of its leaves (but potentially also coordinates of other parts in the partition). Given $I_1, \ldots, I_\ell$, the partition $S_1, \ldots, S_\ell$ is uniquely determined (by setting $S_i = I_i \backslash (I_1 \cup \cdots \cup I_{i-1})$). We remark that each pair of *intervals* can be either disjoint or nested (i.e., either $I_i \cap I_j = \emptyset$ or $I_i \subsetneq I_j$).

In light of the foregoing discussion, the prover can send to the verifier the intervals $I_1, \ldots, I_k$ and the variables $A_1, \ldots, A_\ell$ of the roots of the subtrees $T_1, \ldots, T_k$ (respectively). Note that the root of the last subtree T_k is in fact the root of the original derivation tree T (and thus $A_k = A_{\mathsf{start}}$) and that its corresponding interval I_k is $[n]$.

Let $I_{i_1}, \ldots, I_{i_k}$ be the ordered (from left to right) maximal intervals of $I_k = [n]$. That is, the (disjoint) intervals that are contained in I_k but are not contained in any of the other intervals. Observe that if the intervals were generated as prescribed, then A_{start} yields a string x' (composed of terminals and variables) that results from x by replacing the substring $x[I_{i_j}]$ with the variable A_{i_j}, for every $j \in [k]$. Denote the language that contains all such strings by $\mathcal{L}_k$. Similarly, for any interval $I_{i_j} \in \{I_{i_1}, \ldots, I_{i_k}\}$, observe that A_{i_j} yields the string that results from $x[I_{i_j}]$ by replacing coordinates in the maximal intervals that I_{i_j} contains with the corresponding variables. Denote the language of all such strings by $\mathcal{L}_{i_j}$. We show that by applying this idea iteratively we obtain languages $\mathcal{L}_1, \ldots, \mathcal{L}_k$ such that (1) if $x \in \mathcal{L}$, then $x[S_i] \in \mathcal{L}_i$ for every $i \in [k]$; and (2) if x is ε-far from $\mathcal{L}$, then $x[S_i]$ is ε-far from $\mathcal{L}_i$, for an average $i \in [k]$, where the average is weighted proportionally to the sizes of $S_1, \ldots, S_k$.

Given the partition above, verifying that $x \in \mathcal{L}$ is reduced to testing that the sub-input $x[S_i]$ is close to $\mathcal{L}_i$, for $i \in [k]$ distributed as above. Hence, as before, the verifier chooses i at random, sends i to the prover and the two parties recursively run an $\mathcal{IPP}$ for verifying that $x[S_i]$ is ε-close to $\mathcal{L}_i$.

We emphasize that, as was the case for $k = 2$, the languages $\mathcal{L}_1, \ldots, \mathcal{L}_k$ are not necessarily *context-free languages* but are rather "partial derivation languages". Indeed, for the recursion to go through, we design the $\mathcal{IPP}$ to work for such languages (rather than just context-free languages).

1.2.4 Digest and Relation to Concatenation Problems

The proofs of Theorems 1.1-1.4 are based on a natural paradigm for designing proofs of proximity. This paradigm consists of two steps: (1) partition the problem into smaller related sub-problems, and (2) verifying a small random sample of the sub-problems. This basic approach was taken by [RVW13] in their construction of an $\mathcal{IPP}$ for the Hamming weight problem (i.e., approximating whether a given string has Hamming weight $n/2$). The partitioning in this case is into several intervals of equal length and the $\mathcal{IPP}$ prover specifies the Hamming weight of each substring. A more general instantiation of this paradigm was used in [GR15] to construct $\mathcal{MAP}$s for *parameterized concatenation problems*. Loosely speaking, a language $\mathcal{L}$ is a parameterized concatenation problem if $\mathcal{L} = \mathcal{L}_{\alpha_1} \times \cdots \times \mathcal{L}_{\alpha_k}$, for some integer k, where each language $\mathcal{L}_{\alpha_i}$ is a language parameterized by α_i; thus, the partitioning is done by providing the parameters $\alpha_1, \ldots, \alpha_k$.

In this work we significantly extend the foregoing framework in several aspects: The partition is not restricted to contiguous intervals, but is rather more involved and depends more dramatically on the structure of the specific language and, moreover, also on the specific input. Furthermore, whereas for concatenation problems the parameterization of each problem is "light" (typically having a logarithmic description length), in our settings the parameterization can be quite extensive, as in *massively parameterized problems* (see survey by Newman [New10]).

1.3 Organization

In Section 2 we provide the necessarily preliminaries regarding proofs of proximity, context-free languages, and branching programs. In Section 3 we construct $\mathcal{MAP}$s and $\mathcal{IPP}$s for languages accepted by ROBPs (with additional discussion on testing affine subspaces in Section 3.3). In Section 4 we construct $\mathcal{MAP}$s and $\mathcal{IPP}$s for context-free languages (with additional discussion on the Dyck languages in Section 4.3). Sections 3 and 4 can be read independently of each other. We note that the implementation of the outline provided in Section 1.2 is far more involved in the case of context-free languages.

2 Preliminaries

We begin with some standard notations:

- We denote the concatenation of two strings $x \in \Sigma^n$ and $y \in \Sigma^m$ (over a common alphabet Σ) by $x \circ y \in \Sigma^{n+m}$.
- We denote the *absolute distance* between two (equal length) strings $x \in \Sigma^n$ and $y \in \Sigma^n$ by $\overline{\Delta}(x, y) \stackrel{\text{def}}{=} |\{x_i \neq y_i : i \in [n]\}|$, and their *relative distance* by $\Delta(x, y) \stackrel{\text{def}}{=} \frac{\overline{\Delta}(x,y)}{n}$. If $\Delta(x, y) \leq \varepsilon$, we say that x is ε-close to y, and otherwise we say that x is ε-far from y. Similarly, we denote the *absolute distance* of x from a non-empty set $S \subseteq \Sigma^n$ by $\overline{\Delta}(x, S) \stackrel{\text{def}}{=} \min_{y \in S} \overline{\Delta}(x, y)$ and the *relative distance* of x from S by $\Delta(x, S) \stackrel{\text{def}}{=} \min_{y \in S} \Delta(x, y)$. If $\Delta(x, S) \leq \varepsilon$, we say that x is ε-close to S, and otherwise we say that x is ε-far from S.

- We denote the projection of a string $x \in \Sigma^n$ to a subset of coordinates $S \subseteq [n]$ by $x[S]$. For every $i, j \in [n]$, we denote by $x[i, j]$ the projection of x to the interval $[i, j]$ (if $i > j$ then the interval is empty).
- We denote by $A^x(y)$ the output of algorithm A, given direct access to input y and query (i.e., oracle) access to the string x. Given two interactive machines A and B, we denote by $(A^x, B(y))(z)$ the output of A when interacting with B, where A (resp., B) is given oracle access to x (resp., direct access to y) and both parties have direct access to z.

Integrality. Throughout this work, for simplicity of notation, we use the convention that all (relevant) integer parameters that are stated as real numbers are implicitly rounded to the closest integer.

2.1 Property Testing, $\mathcal{MAP}$s and $\mathcal{IPP}$s

In this section we define testers, $\mathcal{MAP}$s and $\mathcal{IPP}$s. Actually, testers and $\mathcal{MAP}$s will be defined as restrictions of $\mathcal{IPP}$s.

2.1.1 $\mathcal{IPP}$

We define a language, over an alphabet Σ, as an ensemble $\mathcal{L} \stackrel{\text{def}}{=} \cup_{n\in\mathbb{N}}\mathcal{L}_n$, where $\mathcal{L}_n \subseteq \Sigma^n$ for every $n \in \mathbb{N}$. The definition of an $\mathcal{IPP}$ is a natural extension of the standard definition of $\mathcal{IP}$ (interactive proof) where the main distinction is that the verifier only has oracle access to the input. Also, since our focus is on the query and communication complexities, we do not restrict the computational complexity of the verifier (see discussion at the end of Section 1).

Definition 2.1 (Interactive Proof of Proximity ($\mathcal{IPP}$) [EKR04, RVW13])**.** *An* interactive proof of proximity *($\mathcal{IPP}$) for the language $\mathcal{L} = \cup_{n\in\mathbb{N}}\mathcal{L}_n$ is an interactive protocol with two parties: a (computationally unbounded) prover $\mathcal{P}$, which has* free *access to input x, and a verifier $\mathcal{V}$, which is a probabilistic computationally unbounded algorithm which has oracle access to x. The parties send messages to each other, and at the end of the communication, the following two conditions are satisfied:*

1. **Completeness:** *For every $n \in \mathbb{N}$, proximity parameter $\varepsilon > 0$ and $x \in \mathcal{L}_n$ it holds that*

$$\Pr\left[(\mathcal{V}^x, \mathcal{P}(x))(n, \varepsilon) = 1\right] \geq 2/3.$$

   where the probability is over the coin tosses of $\mathcal{V}$.

2. **Soundness:** *For every $n \in \mathbb{N}$, $\varepsilon > 0$, and $x \in \{0,1\}^n$ that is ε-far from $\mathcal{L}_n$ and for every computationally unbounded (cheating) prover $\mathcal{P}^*$ it holds that*

$$\Pr\left[(\mathcal{V}^x, \mathcal{P}^*)(n, \varepsilon) = 0\right] \geq 2/3.$$

   where the probability is over the coin tosses of $\mathcal{V}$.

If the completeness condition holds with probability 1, then we say that the $\mathcal{IPP}$ has a one-sided error *and otherwise the $\mathcal{IPP}$ is said to have a* two-sided error*. If all of the verifier's messages are uniformly distributed and independent random strings then the $\mathcal{IPP}$ is said to be* public-coin*.*

An $\mathcal{IPP}$ for $\mathcal{L} = \cup_{n\in\mathbb{N}}\mathcal{L}_n$ is said to have query complexity $q : \mathbb{N} \times \mathbb{R}^+ \to \mathbb{N}$ if, for every[14] $n \in \mathbb{N}$, $\varepsilon > 0$ and $x \in \mathcal{L}_n$, the verifier $\mathcal{V}$ makes at most $q(n, \varepsilon)$ queries to x when interacting with $\mathcal{P}$. The $\mathcal{IPP}$ is said to have communication complexity $c : \mathbb{N}\times\mathbb{R}^+ \to \mathbb{N}$ if, for every $n \in \mathbb{N}$, $\varepsilon > 0$ and $x \in \mathcal{L}_n$, the communication between $\mathcal{V}$ and $\mathcal{P}$ consists of at most $c(|x|, \varepsilon)$ bits. A round of communication consists of a single message sent from the verifier to the prover followed by a single message sent from the prover to the verifier. The $\mathcal{IPP}$ is said to have r rounds (sometimes called an r-round $\mathcal{IPP}$), for $r : \mathbb{N} \times \mathbb{R}^+ \to \mathbb{N}$ if, for every $n \in \mathbb{N}$, $\varepsilon > 0$ and $x \in \mathcal{L}_n$, if the number of rounds in the interaction between $\mathcal{V}$ and $\mathcal{P}$ on input x is at most $r(|x|, \varepsilon)$.

The standard definition of a property tester may be derived from Definition 2.1 by restricting the communication complexity to 0. The definition of an $\mathcal{MAP}$ can be derived by restricting the communication to be only from the prover to the verifier (see [GR15] for further details on $\mathcal{MAP}$s).

Non-uniform $\mathcal{IPP}$s. While Definition 2.1 refers to a *uniform* definition of $\mathcal{IPP}$, throughout this work it will be convenient for us to use a *non-uniform* definition. That is, we fix an integer $n \in \mathbb{N}$, which we think of as a variable parameter, and restrict Definition 2.1 to inputs of length n. Hence, unless stated otherwise, by an $\mathcal{IPP}$ we actually mean the *non-uniform* variant. Despite the fact that the integer n is fixed, we view it as a generic parameter and allow ourselves to write asymptotic expressions such as $O(n)$. We also note that while our results are proved in terms of non-uniform $\mathcal{IPP}$, they can be extended to the uniform setting in a straightforward manner.

2.1.2 Proximity Oblivious $\mathcal{IPP}$

Extending the notion of proximity oblivious testers [GR11], we define a *proximity oblivious* $\mathcal{IPP}$, as a variant of an $\mathcal{IPP}$ in which neither party receives the proximity parameter as input and for every $\varepsilon > 0$, the verifier is required to reject inputs that are ε-far from the language with some probability $\rho(\varepsilon)$.

Definition 2.2 (Proximity Oblivious $\mathcal{IPP}$)**.** *Let $\rho : (0, 1] \to (0, 1]$ be a monotone function. A* proximity oblivious $\mathcal{IPP}$ with detection probability ρ*, for the language $\mathcal{L} = \cup_{n\in\mathbb{N}}\mathcal{L}_n$ is similar to Definition 2.1, except that the neither the verifier nor the prover[15] receive the proximity parameter as input, and the completeness and soundness conditions are modified as follows:*

1. **Completeness:** *For every $n \in \mathbb{N}$, and $x \in \mathcal{L}_n$ it holds that*[16]

$$\Pr\left[(\mathcal{V}^x, \mathcal{P}(x))(n) = 1\right] = 1.$$

2. **Soundness:** *For every $n \in \mathbb{N}$, every $x \in \Sigma^n$, and for every computationally unbounded (cheating) prover $\mathcal{P}^*$ it holds that*

$$\Pr\left[(\mathcal{V}^x, \mathcal{P}^*)(n) = 0\right] \geq \rho(\Delta(x, \mathcal{L}_n))$$

[14] We measure the resources used in the protocol only when the verifier interacts with the *honest* prover. However, in the general case, the verifier can simply halt once one of its resources exceeds the corresponding bound (since it knows that it must be interacting with a *cheating* prover).

[15] Since we do not bound the computational resources of the prover, it can simply deduce the proximity parameter from the input.

[16] Note that we require the verifier to accept inputs $x \in \mathcal{L}$ with probability 1. A more general definition could allow this probability to be some smaller constant or even a function of ε (see [GS12]). For simplicity (and since it suffices for our purposes), in this work we only consider proximity oblivious $\mathcal{IPP}$s with perfect completeness.

In both condiditons the probability is over the coin tosses of $\mathcal{V}$.

Note that any proximity oblivious $\mathcal{IPP}$ with detection probability $\rho(\cdot)$, can be transformed into a standard $\mathcal{IPP}$ (as in Definition 2.1) by repeating the proximity oblivious $\mathcal{IPP}$ $O(1/\rho(\varepsilon))$ times in parallel (see Appendix A for details on parallel repetition for $\mathcal{IPP}$s).

2.2 Read-Once Branching Programs (ROBPs)

In this section we provide the necessary background on ROBPs (needed only for Section 3). An ROBP is defined as follows

Definition 2.3 (ROBP)**.** *A* branching program *on n variables is a directed acyclic graph that has a unique source vertex with in-degree 0 and (possibly) multiple sink vertices with out-degree 0. Each sink vertex is labeled either with 0 (i.e.,* reject*) or 1 (i.e.,* accept*). Each non-sink vertex is labeled by an index $i \in [n]$ and has exactly 2 outgoing edges, which are labeled by 0 and 1.*

The output of the branching program B on input $x \in \{0,1\}^n$, denoted $B(x)$, is the label of the sink vertex reached by taking a walk, starting at the source vertex such that at every vertex labeled by $i \in [n]$, the step taken is on the edge labeled by x_i.

The branching program is said to be read-once *(or ROBP for short) if along every path from source to sink, every index $i \in [n]$ appears at most once. The* size *of a branching program B, denoted $|B|$, is the number of vertices in its graph.*

Let φ and ψ be vertices in the branching program B on n variables and let $x \in \{0,1\}^n$. Loosely speaking, we write $\varphi \overset{x,k}{\rightsquigarrow} \psi$ if the walk of length k corresponding to x that starts at φ ends at ψ. Note that only k coordinates of x are read (adaptively) in this walk and that φ itself only determines the first variable read. Formally, we write $\varphi \overset{x,1}{\rightsquigarrow} \psi$ if the edge (φ, ψ) appears in B and is labeled by x_i, where i is the label of φ, and we (inductively) write $\varphi \overset{x,k}{\rightsquigarrow} \psi$ if there exists a vertex ζ in B such that $\varphi \overset{x,1}{\rightsquigarrow} \zeta$ and $\zeta \overset{x,k-1}{\rightsquigarrow} \psi$.

2.3 Context-Free Languages

In this section we provide the necessary background on context-free languages (needed only for Section 4). To define *context-free languages*, we first define context-free grammars (see [HMU06] for more details).

Definition 2.4 (Context-free grammar)**.** *A* context-free grammar *is a tuple $G = (V, \Sigma, R, A_{\mathsf{start}})$ such that V is a (finite) set of symbols, referred to as* variables*; Σ is a (finite) set of symbols, referred to as* terminals*; $R \subseteq V \times (V \cup \Sigma)^*$ is a (finite) relation, where each $(A, \alpha) \in R$ is referred to as a* production rule *and is denoted by $A \to \alpha$; $A_{\mathsf{start}} \in V$ is a variable that is referred to as the* start variable*.*

Let $G = (V, \Sigma, R, A_{\mathsf{start}})$ be a context-free grammar, and let $\alpha, \beta \in (V \cup \Sigma)^*$ be strings of terminals and variables. We say that α directly yields β, denoted by $\alpha \Rightarrow \beta$, if there exists a production rule $A \to \gamma$ in R such that β is obtained from α by replacing exactly one occurrence of the variable A in α with the string $\gamma \in (V \cup \Sigma)^*$. We say that α yields β, denoted $\alpha \overset{*}{\Rightarrow} \beta$ if there exists a finite sequence of strings $\alpha_0, \ldots, \alpha_k \in (V \cup \Sigma)^*$ such that $\alpha_0 = \alpha$, $\alpha_k = \beta$, and $\alpha_0 \Rightarrow \ldots \Rightarrow \alpha_k$. The language $\mathcal{L}_n \subseteq \Sigma^n$ is a context-free language if there exists a grammar

$G = (V, \Sigma, R, A_{\mathsf{start}})$ such that $\mathcal{L}_n = \{x \in \Sigma^n : A_{\mathsf{start}} \stackrel{*}{\Rightarrow} x\}$, where the derivation is with respect to the rules in R.

Derivation Tree. Let $G = (V, \Sigma, R, A_{\mathsf{start}})$ be a context-free grammar. For $A \in V$ and $x \in \Sigma^*$, a derivation tree, corresponding to the derivation $A \stackrel{*}{\Rightarrow} x$, is a rooted, directed, ordered, and labeled tree T (with edges oriented away from the root) that satisfies the following properties:

- Each internal vertex is labeled by some variable, and the root is labeled by the variable A.
- Each leaf is labeled by a terminal symbol, where the i^{th} leaf is labeled by the i^{th} symbol of x.
- For every internal vertex v, if v is labeled by the variable $A' \in V$ and for every $i \in [k]$ (where k is the number of children of v) the i^{th} child of v is labeled by $\alpha_i \in V \cup \Sigma$, then the production rule $A' \to \alpha_0 \circ \cdots \circ \alpha_k$ must belong to R.

For every derivation $A \stackrel{*}{\Rightarrow} x$ there exists a corresponding derivation tree.

Trees and the Lewis-Stearns-Hartmanis Lemma. In this work we only consider trees that are rooted, directed, and ordered (e.g., derivation trees as above). Thus, throughout this work, whenever we say tree we mean a rooted, directed, and ordered tree (with edges oriented away from the root). Note that the fact that the tree is ordered induces an ordering of its leaves. We define the arity of a tree to be the maximal number of children of any vertex in the tree. We follow the data-structure literature and define a subtree of a tree T as a tree consisting of a node in T and all of its descendants in T.[17]

For a tree T, we denote by $L(T)$ the number of leaves of T. We will use the following straightforward generalization of the Lewis-Stearns-Hartmanis Lemma [LSH65]:

Lemma 2.5. *Let T be a tree with arity d and let $t \in [L(T)]$. Then, there exists a subtree T' of T with $L(T') \in [t/d, t]$ leaves.*

The Lewis-Stearns-Hartmanis lemma corresponds to the special case of Lemma 2.5 in which $d = 2$ (i.e., a binary tree) and $t = 2n/3$.

Proof of Lemma 2.5. We prove by induction on the *size* of the tree (not on the number of leaves), noting that the base case holds trivially. Suppose that the lemma holds for all trees of size less than n. Let T be a tree of size n and let $t \in [L(T)]$.

If $t = L(T)$, then we are done (since $L(T) \in [t/d, t]$ and so T itself has the desired property). Otherwise, if $t < L(T)$, then T has a subtree T' (rooted at one of its children) such that $L(T') \geq t/d$ (since otherwise T has a total of less than $d \cdot t/d = t < L(T)$ leaves). If $L(T') \leq t$, then we are also done (since $L(T') \in [t/d, t]$ and so T' satisfies the desired property). Otherwise, $t \in [L(T')]$ and the lemma follows by applying the induction hypothesis on T'. □

3 $\mathcal{MAP}$s and $\mathcal{IPP}$s for Read-Once Branching Programs

In this section we prove Theorem 1.4 (and Theorem 1.2 will follow as a special case of the proof) by constructing an $\mathcal{IPP}$ for every language that is accepted by an ROBP.

[17] This definition differs from the graph-theoretic definition that defines a subtree as any connected subgraph of a tree. For example, the root of a tree is a subtree in the graph theoretic sense but not according to our definition (unless the tree has exactly one vertex).

3.1 $\mathcal{IPP}$s for ROBPs

Before presenting the $\mathcal{IPP}$ formally, we provide a short overview of the protocol (for a more detailed overview, see Section 1.2.1). Let B be an ROBP on n variables. We construct an r-round public-coin $\mathcal{IPP}$ for the language that is accepted by B. The $\mathcal{IPP}$ runs recursively, where each round of communication is as follows. Given an input x that is accepted by B within $n' \leq n$ steps, the prover finds the accepting path $\varphi_0 \to \varphi_1 \to \cdots \to \varphi_{n'}$ and sends to the verifier the subsequence $\varphi_{n'/k}, \ldots, \varphi_{i \cdot n'/k}, \ldots, \varphi_{n'}$ that contains every $(n'/k)^{\text{th}}$ vertex along this path. The verifier checks that every two consecutive vertices in the prover's message are connected (this can be done *without* making queries to the input x), then selects uniformly at random $i \in [k]$, sends i to the prover, and defines a new ROBP B_i that has the same graph as B, but its source vertex is $\varphi_{(i-1) \cdot n'/k}$ and its *unique* accepting sink is $\varphi_{i \cdot n'/k}$.[18] Subsequently, both parties (recursively) invoke the $\mathcal{IPP}$ protocol on input x and the ROBP B_i.

A crucial point is that although the input x does not not shrink in each recursive call, the *effective length of the input*, n', which is the alleged number of bits of x that need to be read in order to get from the source to the accepting sink, does shrink (by a factor of k). Hence, after r such rounds, the verifier can read all bits of x that are required to verify the current statement (which refers to a path of length n/k^r). Interestingly, and in contrast to the simpler case of OBDDs, in this last step the verifier reads the n/k^r bits of the input *adaptively*, based on the steps taken by the ROBP.

In the $\mathcal{IPP}$ that we construct, we assume for simplicity that the verifier is given an integer $n' \leq n$, and the claim (which the verifier is trying to validate) is that $B(x) = 1$ after reading *exactly* n' bits of the input x. Furthermore, we assume that the ROBP B is such that there exists *some* accepting path (i.e., from the source to some accepting sink) of length n'. We can reduce the general case to this restricted setting by having the prover send n' as part of its first message and having the verifier explicitly check that there is some accepting path of length n' (this check requires no queries to the main input x).

Recall that, as noted in Section 1.2, to facilitate the recursion we use the notion of a *proximity oblivious* $\mathcal{IPP}$ (see Section 2.1.2). When handling general ROBPs (rather than OBDDs), we follow this approach in *spirit* but, due to technical reasons, the construction will not exactly fit Definition 2.2. More specifically, since in each step of the recursion only the *effective* length of the input shrinks (but the actual length of the input stays the same), throughout the proof it will be more convenient for us to use *absolute* distances, denoted by $\overline{\Delta}$ (see Section 2) rather than with distances that are relative to n. Hence, the detection probability $\overline{\rho}$ of the verifier will be a function of the *absolute distance* (rather than of the relative distance) of the input from the language. That is, we will construct an absolute proximity oblivious $\mathcal{IPP}$ with detection probability $\{\overline{\rho}_n : \{0, \ldots, n\} \to (0, 1]\}_{n \in \mathbb{N}}$, which is the same as Definition 2.2 except that we modify the soundness condition as follows:

- ***Soundness:*** For every $n \in \mathbb{N}$, $x \in \Sigma^n$, and for every computationally unbounded (cheating) prover $\mathcal{P}^*$ it holds that

$$\Pr\left[(\mathcal{V}^x, \mathcal{P}^*)(n) = 0\right] \geq \overline{\rho}_n\big(\overline{\Delta}\left(x, \mathcal{L}_n\right)\big) \tag{3.1}$$

[18] One could alternatively define B_i to consist only of vertices at distance at most n'/k from $\varphi_{(i-1) \cdot n'/k}$. We refrain from taking this approach due to technical reasons. Note that also when using this alternate definition, when considering general ROBPs (rather than OBDDs), B_i could potentially look at all of the n bits of the input (see discussion at the end of Section 1.2.1).

Again, we can transform an *absolute* proximity oblivious $\mathcal{IPP}$ with detection probability $\{\overline{\rho}_n : \{0,\ldots,n\} \to (0,1]\}_{n\in\mathbb{N}}$ into a standard $\mathcal{IPP}$ (as in Definition 2.1) by repeating the base protocol $O(1/\overline{\rho}_n(\varepsilon \cdot n))$ times in parallel.

The absolute proximity oblivious $\mathcal{IPP}$ for ROBPs, denoted ROBP-IPP, is presented in Fig. 1 (recall that the notation $\varphi \overset{x,m}{\rightsquigarrow} \psi$, which is used in Fig. 1 means that, given input x, the ROBP walks from φ to ψ in m steps, see Section 2.2 for details).

It can be easily verified that the round complexity is r, the communication complexity is $O(rk \cdot \log(|B|))$ and the query complexity is $O(n/k^r)$. We proceed to show that completeness and soundness hold.

Completeness. Let B be a ROBP on n variables, let $r \geq 0$, $n' \in [n]$, and let $x \in \{0,1\}^n$ such that $B(x) = 1$ after reading exactly n' bits of the input. (Perfect) completeness follows by induction on r as follows.

For $r = 0$, the verifier just reads the appropriate n' bits of the input and accepts with probability 1. For $r \geq 1$, let $(\varphi_0, \varphi_1, \ldots, \varphi_{n'})$ be the accepting path corresponding to x. The checks that the verifier performs in the current round pass, since $\varphi_{n'}$ is indeed an accepting sink and $\varphi_{(i-1)\cdot n'/k^r} \overset{x,n'/k^r}{\rightsquigarrow} \varphi_{i\cdot n'/k^r}$, for every $i \in [p^r]$. Furthermore, since for every choice of $i \in [k]$ (made by the verifier) it holds that $\varphi_{(i-1)\cdot n'/k} \overset{x,n'/k}{\rightsquigarrow} \varphi_{i\cdot n'/k}$, the two parties recursively run the $r-1$ round protocol on a branching program B_i such that $B_i(x) = 1$ after reading exactly n'/k bits of x. Hence, by the inductive hypothesis the verifier accepts with probability 1.

Soundness. Soundness follows directly from the following lemma, which is proved by induction on the number of rounds r. We suggest to the reader to first consider the case that $n' = n$ in both the lemma statement and its proof. Nevertheless, we stress that in lower levels of the recursion, the parameter n' becomes much smaller than n.

Lemma 3.1. *Let $n \in \mathbb{N}$, $n' \in [n]$ and $r \geq 0$. For every ROBP B of size s on n variables that has an accepting path of length n', every x that is in* absolute *distance $\varepsilon \cdot n'$ from $\{z \in \{0,1\}^n : B(z) = 1\}$, and for every cheating prover strategy $\mathcal{P}^*$ it holds that:*

$$\Pr[(\mathcal{V}^x, \mathcal{P}^*)(n, n', B, r) = 0] \geq \varepsilon,$$

where $\mathcal{V}$ is the r-round verifier of ROBP-IPP *(of Fig. 1).*

Proof. We prove Lemma 3.1 by induction on the number of rounds $r \geq 0$. In the base case, corresponding to $r = 0$, the verifier simply ignores the prover and reads the appropriate n' bits of x. Hence, if $B(x) \neq 1$, then the verifier rejects with probability 1.[19]

In the inductive step, for $r \geq 1$, let $x \in \{0,1\}^n$ that is at absolute distance $\varepsilon \cdot n'$ from $\{z \in \{0,1\}^n : B(z) = 1\}$ and let P^* be a (deterministic) cheating prover strategy for the protocol ROBP-IPP of Fig. 1 (with r rounds). Let $(\varphi_{n'/k}, \varphi_{2n'/k}, \ldots, \varphi_{n'})$ be the first message sent by P^* to $\mathcal{V}$ and let φ_0 be the source. Since the verifier explicitly checks these conditions, it must be the case that $\varphi_{n'}$ is an accepting sink, and that for every $i \in [k]$, there exists some

[19] In the trivial case that $\varepsilon = 0$ (i.e., $B(x) = 1$), the verifier also satisfies the requirement, since it rejects with probability at least $0 = \varepsilon$. It may also be worth mentioning that it always holds that $\varepsilon \leq 1$, since B has an accepting path of length n'.

The Protocol $\mathsf{ROBP\text{-}IPP}^{B}_{n,n',r}$:

Common Input: Integers $n, n' \in \mathbb{N}$, a ROBP B on n variables such that there exists some accepting path of length n' in B, and a parameter $r \in \mathbb{N}$.

Prover's Input: Direct access to $x \in \{0,1\}^n$ such that $B(x) = 1$ in exactly n' steps.

Verifier's Input: Oracle access to the same x.

1. If $r = 0$, the verifier $\mathcal{V}$ checks whether $B(x) = 1$ after exactly n' steps by (adaptively) reading the appropriate n' bits of x. If $B(x) = 1$, then $\mathcal{V}$ accepts, otherwise it rejects, and in either case both parties terminate the protocol.
2. The Prover $\mathcal{P}$:
   (a) Let $\varphi = (\varphi_0, \ldots, \varphi_{n'}) \in [|B|]^{n'}$ be the sequence of vertices in the accepting path (of length $n' \leq n$) in B that corresponds to the evaluation of B on input x.
   (b) Send $(\varphi_{n'/k}, \varphi_{2n'/k}, \ldots, \varphi_{n'})$ to $\mathcal{V}$.[a]
3. The Verifier $\mathcal{V}$:
   (a) Check that $\varphi_{n'}$ is an accepting sink of B.
   (b) Let φ_0 be the source of B.
   (c) For every $i \in [k]$, check that there exists some input $x^{(i)} \in \{0,1\}^n$ such that $\varphi_{(i-1)\cdot n'/k} \xrightsquigarrow{x^{(i)}, n'/k} \varphi_{i \cdot n'/k}$.[b]
   (d) Select uniformly at random an index i in $[k]$, and send i to $\mathcal{P}$.
4. Denote by B_i the ROBP that has the same graph as B, except that its source is $\varphi_{(i-1)\cdot n'/k}$, and its *unique* accepting vertex is $\varphi_{i \cdot n'/k}$.
5. Both parties (recursively) invoke $\mathsf{ROBP\text{-}IPP}^{B_i}_{n,n'',r-1}$, where $n'' = n'/k$, on input x.

[a] The accepting sink $\varphi_{n'}$ is also sent since (in the first step of the recursion) there could be multiple accepting sinks.

[b] This check is performed without making any queries to the main input x. Although our focus is not on *computational* complexity, we note that this can be done in $\mathsf{poly}(s)$ time.

Figure 1: $\mathcal{IPP}$ for ROBPs

$x^{(i)} \in \{0,1\}^n$ such that $\varphi_{(i-1)\cdot n'/k} \overset{x^{(i)},n'/k}{\rightsquigarrow} \varphi_{i\cdot n'/k}$. Furthermore, for every $i \in [k]$, we assume without loss of generality that $x^{(i)}$ is the string z that minimizes the distance of x to the set $\left\{z \in \{0,1\}^n \,:\, \varphi_{(i-1)\cdot n'/k} \overset{z,n'/k}{\rightsquigarrow} \varphi_{i\cdot n'/k}\right\}$.

Recall that $\overline{\Delta}$ denotes absolute distance (see Section 2) and let $\varepsilon_i \overset{\text{def}}{=} \frac{\overline{\Delta}(x,x^{(i)})}{n'/k}$. The following claim, which crucially uses the fact that B is *read-once*, shows that the average of the ε_i's (which will later be shown to lower bound the rejection probability of $\mathcal{V}$) is at least ε.

Claim 3.1.1. $\mathbf{E}_i[\varepsilon_i] \geq \varepsilon$.

Proof. For every $i \in [k]$, let $J_i \subseteq [n]$ be the set of n'/k variables that are read when going from $\varphi_{(i-1)\cdot n'/k}$ to $\varphi_{i\cdot n'/k}$ on input $x^{(i)}$. We first prove that the sets J_i's are disjoint. Assume otherwise; that is, that there exists $j \in J_{i_1} \cap J_{i_2}$ for some $i_1, i_2 \in [k]$. For every $i \in [k]$, denote the path from $\varphi_{(i-1)\cdot n'/k}$ to $\varphi_{i\cdot n'/k}$ (in B) on input $x^{(i)}$ by P_i. Consider the concatenated path $P_1 \circ \cdots \circ P_k$. This is a path in B in which the label j appears twice (both in P_{i_1} and in P_{i_2}) in contradiction to our assumption that B is a *read-once* branching program.

Define $x' \in \{0,1\}^n$ as follows. For every $j \in [n]$, if $j \in J_i$ for some $i \in [k]$ (which must be unique as just shown), then $x'[j] \overset{\text{def}}{=} x^{(i)}[j]$, and otherwise (i.e., if $j \notin J_1 \cup \cdots \cup J_p$) we set $x'[j] \overset{\text{def}}{=} x[j]$. Note that

$$\varphi_0 \overset{x',n'/k}{\rightsquigarrow} \varphi_{n'/k} \overset{x',n'/k}{\rightsquigarrow} \cdots \overset{x',n'/k}{\rightsquigarrow} \varphi_{n'}$$

and therefore $B(x') = 1$. The claim follows by noting that

$$\varepsilon \cdot n' \leq \overline{\Delta}\left(x, x'\right) = \sum_{i\in[k]} \overline{\Delta}\left(x[J_i], x'[J_i]\right) = \sum_{i\in[k]} \overline{\Delta}\left(x[J_i], x^{(i)}[J_i]\right) \leq \sum_{i\in[k]} \overline{\Delta}\left(x, x^{(i)}\right) = \sum_{i\in[k]} \varepsilon_i \cdot n'/k,$$

where the first inequality follows from the fact that x is in absolute distance $\varepsilon \cdot n'$ from $\{z \in \{0,1\}^n \,:\, B(z) = 1\}$ combined with the fact that $B(x') = 1$, and the first and second equality follow from the definition of x' (the first equality also uses the fact that the J_i's are disjoint). The claim follows. □

For every $i \in [k]$, let B_i be the ROBP that has the same graph as B, but its source is $\varphi_{(i-1)\cdot n'/k}$, and its *unique* accepting vertex is $\varphi_{i\cdot n'/k}$. Let P_i^* be the residual $r-1$ round strategy of P^* after receiving the message i from $\mathcal{V}$ in the first round, and let $\mathcal{V}_i$ be the residual strategy of $\mathcal{V}$ after fixing its first message to i. Note that $\mathcal{V}_i$ is exactly the strategy of the verifier in $\mathsf{ROBP\text{-}IPP}^{B_i}_{n,n',r-1}$.

Claim 3.1.2. *For every $i \in [k]$, it holds that*

$$\Pr[(\mathcal{V}_i^x, P_i^*)(n, n'', B_i, r-1) = 0] \geq \varepsilon_i,$$

where $n'' = n'/k$.

Proof. Let $i \in [k]$. Recall that $x^{(i)}$ was chosen as $z \in \{0,1\}^n$ that minimizes the distance of x to the set $S_i \overset{\text{def}}{=} \{z \in \{0,1\}^n \,:\, B_i(z) = 1 \text{ using exactly } n'/k \text{ steps}\}$. Hence,

$$\overline{\Delta}\left(x, S_i\right) = \overline{\Delta}\left(x, x^{(i)}\right) = \varepsilon_i \cdot n'/k.$$

Hence, $(\mathcal{V}_i^x, P_i^*)(n, n'', B_i, r-1)$ corresponds to an invocation of the $r-1$ round version of the protocol on an input x that is in absolute distance $\varepsilon_i \cdot n'/k$ from S_i. Therefore, by the inductive hypothesis, the verifier $\mathcal{V}_i$ rejects with probability at least ε_i. □

Using Claim 3.1.1 and Claim 3.1.2 we obtain that

$$\Pr[(\mathcal{V}^x, P^*)(n, n', B, r) = 0] = \mathop{\mathbf{E}}_{i \in [k]} \Big[\Pr[(\mathcal{V}_i^x, P_i^*)(n, n'/k, B_i, r-1) = 0] \Big] \geq \mathop{\mathbf{E}}_{i \in [k]} [\varepsilon_i] \geq \varepsilon, \tag{3.2}$$

and the lemma follows. □

This concludes the proof of Theorem 1.4.

Remark 3.2 (Computational Complexity)**.** *The running time of the $\mathcal{IPP}$ prover in Fig. 1 is polynomial in its input (i.e.,* $\mathsf{poly}(|B|, n, k, r, 1/\varepsilon)$*). As for the $\mathcal{IPP}$ verifier, if the representation of the ROBP B allows one to check if two vertices in the graph of B are connected in* $\mathsf{polylog}(|B|)$ *time, then the verifier runs in time* $\mathsf{poly}(\log n, k, r, \log(|B|), 1/\varepsilon)$*. If such a representation is not available, then it can be generated in a relatively expensive (i.e.,* $\mathsf{poly}(|B|)$ *time) pre-processing step, which does not depend on the input x and can be re-used for multiple inputs.*

Alternatively, for some other natural representations, the verifier can employ the prover to efficiently check if two vertices in B are connected. Consider for example a natural representation in which there exists a polynomial (i.e., $\mathsf{poly}(\log(|B|))$*) size circuit C that on input a vertex v (in the graph of B) and a bit $\sigma \in \{0,1\}$ outputs the neighbor u of v that σ leads to (i.e., the edge (v,u) is labeled by σ). Suppose further that C is $O(\log(|B|)$-space uniform (i.e., can be generated by an $O(\log(|B|))$-space Turing machine). In such case we can use the prover to check connectivity, as described next, and so we obtain sub-linear verification.*

In order to verify connectivity efficiently, we first observe that there exists an ($O(\log(|B|))$-space uniform) circuit, of $\mathsf{polylog}(|B|)$*-depth and* $\mathsf{poly}(|B|)$*-size, that on input two vertices v and u outputs 1 if and only if they are connected (possibly via a long path).*[20] *Now we can apply the efficient interactive proof-system for low-depth computation*[21] *of Goldwasser et al. [GKR08, Theorem 1] to obtain an interactive proof-system that verifies that two given vertices are connected, where the verifier runs in time* $\mathsf{polylog}(|B|)$ *and the prover runs in time* $\mathsf{poly}(|B|)$*.*[22] *We note that employing this proof-system inside our $\mathcal{IPP}$ increases the round complexity of the $\mathcal{IPP}$ by a* $\mathsf{polylog}(|B|)$ *factor.*

Remark 3.3 ($\mathcal{IPP}$s for Ordered Binary Decision Diagrams)**.** *Recall that an ordered binary decision diagram (OBDD) is an ROBP that is both* layered *and* ordered *(see Section 1.2.1). We observe that the communication complexity in Theorem 1.4 can be slightly improved for OBDDs of width w and size $s = O(nw)$ from $O((pr \log s) \cdot \varepsilon^{-1})$ to $O((pr \log w) \cdot \varepsilon^{-1})$, by noting that the i^{th} vertex specified*

[20] The circuit first uses C to generate the entire adjacency matrix of B and then checks whether v and u are connected by repeated squaring of the adjacency matrix. Note that all actions can be implemented in $\mathsf{polylog}(|B|)$-depth and $\mathsf{poly}(|B|)$-size.

[21] Goldwasser *et al.* show that any language that is accepted by a ($O(\log(S(n)))$-space uniform) circuit of depth $D(n)$ and size $S(n)$, has an interactive proof-system, where the verifier runs in time $(n + D(n)) \cdot \mathsf{polylog}(S(n))$ and the prover runs in time $\mathsf{poly}(S(n))$.

[22] We stress that the interactive proof-system for verifying connectivity is a standard interactive proof-system and not a "proof of proximity" (i.e., not an $\mathcal{IPP}$). Indeed, this is crucial for our application since we use the interactive proof-system for connectivity as a subroutine within our $\mathcal{IPP}$, and the $\mathcal{IPP}$ verifier should reject if at any point it encounters a pair of vertices that are disconnected (even if the pair is "close" to being connected).

by the prover (say, in the first round) must *be in layer $i \cdot n/p$ and therefore it can be specified using only $\log_2 w$ bits.*

3.2 $\mathcal{MAP}$s for ROBPs

We observe that Theorem 1.2 follows almost directly from the proof of Theorem 1.4, when restricted to the case $r = 1$. Indeed, the only two gaps (which are easily resolved) are:

1. Interaction: Theorem 1.4 (restricted to $r = 1$) guarantees a 1-round $\mathcal{IPP}$ for languages recoginzed by ROBPs. In general, a 1-round $\mathcal{IPP}$ is not necessarily an $\mathcal{MAP}$, since it may include a message sent from the verifier to the prover. Nevertheless, the order of the messages in our protocol is such that first the prover sends a message to the verifier and then the verifier responds. The last message can clearly be avoided and so we obtain an $\mathcal{MAP}$.

2. Dependence on the Proximity Parameter in the Proof Length: Recall that there is a linear dependence on $1/\varepsilon$ in the communication complexity in Theorem 1.4, due to the $O(1/\varepsilon)$ parallel repetitions that were used. However, for $\mathcal{MAP}$s, parallel repetition can be performed without increasing the proof length, since the proof is a deterministic function of the input. Hence, we can save the additional $O(1/\varepsilon)$ factor that is used for general $\mathcal{IPP}$s.

3.3 $\mathcal{MAP}$s and $\mathcal{IPP}$s for Affine Spaces

In this section, as an example, we show how Theorems 1.2 and 1.4 can yield $\mathcal{MAP}$s and $\mathcal{IPP}$s for any *affine space*.

Before proceeding to the proof, we remark that Rothblum *et al.* [RVW13] identified a specific affine space, called PVAL, as being "complete" for the construction of $\mathcal{IPP}$s for the class $\mathcal{NC}$.[23] They constructed an $\mathcal{IPP}$ for PVAL and thereby obtain $\mathcal{IPP}$s for all of $\mathcal{NC}$. Interestingly, PVAL is an affine space and so the results of this section yield an alternative $\mathcal{IPP}$ for it. Unfortunately though, the parameters obtained by our $\mathcal{IPP}$ are inferior[24] to those of [RVW13] and do not yield an alternative $\mathcal{IPP}$ for $\mathcal{NC}$.

Definition 3.4. *Let $\mathbb{F}$ be a finite field, $n \in \mathbb{N}$ and $t \in [n]$. An* affine subspace *of the vector space $\mathbb{F}^n$, denoted $\mathsf{AffineSpace}_{A,b}$, is parametrized by a matrix $A \in \mathbb{F}^{t \times n}$ and a vector $b \in \mathbb{F}^t$ and consists of all strings $x \in \mathbb{F}^n$ such that $Ax = b$.*

Our construction of an $\mathcal{IPP}$ for every affine space follows directly from Theorem 1.4 by showing that membership in an affine subspaces can be recognized by a small-width OBDD.

Proposition 3.5. *Let $\mathbb{F}$ be a finite field, $n \in \mathbb{N}$ and $t \in [n]$. For every $A \in \mathbb{F}^{t \times n}$ and $b \in \mathbb{F}^t$, there exists a width $|\mathbb{F}|^{O(t)}$ OBDD that accepts $\mathsf{AffineSpace}_{A,b}$.*

[23] The language PVAL is parameterized by a sequence of points in a finite vector space and a sequence of values, and consists of all strings x whose low degree extension $\mathsf{LDE}(x)$ is equal to the given sequence of values at the corresponding sequence of points

[24] More specifically, for a PVAL instance parameterized by t points, the communication complexity in our protocol is $O(t \cdot 1/\varepsilon \cdot \mathsf{polylog}(n))$, whereas in [RVW13] it is $O(t \cdot (1/\varepsilon)^{o(1)} \cdot \mathsf{polylog}(n))$. Our result is insufficient since in the context of the proof of $\mathcal{IPP}$s for $\mathcal{NC}$, $t = \sqrt{n}$ and $\varepsilon = 1/\sqrt{n}$.

Proof. We describe a deterministic streaming algorithm for deciding membership in $\mathsf{AffineSpace}_{A,b}$. The algorithm gets access to a stream of n fields elements, reads the input element-by-element (in order) and stores a total of t field elements at any given time. Transforming the latter into an OBDD, as required, is straightforward.[25]

Denote the columns of A by $a_1, \ldots, a_n \in \mathbb{F}^t$. The algorithm maintains a vector $c \in \mathbb{F}^t$ which is initialized to 0. The streaming algorithm reads the input $x \in \mathbb{F}^n$ element-by-element and after reading the i^{th} element, the algorithm sets $c \leftarrow c + x_i a_i$ (where the addition is over $\mathbb{F}$). In the end, it holds that

$$c = \sum_{i=1}^{n} x_i a_i = Ax$$

and therefore it suffices for the algorithm to accept if $c = b$ and reject otherwise.

□

By applying Theorem 1.4, we obtain the following corollary.

Corollary 3.6. *Let $\mathbb{F}$ be a finite field, $n \in \mathbb{N}$ and $t \in [n]$. For every $A \in \mathbb{F}^{t \times n}$, $b \in \mathbb{F}^t$ and for every $k = k(n) \geq 2$ and $r = r(n) \geq 1$ such that $k^r \leq n$, there exists an r-round $\mathcal{IPP}$ for $\mathsf{AffineSpace}_{A,b}$ with communication complexity $O((rk \cdot t \log |\mathbb{F}|) \cdot \varepsilon^{-1})$ and query complexity $O\left(\frac{n}{k^r} \cdot \varepsilon^{-1}\right)$. Furthermore, the $\mathcal{IPP}$ is public-coin and has one-sided error.*

4 $\mathcal{MAP}$s and $\mathcal{IPP}$s for Context-Free Languages

In this section we prove Theorem 1.3 by constructing an $\mathcal{IPP}$ for any context-free language. As noted in the introduction, the proof of Theorem 1.1 will follow as a special case of this $\mathcal{IPP}$.

The proof of Theorem 1.3 extensively uses the notions of a *partial derivation* and a *partial derivation language*. Recall that a partial derivation of a grammar G is a derivation, according to the production rules of G, in which not all variables are expanded. Our notion of a *partial derivation language* is more complex. In particular, it does *not* refer to the language that consist of all possible partial derivations of the grammar (i.e., $\{x \in (\Sigma \cup V)^* : A_{\mathsf{start}} \stackrel{*}{\Rightarrow} x\}$). Rather, we define a partial derivation language as a language that consists of the subsequence of *terminal symbols* that correspond to partial derivations that start at some fixed variable. Furthermore, we consider only partial derivations in which the subsequence of variables in the partial derivation occur in specific locations. More concretely, a partial derivation language is parameterized by (1) a start variable A_0; (2) the number of terminals m; (3) a sequence of ℓ locations $i_1, \ldots, i_\ell$; and (4) a corresponding sequence of variables $A_1, \ldots, A_\ell$. The language consists of strings z of length m such that the string $z' = z[1, i_1 - 1] \circ A_1 \circ z[i_1, i_2 - 1] \circ \cdots \circ A_\ell \circ z[i_\ell, m]$ can be derived from A_0. More formally,

Definition 4.1 (Partial Derivation Language)**.** *A* partial derivation language *of the grammar $G = (V, \Sigma, R, A_{\mathsf{start}})$ is a language $\mathcal{L} \subseteq \Sigma^m$, parameterized by indices $1 \leq i_1 \leq \ldots \leq i_\ell \leq m$ and variables $A_0, \ldots, A_\ell \in V$ such that*

$$\mathcal{L} \stackrel{\text{def}}{=} \left\{ z \in \Sigma^m \; : \; A_0 \stackrel{*}{\Rightarrow} z[1, i_1 - 1] \circ A_1 \circ z[i_1, i_2 - 1] \circ \cdots \circ A_\ell \circ z[i_\ell, m] \right\}.$$

[25] Loosely speaking, each layer of the OBDD will consist of the $2^{O(t \log |\mathbb{F}|)}$ possible configurations of the streaming algorithm (which include both its current state and possibly some of the bits of the element that is currently being read).

The concise description of a partial derivation language $\mathcal{L} \subseteq \Sigma^m$*, parameterized by* $\bar{i} = (i_1, \ldots, i_\ell)$ *and* $\overline{A} = (A_0, \ldots, A_\ell)$*, is denoted by* $\langle \mathcal{L} \rangle \stackrel{\text{def}}{=} (m, \bar{i}, \overline{A})$*.*

We stress that $z \in \mathcal{L}$, where $\mathcal{L}$ is a partial derivation language and $\langle \mathcal{L} \rangle = (m, (i_1, \ldots, i_\ell), (A_0, \ldots, A_\ell))$, means that z is a string of *terminal* symbols such that $A_0 \stackrel{*}{\Rightarrow} z'$, where z' is an interleaving of z and $A_1, \ldots, A_\ell$, in which A_j appears in coordinate $i_j + j - 1$. Indeed, there is a natural 1-1 correspondence between the indices $i_j \in [m]$ that are the locations in z in which the variables should be inserted, and the indices $i'_j \in [m + \ell]$, where $i'_j \stackrel{\text{def}}{=} i_j + j - 1$, that are the locations in the string $z' = z[1, i_1 - 1] \circ A_1 \circ z[i_1, i_2 - 1] \circ \cdots \circ A_\ell \circ z[i_\ell, m]$ in which the fixed variables appear.

Our construction of an $\mathcal{IPP}$ is recursive, and to facilitate the recursion, as discussed in Section 1.2.2, it will be useful for us to construct an $\mathcal{IPP}$ for *partial derivation languages* rather than just *context-free languages*. Additionally, as discussed in Section 1.2, the $\mathcal{IPP}$ will be proximity oblivious[26] (see Section 2.1.2). That is, we prove the following (more general) lemma:

Lemma 4.2. *Let* G *be a context-free grammar, let* $\mathcal{L}$ *be a* partial derivation language *corresponding to* G*, parameterized by* $\langle \mathcal{L} \rangle = (n, (i_1, \ldots, i_\ell), (A_0, \ldots, A_\ell))$*. For every integers* $k \geq 2$ *and* $r \geq 1$ *such that* $k^r \leq n$*, there exists an* r*-round proximity oblivious* $\mathcal{IPP}$ *for* $\mathcal{L}$ *with detection probability* $\rho(\varepsilon) = \varepsilon$*, communication complexity* $O(rk \log(n + \ell))$ *and query complexity* $O(\frac{n+\ell}{k^r})$*. Furthermore, the proximity oblivious* $\mathcal{IPP}$ *is public-coin.*

Theorem 1.3 follows directly from Lemma 4.2 by observing that (1) every context-free language is a partial derivation language, without any fixed variables (i.e., $\ell = 0$), and (2) we can transform any proximity oblivious $\mathcal{IPP}$ into a standard $\mathcal{IPP}$ (by repeating the former $O(1/\varepsilon)$ times in parallel).

Lemma 4.2 is proved in Sections 4.1 and 4.2. Specifically, in Section 4.1, which contains the more involved (and interesting) part of the proof, we show a scheme for partitioning partial derivation languages into several smaller partial derivation languages. Then, in Section 4.2 we use this partition to construct an $\mathcal{IPP}$ for partial derivation languages (which is a fairly straightforward implementation of the outline presented in Sections 1.2.2 and 1.2.3), as well as describe the steps required to derive an $\mathcal{MAP}$ (thereby proving Theorem 1.1). Finally, in Section 4.3 we show how to improve the efficiency of the foregoing $\mathcal{MAP}$ for the Dyck languages (i.e., the languages of balanced parentheses expressions).

4.1 Partitioning Partial Derivation Languages

Let $\mathcal{L} \subseteq \Sigma^n$ be a partial derivation language[27] of a context-free grammar $G = (V, \Sigma, R, A_{\mathsf{start}})$, parameterized by $\langle \mathcal{L} \rangle = (n, (i_1, \ldots, i_\ell), (A_0, \ldots, A_\ell))$, and let $d = O(1)$ be the length of the longest production rule in R (so that every $x \in \mathcal{L}$ has a derivation tree with arity at most d).

In this section we describe a technique for partitioning $\mathcal{L}$ into several partial derivation languages $\mathcal{L}_1, \ldots, \mathcal{L}_k$ (of shorter strings), while preserving distances. That is, inputs x that belongs to $\mathcal{L}$ will be partitioned into k parts such that for every $j \in [k]$, the j^{th} part of x belongs to $\mathcal{L}_j$, whereas, for inputs x that are far from $\mathcal{L}$, the j^{th} part of x will be far from $\mathcal{L}_j$, for an average j. Later, in Section 4.2, we use this partition to construct an $\mathcal{IPP}$ for $\mathcal{L}$. (See Sections 1.2.2 and 1.2.3 for a high-level overview.)

[26] In contrast to the case of ROBPS (see Section 3), here we can directly use Definition 2.2 without any modifications.

[27] We suggest to the reader to consider the case that $\mathcal{L}$ is a context-free language (i.e., no variables are fixed) at first reading, since it is somewhat simpler. However, we stress that we have to handle general partial derivation languages for the recursion to go through.

The partition, which will be constructed jointly by the $\mathcal{IPP}$ prover and verifier, has two different representations. The first representation, which we call the *interval representation*, is a concise representation that is generated by the prover and sent to the verifier. The advantage of this representation is has a simple *syntactic* structure. The second representation, which is the actual partition, will be derived by the verifier from the interval representation. The main advantage of the latter representation is that it facilitates the verification of the *semantic* relation of the partition to the main input x.

We begin by describing the procedure that is used to generate the *interval representation* of the partition. The procedure, called `Generate-Intervals`(x,t), is given as input $x \in \mathcal{L}$ (recall that $\mathcal{L}$ is parameterized by $\langle \mathcal{L} \rangle = (n,(i_1,\ldots,i_\ell),(A_0,\ldots,A_\ell))$ and a parameter $t \in [n']$, where $n' \stackrel{\text{def}}{=} n+\ell$ and t specifies the desired size of each part in the partition. We assume for simplicity that $t \geq 2d$, and the case that $t < 2d = O(1)$ will be handled separately (and trivially) in Section 4.2. First, the procedure constructs[28] a derivation tree T corresponding to the derivation $A_0 \stackrel{*}{\Rightarrow} x'$, where $x' \stackrel{\text{def}}{=} x[1,i_1-1] \circ A_1 \circ x[i_1,i_2-1] \circ \cdots \circ A_\ell \circ x[i_\ell,n]$ ($A_0 \stackrel{*}{\Rightarrow} x'$ follows from the fact that $x \in \mathcal{L}$). Next, using Lemma 2.5, the procedure finds $k = O(n'/t)$ rooted subtrees[29] $T_1,\ldots,T_k$ of T such that (1) every vertex of T belongs to at least one of the subtrees, and (2) for each $i<j$ either T_i and T_j are disjoint or T_i is a subtree of T_j. The procedure outputs $\overline{I} = (I_1,\ldots,I_k) \in ([n']^2)^k$ and $\overline{B} = (B_1,\ldots,B_k) \in V^k$ where B_j is the label of the root of T_j and $I_j \subseteq [n']$ is the minimal interval that contains all the leaves of T_j, for every $j \in [k]$. Each pair of intervals is either disjoint or contained in one other. The `Generate-Intervals` procedure is detailed in Fig. 2.

To see that `Generate-Intervals` halts with $k \leq \frac{n'}{t/d-1} \leq 2d \cdot \frac{n'}{t}$ intervals, observe that in each iteration the number of leaves of the tree T' (defined in Step 3a) decreases additively by at least $t/d-1$ and that we assumed that $t \geq 2d$.

As noted above, the output $(\overline{I},\overline{B})$ of `Generate-Intervals` is in the first representation of the partition, which we called the interval representation. Next, we show a transformation $\mathcal{T}$ (which will be applied by the $\mathcal{IPP}$ verifier) that transforms the interval representation of the partition into an actual partition of the main input x.

Actually, instead of partitioning the input x into parts $S_1,\ldots,S_k \subseteq [n]$, it will be more convenient to view the partition as a partition of the *terminal* coordinates of $x' = x[1,i_1-1] \circ A_1 \circ x[i_1,i_2-1] \circ \cdots \circ A_\ell \circ x[i_\ell,n]$.[30] That is, instead of a partition of $[n]$, we will find a partition of $[n']\backslash\{i'_1,\ldots,i'_\ell\}$, where $i'_j \stackrel{\text{def}}{=} i_j + j - 1$, for every $j \in [\ell]$ (indeed, the non-terminal coordinates of x' are precisely $\{i'_1,\ldots,i'_\ell\}$).

Our aim is to design a transformation $\mathcal{T}$ that maps $(\overline{I},\overline{B})$ into a partition $S_1,\ldots,S_k$ of $[n']\backslash\{i'_1,\ldots,i'_\ell\}$, where the parts have roughly the same length, together with (concise descriptions of) partial derivation languages $\mathcal{L}_1,\ldots,\mathcal{L}_k$ that satisfy the following conditions:

- **Completeness:** If $x \in \mathcal{L}$ and $(\overline{I},\overline{B})$ is the output of `Generate-Intervals`(x,t), then $x'[S_j] \in \mathcal{L}_j$, for every $j \in [k]$.
- **Soundness:** If x is ε-far from $\mathcal{L}$, then for *every* $(\overline{I},\overline{B}) \in ([n']^2)^k \times V^k$ it holds that $x'[S_j]$ is ε-far from $\mathcal{L}_j$, for an average $j \in [k]$ (where the average is weighted based on the lengths of

[28] Although our focus is not on computational complexity, we remark that such a derivation tree can be constructed in time $\mathsf{poly}(n')$, see [HMU06] for details.

[29] Recall that we define a *subtree* of a tree T as a tree consisting of a node in T together with *all* of its descendants, see Section 2.3.

[30] Of course, the distinction disappears in the simpler case that $\mathcal{L}$ is a context-free language (i.e., $\ell = 0$).

`Generate-Intervals`(x, t)

Input: $x \in \mathcal{L}$ (where $\mathcal{L}$ is a partial derivation language parameterized by $\big(n, (i_1, \ldots, i_\ell), (A_0, \ldots, A_\ell)\big)$) and $t \in [2d, n']$, where $n' = n + \ell$.

1. Construct a derivation tree T of arity d, with n' leaves, corresponding to the derivation $A_0 \overset{*}{\Rightarrow} x[1, i_1 - 1] \circ A_1 \circ x[i_1, i_2 - 1] \circ \cdots \circ A_\ell \circ x[i_\ell, n]$ (according to the grammar G).
2. Set $j = 1$.
3. Repeat: (prior to the j^{th} iteration, we have already constructed subtrees $T_1, \ldots, T_{j-1}$ of T).
   (a) Construct a tree T' from T by removing all the vertices of $T_{j'}$ except for the root of $T_{j'}$, for every $j' \in [j - 1]$. Note that there is a natural correspondence between the vertices of T' and the vertices of T from which they were copied.
   (b) If the number of leaves of T' is less than t, then exit the loop.
   (c) Applying Lemma 2.5 to T', with size parameter t, find a subtree of T' with t' leaves such that $t' \in [t/d, t]$. Denote the root of this subtree by v'. Let v be the vertex in T that corresponds to v', and define T_i as the subtree of T rooted at v.
   (d) Increment j by 1.
4. Set $k = j$ and $T_k = T$.
5. For every $j \in [k]$, let B_j be the label of (i.e., the variable associated with) the root of T_j, and let $I_j \subseteq [n']$ be the minimal interval that contains all the leaves of T_j in T.
6. Output $(\overline{I}, \overline{B})$, where $\overline{I} = (I_1, \ldots, I_k)$ and $\overline{B} = (B_1, \ldots, B_k)$.

Figure 2: The `Generate-Intervals` Procedure for the Partial Derivation Language $\mathcal{L}$.

the parts).

We begin with a high-level overview of the transformation $\mathcal{T}$ in the special and slightly simpler case that $\mathcal{L}$ is a context-free language (i.e., $\ell = 0$). In this case, given input $(\overline{I}, \overline{B})$, where $\overline{I} = (I_1, \ldots, I_k)$ and $\overline{B} = (B_1, \ldots, B_k)$, the transformation first constructs a partition of $[n]$ into k parts $S_1, \ldots, S_k$ by setting $S_j = I_j \backslash (I_1 \cup \cdots \cup I_{j-1})$, for every $j \in [k]$. The transformation outputs $S_1, \ldots, S_k$ as well as (concise) descriptions of k partial derivation languages $\mathcal{L}_1, \ldots, \mathcal{L}_k$ such that for every $j \in [k]$, the language $\mathcal{L}_j$ is a partial derivation language corresponding to a partial derivation starting from B_j into strings that have variables B_{j_i} at fixed coordinates corresponding to the relative position of all subintervals I_{j_i} of I_j. The transformation also checks that the languages $\mathcal{L}_1, \ldots, \mathcal{L}_k$ are non-empty so that the distance of $x'[S_j]$ from the corresponding language $\mathcal{L}_j$ is well defined (this check is indeed necessary — see discussion in Section 1.2).

The case that $\mathcal{L}$ is a partial derivation language (rather than a context-free language) is quite similar, where a fairly minor complication that arises is that we need to remove the non-terminal coordinates from the partition, and so we set $S_j = I_j \backslash (I_1 \cup \cdots \cup I_{j-1} \cup \{i'_1, \ldots, i'_\ell\})$. For technical reasons, it is more convenient for us to view each one of the non-terminal coordinates $i'_1, \ldots, i'_\ell$ as an additional *artificial* singleton interval. The transformation $\mathcal{T}$ is detailed in Fig. 3, and the *completeness* and *soundness* requirements (which were stated loosely above) are stated formally in the following two lemmas (Lemmas 4.3 and 4.4).

Lemma 4.3 (Completeness of $\mathcal{T}$)**.** *For every $x \in \mathcal{L}$ (where $\mathcal{L}$ is a partial derivation language parameterized by $(n, (i_1, \ldots, i_\ell), (A_0, \ldots, A_\ell))$) and parameter $t \in [2d, n']$, if $(\overline{I}, \overline{B}) \in ([n']^2)^k \times V^k$ is the output of* `Generate-Intervals`(x, t)*, then the transformation $\mathcal{T}(\overline{I}, \overline{B})$ does not reject, but rather outputs $((S_1, \langle \mathcal{L}_1 \rangle) \ldots, (S_k, \langle \mathcal{L}_k \rangle))$ such that for every $j \in [k]$:*

1. *$\mathcal{L}_j \subseteq \Sigma^{|S_j|}$ is a partial derivation language on strings of length $n_j = |S_j|$ with ℓ_j fixed variables such that $n_j + \ell_j \leq t$; and,*
2. *$x'[S_j] \in \mathcal{L}_j$, where $x' = x[1, i_1 - 1] \circ A_1 \circ x[i_1, i_2 - 1] \circ \cdots \circ A_\ell \circ x[i_\ell, n]$.*

Proof. Let $x \in \mathcal{L}$ and let $(\overline{I}, \overline{B})$ be the output of `Generate-Intervals`(x, t), where $\overline{I} = (I_1, \ldots, I_k)$ and $\overline{B} = (B_1, \ldots, B_k)$. Since $I_k = [n']$ and $B_k = A_0$, the transformation $\mathcal{T}(\overline{I}, \overline{B})$ does not reject, but rather outputs $((S_1, \langle \mathcal{L}_1 \rangle), \ldots, (S_k, \langle \mathcal{L}_k \rangle))$. Let $I'_1, \ldots, I'_{\ell+k}$ be as defined in $\mathcal{T}$ (see Fig. 3).

The fact that, for every $j \in [k]$, it holds that $\mathcal{L}_j \subseteq \Sigma^{|S_j|}$ is a partial derivation language on strings of length n_j with ℓ_j fixed variables such that $n_j + \ell_j \leq t$ follows from the fact that the quantity $n_j + \ell_j$ corresponds to the number of leaves of the subtree that was constructed in Item 3c in the `Generate-Intervals` procedure (recall that this subtree had at most t leaves).

To complete the proof of Lemma 4.3, we need to show that $x'[S_j] \in \mathcal{L}_j$, where $x' \stackrel{\text{def}}{=} x[1, i_1 - 1] \circ A_1 \circ x[i_1, i_2 - 1] \circ \cdots \circ A_\ell \circ x[i_\ell, n]$, for every $j \in [k]$. Let $j \in [k]$, and let $\ell_j, i'_{j,1}, \ldots, i'_{j,\ell_j}, B'_{j,1}, \ldots, B'_{j,\ell_j}$ be as in Fig. 3. Let $w = x'[S_j]$, and observe that by construction,

$$B_j \stackrel{*}{\Rightarrow} w[1, i'_{j,1} - 1] \circ B'_{j,1} \circ w[i'_{j,1}, i'_{j,2} - 1] \circ \cdots \circ B'_{j,\ell_j} \circ w[i'_{j,\ell_j}, |S_i|].$$

Hence, $w \in \mathcal{L}_j$ and completeness follows.

□

Lemma 4.4 (Soundness of $\mathcal{T}$)**.** *For every $\varepsilon \in [0, 1]$, every $x \in \Sigma^n$ that is ε-far from $\mathcal{L}$ (parameterized by $\langle \mathcal{L} \rangle = (n, (i_1, \ldots, i_\ell), (A_0, \ldots, A_\ell))$) and every $(\overline{I}, \overline{B}) \in ([n']^2)^k \times V^k$, it holds that $\mathcal{T}(\overline{I}, \overline{B})$ either rejects or outputs a sequence $((S_1, \langle \mathcal{L}_1 \rangle), \ldots, (S_k, \langle \mathcal{L}_k \rangle))$ such that:*

The Transformation $\mathcal{T}(\overline{I}, \overline{B})$

Input: $\overline{I} = (I_1, \ldots, I_k) \in ([n']^2)^k$ and $\overline{B} = (B_1, \ldots, B_k) \in V^k$ (recall that $n' = n + \ell$ and that $\langle \mathcal{L} \rangle = \big(n, (i_1, \ldots, i_\ell), (A_0, \ldots, A_\ell)\big)$).

1. Check that $(\overline{I}, \overline{B})$ is well formed: for every $j < i$ either $I_j \subsetneq I_i$ or $I_j \cap I_i = \emptyset$, and $I_k = [n']$ and $B_k = A_0$ (recall that $A_0 \in V$ is a variable such that all partial derivations in $\mathcal{L}$ start from A_0). If any test fails, then reject[a] and halt.
2. For $j \in [\ell]$, let $I'_j = \{i_j\}$.
3. For $j \in [k]$, let $I'_{\ell+j} = I_j$.
4. For every $j \in [k]$:
   (a) Let $I'_{j,1}, \ldots, I'_{j,\ell_j}$ be the ordered sequence (from left to right) of all maximal (strict) sub-intervals of $I_j = I'_{\ell+j}$ from the set of intervals $\{I'_1, \ldots, I'_{\ell+k}\}$. That is, all intervals (in order) from the set of intervals $\{I'_1, \ldots, I'_{\ell+k}\}$ that are strictly contained in I_j but are not contained in any other interval that is strictly contained in I'_j.[b]
   (b) Let $S_j = I_j \backslash (I'_{j,1} \cup \cdots \cup I'_{j,\ell_j})$.[c]
   (c) For every $s \in [\ell_j]$, let $i_{j,s} \in [|I_j|]$ be the *relative* starting position of the sub-interval $I'_{j,s}$ inside I_j, let $i'_{j,s} = i_{j,s} - \sum_{s'<s} |I'_{j,s'}|$, and let $B'_{j,s}$ be the label of the root of the subtree that corresponds to the interval $I'_{j,s}$. Define the following partial derivation language of G:
   $$\mathcal{L}_j \stackrel{\text{def}}{=} \left\{ w \in \Sigma^{|S_j|} \,:\, B_j \stackrel{*}{\Rightarrow} w[1, i'_{j,1} - 1] \circ B'_{j,1} \circ w[i'_{j,1}, i'_{j,2} - 1] \circ \cdots \circ B'_{j,\ell_j} \circ w[i'_{j,\ell_j}, |S'_j|] \right\}$$
   (see also Fig. 6). That is, $\langle \mathcal{L}_j \rangle = \big(|S_j|, (i'_{j,1}, \ldots, i'_{j,\ell_j}), (B'_{j,1}, \ldots, B'_{j,\ell_j})\big)$.
   (d) If $\mathcal{L}_j = \emptyset$, then reject and halt.[d]
5. Output $\big((S_1, \langle \mathcal{L}_1 \rangle), \ldots, (S_k, \langle \mathcal{L}_k \rangle)\big)$.

[a] In case the reader is bothered by the fact that the transformation may "reject", we can easily avoid rejecting by outputting instead some canonical representation of a "partition" that will always be rejected by the $\mathcal{IPP}$ verifier.

[b] In other words, an interval $I' \in \{I'_1, \ldots, I'_{\ell+k}\}$ is contained in the sequence if and only if $I' \subsetneq I_j$ and $I' \cap I'' \neq I'$, for every $I'' \in \{I'_1, \ldots, I'_{\ell+k}\} \backslash \{I'\}$ such that $I'' \subsetneq I_j$.

[c] Equivalently, $S_j = I_j \setminus (I'_1 \cup \cdots \cup I'_{\ell+j-1})$. We use the slightly more complicated definition to facilitate the proof.

[d] This check, which only requires access to $\langle L_j \rangle$ and the grammar G, can be done in $\mathsf{poly}(n')$ time.

Figure 3: The Transformation $\mathcal{T}$.

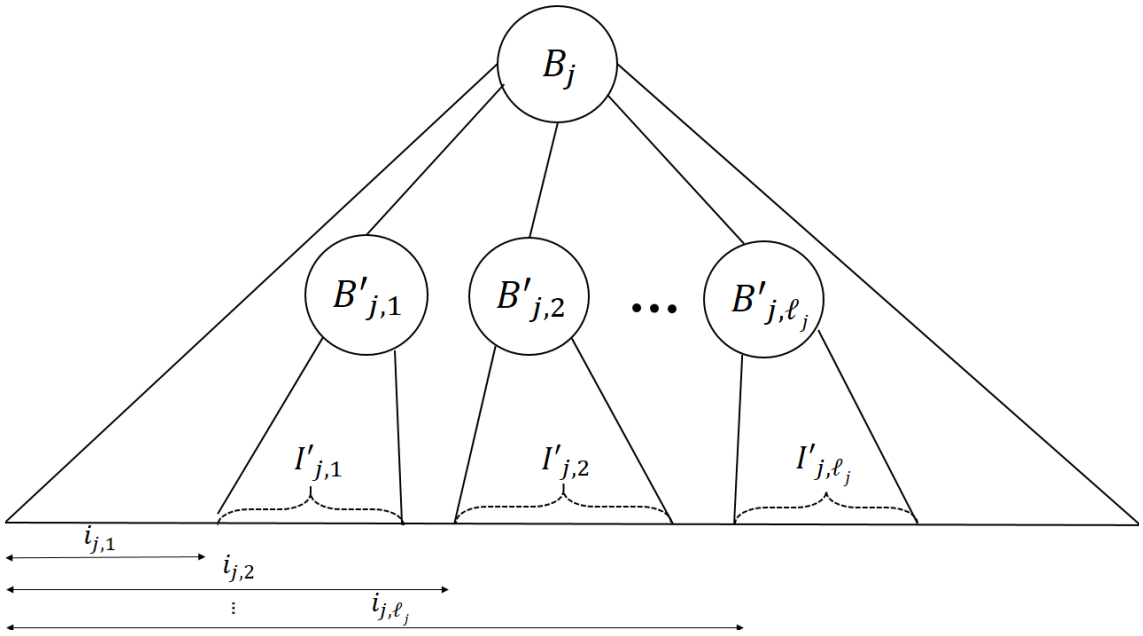

Figure 4: The partial derivation tree that describes the partial derivation $B_j \overset{*}{\Rightarrow} w[1, i'_{j,1}-1] \circ B'_{j,1} \circ w[i'_{j,1}, i'_{j,2}-1] \circ \cdots \circ B'_{j,\ell_j} \circ w[i'_{j,\ell_j}, |S'_j|]$.

1. *The sets $S_1, \ldots, S_k \subseteq [n']\backslash\{i'_1, \ldots, i'_\ell\}$ form a partition of $[n']\backslash\{i'_1, \ldots, i'_\ell\}$.*
2. *It holds that*
$$\mathop{\mathbf{E}}_{j\sim\mathcal{D}}\left[\Delta\left(x'[S_j], \mathcal{L}_j\right)\right] \geq \varepsilon,$$
where $x' = x[1, i_1 - 1] \circ A_1 \circ x[i_1, i_2 - 1] \circ \cdots \circ A_\ell \circ x[i_\ell, n]$ and $\mathcal{D}$ is a distribution over $[k]$ such that $\Pr_{j\sim\mathcal{D}}[j = j'] = |S_{j'}|/n$ for every $j' \in [k]$.

Proof. Let $x \in \Sigma^n$ and let $\overline{I} = (I_1, \ldots, I_k) \in ([n']^2)^k$ be a sequence of intervals and $\overline{B} = (B_1, \ldots, B_k) \in V^k$ a sequence of variables such that the transformation $\mathcal{T}(\overline{I}, \overline{B})$ does not reject and outputs $((S_1, \langle\mathcal{L}_1\rangle), \ldots, (S_k, \langle\mathcal{L}_k\rangle))$. Let $I'_1, \ldots, I'_{\ell+k}$be as defined in $\mathcal{T}$ (see Fig. 3).

To see that $S_1, \ldots, S_k$ form a partition of $[n']\backslash\{i'_1, \ldots, i'_\ell\}$, observe that for each $j \in [k]$, it holds that $S_j = I_j\backslash(I'_{j,1} \cup \cdots \cup I'_{j,\ell_j})$, where $I'_{j,1}, \ldots, I'_{j,\ell_j}$ are the ordered sequence (from left to right) of all maximal sub-intervals of I'_j out of $I'_1, \ldots, I'_{\ell+k}$ (i.e., all intervals that are contained in I_j but are not contained in any other interval that is strictly contained in I_j). Thus, the S_j's are disjoint. Furthermore, since $I'_{\ell+k} = [n']$, for every index $i \in [n']$ there exists $j \in [\ell + k]$ such that $i \in I'_j$. Hence, either $i \in \{i'_1, \ldots, i'_\ell\}$ (in case $j \in [\ell]$) or $i \in S_{j'}$ for some $j' \in [k]$, and so $S_1, \ldots, S_k$ form a partition of $[n']\backslash\{i'_1, \ldots, i'_\ell\}$.

For every $j \in [k]$, let $\varepsilon_j = \Delta\left(x'[S_j], \mathcal{L}_j\right)$. Let $\mathcal{D}$ be the distribution as in the lemma's statement (i.e., $\Pr_{j\sim\mathcal{D}}[j = j'] = |S_{j'}|/n$, for every $j' \in [k]$). Suppose that $\mathbf{E}_{j\sim\mathcal{D}}[\varepsilon_j] < \varepsilon$, for some $\varepsilon \in [0, 1]$, where $x' = x[1, i_1 - 1] \circ A_1 \circ x[i_1, i_2 - 1] \circ \cdots \circ A_\ell \circ x[i_\ell, n]$. We will show that x is ε-close to $\mathcal{L}$.

For every $j \in [k]$, since the transformation *explicitly* checks[31] (in Step 4d) that $\mathcal{L}_j \neq \emptyset$, there

[31] Indeed, this was the reason that we added this additional check, and without it soundness would not hold. See further discussion in Section 1.2.

exists a string $z_j \in \Sigma^{|S_j|}$ such that $z_j \in \mathcal{L}_j$ and $\Delta(x'[S_j], z_j) = \varepsilon_j$ (i.e., $z_j \in \mathcal{L}_j$ minimizes the distance of $x'[S_j]$ to $\mathcal{L}_j$).

Using $z_1, \ldots, z_k$, we construct a string $z \in \mathcal{L}$ that is ε-close to x as follows. Let $z \in \Sigma^n$ such that the string $z' = z[1, i_1 - 1] \circ A_1 \circ z[i_1, i_2 - 1] \circ \cdots \circ A_\ell \circ z[i_\ell, n]$ satisfies $z'[S_j] = z_j$, for every $j \in [k]$. (The fact that such a string z exists follows from the fact that $S_1, \ldots, S_k$ are a partition of $n' \setminus \{i'_1, \ldots, i'_\ell\}$.)

Observe that $\Delta(x, z) = \Delta(x', z') \leq \mathbf{E}_{j \sim \mathcal{D}}\left[\Delta(x'[S_j], z'[S_j])\right] = \mathbf{E}_{j \sim \mathcal{D}}[\varepsilon_j] < \varepsilon$ and so x is ε-close to z. By applying the following claim, with respect to $j = k$, and using the fact that the transformation explicitly checks that $I_k = [n']$ and $B_k = A_0$, we obtain that $A_0 \stackrel{*}{\Rightarrow} z'$, and therefore $z \in \mathcal{L}$. Hence x is ε-close to a string $z \in \mathcal{L}$, and soundness follows.

Claim 4.4.1. *For every $j \in [k]$, it holds that $B_j \stackrel{*}{\Rightarrow} z'[I_j]$.*

Proof. We prove the claim by induction on j. Let $j \in [k]$, and suppose that the claim holds for every $j' < j$. Let $y = z'[S_j]$. Note that $y \in \mathcal{L}_j$. We show that $B_j \stackrel{*}{\Rightarrow} z'[I_j]$.

Recall that $I'_1, \ldots, I'_{\ell+k}$ were fixed above as in Fig. 3. That is, for $j \in [\ell]$, it holds that $I'_j = \{i_j\}$, and for $j \in [\ell + 1, \ell + k]$ it holds that $I'_j = I_{j-\ell}$.

Let $I'_{j,1}, \ldots, I'_{j,\ell_j}$ be the ordered maximal sub-intervals (in the set $\{I'_1, \ldots, I'_{\ell+k}\}$) of I_j. By the construction of $\mathcal{T}$ it holds that

$$\mathcal{L}_j = \left\{ w \in \Sigma^{|S_j|} : B_j \stackrel{*}{\Rightarrow} w[1, i'_{j,1} - 1] \circ B'_{j,1} \circ w[i'_{j,1}, i'_{j,2} - 1] \circ \cdots \circ B'_{j,\ell_j} \circ w[i'_{j,\ell_j}, |S_j|] \right\},$$

where $i_{j,s}$ is the *relative* starting position of the interval $I'_{j,1}$ inside I_j, $i'_{j,s} \stackrel{\text{def}}{=} i_{j,s} - \sum_{s' < s} |I'_{j,s'}|$ and $B'_{j,s}$ is the label of the subtree that corresponds to the interval $I'_{j,s}$, for every $s \in [\ell_j]$. Therefore, since $y \in \mathcal{L}_j$, it holds that

$$B_j \stackrel{*}{\Rightarrow} y[1, i'_{j,1} - 1] \circ B'_{j,1} \circ y[i'_{j,1}, i'_{j,2} - 1] \circ \cdots \circ B'_{j,\ell_j} \circ y[i'_{j,\ell_j}, |S_i|]. \tag{4.1}$$

On the other hand, for every $i \in [\ell_j]$, it holds that

$$B'_{j,s} \stackrel{*}{\Rightarrow} z'[I'_{j,s}], \tag{4.2}$$

where Eq. (4.2) follows from the inductive hypothesis and from the fact that $B'_{j,s} = A_{j,s}$ and $z'[I'_{j,s}] = z'_{j,s} = A_{j,s}$. for $s \in [\ell_j]$.

By combining Eq. (4.1), Eq. (4.2), and the definition of $i'_{j,s}$ we obtain that

$$B_j \stackrel{*}{\Rightarrow} y[1, i'_{j,1}] \circ z'[I'_{j,1}] \circ y[i'_{j,1}, i'_{j,2} - 1] \circ \ldots z'[I'_{j,\ell_j}] \circ y[i'_{j,\ell_j}, |S_i|].$$

The claim follows by observing that

$$z'[I_j] = y[1, i'_{j,1}] \circ z'[I'_{j,1}] \circ y[i'_{j,1}, i'_{j,2} - 1] \circ \ldots z'[I'_{j,\ell_j}] \circ y[i'_{j,\ell_j}, |S_i|],$$

and therefore $B_j \stackrel{*}{\Rightarrow} z'[I_j]$. □

This completes the proof of Lemma 4.4 □

4.2 $\mathcal{IPP}$ for Partial Derivation Languages

Using Lemmas 4.3 and 4.4, we complete the proof of Lemma 4.2 (which is a relatively straightforward implementation of the ideas outlined in Section 1.2).

Proof of Lemma 4.2. Let $G = (V, \Sigma, R, A_{\mathsf{start}})$ be a context-free grammar. We construct a proximity oblivious $\mathcal{IPP}$ for every partial derivation language $\mathcal{L} \subseteq \Sigma^n$ of the grammar G.

The proximity oblivious $\mathcal{IPP}$ has two parameters: r which is the round complexity, and k which roughly corresponds to the amount of communication in each round. The $\mathcal{IPP}$ runs recursively, where each round of communication proceeds as follows. The (honest) prover uses the `Generate-Intervals` procedure on its input x and parameter $t = n'/k$ (where $n' = n+\ell$), to obtain $(\overline{I}, \overline{B})$ and sends $(\overline{I}, \overline{B})$ to the verifier. The verifier applies the transformation $\mathcal{T}(\overline{I}, \overline{B})$ to derive the partition $S_1, \ldots, S_k$ and the corresponding partial derivation languages $\mathcal{L}_1, \ldots, \mathcal{L}_k$. Then, the verifier selects at random $j \in [k]$ and sends j to the prover (where j is distributed according to D as above). The two parties then recurse on input $x'[S_j]$, where $x' \stackrel{\text{def}}{=} x[1, i_1 - 1] \circ A_1 \circ x[i_1, i_2 - 1] \circ \cdots \circ A_\ell \circ x[i_\ell, n]$, with respect to the partial derivation language $\mathcal{L}_j$. The recursion stops once either:

1. $n' \leq O(k)$ (i.e., the input is very short), in which case the prover can send $x^* = x$ to the verifier.[32] Then, the verifier checks that $x^* \in \mathcal{L}$ and that x^* is consistent with x at a randomly selected coordinate; or,

2. r rounds have passed, in which case the verifier reads its entire input x (which has shortened by a multiplicative factor of roughly k in each step of the recursion) and verifies that $x \in \mathcal{L}$.

The $\mathcal{IPP}$ for $\mathcal{L}$, denoted $\mathsf{CFL\text{-}IPP}$, is detailed in Fig. 5.

Without loss of generality, we can measure the complexity of the protocol only when the verifier interacts with the *honest* prover (see discussion in Section 2.1). It can be easily verified that the round complexity is at most r rounds. By Lemma 4.3, the protocol recurses on a partial derivation language $\mathcal{L}_j$ on strings of length n_j with ℓ_j fixed variables such that $n_j + \ell_j \leq n'/k$. Hence, after at most r rounds, the current input length has length at most n'/k^r, where $n' = n + \ell$, and so the query complexity of the $\mathcal{IPP}$ is $O(n'/k^r)$. Since in each round the communication is at most $O(k \log n')$, the communication complexity of the $\mathcal{IPP}$ is $O(rk \log n')$.

Completeness. Let $\mathcal{L}$ be a partial derivation language, with $\langle \mathcal{L} \rangle \stackrel{\text{def}}{=} (n, \bar{i}, \overline{A})$, and let $x \in \mathcal{L}$. We show that perfect completeness hold by induction on r. For $r = 0$ or $n' = O(p)$, perfect completeness follows from the fact that $\mathcal{V}$ just checks that $x \in \mathcal{L}$. For $r > 1$ (with $n'/k \geq 2d$), by Lemma 4.3, the verifier produces $((S_1, \langle \mathcal{L}_1 \rangle), \ldots, (S_k, \langle \mathcal{L}_k \rangle))$ such that $\mathcal{L}_j$ is a partial derivation language and $x'[S_j] \in \mathcal{L}_j$, for every $j \in [k]$ (in particular, $\mathcal{L}_j \neq \emptyset$). Hence, by the inductive hypothesis, the verifier in the $r - 1$ round protocol for $\mathcal{L}_j$ will accept on input $x'[S_j]$ with probability 1.

Soundness. Soundness follows from the following lemma, which is proved by induction on the number of rounds r.

[32] This check is to ensure that the parameter $t = n'/k$ is larger than $2d$.

The Protocol $\mathsf{CFL\text{-}IPP}^{\mathcal{L}}_{k,r}$

Parameters: $\mathcal{L} \subseteq \Sigma^n$ is a partial derivation language, with $\langle \mathcal{L} \rangle = \big(n, (i_1, \ldots, i_\ell), (A_0, \ldots, A_\ell)\big)$, the parameters $k, r \in \mathbb{N}$ correspond (roughly) to the amount of communication in each round and to the number of rounds, respectively. Let $n' = n + \ell$.

Prover's Input: Direct access to $x \in \mathcal{L}$, with $n \stackrel{\text{def}}{=} |x|$.

Verifier's Input: Oracle access to x, and direct access to $\langle \mathcal{L} \rangle$.

1. If $r = 0$, then the verifier $\mathcal{V}$ checks whether $x \in \mathcal{L}$ by explicitly reading all of x. If $x \in \mathcal{L}$, then $\mathcal{V}$ accepts, otherwise it rejects, and in either case both parties terminate the protocol.
2. If $n' = O(k)$, the prover sends $x^* = x$ to $\mathcal{V}$. The verifier $\mathcal{V}$ accepts if $x^* \in \mathcal{L}$ *and* x^* agrees with x at a randomly chosen coordinate. Otherwise $\mathcal{V}$ rejects, and in either case both parties terminate the protocol.
3. The Prover $\mathcal{P}$:
   (a) Invoke $\texttt{Generate-Intervals}(x, n'/k)$ to obtain $\big(\overline{I}, \overline{B}\big)$.
   (b) Send $\big(\overline{I}, \overline{B}\big)$ to $\mathcal{V}$.
4. The Verifier $\mathcal{V}$:
   (a) Invoke $\mathcal{T}\big(\overline{I}, \overline{B}\big)$. If the transformation rejects, then immediately reject and halt. Otherwise, denote the output of the transformation by $\big((S_1, \langle \mathcal{L}_1 \rangle), \ldots, (S_k, \langle \mathcal{L}_k \rangle)\big)$.[a]
   (b) Select $j \sim \mathcal{D}$, where $\mathcal{D}$ is the distribution in the statement of Lemma 4.4 (i.e., $\Pr_{j \sim \mathcal{D}}[j = j'] = |S_{j'}|/n$, for every $j' \in [k]$).
   (c) Send j to $\mathcal{P}$.
5. Both parties (recursively) invoke $\mathsf{CFL\text{-}IPP}^{\mathcal{L}_j}_{r-1,k}$ on input $x'[S_j]$.

[a] The reader may note that, in contrast to Fig. 1, the verifier does not check that $\mathcal{L}_j \neq \emptyset$, for every $j \in [k]$. This check is actually performed *within* the transformation $\mathcal{T}$ (see Step 4d in Fig. 3).

Figure 5: $\mathcal{IPP}$ for Context-Free Languages

Lemma 4.5. *Let $\mathcal{L}$ be a partial derivation language, and let $k \geq 1$ and $r \geq 0$. For every $\varepsilon \in [0,1]$ and every x that is ε-far from $\mathcal{L}$, and for every cheating prover strategy P^* it holds that:*

$$\Pr\left[(V, P^*)(x) = 0\right] \geq \varepsilon,$$

where $\mathcal{V}$ is the verifier in $\mathsf{CFL\text{-}IPP}^{\mathcal{L}}_{r,p}$ (see Fig. 5).

Proof. We first consider the trivial case that $n' = O(k)$. In this case, if x^* is ε-close to x, then $x^* \notin \mathcal{L}$ (since x is ε-far from $\mathcal{L}$) and the verifier rejects with probability $1 \geq \varepsilon$. Otherwise, x^* is ε-far from $\mathcal{L}$ and the verifier rejects with probability at least ε when checking the consistency of x^* and x.

We proceed to the more interesting case, in which $n'/k > 2d$, and prove by induction on r. For $r = 0$, the verifier ignores the prover and reads all of x. Hence, if $B(x) \neq 1$, then the verifier rejects with probability 1.[33]

For $r \geq 1$, let $\varepsilon \in [0,1]$, let $x \in \Sigma^n$ be ε-far from $\mathcal{L}$, and let P^* be a *deterministic* cheating prover strategy for the protocol $\mathsf{CFL\text{-}IPP}^{\mathcal{L}}_{r,k}$ of Fig. 5 (with r rounds). Let $(\overline{I}, \overline{B})$ be the first message sent by P^* to $\mathcal{V}$. Assume that the invocation of the transformation $\mathcal{T}(\overline{I}, \overline{B})$ does not reject (otherwise the verifier rejects with probability 1, and we are done), and denote its output by $((S_1, \langle \mathcal{L}_1 \rangle), \ldots, (S_k, \langle \mathcal{L}_k \rangle))$.

For every $j \in [k]$, let $\varepsilon_j = \Delta\left(x'[S_j], \mathcal{L}_j\right)$ denote the relative distance of $x'[S_j]$ from $\mathcal{L}_j$, and let $\mathcal{D}$ be the distribution as in $\mathsf{CFL\text{-}IPP}^{\mathcal{L}}_{r,k}$. By Lemma 4.4, it holds that

$$\mathop{\mathbf{E}}_{j \sim \mathcal{D}}[\varepsilon_j] \geq \varepsilon. \tag{4.3}$$

For every $j \in [k]$, let P^*_j be the residual $r-1$ round strategy of P^* after receiving the message j from $\mathcal{V}$ in the first round, and let $\mathcal{V}_j$ be the residual strategy of $\mathcal{V}$ after fixing its first message to j. Observe that, by construction, $\mathcal{V}_j$ is simply the strategy of the verifier in the protocol $\mathsf{CFL\text{-}IPP}^{\mathcal{L}_j}_{k,r-1}$. Hence, by the inductive hypothesis, for every $j \in [k]$ it holds that

$$\Pr\left[(\mathcal{V}_j, \mathcal{P}^*_j)(x'[S_j]) = 0\right] \geq \varepsilon_j. \tag{4.4}$$

Using Eqs. (4.3) and (4.4) we obtain that:

$$\Pr[(V, P^*)(x) = 0] = \mathop{\mathbf{E}}_{j \sim \mathcal{D}}\left[\Pr[(\mathcal{V}_j, \mathcal{P}^*_j)(x'[S_j]) = 0]\right] \geq \mathop{\mathbf{E}}_{j \sim \mathcal{D}}[\varepsilon_j] \geq \varepsilon, \tag{4.5}$$

and the lemma follows. □

This concludes the proof of Lemma 4.2 and Theorem 1.3. □

Remark 4.6 (Computational Complexity)**.** *The $\mathcal{IPP}$ prover in Fig. 5 can be implemented in time $\mathsf{poly}(n,k,r)$. As for the $\mathcal{IPP}$ verifier, Step 4d in Fig. 3 can be implemented in time $\mathsf{poly}(n)$, and so we obtain a total running-time of $\mathsf{poly}(n,k,r)$, which is super-linear. We remark that for context-free languages whose* partial derivation languages *are themselves context-free languages, we can actually do better and obtain running time $\mathsf{poly}(\log n, k, r)$ (an example for such a context-free*

[33] In the trivial case that $\varepsilon = 0$ (i.e., $B(x) = 1$), the verifier also satisfies the requirement, since it rejects with probability at least $0 = \varepsilon$.

language is the language of balanced parentheses expressions, see Section 4.3). See Appendix D for details.

Alternatively, by increasing the round complexity of our $\mathcal{IPP}$, we can also obtain sub-linear time verification. The technique is similar to that described in Remark 3.2. More specifically, we can implement Step 4d in Fig. 3 (i.e., checking that a given partial derivation language is non-empty (which is the main bottleneck in our $\mathcal{IPP}$)) via an interactive proof-system. To do so, we first construct a (logspace) uniform low-depth circuit that, given the description of a partial derivation language, outputs 1 if and only if the language is non-empty. An efficient interactive proof-system follows from the efficient interactive proof-system for low-depth computation of Goldwasser et al. [GKR08, Theorem 1]. Details follow.

Fix the grammar $G = (V, \Sigma, R, A_{\mathsf{start}})$ and consider a description $(m, \bar{i}, \overline{A})$ of a partial derivation language, where $\bar{i} = (i_1, \ldots, i_\ell)$ and $\overline{A} = (A_0, \ldots, A_\ell)$. Given $(m, \bar{i}, \overline{A})$, the circuit first constructs a string $z \in (V \cup \{*\})^{m+\ell}$, where $'*'$ is some character that does not belong to $V \cup \Sigma$ and $z \stackrel{\text{def}}{=} *^{i_1-1} \circ A_1 \circ *^{i_2-i_1} \circ \cdots \circ A_\ell \circ *^{m-i_\ell+1}$. The circuit then checks whether z can be derived from A_0, according to an auxiliary (unary) grammar G', which is identical to G except that all the terminals are replaced by the unique terminal $'*'$. By a result of Ruzzo [Ruz81], membership in context-free languages can be computed by a (logspace uniform) $\mathcal{NC}_2$ circuit, and so we obtain a ($O(\log(m) + \log(|\ell|))$-space uniform) circuit that checks if the partial derivation language is non-empty, in depth $\mathsf{polylog}(m + \ell)$ and size $\mathsf{poly}(m, \ell)$.

Given the above circuit, we can use [GKR08, Theorem 1] to obtain an interactive proof-system in which the verifier runs in $\ell \cdot \mathsf{poly}(\log(\ell), \log(m))$ time and the prover runs in time $\mathsf{poly}(m, \ell)$. We note that using this proof-system inside our $\mathcal{IPP}$ increases the round complexity of our $\mathcal{IPP}$ by a poly-logarithmic factor.

Remark 4.7 ($\mathcal{MAP}$s for Context-Free Languages)**.** *Theorem 1.2 follows directly from the proof of Lemma 4.2, while noting that the two issues the arise in the case of $\mathcal{MAP}$s for ROBPs (see Section 3.2) apply also here and can be resolved similarly.*

4.3 Improved $\mathcal{MAP}$s for Specific Context-Free Languages

In this section we show that the efficiency of the $\mathcal{MAP}$s for general context-free languages (i.e., Theorem 1.1) can be improved for context-free languages whose corresponding partial derivation languages have *efficient* testers (which do not use a proof). Most notably, we obtain such an improvement for the Dyck languages (i.e., languages of balanced parentheses expressions).

Recall that in the proof of Theorem 1.1, given the $\mathcal{MAP}$ proof, the $\mathcal{MAP}$ verifier (implicitly) constructs a partition $S_1, \ldots, S_k$ of $[n]$ and partial derivation languages $\mathcal{L}_1, \ldots, \mathcal{L}_k$. Then, the verifier chooses an index $j \in [k]$ at random and checks whether $x[S_j] \in \mathcal{L}_j$ by explicitly reading all of $x[S_j]$. However, by Lemma 4.4, the $\mathcal{MAP}$ verifier does not really have to check that $x[S_j] \in \mathcal{L}_j$ *exactly*, but rather it suffices to check that $x[S_j]$ is *close* to $\mathcal{L}_j$. Since no non-trivial tester is known for general context-free languages (let alone for their corresponding partial derivation languages), we could not use this fact to our advantage in the general case. However, for some *specific* languages, such as the Dyck languages, more efficient testers are known and we can utilize them to improve the efficiency of our $\mathcal{MAP}$s.

A technical difficulty that we encounter when taking this approach is that when testing whether $x[S_j]$ is close to $\mathcal{L}_j$ it is not a priori clear which value of the proximity parameter the verifier should use (recall that Lemma 4.4 only guarantees that $x[S_j]$ is ε-far for an *average* $j \in [k]$ but not

necessarily for every $j \in [k]$). Of course, if $\mathcal{L}_j$ has a *proximity-oblivious tester*, then the issue is mute and we can just run the tester directly. In the more general case, we can simply apply an averaging argument. The naive averaging argument shows that for an $\varepsilon/2$ fraction of $j \in [k]$, it holds that $x[S_j]$ is $\varepsilon/2$ far from $\mathcal{L}_j$. However, by applying a more refined averaging argument, due to Levin [Lev85] (see [Gol13, Appendix A.2]), we obtain an additional improvement.

Lemma 4.8. *Let G be a context-free grammar and $\alpha \geq 0$ and $\beta \geq 1$ be constants. Suppose that every partial derivation language of G (as in Definition 4.1) has a property tester with query complexity $O\left(m^{\alpha} \cdot \delta^{-\beta}\right)$ for inputs of length m and proximity parameter $\delta > 0$. Then, for every $k \geq 1$ the language $\mathcal{L}$ has an $\mathcal{MAP}$ with proof complexity $O(k \log n)$ and query complexity $O((n/k)^{\alpha} \cdot \varepsilon^{-\beta} \cdot \log^2(1/\varepsilon))$. Furthermore, if $\alpha = 0$, then the query complexity is at most $O((n/k)^{1-1/\beta} \cdot \varepsilon^{-1} \cdot \log^3(1/\varepsilon))$.*

The $\mathcal{MAP}$ in Lemma 4.8 has one-sided error if and only if the testers for the partial derivation languages have one-sided errors. However, even if the resulting $\mathcal{MAP}$ has two-sided error, a one-sided error $\mathcal{MAP}$ (with only a poly-logarithmic overhead) can be obtained by applying a generic transformation from two-sided error $\mathcal{MAP}$s into one-sided error $\mathcal{MAP}$s (see of [GR15, Theorem 4.3]).

Note that the alternative bound for $\alpha = 0$ improves over the general case only for sub-constant values of the proximity parameter (i.e., $\varepsilon < (n/k)^{-1/\beta} \cdot \mathsf{polylog}(n)$). The bound is obtained by observing that, for very small values of the proximity parameter, it is advantageous to read the entire input rather than apply the tester. We defer the proof of Lemma 4.8, which is relatively straightforward, to Appendix C.

Using Lemma 4.8 we now show how to construct an improved $\mathcal{MAP}$ for the Dyck languages. Loosely speaking, the *κ^{th}-order Dyck language* consists of all of strings that form a balanced parenthesis expression with κ distinct types of parentheses. The Dyck languages can be defined via a context-free grammar as follows.

Definition 4.9. *Let $\kappa \in \mathbb{N}$ be a constant. The κ^{th}-order Dyck language, denoted Dyck_κ, is the language generated by the context-free grammar $G_{\mathsf{Dyck}_\kappa} = (V, \Sigma_\kappa, R, A_{\mathsf{start}})$, where $V = \{A\}$, $A_{\mathsf{start}} = A$, $\Sigma_\kappa = \{`[_1', `]_1', `[_2', `]_2', \ldots, `[_\kappa', `]_\kappa'\}$, and the production rules R consist of: (1) $A \Rightarrow [_i\, A\,]_i$ for every $i \in [\kappa]$, (2) $A \Rightarrow AA$, (3) $A \Rightarrow \lambda$, where λ denotes the empty string.*

Alon *et al.* [AKNS00] showed a tester (with two-sided error) for the first order Dyck language (i.e., Dyck_1) with query complexity $\widetilde{O}(1/\varepsilon^2)$. As for higher order Dyck languages, Parnas *et al.* [PRR01] showed that any Dyck language (of any fixed order) can be tested (with two-sided error) by making $O(n^{2/3} \cdot \varepsilon^{-3})$ queries.[34] Furthermore, by the following proposition, the foregoing results can be extended to the case of *partial derivation languages* of the Dyck languages (with respect to the foregoing grammars).

Proposition 4.10. *Let $m, \kappa \in \mathbb{N}$. If $\mathcal{L} \subseteq (\Sigma_\kappa)^m$ is a partial derivation language of the grammar G_{Dyck_κ}, then $\mathcal{L}$ is equal to $\mathsf{Dyck}_\kappa \cap (\Sigma_\kappa)^m$.*

Proof. Let $\mathcal{L} \subseteq (\Sigma_\kappa)^m$ be a partial derivation language of G_{Dyck_κ} such that $\langle \mathcal{L} \rangle = (m, (i_1, \ldots, i_\kappa), (A, \ldots, A))$ (here we used the fact that the grammar G_{Dyck_κ} uses only a single variable – A).

[34] For perspective, recall that Parnas *et al.* [PRR01] also showed that, for $\kappa \geq 2$, any tester (which does not use a proof) for Dyck_κ must make at least $\widetilde{\Omega}(n^{1/11})$ queries.

On one hand, if $x \in \mathcal{L}$, then $A \stackrel{*}{\Rightarrow} x[1, i_1 - 1] \circ A \circ x[i_1, i_2 - 1] \circ \cdots \circ A \circ x[i_\ell, m]$. Using the production rule $A \Rightarrow \lambda$ we have that $A \stackrel{*}{\Rightarrow} x[1, i_1 - 1] \circ x[i_1, i_2 - 1] \circ \cdots \circ x[i_\ell, m] = x$ and therefore $x \in \mathsf{Dyck}_\kappa \cap (\Sigma_\kappa)^m$.

On the other hand, if $x \in \mathsf{Dyck}_\kappa \cap (\Sigma_\kappa)^m$, then $A \stackrel{*}{\Rightarrow} x$. The following claim shows that, for the Dyck grammars, we can generate a partial derivation in which A is inserted in any desired sequence of positions. Therefore, $A \stackrel{*}{\Rightarrow} x[1, i_1 - 1] \circ A \circ x[i_1, i_2 - 1] \circ \cdots \circ A \circ x[i_\ell, m]$, which implies that $x \in \mathcal{L}$.

Claim 4.10.1. *Let $\alpha \in (\Sigma_\kappa \cup \{A\})^{m'}$, for some $m' \in \mathbb{N}$, and let $i \in [m']$. If $A \stackrel{*}{\Rightarrow} \alpha$, then $A \stackrel{*}{\Rightarrow} \alpha[1, i-1] \circ A \circ \alpha[i, m']$.*

Proof. Since $A \stackrel{*}{\Rightarrow} \alpha$ (according to the grammar G_{Dyck_κ}), there exists a corresponding partial derivation tree T, in which all internal vertices are labeled by the variable A and each leaf is labeled by either 'A', '$[_j$', '$]_j$', for some $j \in [\kappa]$. We prove the claim by extending T into a partial derivation tree T' that corresponds to the partial derivation $A \stackrel{*}{\Rightarrow} \alpha[1, i-1] \circ A \circ \alpha[i, m']$.

Denote the i^{th} leaf of T by v and denote v's parent by u. The specific way in which T' is constructed from T depends on whether the label of v is 'A', '$[_j$' or '$]_j$' (for some $j \in [\kappa]$), and is detailed in Fig. 6.

□

This concludes the proof of Proposition 4.10. □

Thus, the property testers of [PRR01] for the Dyck languages are also testers for the partial derivation languages (of the Dyck languages), and we obtain the following result.

Theorem 4.11. *Let $\kappa \geq 2$. For every p such that $2 \leq p \leq n$, there exists an $\mathcal{MAP}$ for Dyck_κ that uses a proof of length $O(p \log n)$ and has query complexity $O((n/p)^{2/3} \cdot \varepsilon^{-3} \cdot \log^2(1/\varepsilon))$. Furthermore, there exists an $\mathcal{MAP}$ with* one-sided error *for Dyck_κ that uses a proof of length $O(p \log n + \mathsf{polylog}(n))$ and has query complexity $(n/p)^{2/3} \cdot \varepsilon^{-3} \cdot \mathsf{polylog}(n)$.*

The furthermore clause is obtained by applying the generic transformation from one-sided error $\mathcal{MAP}$ into two-sided error $\mathcal{MAP}$ (see [GR15, Theorem 4.3]) and using the fact that without loss of generality we may assume that $\varepsilon \geq 1/n$ (and so $\log^2(1/\varepsilon) \leq \mathsf{polylog}(n)$). We conclude this section with some second order remarks.

Improvement for Dyck_1 and $\varepsilon \ll 1/\sqrt{n}$. For Dyck_1 (i.e., $\kappa = 1$), and for small values of the proximity parameter (i.e., $\varepsilon < \frac{1}{\sqrt{n} \cdot \mathsf{polylog}(n)}$) we can improve Theorem 4.11, by using the tester of Alon *et al.* [AKNS00] (which has query complexity $\widetilde{O}(1/\varepsilon^2)$). Using the special case of Lemma 4.8, we obtain query complexity $O(\sqrt{n/p} \cdot \varepsilon^{-1} \cdot \log^3(1/\varepsilon))$ with a proof of length $O(p \log n)$.

Extension to $\mathcal{IPP}$s. The idea of applying non-trivial testers can also be used to obtain improved $\mathcal{IPP}$s, by applying the tester after the last round of interaction (instead of running the trivial tester that reads the entire (current) input). The savings in this case are less significant since the query and communication complexities of our $\mathcal{IPP}$s are already fairly small. Hence, we only elaborate briefly on these $\mathcal{IPP}$s below.

If the partial derivation languages of the grammar have *proximity-oblivious* testers, then the latter can simply be employed in the last step of the recursion in Fig. 5. However, if only standard

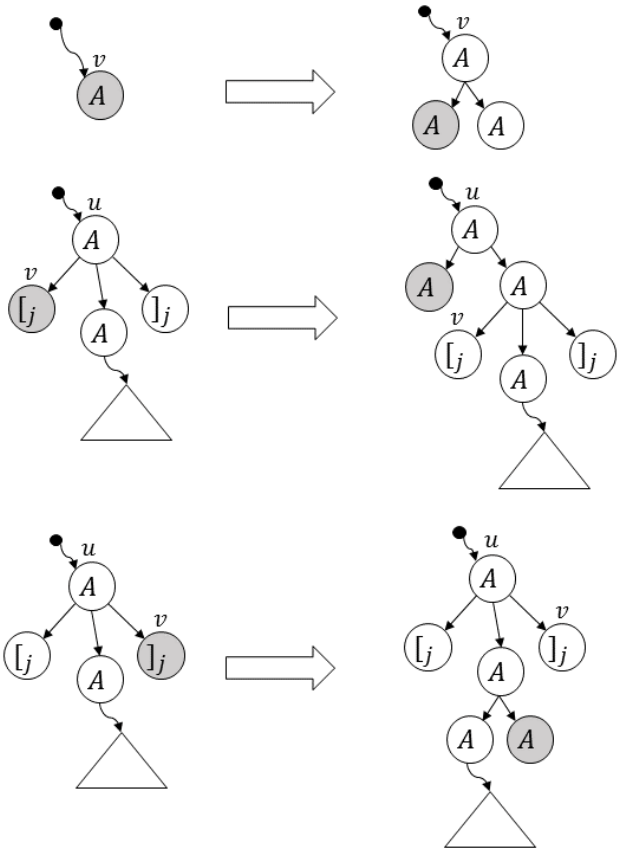

Figure 6: Construction of T' from T. The original tree T is on the left, and the new tree T' is on the right. In each case the i^{th} leaf of the tree has a shaded background, both in T and in T' (note that in all cases the i^{th} leaf of T is v and the i^{th} leaf of T' is labeled by A, the newly inserted symbol).

testers (which are not proximity oblivious) are available, then we can generalize the strategy in the proof of Lemma 4.8 by applying an averaging argument in each step of the recursion, while incurring an $\widetilde{O}(1/\varepsilon)$ multiplicative overhead in each round. Unfortunately, the latter strategy results in an exponential dependence on the round complexity of the protocol.

Computational Complexity for Dyck Languages. In general, as noted in Remark 4.6, the running time of the verifier in Fig. 5 is $\mathsf{poly}(n)$ (because it verifies that each of the languages $\mathcal{L}_1, \ldots, \mathcal{L}_k$ is non-empty). However, as shown in Proposition 4.10, for the Dyck languages, the partial derivation languages $\mathcal{L}_1, \ldots, \mathcal{L}_k$ are themselves Dyck languages. Since the Dyck language on m-bit strings is non-empty if and only if m is even, the running time of the verifier can be reduced to $\mathsf{poly}(\log n, k, r)$ (see also Appendix D).

The $\mathcal{MAP}$ proof in Theorem 4.11 is generated efficiently (i.e., in time $\mathsf{poly}(n)$) for every context-free language, and in particular for the Dyck language. However, for the furthermore clause of Theorem 4.11, we apply the transformation of [GR15, Theorem 4.3], which in general does not preserve *computational* efficiency of the proof generating procedure. Hence, we do not obtain

an $\mathcal{MAP}$ for the Dyck languages that simultaneously has both one-sided error and an efficient procedure of generating the $\mathcal{MAP}$ proof.

References

[AKNS00] Noga Alon, Michael Krivelevich, Ilan Newman, and Mario Szegedy. Regular languages are testable with a constant number of queries. *SIAM J. Comput.*, 30(6):1842–1862, 2000.

[Bol05] Beate Bollig. Property testing and the branching program size of boolean functions. In *Fundamentals of Computation Theory, 15th International Symposium, FCT 2005, Lübeck, Germany, August 17-20, 2005, Proceedings*, pages 258–269, 2005.

[BSGH+06] Eli Ben-Sasson, Oded Goldreich, Prahladh Harsha, Madhu Sudan, and Salil P. Vadhan. Robust PCPs of proximity, shorter PCPs, and applications to coding. *SIAM J. Comput.*, 36(4):889–974, 2006.

[DR06] Irit Dinur and Omer Reingold. Assignment testers: Towards a combinatorial proof of the PCP theorem. *SIAM J. Comput.*, 36(4):975–1024, 2006.

[EKR04] Funda Ergün, Ravi Kumar, and Ronitt Rubinfeld. Fast approximate probabilistically checkable proofs. *Inf. Comput.*, 189(2):135–159, 2004.

[FGL14] Eldar Fischer, Yonatan Goldhirsh, and Oded Lachish. Partial tests, universal tests and decomposability. In *Innovations in Theoretical Computer Science, ITCS'14, Princeton, NJ, USA, January 12-14, 2014*, pages 483–500, 2014.

[GGR98] Oded Goldreich, Shafi Goldwasser, and Dana Ron. Property testing and its connection to learning and approximation. *Journal of the ACM (JACM)*, 45(4):653–750, 1998.

[GKR08] Shafi Goldwasser, Yael Tauman Kalai, and Guy N. Rothblum. Delegating computation: interactive proofs for muggles. In *STOC*, pages 113–122, 2008.

[Gol99] Oded Goldreich. *Modern cryptography, probabilistic proofs and pseudorandomness*, volume 17 of *Algorithms and Combinatorics*. Springer-Verlag, 1999.

[Gol13] Oded Goldreich. On multiple input problems in property testing. *Electronic Colloquium on Computational Complexity (ECCC)*, 20:67, 2013.

[GR62] Seymour Ginsburg and H Gordon Rice. Two families of languages related to ALGOL. *Journal of the ACM (JACM)*, 9(3):350–371, 1962.

[GR11] Oded Goldreich and Dana Ron. On proximity-oblivious testing. *SIAM Journal on Computing*, 40(2):534–566, 2011.

[GR15] Tom Gur and Ron D. Rothblum. Non-interactive proofs of proximity. In *Proceedings of the 2015 Conference on Innovations in Theoretical Computer Science, ITCS 2015, Rehovot, Israel, January 11-13, 2015*, pages 133–142. ACM, 2015.

[GS12] Oded Goldreich and Igor Shinkar. Two-sided error proximity oblivious testing - (extended abstract). In *APPROX-RANDOM*, pages 565–578, 2012.

[HMU06] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[KR14] Yael Tauman Kalai and Ron D. Rothblum. Arguments of proximity. Manuscript, 2014.

[KW88] Klaus Kriegel and Stephan Waack. Lower bounds on the complexity of real-time branching programs. *ITA*, 22(4):447–459, 1988.

[Lev85] Leonid A. Levin. One-way functions and pseudorandom generators. In *STOC*, pages 363–365, 1985.

[LSH65] Philip M. Lewis, Richard Edwin Stearns, and Juris Hartmanis. Memory bounds for recognition of context-free and context-sensitive languages. In *SWCT (FOCS)*, pages 191–202, 1965.

[New02] Ilan Newman. Testing membership in languages that have small width branching programs. *SIAM Journal on Computing*, 31(5):1557–1570, 2002.

[New10] Ilan Newman. Property testing of massively parametrized problems - a survey. In *Property Testing*, pages 142–157, 2010.

[PRR01] Michal Parnas, Dana Ron, and Ronitt Rubinfeld. Testing parenthesis languages. In *RANDOM-APPROX*, pages 261–272, 2001.

[RS96] Ronitt Rubinfeld and Madhu Sudan. Robust characterizations of polynomials with applications to program testing. *SIAM J. Comput.*, 25(2):252–271, 1996.

[Ruz81] Walter L. Ruzzo. On uniform circuit complexity. *J. Comput. Syst. Sci.*, 22(3):365–383, 1981.

[RVW13] Guy N. Rothblum, Salil Vadhan, and Avi Wigderson. Interactive proofs of proximity: Delegating computation in sublinear time. In *Proceedings of the 45th annual ACM Symposium on Theory of Computing (STOC)*, 2013.

A Parallel Repetition of $\mathcal{IPP}$s

The k-fold parallel repetition of an $\mathcal{IPP}$ $(\mathcal{V}_1, \mathcal{P}_1)$ is an $\mathcal{IPP}$ $(\mathcal{V}_k, \mathcal{P}_k)$ in which the two parties perform k parallel repetitions of $(\mathcal{V}_1, \mathcal{P}_1)$, using independent random coins for each invocation. Note that the query and communication complexities of $(\mathcal{V}_k, \mathcal{P}_k)$ are k times the query and communication complexities of $(\mathcal{V}_1, \mathcal{P}_1)$, respectively. The verifier $\mathcal{V}_k$ accepts if $\mathcal{V}_1$ accepts in a majority of the k invocations. For our applications it suffices to focus on the case that $(\mathcal{V}_1, \mathcal{P}_1)$ has a one-sided error, in which case $\mathcal{V}_k$ can just check that $\mathcal{V}_1$ accepts in *all* the k invocations.

It is clear that if $(\mathcal{V}_1, \mathcal{P}_1)$ has perfect completeness, then so does $(\mathcal{V}_k, \mathcal{P}_k)$. The main challenge is in proving that the soundness error decreases exponentially with k since if P^* is the optimal

cheating strategy against $\mathcal{V}$, it is not a priori clear that the optimal cheating strategy against $\mathcal{V}_k$ is k independent copies of $\mathcal{P}^*$.

Nevertheless, the following lemma, taken verbatim from [Gol99, Lemma C.1] shows that the soundness error for any interactive machine $\mathcal{V}_k$ does decrease exponentially.

Lemma A.1 ([Gol99, Lemma C.1])**.** *Let $\mathcal{V}_1$ be an interactive machine, and $\mathcal{V}_k$ be an interactive machine obtained from $\mathcal{V}_1$ by playing k versions of $\mathcal{V}_1$ in parallel. Let*

$$p_1(x) \stackrel{\text{def}}{=} \max_{\mathcal{P}^*} \{\Pr[(\mathcal{P}^*, \mathcal{V}_1)(x) = 1]\}, \text{ and}$$
$$p_k(x) \stackrel{\text{def}}{=} \max_{\mathcal{P}^*} \{\Pr[(\mathcal{P}^*, \mathcal{V}_k)(x) = 1]\}.$$

Then,

$$p_k(x) = (p_1(x))^k.$$

We stress that Lemma A.1 holds for any x and is independent of the operation of $\mathcal{V}_1$. It holds as long as $\mathcal{V}_k$ executes k independent copies of $\mathcal{V}_1$ and accepts if all copies accept. Hence, it holds also when $\mathcal{V}_1$ is an $\mathcal{IPP}$ verifier; in that case $\mathcal{V}_k$ has query complexity that is k times that of $\mathcal{V}_1$.

B Computing ROBPs in Low-Depth

For any branching program B (including branching programs that are not *read-once*), we show that the language $\mathcal{L}_B = \{x \in \{0,1\}^* : B(x) = 1\}$ can be recognized by a $\mathsf{poly}(|B|, n)$-size circuit of depth $O((\log(|B|))^2)$ (with fan-in 2). We stress that the branching program B is fixed and the circuit only gets x as input. For simplicity, we assume without loss of generality that B has a *unique* accepting sink (otherwise we can add a new unique accepting sink t and have all former accepting sinks direct to t).

The idea (which is in essence the folklore proof that (non-deterministic) log-space is contained in $\mathcal{NC}_2$) proceeds as follows. First, based on the input x (and the fixed branching program B), compute a $|B| \times |B|$ matrix M_x whose $(u,v)^{\text{th}}$ entry is 1 if the branching program traverses from the vertex $u \in B$ to $v \in B$ on input x *in a single step*. In addition, for every sink $t \in B$ we set the $(t,t)^{\text{th}}$-entry of M_x to 1 (these correspond to self loops). All other entries of M_x are set to 0. Given input x, the matrix M_x (which is a permutation matrix) can be computed by a constant-depth circuit of size $\mathsf{poly}(|B|)$ (in fact, every entry in M_x is either a fixed constant, or equal to some variable or its negation).

Observe that for every $k \geq 1$, the $(u,v)^{\text{th}}$-th entry of $(M_x)^k$ is equal to 1 if and only if the branching program traverses from u to v, on input x, in k steps (or at most k steps if v is a sink). Hence, to check whether the source s leads to the (unique) *accepting* sink t on input x, it suffices to check whether the $(s,t)^{\text{th}}$-th entry of $(M_x)^{|B|}$ is equal to 1. Using repeated squaring we can compute $(M_x)^{|B|}$ in $O(\log^2(|B|))$ depth and we obtain a circuit as required.

C Proof of Lemma 4.8

We proceed to describe the $\mathcal{MAP}$, which is similar to the $\mathcal{MAP}$ of Theorem 1.1 except that we use the guaranteed property testers for the partial derivation languages. Given $x \in \mathcal{L}$, the $\mathcal{MAP}$ proof is the output $(\overline{I}, \overline{B})$ of `Generate-Intervals`(x,t) (see Fig. 2), where $t = n/k$ and as in the

proof of Theorem 1.1 we assume that $t \geq 2d$. The $\mathcal{MAP}$ verifier, given direct access to $(\overline{I}, \overline{B})$ and oracle access to $x \in \Sigma^n$, first runs $\mathcal{T}(\overline{I}, \overline{B})$ to obtain $(S_1, \langle \mathcal{L}_1 \rangle), \ldots, (S_\ell, \langle \mathcal{L}_\ell \rangle)$ and rejects if $\mathcal{T}$ rejects. Otherwise, the verifier runs the following procedure for every $j \in [\lceil \log_2(2/\varepsilon) \rceil]$:

1. Select uniformly at random $O\left(\frac{\log(1/\varepsilon)}{2^j \varepsilon}\right)$ indices in $[\ell]$. Denote the chosen indices by $\mathcal{I}$.
2. For every index $i \in \mathcal{I}$, run the property tester for $\mathcal{L}_i$ on input $x[S_i]$ (while simulating its oracle queries with queries to x), with respect to proximity parameter 2^{-j} and with completeness and soundness errors $\mathsf{poly}(\varepsilon)$ (as usual, the latter can be obtained by taking the majority of $O(\log(1/\varepsilon))$ independent tests). If the tester rejects then reject and halt.

If none of the above test fails then the verifier accepts.

We first show that completeness and soundness hold and later show that the query complexity is as stated.

Completeness. If $x \in \mathcal{L}$, by Lemma 4.3, the transformation $\mathcal{T}$ produces $((S_1, \langle \mathcal{L}_1 \rangle), \ldots, (S_k, \langle \mathcal{L}_k \rangle))$ such that $\mathcal{L}_j$ is a partial derivation language and $x[S_j] \in \mathcal{L}_j$, for every $j \in [k]$. Since the tester for each partial derivation language $\mathcal{L}_j$ has completeness error $\mathsf{poly}(\varepsilon)$ and we perform Step 2 $O(\varepsilon^{-1} \cdot \log^2(1/\varepsilon))$ times in total, the verifier accepts in all tests with probability at least $2/3$. Furthermore, if the testers for the partial derivation languages have a one-sided error, then the $\mathcal{MAP}$ verifier accepts with probability 1 and otherwise we can apply a generic transformation (as discussed in the beginning of the proof) to obtain a one-sided error.

Soundness. Let $x \in \Sigma^n$ that is ε-far from $\mathcal{L}$, and let $(\overline{I}, \overline{B})$ be an alleged proof. By Lemma 4.4, the transformation $\mathcal{T}$ either rejects (in which case the verifier rejects and we are done), or produces $((S_1, \langle \mathcal{L}_1 \rangle), \ldots, (S_k, \langle \mathcal{L}_\ell \rangle))$, where $S_1, \ldots, S_\ell$ form a partition of $[n]$ and $\mathcal{L}_j$ is a partial derivation language, such that x is ε-far from $\{z \in \Sigma^n : \forall j \in [k], z[S_j] \in \mathcal{L}_j\}$. The following claim, which is a refined averaging argument, shows that either there are many indexes $i \in [k]$ such $x[S_i]$ is mildly far from $\mathcal{L}_i$ or there are few indexes $i \in [\ell]$ such that $x[S_i]$ is extremely far from $\mathcal{L}_i$ (or anything in between).

Lemma C.1 (Precision Sampling)**.** *There exists $j^* \in [\lceil \log_2 2/\varepsilon \rceil]$ such that for a $\frac{2^{j^*}\varepsilon}{4 \cdot \lceil \log_2(2/\varepsilon) \rceil}$ fraction of the indexes $i \in [\ell]$ it holds that $x[S_i]$ is 2^{-j^*}-far from $\mathcal{L}_i$.*

For completeness, we provide the proof of Lemma C.1, which is standard.

Proof. Let $d \stackrel{\text{def}}{=} \lceil \log_2(2/\varepsilon) \rceil$. Recall that $\Delta(z, W)$ is the minimal *relative* Hamming distance of z from the set W. For every $k \in [d]$, let

$$B_k \stackrel{\text{def}}{=} \left\{ i \in [\ell] : \Delta(x[S_i], \mathcal{L}_i) \in \left(2^{-k}, 2^{-(k-1)}\right] \right\},$$

and let $B_{d+1} = [\ell] \backslash (\cup_{i \in [d]} B_k)$. Note that the sets $B_0, \ldots, B_d, B_{d+1}$ form a partition $[\ell]$. Also note that by our setting of d, for every $i \in B_{d+1}$ it holds that $x[S_i]$ is $\varepsilon/2$-close to $\mathcal{L}_i$.

Suppose towards a contradiction that for every $k \in [d]$ it holds that $|B_k| < \frac{2^k \varepsilon}{4d} \cdot \ell$. Using the fact that for every $i \in B_k$ it holds that $x[S_i]$ is $2^{-(k-1)}$-close to $\mathcal{L}_i$, we obtain that

$$\begin{aligned}
\Delta(x, \mathcal{L}) &\leq \frac{1}{\ell} \sum_{i=1}^{\ell} \Delta(x[S_i], \mathcal{L}_i) \\
&= \frac{1}{\ell} \sum_{i \in B_{d+1}} \Delta(x[S_i], \mathcal{L}_i) + \frac{1}{\ell} \sum_{k \in [d]} \sum_{i \in B_k} \Delta(x[S_i], \mathcal{L}_i) \\
&\leq \frac{|B_{d+1}|}{\ell} \cdot \frac{\varepsilon}{2} + \frac{1}{\ell} \sum_{k \in [d]} 2^{-(k-1)} \cdot |B_k| \\
&< \frac{\varepsilon}{2} + \sum_{k \in [d]} \frac{\varepsilon}{2d} \\
&= \varepsilon,
\end{aligned}$$

in contradiction to our assumption that x is ε-far from $\mathcal{L}$. □

Next, consider the execution of iteration j^* of the verifier, where j^* is as guaranteed by Lemma C.1. Since the verifier selects uniformly at random $O\left(\frac{\log(1/\varepsilon)}{2^{j^*}\varepsilon}\right)$ indices in $[k]$, with probability at least $9/10$ it selects at least one index $i \in [k]$ such that $x[S_i]$ is 2^{-j^*}-far from $\mathcal{L}_i$. In this case, the tester for $\mathcal{L}_i$, with respect to proximity parameter 2^{-j^*} will reject $x[S_i]$ with probability $1 - \mathsf{poly}(\varepsilon)$. Thus, the verifier rejects x with probability at least $(1 - \mathsf{poly}(\varepsilon)) \cdot 9/10 \geq 2/3$.

Query Complexity. Recall that we assumed that every partial derivation language has a tester with query complexity $Q(m, \delta) = O(m^\alpha \cdot \delta^{-\beta})$, for inputs of length m with respect to proximity parameter $\delta > 0$. By definition, it holds that $|S_i| \leq t = n/p$, for every $i \in [\ell]$. Thus, the query complexity is at most

$$\begin{aligned}
\sum_{j \in [\lceil \log_2 2/\varepsilon \rceil]} \sum_{i \in \mathcal{I}} \left(\log(1/\varepsilon) \cdot Q\left(n/k, 2^{-j}\right) \right) &= O\left(\log(1/\varepsilon) \cdot \sum_{j \in [\lceil \log_2 2/\varepsilon \rceil]} 2^{j\beta} \cdot \frac{\log(1/\varepsilon)}{2^j \cdot \varepsilon} \cdot (n/k)^\alpha \right) \\
&= O\left((n/k)^\alpha \cdot \frac{(\log(1/\varepsilon))^2}{\varepsilon} \cdot \sum_{j \in [\lceil \log_2 2/\varepsilon \rceil]} 2^{(\beta-1)j} \right) \\
&= O\left((n/k)^\alpha \cdot \varepsilon^{-\beta} \cdot \log^2(1/\varepsilon) \right).
\end{aligned}$$

For the particular case in which $\alpha = 0$, we tighten the analysis for small values of ε by noting that the query complexity for any language is upper bounded by the size of the object:

$$\begin{aligned}
\sum_{j \in [\lceil \log_2 2/\varepsilon \rceil]} \sum_{i \in \mathcal{I}} \left(\log(1/\varepsilon) \cdot Q\left(n/k, 2^{-j}\right) \right) &= O\left(\log(1/\varepsilon) \cdot \sum_{j \in [\lceil \log_2 2/\varepsilon \rceil]} \frac{\log(1/\varepsilon)}{2^j \cdot \varepsilon} \cdot \min\left(n/k, 2^{j\beta}\right) \right) \\
&= O\left((n/k)^{1-1/\beta} \cdot \varepsilon^{-1} \cdot \log^3(1/\varepsilon) \right),
\end{aligned}$$

where the last equality follows since $\min\left(n/k, 2^{j\beta}\right) \leq (n/k)^{1-1/\beta} \cdot \left(2^{j\beta}\right)^{1/\beta}$, for every $j \geq 1$ (while using the fact that $\beta \geq 1$). Note that $\log^3(1/\varepsilon) \leq \mathsf{polylog}(n)$ since without loss of generality we may assume that $\varepsilon \geq 1/n$.

D Efficient Verification for Special Context-Free Languages

As stated in Remark 4.6, in this section we show that for special context-free languages we can improve the running time of the verifier in Fig. 5 from $\mathsf{poly}(n, k, r)$ to $\mathsf{poly}(\log n, k, r)$. Specifically, we refer to context-free languages whose *partial derivation languages* are themselves context-free languages (e.g., the Dyck language, see Proposition 4.10).

The crucial step in improving the verifier's running-time is an efficient implementation of Item 4d in Fig. 3. In the general case, this step can be implemented in time $\mathsf{poly}(n)$, but we show that if the partial derivation languages are context-free languages, then we obtain running time $\mathsf{polylog}(n)$.

Lemma D.1. *For every context-free language $\mathcal{L}$ over an alphabet Σ, there exist an algorithm that given an integer $n \in \mathbb{N}$, runs in time $\mathsf{polylog}(n)$ and accepts if and only if $\mathcal{L} \cap \Sigma^n \neq \emptyset$.*

Proof. Let G be a context-free grammar that accepts $\mathcal{L}$, and let G' be the context-free grammar that is obtained from G by replacing all the terminal symbols in G by a single terminal symbol, denoted 0. Note that $\mathcal{L} \cap \Sigma^n \neq \emptyset$ if and only if G' accepts 0^n.

Observe that the language $\mathcal{L}'$ accepted by G' is a *unary* context-free language. Ginsburg and Rice [GR62] showed that such a language must be *regular*.

Proposition D.2 ([GR62])**.** *Every* unary *context-free language is* regular*.*

Hence, there exists a finite-state automaton over the unary alphabet that accepts $\mathcal{L}'$. Such an automaton can be viewed as a directed graph with a single outgoing edge from each node. Hence, the graph is a directed path (from the start node) of length a feeding into a directed cycle of length b, and some of the nodes are accepting. Hence, the accepted lengths have the form $j + i \cdot b$, where $j \in [a + b - 1]$ and $i \geq 0$.

The lemma follows by observing that an algorithm can easily check in $\mathsf{polylog}(n)$ time if the given input n has the desired form, by checking if $n - j$ is divisible by b, for the specific set of $j \in [a + b - 1]$ that correspond to accepting nodes of the automaton. □