Electronic Colloquium on Computational Complexity, Report No. 210 (2026)

Improved Pseudorandom Generators for Read-k Branching Programs

Dean Doron*
Ben Gurion University
deand@bgu.ac.il

Yonatan Lang*
Ben Gurion University
langy@post.bgu.ac.il

Abstract

We construct improved pseudorandom generators for read-k oblivious branching programs with a known reading sequence. For width-w branching programs over n variables, and designated error ε, our generator has seed length

$$\mathcal{O}\left(n^{1-\frac{1}{2k-1}}\log n\left(k\log w+\log\frac{n}{\varepsilon}\right)\right).$$

This improves upon the previous state-of-the-art due to Gurjar and Volk (ACM ToCT 2020), that has $1-\frac{1}{2^{k-1}}$ as the exponent of n (and a multiplicative factor of $\exp(k^2)$), whenever $k\geq 4$. In particular, whenever w and $1/\varepsilon$ are not too large, our seed length remains sublinear whenever $k=o(\log n/\log\log n)$, whereas the Gurjar and Volk's bound is sublinear only when $k=\mathcal{O}(\log\log n)$.

Our main conceptual contribution is a new way of modeling branching programs within the communication network of the classical INW generator of Impagliazzo, Nisan, and Wigderson (STOC 1994). Whereas nearly all applications of the INW generator model communication over a branching program using a simple path graph, we instead design a binary-tree communication network whose leaves correspond to the input variables, that allows substantially more efficient routing between variables that may be read multiple times. Toward this end, we identify a structural condition on the reading sequence under which routing the branching program's state through this network requires only $\mathcal{O}(k\log w)$ bits of communication per processor.

*The Stein Faculty of Computer and Information Science. Supported in part by Israel Science Foundation grant #857/25.

ISSN 1433-8092

Contents

1 Introduction

A *pseudorandom generator* (PRG) against a function class $\mathcal{C} \subseteq \{0,1\}^m \to \{0,1\}$ is a deterministic function that stretches a short uniform *seed* into m-bit strings, that *fools* any function from $\mathcal{C}$. Formally:

Definition 1.1 (pseudorandom generator)**.** *Let $\mathcal{C}$ be a class of Boolean functions on $\{0,1\}^m$. A function*

$$G\colon \{0,1\}^d \to \{0,1\}^m$$

is an ε-error PRG for $\mathcal{C}$ to if, for every $f \in \mathcal{C}$,

$$\left| \Pr_{s \leftarrow U_d}[f(G(s)) = 1] - \Pr_{x \leftarrow U_m}[f(x) = 1] \right| \leq \varepsilon.^{1}$$

The quantity d is called the seed length *of G.*

A central challenge in complexity theory is to construct explicit PRGs with short seed length for a broad range of function classes. Among the most extensively studied are those computed by various models of *branching programs* (BPs). This is largely motivated by the longstanding quest to understand the power of randomness in space-bounded computation.

Designing pseudorandom generators against BPs has received a lot of attention and is intimately connected to the question of understanding the power of randomness vs. space. The $\mathbf{BPL}$ vs. $\mathbf{L}$ problem asks whether every language decidable with bounded error in randomized logarithmic space is also decidable in deterministic logarithmic space.

One promising way to answers this question affirmatively and prove that $\mathbf{BPL} = \mathbf{L}$ is via designing a suitable PRG with logarithmic seed length. More specifically, the way that a space-bounded algorithm acts on its random bits can be modeled by a (layered, oblivious) *read-once branching program*. The seminal work of Nisan [Nis92] gave an ε-error PRG that fools length-n, width-w read-once BPs with seed length

$$\mathcal{O}\left(\log n \cdot \left(\log w + \log \frac{n}{\varepsilon}\right)\right).$$

For $w = \mathrm{poly}(n)$ and constant error, the setting that is required for derandomizing $\mathbf{BPL}$ via standard derandomization, this gives a seed length of $O(\log^2 n)$. Impagliazzo, Nisan, and Wigderson [INW94] subsequently introduced a second construction, which, although giving similar parameters to those in [Nis92], turned out to be more flexible, and will also be the main building block in the current paper. More broadly than read-once BPs, the INW generator is designed to fool *communication networks* (see Definition 3.2). Modeling a read-once branching program on a path graph communication network, their construction likewise yields seed length $\mathcal{O}(\log^2 n)$ for polynomial-width read-once branching programs.

Unfortunately, despite decades of substantial research, $O(\log^2 n)$ remains the best known seed length for polynomial-width read-once BPs, yet the quest for better PRGs led to substantial advances in other settings.

- *Weaker models*, such as constant-width read-once BPs (e.g., [BDVY13, MRT19]), regular and permutation BPs (e.g., [De11, BRRY14, HPV21]), and combinatorial rectangles and related models (e.g., [GMR+12, MZ13, GKM18]).

[1] Here and throughout, for an integer s, U_s denotes the uniform distribution over $\{0,1\}^s$.

- *Stronger models*, such as unknown-order read-once BPs [FK18], and read-many BPs [GV20].

Our work focuses on PRGs for read-many oblivious BPs.

Definition 1.2 (see also Definition 3.1)**.** *A width-w (oblivious) read-k BP on n input bits is a directed graph whose vertices (called "states") are partitioned into $nk+1$ layers, each containing at most w states. The first layer contains a designated start state, and the states of the last layer are labeled either "accept" or "reject".*

Each layer V_t (but the last one) is associated with an input bit $i_t \in [n]$, and every input bit appears exactly k times. From every state in a non-final layer, we have two outgoing edges,[2] labeled 0 and 1, leading to the next layer. On input $x \in \{0,1\}^n$, the computation begins at the start state and follows the edges labeled x_{i_t}. The program accepts *x if the computation ends in an accepting state.*

The case of $k = 1$ corresponds to read-once BPs. Beyond its potential to drive progress on read-once BPs, the study of pseudorandomness for read-k BPs is interesting in its own right: these programs are substantially more powerful than their read-once counterparts. For example, Bollig, Sauerhoff, Sieling, and Wegener [BSSW98] showed that there exists a Boolean function computable by a polynomial-width branching program over a read-2 permutation sequence (see Definition 2.2) in which the two permutations are identical, while every read-once branching program computing the same function requires subexponential width.

The increased computational power of read-k branching programs is reflected in the difficulty of constructing PRGs for them. While a probabilistic argument shows that there exists a PRG with seed length $\mathcal{O}(\log(nwk/\varepsilon))$ that ε-fools this model, explicit constructions, however, remain far from this existential bound. Prior to this work, the best-known PRG for read-k branching programs with a known reading sequence was due to Gurjar and Volk [GV20], whose generator has seed length

$$\mathcal{O}\left(\exp(k^2)\cdot n^{1-\frac{1}{2^k-1}}\cdot \log n\cdot\left(k\log w+\log\frac{n}{\varepsilon}\right)\right).$$

Our main result improves the dependence on k for $k \geq 4$: we replace the 2^{k-1} term in the exponent of n by $2k-1$, while also eliminating the $\exp(k^2)$ multiplicative factor.

Theorem 1.3 (main result, see Theorem 4.4)**.** *Given a read-k sequence over n variables, there is an explicit[3] PRG that ε-fools every width-w branching program with this reading sequence, with seed length*

$$d=\mathcal{O}\left(n^{1-\frac{1}{2k-1}}\log n\left(k\log w+\log\frac{n}{\varepsilon}\right)\right).$$

For the more restricted case in which the reading sequence is a sequence of permutations (see Definition 2.2), we obtain an improved seed length, replacing the $1-\frac{1}{2k-1}$ term with $1-\frac{1}{k}$.

The improvement of Theorem 1.3 compared to [GV20] becomes particularly significant when k is allowed to grow with n. For polynomial width and inverse-polynomial error, the previous bound of Gurjar and Volk is sublinear in n only for $k=\mathcal{O}(\log\log n)$, whereas our bound remains sublinear for

$$k=o\left(\frac{\log n}{\log\log n}\right).$$

[2] In this paper we focus on BPs over a binary alphabet.

[3] By "explicit", we mean that the reading sequence can be preprocessed in time polynomial in its length to obtain a description of the generator. Given read-only access to this description, and a seed, the generator can be evaluated in time polynomial in the lengths of the reading sequence and seed, using working space linear in the seed length. See Theorem 4.4 and Lemma 4.5.

This larger range also yields an improved generator for arbitrary oblivious branching programs of linear length. As in [GV20], we sample frequently read variables uniformly at random, and apply our read-k generator to the remaining variables.

Corollary 1.4 (informal; see Theorem 4.6)**.** *There is an explicit PRG that fools, to inverse-polynomial error, length-$\mathcal{O}(n)$ read-many branching program over n variables and polynomial width, with seed length*

$$\mathcal{O}\left(\frac{n \log\log n}{\log n}\right).$$

This improves the $\mathcal{O}(n/\log\log n)$ seed length obtained by Gurjar and Volk [GV20] for the same setting.

1.1 Our Technique

Our technique follows a similar high-level approach to that of Gurjar and Volk [GV20]: we partition the variable set into sufficiently large subsets and construct a PRG per subset.

More concretely, we consider a branching program whose reading sequence S is over a variable set X, where S specifies the order in which the program reads its variables. For a subset $X' \subseteq X$, we denote by $S|_{X'}$ the restriction of S to X', obtained by deleting from S all occurrences of variables outside X'. Our construction partitions

$$X = X_1 \sqcup \cdots \sqcup X_\ell$$

and, for each $i \in [\ell]$, constructs a PRG that fools branching programs whose reading sequence is $S|_{X_i}$ and has seed length $\mathcal{O}(\log n(k \log w + \log(n/\varepsilon)))$. We sample their seeds independently and combine their outputs to obtain a single assignment to the variables in X. Using a standard hybrid argument, one can show that the resulting distribution fools the original branching program, at the cost of a corresponding loss in the error parameter ε.

Since we choose the seeds independently, we seek a partition into subsets that are *as large as possible*, while ensuring that each restricted reading sequence $S|_{X_i}$, for each $i \in [\ell]$, is *sufficiently structured* for the corresponding branching programs to be fooled efficiently. Gurjar and Volk obtain their partition using a result of Anderson et al. [AFS+16], based on repeated applications of the Erdős–Szekeres theorem together with additional structural arguments. We obtain our partition by a different method. Our method results in subsets that are *larger* (thereby reducing the number of subsets in the partition), and in order to retain small seed bound for each individial PRG, we introduce a new application of the INW generator.

The INW generator. The INW generator fools protocols on communication networks using a short seed when both the *communication cost* of the protocol and the *width* of the network are small. Informally, this means that the network can be recursively split into balanced parts by removing only a few vertices, while each processor sends and receives only a small amount of information in total during the protocol (see also Theorem 3.6).

Nearly all previous applications of the INW generator (implicitly) model a read-once branching program using a path graph communication network.[4] This is particularly natural in the read-once setting: the path graph has width 1, while the branching program can still be simulated

[4] In fact, we were able to find only one other paper that implicitly uses INW with a different topolpgy, [KKL+21].

by a c-protocol on a network with this topology with essentially optimal communication cost, $c = \log w$, where w is the branching programs width. Each vertex in the path graph corresponds to a different variable from the variable set and their ordering is exactly that of the reading sequence. The communication cost stems from the fact that up to $\log w$ bits of information are required to represent the state of the branching program at any given time.

For read-k branching programs, however, this topology quickly becomes ineffective. There are read-2 branching programs for which modeling the branching program on the path graph similarly, regardless of how the vertices are ordered, requires communication cost $c = \Omega(n)$, making it essentially useless.[5]

A tree topology. We introduce a different topology for simulating branching programs: a binary tree. More specifically, we use a binary tree whose leaves correspond to the variables, while its internal vertices facilitate communication between the leaves. This topology, like the path graph, has constant width,[6] while also allowing information to flow through the network more efficiently. In order to traverse the graph from one leaf to another we require $\mathcal{O}(\log n)$ steps, whereas in the path topology we required $\mathcal{O}(n)$. To model a branching program on this communication network, we pass information (the current state of the branching program) from one leaf to another via the shortest path route in the order of the reading sequence.

To keep the communication cost of this protocol low, we need each vertex in the binary tree to be visited only $\mathcal{O}(k)$ times. We achieve this through the notions of *alternating sequences* (see Definition 2.3) and *recursively alternating sequences* (see Definition 2.5). Informally, an alternating sequence admits a partition of its variables into two nearly equal parts and consists of contiguous *clusters* that alternate between these parts. Each cluster contains only variables from its corresponding part, with every variable in that part appearing the same number of times. A sequence is recursively alternating if this structure holds recursively within the restriction to each part, down to individual variables. These successive partitions define the communication tree and ensure that whenever the walk enters a subtree, it reads every variable in that subtree before leaving. Since each variable is read only k times, this bounds the number of crossings of each edge, and hence the number of visits to each vertex, by $\mathcal{O}(k)$.

In Section 2, we show that every read-k sequence S over a variable set X of size n admits a subset $X' \subseteq X$ of size $\Omega(n^{\frac{1}{2k-1}})$ such that $S|_{X'}$ is recursively alternating. For *permutation sequences* (see Definition 2.2), which consist of a concatenation of k permutations of X, we exploit this additional structure to obtain the stronger bound $\Omega(n^{\frac{1}{k}})$ on the size of X'.

Then, in Section 3 we show how to construct a PRG for branching programs over sequences that are recursively alternating. The construction is based on the INW generator and is very natural, as it uses the same *partition tree* (see Definition 2.4) of the recursively alternating reading sequence as the topology of the communication network used to model the branching program.

[5]For example, make the reading sequence correspond to an Euler tour of a 4-regular expander, omitting the last vertex. Each variable appears exactly twice in the resulting reading sequence. Removing the middle edge of a path graph modeling the branching program splits the variable set to two sets with a large cut in the original expander, implying the removed edge must be crossed $\Omega(n)$ times in order to model the branching program.

[6]Moreover, adding edges between consecutive leaves in their left-to-right order forms a path along the "floor" of the tree. The resulting graph has treewidth at most 3 and hence still has constant *width*. Thus, this augmented topology contains the path graph topology used in the previous construction while providing additional communication routes through the tree. Our construction does not require these additional edges, and we therefore omit them.

Finally, in Section 4 we combine these PRGs into one PRG for the original branching program, using the aforementioned hybrid argument.

2 Sequences

We say that a sequence S is "over a variable set X" if its items are variables from X, and we denote by S_i the i'th item in S.

A major part of our proof consists of finding subsequences within some sequence by restricting the variable set. For that, we remind the reader that we denote by $S|_{X'}$, for some subset $X' \subseteq X$, the sequence obtained by restricting the sequence S to variables in X'. That is, $S|_{X'}$ is the sequence we would get by deleting every variable not in X' from S.

For a sequence S over the variable set X, the sequence $S^{(i)}$ is the sequence we obtain by restricting S to only the i'th appearance of every variable in X (if it exists). Notice that if all variables in X appear in S at least i times, $S^{(i)}$ is a permutation of X.

2.1 Branching Program Sequences

Branching programs (see Definition 3.1) use a predefined sequence over their variable set. A rather natural and well-studied way to categorize such sequences is by the number of times each variable appears in the sequence.

Definition 2.1 (read-k sequence)**.** *Let $k \geq 1$ be an integer and let $X = \{x_1, \ldots, x_n\}$ be a finite nonempty variable set. A read-k sequence over X is a sequence of length nk in which every variable appears exactly k times.*

We call S a "read sequence" and omit k if it is a read-k sequence for some unknown or irrelevant k.

Notice that branching programs over sequences where each variable appears *at most* k times can be turned into branching programs over read-k sequences by padding the sequence and adding trivial transitions.

In this paper we also discuss a more specific type of read-k sequence, namely one that consists of a concatenation of permutations of the variable set.

Definition 2.2 (read-k permutation sequence)**.** *A read-k sequence S over the variable set $X = \{x_1, ..., x_n\}$ is called a* permutation sequence *if it can be written as a concatenation*

$$S = P_1 P_2 \cdots P_k$$

where each P_i is a permutation of X.

The following definitions will be useful for our construction.

Definition 2.3 (alternating sequence)**.** *Let S be a read-k sequence over a variable set X, and let $X = A \sqcup B$ be a balanced [7] partition of X. We say that S is (A, B)-*alternating *if S can be written as a concatenation of "clusters"*

$$S = C_1 C_2 \cdots C_m,$$

[7] The partition is balanced if $|A|, |B| \leq \lceil \frac{n}{2} \rceil$.

such that for every odd i, the cluster C_i is a read-k_i sequence over the variable set A, and for every even i, the cluster C_i is a read-k_i sequence over the variable set B, for some integer $1 \leq k_i \leq k$. We call these clusters an "alternating decomposition" of S.

Notice the number of clusters, m, is at most $2k$. We say that S is alternating and omit (A, B) when it is clear what they are, or irrelevant.

Definition 2.4 (partition tree)**.** *Let X be a finite nonempty set and let $d \geq 0$ be an integer. A partition tree for X with depth parameter d is defined recursively as follows.*

- *If $d = 0$ or $|X| = 1$, the tree consists of a single vertex labeled by X.*
- *Otherwise, its root is labeled by X, and its two children are the roots of partition trees with depth parameter $d - 1$ for sets A and B, where $X = A \sqcup B$ is a balanced partition into nonempty sets.*

The resulting tree has depth at most d. Every leaf is either at depth d or labeled by a singleton set.

Definition 2.5 (recursively alternating sequence)**.** *Let S be a read-k sequence over a variable set X, and let $d \geq 0$ be an integer.*

*We say that S is d-*recursively alternating *if there exists a partition tree T for X with depth parameter d such that, for every internal vertex v of T, the following holds. If v is labeled by Y and its two children are labeled by Y_0 and Y_1, then*

$$S|_Y$$

is (Y_0, Y_1)-alternating.

Since every partition in T is balanced, when

$$d \geq \lceil \log |X| \rceil ,$$

all leaves of T are labeled by singleton sets. In this case, we simply say that S is recursively alternating*.*

In particular, every read sequence over a singleton variable set is recursively alternating, witnessed by the one-vertex partition tree.

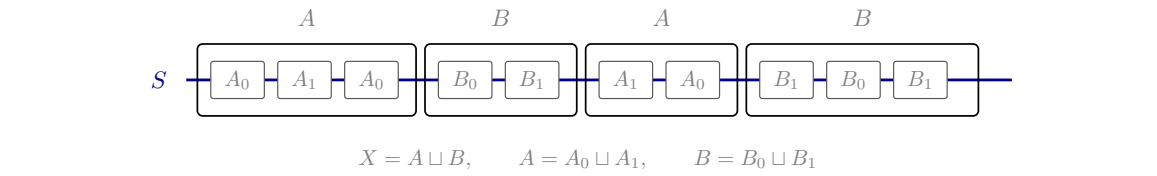

Figure 1: An example of the structure of a 2-recursively alternating sequence. At the first level, S alternates between clusters over A and B. At the second level, $S|_A$ alternates between clusters over A_0 and A_1, while $S|_B$ alternates between clusters over B_0 and B_1. The different top-level clusters need not have the same internal structure.

Figure 1 shows an example for what the structure of a 2-recursively alternating sequence could look like. Notice, that not only do $S|_A$ and $S|_B$ alternate, but each A and B cluster is an alternating sequences on its own. This is no coincidence, it is an inherit attribute of d-recursively alternating sequences, and later on in this section we prove it for the case of recursively alternating sequences (see Lemma 2.14).

2.2 Recursively Alternating Sequences From Permutation Sequences

Consider a permutation sequence S over a variable set X. Our goal is to find a sufficiently large subset $X' \subseteq X$ such that $S|_{X'}$ recursively alternates. We begin by first finding a subset $X' \subseteq X$ such that $S|_{X'}$ alternates.

Lemma 2.6. *Let S be a read-k permutation sequence with variable set X, where $|X| = n \geq 2$. Then there exist subsets $A, B \subseteq X$, of sizes*

$$|A|, |B| \geq \left\lfloor \frac{n}{2^k} \right\rfloor,$$

such that $S|_{A \cup B}$ is (A, B)-alternating and both $S|_A$ and $S|_B$ are permutation sequences over their respective variable sets.

Additionally, these subsets can be computed in $\mathcal{O}(kn)$ time.

Proof: We begin by partitioning X into $A \sqcup B$, where A consists of the first $\lfloor \frac{n}{2} \rfloor$ variables in $S^{(1)}$ and B consists of the remaining variables. Thus, in $S^{(1)}$, all variables of A precede all variables of B.

Now consider $S^{(2)}$. Its first half contains at least $\lfloor \frac{n}{4} \rfloor$ variables from either A or B. Assume without loss of generality that it contains at least $\lfloor \frac{n}{4} \rfloor$ variables from A. Then its remaining half contains at least $\lfloor \frac{n}{4} \rfloor$ variables from B. We restrict A to variables appearing in the first half of $S^{(2)}$, and B to variables appearing in the second half.

After this restriction,

$$S^{(1)}|_{A \cup B}\, S^{(2)}|_{A \cup B}$$

is (A, B)-alternating, and

$$|A|, |B| \geq \left\lfloor \frac{n}{4} \right\rfloor.$$

More generally, suppose that after considering $S^{(1)}, \ldots, S^{(d)}$, for some $d < k$, we have disjoint sets A, B such that

$$|A|, |B| \geq \left\lfloor \frac{n}{2^d} \right\rfloor$$

and

$$S^{(1)}|_{A \cup B} \cdots S^{(d)}|_{A \cup B}$$

is (A, B)-alternating.

Consider $S^{(d+1)}|_{A \cup B}$. Its first half contains at least half of the variables of either A or B. If it contains at least half of the variables of A, we restrict A to those variables and restrict B to the variables appearing in the second half. Otherwise, we do the opposite.

This preserves the (A, B)-alternation of the previous permutations and makes $S^{(d+1)}|_{A \cup B}$ consist of one A-block and one B-block. Moreover, the sizes of both sets decrease by at most a factor of 2. Repeating this process for all k permutations yields

$$|A|, |B| \geq \left\lfloor \frac{n}{2^k} \right\rfloor$$

and $S|_{A \cup B}$ is (A, B)-alternating.

Finally, each $S^{(i)}|_A$ is a permutation of A, and each $S^{(i)}|_B$ is a permutation of B. Hence $S|_A$ and $S|_B$ are permutation sequences over A and B, respectively.

The construction is deterministic and runs in $\mathcal{O}(kn)$ time. Each of the k steps scans the corresponding permutation and restricts A and B, taking $\mathcal{O}(n)$ time. A final scan of S constructs the restricted sequences $S|_A$ and $S|_B$, also in $\mathcal{O}(kn)$ time.

■

We can now use this lemma to construct sets such that the restricting the sequence to them produces recursively alternating sequence.

Lemma 2.7. *Let S be a read-k permutation sequence over the variable set X, where $|X| = n$. Then there exists a subset $X' \subseteq X$ of size*

$$|X'| = \Omega\left(n^{1/k}\right)$$

such that $S|_{X'}$ is recursively alternating.

Additionally, this subset can be computed in $\mathcal{O}(kn)$ time.

Proof: Lemma 2.6 lets us find disjoint subsets $A, B \subseteq X$ such that $S|_{A\cup B}$ is (A, B)-alternating and

$$|A|, |B| \geq \left\lfloor \frac{n}{2^k} \right\rfloor.$$

Additionally, $S|_A$ and $S|_B$ are both permutation sequences. Therefore, we can apply Lemma 2.6 separately to each of them. We show that repeating this process for

$$d = \left\lfloor \frac{\log n}{k} \right\rfloor$$

levels and taking one variable from each of the resulting 2^d subsets produces a recursively alternating sequence of the desired size.

We begin with the size analysis. At each level, we apply Lemma 2.6 to every $S|_Y$, where Y is a subset selected at the previous level. We select two disjoint subsets of Y, each of size at least

$$\left\lfloor \frac{|Y|}{2^k} \right\rfloor.$$

Thus, after d levels, we have 2^d subsets, each of size at least

$$\left\lfloor \frac{n}{2^{dk}} \right\rfloor \geq 1.$$

Choosing one variable from each subset gives a set X' satisfying

$$|X'| = 2^d \geq \frac{1}{2} n^{1/k} = \Omega(n^{1/k}).$$

We now show that $S|_{X'}$ is recursively alternating. Let T be the complete binary tree of depth d recording the selection process. Its root is labeled by X, and the children of each internal vertex labeled by Y are labeled by the two selected subsets $Y_A, Y_B \subseteq Y$. Recall that X' contains exactly one variable chosen from each leaf of T.

Let T' be obtained from T by replacing each label Y with $Y \cap X'$. Every variable of $Y \cap X'$ was chosen from a leaf below one of the two children of Y, so

$$Y \cap X' = (Y_A \cap X') \sqcup (Y_B \cap X').$$

Moreover, the two child subtrees have the same number of leaves, and we choose exactly one variable from each leaf. Hence

$$|Y_A \cap X'| = |Y_B \cap X'|.$$

Thus, T' is a partition tree for X', and all its partitions are balanced.

By construction, for each internal vertex of T labeled by Y with children labeled by Y_A, Y_B, the sequence $S|_{Y_A \cup Y_B}$ is (Y_A, Y_B)-alternating. Restricting this sequence further to X' shows that

$$S|_{Y \cap X'}$$

is $(Y_A \cap X', Y_B \cap X')$-alternating: each cluster remains a read sequence over the retained variables of its corresponding part.

Therefore, $S|_{X'}$ is d-recursively alternating, witnessed by T'. Since

$$d = \log |X'| = \left\lceil \log |X'| \right\rceil,$$

it follows that $S|_{X'}$ is recursively alternating.

The construction can be implemented deterministically in $\mathcal{O}(kn)$ time. For $k \geq 2$, we trim each selected child set of a node labeled Y to exactly $\lfloor \frac{|Y|}{2^k} \rfloor$ variables, preserving alternation and the size guarantee. The total cardinality of the sets processed at each successive level therefore decreases by a factor of at least $2^{k-1} \geq 2$. Since processing a set Y, including constructing its children's restricted sequences, takes $\mathcal{O}(k|Y|)$ time, summing over all levels gives $\mathcal{O}(kn)$ time. For $k = 1$, we simply take $X' = X$. ∎

2.3 Recursively Alternating Sequences From Arbitrary Sequences

2.3.1 Variable Sequences and Coloring

Another notion of sequences we use are the ones produced when looking at the indices of certain variables within a read-k sequence S.

Definition 2.8. *For a variable $x \in X$ in a read-k sequence S, we denote by $P_S^i(x)$ the position of the i'th appearance of x in S[8]. We define $S(x)$ to be the increasingly monotone sequence of positions of x in S. That is,*

$$S(x) = (P_S^1(x), P_S^2(x), ..., P_S^k(x)).$$

We define $S(x, y)$, for two distinct variables $x, y \in X$, to be the increasingly monotone merged sequence of $S(x)$ and $S(y)$.

The merged sequence of two variables will be particularly helpful for us in proving the existence of subsets $X' \subseteq X$ that alternate. To understand how, we introduce the concept of coloring sequences.

Definition 2.9 (sequence coloring)**.** *For a pair of distinct variables $x, y \in X$ we define the coloring of $S(x, y)$ as a string c in $\{0, 1\}^{2k}$ with weight k such that*

$$c_i = 1 \iff S_{S(x,y)_i} = y$$

for all $1 \leq i \leq 2k$. That is, c has 1's in indices that correspond to the positions of y, and 0's in the positions of x. We denote this coloring as $\chi_S(x, y)$.

[8] if x appears in S less than i times, let it be ∞.

Notice, for example, that $\chi_S(x,y) = \overline{\chi_S(y,x)}$. The following lemma connects sequence coloring to alternating sequences.

Claim 2.10. *Let S be a read-k sequence over the variable set X. Let $A, B \subseteq X$ be disjoint, nonempty and balanced*[9] *subsets such that for any $a \in A$ and $b \in B$, $\chi_S(a,b)$ is the same color, say*

$$\chi_S(a,b) = \widetilde{c}.$$

Then $S|_{A\cup B}$ is either (A,B) or (B,A) alternating.

Proof: We show that $S|_{A\cup B}$ consists of clusters that correspond to runs of 0's and 1's in $\widetilde{c}$. Write

$$\widetilde{c} = c_1 c_2 \cdots c_m.$$

Assume first that odd-indexed runs consist of 0's and even-indexed runs consist of 1's. Let $k_i = |c_i|$.

We claim that the first $|A| \cdot k_1$ items in $S|_{A\cup B}$ form a read-k_1 sequence over A. If a variable from B appears among these items, let b be the first such variable to appear. Then some variable $a \in A$ occurs fewer than k_1 times before this occurrence of b. Thus $\chi_S(a,b)$ begins with fewer than k_1 0's, contradicting $\chi_S(a,b) = \widetilde{c}$.

Moreover, these first $|A| \cdot k_1$ items must contain every variable in A exactly k_1 times. Otherwise, some variable appears more than k_1 times, and its color with any variable in B begins with more than k_1 0's, again a contradiction.

Therefore we can write

$$S|_{A\cup B} = S_1 \cdot S',$$

where S_1 is a read-k_1 sequence over A and S' is the remainder of $S|_{A\cup B}$. In every two-variable restriction, deleting S_1 removes precisely the initial run c_1. We can therefore repeat the same argument for the remaining nonfinal runs, alternating the roles of A and B. When only c_m remains, the remaining suffix is a read-k_m sequence over the corresponding side. Since $m \le 2k$, this shows that $S|_{A\cup B}$ is an (A,B)-alternating sequence.

If instead odd-indexed runs consist of 1's and even-indexed runs consist of 0's, the same argument with A and B interchanged shows that $S|_{A\cup B}$ is a (B,A)-alternating sequence. The claim follows. ∎

The following claim will be useful in our construction of alternating sequences.

Claim 2.11 (median separation)**.** *Let A and B be disjoint, finite, nonempty subsets of $\mathbb{R}$. Then, there exist subsets $A' \subseteq A$ and $B' \subseteq B$ such that*

$$|A'| \ge \frac{|A|}{2}, \qquad |B'| \ge \frac{|B|}{2},$$

and either

$$\forall a \in A', \forall b \in B' : a < b,$$

or

$$\forall a \in A', \forall b \in B' : b < a.$$

The proof follows from comparing the median of both sets.

[9] That is, $||A| - |B|| \le 1$.

Proof: For each of the sets, let its median be the element of rank $\lceil |A|/2 \rceil$ and $\lceil |B|/2 \rceil$, respectively, in increasing order. Denote these medians by m_A and m_B. Suppose first that $m_A < m_B$. Let

$$A' = \{a \in A : a \leq m_A\} \qquad \text{and} \qquad B' = \{b \in B : b \geq m_B\}.$$

By the definition of the medians,

$$|A'| \geq \frac{|A|}{2} \qquad \text{and} \qquad |B'| \geq \frac{|B|}{2}.$$

Moreover, for every $a \in A'$ and $b \in B'$,

$$a \leq m_A < m_B \leq b,$$

and hence $a < b$. If $m_B < m_A$, we instead take

$$A' = \{a \in A : a \geq m_A\} \qquad \text{and} \qquad B' = \{b \in B : b \leq m_B\}.$$

The same argument shows that both subsets have at least half the size of their respective sets and that $b < a$ for every $a \in A'$ and $b \in B'$. ■

We now show how to create subsets that alternate.

Lemma 2.12. *Let S be a read-k sequence over the variable set X, where $|X| = n \geq 2$. Then there exist a subset $X' \subseteq X$, size*

$$|X'| \geq \left\lfloor \frac{n}{2^{2k-2}} \right\rfloor,$$

such that $S|_{X'}$ is alternating.

Additionally, this subset can be computed in $\mathcal{O}(kn)$ time.

Proof: We show that there exist disjoint, nonempty, balanced subsets $A, B \subseteq X$ and a color $\widetilde{c}$ such that

$$|A \cup B| \geq \left\lfloor \frac{n}{2^{2k-2}} \right\rfloor$$

and, for every $a \in A$ and $b \in B$,

$$\chi_S(a, b) = \widetilde{c}.$$

The result then follows from Claim 2.10.

We begin by letting A be the set of variables whose first appearances form the first $\lfloor n/2 \rfloor$ positions in $S^{(1)}$, and let $B = X \setminus A$. Thus, for every $a \in A$ and $b \in B$,

$$P_S^1(a) < P_S^1(b),$$

and hence the first bit of $\chi_S(a, b)$ is 0.

If $k = 1$, then S is already (A, B)-alternating, so taking $X' = X = A \cup B$ proves the claim. Henceforth assume $k \geq 2$.

We next compare the second appearances of variables in A with the first appearances of variables in B. Consider the sets

$$A' = \{P_S^2(a) : a \in A\} \qquad \text{and} \qquad B' = \{P_S^1(b) : b \in B\}.$$

By Claim 2.11, we may restrict A and B, losing at most half of each set, so that either

$$P_S^2(a) < P_S^1(b)$$

for every $a \in A$ and $b \in B$, or

$$P_S^1(b) < P_S^2(a)$$

for every $a \in A$ and $b \in B$. To keep the sets balanced, let p and q denote their sizes before the restriction, and trim the retained sets to sizes $\lceil p/2 \rceil$ and $\lceil q/2 \rceil$, respectively. These sizes are available by Claim 2.11. Since $|p - q| \leq 1$, the resulting sizes also differ by at most one. We use this trimming rule after every comparison.

In the first case,

$$P_S^1(a) < P_S^2(a) < P_S^1(b),$$

so the second bit of every $\chi_S(a, b)$ is 0. In the second case,

$$P_S^1(a) < P_S^1(b) < P_S^2(a),$$

so the second bit of every $\chi_S(a, b)$ is 1.

We continue in the same manner. Suppose that, for some $0 \leq r, s \leq k$, the first r appearances of every variable in A and the first s appearances of every variable in B have already been ordered uniformly in every merged sequence $S(a, b)$. We compare the next undecided appearances by applying Claim 2.11 to

$$\{P_S^{r+1}(a) : a \in A\} \qquad \text{and} \qquad \{P_S^{s+1}(b) : b \in B\}.$$

After restricting A and B, losing at most half of each set, either

$$P_S^{r+1}(a) < P_S^{s+1}(b)$$

for every $a \in A$ and $b \in B$, or

$$P_S^{s+1}(b) < P_S^{r+1}(a)$$

for every $a \in A$ and $b \in B$.

In the first case, the $(r + 1)$-st appearance of a is the next appearance in every merged sequence $S(a, b)$. Thus, the next bit of $\chi_S(a, b)$ is 0, and we increase r by one. In the second case, the $(s + 1)$-st appearance of b is the next appearance in every merged sequence. Thus, the next bit is 1, and we increase s by one.

We repeat this process until either $r = k$ or $s = k$. Once $r = k$, all appearances of every variable in A have already occurred, so all remaining appearances belong to variables in B. Similarly, if $s = k$, all remaining appearances belong to variables in A. Consequently, the remaining bits of $\chi_S(a, b)$ are determined without any further restriction, and there is a single color $\widetilde{c}$ such that

$$\chi_S(a, b) = \widetilde{c}$$

for every $a \in A$ and $b \in B$. After fixing the first bit, at most $2k - 2$ further comparisons are required before one of r and s reaches k. Each comparison reduces both A and B by at most a factor of 2. Consequently,

$$|A \cup B| \geq \left\lfloor \frac{n}{2^{2k-2}} \right\rfloor.$$

By Claim 2.10, $S|_{A\cup B}$ is (A, B)-alternating. Therefore, setting

$$X' = A \cup B$$

completes the proof.

Notice this construction is deterministic and runs in $\mathcal{O}(kn)$ time. We first compute all occurrence positions $P_S^i(x)$ in $\mathcal{O}(kn)$ time. Each of the at most $2k-2$ median-separation steps takes $\mathcal{O}(n)$ time, using deterministic linear-time selection to find the medians and scanning the current sets to retain the appropriate variables. Constructing the restricted sequence $S|_{X'}$ also takes $\mathcal{O}(kn)$ time. ■

This construction of alternating subsets allows us, like in Section 2.2, to construct subsets that recursivly alternate.

Lemma 2.13. *Let S be a read-k sequence over the variable set X, where $|X| = n$. Then there exists a subset $X' \subseteq X$ of size $\Omega(n^{\frac{1}{2k-1}})$ such that $S|_{X'}$ is recursively alternating.*

Additionally, this subset can be computed in $\mathcal{O}(kn)$ time.

Proof: We repeatedly apply Lemma 2.12. At each step, for every current variable set Y, the lemma gives disjoint subsets $Y_A, Y_B \subseteq Y$ such that

$$|Y_A|, |Y_B| \geq \left\lfloor \frac{|Y|}{2^{2k-1}} \right\rfloor$$

and

$$S|_{Y_A \cup Y_B}$$

is (Y_A, Y_B)-alternating. We then continue the construction separately inside Y_A and Y_B. We repeat this process for

$$d = \left\lfloor \frac{\log n}{2k-1} \right\rfloor$$

levels, and then choose one variable from each of the resulting 2^d sets.

We first analyze the size of the resulting set. After d levels, every resulting set has size at least

$$\left\lfloor \frac{n}{2^{d(2k-1)}} \right\rfloor \geq 1.$$

Thus, choosing one variable from each set gives a set X' satisfying

$$|X'| = 2^d \geq \frac{1}{2} n^{\frac{1}{2k-1}}.$$

It remains to show that $S|_{X'}$ is recursively alternating. Let T be the tree describing the construction: its root is labeled by X, and whenever the construction applies Lemma 2.12 to a set Y, its two children are labeled by the resulting sets Y_A and Y_B. The leaves of T are the nonempty sets obtained after d levels.

Let T' be the tree obtained from T by replacing every label Y with

$$Y' = Y \cap X'.$$

Since exactly one variable was chosen from each leaf of T, the two children of every internal vertex of T' have the same number of variables. Hence every partition in T' is balanced.

Moreover, if an internal vertex of T is labeled by Y and its children are labeled by Y_A and Y_B, then $S|_{Y_A \cup Y_B}$ is (Y_A, Y_B)-alternating. Restricting this sequence further to X' shows that

$$S|_{(Y_A \cap X') \cup (Y_B \cap X')}$$

is $(Y_A \cap X', Y_B \cap X')$-alternating. Therefore, T' witnesses that $S|_{X'}$ is d-recursively alternating. Finally, T' has $2^d = |X'|$ singleton leaves and depth

$$d = \log |X'| = \lceil \log |X'| \rceil .$$

Thus, $S|_{X'}$ is recursively alternating.

The runtime analysis follows Lemma 2.7. For $k \geq 2$, we now trim each child of Y to $\left\lfloor \frac{|Y|}{2^{2k-1}} \right\rfloor$ variables, so the total cardinality decreases by a factor of at least $2^{2k-2} \geq 4$ per level. Since processing Y takes $\mathcal{O}(k|Y|)$ time by Lemma 2.12, the same summation over levels gives deterministic $\mathcal{O}(kn)$ time. The case $k = 1$ is handled as before. ■

We end this section with a useful lemma. Consider a recursively alternating sequence S and an internal vertex v with children v_A, v_B and a sibling u labeled Y, Y_A, Y_B and X respectively. Now consider the alternating decomposition of $S|_{Y \cup X}$

$$S|_{Y \cup X} = C_1 C_2 \cdots C_m,$$

and, assume without loss of generality, that the odd clusters correspond to Y. That is, each odd cluster is a read sequence over Y and

$$S|_Y = C_1 C_3 \cdots .$$

We know, since S recursively alternates, that $S|_Y$ is (Y_A, Y_B)-alternating.

But is every odd cluster in this decomposition also alternating on its own? The following lemma shows that yes, every one of those Y associated clusters is, on its own, an alternating sequence. It is either (Y_A, Y_B)-alternating or (Y_B, Y_A)-alternating.

Lemma 2.14 (inheritance lemma)**.** *Let $S = S_1 S_2 \cdots S_m$ be a sequence over a variable set X, where each S_i is a read-k_i sequence over X. If S is (A, B)-alternating, then every S_i is either (A, B)-alternating or (B, A)-alternating.*

Proof: Let

$$S = C_1 C_2 \cdots C_t$$

be the alternating decomposition of S, such that for odd i, C_i is a read sequence over A, and for even i, it is a read sequence over B.

Consider each boundary between S_j and S_{j+1}, for $1 \leq j \leq m-1$. Notice that if neither boundary of S_j splits a cluster, then S_j is simply a concatenation of the clusters it contains, which alternate between A and B, possibly starting with either set. We show that whenever a boundary splits a cluster C_i into two pieces, both pieces are read sequences over the same variable set as C_i.

Consider a boundary after

$$S_1 S_2 \cdots S_j$$

that lies inside a cluster C_i. Write

$$C_i = C_i^0 C_i^1,$$

where C_i^0 lies before the boundary and C_i^1 lies after it. Suppose that C_i is a read-ℓ sequence over A; the case of a B-cluster is identical.

Let

$$K_j = \sum_{h=1}^{j} k_h.$$

The prefix $S_1 S_2 \cdots S_j$ is a read-K_j sequence over X, so every variable of A appears exactly K_j times in this prefix. Let r be the total read multiplicity of all A-clusters preceding C_i. Those preceding clusters contain every variable of A exactly r times. Therefore, every variable of A appears exactly

$$K_j - r$$

times in C_i^0. Hence C_i^0 is a read-$(K_j - r)$ sequence over A. Since C_i is a read-ℓ sequence over A, the remaining piece C_i^1 is a read-$\big(\ell - (K_j - r)\big)$ sequence over A.

Thus, after inserting all boundaries between the sequences S_j, every resulting nonempty piece is a read sequence over either A or B. Fix j. The pieces contained in S_j come from consecutive clusters of the original alternating decomposition, with only the first and last possibly cut by the boundaries of S_j. Consequently, these pieces alternate between A and B, possibly starting with either set. Hence every S_j is either (A, B)-alternating or (B, A)-alternating. ■

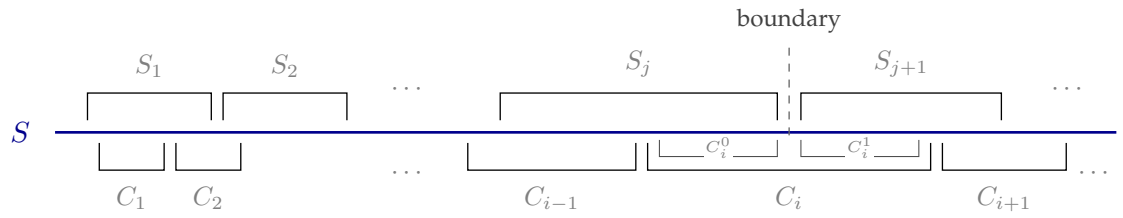

Figure 2: A boundary between two consecutive sequences S_j and S_{j+1} may cut through a cluster C_i in the alternating decomposition $S = C_1 C_2 \cdots C_t$. In that case, C_i is split into two consecutive pieces C_i^0 and C_i^1. The prefix $S_1 \cdots S_j$ and the preceding whole clusters each contain every variable of C_i equally often. Thus C_i^0, and hence also C_i^1, is a read sequence over the same variable set as C_i. Hence the induced pieces inside each S_r still alternate between A-clusters and B-clusters, possibly starting with either side.

As a result of this lemma, we get the strong attribute of recursively alternating sequences we discussed above. This attribute will be particularly helpful to us later when we prove that our communication network can simulate the branching program effectively. Consider a "walk" on the partition tree where we want to read the sequence S by traversing from leaf to leaf via the shortest path. Then this attribute tells us that locally this walk will look similar to a DFS. Every time we enter some subtree, we must read all the variables in that subtree at least once before leaving it. Therefore, we path through every edge at most $2k$ times.

Corollary 2.15. *Let S be a recursively alternating sequence, witnessed by a partition tree $T = (V, E)$, and let $v \in V$ be labeled by Y. Then every maximal contiguous block of S consisting of variables in Y is a read sequence over Y.*

Proof: We proceed by induction on the depth of v. The claim is immediate for the root, since S is a read sequence over its entire variable set.

Let v be a non-root vertex labeled by Y, and let its parent be labeled by Z. By the induction hypothesis, the maximal contiguous Z-blocks of S are read sequences over Z. These blocks concatenate to $S|_Z$, which alternates between Y and $Z \setminus Y$. The inheritance lemma therefore implies that each of these blocks also alternates between the two sets. Consequently, every maximal Y-block within such a block is a read sequence over Y.

Finally, distinct maximal Z-blocks are separated in S by variables outside Z, and hence outside Y. Thus every maximal contiguous Y-block of S lies within a single Z-block, proving the claim. ■

3 The INW Pseudorandom Generator

We begin this section by formally defining the model of computation we wish to fool, oblivious read-k *branching programs*.

Definition 3.1 (branching program)**.** *Let $X = \{x_1, \ldots, x_n\}$ be a set of Boolean variables. A length-m, width-w oblivious branching program over X is a layered directed graph*

$$P = (V, E), \qquad V = V_0 \sqcup V_1 \sqcup \cdots \sqcup V_m,$$

together with a sequence

$$S = (S_1, \ldots, S_m) \in X^m,$$

called the reading sequence*, satisfying the following conditions:*

1. *V_0 contains a single start vertex;*
2. *$|V_t| \le w$ for every $t \in \{0, \ldots, m\}$;*
3. *every edge leaving V_{t-1} enters V_t;*
4. *for every $v \in V_{t-1}$ and $b \in \{0, 1\}$, there is exactly one outgoing edge from v labeled b;*
5. *every vertex in the final layer V_m is labeled by an output value in $\{0, 1\}$.*

On input $z \in \{0,1\}^X$, the computation starts at the start vertex. At step t, the program reads the variable S_t and follows the outgoing edge labeled z_{S_t}. The output $P(z)$ is the label of the vertex reached in the final layer.

All branching programs considered in this paper are oblivious, so we omit the adjective "oblivious."

A read-k branching program is branching program whose reading sequence S is read-k (see Definition 2.1). When $k = 1$ the branching program is "read-once".

Our construction uses the INW pseudorandom generator [INW94] to fool branching programs by simulating them on a *communication network*.

Definition 3.2 (communication networks and protocols)**.** *A* communication network *is an undirected graph*

$$H = (V, E),$$

with one computationally unbounded processor associated with each vertex $v \in V$. Each processor has access to a private input and to its own private random bits.

A protocol *on a communication network H is a distributed algorithm executed by the processors of H, where communication is allowed only between neighboring processors, in synchronous rounds. In each round, every processor may send messages to any number of its neighbors in H. The local computation performed by the processors is unrestricted. For our purposes, we assume that each processor receives a single input bit (thus, the total input is n bits) and the computation's output is a single bit, given by one of the processors after the last round.*

One important parameter of the graph H is its *width*.

Definition 3.3 (partition tree and width [INW94])**.** *Let $H = (V, E)$ be a graph. A* partition tree *for H is a rooted binary tree T together with a bijection between V and the leaves of T. The tree is* balanced *if its depth is $O(\log |V|)$.*

Every internal node $z \in V(T)$ partitions V into three sets:

$$A_z, \qquad B_z, \qquad C_z,$$

where A_z and B_z are the vertices corresponding to leaves below the left and right children of z, respectively, and C_z consists of all remaining vertices.

The cut *associated with z is*

$$C(z) = \big\{\{x, y\} \in E : x \text{ and } y \text{ belong to different sets among } A_z, B_z, C_z\big\}.$$

The width *of z is the minimum size of a vertex set that covers all edges in $C(z)$. The width of T is the maximum width of any internal node of T, and the width of H is the minimum width over all balanced partition trees for H.*

We refer to the width of a communication network as the width of its underlying graph. We use t to denote graph width to distinguish it from branching-program width, usually denoted by w. This notion is closely related to *treewidth*: a graph of treewidth t has width $\mathcal{O}(t)$ [Ree92]. In particular, trees have constant width. Moreover, a balanced binary tree on N vertices admits a balanced partition tree of constant width whose structure can be accessed directly from the original tree in polynomial time and $O(\log N)$ working space[10].

INW uses specific types of protocols.

Definition 3.4 (normal protocol)**.** *A protocol on a communication network $H = (V, E)$ is called* normal *if it proceeds in synchronous rounds and, whenever a processor receives a message, the length of that message—possibly zero—is determined in advance.*

Consequently, no information is conveyed implicitly through the length or timing of a message, and the information received by a processor is completely accounted for by the bits explicitly sent to it.

Definition 3.5 (c-protocol)**.** *Let $H = (V, E)$ be a communication network. A normal protocol P on H is called a c-*protocol *if, for every choice of the processors' inputs and private random bits, every processor sends and receives at most c bits in total throughout the execution of P.*

The parameter c is often called the communication cost *of these protocols.*

[10] Split each rooted subtree into its root (a singleton leaf) and its descendants; split the latter into the two child subtrees and recurse, omitting empty parts. Each cut is covered by the corresponding root, and the depth at most doubles. Each new node is encoded by an original vertex and one of constantly many types, allowing navigation in $O(\log N)$ working space.

We now present the INW generator for fooling c -protocols.

Theorem 3.6 ([INW94], Theorem 2). *Let $H = (V, E)$ be a communication network with $|V| = n$, and let T be a balanced partition tree for H of width t. For every communication bound c and error $\varepsilon \in (0, 1)$, there is an explicit generator*

$$G_{H,T,c,\varepsilon} : \{0,1\}^d \to \{0,1\}^n$$

that ε-fools every c-protocol on H and has seed length

$$d = \mathcal{O}\left(\log n \cdot \left(tc + \log\left(\frac{n}{\varepsilon}\right)\right)\right).$$

Given read-only access to H and T, the output $G_{H,T,c,\varepsilon}(r)$ can be computed in time $\mathrm{poly}(n)$ and space $\mathcal{O}(d)$.

From this result one can deduce the known result of fooling read-once branching programs with seed length

$$d = \mathcal{O}\left(\log n \cdot \log \frac{nw}{\varepsilon}\right),$$

using a rather simple communication network, a path graph, to simulate the branching program. Indeed, most previous works that invoked the INW generator to fool branching programs only used the path graph instantiation. Our construction uses a new approach, using a binary tree instead to simulate the branching program.

Theorem 3.7 (INW on recursively alternating sequences)**.** *Let S be a read-k sequence over a variable set X of size n, and let T be a partition tree witnessing that S is recursively alternating. For every width parameter w and error $\varepsilon \in (0, 1)$, there is an explicit generator*

$$G_{S,T,w,\varepsilon} : \{0,1\}^d \to \{0,1\}^n$$

that ε-fools every width-w branching program with reading sequence S and has seed length

$$d = \mathcal{O}\left(\log n \cdot \left(k \log w + \log \frac{n}{\varepsilon}\right)\right).$$

Given read-only access to S and T, the output $G_{S,T,w,\varepsilon}(r)$ can be computed in time $\mathrm{poly}(n)$ and space $\mathcal{O}(d)$.

Proof: By Definition 2.3, the root of T is labeled by X, every leaf is labeled by a singleton variable set, and every internal vertex v_Y, labeled by Y, has two children v_A, v_B whose labels form a balanced partition

$$Y = A \sqcup B.$$

Moreover, $S|_Y$ is (A, B)-alternating. The leaves of T are in one-to-one correspondence with the variables of X. Hence T has n leaves and $N = |V(T)| = \mathcal{O}(n)$ vertices. Since all partitions are balanced, its depth is $\mathcal{O}(\log n)$.

Fix any width-w branching program P with reading sequence S. We describe a communication protocol on T that computes P. The leaf corresponding to $x \in X$ receives the input bit assigned to x. The internal vertices receive arbitrary dummy input bits, which the protocol ignores. We assume that every processor knows P, S, T, and the communication schedule.

The protocol maintains the current state of P, represented using $\lceil \log w \rceil$ bits. Initially, the leaf corresponding to the first variable in S holds the initial state of P. At each occurrence of a variable x, its leaf uses its input bit to apply the appropriate transition of P. The resulting state is then

routed along the unique shortest path from that leaf to the leaf corresponding to the next variable in S. After the final transition, the last leaf outputs 1 if and only if the resulting state is accepting. Thus, on every assignment $z \in \{0,1\}^X$, the output of the protocol is exactly $P(z)$.

We now bound the communication through each vertex of T. Let

$$e = (v_Y, v_A)$$

be an edge, where v_Y is labeled by $Y = A \sqcup B$ and v_A is labeled by A. Removing e separates the variables in A from all variables outside A. Thus, the state crosses e precisely when two consecutive queries lie on opposite sides of this partition.

By Corollary 2.15, every maximal contiguous A-cluster is a read sequence over A and therefore contains every variable in A at least once. Since each variable occurs at most k times in S, there are at most k such clusters. Consequently, every edge is crossed at most $2k$ times.

Each message contains $O(\log w)$ bits. Since every vertex of T has degree at most 3, the total number of bits sent and received by each processor is bounded by

$$c = \mathcal{O}(k \log w).$$

The active edge in every round is determined in advance by S and T, and every message has the predetermined length $\lceil \log w \rceil$. Thus, the protocol is normal and is a c-protocol.

Since T is a balanced binary tree, it admits a balanced partition tree $\mathcal{T}$ of width $O(1)$ whose structure can be accessed directly from T in polynomial time and $O(\log n)$ working space. Applying Theorem 3.6 with communication network T, partition tree $\mathcal{T}$, and communication bound c gives an explicit generator

$$\widetilde{G} = G_{T,\mathcal{T},c,\varepsilon} : \{0,1\}^d \to \{0,1\}^N$$

with seed length

$$\begin{aligned} d &= \mathcal{O}\left(\log N \left(c + \log \frac{N}{\varepsilon}\right)\right) \\ &= \mathcal{O}\left(\log n \left(k \log w + \log \frac{n}{\varepsilon}\right)\right) \end{aligned}$$

that ε-fools every c-protocol on T.

Define $G_{S,T,w,\varepsilon}$ by retaining the coordinates of $\widetilde{G}$ corresponding to the leaves of T, ordered according to their variable labels. The protocol ignores the internal-vertex coordinates, and its leaf coordinates are precisely the input bits of P. Therefore,

$$\left| \Pr_{r \sim U_d} [P(G_{S,T,w,\varepsilon}(r)) = 1] - \Pr_{z \sim U_n} [P(z) = 1] \right| \leq \varepsilon.$$

Since P was arbitrary, this proves the fooling guarantee.

Finally, explicitness follows from Theorem 3.6 and the direct access to $\mathcal{T}$ from the read-only description of T. The generator $\widetilde{G}$ can be evaluated in polynomial time in n and $O(d)$ working space. Computing the required leaf coordinates one at a time preserves these bounds, establishing the explicitness of $G_{S,T,w,\varepsilon}$. ∎

4 Our Construction

Our construction follows a similar route to that of [GV20], which is to partition the variable set X of the branching program into large enough subsets

$$X_1, X_2, ..., X_m$$

such that branching programs with the sequences $S|_{X_i}$, for $1 \leq i \leq m$, are all relatively "easy" to fool using the INW generator we described earlier in Theorem 3.7. We can then draw independent samples for each PRG and apply the following lemma, an easy application of the hybrid argument, to get a PRG for branching programs with S as its read sequence.

Lemma 4.1 (Gurjar–Volk [GV20])**.** *Let P be a read-k branching program with width w over a read sequence S with variable set X of size n. Let*

$$X = X_1 \sqcup X_2 \sqcup \cdots \sqcup X_m$$

be any partition of X. Assume that for every $i \in [m]$, there exists a PRG

$$G_i : \{0,1\}^{d_i} \to \{0,1\}^{|X_i|}$$

that ε-fools every branching program with width w over the sequence $S|_{X_i}$. Then, there exists a PRG

$$G : \{0,1\}^{md} \to \{0,1\}^n$$

that $m\varepsilon$-fools P, where $d = \max_{1 \leq i \leq m} d_i$.

4.1 Fooling Branching Programs Over Read-k Permutation Sequences

Sequences that recursively alternate are "easy" to fool, so they will serve as the partition of X we desire. We can now use Theorem 3.7 and Lemma 4.1 to get the following result.

Theorem 4.2. *For every read-k permutation sequence S over a variable set X of size n, width bound w, and error $\varepsilon \in (0,1)$, there is an explicit generator*

$$G_{S,w,\varepsilon} : \{0,1\}^d \to \{0,1\}^n$$

that ε-fools every width-w branching program with reading sequence S and has seed length

$$d = \mathcal{O}\left(n^{1-\frac{1}{k}} \log n \cdot \left(k \log w + \log \frac{n}{\varepsilon}\right)\right).$$

The construction first partitions X into subsets on which S recursively alternates and finds a witnessing tree for each subset. The following lemma bounds the time needed for this preprocessing step.

Lemma 4.3. *Given a read-k permutation sequence S over n variables, one can deterministically construct a partition*

$$X = X_1 \sqcup \cdots \sqcup X_m \sqcup R, \qquad m = \mathcal{O}(n^{1-\frac{1}{k}}), \qquad |R| \leq 1,$$

together with trees T_i witnessing that each $S|_{X_i}$ is recursively alternating, in $O(kn^{2-\frac{1}{k}})$ time.

Proof: If $n = 1$, take $m = 0$ and $R = X$. If $k = 1$, take $X_1 = X$ and $R = \varnothing$. A witnessing partition tree is obtained by repeatedly splitting the variable order into consecutive halves whose sizes differ by at most one. This takes $O(n)$ time. Henceforth, assume $n, k \geq 2$.

Apply the deterministic construction of Lemma 2.7 repeatedly to the remaining variables until at most one variable remains. At the i'th application, obtain a set X_i and a tree T_i witnessing that $S|_{X_i}$ is recursively alternating. We obtain T_i by recording the recursion tree of Lemma 2.7, labeling each leaf by its selected variable and each internal node by the set of variables labeling its descendant leaves.

Let m be the number of applications and set $\alpha = \frac{1}{k}$. When N variables remain, each application removes at least $\frac{1}{2}N^{\alpha}$ variables. While

$$\frac{n}{2^{j+1}} < N \leq \frac{n}{2^j},$$

each application therefore removes at least $\frac{1}{2}(\frac{n}{2^{j+1}})^{\alpha}$ variables. Consequently, at most $O((\frac{n}{2^j})^{1-\alpha})$ applications occur in this range. Summing these bounds over $j = 0, \ldots, \lceil \log n \rceil - 1$ gives

$$m = \mathcal{O}\left(\sum_{j=0}^{\lceil \log n \rceil - 1} \left(\frac{n}{2^j}\right)^{1-\alpha}\right) = \mathcal{O}\left(\frac{n^{1-\alpha}}{1 - 2^{-(1-\alpha)}}\right) = \mathcal{O}\left(n^{1-\frac{1}{k}}\right).$$

The last equality uses $\alpha \leq \frac{1}{2}$, so the implicit constant is independent of k. Let R, $|R| \leq 1$, be the set of remaining variables. Then, the selected sets, together with R, form the required partition.

Letting N_i be the number of variables remaining before the i's application, Lemma 2.7 takes time $O(kN_i)$, including recording the witnessing tree. Forming the restricted sequences and updating the remaining sequence also takes time $O(kN_i)$. Thus the total preprocessing time is

$$\mathcal{O}\left(k \sum_{i=1}^{m} N_i\right) = \mathcal{O}(knm) = \mathcal{O}(kn^{2-\frac{1}{k}}).$$

■

Lemma 4.3 supplies the partition and witnessing trees needed to apply Theorem 3.7 to each subset. This preprocessing depends only on S and is performed once, independently of the seed. At the end of the proof of Theorem 4.2 below, we will show that given (read-only) access to the preprocessing output and the seed, evaluation uses polylogarithmic space.[11]

Proof of Theorem 4.2: If $n = 1$, use the identity generator on one random bit. Henceforth assume $n \geq 2$. Use Lemma 4.3 to obtain the partition

$$X = X_1 \sqcup \cdots \sqcup X_m \sqcup R$$

and the witnessing trees T_i, where $m = \mathcal{O}(n^{1-\frac{1}{k}})$ and $|R| \leq 1$.

Set $\varepsilon' = \frac{\varepsilon}{n}$. For every i with $|X_i| \geq 2$, apply Theorem 3.7 to obtain

$$G_i = G_{S|_{X_i}, T_i, w, \varepsilon'}$$

[11] For $k \geq 2$. When $k = 1$, the construction runs in space linear in the seed.

that ε'-fools every width-w branching program with reading sequence $S|_{X_i}$. Its seed length is

$$d_i = \mathcal{O}\left(\log|X_i|\left(k\log w + \log\frac{|X_i|}{\varepsilon'}\right)\right) = \mathcal{O}\left(\log n\left(k\log w + \log\frac{n}{\varepsilon}\right)\right).$$

For singleton sets X_i, use the identity generator on one bit. By padding seeds if necessary, take all generators to have seed length

$$\ell = \mathcal{O}\left(\log n\left(k\log w + \log\frac{n}{\varepsilon}\right)\right).$$

Write the seed as $r = (r_1, \dots, r_m, r_R)$, where each r_i has length ℓ and r_R has length $|R|$. Define the output by

$$G_{S,w,\varepsilon}(r)|_{X_i} = G_i(r_i), \qquad G_{S,w,\varepsilon}(r)|_R = r_R.$$

Under a uniform seed, these blocks are independent. By Lemma 4.1, the resulting generator fools every width-w branching program with reading sequence S with error at most $m\varepsilon' \le \varepsilon$. Its seed length is

$$d = m\ell + |R| = \mathcal{O}\left(n^{1-\frac{1}{k}}\log n\left(k\log w + \log\frac{n}{\varepsilon}\right)\right).$$

Finally, we verify explicitness. Given (read-only) access to the partition, restricted sequences, witnessing trees, and seed, each INW generator can be evaluated in time $\mathrm{poly}(nk+\ell)$ and $O(\ell)$ working space by Theorem 3.7. We produce the output in the original variable order, computing the required coordinates for each individual PRG one at a time and reusing the working memory between computations. The total working space is

$$\mathcal{O}(\ell + \log n + \log d) = \mathcal{O}(\ell).$$

Computing all n output coordinates in this manner takes $\mathrm{poly}(nk+\ell)$ time. ∎

4.2 Fooling Branching Programs Over Arbitrary Read-k Sequences

We extend the construction to arbitrary read-k sequences, using Lemma 2.13 in place of Lemma 2.7.

Theorem 4.4 (main theorem)**.** *For every read-k sequence S over a variable set X of size n, width bound w, and error $\varepsilon \in (0,1)$, there is an explicit generator*

$$G_{S,w,\varepsilon} : \{0,1\}^d \to \{0,1\}^n$$

that ε-fools every width-w branching program with reading sequence S and has seed length

$$d = \mathcal{O}\left(n^{1-\frac{1}{2k-1}}\log n\cdot\left(k\log w + \log\frac{n}{\varepsilon}\right)\right).$$

As before, we first construct a partition into subsets whose restrictions recursively alternate, together with their witnessing trees.

Lemma 4.5. *Given a read-k sequence S over n variables, one can deterministically construct a partition*

$$X = X_1 \sqcup \cdots \sqcup X_m \sqcup R, \qquad m = \mathcal{O}\left(n^{1-\frac{1}{2k-1}}\right), \qquad |R| \le 1,$$

together with trees T_i witnessing that each $S|_{X_i}$ is recursively alternating, in $\mathcal{O}\left(kn^{2-\frac{1}{2k-1}}\right)$ time.

Proof: The cases $n = 1$ and $k = 1$ are handled as in Lemma 4.3. For $n, k \geq 2$, follow the same construction, using Lemma 2.13 instead of Lemma 2.7. When N variables remain, each application extracts at least $\frac{1}{2}N^{\frac{1}{2k-1}}$ variables. The counting argument of Lemma 4.3, with $\alpha = \frac{1}{2k-1} \leq \frac{1}{3}$, therefore gives

$$m = \mathcal{O}\left(n^{1-\frac{1}{2k-1}}\right),$$

with an implicit constant independent of k. Each application takes $O(kN)$ time, including recording the witnessing tree and forming the restricted sequences. Thus the total preprocessing time is

$$\mathcal{O}(knm) = \mathcal{O}\left(kn^{2-\frac{1}{2k-1}}\right).$$

■

With this preprocessing complete, we proceed as in the proof of Theorem 4.2. The same arguments establish correctness and explicitness, while the new bound on the number of subsets gives the claimed seed length.

Proof of Theorem 4.4: The cases $n = 1$ and $k = 1$ are handled as in Theorem 4.2. For $n, k \geq 2$, follow that proof, using Lemma 4.5 in place of Lemma 4.3. This gives $m = \mathcal{O}\left(n^{1-\frac{1}{2k-1}}\right)$ subsets. With the same seed length ℓ for each individual PRG, the total seed length is now

$$d = m\ell + |R| = \mathcal{O}\left(n^{1-\frac{1}{2k-1}}\ell\right),$$

which gives the claimed bound. The error and explicitness arguments apply without change. ■

Finally, we proceed to discuss arbitrary read-many branching programs of linear length. Notice that a sequence of length at most cn contains at most cn/k variables that are read more than k times. We can then sample these variables uniformly at random and apply our generator on the remaining variables. By choosing k carefully, we get the following.

Theorem 4.6. *Fix constants $c, a, b > 0$. Let S be a known reading sequence of length at most cn over a variable set X of size n. For every width bound $w \leq n^a$ and error parameter $n^{-b} \leq \varepsilon < 1$, there exists an explicit generator*

$$G_{S,w,\varepsilon} : \{0,1\}^d \to \{0,1\}^n$$

that ε-fools every width-w oblivious branching program with reading sequence S, where

$$d = \mathcal{O}\left(\frac{n \log\log n}{\log n}\right).$$

Proof: For sufficiently large n, set

$$L = \log n, \qquad k = \left\lfloor \frac{L}{8 \log L} \right\rfloor.$$

Let $F \subseteq X$ consist of the variables appearing more than k times in S. Since $|S| \leq cn$,

$$|F| \leq \frac{cn}{k} = \mathcal{O}\left(\frac{n \log L}{L}\right).$$

Fixing any assignment to F leaves a branching program of width at most w with reading sequence $S|_{X\setminus F}$, in which every variable appears at most k times. Append trivial transitions, so that every remaining variable is read exactly k times. The resulting reading sequence depends only on S. Thus, Theorem 4.4 gives a PRG that ε-fools every such restricted program.

Our choice of k gives

$$n^{\frac{-1}{2k-1}} = 2^{\frac{-L}{2k-1}} \le 2^{-4\log L} = L^{-4}.$$

Moreover, since $w \le n^a$ and $\varepsilon \ge n^{-b}$,

$$L\left(k\log w + \log\frac{n}{\varepsilon}\right) = \mathcal{O}\left(\frac{L^3}{\log L}\right).$$

Consequently, the generator for the remaining variables uses at most

$$\mathcal{O}\left(n^{1-\frac{1}{2k-1}} L\left(k\log w + \log\frac{n}{\varepsilon}\right)\right) = \mathcal{O}\left(\frac{n}{L\log L}\right)$$

seed bits.

Sample the variables in F uniformly and independently of the generator. Averaging the error bound over all assignments to F shows that the combined generator ε-fools every width-w program with sequence S. Its total seed length is

$$|F| + \mathcal{O}\left(\frac{n}{L\log L}\right) = \mathcal{O}\left(\frac{n\log\log n}{\log n}\right).$$

■

AI Usage. All ideas, constructions, and proofs, were developed by the authors. ChatGPT-6 Astra was used to assist with verifications, parameter bookkeeping, language editing, and creating the figures.

References

[AFS+16] Matthew Anderson, Michael A. Forbes, Ramprasad Saptharishi, Amir Shpilka, and Ben Lee Volk. Identity testing and lower bounds for Read-k oblivious algebraic branching programs. In Ran Raz, editor, *31st Conference on Computational Complexity (CCC 2016)*, volume 50 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 30:1–30:25. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2016.

[BDVY13] Andrej Bogdanov, Zeev Dvir, Elad Verbin, and Amir Yehudayoff. Pseudorandomness for width-2 branching programs. *Theory of Computing*, 9(7):283–293, 2013.

[BRRY14] Mark Braverman, Anup Rao, Ran Raz, and Amir Yehudayoff. Pseudorandom generators for regular branching programs. *SIAM Journal on Computing*, 43(3):973–986, 2014.

[BSSW98] Beate Bollig, Martin Sauerhoff, Detlef Sieling, and Ingo Wegener. Hierarchy theorems for kOBDDs and kIBDDs. *Theoretical Computer Science*, 205(1–2):45–60, 1998.

[De11] Anindya De. Pseudorandomness for permutation and regular branching programs. In *Conference on Computational Complexity (CCC)*, pages 221–231. IEEE Computer Society, 2011.

[FK18] Michael A. Forbes and Zander Kelley. Pseudorandom generators for read-once branching programs, in any order. In *Symposium on Foundations of Computer Science (FOCS)*, pages 946–955. IEEE, 2018.

[GKM18] Parikshit Gopalan, Daniel M. Kane, and Raghu Meka. Pseudorandomness via the discrete fourier transform. *SIAM Journal on Computing*, 47(6):2451–2487, 2018.

[GMR+12] Parikshit Gopalan, Raghu Meka, Omer Reingold, Luca Trevisan, and Salil Vadhan. Better pseudorandom generators from milder pseudorandom restrictions. In *Symposium on Foundations of Computer Science (FOCS)*, pages 120–129. IEEE, 2012.

[GV20] Rohit Gurjar and Ben Lee Volk. Pseudorandom bits for oblivious branching programs. *ACM Transactions on Computation Theory*, 12(2):8:1–8:12, 2020.

[HPV21] William M. Hoza, Edward Pyne, and Salil Vadhan. Pseudorandom generators for unbounded-width permutation branching programs. In *Innovations in Theoretical Computer Science Conference (ITCS)*, volume 185, pages 7:1–7:20. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021.

[INW94] Russell Impagliazzo, Noam Nisan, and Avi Wigderson. Pseudorandomness for network algorithms. In *Symposium on Theory of Computing (STOC)*, pages 356–364. ACM, 1994.

[KKL+21] Valentine Kabanets, Sajin Koroth, Zhenjian Lu, Dimitrios Myrisiotis, and Igor C. Oliveira. Algorithms and lower bounds for De Morgan formulas of low-communication leaf gates. *ACM Transactions on Computation Theory*, 13(4):23:1–23:37, 2021.

[MRT19] Raghu Meka, Omer Reingold, and Avishay Tal. Pseudorandom generators for width-3 branching programs. In *Symposium on Theory of Computing (STOC)*, pages 626–637. ACM, 2019.

[MZ13] Raghu Meka and David Zuckerman. Pseudorandom generators for polynomial threshold functions. *SIAM Journal on Computing*, 42(3):1275–1301, 2013.

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

[Ree92] Bruce A. Reed. Finding approximate separators and computing tree-width quickly. In *Symposium on Theory of Computing (STOC)*, pages 221–228. ACM, 1992.

ECCC ISSN 1433-8092

https://eccc.weizmann.ac.il