Electronic Colloquium on Computational Complexity, Revision 2 of Report No. 14 (2022)

Tighter **MA**/1 Circuit Lower Bounds From Verifier Efficient PCPs for PSPACE

Joshua Cook* Dana Moshkovitz†

July 7, 2023

Abstract

We prove that for some constant $a > 1$, for all $k \le a$,

$$\mathbf{MATIME}[n^{k+o(1)}]/1 \not\subset \mathbf{SIZE}[O(n^k)],$$

for some specific $o(1)$ function. This is a super linear polynomial circuit lower bound.

Previously, Santhanam [San07] showed that there exists a constant $c > 1$ such that for all $k > 1$:

$$\mathbf{MATIME}[n^{ck}]/1 \not\subset \mathbf{SIZE}[O(n^k)].$$

Inherently to Santhanam's proof, c is a large constant and there is no upper bound on c. Using ideas from Murray and Williams [MW18], one can show for all $k > 1$:

$$\mathbf{MATIME}[n^{10k^2}]/1 \not\subset \mathbf{SIZE}[O(n^k)].$$

To prove this result, we construct the first **PCP** for **SPACE**$[n]$ with quasi-linear verifier time: our **PCP** has a $\tilde{O}(n)$ time verifier, $\tilde{O}(n)$ space prover, $O(\log(n))$ queries, and polynomial alphabet size. Prior to this work, **PCP**s for **SPACE**$[O(n)]$ had verifiers that run in $\Omega(n^2)$ time. This **PCP** also proves that **NE** has **MIP** verifiers which run in time $\tilde{O}(n)$.

*jac22855@utexas.edu. Department of Computer Science, UT Austin. This material is based upon work supported by the National Science Foundation under grant number 1705028.

†danama@cs.utexas.edu. Department of Computer Science, UT Austin. This material is based upon work supported by the National Science Foundation under grant number 1705028.

ISSN 1433-8092

Contents

1 Introduction

Some of the most fundamental problems in complexity theory are proving circuit lower bounds for uniform complexity classes. One such conjecture is that **NP** does not have polynomial size circuits, which is a strong version of $\mathbf{P} \neq \mathbf{NP}$. Very little is known on such lower bounds. In particular, there are no known proofs that **NEXP** does not have polynomial sized circuits! However, there are some closely related results that could be loosely seen as relaxations.

One can strengthen **NP** slightly by giving the non-deterministic algorithm access to randomness, as well as an extra bit of trusted advice. This gives the complexity class $\mathbf{MA}/1$. We can weaken polynomial sized circuits to circuits of fixed polynomial size: $\mathbf{SIZE}[n^k]$ for constant k.

Santhanam [San07] proved that for any constant k, $\mathbf{MA}/1 \subsetneq \mathbf{SIZE}[n^k]$. The $\mathbf{MA}/1$ algorithm runs in time n^{ck} for a large $c > 1$. In fact, inherently to Santhanam's proof, there is no upper bound on c (We will explain why when we describe Santhanam's proof in Section 1.2.1). One can use ideas from Murray and Williams [MW18] to get, for some explicit c with $2 < c < 10$, the result $\mathbf{MATIME}[n^{ck^2}]/1 \not\subset \mathbf{SIZE}[n^k]$.

The goal of this paper is to prove a fine grained separation of $\mathbf{MA}/1$ from fixed polynomial size circuits, namely,

$$\mathbf{MATIME}[n^{k+o(1)}]/1 \not\subset \mathbf{SIZE}[n^k].$$

We believe that the gold standard for separations should be fine grained separations. Fine grained separations are necessary for key results in complexity theory, e.g., Williams' program (See, e.g., [Wil11]) and optimal derandomization [Dor+20].

Some fine grained separations are known, namely, hierarchy theorems that show that giving algorithms more time allows them to solve more problems [HS65; Coo72]. Hierarchy theorems are known for many complexity classes. While no hierarchy theorems are known for **MA**, they are known for $\mathbf{MA}/1$. Fortnow, Santhanam, and Trevisan showed that **MA** with a small amount of advice can solve more problems when given more time [FST05]. Van Melkebeek and Pervyshev showed that for any $1 < b < d$, $\mathbf{MATIME}[n^b]/1 \subsetneq \mathbf{MATIME}[n^d]/1$ [MP06]. In particular, they imply that even $\mathbf{MATIME}[n^{2k}]/1$ is much larger than $\mathbf{MATIME}[n^{k+o(1)}]/1$.

1.1 Results

In this work, we give a fine grained separation for $\mathbf{MA}/1$ and $\mathbf{SIZE}[n^k]$. We show that for at least some $k > 1$, there is an **MA** protocol with one bit of advice whose verifier has time almost n^k such that any circuit solving the same problem also requires size n^k. Formally:

Theorem 1.1.1 (Fine Grained **MA** Lower Bound)**.** *There exists a constant $a > 1$, such that for all $k < a$, for some $f(n) = o(1)$,*

$$\mathbf{MATIME}[O(n^{k+f(n)})]/1 \not\subset \mathbf{SIZE}[O(n^k)].$$

We stress that we give **super linear** polynomial lower bounds. Our result holds for some k *strictly* greater than 1, even though we don't know which k. This result removes the large polynomial factor in the gap between the $\mathbf{MA}/1$ time and the circuit size in Santhanam's result. It may be the case that a is small, like $a = 1.0001$. But in that case, we get the following result for all k:

Theorem 1.1.2 (**MA** Lower Bound for Small a)**.** *If the a from Theorem 1.1.1 is finite, then for all $k > 0$, for some $f(n) = o(1)$,*

$$\mathbf{MATIME}[O(n^{ak+f(n)})]/1 \not\subset \mathbf{SIZE}[O(n^k)].$$

This gives us a win-win scenario: if a is large, we get a strong result for a large range of k, but if a is small we get a similar result for all k.

When we describe our proof we will explain why we only get separations for $k < a$ for an (unknown) $a > 1$ and not for all $k > 1$. For now we would like to stress that: (1) under plausible complexity assumptions the upper bound a is in fact super-constant in n; (2) even the case of a constant $a > 1$ as promised in our theorem is highly interesting, since it is unknown how to prove that $\mathbf{NP} \subsetneq \mathbf{SIZE}[n^k]$ for *any* $k > 1$.

Santhanam's original proof uses an interactive protocol for **PSPACE**. To prove our circuit lower bound, we replace the interactive protocol with a new, more efficient **PCP**. To get our fine grained results, we need a **PCP** for space $S = O(n)$ and time $T = 2^{O(n)}$ algorithms, where the verifier simultaneously has $\tilde{O}(n)$ time and $\mathbf{poly}(\log(n))$ many queries. Further, the **PCP** needs a prover that can compute any bit of the proof in $\tilde{O}(n)$ space. Notably, we do not need any bounds on the proof length.

The **PCP** given by Babai, Fortnow, and Lund in their proof that **MIP** = **NEXP** [BFL90] required $\Omega(\log(T))$ queries, while we want $O(\log(\log(T)))$ queries.

Holmgren and Rothblum in their work on delegated computation [HR18] improved on the BFL **PCP** in several ways that can[1] be used to give a **PCP** with verifier time $\tilde{O}(n + \log(T))$. Unfortunately, it still requires $\Omega(\log(T))$ queries.

Ben-Sasson, Goldreich, Harsha, Sudan, and Vadhan [Ben+05] gave a **PCP** that uses a constant number of queries, but has verifier time $\mathbf{poly}(\log(T))$, while we need $\tilde{O}(n + \log(T))$ verifier time. Similar results were given by subsequent work [Mei09; BV14; Ben+13].

The small space requirement for the prover is achieved by Holmgren and Rothblum [HR18]. In some **PCP**s, like the **PCP** in Ben-Sasson, Chiesa, Genkin, and Tromer's work on the concrete efficiency of **PCP**s [Ben+13], the prover requires space $\Omega(T)$. In contrast, our result needs prover space $\tilde{O}(S + n)$.

A sufficiently efficient **PCP** was not known, so we construct a new **PCP**.

Theorem 1.1.3 (Verifier Efficient **PCP**)**.** *Let $S, T = \Omega(n)$ be functions, and L be any language computed by a simultaneous time T and space S algorithm. Let $\delta \in (0, 1/2)$ be a constant. Then there is a* **PCP** *for L with:*

1. *Verifier time $\tilde{O}(n + \log(T))$.*
2. *Query time $\tilde{O}(\log(T))$.*
3. $O(\log(n) + \log(\log(T)))$ *queries.*
4. *Alphabet Σ with* $\log(|\Sigma|) = O(\log(\log(T)))$.
5. *Log of proof length $\tilde{O}(\log(T))$.*
6. *Prover space $\tilde{O}(S)$.*
7. *Perfect completeness and soundness δ.*

We believe we can achieve a similar verifier time, query time and prover space while also achieving constant number of queries and alphabet size. We do not need these improvements for our main result, so we only prove this simpler result.

Only our prover requires the space bound for its efficient computation. If we remove this space limitation, we get a similar **PCP** for nondeterministic algorithms.

Theorem 1.1.4 (Verifier Efficient **PCP** for Nondeterministic Algorithms)**.** *Let $T = \Omega(n)$, $\delta \in (0, 1/2)$ be a constant, and $L \in \mathbf{NTIME}[T]$. Then there is a* **PCP** *for L with:*

1. *Verifier time $\tilde{O}(n + \log(T))$.*
2. *Query time $\tilde{O}(\log(T))$.*
3. $O(\log(n) + \log(\log(T)))$ *queries.*
4. *Alphabet Σ with* $\log(|\Sigma|) = O(\log(\log(T)))$.
5. *Log of proof length $\tilde{O}(\log(T))$.*
6. *Perfect completeness and soundness δ.*

An immediate corollary of Theorem 1.1.4 is a more fine grained equivalence between **MIP** and **NEXP**.

[1] The **PCP** constructed by Holmgren and Rothblum was built to have no signalling soundness and has many steps that take longer than $\tilde{O}(\log(T))$ time to compute. Still, the basic elements of of their **PCP** needed for a standard **PCP** are computable in $\tilde{O}(n + \log(T))$ time.

Corollary 1.1.5 (Fine Grained Equivalence of **MIP** = **NEXP**)**.** *For any time constructable function $p(n) = \Omega(n)$, language $L \in$ **NTIME**$[2^{\tilde{O}(p(n))}]$ if and only if there is a two prover, one round* **MIP** *protocol for L whose verifier runs in time $\tilde{O}(p(n))$.*

Note this equivalence implies a hierarchy theorem for **MIP** since there are hierarchy theorems for **NTIME** [Coo72; SFM78; Žá83; FS11].

A special case is **MIP** protocols for **NE**.

Corollary 1.1.6 (**NE** Has Quasi-linear Time Verifiers)**.** *For any language $L \in$ **NE**, there is a two prover, one round* **MIP** *protocol for L whose verifier runs in time $\tilde{O}(n)$.*

Note this verifier time is nearly optimal since the verifier requires linear time to read its entire input.

All previous **PCP**s fail to achieve such an efficient **MIP** verifier. If the original **PCP** makes $\Omega(n)$ queries of size $\Omega(n)$, then it takes $\Omega(n^2)$ time to send the queries even if we allow more provers. And all previous **PCP**s with fewer queries require verifier time $\Omega(n^2)$ to either verify the response or compute the queries.

1.2 Proof Idea

1.2.1 MA Lower Bounds Using PCP

We first review Santhanam's original proof.

Santhanam's original result uses the fact that if **PSPACE** $\subset$ **P/poly**, then **PSPACE** = **MA**. This follows from the famous result that **IP** = **PSPACE** [Sha92; Lun+92]. The idea is that if **PSPACE** $\subset$ **P/poly**, then an **MA** protocol can guess a circuit computing any problem in **PSPACE**. The prover in the interactive protocol for **PSPACE** is also computable in **PSPACE**. So to solve any **PSPACE** problem in **MA**, the **MA** protocol first guesses the circuit for a prover, then simulates the verifier using the circuit we guessed as the prover.

Using this, Santhanam's original proof then considered two cases: either **PSPACE** $\subset$ **P/poly**, or **PSPACE** $\not\subset$ **P/poly**.

If **PSPACE** $\subset$ **P/poly**, then we already know **PSPACE** = **MA**. Now we just need a problem not computable by a size n^k circuit. But there is a straightforward algorithm that exhaustively finds a circuit of size larger than n^k that computes a function that cannot be computed by a smaller circuit. In fact, such an algorithm only requires space $\tilde{O}(n^k)$. So **PSPACE** $\not\subset$ **SIZE**$[n^k]$. In this case, **PSPACE** = **MA**, so **MA** $\not\subset$ **SIZE**$[n^k]$.

If **PSPACE** $\not\subset$ **P/poly**, then we know a hard problem that is not in **SIZE**$[n^k]$, namely any **PSPACE** complete problem. Let us take a **PSPACE** complete, downward self reducible language, Y. Now Y may be too hard for **MA** to solve, but if we give it enough padding, eventually the padded version of Y will be computable by size n^k circuits. But for this amount of padding, **MA** can pull the same trick it does in the **PSPACE** $\subset$ **P/poly** case. Namely, guess a circuit for Y and then simulate the **IP** protocol for Y. For some **PSPACE** complete Y, the language itself is its proof and this works. The trick is to use just the right amount of padding so it requires circuits of at least size n^k, but not much larger. Santhanam uses the single bit of advice in a clever way to figure out when there is just the right amount of padding.

In either case, the time of this protocol is roughly the time of the verifier in the **IP** protocol, plus the size of the prover circuit times the number of times the prover is queried.

There are two reasons the **MA** protocol could take polynomially more time than the size of the circuits it wants to compute in the case **PSPACE** $\subset$ **P/poly**. One is that the **IP** from the original Santhanam result has polynomial verifier time and a polynomial time interaction with the prover, making the verifier in the **MA**/1 protocol take polynomially longer than the circuit complexity of the problem being solved. By using a **PCP**, we get better results. The other is that the prover circuit complexity could be large, depending on the circuit size required for **PSPACE** (could be any polynomial when **PSPACE** $\subset$ **P/poly**). This is the reason there is no upper bound on the polynomial run time of the **MA**/1 protocol in Santhanam's proof. To avoid this issue we consider a finer case analysis.

We break the problem into three cases. For some **SPACE**$[O(n)]$ complete language, X, we have one[2] of the following:

[2]This is a trichotomy in an asymptotic sense: for every constant a, either $X \in$ **SIZE**$[O(n^a)]$ or it is not. See Section 3.5 for details.

1. $X \notin \mathbf{P/poly}$.
2. $X \in \mathbf{SIZE}[n^{1+o(1)}]$.
3. $X \in \mathbf{SIZE}[n^{a+o(1)}] \setminus \mathbf{SIZE}[n^{a-o(1)}]$ for some $a > 1$.

The original proof only used the two cases $X \notin \mathbf{P/poly}$ and $X \in \mathbf{P/poly}$. The case where $X \notin \mathbf{P/poly}$ is completely unchanged. Note that this is the plausible case, and here there is no constant upper bound a on k.

If $X \in \mathbf{P/poly}$, we use our efficient **PCP**, Theorem 1.1.3, instead of the **IP** Santhanam uses. With this substitution, the case where $X \in \mathbf{SIZE}[n^{1+o(1)}]$ is almost unchanged from the original proof. By separating this into it's own case, we get tight bounds for all k in this case.

If $X \in \mathbf{SIZE}[n^{a+o(1)}] \setminus \mathbf{SIZE}[n^{a-o(1)}]$ for some $a > 1$, then we use the same padding technique we use if $X \notin \mathbf{P/poly}$, just using our new **PCP**. In this case, we can only do this if for some $k < a$, we are trying to show $\mathbf{MATIME}[n^{k+o(1)}]/1 \not\subset \mathbf{SIZE}[n^{k-o(1)}]$. This is the case where a is finite, but in this case, we can use Santhanam's argument using our **PCP** to get Theorem 1.1.2.

To see why $k > a$ poses a difficulty, suppose $\mathbf{SPACE}[O(n)] \not\subseteq \mathbf{SIZE}[o(n^2)]$, but $\mathbf{SPACE}[O(n^2)] \subseteq \mathbf{SIZE}[O(n^2)]$. Then to get a language requiring size n^3 circuits, we need to use a space n^3 algorithm. But the prover for a space n^3 language is a language running on an input with length n^3, and using space linear in its input length. Thus we may need a size $(n^3)^2 = n^6$ circuit for our prover. So the verifier takes time at least n^6 to even read the prover circuit, thus can't run in time n^3. See Item 2 in our open problems for further explanation.

Remark. *We note our verifier in Theorem 1.1.1 is a RAM machine,* not *a standard Turing Machine. This is because we know how to efficiently simulate a circuit on a RAM machine, but not on a standard Turing Machine.*

1.2.2 Verifier Efficient PCP

Now we explain the **PCP** we actually use in the **MA** protocol. We start with a **PCP** similar to [HR18] and [BFL90] that we refer to as our base **PCP**. This **PCP** has a verifier that runs in time $\tilde{O}(n + \log(T))$ and uses $O(\log(T))$ queries. To reduce the number of queries, we use **PCP** composition [AS98; BS+04; DR04; MR08; DH09].

To perform **PCP** composition, we need a robust **PCP**. Loosely, a robust **PCP** is a **PCP** so that when $x \notin L$, for any proof, most sets of queries to that proof return not only a rejected response, but a response that is far from any accepted response. To make our base **PCP** robust, we use the aggregation through curves technique [Aro+98]. Now we briefly explain how to use aggregation through curves to convert our base **PCP** into a robust **PCP**.

An honest proof for our base **PCP** is a single low degree polynomial. Suppose our base **PCP** has q queries. To make our **PCP** robust, we first choose the randomness for the base **PCP**, and another random point in the **PCP** proof. Then we find the degree q curve that goes through all these points. Then we check if the proof, restricted to this curve, is a low degree polynomial, and whether the base **PCP** would have accepted on this input. Since a low degree polynomial is an error correcting code, this gives robustness.

One concern one might have with this robust **PCP** is that it actually requires $\Omega(\log(T)^2)$ queries. We don't need to actually calculate all of these query locations. Since we reduce the actual number of queries with **PCP** composition, we only need to be able to calculate any individual query location quickly. To find these query locations requires us to compute a point on the degree q curve going through each of our q points our base **PCP** queries plus a random point. In our base **PCP**, $q = O(\log(T))$ and our proof has dimension $O(\log(T))$. So the naive way to compute this curve is to calculate each coordinate independently, which would take time $\tilde{O}(\log(T)^2)$.

To efficiently compute low degree curves through points, or to extrapolate a function going through those points, we introduce the concept of time extrapolatable functions.

Definition 1.2.1 (Extrapolatability)**.** *For any $n, q, t > 0$, and field $\mathbb{F}$, we call $Q : [q] \to \mathbb{F}^n$ "t extrapolatable" (or time t extrapolatable) if there is a time t algorithm taking any $v \in \mathbb{F}^q$, that outputs*

$$\sum_{i \in [q]} v_i Q(i).$$

Equivalently, if we think of Q as outputting the columns of a matrix, then we say Q is time t extrapolatable if one can multiply a vector with it in time t. An important property of extrapolatable functions is that an extrapolation of an extrapolatable function can be computed efficiently. This is where it gets its name.

Our base **PCP** is just a sum check and a few point checks. Each of these are time $\tilde{O}(\log(T))$ extrapolatable. Our robust **PCP** only queries locations easily computable given the extrapolation of our base **PCP** query locations. Extrapolations of extrapolatable functions are easy to compute, so we can easily compute the query locations of the robust **PCP**.

We also introduce the concept an extrapolatable **PCP** (**ePCP**) as one where an honest proof is a low degree polynomial, and the query locations after fixing a choice of randomness are extrapolatable. We show that any **ePCP** can be extended into a robust **PCP** where the query locations of that robust **PCP** can be computed efficiently.

1.3 Generalization And Sharpness

We actually prove a stronger result than Theorem 1.1.1 that is sharp. First, our **MA** protocol is input oblivious: the message from Merlin is just a program for computing a **PSPACE** complete language and doesn't depend on the specific input, just its length. Second, the hardness is against the model used in Merlin's message. We used circuits, but we can describe a randomized algorithm directly to save some polynomial factors.

We define input oblivious Merlin-Arthur time, **OMATIME**, the same way as Fortnow, Santhanam, and Williams [FSW09]. Input oblivious Merlin-Arthur are languages solvable with untrusted advice, where the advice only depends on the input length. In our case, Merlin gets to send a long, untrusted message for every input length, and Arthur also gets a single bit of trusted advice. See Definition 2.0.3. Note that Santhanam's original proof implicitly also uses input oblivious **MA**.

The main property of circuits we use is that a randomized algorithm can efficiently simulate it. We can instead use $\mathbf{BPTIME}[n^k]/n^k$, that is, randomized algorithms running in time n^k with description length n^k. This uses the same model of computation as our verifier, allowing it to more efficiently simulate **OMATIME**.

Using **OMATIME** instead of **MATIME** and **BPTIME** instead of **SIZE**, we can follow the same proof as our main result to show:

Theorem 1.3.1 (**OMATIME** Lower Bound Against **BPTIME**)**.** *There exists constant $a > 1$, such that for all $k < a$, for some $f(n) = o(1)$,*

$$\mathbf{OMATIME}[O(n^{k+f(n)})]/1 \not\subset \mathbf{BPTIME}[O(n^k)]/O(n^k).$$

This result is tight in the sense that for any function $f(n)$, we have

$$\mathbf{OMATIME}[f(n)]/1 \subseteq \mathbf{BPTIME}[O(f(n))]/(f(n)+1).$$

To get stronger results, we need to use nondeterminism that depends on the input. So one could say our result is less about the power of nondeterminism, and more about the power of trusted versus untrusted advice. Specifically: trusting advice doesn't always buy (much) time in the randomized setting, as long as we have *some* trusted advice.

Let us briefly outline what would need to change in our main proof to prove Theorem 1.3.1, and justify why those changes would work.

First we need to make a class of randomized programs that act more like circuits. Consider the class of programs, $\mathcal{C}$, that contain randomized algorithms that work only a specific input length. For any $C \in \mathcal{C}$, we say $C(x)$ is the random variable that simulates C on input x for time $|C|$, and outputs what C does if C terminates in time $|C|$, and outputs 0 otherwise. See that $\mathcal{C}$ behaves like circuits in the following important ways:

1. Given a program $C \in \mathcal{C}$, a randomized algorithm can calculate the random variable $C(x)$ in time $O(|C|)$.
2. For any function $f(n)$ and language $L \in \mathbf{BPTIME}[f(n)]/f(n)$, for every n, there is a $C_n \in \mathcal{C}$ such that $|C_n| = O(f(n))$ and with high probability $C_n(x) = 1_{x \in L}$.

A few notes on using $\mathcal{C}$ in our proof, as opposed to circuits.

1. First, see that $\mathbf{SPACE}[O(n^k)] \not\subseteq \mathbf{BPTIME}[o(n^k)]/o(n^k)$. This follows using the same exhaustive search type algorithm used for $\mathbf{SIZE}[o(n^k)]$.

   For any polynomial n^k, there is a deterministic program, A, with length $O(n^k)$ running in time $O(n^k)$, but is not computable with high probability by any program $C \in \mathcal{C}$ with length $o(n^k)$. This follows from a simple counting argument: length n^k deterministic programs contain more than $2^{\alpha n^k}$ functions for some constant α (just use some lookup table), while there are only $2^{\alpha n^k}$ length αn^k programs.

   Such an A can still be found by exhaustive search in space $O(n^k)$, since given $C \in \mathcal{C}$, we can space efficiently check every choice of randomness and calculate majority. This gives us that

   $$\mathbf{SPACE}[O(n^k)] \not\subseteq \mathbf{BPTIME}[o(n^k)]/o(n^k).$$

2. Given that $L \in \mathbf{BPTIME}[f(n)]/f(n)$, then for any n, there is some advice (notably, a program $C_n \in \mathcal{C}$ with size $|C_n| \leq f(n)$) such that a randomized algorithm given that advice can compute whether $x \in L$ with probability $1-\epsilon$ in time $O(f(n)\log(\frac{1}{\epsilon}))$.

   This allows our verifier to efficiently compute a $\mathbf{SPACE}[O(n)]$ complete problem, L, in time nearly $f(n)$ if $L \in \mathbf{BPTIME}[f(n)]/f(n)$, given correct advice.

3. We also note that for some programs $C \in \mathcal{C}$, for some inputs x, our program C evaluated on x may answer one or zero with very close to half probability. That is, the syntax of $\mathcal{C}$ does not only give bounded error randomized algorithms, just a randomized algorithm.

   This is not an issue, because in the completeness case, there will be a program C that does have bounded error the prover should provide. And in the soundness case, the soundness of our **PCP** holds against any multi-prover strategy, even a randomized strategy. So no program provided will convince the verifier with high probability.

Finally, see that in all cases of our proof, Arthur only asks Merlin for a program computing some $\mathbf{SPACE}[O(n)]$ complete problem. This advice does not depend on the specific input, only on the input size.

2 Preliminaries

We assume some familiarity with basic complexity theory. See Arora and Barak's book for background [AB09]. In this paper, by algorithm, we mean algorithm on a RAM machine, and by circuit, we mean a fan in 2 circuit with unbounded depth. A randomized algorithm is a deterministic algorithm with an extra input for randomness. We will assume in this paper that all time and space bounds for algorithms are sufficiently easily computable.

Now recall that **MA** is the complexity class of problems with polynomial sized certificates that can be verified with bounded error by a randomized, polynomial time algorithm. This is like **NP** with a randomized verifier.

Then we define **MATIME** in an analogous way to **NTIME**. Our results have perfect completeness, so we only define **MATIME** with perfect completeness.

Definition 2.0.1 (**MATIME**). *For any function $f : \mathbb{N} \to \mathbb{N}$, $\mathbf{MATIME}[f(n)]$ is the class of languages, L, such that there is a time $f(n)$ algorithm M taking three inputs, an input x, a random input r, and a witness w, so that*

Completeness *If $x \in L$ and $n = |x|$, then there exists w with $|w| \leq f(n)$ such that*

$$\Pr_r[M(x,r,w) = 1] = 1.$$

Soundness *If $x \notin L$, then for every w,*

$$\Pr_r[M(x,r,w) = 1] < 1/2.$$

An algorithm with trusted advice is an algorithm with an extra input for advice, where the advice is fixed for every input of a given length. Complexity class **MATIME**$[f(n)]/1$ is **MATIME**$[f(n)]$ with 1 bit of trusted advice.

Definition 2.0.2 (**MATIME**/1)**.** *For any function $f : \mathbb{N} \to \mathbb{N}$, define* **MATIME**$[f(n)]/1$ *as the set of languages, L, such that there is a function $b : \mathbb{N} \to \{0,1\}$ and a time $f(n)$ randomized algorithm M taking four inputs, an input x, a random input r, a witness w, and an advice bit such that*

Completeness *If $x \in L$ and $n = |x|$, then there exists w with $|w| \leq f(n)$ such that*

$$\Pr_r[M(x, r, w, b(n)) = 1] = 1.$$

Soundness *If $x \notin L$ and $n = |x|$, then for every w,*

$$\Pr_r[M(x, r, w, b(n)) = 1] < 1/2.$$

As described in the results section, we also define input oblvious Merlin-Arthur. Note that in our sharper results of Section 1.3, our advice only has bounded error, so we define **OMATIME** with *im*perfect completeness.

Definition 2.0.3 (**OMATIME**/1)**.** *For function $f : \mathbb{N} \to \mathbb{N}$, define* **OMATIME**$[f(n)]/1$ *as the set of languages, L, such that there is a trusted advice function $b : \mathbb{N} \to \{0,1\}$, an untrusted advice function $w : \mathbb{N} \to \{0,1\}^*$ with $|w(n)| \leq f(n)$ and a time $f(n)$ randomized algorithm M taking four inputs, an input x, a random input r, untrusted advice, and a trusted advice bit such that*

Completeness *If $x \in L$ and $n = |x|$, then*

$$\Pr_r[M(x, r, w(n), b(n)) = 1] > 2/3.$$

Soundness *If $x \notin L$ and $n = |x|$, then for every w',*

$$\Pr_r[M(x, r, w', b(n)) = 1] < 1/3.$$

We let **SIZE** denote the class of languages with circuits of a given size.

Definition 2.0.4 (**SIZE**)**.** *For any function $f : \mathbb{N} \to \mathbb{N}$,* **SIZE**$[f(n)]$ *is the class of languages, L, where for each input length n, there is a circuit of size $f(n)$ with n inputs computing L for inputs of length n.*

Further, **SIZE**$[O(f(n))]$ *is the class of languages, L, such that for some $g(n) = O(f(n))$, we have $L \in$* **SIZE**$[g(n)]$*. Similarly for* **SIZE**$[o(f(n))]$*.*

To show that $L \notin$ **SIZE**$[o(f(n))]$, we will show that for some constant $c > 0$, for infinitely many n, language L on length n inputs requires circuits of size at least $cf(n)$. This implies that for any $g(n) = o(f(n))$, language L must have size greater than $g(n)$ infinitely often, because eventually, $g(n)$ must stay below $cf(n)$.

While super linear circuit lower bounds have been hard to prove, one can easily get linear circuit lower bounds for any language that depends on every bit in the input, for instance, the parity function.

Lemma 2.0.5 (Parity Requires Large Circuits)**.** *Let L be the language of strings with an odd number of 1s. Then $L \in$* **TIME**$[O(n)]$*, but L on length n inputs requires circuits of size $n/2$.*

This lower bound comes from the fact that parity as a function depends on every input, and since each gate only has fan in 2, we need at least $n/2$ gates to make the circuit a function of every input. Similarly, since **TIME**$[O(n)] \subseteq$ **MATIME**$[O(n)]$, we get a similar result for **MATIME**. Since we can run an algorithm that only computes parity on some specific subset of the input, we can extend this to sublinear time as well.

Corollary 2.0.6 (Sub-linear Circuit Lower Bounds Are Easy)**.** *For any time constructible $S(n) \leq n/2$, there exists a language $L \in$* **TIME**$[O(S(n))]$ *but for every n, language L on length n inputs requires circuits of size $S(n)$.*

We assume a model of computation where n is provided to the algorithm in binary. Then the language is just parity on the first $2S(n)$ bits. By Lemma 2.0.5, this requires a size $S(n)$ circuit. Since $S(n)$ is time constructable, we can construct $S(n)$ then run parity on the first $S(n)$ bits in $O(S(n))$ time.

We will occasionally need to look at projections of a string onto some indexes.

Definition 2.0.7 (Projection)**.** *For any set Σ, naturals $n, m \in \mathbb{N}$, string $\pi = (\pi_1, \dots, \pi_n) \in \Sigma^n$, and indices $I = (I_1, \dots I_m) \in [n]^m$, we define the projection $\pi_I = (\pi_{I_1}, \dots, \pi_{I_m})$. We may also write for $i \in [n]$, $\pi(i) = \pi_i$, and $\pi(I) = \pi_I$.*

In this paper, we will focus on time and space efficient, non-adaptive **PCP**s with perfect completeness. Because we need to pay close attention to the amount of time it takes to make a single query to the proof, we separate the algorithm for producing queries, Q, from the algorithm for verifying the response, V. We also separate the function that gives all the query locations for a choice of randomness, I, from the algorithm that gives a single one of those query locations, Q.

So at a high level, a **PCP** protocol does the following:

1. Chooses a common random string, r.
2. Runs query function Q with randomness r for $q(n)$ many times to get all query locations, I.
3. Looks up all query locations, I, into a provided proof, π, to get proof window π_I.
4. Runs verifier V with randomness r and proof window π_I and outputs if V accepts.

Then if the input is in a language L, we want some proof π to always make the verifier accept. But if an input is not in language L, we want for any proof π, the probability the verifier accepts to be small. We also want a prover, P, that can compute any symbol of the proof using low space.

Now we formally define a **PCP**.

Definition 2.0.8 (**PCP**)**.** *We say that a language L has a non-adaptive* **PCP***, A, with perfect completeness if there exists verifier V, prover P, index function I, and query function Q, such that, for some alphabet Σ, $\delta \in [0, 1]$, and functions $r, l, q : \mathbb{N} \to \mathbb{N}$:*

1. *I takes 2 inputs, an input of length n and randomness of length $r(n)$, and outputs an element of $[l(n)]^{q(n)}$. That is, I outputs $q(n)$ indexes in a length $l(n)$ string,*
2. *Q is an algorithm with three inputs, an input x of length n, randomness r of length $r(n)$, and an index $i \in [q(n)]$ and outputs an element of $[l(n)]$ such that $Q(x, r, i) = I(x, r)_i$.*
3. *V is an algorithm with three inputs, an input of length n, randomness of length $r(n)$, and $q(n)$ symbols from Σ, and outputs either accept or reject.*
4. *P is an algorithm that takes two inputs, an input of length n, and an index $i \in [l(n)]$, and outputs a symbol from Σ.*

Completeness *If $x \in L$ and $n = |x|$, then there exists $\pi^x \in \Sigma^{l(n)}$ such that*

$$\Pr_r[V(x, r, \pi^x_{I(x,r)}) = 1] = 1,$$

and for every $i \in [l(n)]$, $P(x, i) = \pi^x_i$.

Soundness *If $x \notin L$ then for every π',*

$$\Pr_r[V(x, r, \pi'_{I(x,r)}) = 1] \leq \delta.$$

Then we also say:

1. *A has proof length $l(n)$.*
2. *A has alphabet Σ.*

3. *A has soundness δ.*

4. *A uses $q(n)$ queries.*

5. *A uses $r(n)$ bits of randomness.*

6. *If V runs in time $t(n)$, A has verifier time $t(n)$.*

7. *If V runs in space $s(n)$, A has verifier space $s(n)$.*

8. *If P runs in space $s'(n)$, A has prover space $s'(n)$.*

9. *If Q is computable in time $t'(n)$, A has query time $t'(n)$.*

For convenience, we assume that any alphabet or field is always encoded with some canonical binary encoding. We generally will not worry too much about encoding as we switch from models of computation and we will assume inputs are encoded in binary using a small power of two bits.

We use big O and little o notation extensively in this paper. We will use the result that sub-polynomial functions remain sub-polynomial when composed with polynomials.

Lemma 2.0.9 (Composing Sub-polynomials with Polynomials gives Sub-polynomials.)**.** *If $h(n) = o(1)$, and for some constant k, we have $D(n) = O(n^k)$, then for some $h'(n) = o(1)$,*

$$D(n)^{h(D(n))} = O(n^{h'(n)}).$$

Proof. Let $G(n) = n^{h(n)}$ so that $G(D(n)) = D(n)^{h(D(n))}$. Then we can bound $\log(G(n))$:

$$\log(G(n)) = h(n)\log(n) = o(\log(n)).$$

Using that $\log(n)$ is increasing and unbounded, we can bound $\log(G(D(n)))$.

$$\log(G(D(n))) = o(\log(D(n))) = o(\log(n)).$$

This is equivalent to, for some $h'(n) = o(1)$,

$$\log(G(D(n))) = h'(n)\log(n).$$

This gives the result.

$$D(n)^{h(D(n))} = 2^{\log(G(D(n)))} = n^{h'(n)}.$$

□

3 Efficient PCP To Fine Grained Lower Bounds

Our analysis depends on the circuit complexity of some **PSPACE** complete problem. So we start by choosing a **SPACE**$[O(n)]$ complete problem. We use a version of `SPACE TMSAT` (on page 83 of [AB09]).

Definition 3.0.1 (Specific Problem)**.** *`SPACE TMSAT` is the language*

$$\{(M, x, 1^n, 0^*) : \textit{Turing machine } M \textit{ accepts } x \textit{ using at most } n \textit{ space.}\}$$

Note: `SPACE TMSAT` $\in$ **SPACE**$[O(n)]$ and `SPACE TMSAT` is **SPACE**$[O(n)]$ complete. The 0^* is just there to make it explicit the language is paddable. In particular, this means that the circuit complexity of `SPACE TMSAT` is non-decreasing.

Lemma 3.0.2 (`SPACE TMSAT` Circuit Complexity is Non-Decreasing)**.** *If $A'(n)$ is the size of the minimum circuit solving `SPACE TMSAT` for inputs of length n, then $A'(n)$ is non-decreasing.*

Proof. Let C be the circuit of size $A'(n+1)$ solving `SPACE TMSAT` for length $n+1$ inputs. Then to get a circuit for length n inputs, use C with an extra 0 hard coded into the last input. The resulting circuit will be at most the size of C and solve length n inputs. Thus $A'(n+1) \geq A'(n)$. □

Then using Theorem 1.1.3, we can get a **PCP** for `SPACE TMSAT` by setting $T = 2^{O(n)}$ and $S = O(n)$. This can be turned into a **PCP** with a binary alphabet by replacing every query for a symbol in Σ with $O(\log(n))$ queries to the individual bits of that symbol.

Corollary 3.0.3 (**PCP** for `SPACE TMSAT`)**.** *There is a* **PCP** *for `SPACE TMSAT` with:*

1. *Verifier time* $\tilde{O}(n)$.
2. *Query time* $\tilde{O}(n)$.
3. $\mathbf{poly}(\log(n))$ *queries.*
4. *Binary alphabet.*
5. *Log of proof length* $\tilde{O}(n)$.
6. *Prover space* $\tilde{O}(n)$.
7. *Soundness* $1/2$ *and perfect completeness.*

We prove three different **MATIME**/1 lower bounds that are based on three different hard problems. Different ones work better in different parameter regimes. After constructing them all, we show we always fall into some range of parameters so that we can get the lower bounds of Theorem 1.1.1.

3.1 Implicitly Encoding Advice in Input Length

In each of our cases, we will use advice to find the size of some prover circuit. To do this, we implicitly encode a number in the input length. If that implicitly encoded number describes the size, our advice bit will be 1. Otherwise, the advice bit is 0.

For any input length $n \in \mathbb{N}$, for some $l \in \mathbb{N}$, we have $n \in [2^l, 2^{l+1})$. For such an l, there is some $m \in \mathbb{N}$ such that $n = 2^l + m$. This m, or equivalently this l, is our implicitly encoded number. Because we will use this decomposition a lot, we will explicitly define some functions that perform this decomposition.

Definition 3.1.1 (Implicit Encoding In Input)**.** *For natural* $n \geq 1$*, let* $l \geq 0$ *be an integer so that* $n \in [2^l, 2^{l+1})$*, and* $m \geq 0$ *be an integer so that* $n = 2^l + m$*. Then define* $\mu(n) = m$ *and* $\rho(n) = l$*.*

There is a simple interpretation of this $m = \mu(n)$ and $l = \rho(n)$ in terms of the binary representation of n. You can think of l as the length of the binary number, and m the binary number after the top bit is removed.

3.2 `SPACE TMSAT` $\notin$ P/poly

In this case, we follow the proof in the original work [San07] where **PSPACE** $\not\subset$ **P/poly**. We present the same arguments here in more generality and with more precise parameters.

When **PSPACE** $\not\subset$ **P/poly**, the circuit complexity of different input sizes for `SPACE TMSAT` could change drastically and in a way that may be hard to analyze. This is an issue because the **PCP** for `SPACE TMSAT` needs a prover with a longer input than the input being verified, thus might require a much larger circuit.

Instead, we use a downward self reducible **PSPACE** complete language. Specifically, a language that has a sound interactive protocol with queries the same length as its input and whose prover is the language itself. We cite the result from Lemma 11 in [San07]:

Lemma 3.2.1 (Same Size, Self Proving **PSPACE** Complete Language)**.** *There is a* **PSPACE***-complete language* Y *and a probabilistic polynomial-time oracle Turing machine* M *such that for any input* x*:*

1. M *only asks its oracle queries of length* $|x|$.
2. *If* M *is given* Y *as oracle and* $x \in Y$*, then* M *accepts with probability* 1.
3. *If* $x \notin Y$*, then irrespective of the oracle given to* M*,* M *rejects with probability at least* $1/2$.

The important feature of language Y is that for an input x, the prover for x is the same language Y, and queries to the prover have the same length as x. This means Y, and the prover for Y, have the same circuit.

Now using Lemma 3.2.1, we can get the following bound.

Lemma 3.2.2 (Bound Using Padded Y as Hard Problem)**.** *Using Y from Lemma 3.2.1, if for some $g(n) = \omega(1)$ we have $Y \notin \mathbf{SIZE}[O(n^{g(n)})]$ then for any time constructable, non-decreasing, unbounded $S(n)$ such that $S(n) = o(n^{g(n)})$, for some*[3] *$f(n) = o(1)$:*

$$\mathbf{MATIME}[O(S(n)^{1+f(n)})]/1 \not\subset \mathbf{SIZE}[o(S(n/4))].$$

Proof. Let $a > 0$ be the constant so that the verifier (M in Lemma 3.2.1) for Y's interactive protocol runs in time $O(n^a)$.

Now we define our language, W, in $\mathbf{MATIME}[S(n)^{1+o(1)}]/1$ but not in $\mathbf{SIZE}[o(S(n))]$. For any input size, n, using Definition 3.1.1, let $m = \mu(n)$ and $l = \rho(n)$. Let our advice bit be 1 if

1. Y on length m inputs does not have circuits of size $m^{g(m)}$,
2. Y on length m inputs has circuits with size $S(n)$, and
3. for all integers l' with $l' < l$ and $2^{l'} > m$, Y on length m inputs does not have circuits of size $S(2^{l'} + m)$.

   This condition requires the advice bit to only be 1 for a given m exactly once, whenever it can be used first. This simplifies the analysis, giving us a one to one function from n where the advice bit is 1, to m.

Then $x \in W$ for some x with $|x| = n$ if and only if the advice bit is 1 and for some $y \in Y$ with $|y| = m$ we have $x = y1^{n-m}$.

Now we will show that infinitely often the advice bit is 1 and W does not have circuits with size $S(n/4)$.

Since $Y \notin \mathbf{SIZE}[O(n^{g(n)})]$, for some infinite set U', for $m \in U'$, the language Y on input length m does not have circuits of size $m^{g(m)}$. Since $S(m) = o(m^{g(m)})$, for some n', for all $m \geq n'$, $S(m) < m^{g(m)}$. So let $U = U' \cap [n', \infty)$. See that $|U| = \infty$.

For $m \in U$, since $S(n)$ is non-decreasing and unbounded, for large enough l, language Y on length m inputs has circuits of size at most $S(2^l + m)$. Then there is a smallest such l with $2^l > m$ and for $n = 2^l + m$, the language Y on length m inputs has circuits of size $S(n)$. For such n, the advice bit is 1.

Now either $2^{l-1} \leq m$, or $2^{l-1} > m$.

$2^{l-1} \leq m$ Then $2m \geq 2^l$, and $m > n/4$. Since $m \in U$, language Y on length m inputs does not have circuits of size $S(m)$. Since $S(n)$ is monotone, Y on length m inputs also doesn't have circuits of size $S(n/4)$.

$2^{l-1} > m$ Then by choice of l, Y on length m inputs does not have circuits of size $S(2^{l-1} + m)$. Since by definition of n, we have $2^{l-1} + m > n/2$ and $S(n)$ is monotone, Y on length m inputs does not have circuits of size $S(n/2)$.

So W does not have circuits with size less than $S(n/4)$.

Since U has infinitely many elements, and for every $m \in U$, there is an $n > m$ such that W on length n inputs does not have circuits of size $S(n/4)$, for infinitely many n, language W on length n inputs does not have circuits of size $S(n/4)$. So $W \notin \mathbf{SIZE}[o(S(n/4))]$.

Now we define $f(n)$. Let $\mu_1(n)$ be the partial function from n where the advice bit is 1, to $\mu(n)$. We claim $\mu_1(n) = \omega(1)$. This is because for any m, the advice bit can only be 1 once. Thus μ_1 is one to one. Any one to one function into the naturals is $\omega(1)$, since for any b, there is a max n such that for some $m < b$, $\mu_1(n) = m$, and for all $i > n$, $\mu_1(i) \geq b$. Then let

$$D(n) := \begin{cases} \mu_1(n) & \text{Advice bit for } n \text{ is } 1 \\ D(n-1) & \text{Otherwise} \end{cases}.$$

Since $\mu_1(n) = \omega(1)$, we also have $D(n) = \omega(1)$. Then since $g(n) = \omega(1)$, we also have that for $f(n) = a/g(D(n))$, we have $f(n) = o(1)$.

[3] More generally, if $A(n)$ is the minimum circuit size for Y, the $\mathbf{MA}$ verifier will run in time similar to $S(n)\mathbf{poly}(A^{-1}(S(n)))$. Since $A^{-1}(n)$ is not simple, we avoid proving a more detailed result here.

Now we show that $W \in \mathbf{MATIME}[O(S(n)^{1+f(n)})]/1$. If the advice bit is 0, this is true trivially. Suppose that the advice bit is 1.

For an n where the advice bit is 1, inputs of length $m = \mu(n)$ for Y have circuits of size $S(n)$, which can be guessed. Then from Lemma 3.2.1, there is a time m^a algorithm that can verify membership in Y with a circuit for Y. This gives an $\mathbf{MA}$ protocol for Y on length m that runs in time $O(S(n)m^a)$.

Then since the advice bit is 1, there are circuits for length m instances of Y with size $S(n)$, but not $m^{g(m)}$. Thus $S(n) > m^{g(m)}$, so $S(n)^{1/g(m)} > m$. So the time of the $\mathbf{MA}$ verifier is at most $O(S(n)m^a) = O(S(n)S(n)^{a/g(m)}) = O(S(n)^{1+f(n)})$. The $\mathbf{MA}$ protocol is complete and sound since the protocol for Y is. So $W \in \mathbf{MATIME}[O(S(n)^{1+f(n)})]/1$.

Therefore

$$W \in \mathbf{MATIME}[O(S(n)^{1+f(n)})]/1 \setminus \mathbf{SIZE}[o(S(n/4))].$$

□

We show that when $\mathbf{PSPACE} \not\subset \mathbf{P/poly}$, there is some $g(n) = \omega(1)$ such that $Y \notin \mathbf{SIZE}[n^{g(n)}]$. Thus we can apply Lemma 3.2.2.

Corollary 3.2.3 (Bound if **PSPACE** does not have Polynomial Sized Circuits)**.** *If* `SPACE TMSAT` $\notin \mathbf{P/poly}$*, then for any* $k > 0$*, and some* $f(n) = o(1)$*:*

$$\mathbf{MATIME}[O(n^{k+f(n)})]/1 \not\subset \mathbf{SIZE}[O(n^k)].$$

Proof. We want to use Lemma 3.2.2 with $S(n) = n^k \log(n)$ and some $g(n) = \omega(1)$. Let Y be the langauge from Lemma 3.2.1.

Since `SPACE TMSAT` is in **PSPACE**, `SPACE TMSAT` $\notin \mathbf{P/poly}$ and Y is **PSPACE** complete, $Y \notin \mathbf{P/poly}$. We will show, since $Y \notin \mathbf{P/poly}$, for some $g(n) = \omega(1)$, we have $Y \notin \mathbf{SIZE}[o(n^{g(n)})]$.

Let $A(n)$ be the size of the smallest circuit computing Y on length n inputs. Let $g'(n) = \frac{\log(A(n))}{\log(n)}$. Suppose for contradiction that $g'(n)$ was bounded above by a constant, c. Then for all n, we have $g'(n) \leq c$ and $A(n) = n^{g'(n)} \leq n^c$. Thus Y has polynomial sized circuits. But Y doesn't, so $g'(n)$ is unbounded.

Let $g^*(n) = \max_{i \in [n]} g'(i)$. Since $g^*(n) \geq g'(n)$, we also know $g^*(n)$ is unbounded. By definition, $g^*(n)$ is non-decreasing. Thus $g^*(n) = \omega(1)$.

For infinitely many n, we know $g'(n) = g^*(n)$, since $g'(n)$ is unbounded. So for n such that $g'(n) = g^*(n)$, our problem Y does not have circuits of size less than $n^{g'(n)} = A(n)$. So infinitely often, Y does not have circuits of size $n^{g^*(n)}/2$. Thus $Y \notin \mathbf{SIZE}[o(n^{g^*(n)})]$.

Now let $g(n) = g^*(n) - 1$. Then $g(n) = \omega(1)$ and $n^{g(n)} = o(n^{g^*(n)})$, so $Y \notin \mathbf{SIZE}[O(n^{g(n)})]$.

Since $g(n) = \omega(1)$, see that $n^k \log(n) = o(n^{g(n)})$. Then using Lemma 3.2.2, we have that for some $f(n) = o(1)$,

$$\mathbf{MATIME}[O((n^k \log(n))^{1+f(n)})]/1 \not\subset \mathbf{SIZE}[o((n/4)^k \log(n/4))].$$

Now to simplify this. Since k is a constant, we have that $n^k = o((n/4)^k \log(n/4))$. Thus

$$\mathbf{SIZE}[O(n^k)] \subset \mathbf{SIZE}[o((n/2)^k \log(n/2))].$$

Now for $f'(n) := kf(n) + (1 + f(n))\frac{\log(\log(n))}{\log(n)}$ we have $f'(n) = o(1)$ and $(n^k \log(n))^{1+f(n)} = n^{k+f'(n)}$. Thus

$$\mathbf{MATIME}[O((n^k \log(n))^{1+f(n)})]/1 \subseteq \mathbf{MATIME}[O(n^{k+f'(n)})]/1.$$

Together

$$\mathbf{MATIME}[O(n^{k+f'(n)})]/1 \not\subset \mathbf{SIZE}[O(n^k)].$$

□

3.3 SPACE TMSAT $\in$ $\mathbf{SIZE}[n^{1+o(1)}]$

The idea in this case is to use a brute force, small space algorithm that finds a problem not in a fixed polynomial size. In particular, for circuit size $S(n)$, the brute force algorithm uses space $O(S(n))$ to compute some function with minimum circuit size $\Theta(S(n))$. Then we want to simulate the **PCP** from Corollary 3.0.3 to prove the output of this algorithm. Since the **PCP** is efficient, the prover for this algorithm does not use much more space than the brute force algorithm itself.

If `SPACE TMSAT` has almost linear sized circuits, the prover doesn't require much larger circuits than the space of the prover. Finally, our **PCP** is efficient, so the time of the **MA** verifier isn't much more than the size of the prover circuit. So the **MA** protocol doesn't require much more time then the size of the circuit it proves the output of.

If `SPACE TMSAT` requires larger circuits, say quadratic circuits, then the size of the prover circuits would be quadratically larger than the input length of the prover. That is, the prover circuit would be quadratically larger than the circuit it is trying to prove. This would give quadratic overhead for the **MA** verifier time over the size of the circuit it verifies. So this construction only works well enough when `SPACE TMSAT` has almost linear sized circuits.

Lemma 3.3.1 (Bound From Exhaustive Search As Hard Problem)**.** *If for some non-decreasing $A(n)$ we have `SPACE TMSAT` $\in \mathbf{SIZE}[O(A(n))]$, then there is some non-decreasing $B(n) = \tilde{\Theta}(n)$ such that for any time constructable, non-decreasing $S(n)$ with $S(n) < \frac{2^n}{n}$ and $S(n)2^n = \omega(A(B(S(n))))$:*

$$\mathbf{MATIME}[\tilde{O}(A(B(S(n))))]/1 \not\subset \mathbf{SIZE}[o(S(n/2))].$$

The $B(n)$ in this problem comes directly from the prover space and the log of the proof length[4] of our **PCP** given in Corollary 3.0.3. The outer polylogarithmic factors in the **MA** verifier time come from the number of queries made by the **PCP**, the query time, and the **PCP** verifier time.

Proof. One can show `SPACE TMSAT` requires circuits of size $\Omega(n)$ since it can compute parity and thus needs to read most of the bits in the input, so $A(n) = \Omega(n)$. If $S(n) = O(n)$, then use Corollary 2.0.6. Otherwise, we can assume $S(n) > 10n$.

The proof proceeds in five steps.

1. Find a language $L \in \mathbf{SPACE}[\tilde{O}(S(n))] \setminus \mathbf{SIZE}[S(n)/10]$. In particular, for every input length n, language L has circuits of size $S(n)$ but not $S(n)/10$.

2. Reduce L to `SPACE TMSAT` and use Corollary 3.0.3. In particular, find a circuit, C_n, for the prover in an **MA** protocol for L on length n inputs.

3. Define our advice bit to implicitly give an upper bound for the size of C_m for some m within a factor of 2 of n. Then we define W to be length m elements of L, padded to length n.

4. Show that infinitely often the advice bit is 1 and W does not have small circuits.

5. Show that W has an efficient **MA** protocol.

With that outline in mind, let us begin the proof.

1. Find a language $L \in \mathbf{SPACE}[\tilde{O}(S(n))] \setminus \mathbf{SIZE}[o(S(n))]$.

   By theorem premise $S(n) < 2^n/n$. So from the non-uniform hierarchy (see Theorem 6.22 in Arora and Barak [AB09]), there is a language $L \in \mathbf{SIZE}[S(n)] \setminus \mathbf{SIZE}[S(n)/10]$. In particular, for every n, language L on length n has circuits size $S(n)$ but not size $S(n)/10$.

   Consider an algorithm, M, recognizing such an L which checks all circuits of size $S(n)$, and compares them with every circuit of size $S(n)/10$ on every input, and returns the output from the first circuit of size $S(n)$ that disagrees with every circuit of size $S(n)/10$ on some input.

   Then M runs in space $\tilde{O}(S(n))$ (there may be a logarithmic overhead between the size of a circuit, and the size of its description) and recognizes an $L \notin \mathbf{SIZE}[S(n)/10]$. So we have an $L \in \mathbf{SPACE}[\tilde{O}(S(n))] \setminus \mathbf{SIZE}[S(n)/10]$. In particular, for every n, language L on length n does not have circuits of size $S(n)/10$.

[4] The log of the proof length of a **PCP** gives the length of a query to the prover.

2. Reduce L to `SPACE TMSAT` and use Corollary 3.0.3.

   Since M only uses $g(n)$ space, for some $g(n) = \tilde{O}(S(n))$, we know $x \in L$ if and only if $(M, x, 1^{g(n)}, 0) \in$ `SPACE TMSAT`. We know `SPACE TMSAT` on length $\tilde{O}(S(n))$ inputs has a **PCP** protocol from Corollary 3.0.3 that uses $\mathbf{poly}(\log(S(n)))$ many length $\tilde{O}(S(n))$ queries to a space $\tilde{O}(S(n))$ prover, P, where each query can be calculated by a time $\tilde{O}(S(n))$ algorithm, Q, and the results from P are verified by a time $\tilde{O}(S(n))$ verifier, V.

   Now we reduce the prover P to `SPACE TMSAT` so we can use that `SPACE TMSAT` $\in \mathbf{SIZE}[O(A(n))]$ to get a circuit for P.

   A length $\tilde{O}(S(n))$ query, q, to P can be converted into a length $\tilde{O}(S(n))$ input, q', for `SPACE TMSAT` by providing the algorithm for P and $\tilde{O}(S(n))$ 1s. In particular, for some $B(n) = \tilde{O}(n)$, proof input q' has length $B(S(n))$. We can also take $B(n) = \Omega(n)$. Call the circuit for `SPACE TMSAT` on length $|q'|$ inputs C_n. Since `SPACE TMSAT` $\in \mathbf{SIZE}[O(A(n))]$, we know C_n has size $O(A(B(S(n))))$.

3. Define our advice bit.

   Now an **MA** protocol can guess C_n, but we may not be able to compute how large C_n needs to be. The function $A(n)$ may be hard to compute. So we use advice.

   Let $l = \rho(n)$, $m = 2^l$ and $t = \mu(n)$ so that $n = m + t$. Then let the advice bit be 1 if

   (a) Circuit C_m has size $S(m)2^t$.

   (b) For any natural t' less than t, circuit C_m does not have size $S(m)2^{t'}$.
   This condition allows us to use the smallest t possible for a given m.

   Then $x \in W$ for some x with $|x| = n$ if and only if the advice bit is 1 and for some $y \in$ `SPACE TMSAT` with $|y| = m$ we have $x = y1^{n-m}$.

4. Show W does not have small circuits.

   First we show that for every large enough l, for $m = 2^l$, there will be one t such that this advice bit is 1. To show this, we will show that for some t, C_m has size $S(m)2^t$. Then for the minimum such t, the advice bit will be one.

   For $t = m - 1$, by premise of the theorem, we have

   $$S(m)2^t = S(m)2^m/2 = \omega(A(B(S(m)))).$$

   This is eventually larger than C_m since C_m has size $O(A(B(S(m))))$. Then for large enough l with $m = 2^l$, there will be a smallest t so that C_m has size $S(m)2^t$ circuits, since it will for $t = m - 1$. The advice bit for such an $n = m + t$ must be 1. So infinitely often, the advice bit will be 1.

   When the advice bit is 1, the language W on length $n = m + t$ inputs is equal to L on length m inputs. Language L on length m inputs does not have circuits of size $S(m)/10$. See by choice of m that $2m > n$, and $S(n)$ is monotone, so $S(m)/10 > S(n/2)/10$. Thus infinitely often, W does not have size $S(n/2)/10$ circuits. Thus $W \notin \mathbf{SIZE}[o(S(n/2))]$.

5. Show W has an efficient **MA** protocol.

   If the advice bit is 0, this is trivially true. For $n = 2^l + t$ so that the advice bit is 1 and $m = 2^l$, either

   $t = 0$: Then C_m has size $S(m)$. Since $A(n) = \Omega(n)$ and $B(n) = \Omega(n)$, we know C_m has size $O(A(B(S(m))))$.

   $t \geq 1$: Then C_m has size $S(m)2^t$ but not $S(m)2^{t-1}$. Since C_m does have circuits of size $O(A(B(S(m))))$:

   $$\begin{aligned} S(m)2^{t-1} &= O(A(B(S(m)))) \\ S(m)2^t &= O(A(B(S(m)))). \end{aligned}$$

In either case, an **MA** protocol can guess C_m with a circuit with size $O(A(B(S(m))))$.

Then an **MA** protocol for $x = y1^{n-m}$ and an advice bit of 1 can verify if $y \in L$ by first guessing a circuit for C_m, then using it as the prover in the **PCP** protocol from Corollary 3.0.3.

The **MA** verifier needs to calculate $\mathbf{poly}(\log(S(m)))$ queries with Q, run C_m on each of those queries, and run V on those results. Since C_m has size $O(A(B(S(m))))$, and Q and V run in time $\tilde{O}(S(m))$, calculating all query locations, running C_n on each of those locations, and V on those outputs takes time

$$
\begin{aligned}
&\mathbf{poly}(\log(S(m)))(\tilde{O}(S(m)) + O(A(B(S(m))))) + \tilde{O}(S(m)) \\
=&\tilde{O}(A(B(S(m))) + S(m)) \\
=&\tilde{O}(A(B(S(m)))).
\end{aligned}
$$

The last equality comes from the fact $A(n) = \Omega(n)$ and $B(n) = \Omega(n)$. Finally, since A, B and S are non-decreasing and $m < n$, the **MA** verifier runs in time $\tilde{O}(A(B(S(n))))$.

The **MA** protocol is complete and sound since the **PCP** is. Thus $W \in \mathbf{MATIME}[\tilde{O}(A(B(S(n))))]/1$.

Therefore

$$W \in \mathbf{MATIME}[\tilde{O}(A(B(S(n))))]/1 \setminus \mathbf{SIZE}[o(S(n/2))].$$

$\square$

And in the special case where `SPACE TMSAT` has almost linear sized circuits, we get:

Corollary 3.3.2 (Bound if `SPACE TMSAT` has Size $n^{1+o(1)}$)**.** *If for some $g(n) = o(1)$ and some non-decreasing function $A(n) = n^{1+g(n)}$ we have `SPACE TMSAT` $\in \mathbf{SIZE}[O(A(n))]$, then for any $k > 0$, there is an $f(n) = o(1)$ such that:*

$$\mathbf{MATIME}[O(n^{k+f(n)})]/1 \not\subset \mathbf{SIZE}[O(n^k)].$$

Proof. We want to use Lemma 3.3.1 with $S(n) = n^k \log(n)$. The size upper bound on $S(n)$ is clear: $S(n) = o(2^n/n)$. We need to show $S(n)2^n = \omega(A(B(S(n))))$. Well for any $B(n) = \tilde{O}(n)$,

$$
\begin{aligned}
A(B(S(n))) =&B(n^k \log(n))^{1+g(n)} \\
=&\tilde{O}(n^{k+kg(n)}) \\
=&o(2^n) \\
S(n)2^n =&\omega(A(B(S(n)))).
\end{aligned}
$$

So by Lemma 3.3.1, for some $B(n) = \tilde{O}(n)$,

$$\mathbf{MATIME}[\tilde{O}(A(B(S(n))))]/1 \not\subset \mathbf{SIZE}[o(S(n/2))].$$

See that for some $f(n) = o(1)$,

$$\tilde{O}(A(B(S(n)))) = \tilde{O}(n^{k+kg(n)}) = O(n^{k+f(n)}).$$

Similarly $n^k = o(S(n/2))$, so we also have

$$\mathbf{MATIME}[O(n^{k+f(n)})]/1 \not\subset \mathbf{SIZE}[O(n^k)].$$

$\square$

3.4 SPACE TMSAT $\in \mathbf{SIZE}[n^{a+o(1)}] \setminus \mathbf{SIZE}[n^{a-o(1)}]$ for $a > 1$

This is the "bad" case, where we can't prove the result for every constant k, only for $k < a$. This is the most complicated case, requiring us to both pad the input to get the correct problem difficulty, and use advice to get the size of the circuits for the prover.

Lemma 3.4.1 (Bound from SPACE TMSAT as Hard Problem)**.** *If for some non-decreasing $A(n)$ we have SPACE TMSAT $\in \mathbf{SIZE}[O(A(n))] \setminus \mathbf{SIZE}[o(A(n))]$, then there is some non-decreasing $B(n) = \tilde{\Theta}(n)$ and $D(n) = O(n)$ such that if for some time constructable, non-decreasing $S(n)$ with $S(2n) = o(A(n))$ and $S(n)2^n = \omega(A(B(n)))$, we have:*

$$\mathbf{MATIME}\left[\tilde{O}\left(S(n)\frac{A(B(D(n)))}{A(D(n)/2)}\right)\right]/1 \not\subset \mathbf{SIZE}[o(S(n/4))].$$

Before we give the proof, we explain the parameters in this result. In this problem, you can think of $D(n)$ as being similar to $A^{-1}(S(n))$, though to simplify the analysis, we use a more trivial bound of $D(n) = O(n)$. The $B(n)$ comes from the prover space and log of the proof length of Corollary 3.0.3. Then this fraction term in the **MA** verifier time, loosely, accounts for the increase in circuit size for SPACE TMSAT on length n inputs versus length $B(n)$ inputs.

If SPACE TMSAT $\in \mathbf{P/poly}$, the difference between the size of circuits for SPACE TMSAT on length n inputs and length $B(n)$ inputs will be small (at least for n where SPACE TMSAT requires circuits with size near the polynomial that upper bounds the size of SPACE TMSAT). But if SPACE TMSAT requires larger than polynomial sized circuits, then the difference in circuit size between length n inputs and length $B(n)$ may become large.

So the idea is to solve SPACE TMSAT on a padded version of the input using our **PCP**. So we need the advice to tell us three things:

1. Some m so that SPACE TMSAT on length m inputs requires circuits of size $S(n/4)$.

2. Further, we need SPACE TMSAT on length m inputs to require circuits of size near $A(m/2)$. This keeps the prover from requiring circuits too much larger than SPACE TMSAT on length m inputs does.

3. How big the circuit for the prover in Corollary 3.0.3 needs to be.

Similar to the previous cases, this advice will come implicitly from the input length, and the single advice bit will be 1 if and only if the input length encodes valid advice.

Proof. If $S(n) = O(n)$, we use Corollary 2.0.6. Otherwise, we want to solve a smaller instance of SPACE TMSAT that requires circuits of size $S(n/4)$, and we also need advice to tell us the size of circuits needed to prove SPACE TMSAT. The advice for this will come implicitly from the input length.

For input x of length n, (using ρ and μ from Definition 3.1.1) let $l = \rho(n)$, $l' = \rho(\mu(n))$, and $t = \mu(\mu(n))$ so that $n = 2^l + 2^{l'} + t$. We want to solve SPACE TMSAT on length $2^{l'}$ inputs, so we let $m := 2^{l'}$. Let $D(n) := m$. Then $n = 2^l + m + t$ and our language will solve length m inputs for SPACE TMSAT using prover circuits of size $S(2^l)2^t$. Then the advice bit will only be 1 only when this advice is good.

So then m is the input length to SPACE TMSAT we want to solve, 2^l is how much padding is needed to make length m problems the right difficulty, and $S(2^l)2^t$ is the size of the circuits needed for our **PCP** prover.

The proof proceeds in 4 steps.

1. Define circuits C_m that prove SPACE TMSAT for length m inputs using our **PCP** and our theorem assumptions on circuits for SPACE TMSAT.

2. Define when the advice bit should be 1.

3. Show infinitely often the advice bit is 1 and $W \notin \mathbf{SIZE}[o(S(n/4))]$.

4. Show that
$$W \in \mathbf{MATIME}\left[\tilde{O}\left(S(n)\frac{A(B(D(n)))}{A(D(n)/2)}\right)\right]/1.$$

Now following this outline:

1. Define circuits C_m that prove `SPACE TMSAT` for length m inputs.

   Then `SPACE TMSAT` on length m inputs has a **PCP** protocol with verifier time $\tilde{O}(m)$, log of proof length $\tilde{O}(m)$, and prover space $\tilde{O}(m)$. Then for some strictly increasing $B(m) = \tilde{O}(m)$, the prover for `SPACE TMSAT` on length m inputs can be reduced to a circuit for `SPACE TMSAT` with length $B(m)$ inputs. Then `SPACE TMSAT` on length $B(m)$ inputs has a circuit, C_m, of size at most $O(A(B(m)))$. We can also take $B(m) = \Omega(m)$ so that $B(m) = \tilde{\Theta}(m)$.

2. Define when the advice bit should be 1.

   Since `SPACE TMSAT` $\notin$ **SIZE**$[o(A(n))]$, for some $c_1 > 0$, for some infinite set, U', for all $n' \in U'$, language `SPACE TMSAT` on length n' inputs does not have circuits with size $c_1 A(n')$.

   Let the advice bit be 1 if and only if each of the following hold:

   (a) `SPACE TMSAT` on length m inputs does not have circuits with size at most $c_1 A(m/2)$.
   This restricts us to m where the circuits for `SPACE TMSAT` require size near our upper bound. This limits how much bigger C_m needs to be than the circuits for `SPACE TMSAT` on length m inputs.

   (b) `SPACE TMSAT` on length m inputs does not have a circuit with size $S(2^{l-1})$. Note $S(2^{l-1}) \geq S(n/4)$.

   (c) `SPACE TMSAT` on length m inputs does have a circuit with size $S(2^l)$.

   (d) Circuit C_m has size $S(2^l)2^t$.

   (e) Either $t = 0$, or C_m does not have size $S(2^l)2^{t-1}$.

   Then $x \in W$ for some x with $|x| = n$ if and only the advice bit is 1 and for some $y \in$ `SPACE TMSAT` with $|y| = m$ we have $x = y1^{n-m}$.

3. Now we will argue that infinitely often the advice bit is 1 and W does not have circuits with size $S(n/4)$. We do this in a few steps:

   - First restrict our focus to m large enough and where `SPACE TMSAT` on length m inputs has size near $A(m/2)$. This will be the set of input lengths, U.
     Since, by theorem premise, $S(2n) = o(A(n))$, for some n_2, for all $n' > n_2 : S(2n') < c_1 A(n')$.
     Since, by theorem premise, $S(n)2^n = \omega(A(B(n)))$, and C_n has size at most $O(A(B(n)))$, we have $|C_n| = o(S(n)2^n)$. So for some n_3, for all $n' > n_3$, circuit $C_{n'}$ has size $S(n')2^{n'-1}$.
     Take U^* to be the $n' \in U'$ larger than $\max\{n_1, n_2, n_3\}$. See that U^* is still an infinite set. For each length $n' \in U^*$, we will find a length $n > n'$ so the advice bit is 1.
     For $n' \in U^*$, let $m = 2^{l'}$ be the smallest power of 2 greater than n'. That is, $m > n'$, but $2n' \geq m$.
     By choice of $U^* \subseteq U'$, language `SPACE TMSAT` on length n' inputs does not have circuits of size $c_1 A(n')$. Recall that the min circuit length for `SPACE TMSAT` is monotone (see Lemma 3.0.2), so since $m > n'$, language `SPACE TMSAT` on length m inputs does not have circuits of size $c_1 A(n')$.
     Since A is monotone and $m \leq 2n'$, we know $c_1 A(m/2) \leq c_1 A(n')$. Since $n' > n_2$, we know $S(2n') < c_1 A(n')$. Since S is monotone and $m \leq 2n'$, we have $S(m) \leq S(2n')$. So we know $S(m) \leq c_1 A(n')$. Then since `SPACE TMSAT` on length m inputs does not have circuits of size $c_1 A(n')$, we also have `SPACE TMSAT` on length m inputs does not have circuits of size $S(m)$. Similarly, since $n' \geq m/2$ and A is monotone, `SPACE TMSAT` on length m inputs does not have circuits of size $c_1 A(m/2)$.
     Let U be the set of m from each $n' \in U^*$. See that U is an infinite set since for each $n' \in U^*$, there is an $m \in U$ greater than n', and U^* is an infinite set. Then for $m \in U$, language `SPACE TMSAT` on length m inputs does not have circuits of size $S(m)$ or $c_1 A(m/2)$ and $m > \max\{n_1, n_2, n_3\}$.
   - For each $m \in U$, find appropriate l and t.
     Take the smallest l so that `SPACE TMSAT` on length m inputs does have a circuit of size $S(2^l)$. Note that $l > l' = \log(m)$, since `SPACE TMSAT` on length m inputs does not have circuits with size $S(m)$.
     Let t be the smallest t such that C_m has size $S(2^l)2^t$. Since $m > n_3$, we know C_m has size at most $S(m)2^{m-1} < S(2^l)2^{m-1}$. Thus $t \leq m - 1 < m$.

- Now for $n = 2^l + m + t$, we show the advice bit is 1 and language `SPACE TMSAT` on length n inputs does not have circuits with size $S(n/4)$.

  First, see that $t < m$, so $m + t < 2^{l'+1}$. As noted before, $l' < l$, so $2^{l'+1} \leq 2^l$. Thus $2^l > m + t$ and $\rho(n) = l$. Similarly $l' = \rho(\mu(n))$, $m = 2^{l'}$, and $t = \mu(\mu(n))$. Then

  (a) By choice of U, language `SPACE TMSAT` on length m inputs does not have circuits of size $c_1 A(m/2)$.
  (b) `SPACE TMSAT` on length m inputs does not have a circuit with size $S(2^{l-1})$, since we chose the smallest l so that `SPACE TMSAT` on length m inputs has a circuit with size $S(2^l)$.
  (c) For the same reason, `SPACE TMSAT` on length m inputs does have a circuit with size $S(2^l)$,
  (d) By choice of t, circuit C_m has size $S(2^l)2^t$.
  (e) Specifically, t is the smallest such that C_m has size $S(2^l)2^t$. So either $t = 0$, or C_m does not have size $S(2^l)2^{t-1}$.

  So for that n, the advice bit is 1.

  Since for every $m \in U$ for some $n > m$ the advice bit is 1, and since U is an infinite set, the advice bit is one infinitely often. For input lengths where the advice bit is 1, `SPACE TMSAT` does not have circuits of size $S(2^{l-1}) \geq S(n/4)$. So `SPACE TMSAT` does not have circuits of size $S(n/4)$ infinitely often. Therefore

$$W \notin \mathbf{SIZE}[o(S(n/4))].$$

4. Show that

$$W \in \mathbf{MATIME}\left[\tilde{O}\left(S(n)\frac{A(B(D(n)))}{A(D(n)/2)}\right)\right]/1.$$

If the advice bit is 0, this is trivial. Otherwise, assume for n the advice bit is 1.

When the advice bit is 1, we know C_m has size at most $S(2^l)2^t$ and either

$t = 0$: Then C_m has size $S(2^l) = O(S(n))$.

$t \geq 1$: Then C_m does not have size $S(2^l)2^{t-1}$ by choice of t. Circuit C_m has size $A(B(m))$. Thus

$$S(2^l)2^{t-1} < A(B(m)).$$

Further, `SPACE TMSAT` on length m inputs does not have circuits with size $c_1 A(m/2)$ since the advice bit is 1, but it does have circuits with size $S(2^l)$. Thus

$$c_1 A(m/2) < S(2^l).$$

Together

$$\begin{aligned} S(2^l)2^{t-1} &< A(B(m)) \\ c_1 A(m/2)2^{t-1} &< A(B(m)) \\ 2^t &< \frac{2}{c_1}\frac{A(B(m))}{A(m/2)}. \end{aligned}$$

Thus C_m has size

$$S(2^l)2^t = O\left(S(n)\frac{A(B(D(n)))}{A(D(n)/2)}\right).$$

The verifier for `SPACE TMSAT` can be simulated in time $\tilde{O}(m)$, and the $\mathbf{poly}(\log(m))$ queries to the prover can be simulated in time

$$\mathbf{poly}(\log(m))S(2^l)2^t = \tilde{O}\left(S(n)\frac{A(B(D(n)))}{A(D(n)/2)}\right).$$

This gives a total **MA** time of $\tilde{O}\left(S(n)\frac{A(B(D(n)))}{A(D(n)/2)}\right)$ for W. Thus

$$W \in \mathbf{MATIME}\left[\tilde{O}\left(S(n)\frac{A(B(D(n)))}{A(D(n)/2)}\right)\right]/1.$$

Therefore

$$W \in \mathbf{MATIME}\left[\tilde{O}\left(S(n)\frac{A(B(D(n)))}{A(D(n)/2)}\right)\right]/1 \setminus \mathbf{SIZE}[o(S(n/4))].$$

$\square$

Now for the special case where `SPACE TMSAT` almost has some fixed polynomial size.

Corollary 3.4.2 (Bound if `SPACE TMSAT` has size $n^{a+o(1)}$)**.** *Suppose for some function $h(n)$ with $|h(n)| = o(1)$ and for some constant $a > 1$, for some function $A(n)$ we have $A(n) = n^{a+h(n)}$. Then if $A(n)$ is non-decreasing and we have* `SPACE TMSAT` $\in \mathbf{SIZE}[O(A(n))] \setminus \mathbf{SIZE}[o(A(n))]$, *then for any $k < a$, for some $f(n) = o(1)$,*

$$\mathbf{MATIME}[O(n^{k+f(n)})]/1 \not\subset \mathbf{SIZE}[O(n^k)].$$

Proof. If $k < 1$, we use Corollary 2.0.6. Otherwise, let $S(n) = n^k \log(n)$. Since $k < a$, we have $S(2n) = o(A(n))$.

To apply Lemma 3.4.1, we need to show that for any $B(n) = \tilde{\Theta}(n)$, we have $S(n)2^n = \omega(A(B(n)))$. But since $A(n)$ and $B(n)$ are both polynomials, they are smaller than 2^n. That is

$$\begin{aligned} A(B(n)) =& o(B(n)^{k+1/2}) \\ =& o(2^n) \\ =& o(S(n)2^n). \end{aligned}$$

This is equivalent to $S(n)2^n = \omega(A(B(n)))$.

Now we can apply Lemma 3.4.1 to get a language W such that

$$W \in \mathbf{MATIME}\left[\tilde{O}\left(n^k \log(n)\frac{A(B(D(n)))}{A(D(n)/2)}\right)\right]/1 \setminus \mathbf{SIZE}[o(n^k \log(n))].$$

Now let's simplify this a bit. Since $W \notin \mathbf{SIZE}[o(n^k \log(n))]$ and $n^k = o(n^k \log(n))$, we have $W \notin \mathbf{SIZE}[O(n^k)]$.

Now we want to bound that fraction:

$$\frac{A(B(D(n)))}{A(D(n)/2)} = \frac{(B(D(n)))^{a+h(B(D(n)))}}{(D(n)/2)^{a+h(D(n)/2)}}.$$

We start by letting $D(n) = m$ and bounding this in terms of m first. Then

$$\begin{aligned} \frac{A(B(D(n)))}{A(D(n)/2)} =& \frac{(B(m))^{a+h(B(m))}}{(m/2)^{a+h(m/2)}} \\ =& \tilde{O}\left(\frac{m^{a+h(B(m))}}{m^{a+h(m/2)}}\right) \\ =& \tilde{O}\left(m^{h(B(m))-h(m/2)}\right). \end{aligned}$$

Since $B(m) = \omega(1)$, and $|h(m)| = o(1)$, we know $|h(B(m))| = o(1)$. So for some $h^*(m)$ with $|h^*(m)| = o(1)$, we have

$$\begin{aligned} \frac{A(B(D(n)))}{A(D(n)/2)} =& O(m^{h^*(m)}) \\ =& O(D(n)^{h^*(D(n))}). \end{aligned}$$

Note that since A and B are both non-decreasing, this fraction is at least 1. So in particular $h^*(n) \geq 0$, and $h^*(n) = o(1)$.

Now using Lemma 2.0.9, since $D(n) = O(n)$, for some $h'(n) = o(1)$, we have

$$\frac{A(B(D(n)))}{A(D(n)/2)} = O(n^{h'(n)}).$$

Thus for some $f(n) = o(1)$, we have

$$\begin{aligned}\tilde{O}\left(n^k \log(n)\frac{A(B(D(n)))}{A(D(n)/2)}\right) =&\tilde{O}\left(n^k n^{h'(n)}\right)\\ =&O(n^{k+f(n)}).\end{aligned}$$

So $W \in \mathbf{SIZE}[O(n^{k+f(n)})]$. Thus we conclude:

$$\mathbf{MATIME}[O(n^{k+f(n)})]/1 \not\subset \mathbf{SIZE}[O(n^k)].$$

$\square$

3.5 Altogether

Altogether, these three cases imply Theorem 1.1.1.

Theorem 1.1.1 (Fine Grained **MA** Lower Bound)**.** *There exists a constant $a > 1$, such that for all $k < a$, for some $f(n) = o(1)$,*

$$\mathbf{MATIME}[O(n^{k+f(n)})]/1 \not\subset \mathbf{SIZE}[O(n^k)].$$

Proof. First, we will find the best polynomial approximation of the circuit complexity of `SPACE TMSAT`. So define set

$$S = \{a \in \mathbb{R} : \texttt{SPACE TMSAT} \in \mathbf{SIZE}[O(n^a)]\}.$$

If $S = \emptyset$, then there is no constant a such that `SPACE TMSAT` $\in \mathbf{SIZE}[O(n^a)]$. Then `SPACE TMSAT` $\notin$ **P/poly**, so we use Corollary 3.2.3. Then Corollary 3.2.3 gives: for any $k > 0$, and some $f(n) = o(1)$:

$$\mathbf{MATIME}[O(n^{k+f(n)})]/1 \not\subset \mathbf{SIZE}[O(n^k)].$$

So suppose $S \neq \emptyset$. Now see that `SPACE TMSAT` requires circuits of size $O(n)$ since we can reduce parity to it and parity requires circuits of size $O(n)$ (see Lemma 2.0.5). Thus for any $a < 1$, we know that `SPACE TMSAT` $\notin \mathbf{SIZE}[O(n^a)]$, that is $a \notin S$. So 1 is a lower bound for S.

Then the set S is nonempty and has a lower bound. So S has an infimum, a, so that for any constant $\epsilon > 0$, we have `SPACE TMSAT` $\in \mathbf{SIZE}[O(n^{a+\epsilon})]$, but `SPACE TMSAT` $\notin \mathbf{SIZE}[O(n^{a-\epsilon})]$.

Before we use Corollary 3.3.2 and Corollary 3.4.2, we need to find an $A(n)$ such that for some $h(n)$, we have $A(n) = n^{a+h(n)}$ and

1. $A(n)$ is non-decreasing.
2. `SPACE TMSAT` $\in \mathbf{SIZE}[O(A(n))] \setminus \mathbf{SIZE}[o(A(n))]$.
3. $|h(n)| = o(1)$.

Let $A'(n)$ be the minimum circuit size of `SPACE TMSAT` on length n inputs. One might hope $A'(n)$ would work for $A(n)$, but the difficulty of `SPACE TMSAT` may not increase smoothly. It may remain near linear for many consecutive n, and only occasionally increase near n^a. So instead, we want a smoother function for $A(n)$ that never drops too far below n^a, but infinitely often is equal to $A'(n)$.

So the idea is just to have $A(n)$ be the maximum of $A'(n)$ and some polynomial just smaller than n^a, say $n^{a-\epsilon}$. Then $A(n)$ won't get far away from n^a if $A'(n)$ becomes small. But we can't use a constant ϵ, or we could get $|h(n)| = \Omega(1)$. So instead, we make ϵ smaller each time $A'(n)$ is larger than $n^{a-\epsilon}$.

Define $m(n)$ so that $m(0) = 0$ and

$$m(n+1) = \begin{cases} m(n)+1 & A'(n) \geq n^{a-2^{-m(n)}} \\ m(n) & \text{otherwise} \end{cases}.$$

Then $\epsilon(n) = 2^{-m(n)}$.

Now we define $A(n) = \max\{A'(n), n^{a-\epsilon(n)}\}$. Then for $h(n) = \frac{\log(A(n))}{\log(n)} - a$, we have $A(n) = n^{a+h(n)}$. Now we show the three conditions.

1. $A(n)$ is non-decreasing.

   $A(n)$ is the maximum of two non-decreasing sequences: $A'(n)$ and $n^{a-\epsilon(n)}$, so is also non-decreasing.

2. SPACE TMSAT $\in$ **SIZE**$[O(A(n))] \setminus$ **SIZE**$[o(A(n))]$.

   By choice of $A(n)$, for all n, $A(n) \geq A'(n)$, the minimum circuit size of SPACE TMSAT, so SPACE TMSAT $\in$ **SIZE**$[O(A(n))]$.

   Now we will argue that infinitely often, $A(n) = A'(n)$. Otherwise, for some n', for all $n \geq n'$, $A'(n) < n^{a-\epsilon(n)}$. If this were true, then for any $n \geq n'$, $m(n) = m(n')$ since for none of these n will $m(n)$ increase. Thus for all $n > n'$, $A'(n) < n^{a-\epsilon(n')}$. Then SPACE TMSAT $\in$ **SIZE**$[O(n^{a-\epsilon(n')})]$. But since $\epsilon(n') > 0$, by choice of a, this cannot happen. Contradiction. So infinitely often, $A(n) = A'(n)$.

   Thus infinitely often, SPACE TMSAT requires circuits of size $A(n)$, thus SPACE TMSAT $\notin$ **SIZE**$[o(A(n))]$.

3. $|h(n)| = o(1)$.

   From the last section, infinitely often, $A'(n) \geq n^{a-\epsilon(n)}$, so $m(n) \to \infty$, and for $h_1(n) = \epsilon(n) = o(1)$, we have a lower bound on $A(n)$ of $A(n) \geq n^{a-h_1(n)}$.

   Let $h_2(n) = \max\{0, \frac{\log(A'(n))}{\log(n)} - a\}$. See that $n^{a+h_2(n)}$ is always at least $n^{a-\epsilon(n)}$ and $A'(n)$, so $A(n) \leq n^{a+h_2(n)}$. Next we show that $h_2(n) = o(1)$.

   Suppose otherwise. Then for some $c > 0$, for infinitely many n, we have $h_2(n) > c$. But for such n, we have $h_2(n) = \frac{\log(A'(n))}{\log(n)} - a$, so
   $$A'(n) = n^{a+h_2(n)} > n^{a+c}.$$

   Then infinitely often, SPACE TMSAT does not have size n^{a+c} circuits. This means SPACE TMSAT $\notin$ **SIZE**$[o(n^{a+c})]$. Specifically, for $c/2 > 0$, we have SPACE TMSAT $\notin$ **SIZE**$[O(n^{a+c/2})]$. But choice of a, this cannot happen. Contradiction. So $h_2(n) = o(1)$.

   Thus
   $$\begin{aligned} n^{a-h_1(n)} \leq \quad & A(n) \leq & n^{a+h_2(n)} \\ a - h_1(n) \leq \quad & \frac{\log(A(n))}{\log(n)} \leq & a + h_2(n) \\ -h_1(n) \leq \quad & h(n) \leq & h_2(n) \\ & |h(n)| \leq & \max\{h_1(n), h_2(n)\} \\ & = & o(1). \end{aligned}$$

If $a = 1$, we use Corollary 3.3.2. See that $|h(n)| = o(1)$ and SPACE TMSAT $\in$ **SIZE**$[O(n^{1+|h(n)|})]$. Thus for any k, for some $f(n) = o(1)$, we have

$$\mathbf{MATIME}[O(n^{k+f(n)})]/1 \not\subset \mathbf{SIZE}[O(n^k)].$$

If $a > 1$, we use Corollary 3.4.2. See that A was specifically constructed to satisfy the theorem requirements. Then for any $k < a$, for some $f(n) = o(1)$, we have

$$\mathbf{MATIME}[O(n^{k+f(n)})]/1 \not\subset \mathbf{SIZE}[O(n^k)].$$

□

To prove Theorem 1.1.2, we use the case of $a > 1$ in the proof of Theorem 1.1.1 and apply Lemma 3.3.1 similar to Corollary 3.3.2.

Theorem 1.1.2 (**MA** Lower Bound for Small a)**.** *If the a from Theorem 1.1.1 is finite, then for all $k > 0$, for some $f(n) = o(1)$,*
$$\mathbf{MATIME}[O(n^{ak+f(n)})]/1 \not\subset \mathbf{SIZE}[O(n^k)].$$

Proof. We want to use Lemma 3.3.1 with $S(n) = n^k \log(n)$, $A(n) = n^{a+h(n)}$ from the proof of Theorem 1.1.1, and $B(n) = \tilde{\Theta}(n)$ from Lemma 3.3.1.

The size upper bound on $S(n)$ is clear: $S(n) = o(2^n/n)$. We need to show $S(n)2^n = \omega(A(B(S(n))))$. Well for any $B(n) = \tilde{O}(n)$,

$$\begin{aligned} A(B(S(n))) =& B(n^k \log(n))^{a+h(n)} \\ =& \tilde{O}(n^{ak+kh(n)}) \\ =& o(2^n) \\ S(n)2^n =& \omega(A(B(S(n)))). \end{aligned}$$

So by Lemma 3.3.1, for some $B(n) = \tilde{O}(n)$,

$$\mathbf{MATIME}[\tilde{O}(A(B(S(n))))]/1 \not\subset \mathbf{SIZE}[o(S(n/2))].$$

See that for some $f(n) = o(1)$,

$$\tilde{O}(A(B(S(n)))) = \tilde{O}(n^{ak+kh(n)}) = O(n^{ak+f(n)}).$$

Similarly $n^k = o(S(n/2))$, so we also have

$$\mathbf{MATIME}[O(n^{ak+f(n)})]/1 \not\subset \mathbf{SIZE}[O(n^k)].$$

□

4 Extrapolatable PCPs

We introduce extrapolatable **PCP**s (**ePCP**s) as an intermediate **PCP** in constructing an efficient **rPCP**. We will later use **rPCP**s in **PCP** composition to reduce the number of queries. Before defining an **ePCP**, **rPCP** or **PCP** composition, we start by introducing several useful properties about extrapolatability.

4.1 Extrapolatable Functions

We defined extrapolatable functions to help construct functions we can compute a low degree extrapolation of efficiently. For any $Q : [q] \to \mathbb{F}^n$, we say its low degree extrapolation is the unique degree $q-1$ function that agrees with Q on its first q values. Any Q computable in time $n\mathbf{polylog}(|\mathbb{F}|)$, we can compute the extrapolation of Q in time $qn\mathbf{polylog}(|\mathbb{F}|)$. But we want to compute the extrapolation of Q in time $(q+n)\mathbf{polylog}(|\mathbb{F}|)$. Recall the definition of an extrapolatable function.

Definition 1.2.1 (Extrapolatability)**.** *For any $n, q, t > 0$, and field $\mathbb{F}$, we call $Q : [q] \to \mathbb{F}^n$ "t extrapolatable" (or time t extrapolatable) if there is a time t algorithm taking any $v \in \mathbb{F}^q$, that outputs*

$$\sum_{i \in [q]} v_i Q(i).$$

We will use Q where $t \leq (q+n)\mathbf{polylog}(|\mathbb{F}|)$.

Various basic combinations of extrapolatable functions give extrapolatable functions.

The function that outputs one extrapolatable function for its first m inputs, and then a second extrapolatable function for its last m' inputs is also extrapolatable.

Lemma 4.1.1 (Extrapolatability Combination 1)**.** *For integers $n, q, q', t, t' > 0$, and field $\mathbb{F}$, if $p : [q] \to \mathbb{F}^n$ is t extrapolatable, and $p' : [q'] \to \mathbb{F}^n$ is t' extrapolatable, then $g : [q+q'] \to \mathbb{F}^n$ is $O(t + t' + n\log(\mathbb{F}))$ extrapolatable where*

$$g(i) = \begin{cases} p(i) & i \leq q \\ p'(i-q) & i > q \end{cases}.$$

Proof. To prove $g(i)$ is extrapolatable, we need an algorithm that takes $v \in \mathbb{F}^{q+q'}$ and outputs

$$\sum_{i \in [q+q']} v_i g(i).$$

We can write this sum as

$$\begin{aligned}\sum_{i \in [q+q']} v_i g(i) &= \sum_{i \in [q]} v_i g(i) + \sum_{i \in [q']} v_{q+i} g(q+i) \\ &= \sum_{i \in [q]} v_i p(i) + \sum_{i \in [q']} v_{q+i} p'(i).\end{aligned}$$

Then use extrapolatability of p to calculate

$$\sum_{i \in [q]} v_i p(i)$$

in time t, and use extrapolatability of p' to calculate

$$\sum_{i \in [q']} v_{q+i} p'(i)$$

in time t'. Then their sum is the answer, and addition takes time $O(n \log(\mathbb{F}))$. □

Similarly, a function that outputs pairs of values from extrapolatable functions is extrapolatable.

Lemma 4.1.2 (Extrapolatability Combination 2)**.** *For integers $n, n', q, t, t' > 0$, and field $\mathbb{F}$, if $p : [q] \to \mathbb{F}^n$ is t extrapolatable, and $p' : [q] \to \mathbb{F}^{n'}$ is t' extrapolatable, then $g : [q] \to \mathbb{F}^{n+n'}$ is $O(t + t')$ extrapolatable where*

$$g(i) = (p(i), p'(i)).$$

Proof. Given $v \in \mathbb{F}^q$, we need to calculate

$$\begin{aligned}\sum_{i \in [q]} v_i g(i) &= \sum_{i \in [q]} v_i (p(i), p'(i)) \\ &= \left(\sum_{i \in [q]} v_i p(i), \sum_{i \in [q]} v_i p'(i) \right).\end{aligned}$$

We use extrapolatability of p to calculate

$$\sum_{i \in [q]} v_i p(i)$$

in time t, then use extrapolatability of p' to calculate

$$\sum_{i \in [q]} v_i p'(i)$$

in time t'. Then concatenate the results. □

As an example of extrapolatable functions, see that any function outputting an arithmetic progression is extrapolatable.

Lemma 4.1.3 (Arithmetic Progressions are Extrapolatable)**.** *For integers $n, q > 0$, and field $\mathbb{F}$, for any $x \in \mathbb{F}^n$ and $y \in \mathbb{F}^n$, the function $f : [q] \to \mathbb{F}^n$ defined by*

$$f(i) = x + iy$$

is time $O((n + q)\textbf{polylog}(|\mathbb{F}|))$ extrapolatable.

Proof. Given $v \in \mathbb{F}^q$, we need to calculate

$$\begin{aligned}\sum_{i\in[q]} v_i f(i) &= \sum_{i\in[q]} v_i(x+iy)\\ &= \left(\sum_{i\in[q]} v_i\right)x + \left(\sum_{i\in[q]} v_i i\right)y.\end{aligned}$$

Then one can calculate $\alpha = \sum_{i\in[q]} v_i$ using just q field additions, which takes time $O(q\log(|\mathbb{F}|))$. Similarly, one can calculate $\beta = \sum_{i\in[q]} v_i i$ using q multiplications and additions, which takes time $O(q\textbf{polylog}(|\mathbb{F}|))$.

Now we need to calculate $\alpha x + \beta y$. Since $x \in \mathbb{F}^n$, it only n field operations to multiply x by α, so αx only takes time $O(n\textbf{polylog}(|\mathbb{F}|))$ to calculate. Similar for βy and the sum of of αx with βy.

So altogether, this algorithm only takes time $O((q+n)\textbf{polylog}(|\mathbb{F}|)$ to compute $\sum_{i\in[q]} v_i f(i)$. □

Now we show that we can efficiently extrapolate (compute the low degree extrapolation of) an extrapolatable function.

Lemma 4.1.4 (Efficient Polynomials From Extrapolatability)**.** *For any $n, q, t > 0$, field $\mathbb{F}$ where $|\mathbb{F}| > q$, and t extrapolatable $Q : [q] \to \mathbb{F}^n$, there is a time*

$$O(t + q\textbf{polylog}(|\mathbb{F}|))$$

algorithm computing the value of a degree $q-1$ polynomial, g, such that for all $i \in [q]$

$$g(i) = Q(i).$$

Proof. We use Lagrange interpolation. For a given q, and $i \in [q]$, the ith Lagrange basis polynomial is:

$$l_i^q(x) = \prod_{j\in[q]\setminus\{i\}} \frac{x-j}{i-j}.$$

This is the degree $q-1$ polynomial that is 1 at $x = i$, but 0 for all other $x \in [q] \setminus \{i\}$.

Then we can easily write our desired g in terms of the Lagrange basis polynomials:

$$g(x) = \sum_{i\in[q]} l_i^q(x)Q(i).$$

A naive, straightforward evaluation of this sum takes time $O(nq\textbf{polylog}(|\mathbb{F}|))$. But since Q is t extrapolatable, if we can calculate $l_1^q(x), \ldots, l_q^q(x)$, we can use these to calculate g in time t.

For a fixed x, and i, we can define

$$\begin{aligned}\alpha_i &= \prod_{j\in[i-1]} (x-j)\\ \alpha_i' &= \prod_{j\in[q]\setminus[i]} (x-j)\\ \beta_i &= \prod_{j\in[i-1]} j\\ \beta_i' &= \prod_{j\in[q-i]} (-j)\end{aligned}$$

so that

$$\begin{aligned}l_i^q(x) &= \prod_{j\in[q]\setminus\{i\}} \frac{x-j}{i-j}\\ &= \frac{\alpha_i\alpha_i'}{\beta_i\beta_i'}.\end{aligned}$$

Each one of these sequences $(\alpha, \alpha', \beta, \beta')$ can be entirely computed in time $O(q\textbf{polylog}(|\mathbb{F}|))$. For instance, see that for $i < q$, $\alpha_{i+1} = (x-i)\alpha_i$, which can be computed with two field operations. So all q of the α_i can be computed in time $O(q\textbf{polylog}(|\mathbb{F}|))$. Similarly for α', β, and β'.

Now given each of α, α', β, and β' have already been calculated, we can calculate $l_i^q(x)$ in four field operations. Thus, every $l_1^q(x), \ldots, l_q^q(x)$ can be calculated in time $O(q\textbf{polylog}(|\mathbb{F}|))$.

Finally, since Q is time t extrapolatable, we can calculate $g(x)$ in time t, giving a total time of $O(t + q\textbf{polylog}(|\mathbb{F}|))$. □

4.2 Robust PCPs and Extrapolatable PCPs

The purpose of an **ePCP** is to give an easy, efficient way to construct a robust **PCP** (**rPCP**). This construction uses low degree testing. So before we define **ePCP** and show how to convert one to a **rPCP**, we first define **rPCP** and review low degree testing.

Loosely, a robust **PCP** is a **PCP** so that when $x \notin L$, for any proof, most sets of queries to that proof return not only a rejected response, but a response that is far from any accepted response.

To formally define a robust **PCP**s, we need to define Hamming distance.

Definition 4.2.1 (Distance)**.** *For $x, y \in \Sigma^n$, define distance by the function, Δ:*

$$\Delta(x, y) = \frac{\sum_{i \in [n]} 1_{x_i \neq y_i}}{n} = \Pr_{i \in [n]}[x_i \neq y_i].$$

For $Y \subset \Sigma^n$, define

$$\Delta(x, Y) = \min_{y \in Y} \Delta(x, y).$$

The only difference between a **PCP** (see Definition 2.0.8) and an **rPCP** is a strengthening of the of soundness to robust soundness. Now we formally define an **rPCP**.

Definition 4.2.2 (Robust **PCP**)**.** *For language L with a non-adaptive* **PCP** *protocol, A, with verifier V, index function I, and alphabet Σ, we say A is a robust* **PCP** *(***rPCP***) if:*

Robust Soundness *If $x \notin L$ then for every π', for $Y_r = \{\sigma : V(x, r, \sigma) = 1\}$,*

$$E_r[\Delta(\pi'_{I(x,r)}, Y_r)] \geq 1 - \delta.$$

Completeness *If $x \in L$ and $n = |x|$, then there exists $\pi^x \in \Sigma^{l(n)}$ such that*

$$\Pr_r[V(x, r, \pi^x_{I(x,r)}) = 1] = 1.$$

Then we say A has robust soundness δ.

Now we introduce extrapolatable **PCP**s (**ePCP**s) as an intermediate between a **PCP** and an **rPCP**. An **ePCP** is a **PCP** where:

1. An honest **PCP** proof is a low degree polynomial: $\pi : \mathbb{F}^m \to \mathbb{F}$.

   This allows us to make the **PCP** robust using an aggregation through curves type technique.

2. We relax soundness to only be against low degree proofs.

   This makes constructing **ePCP**s easier, since it lets us assume proofs are low degree functions, and low degree polynomials are error correcting codes.

3. The query function is extrapolatable (see Definition 1.2.1).

   This makes the query locations of the robust **PCP** efficient to compute individually.

Now we formally define an extrapolatable **PCP** (see standard **PCP**s, Definition 2.0.8, for reference).

Definition 4.2.3 (Extrapolatable **PCP**)**.** *We say a non-adaptive* **PCP***, A, for language L with verifier V, prover P, and query function Q is an extrapolatable* **PCP** *(***ePCP***) if for some m and d:*

1. *For some field $\mathbb{F}$, A uses alphabet $\mathbb{F}$.*
2. *The proof length is $|\mathbb{F}|^m$.*
   That is, any proof, π, can be viewed as a function $\pi : \mathbb{F}^m \to \mathbb{F}$.

Low Degree Completeness *If $x \in L$ and $n = |x|$, then there exists a polynomial $\pi^x : \mathbb{F}^n \to \mathbb{F}$ of degree at most d such that*

$$\Pr_r[V(x, r, \pi^x(I(x, r))) = 1] = 1,$$

and for every $i \in [l(n)]$, we have $P(x, i) = \pi^x_i$.

Low Degree Soundness *If $x \notin L$ then for every polynomial $\pi' : \mathbb{F}^n \to \mathbb{F}$ of degree at most d has*

$$\Pr_r[V(x, r, \pi'(I(x, r))) = 1] \leq \delta.$$

Further, we say A has:

1. *Extrapolation time $t(n)$ if for any x, r, the function $Q_{x,r}(i) = Q(x, r, i)$ is time $t(n)$ extrapolatable.*
2. *Degree d and m variables.*
3. *Low degree soundness δ.*
4. *Perfect low degree completeness.*

4.3 Low Degree Testing

Low degree testing checks if there is a global low degree polynomial a proof is close to. We use this to find a low degree proof for an **ePCP** if a proof for our **rPCP** is too often close to accepting inputs.

We use the "line versus point" low degree test. Our definition of the line versus point test has more redundancy than necessary (we allow multiple claimed polynomials per line), but this is equivalent and simplifies our analysis.

Definition 4.3.1 (Line versus Point test)**.** *Let $\mathbb{F}$ be a field, f be a function $f : \mathbb{F}^m \to \mathbb{F}$, and degree d be an integer. For each line given by $l : \mathbb{F} \to \mathbb{F}^m$, let there be a degree d polynomial $g_l : \mathbb{F} \to \mathbb{F}$.*

The line vs point test uniformly samples a line given by, $l : \mathbb{F} \to \mathbb{F}^m$ and a uniform $t \in \mathbb{F}$ then accepts if and only if

$$f(l(t)) = g_l(t).$$

Let $LvP_d(f)$ be the random variable that this test fails on function f for the set of g_l that fails with the lowest probability.

The failure probability of the line versus point test is related to the distance to a low degree polynomial [AS97]. We will be using the result from [FS95].

Lemma 4.3.2 (Line vs Point Test Measures Distance to Degree)**.** *For some constant c, for any integer d and field $\mathbb{F}$ with $|\mathbb{F}| \geq cd$, for any function $f : \mathbb{F}^m \to \mathbb{F}$, if*

$$\Pr[LvP_d(f)] \leq 0.12,$$

then there exists a degree d polynomial g so that

$$\Delta(f, g) \leq 2 \Pr[LvP_d(f)].$$

If f is a degree d polynomial, then

$$\Pr[LvP_d(f)] = 0.$$

4.4 Extrapolatable PCPs to Robust PCPs

The idea is to take an **ePCP**, and instead of querying points, query all the points along a curve that goes through those points. Since low degree functions are an error correcting code, restricting low degree proofs to a low degree curve gives an error correcting code. So by querying entire curves, we can make the set of accepted query values for our **PCP** verifier an error correcting code.

Querying along a line and checking if it is low degree performs a low degree test. A low degree test only succeeds with high probability if a proof is close to a global, low degree polynomial. Then since low degree polynomials are error correcting codes, if the query values are close to both the global low degree polynomial and an accepted proof, the accepted proof is the global low degree polynomial. If the query values for a proof are close to being accepted often, we show a global low degree proof for the original **ePCP** succeeds often.

Then the idea of the protocol is to choose the randomness for the **ePCP**, take a curve through all the query points of the **ePCP**, query all the locations along this curve and check if the the curve is low degree, and the **ePCP** accepts the proof on this curve. This is almost what the **rPCP** does, with a few caveats:

1. We also perform a robust line versus point test. This is just like a regular line versus point test, except we check every point along the line. This gives our line versus point test robustness since low degree polynomials are an error correcting code.

2. To guarantee the curve is consistent with the global low degree polynomial with high probability, we need a random point on the curve to be approximately uniform over $\mathbb{F}^m$. So we also choose another random point in the proof, and use a curve that goes through the **ePCP** queries and that point.

   From Lemma 4.1.1, the function going through all these points is still extrapolatable, and so by Lemma 4.1.4 we can efficiently compute the curve going through them.

Theorem 4.4.1 (**ePCP** gives efficient **rPCP**)**.** *For any language L with an* **ePCP***, A, with*

1. *Verifier time $t(n)$.*
2. *Verifier space $s(n)$.*
3. *Extrapolation time $t'(n)$.*
4. *Randomness $r(n)$.*
5. *Degree $d(n)$ and $m(n)$ variables.*
6. *$q(n)$ queries.*
7. *Alphabet $\mathbb{F}$ where $|\mathbb{F}| > 10q(n)d(n)$.*
8. *Prover P.*
9. *Low degree soundness 0.1.*
10. *Perfect low degree completeness.*

Language L has an **rPCP***, B, with*

1. *Verifier time $O(t(n) + |\mathbb{F}|^3 \mathbf{polylog}(|\mathbb{F}|))$.*
2. *Verifier space $O(s(n) + \log(|\mathbb{F}|))$.*
3. *Randomness $r(n) + O(m(n)\log(|\mathbb{F}|))$.*
4. *Query time $O(t'(n) + (q(n) + m(n))\mathbf{polylog}(|\mathbb{F}|))$.*
5. *$O(|\mathbb{F}|)$ queries.*
6. *Prover P with perfect completeness.*

7. *Soundness at most* 0.99.

Some of these parameters are not great, like verifier time, soundness, or number of queries, but they are good enough for our purposes. But it does have very good query time, space, and small enough randomness, which are important during composition.

Proof. Let V be the verifier, Q the query function, I the index function, and P the prover from **ePCP**, A. We construct a new **rPCP**, B, that expects the same low degree polynomial as proof as A. Our new verifier will be V' and our new query function Q'. We will start by describing our protocol from a high level, pointing out which parts are done by a new query function Q' and V' later.

First, on input x, and proof π, our **PCP** protocol B will choose the randomness for A, call it r. This determines the query points for A, which are $I(x, r)$. By assumption, $Q_{x,r}(i) = Q(x, r, i)$ is time $t'(n)$ extrapolatable.

Then B chooses some random $y \in \mathbb{F}^m$. See that the function taking 1 to y is time $O(m\mathbf{polylog}(|\mathbb{F}|))$ extrapolatable. Let $g : \mathbb{F} \to \mathbb{F}^m$ be the degree $q(n)$ function such that, for each $t \in [q(n)]$, we have $g(t) = Q(x, r, t)$, and $g(q(n) + 1) = y$.

Then $\pi \circ g$ is a degree $d' = dq$ polynomial if π is actually a degree d polynomial. Our new **rPCP** verifier V' will check every point along $\pi \circ g$ and verify it is a degree d' polynomial that would cause our **ePCP** verifier V to accept.

Next B chooses a random $z \in \mathbb{F}^m$ to run a robust line versus point test with line $l(i) = y + i \cdot z$. Altogether, B uses randomness $r' = (r, y, z)$ and $|r'| = r(n) + (2m(n)) \log(|\mathbb{F}|)$.

Then query function $Q' : [2|\mathbb{F}|] \to \mathbb{F}^m$ is defined by

$$Q'(x, r', i) = \begin{cases} g(i) & i \leq |\mathbb{F}| \\ l(i - |\mathbb{F}|) & i > |\mathbb{F}| \end{cases}$$

(where we define $g(|\mathbb{F}|) = g(0)$ and $l(|\mathbb{F}|) = l(0)$).

We call the first $|\mathbb{F}|$ queries the curve queries and the second $|\mathbb{F}|$ queries the line queries. Similarly, we call π evaluated on the first $\mathbb{F}$ queries the curve values and π on the second $\mathbb{F}$ queries the line values.

The verifier V' first checks if the **ePCP** would accept the curve values, that is, if $V(x, r, \pi_{I(x,r)}) = 1$. It can do this since the first q queries of Q' are the same as Q. Then the verifier checks if the curve values are a degree d' polynomial. Finally, it checks if the line values are a degree d polynomial. Our new verifier V' accepts only if all of these checks pass.

Now to keep verifier space down, we need to be a little careful how we implement our low degree test, so we describe that first. Let $f := \pi \circ g$ so that f is a function outputting the curve values. Using the degree d' interpolating polynomials,

$$l_i^{d'}(x) = \prod_{j \in [d'] \setminus \{i\}} \frac{x - j}{i - j},$$

we can write a degree d' polynomial, h:

$$h(x) = \sum_{i \in [d']} l_i^{d'}(x) f(i).$$

If f is a degree d' polynomial, then $f = h$. To see if $f = h$, we calculate h at each point and compare to f.

Each $l_i^{d'}$ can be computed directly by simply looping through each terms in the sums and products, calculating them from the definition, and reusing the space each time. Notably, we do NOT calculate the interpolating polynomials the same way we computed them in Lemma 4.1.4. That version uses more memory, but less time, and in this case we need less memory but allow for more time. Instead, we use the naive algorithm following the definition directly. We do a similar thing for the line versus point test.

Now to argue we achieve the stated performance.

1. Now we show the verifier time is $O(t(n) + |\mathbb{F}|^3\mathbf{polylog}(|\mathbb{F}|))$.

   The verifier time is just the time to simulate V, which is $t(n)$, plus the time it takes to perform the low degree tests. To test the low degree of f takes $O(|\mathbb{F}|)$ calculations of $h(x)$. Each $h(x)$ only takes

$O(d')$ calculations of $l_i^{d'}(x)$. Each $l_i^{d'}(x)$ only takes time $O(d'\mathbf{polylog}(|\mathbb{F}|))$. Thus the total time for the low degree test of f is

$$O(|\mathbb{F}|d'd'\mathbf{polylog}(|\mathbb{F}|)) = O(|\mathbb{F}|^3\mathbf{polylog}(|\mathbb{F}|)).$$

The line versus point checks take at most this long, so the overall time

$$O(t(n) + |\mathbb{F}|^3\mathbf{polylog}(|\mathbb{F}|).$$

2. Now we show the verifier space is $O(s(n) + \log(|\mathbb{F}|))$.

   Calculating a single $l_i^{d'}$ only requires keeping track of a constant number of field elements and a pointer for j. Then given that, $h(x)$ only needs the additional space for another counter for i and another field element. Finally, comparing all of the $h(x)$ to the $f(x)$ only takes space for another pointer for the x and another field element. So it only requires a constant number of pointers and field elements. Since $|\mathbb{F}| > d'$, this only requires $O(\log(|\mathbb{F}|))$ space.

   We do a similar thing for the line versus point test.

   So the total space of V' is the space used to run V plus $O(\log(|\mathbb{F}|))$. So the total space is $O(s(n) + \log(|\mathbb{F}|))$.

3. As already shown, B uses randomness $r(n) + (2m(n) + 1)\log(|\mathbb{F}|)$.

4. Next, we show the query time of the robust **PCP**.

   By assumption, the query locations of Q are time $t'(n)$ extrapolatable. And by Lemma 4.1.1, adding y gives a $O(t' + m\log(|\mathbb{F}|))$ extrapolatable function. And g is the low degree extrapolation of this sequence.

   By Lemma 4.1.4, we can calculate g in time $O(t'(n) + (m + q)\mathbf{polylog}(|\mathbb{F}|))$. This handles the curve queries, as these are just evaluations of g.

   The line queries just return a point in $l(i) = y + i \cdot z$. These can be calculated in $O(m\mathbf{polylog}(|\mathbb{F}|))$ time.

   In either case, we calculate Q' in time

   $$O(t'(n) + (m + q)\mathbf{polylog}(|\mathbb{F}|)).$$

5. The number of queries are $2|\mathbb{F}| = O(|\mathbb{F}|)$.

6. Now we need to show the proof provided by P has perfect completeness.

   This prover works, since by assumption, if $x \in L$, then P computes a proof, π^x, that V accepts and π^x has degree d. Then $\pi^x \circ g$ has degree d', and $\pi^x \circ l$ has degree d, so the low degree tests will also succeed. Thus with probability 1 will V' accept when given queries from π^x.

7. Now we need to show soundness 0.99.

   Let $q'(n) = 2|\mathbb{F}|$ be the number of queries made by our new protocol. Let $I'(x,r) = (Q'(x,r',i))_{i\in[q'(n)]}$ be the index function of our **rPCP**.

   We want to show that if $x \notin L$, then for any proof π, the expected distance of $\pi_{I'(x,r)}$ to any string that would make the verifier accept is more than 0.01. We prove the contrapositive.

   Let $Y_r = \{\sigma : V'(x,r,\sigma) = 1\}$. Then we want to show if there exists a proof $\pi : \mathbb{F}^m \to \mathbb{F}$ such that

   $$\mathrm{E}_r[\Delta(\pi_{I'(x,r)}, Y_r)] \leq 0.01$$

   then $x \in L$.

   Suppose $\mathrm{E}_r[\Delta(\pi_{I'(x,r)}, Y_r)] < 0.01$. First recall that our queries have 2 equal length parts: the curve and the line queries.

   The proof idea is the following:

(a) With high probability, for randomness r, individually, the curve and line queries agree on most points with a single proof window, $\sigma_r \in Y_r$, that is accepted on r.

(b) This implies a protocol for the line versus point test that frequently succeeds. So π is close to low degree proof π'.

(c) Then with good probability, π on curve queries will agree with π' at most places.

(d) Often, π on curve queries agree with σ_r and π' on most locations. So σ_r and π' are equal on the curve queries. So π' is accepted often by our **ePCP**, A.

(e) Then $x \in L$, since π' is a low degree proof that is accepted often by our **ePCP** and our **ePCP** has low degree soundness.

Let's follow this outline.

(a) Let C be the set of randomness r so that for some $\sigma_r \in Y_r$, the distance between the $\pi_{I(x,r)}$ and σ_r is at most 0.1. We want to show the probability $r \notin C$ is at most 0.1.

By definition of C, for any $r \notin C$, we have $\Delta(\pi_{I(x,r)}, Y_r) \geq 0.1$. Then see that

$$\begin{aligned} 0.01 \geq & \mathrm{E}_r[\Delta(\pi_{I(x,r)}, Y_r)] \\ \geq & \Pr[r \notin C]0.1 \\ 0.1 \geq & \Pr[r \notin C] \\ \Pr[r \in C] \geq & 0.9. \end{aligned}$$

Notice in particular that for $r \in C$, the distance between $\pi_{I(x,r)}$ and σ_r further restricted to the curve or line values is at most 0.2.

(b) Now our **PCP** encodes an implicit line versus point test that chooses a line, and checks a random point along it. We will use this to show that for some degree d function π', we have $\Delta(\pi, \pi') < 0.24$.

For a given randomness r, let l be the line Q' queries. Let p_l be the degree d function that agrees with the line values at the most places. Now we show that for $r \in C$, we have p_l is equal to the line values of σ_r.

For $r \in C$, some σ_r disagrees with π on the line values on at most 0.2 fraction of places. Thus p_l must also disagree with π on at most 0.2 fraction of places. Thus p_l and σ_r agree with each other on at least 0.6 fraction of places. So σ_r and p_l agree on more than d points. Since σ_r causes the verifier to accept, it's line values have degree at most d. So p_l and σ_r on the line values are degree d polynomials that agree on more than d places. So $p_l = \sigma_r$ on the line values.

Now, let us consider the probability that the line versus point test fails. This is at most the probability that $r \notin C$ plus the probability that $r \in C$ and it fails for r. So

$$\Pr[\mathrm{LvP}_d(\pi)] \leq \Pr[r \in C \wedge \mathrm{LvP}_d(\pi)] + \Pr[r \notin C].$$

The probability that a line versus point test fails for $r \in C$ is just the probability a random point in the point queries disagrees with p_l. But this is the same as the probability a random point in the point queries disagrees with σ_r. This is at most twice times the distance between σ_r and π:

$$\Pr_{r \in C}[\mathrm{LvP}_d(\pi)] \leq 2\mathrm{E}_{r \in C}[\Delta(\sigma_r, \pi_{I(x,r)})].$$

Thus

$$\begin{aligned} & \Pr[r \in C \wedge \mathrm{LvP}_d(\pi)] \\ = & \Pr[r \in C] \Pr_{r \in C}[\mathrm{LvP}_d(\pi)] \\ \leq & \Pr[r \in C] 2\mathrm{E}_{r \in C}[\Delta(\sigma_r, \pi_{I(x,r)})] \\ \leq & 2\mathrm{E}_r[\Delta(Y_r, \pi_{I(x,r)})]. \end{aligned}$$

So we can write

$$\begin{aligned}
\Pr[\mathrm{LvP}_d(\pi)] \leq& \Pr[r \in C \wedge \mathrm{LvP}_d(\pi)] + \Pr[r \notin C] \\
\leq& 2\mathrm{E}[\Delta(Y_r, \pi_{I(x,r)})] + \Pr[r \notin C] \\
\leq& 0.02 + 0.1 \\
\leq& 0.12.
\end{aligned}$$

Thus the line versus point test fails with probability at most 0.12. Then by Lemma 4.3.2, π is within 0.76 of a degree d polynomial, π'. That is

$$\Delta(\pi, \pi') < 0.24.$$

(c) Let D be the set of randomness so that on the curve queries, π has distance at most 0.6 from π'. We want to show that $\Pr[r \in D] \geq 0.4$.

First, we show that any individual point in the curve query past the first d' queries is uniformly random. That is, for $i \in [d'+1, |\mathbb{F}|]$, function $g(i)$ is uniform as a function of r, or more particularly, y.

This is because each different value of y encodes a specific value of $g(i)$. But alternatively, we could make $g(i)$ be any uniform value and decide our degree $q(n)$ polynomial that way, which would imply a value for y. So there is a bijection between choices for y and choices for $g(i)$, and choices of y are uniform, so choices for $g(i)$ must be too.

Let $I_c = I'_{[d'+1,|\mathbb{F}|]}(x, r)$ be the set of curve queries for randomness r, except the first d', which are not uniformly distributed. By linearity of expectation, the expected distance between π and π' on I_c is the expected distance between π and π' overall, which is at most 0.24. So then

$$\mathrm{E}[\Delta(\pi_{I_c}, \pi'_{I_c})] = \Delta(\pi, \pi') < 0.24.$$

Now the distance of π on the curve queries from π' is at most the distance on I_c plus the fraction of curve queries not in I_c. So by a Markov inequality,

$$\begin{aligned}
\Pr[r \notin D] \leq& \Pr[\Delta(\pi_{I_c}, \pi'_{I_c}) + \frac{d'}{|\mathbb{F}|} \geq 0.6] \\
\leq& \frac{\mathrm{E}[\Delta(\pi_{I_c}, \pi'_{I_c})] + 0.1}{0.6} \\
<& \frac{0.34}{0.6} \\
<& 0.6
\end{aligned}$$

(d) Now suppose $r \in C \cap D$. We want to show that our V accepts π' on this choice of randomness.

Since $r \in C$, for some proof σ_r accepted by the verifier, the distance between σ_r and π on the curve queries is at most 0.2. Since $r \in D$, the distance between π and π' on the curve queries is at most 0.6. So σ_r and π' agree on at least 0.2 fraction of curve queries.

Since σ_r is accepted by the verifier, it has degree d' on the curve queries. Since $\pi' \circ g$ is a composition of a degree d and q polynomial, it has degree $d' = dq$. Then both σ_r and $\pi' \circ g$ on the curve queries are degree d' and agree on more than d' locations. Thus σ_r and π' are equal on the curve queries.

Since σ_r is accepted by V, so is π' since they agree on the curve queries. Thus for $r \in C \cap D$, we have $\pi'_{I(x,r)}$ is accepted by the original **ePCP** verifier, V.

(e) By a union bound, with probability at least 0.3, we have $r \in C \cap D$. Thus with probability at least 0.3, our original **ePCP** protocol, A, accepts proof π'. But this can't be if $x \notin L$ since A has degree d soundness 0.1. So $x \in L$.

Thus B has soundness 0.99.

□

5 Constructing our ePCP

We use a BFL style base **PCP**. Our **ePCP** verifier asks for a multilinear extension of the computation history of an algorithm, and constructs a simple formula that indicates an inconsistency in the computation history. Then it does a sum check to verify that the arithmetization of this formula constructed from the computation history is 0 on all Boolean inputs.

Then the base **ePCP** consists broadly of 4 parts.

1. Check consistency of input with the claimed multilinear extension of the computation history. The multilinear extension of the input can be calculated in a straightforward matter by the verifier. Then we just need to compare our proof at the time 0 configuration to what it should actually be at a random point

2. Check consistency of the claimed multilinear extension of the computation history with the claimed value of an arithmetization of the inconsistency formula. This can be done by using the claimed multilinear extension of the computation history to calculate a random point in the arithmetization of the inconsistency formula and checking if they are equal.

3. Run a sum check on the claimed arithmetization of the inconsistency formula to verify it is constant 0 on Boolean inputs. Specifically, we check if the multilinear function consistent with the inconsistency formula is the constant 0 at a random point using a sum check.

5.1 Arithmetization

This paper frequently uses arithmetizations of boolean functions. We say that a function $f : \mathbb{F}^n \to \mathbb{F}$ is consistent with a boolean function $g : \{0,1\}^n \to \{0,1\}$ if f agrees with g when restricted to boolean inputs. If further f is a low degree polynomial, f is often called an arithmetization of g.

An example of an arithmetization is the multilinear extension of a boolean function. That is just the unique multilinear function, f, that agrees with g on boolean inputs. These can often be constructed very efficiently. For instance, the multilinear extension of the equality function.

Definition 5.1.1 (Equality Arithmetization)**.** *For field $\mathbb{F}$, and $l \geq 1$, define $equ : \mathbb{F}^l \times \mathbb{F}^l \to \mathbb{F}$ as:*

$$equ(u, v) = \prod_{i \in [l]} u_i \cdot v_i + (1 - u_i) \cdot (1 - v_i).$$

Observe that equ is the multilinear extension of the Boolean equality function.

But even for Boolean functions whose multilinear extensions can't be computed time efficiently, there is a space efficient, brute force way to compute it.

Lemma 5.1.2 (Multilinear Extensions Require Low Space)**.** *Suppose function $G : \{0,1\}^n \to \{0,1\}$ is computable in space S. Then the multilinear function g consistent with G on Boolean inputs is computable in space $O(n \log(|\mathbb{F}|) + S)$.*

Proof. This follows from the fact g can be written as

$$g(x) = \sum_{y \in \{0,1\}^n} G(y) \mathrm{equ}(y, x).$$

Then this can be evaluated using only a pointer for y, a small amount of space for equ, $O(n)$ field elements, and the space to evaluate G. $\square$

5.2 Simulating With Automata

First, we need to translate from the RAM model of computation our algorithms use to cellular automata. This is because we will be looking at an arithmetization of a uniform, local, consistency check. It is important for us to keep the degree of the arithmetization low, which requires very local checks.

Equivalently, one can think of this as just using the Cook-Levin reduction, but we use the cellular automata point of view because it makes the local nature of the computation clearer and more direct. If prover efficiency is a concern, one can use more efficient cellular automata, as was done by [HR18].

A cell's value in the next step of computation is only a function of it and its neighboring cells. So a cellular automata is very local. Further, cellular automata can simulate any RAM algorithm with only polynomial overhead in time, and very little overhead in space.

In this lemma, think of $S = O(n)$ and $T = 2^{O(n)}$. We will assume S and T are efficiently computable. Then we have the following, direct conversion from a RAM algorithm and an associated cellular automata.

Lemma 5.2.1 (RAM algorithms have simple cellular automata)**.** *Let A be a RAM algorithm recognizing L, running in time T and space S where $S = \Omega(\log(n))$ and $T = \Omega(S)$. Further, A uses input coming from a read only space of n bits.*

Then there is a 1 dimensional cellular automata, B, simulating A, such that

1. *B runs in time $T' = \mathbf{poly}(T, n)$, and space $S' = O(n + S)$.*
2. *B has a constant size alphabet, Σ, where for some k, we have $|\Sigma| = 2^{2^k}$. That is, Σ is represented by a power of 2 number of bits.*
3. *For any input x for A, there is a corresponding input for B, y_x, of length S'. And we also have that $y_x = (y^1, y^2_x, y^3)$ where*
   (a) *y^1 has length $O(\log(S'))$ and is independent of the specific x, only the length of x, and y^1 is computable in time $O(|y^1|)$.*
   (b) *y^2_x is exactly n symbols where for some $f : \{0, 1\} \to \Sigma$, for each $i \in [n]$, $(y^2_x)_i = f(x_i)$, where f is computable in constant time.*
   (c) *y^3 is exactly S copies of a specific symbol in Σ.*
4. *Not all transitions for B will be defined, and A accepts on x if and only if after time T' starting on y_x, B reaches a steady state. Similarly, A rejects on x if and only if there is no sequence of T' valid transitions in B starting from y_x.*
5. *If B has a starting state that is (y^1, z) for any z that is not (y^2_x, y^3) for some $x \in L$, then B will not have T' valid transitions.*
6. *Let $x \in L$ be an input for A, with transformed input for B, y_x. Given a time $t \in [T']$ and a memory location $s \in [S']$, there is a RAM algorithm C that can compute the symbol in cell s at time t in B's computation history on y_x in time $O(T)$ and space $O(S)$ given read only access to x.*

Remark. *The exact structure of the transformed input may seem overly specific, but we need this extra structure to construct our decodable* **PCP**. *In particular, our decodable* **PCP** *will need to know which cells encode a known, explicit, first input, and which cells encode an unknown, implicit, second input.*

We explain the translation more thoroughly in Appendix B.

For the purpose of analyses, it will be useful to look at a multilinear extension associated with a low degree polynomial. So we define the following purely to simplify the analysis of our **PCP**.

Definition 5.2.2 (Multilinear Extension of Binarized function (MLB))**.** *For any function $f : \mathbb{F}^n \to \mathbb{F}$, there is a unique, multilinear function, $g : \mathbb{F}^n \to \mathbb{F}$, such that for all binary $x \in \{0, 1\}^n$,*

$$g(x) = \begin{cases} 0 & f(x) = 0 \\ 1 & f(x) \neq 0 \end{cases}.$$

Then we say $MLB(f) = g$.

We can construct our inconsistency check from a claimed multilinear extension of a computation history. This comes from arithmetizing the transition rules of a cellular automata. The construction is straightforward, but details can be found in Appendix B.

Lemma 5.2.3 (Inconsistency Function). *Let k be a constant, s, t be integers, and $\mathbb{F}$ be a field. Let B be a cellular automata with 2^{2^k} different states per cell running in $S = 2^s$ cells, and time $T = 2^t$. Then there is a function Γ_B taking any function $X : \mathbb{F}^s \times \mathbb{F}^k \times \mathbb{F}^t \to \mathbb{F}$ and returning a function $Y : \mathbb{F}^{3s+2t+4(2^k)} \to \mathbb{F}$ such that:*

1. *If X is Boolean on Boolean inputs, Y is Boolean on Boolean inputs.*
2. *If X is Boolean on Boolean inputs, then Y is 0 on all Boolean inputs if and only if X on Boolean inputs encodes a valid computation history for B.*
3. *If X is degree d, then Y is degree $O(s + t + d)$. If X is degree d in every variable individually, Y is degree $O(d)$ in every variable individually.*
4. *Given oracle access to X, Y can be computed in time $O((t + s)\textbf{polylog}(|\mathbb{F}|))$ with a constant number of calls to X.*
5. *If $Y = \Gamma_B(X)$ is 0 on all Boolean inputs, then $\Gamma_B(MLB(X))$ is also 0 on all Boolean inputs.*

5.3 Sum Check Protocols

Our **PCP** uses the sum check protocol. The sum check is a standard element of **PCP**s, and all we add is a small bit of analysis that sum check is extrapolatable.

Lemma 5.3.1 (Sum Check Protocol). *Let $n, d \in \mathbb{N}$, and $\mathbb{F}$ be a field with $|\mathbb{F}| > (d + 1)n$. Then there is some protocol, A, so that for any $f : \mathbb{F}^n \times \mathbb{F} \to \mathbb{F}$:*

1. *For some $m = O(nd)$ and $R = 2n$, there is a verifier $V : \mathbb{F}^m \to \{0, 1\}$ and query function $Q : \mathbb{F}^R \times [m] \to \mathbb{F}^n \times \mathbb{F}$ so that*
$$A(f, r) = V(f(Q(r, 1)), f(Q(r, 2)), \ldots, f(Q(r, m))).$$
2. *V runs in time $O(nd\textbf{polylog}(|\mathbb{F}|))$ and space $O(nd \log(|\mathbb{F}|))$.*
3. *For any $r \in \mathbb{F}^R$, for $Q_r(i) = Q(r, i)$, Q_r is time $O(nd\textbf{polylog}(|\mathbb{F}|))$ extrapolatable.*
4. *For any $r \in \mathbb{F}^R$, the last coordinate of Q is always an element of $[n + 1]$. That is, for all $i \in [m]$, $Q(r, i)_{n+1} \in [n + 1]$*

   Further, the last coordinate of Q is only equal to $n + 1$ at most $O(d)$ times.

Completeness *For any $g : \mathbb{F}^n \to \mathbb{F}$ where g has max degree d in any individual variable, if for all $x \in \{0, 1\}^n, g(x) = 0$, then there is some $f : \mathbb{F}^n \times \mathbb{F} \to \mathbb{F}$ so that:*

- *For all $x \in \mathbb{F}^n$ we have $g(x) = f(x, n + 1)$.*
- *Sum check succeeds on f:*
$$\Pr_r[A(f, r) = 1] = 1.$$
- *Function f has degree at most d in each of its first n variables.*
- *If function g is computable in space S, then function f is computable in space $O(n \log(|\mathbb{F}|) + S)$.*

Soundness *for any $g : \mathbb{F}^n \to \mathbb{F}$ where g has max degree d', if there exists $x \in \{0, 1\}^n$ such that $g(x) \neq 0$, then for any $f : \mathbb{F}^n \times \mathbb{F} \to \mathbb{F}$ so that for all $x \in \mathbb{F}^n, g(x) = f(x, n + 1)$, sum check fails with high probability:*
$$\Pr_r[A(f, r) = 1] \leq \frac{(d' + 1)n}{|\mathbb{F}|}.$$

We will prove just the extrapolatable property in the body of this paper. For completeness, we prove the rest of the properties in Appendix C. To prove extrapolatability, we first formally define the sum check algorithm.

Definition 5.3.2 (Sum Check Protocol Definition)**.** *Let $n, d \in \mathbb{N}$, and $\mathbb{F}$ be a field with $|\mathbb{F}| > \max\{d, n\} + 1$. Suppose $f : \mathbb{F}^n \times \mathbb{F} \to \mathbb{F}$. Then the degree d Sum Check Protocol on f is the following randomized algorithm.*

1. *Get $2n$ random field elements, $R = (r_1, \dots, r_n$ and $r'_1, \dots, r'_n)$.*
2. *Reject if $f((r_1, \dots, r_n), 1) \neq 0$.*
3. *For i from 1 to n:*
   (a) *For $j \in [d+1]$, query*
   $$a_i^j = f((r'_1, \dots, r'_{i-1}, j, r_{i+1}, \dots r_n), i+1).$$
   Using these, let $g_i : \mathbb{F} \to \mathbb{F}$ be the degree d polynomial so that for all $j \in [d+1]$, $g_i(j) = a_i^j$.

   (b) *If*
   $$f((r'_1, \dots, r'_{i-1}, r_i, \dots r_n), i) \neq (1 - r_i)g_i(0) + r_i g_i(1)$$
   reject.

   (c) *If*
   $$f((r'_1, \dots, r'_i, r_{i+1}, \dots r_n), i+1) \neq g_i(r'_i)$$
   reject.
4. *If all checks pass, accept.*

This should look familiar to anyone familiar with the sum check protocol. At a high level, this protocol expects a sequence of polynomials where $f_i(x) = f(x, i)$ such that f_i is just f_{i+1} where the ith variable has been made degree 1 and f_{n+1} is degree at most d in each variable. Then sum check iteratively checks if each polynomial is consistent with their definition.

The correctness of this protocol is standard. Here, we only show that this protocol is extrapolatable.

Lemma 5.3.3 (Sum Check Queries Are Extrapolatable)**.** *For $n, d \in \mathbb{N}$, field $\mathbb{F}$ with $|\mathbb{F}| > \max\{d, n\} + 1$, and $r = (r_1, \dots, r_n, r'_1, \dots r'_n) \in \mathbb{F}^{2n}$, the degree d sum check query locations, Q_r, used in Definition 5.3.2, are $O(nd\,\mathbf{polylog}(|\mathbb{F}|))$ extrapolatable.*

Proof. Define the degree d sum check query location function, $Q_r : [(d+3)n] \to \mathbb{F}^n \times \mathbb{F}$, as, for any $l \in [d+3]$ and $i \in [n]$:

$$Q_r((d+3)(i-1)+l) = \begin{cases} \big((r'_1, \dots, r'_{i-1}, r_i, r_{i+1}, \dots r_n), i\big) & l = 1 \\ \big((r'_1, \dots, r'_{i-1}, 1, r_{i+1}, \dots r_n), i+1\big) & l = 2 \\ \vdots & \\ \big((r'_1, \dots, r'_{i-1}, d+1, r_{i+1}, \dots r_n), i+1\big) & l = d+2 \\ \big((r'_1, \dots, r'_{i-1}, r'_i, r_{i+1}, \dots r_n), i+1\big) & l = d+3 \end{cases}.$$

These are the queries made to f by the sum check protocol for randomness $r = (r_1, \dots, r_n, r'_1, \dots, r'_n)$.

To show Q_r is extrapolatable, take $v_1, \dots, v_{(d+3)n}$. We need a time $O(nd\,\mathbf{polylog}(|\mathbb{F}|))$ algorithm computing
$$u = \sum_{i \in [(d+3)n]} v_i Q_r(i).$$
Note u is an $n+1$ component vector.

For any give $j \in [n]$,
$$u_j = \left(\sum_{i=1}^{(d+3)(j-1)+1} v_i r_j\right) + \left(\sum_{i=1}^{d+1} i v_{((d+3)(j-1)+1+i)}\right) + \left(\sum_{i=(d+3)j}^{(d+3)n} v_i r'_j\right).$$
We will handle u_{n+1} at the end.

First, look at the first and last terms. For $j \in [n]$, let

$$\alpha_j = \sum_{i=1}^{(d+3)(j-1)+1} v_i$$

$$\beta_j = \sum_{i=(d+3)j}^{(d+3)n} v_i.$$

Then an iterative algorithm can calculate every α_j and β_j in $O(nd\mathbf{polylog}(|\mathbb{F}|))$ time. Given α_j and β_j, u_j can be calculated in $O(d\mathbf{polylog}(|\mathbb{F}|))$ time. So all u_j for $j \in [n]$ can be calculated in $O(nd\mathbf{polylog}(|\mathbb{F}|))$ time.

Now u_{n+1}, as a single component, can just straightforwardly be evaluated in time $O(nd\mathbf{polylog}(|\mathbb{F}|))$ from the definition. Thus u can be calculated in $O(nd\mathbf{polylog}(|\mathbb{F}|))$ time. □

For completeness, we prove the rest of Lemma 5.3.1 in Appendix C.

5.4 Our Base PCP

The idea of the **PCP** is to ask the prover for the function that takes a time t and a bit of memory s and returns the value of cell s at time t in the cellular automata. Of course, we need this to be error corrected, so we ask for the multilinear extension of this function. We check if this is consistent with the input. Then, given this function, we can compute an arithmetization of whether cell s at time t has an improper transition. Finally, we run a sum check on this arithmetization to see if it is 0 on all Boolean inputs.

This **PCP** is actually an **ePCP**, which can be converted into an **rPCP**. Further, this **PCP** explicitly encodes its initial input, giving us a natural way to extend it into a **dPCP**. We will give the extension to a **dPCP** in the next section.

Lemma 5.4.1 (Base **ePCP**)**.** *Let $S, T = \Omega(n)$, and L be any language computed by a simultaneous time T and space S algorithm.*

There is some constant $\alpha > 0$, so that for any field $\mathbb{F}$ with $|\mathbb{F}| > \alpha \log(T)^2$, we have an **ePCP** *with*

1. *Verifier time $O((\log(T) + n)\mathbf{polylog}(|\mathbb{F}|))$ and space $O(\log(T)\log(|\mathbb{F}|))$.*
2. *Randomness $O(\log(T)\log(|\mathbb{F}|))$.*
3. *Prover space $O(\log(T)\log(|\mathbb{F}|) + S)$.*
4. *$O(\log(T))$ queries.*
5. *Alphabet $\mathbb{F}$.*
6. *Extrapolation time $O(\log(T)\mathbf{polylog}(|\mathbb{F}|))$.*
7. *Degree $O(\log(T))$ and $O(\log(T))$ variables.*
8. *Perfect completeness.*
9. *Low degree soundness $O\left(\frac{\log(T)^2}{|\mathbb{F}|}\right)$.*
10. *Log of proof length $O(\log(T)\log(|\mathbb{F}|))$.*

Proof. Take such a language L computed by RAM algorithm A running in time T and space S. By Lemma 5.2.1, A has a simulation by a cellular automata B with time T' polynomial in T and space S' linear in S.

Let $K = 2^k$ be a constant power of two such that the states in B are encoded in $\{0,1\}^K$. Let s and t be constants so that $S' \leq 2^s$ and $T' \leq 2^t$. There is also a RAM algorithm C that can compute the bits of the computation history of B in time $O(T)$ and space $O(S)$.

Let x be our input, and y be that input encoded for B. Now we will describe an honest prover for $x \in L$.

If $x \in L$, then let $X' : \{0,1\}^s \times \{0,1\}^k \times \{0,1\}^t \to \{0,1\}$ be the function that outputs the computation history of B with the starting input being y. Then by Lemma 5.1.2 the multilinear extension of X', X, can be computed in space $O(\log(T)\log(|\mathbb{F}|) + S)$.

Then by Lemma 5.2.3, using X, we can compute $Y = \Gamma_B(X)$ that is constant degree in each variable, uses constantly many queries to X, and Y is 0 on all binary inputs (as well as the other properties listed in Lemma 5.2.3, which we later use to prove soundness). Let $m = 3s + 2t + 4K$. Then Y is a function $\mathbb{F}^m \to \mathbb{F}$. Abusing notation slightly, let $X : \mathbb{F}^m \to \mathbb{F}$ be X applied to the first $s + k + t$ variables.

Then by the completeness case of Lemma 5.3.1, in space $O(\log(T)\log(|\mathbb{F}|) + S)$, the prover can compute the proof, f, for the sum check for Y. Let $f_i(x) = f(x, i)$.

For all $j \in [m+2]$, let $l_j^{m+2} : \mathbb{F} \to \mathbb{F}$ be the unique $m+1$ degree polynomial that is 1 at j, and 0 for all other $i \in [m+2]$. Then the proof for our **PCP** is supposed to be $\pi : \mathbb{F}^m \times \mathbb{F} \to \mathbb{F}$ where

$$\pi(z,i) = \left(\sum_{j \in [m+1]} l_j^{m+2}(i) f_j(z) \right) + l_{m+2}^{m+2}(i) X(z).$$

See that restricting $i \in [m+1]$ gives f_i, and for $i = m+2$ gives X.

Then we already showed how to compute X and f in space $O(\log(T)\log(|\mathbb{F}|) + S)$. Then we only need additional space to store a pointer to j (which requires only $O(\log(T))$ space), and to compute the interpolating polynomial l_j^m. Recall that

$$l_j^{m+2}(i) = \prod_{h \in [m+2] \setminus \{j\}} \frac{i-h}{j-h}.$$

Which can be straightforwardly computed with a constant number of field elements. So any symbol in π is computable in space $O(\log(T)\log(|\mathbb{F}|) + S)$.

Finally, π has constant degree in each of the first m variables, since X and each of the f_i do. And π has degree $O(m)$ in the last variable since each l_j^{m+2} has degree $O(m)$. This gives π a final degree of $d = O(m)$.

Now we describe the verifier. For a provided proof, π, we will infer the provided X, and f in the obvious way. The verifier runs a few checks for input x.

1. Sum check of f.

   Follow the verifier in the sum check protocol in Lemma 5.3.1. Just make sure the $\mathbb{F}$ is large enough so the soundness is less than $\delta/4$, which is true for large enough α.

2. Consistency of X with f_{m+1}.

   Let W be the set of w so that the sum check queries $f(w, m'+1)$. Sum check will only query constantly many of such w, since the degree of Y is constant. So W has constant size.

   Then for $w \in W$, use Lemma 5.2.3 to calculate $Y(w) = \Gamma_B(X)(w)$. Then check if $f(w, m'+1) = Y(w)$.

3. Consistency of X with y.

   For input x, it has transformed input $y = (y^1, y_x^2, y^3)$ as described in Lemma 5.2.1. Let $n' = O(n + \log(S))$ be so that $n' \geq (|y^1| + |y_x^2|)K$, and n' is a power of 2: $n' = 2^N$.

   Choose a random element, $v \in \mathbb{F}^N$, and compute

   $$u(y,v) = \sum_{z \in \{0,1\}^N} \mathrm{equ}(z,v) y_z.$$

   where we interpret z as a binary number and y_z is the zth bit of y, and equ is the multilinear extension of the equality function from Definition 5.1.1.

   This is basically the multilinear extension of our first input. This should be equal to some entry in X at time 0, with 0 for most space coordinates, besides the first N being v. Specifically, for $v' = (0^{s-N+k}, v, 0^t)$, we should have $X(v') = u(y,v)$.

   Reject if $X(v') \neq u(y,v)$.

Now let us show this has the desired properties.

1. Verifier time $O((\log(T) + n)\mathbf{polylog}(|\mathbb{F}|))$ and space $O(\log(T)\log(|\mathbb{F}|))$.

   The sum check protocol runs in time $O(\log(T)\mathbf{polylog}(|\mathbb{F}|))$ and uses space $O(\log(T)\log(|\mathbb{F}|))$.

   Checking consistency of X with f_{m+1} is only constantly many calculations of Y, which only takes time $O(m\mathbf{polylog}(|\mathbb{F}|)) = O(\log(T)\mathbf{polylog}(|\mathbb{F}|))$.

   When checking the consistency of X with input, we need to calculate $u(y, v)$. This can be done efficiently with a stack of partial calculations of equ and enumerating through z in standard order. In expectation, each value of z only requires a constant number of field operations, and there are only $O(n + \log(S))$ values for z. So it only takes time $O((n + \log(T))\mathbf{polylog}(|\mathbb{F}|))$. Further, the stack of partial calculations only needs to hold $N = O(\log(n + \log(S))) = O(\log(T))$ field elements, which only takes $O(\log(T)\log(|\mathbb{F}|))$ space.

2. Randomness $O(\log(T)\log(|\mathbb{F}|))$.

   The verifier needs to use $O(\log(T)\log(|\mathbb{F}|))$ bits to run the sum check, and to choose v.

3. Prover space $O(\log(T)\log(|\mathbb{F}|) + S)$.

   We already went over the prover space when describing the prover.

4. $O(\log(T))$ queries.

   The sum check only takes $O(\log(T))$ queries, and there are only constantly many other queries.

5. Alphabet $\mathbb{F}$.

   From how we defined our **ePCP**.

6. Extrapolation time $O(\log(T)\mathbf{polylog}(|\mathbb{F}|))$.

   From Lemma 5.3.1, the sum check is time $O(\log(T)\mathbf{polylog}(|\mathbb{F}|))$ extrapolatable. There are only constantly many other queries, so all the other queries are trivially time $O(\log(T)\mathbf{polylog}(|\mathbb{F}|))$ extrapolatable.

   Since the query locations are just constantly many extrapolatable query locations, by Lemma 4.1.1, all together they are time $O(\log(T)\mathbf{polylog}(|\mathbb{F}|))$ extrapolatable.

7. Degree $O(\log(T))$ and $O(\log(T))$ variables.

   We already showed when describing an honest prover that we have degree $O(m)$ which is $O(\log(T))$ and by definition of the proof, it has $O(\log(T))$ variables.

8. Perfect completeness.

   Follows for $x \in L$ with an honest prover. Since $x \in L$, the prover provides the X that is the multilinear extension of the computation history from the proper y, so consistency with input passes. Similarly, f is honestly given to be consistent with Y. So the consistency between X, Y, and f passes. Finally, since X is a valid computation history, Y is 0 on all Boolean inputs, so the sum check succeeds.

9. Low degree soundness $\frac{O(\log(T)^2)}{|\mathbb{F}|}$.

   Suppose $x \notin L$ and we are given a degree $d = O(\log(T))$ proof, π. Then X is a degree d function.

   Now let $\hat{X} = \mathrm{MLB}(X)$, and X' be $\hat{X}$ restricted to binary inputs. Since $x \notin L$, either X' is an invalid computation history, or it does not start with state y.

   If X' does not start with state (y^1, y_x^2, z) for some z, $\hat{X}$ restricted to time 0 and 0 for everything but the first N spaces is not the multilinear extension of the state $y([n'])$. Thus neither is X. Then the probability that v is chosen so that X agrees with $u(y, v)$ is at most $\frac{d}{|\mathbb{F}|}$. So the **ePCP** accepts with probability only $\frac{d}{|\mathbb{F}|}$.

If X' does start with state (y^1, y^2_x, z), and $z \neq y^3$, then X' must be an invalid computation history, since y^1 is correct, by Lemma 5.2.1. Similarly, if $z = y^3$, the computation history must be invalid since $x \notin L$. So then all we have left is the case that X' is not a valid computation history.

Suppose X' is an invalid computation history. Then $\Gamma_B(\hat{X})$ must not be 0 on all Boolean inputs. Then from Lemma 5.2.3, by contrapositive, $\Gamma_B(X)$ must not be 0 on all binary inputs.

Then by the soundness in Lemma 5.3.1, the sum check for $\Gamma_B(X)$ passes with probability at most

$$\frac{(d+1)m}{|\mathbb{F}|} = O\left(\frac{\log(T)^2}{|\mathbb{F}|}\right).$$

So with probability at most $O\left(\frac{\log(T)^2}{|\mathbb{F}|}\right)$ do we accept.

10. Log of proof length $O(\log(T)\log(|\mathbb{F}|))$.

    This comes from the fact that the proof is a function with domain $\mathbb{F}^{m+1}$, and

$$\log(|\mathbb{F}^{m+1}|) = (m+1)\log(|\mathbb{F}|) = O(\log(T)\log(|\mathbb{F}|)).$$

□

6 Decodable PCP and Composition

Our **PCP** uses the standard technique of **PCP** composition [AS98; BS+04; DR04; MR08; DH09] to reduce the number of queries. Here we overview the basics of **PCP** composition using robust and decodable **PCP**s [MR08; DH09], construct our decodable **PCP**, and prove Theorem 1.1.3.

A decodable **PCP** (**dPCP**) is a **PCP** that not only verifies that a solution to a problem exists, but also with high probability decodes a symbol from a single[5] solution.

Together, an **rPCP** and a **dPCP** give a composition theorem. The robust "outer" **PCP** chooses queries to a large proof. For this set of queries, we ask a decodable "inner" **PCP** to prove the outer **PCP** on this set of queries would accept. Then we ask the inner, decodable **PCP** for a symbol that would have been queried by the robust, outer **PCP**. Since the outer **PCP** is robust (see Definition 4.2.2), if $x \notin L$, then for many of the choices of queries, the outer proof disagrees with any accepted queries at many places. Thus the outer proof must often disagree with the symbol decoded by the inner **PCP**, since the inner **PCP** decodes a symbol from a solution.

6.1 Decodable PCPs

A decodable **PCP** (**dPCP**) verifies pairs of inputs together: an explicit input, and an implicit input.

- The explicit input is known to the verifier and contains the input x the **PCP** is trying to verify.
- The implicit input is not known by the verifier, it is only known by the prover. You should think of this as a proof for x, just one too large to read. In our application, it will be polynomially larger than x, so our verifier wouldn't even have time to read it all.

Then the **dPCP**, in addition to verifying that this input as a pair are in a language, needs to decode a symbol from the implicit input with high probability.

Our definition of **dPCP** is very similar to our definition of **PCP** (see Definition 2.0.8), except that

1. The implicit input has some specific, potentially non binary alphabet, Σ', in addition to the alphabet of the **dPCP** proof, Σ.
2. We renamed the verifier V to a decoder D. This is because when D accepts it now outputs a symbol from Σ', which it claims is a decoded symbol from the implicit input. If it rejects, it outputs $\perp$.

[5]Often, a **dPCP** will use list decodability, so that the **dPCP** can actually decode a symbol from a small list of solutions. We only discuss unique decoding.

3. We rename the prover P to encoder E. This is because now the encoder not only has an explicit input it must prove, but an implicit input that D needs to decode.

   In our application, our encoder E also cannot use enough space to hold the entire implicit input. Instead, it will have to recalculate each symbol of the implicit input every time it needs one.

Now we define a decodable **PCP**.

Definition 6.1.1 (Decodable **PCP**). *Let L' be a language containing pairs, where if the first input is length n, the second input is $m(n)$ symbols from alphabet Σ'. We say L' has a decodable* **PCP** *(***dPCP***), B, if for some decoder D, encoder E, index function I, and query function Q, alphabet Σ, constant $\delta \geq 0$, and functions $q, r, l : \mathbb{N} \to \mathbb{N}$:*

1. *I' takes 3 inputs, an input of length n, randomness $r(n)$, and an index in $[m(n)]$ and outputs an element of $[l(n)]^{q(n)}$. That is, I outputs $q(n)$ indexes in a length $l(n)$ string,*
2. *Q is an algorithm with 4 inputs, an input x of length n, randomness r of length $r(n)$, an index $j \in [m(n)]$, and an index $i \in [q(n)]$, and outputs an element of $[l(n)]$ such that $Q(x, r, j, i) = I(x, r, j)_i$.*
3. *D is an algorithm that takes 4 inputs: an input of length n, randomness of length $r(n)$, $q(n)$ symbols from Σ, and an index in $[m(n)]$. The algorithm D outputs either an element of Σ' or $\perp$.*
4. *E is an algorithm that takes three inputs, an input of length n, some $m(n)$ symbols from Σ', and an index $i \in [l(n)]$, and outputs a symbol from Σ.*

Completeness: *For any x of length n and for any $y \in \Sigma'^{m(n)}$ such that $(x, y) \in L'$, there exists a proof $\pi^{x,y}$ such that*

$$\Pr_{r,i}[D(x, r, \pi^{x,y}_{I(x,r,i)}, i) = y_i] = 1.$$

Further for every $i \in [l(|x|)]$, we have $E(x, y, i) = \pi^{x,y}_i$.

Soundness: *For any x and any π, if*

$$\Pr_{r,i}[D(x, r, \pi_{I(x,r,i)}, i) \neq \perp] > \delta,$$

then there is exists y such that $(x, y) \in L'$ and

$$\Pr_{r,i}[D(x, r, \pi_{I(x,r,i)}, i) \notin \{y_i, \perp\}] \leq \delta.$$

Then we also say:

1. *B has proof length $l'(n)$.*
2. *B has alphabet Σ.*
3. *B has soundness δ.*
4. *B uses $q(n)$ queries.*
5. *B uses $r(n)$ bits of randomness.*
6. *If D runs in time $t(n)$, B has decoder time $t(n)$.*
7. *If E runs in space $s'(n)$, B has encoder space $s'(n)$.*
8. *If Q is computable in time $t'(n)$, B has query time $t'(n)$.*

Composing a robust and a decodable **PCP** gives a **PCP** with the number of queries of the decodable **PCP**, plus one to compare the outer **PCP** proof to the symbol decoded by the inner **PCP**. In our application, this allows us to reduce a **PCP** that uses $O(n)$ queries to one that uses $O\log(n))$. The proof is straightforward and included for completeness in Appendix A.

Theorem 6.1.2 (**PCP** Composition)**.** *Suppose L is a language with an* **rPCP***, A, with verifier V, prover P, query function Q, and index function I such that*

1. *Q runs in time $t(n)$.*
2. *P run in space $s(n)$.*
3. *V uses $r(n)$ bits of randomness.*
4. *A uses alphabet Σ.*
5. *A has robust soundness δ.*
6. *A has perfect completeness.*
7. *A has proof length $l(n)$.*

Suppose $L' = \{((x,r),y) : V(x,r,y) = 1\}$. Let $n' = n + r(n)$. Suppose L' has a **dPCP** *protocol, B, with decoder D and encoder E such that*

1. *E runs in space $s'(n')$.*
2. *D runs in time $t'(n')$.*
3. *B uses $q'(n')$ queries.*
4. *B has query time $t^*(n')$.*
5. *B uses alphabet Σ'.*
6. *B has soundness δ'.*
7. *B has perfect completeness.*
8. *B has proof length $l'(n')$.*

Then there is a **PCP** *protocol for L, C, such that*

1. *C has verifier time $O(t'(n'))$*
2. *C uses $O(q'(n'))$ queries.*
3. *C has prover space $O(s(n) + t(n) + s'(n'))$.*
4. *C uses alphabet $\Sigma' \cup \Sigma$.*
5. *C has query time $O(t(n) + t^*(n'))$.*
6. *C has soundness $\delta + \delta'$.*
7. *C has perfect completeness.*
8. *C has proof length $l(n) + 2^{r(n)}l'(n')$.*

6.2 More Low Degree Gadgets

Since our **dPCP** doesn't use the reduction from **ePCP** to **rPCP**, it needs to do low degree testing itself. Here, we mostly use local decoding properties of low degree polynomials.

We show there is an implicit way to run the line versus point test (see Definition 4.3.1) for a function $f : \mathbb{F}^n \to \mathbb{F}$ using queries to just f. We do this by inferring g_l from $f(l(1)), \ldots f(l(d+1))$. This may not be the optimal g_l, but it still gives an upper bound for the probability of $\mathrm{LvP}_d(f)$.

Lemma 6.2.1 (Implicit Line Versus Point Test)**.** *For a field $\mathbb{F}$ and degree d, the implicit line versus point test is defined by the following. An index function I, a query function Q, and a verifier V such that for any function $f : \mathbb{F}^n \to \mathbb{F}$, points $x, y \in \mathbb{F}^m$, and $t \in \mathbb{F}$:*

1. *I gives locations along a line*
$$I(x, y, t) = (x + 1 \cdot y, \ldots, x + (d + 1) \cdot y, x + t \cdot y).$$
2. *Q outputs elements from I:*
$$Q(x, y, t, i) = I(x, y, t)_i.$$
3. *V, given t, and $v \in \mathbb{F}^{d+2}$ as input, finds the degree d polynomial, $g : \mathbb{F} \to \mathbb{F}$, such that for $i \in [d+1]$, we have $g(i) = v_i$. Then V accepts if $g(t) = v_{d+2}$.*
4. *Then the implicit line versus point test is whether*
$$V(t, f(I(x, y, t))) = 1.$$

The implicit line versus point test has the following properties:

1. *V runs in time $O(d\,\mathbf{polylog}(|\mathbb{F}|))$ and Q runs in time $O(\mathbf{polylog}(|\mathbb{F}|))$.*
2. *If x, y, and t are chosen uniformly at random, then*
$$\Pr_{x,y,t}[V(t, f(I(x, y, t))) = 0] \geq \Pr[LvP_d(f)].$$
3. *If f has degree d, then*
$$\Pr_{x,y,t}[V(t, f(I(x, y, t))) = 1] = 1.$$

Proof. Now we prove the properties.

1. To get the verifier running time, we can use Lagrange interpolation. See Lemma 4.1.4 to see how to do Lagrange Interpolation. Function Q is just a multiplication and an addition.
2. To get the probability of acceptance, first note for each $i \in [d+1]$,
$$Q(x, y, t, i) = x + i \cdot y,$$
and
$$Q(y, x, t, d + 2) = x + t \cdot y.$$
If x, y is chosen uniformly random, then g is just a degree d polynomial for the line $l(s) = x + s \cdot y$. Then $f(x + t \cdot y) = f(Q(x, y, t, d + 2))$, and
$$\Pr[g(t) = f(x + t \cdot y)]$$
is at most the probability the probability the optimal g does, which is the g the line versus point test uses. Thus for x, y, and t uniformly at random
$$\Pr_{x,y,t}[V(t, f(I(x, y, t))) = 0] \geq \Pr[\mathrm{LvP}_d(f)].$$

3. Finally, if f has degree d, then composing the line $l(s) = x + s \cdot y$ with f has degree d. So g agrees with that polynomial at d points, so is equal to it. Then g at t is equal to it too.

$\square$

We will use the implicit line versus point test several times to get high confidence that f is actually very close to low degree.

Lemma 6.2.2 (Low degree test)**.** *There is some constant c, so that for any integers d and n, field $\mathbb{F}$ with $|\mathbb{F}| \geq cd$, and constant $\epsilon \in (0, \frac{1}{5})$, there is some protocol, A, so that:*

1. *For some $m = O(d)$ and $r = O(n)$, there is a verifier $V : \mathbb{F}^m \to \{0,1\}$ and query function $Q : \mathbb{F}^r \times [m] \to \mathbb{F}^n$ so that for any $f : \mathbb{F}^n \to \mathbb{F}$,*
$$A(f, r) = V(f(Q(r, 1)), f(Q(r, 2)), \ldots, f(Q(r, m))).$$
2. *V runs in time $O(d\mathbf{polylog}(|\mathbb{F}|))$ and Q runs in time $O(\mathbf{polylog}(|\mathbb{F}|))$.*

Completeness *If $f : \mathbb{F}^n \to \mathbb{F}$ is a degree d polynomial, then*
$$\Pr_r[A(f, r) = 1] = 1.$$

Soundness *If for $f : \mathbb{F}^n \to \mathbb{F}$,*
$$\Pr_r[A(f, r) = 1] \geq \epsilon,$$
then there exists some polynomial, h, of degree at most d so that
$$\Delta(f, h) \leq \epsilon.$$

Proof. Let $B(f)$ be the random variable of the output of the implicit line versus point test on a random input. Let $k = \frac{2\ln(1/\epsilon)}{\epsilon}$. Let A be the protocol that runs the implicit line versus point test (Lemma 6.2.1) k independent times with uniformly random x, y, t and outputs if they all pass.

Then V runs in k times the time of the implicit line versus point test. Since k is constant, this is time $O(d\mathbf{polylog}(|\mathbb{F}|))$. Q is just a query from the implicit line versus point test, which only takes time $O(\mathbf{polylog}(|\mathbb{F}|))$.

If f is a degree d polynomial, then B always succeeds, and so does A.

If $\Pr[\mathrm{LvP}_d(f)] \leq \Pr[B(f) = 0] = \delta$, then the probability of A passing is
$$\begin{aligned}\Pr[A(f) = 1] =& \Pr[B(f) = 1]^k \\ =& (1-\delta)^k \\ =& (1-\delta)^{\frac{2\ln(1/\epsilon)}{\epsilon}} \\ \leq& e^{-\delta\frac{2\ln(1/\epsilon)}{\epsilon}}.\end{aligned}$$

If $\epsilon \leq \Pr[A(f) = 1]$, then
$$\begin{aligned}\epsilon \leq& e^{-\delta\frac{2\ln(1/\epsilon)}{\epsilon}} \\ \ln(\epsilon) \leq& -\delta\frac{2\ln(1/\epsilon)}{\epsilon} \\ \delta\frac{2\ln(1/\epsilon)}{\epsilon} \leq& \ln(1/\epsilon) \\ \delta \leq& \frac{\epsilon}{2} \\ \Pr[\mathrm{LvP}_d(f)] \leq& \frac{\epsilon}{2} \leq 1/10.\end{aligned}$$

Then by Lemma 4.3.2 there exists some d degree polynomial h so that
$$\Delta(f, h) \leq \epsilon.$$

$\square$

We also need to do some error corrected queries to the low degree function f is near. Essentially, we do a bunch of line versus point tests for lines going through our point, and only output the value of f at that point if each of these line versus point tests succeed and agree with the value at that point.

Lemma 6.2.3 (Self Correction Of Approximate Low Degree Polynomials)**.** *For any $\epsilon > 0$, integers d and n, field $\mathbb{F}$ with $|\mathbb{F}| > 4d$, there is a protocol A such that for any function $f : \mathbb{F}^n \to \mathbb{F}$, and for any $x \in \mathbb{F}^n$,:*

1. *For some $m = O(d)$ and $r(n) = O(n)$, there is a verifier $V : \mathbb{F}^m \to \mathbb{F} \cup \{\perp\}$ and query function $Q : \mathbb{F}^{r(n)} \times [m] \to \mathbb{F}^n$ so that*

$$A(f, r) = V(f(Q(r, 1)), f(Q(r, 2)), \ldots, f(Q(r, m))).$$

   Here, $\perp$ is some symbol outside of $\mathbb{F}$ indicating V rejects.

2. *V runs in time $O(d\mathbf{polylog}(|\mathbb{F}|))$ and Q runs in time $O(\mathbf{polylog}(|\mathbb{F}|))$.*

Completeness *If f is a degree d polynomial, for all $r \in \mathbb{F}^{r(n)}$, $A(f, r)$ always outputs $f(x)$.*

Soundness *If for some degree d polynomial h, $\Delta(f, h) < 1/2$, with probability at least $1 - \epsilon$ over $r \in \mathbb{F}^R$, $A(f, r)$ either rejects or outputs $h(x)$. We call ϵ the soundness.*

Proof. Define algorithm B to be the conjunction (AND) of two calls of the the implicit line versus point test where y is uniformly random, x comes from the input, and once $t = 0$, and once t is a uniformly random element of $\mathbb{F}$.

Explicitly, B first chooses a random $y \in \mathbb{F}^m$ and $t \in \mathbb{F}$. Let $g_{x,y} : \mathbb{F} \to \mathbb{F}$ be the degree d function so that for $i \in [d + 1]$, $g_{x,y}(i) = f(x + i \cdot y)$. Then B accepts if $g_{x,y}(0) = f(x)$ and $g_{x,y}(t) = f(x + t \cdot y)$.

Algorithm A runs B for $k = 4\ln(1/\epsilon)$ times, and if they all accept, it outputs $f(x)$. The query time for A is just the time for an implicit line versus point test query. So Q runs in time $O(\mathbf{polylog}(|\mathbb{F}|))$. And V just runs $O(k)$ instances of the implicit line versus test verification, which takes time $O(d\mathbf{polylog}(|\mathbb{F}|))$.

If f is degree d, then the implicit line versus point test always accepts, and we always output $f(x)$.

Now suppose for some degree d polynomial h, $\Delta(f, h) < 1/2$. Then B can only output the wrong value if $f(x) \neq h(x)$, so suppose $f(x) \neq h(x)$. Then B only accepts erroneously if $g_{x,y}(0) = f(x)$, and $g_{x,y}(t) = f(x + t \cdot y)$.

Define the degree d polynomial $h_{x,y}(t) = h(x + t \cdot y)$. Suppose for some y we have $g_{x,y}(0) = f(x)$. Then $g_{x,y} \neq h_{x,y}$, since $h_{x,y}(0) = h(x) \neq f(x)$. Since $g_{x,y}$ is a different degree d polynomial than $h_{x,y}$, they agree on at most d places. Then if t is both one of the places where f agrees with h, but h disagrees with g, we reject.

Observe, $x + t \cdot y$ is uniformly distributed over $\mathbb{F}^m$. Then let C be the event that $g_{x,y}(0) = f(x)$. Then by a union bound:

$$\begin{aligned}
\Pr[B = 1 \wedge h(x) \neq f(x)] \leq & \Pr_{y,t}[C \wedge g_{x,y}(t) = f(x + t \cdot y)] \\
\leq & \Pr_{y,t}[C \wedge (h_{x,y}(t) \neq f(x + t \cdot y) \vee h_{x,y}(t) = g_{x,y}(t))] \\
\leq & \Pr_{y,t}[h(x + t \cdot y) \neq f(x + t \cdot y)] \\
& + \Pr_{y,t}[C \wedge h(x + t \cdot y) = g_{x,y}(t)] \\
\leq & 1/2 + d/|\mathbb{F}| \\
\leq & 3/4.
\end{aligned}$$

Thus if B samples $g_{x,y}(i)$ at any of these at least $1/4$ of the locations that it disagrees with f, B rejects.

Each one of the k samples is independent and uniform. So the probability that A erroneously outputs $f(x)$ is at most $(3/4)^k \leq \epsilon$. Thus with probability $1 - \epsilon$, A outputs $h(x)$, or rejects. $\square$

6.3 Decoding With Our PCP

Now we show how to convert our base **ePCP** (from Lemma 5.4.1) into a **dPCP**. This requires the following changes:

1. We need to do a low degree test.
2. Queries to the multilinear extension of the input now need to be error corrected.
3. We can't verify the implicit input is at time 0 in the provided proof. We can only verify the explicit input is at time 0.
4. We need to perform an extra error corrected query into the implicit input to check a symbol from it. While we need to do $O(\log(|\Sigma|))$ queries to get a single symbol, we only need to do one error corrected query to check it. To check it, we use a similar technique that we use for the explicit input, except only on a single symbol.

Then we get the following **dPCP**:

Lemma 6.3.1 (Making our base **PCP** into a **dPCP**)**.** *Let $S, T = \Omega(n)$, and L be a language of pairs where if the first element has n binary symbols, the second element has $m(n) = \mathbf{poly}(n)$ symbols in some alphabet Σ where $\log(|\Sigma|) = O(\log(T))$. Suppose L is computed by a simultaneous time T and space S algorithm.*

Then for any constant $\delta > 0$, there is some constant $\alpha > 0$, so that for any field $\mathbb{F}$ with $|\mathbb{F}| > \alpha \log(T)^2$, we have a **dPCP** *with*

1. *Decoder time $O((\log(T) + n)\mathbf{polylog}(|\mathbb{F}|))$ and space $O(\log(T)\log(|\mathbb{F}|))$.*
2. *Randomness $O(\log(T)\log(|\mathbb{F}|))$.*
3. *Encoder space $O(\log(T)\log(|\mathbb{F}|) + S)$.*
4. *$O(\log(T))$ queries.*
5. *Alphabet $\mathbb{F}$.*
6. *Query time $O(\log(T)\mathbf{polylog}(|\mathbb{F}|))$.*
7. *Soundness δ.*
8. *Perfect completeness.*
9. *Log of proof length $O(\log(T)\log(|\mathbb{F}|))$.*

Proof. Let A be the time T space S algorithm for L. Assume $|\Sigma| = 2^K$ where $K = 2^k$ for some constant k.

The idea is to use the same **PCP** as Lemma 5.4.1, but adapted to take a pair of inputs. Recall that our base **PCP** checks for a valid computation history of a cellular automata that computes our function. Then we verify that the starting state of that cellular automata is consistent with our input. We do the same thing, except now we need to have the cellular automata also hold the second input, and since we can't know the second input, we can't check it directly.

Recall the cellular automata (from Lemma 5.2.1) just simulates an algorithm that runs in $O(S)$ space. Then we modify our algorithm to add padding on the explicit input. That is, we choose some $n' = 2^N > \max\{n + O(\log(S)), K\}$ space to store our first input and the registers. Then $n' = O(n + \log(T))$. Then the second input will be stored outside that reserve region, followed by the working space.

So for input $x = (x_1, x_2)$ of length $n + m(n)$, we will actually use an algorithm, A', that works on a padded version of this input: $x' = (x_1', x_2')$. Our x_1' is x_1 padded so that $|y^1| + |x_1'| = n'$. Note that while y^1 is dependent on the length of x_1', it is of logarithmic size in x_1' and efficiently computable, so can easily be factored into the padding. Then x_2' can just be x_2. Then we use Lemma 5.2.1 to convert A' into cellular automata B.

Let L' be the language accepted by B. That is, the set of inputs so that B eventually reaches a steady state. Let $\sigma = \{0,1\}^{K'}$ be the alphabet of B where $K' = 2^{k'}$. Then for some

$$\begin{aligned} s =& O(\log(n' + m(n)\log(|\Sigma|) + S) \\ =& O(\log(n) + \log(S)) \\ =& O(\log(T)) \\ t =& O(\log(T)), \end{aligned}$$

we have that B uses 2^s cells of memory and runs in time 2^t.

Our prover for this algorithm is the same as our base **PCP**. We use the same notions of X, Y, f, and π from Lemma 5.4.1. Function $X : \mathbb{F}^s \times \mathbb{F}^{k'} \times \mathbb{F}^t \to \mathbb{F}$ should be the multilinear extension of the computation history of B. Function Y should be the low degree polynomial arithmetization of an inconsistency formula for X. Function f should be the sum check polynomial for Y, and π is a low degree polynomial containing X, and f. Let $d = s + k' + t = O(\log(T))$ be the degree of the correct X, as well as the number of variables in X. Recall that f can be though of as a sequence of $d + 1$ functions.

For the verifier, we need to be able to find the index of a specific symbol in the implicit input. We note each symbol in the second input after encoding into the alphabet for B will be stored in a power of 2 bits: $K \cdot K' = 2^{k+k'}$. Since $n' > K$ and both are powers of two, for some a, we have $n' = aK$. Then the ith symbol of the implicit input is uniquely determined by the area in memory from $n'K' + iKK' = (a+i)KK'$ to $(a + i + 1)KK' - 1$. This allows the prover to query a claimed multilinear extension of a single symbol from the second input using one application of Lemma 6.2.3, in the same way we do for the entire first input.

So our verifer on input x_1, with padded version x_1', does:

1. A low degree test on X using Lemma 6.2.2 so that we accept with probability at most δ if X has distance $\frac{1}{4}$ from every degree d function. Output $\perp$ if the low degree test fails.

2. Sum check of f, same as Lemma 5.4.1. Output $\perp$ if it would reject in our base **PCP**.

3. Consistency of X with f_{d+1}, same as Lemma 5.4.1, except we replace every query to X with an error corrected query using Lemma 6.2.3. That is, for every w so that the sum check step above makes a query to $f_{d+1}(w)$, calculate $Y(w)$ (from Lemma 5.2.3) using Lemma 6.2.3 for every call to X.

   Specifically, do the self corrected queries so that with probability at most $\frac{\delta}{2}$ will any error corrected query fail to return the degree d function closest to the provided X. This can be done since δ, w, and the number of calls to X used to calculate Y are constant. So the number of calls to Lemma 6.2.3 are constant.

   Output $\perp$ if any calls to Lemma 6.2.3 fails, or if any calculated $Y(w)$ disagrees with $f_{d+1}(w)$.

4. Consistency of X with y.

   For input x', it has transformed input $y = (y^1, y^2_{x'}, y^3)$ as described in Lemma 5.2.1. In particular, we can separate $y^2_{x'}$ into y^2_1 as the transformed version of x_1', and y^2_2 as the transformed version of x_2'.

   We already chose n', or rather x', so that $n' = |y^1| + |y^2_1| = 2^N$.

   Choose a random element, $v \in \mathbb{F}^{N+k'}$, and compute

   $$u(y, v) = \sum_{z \in \{0,1\}^{N+k'}} \operatorname{equ}(z, v) y_z.$$

   This can be done since for these z, y_z is either in y^1 or y^2_1, which we know. This is basically the multilinear extension of our first input. This should be equal to some entry in X at time 0, with 0 for most space coordinates, besides the first $N + k'$ being v: v'.

   Then use Lemma 6.2.3 to get an error corrected query to $X(v')$ with soundness $\delta/4$. Output $\perp$ if $X(v') \neq u(y, v)$.

5. Decode a symbol from x_2.

   We get an $i \in [m(n)]$ for the symbol of the second input we want to query. We make KK' queries to get a claimed value of what symbol i should be. These are just the bits at time 0 at locations $(a+i)KK', \ldots, (n'+i+1)KK'-1$. These together give a claimed value for the ith symbol, call it y'. Reject if any of these symbols are non binary.

   Then choose a random $b \in \mathbb{F}^{k+k'}$. Now we want to check if the multilinear extension of the claimed value for the ith symbol above is consistent with the proof. We do this the same way as the primary input, except that all the space field elements are set to the binary values for the ith index, and the bottom $k+k'$ are set to b. Then we can get an error corrected query of this one location using Lemma 6.2.3.

   That is, we calculate

   $$u'(y', b) = \sum_{z \in \{0,1\}^{k+k'}} \mathrm{equ}(z, b) y'_z.$$

   This should be equal to some entry of X at time 0, with the top space coordinates being $a+i$, and the bottom space coordinates being b: b'.

   Then use Lemma 6.2.3 to get an error corrected query to $X(b')$ with soundness $\delta/4$. Output $\perp$ if $X(b') \neq u(y', b)$. Otherwise, the ith symbol is probably what encodes to y'. Output what symbol of Σ that encodes to y'.

Now to prove this achieves the desired results.

1. Decoder time $O((\log(T)+n)\mathbf{polylog}(|\mathbb{F}|))$ and space $O(\log(T)\log(|\mathbb{F}|))$.

   Since $d = O(\log(T))$, the low degree test runs in time $O(d\mathbf{polylog}(|\mathbb{F}|)) = O(\log(T)\mathbf{polylog}(|\mathbb{F}|))$ and space $O(\log(T)\log(|\mathbb{F}|))$. The sum check uses time $O(\log(T)\mathbf{polylog}(|\mathbb{F}|))$ and space $O(\log(T)\log(|\mathbb{F}|))$.

   Checking consistency of X with f_{d+1} is only constantly many calculations of Y, which only takes time $O(d\mathbf{polylog}(|\mathbb{F}|)) = O(\log(T)\mathbf{polylog}(|\mathbb{F}|))$.

   Altogether, we only need to do a constant number of error correcting queries (Lemma 6.2.3), each of which takes time $O(\log(T)\mathbf{polylog}(|\mathbb{F}|))$ and space $O(\log(T)\log(|\mathbb{F}|))$.

   Like Lemma 5.4.1, checking the consistency of X with our input takes time $O((n+\log(T))\mathbf{polylog}(|\mathbb{F}|))$ and space $O(\log(T)\log(|\mathbb{F}|))$.

   When decoding a symbol from the implicit input, we need to compute a multilinear extension of a Boolean function with $O(\log(|\Sigma|)) = O(\log(T))$ inputs, which can be done in simultaneous time $O(\log(T)\mathbf{polylog}(|\mathbb{F}|))$ and space $O(\log(T)\log(|\mathbb{F}|))$.

2. Randomness $O(\log(T)\log(|\mathbb{F}|))$.

   The low degree test takes $O(\log(T)\log(|\mathbb{F}|))$ bits of randomness. The sum check, all the error corrected queries, and choosing the point to compare X to the input also take $O(\log(T)\log(|\mathbb{F}|))$ bits of randomness.

   We need $k'+k = O(\log(\log(|\Sigma|))) = O(\log(T))$ random field elements to choose b when checking the decoded symbol, which uses $O(\log(T)\log(|\mathbb{F}|))$ bits of randomness.

3. Encoder space $O(\log(T)\log(|\mathbb{F}|)+S)$.

   One can follow the proof from *Lemma* 5.4.1 since we are using the same encoder as that prover. Specifically, the simulation of the cellular automata from Lemma 5.2.1 is space efficient and only depends on the space of the RAM algorithm it simulates, not the size of it's input.

   Using oracle access to the state of the cellular automata, X is low space to calculate by Lemma 5.1.2, and Y is low space to calculate by Lemma 5.2.3, and so is the sum check Lemma 5.3.1.

4. $O(\log(T))$ queries.

   The low degree test, sum check, and error corrected queries all only require $O(\log(T))$ queries. We need to query $O(\log(T))$ locations to get the claimed symbol from the implicit input.

5. Alphabet $\mathbb{F}$.

   By definition of the proof.

6. Query time $O(\log(T)\textbf{polylog}(|\mathbb{F}|))$.

   Low degree tests and sum check have query time $O(\log(T)\textbf{polylog}(|\mathbb{F}|))$.

   The queries for decoding a symbol takes time $O(\log(T)\textbf{polylog}(|\mathbb{F}|))$ to calculate the index of the ith implicit input.

7. Soundness δ.

   The soundness argument is similar to our base **PCP**.

   Take any proposed proof, π. Let x_1 be the explicit input, x_1' its padded transformation, and y_1^2 the encoded input of $x' = x_1'$ properly encoded for B. Let L' be the language recognized by B.

   If the low degree test passes only with probability δ, then the probability we don't output $\perp$ is only δ, and we are done. Otherwise, X is within $1/4$ of a degree d polynomial, $\tilde{X}$.

   Let $\hat{X} = \text{MLB}(\tilde{X})$, and X' be $\hat{X}$ restricted to binary inputs.

   If X' does not start with state (y^1, y_1^2, z) for some z, $\hat{X}$ restricted to time 0 and 0 for everything but the first NK' spaces is not the multilinear extension of the state $y([n'])$. Thus neither is $\tilde{X}$. Then the probability that v is chosen so that $\tilde{X}$ agrees with $u(y, v)$ is at most $\frac{d}{|\mathbb{F}|} < \delta/4$. If they don't agree, with probability at most $\delta/4$ will our sample to $\tilde{X}(v)$ fail to show us they disagree. So overall, we succeed with probability at most δ.

   Suppose X' starts with state (y^1, y_1^2, z^1, z^2). If $(y^1, y_1^2, z^1, z^2) \notin L'$, then the computation history of X' is invalid. Then $\Gamma_B(\hat{X})$ (from Lemma 5.2.3) is not 0 on all Boolean inputs, so $\Gamma_B(\tilde{X})$ is not 0 on all boolean inputs. The probability that any $\Gamma_B(\tilde{X})$ is miscalculated is at most $\delta/2$ since the probability any error correcting query returns a response other than $\tilde{X}$ is $\delta/2$.

   By the soundness of Lemma 5.3.1, the sum check passes with probability at most

   $$\frac{O((d+1)\log(T))}{\|\mathbb{F}\|} \leq \frac{\delta}{2}$$

   for large enough α. So the probability the sum check succeeds or we miscalculate $\Gamma_B(\tilde{X})$ is at most δ, and one of these two has to happen to accept. So the probability we accept is at most δ.

   Suppose $(y^1, y_1^2, z^1, z^2) \in L'$. Since y^1 is correct, $z^2 = y^3$, and (y_1^2, y_2^2) must be some valid encoding of an elemnet in L, from Lemma 5.2.1. Since y_1^2 is a valid encoding of x_1', we have $x' = (x_1', x_2')$ for some x_2'. Further, x_2' corresponds to some unpadded x_2 such that $(x_1, x_2) \in L$.

   So we have our valid x_2, now we just have to show that we decode it or output $\perp$ with high probability. For index i, let y' be the state for cell i from directly sampling it from the proof. If $y' = y_2^2(i)$, then we either output $\perp$, or the symbol $y_2^2(i)$ decoded from y', which is what we want. So suppose $y' \neq y_2^2(i)$.

   Then $\tilde{X}$ restricted to the projection of the variables for y_2^2 at cell i is a different degree $d = O(\log(T))$ polynomial than the multilinear extension of y'. Then they agree on at most $\frac{d}{|\mathbb{F}|} < \delta/4$ locations. So the probability we chose a location to check that they agree is at most $\delta/4$. Finally, the probability we fail to accurately query this location in $\tilde{X}$ is at most $\delta/4$. So the probability we don't output $\perp$ is at most δ.

8. Perfect completeness.

   An honest proof has an X of degree d, so passes the low degree test. Will have no inconsistencies, so passes the sum check and X is consistent with Y. Since X is degree d, all error corrected queries succeed. Since X is honest, it will be consistent with the input. Since X is honest, it will pass the check when decoding a symbol from the implicit input.

9. Log of proof length $O(\log(T)\log(|\mathbb{F}|))$.

   Same as Lemma 5.4.1, this follows from the fact the proof is just a function $\pi : \mathbb{F}^{O(\log(T))} \to \mathbb{F}$.

$\square$

6.4 Constructing our Efficient PCP

Finally, we can use Lemma 5.4.1 to get an **ePCP** and Theorem 4.4.1 to get an **rPCP**. Then we use Lemma 6.3.1 to get a **dPCP** that we use in Theorem 6.1.2 to get a query efficient **PCP** with constant soundness. Then we use repetitions for amplification to get a **PCP** with a small constant soundness, which proves Theorem 1.1.3.

Theorem 1.1.3 (Verifier Efficient **PCP**). *Let $S, T = \Omega(n)$ be functions, and L be any language computed by a simultaneous time T and space S algorithm. Let $\delta \in (0, 1/2)$ be a constant. Then there is a* **PCP** *for L with:*

1. *Verifier time $\tilde{O}(n + \log(T))$.*
2. *Query time $\tilde{O}(\log(T))$.*
3. *$O(\log(n) + \log(\log(T)))$ queries.*
4. *Alphabet Σ with $\log(|\Sigma|) = O(\log(\log(T)))$.*
5. *Log of proof length $\tilde{O}(\log(T))$.*
6. *Prover space $\tilde{O}(S)$.*
7. *Perfect completeness and soundness δ.*

Proof. Let $\mathbb{F}$ be a field with $|\mathbb{F}| = \alpha \log(T)^2$ for sufficiently large α. Use Lemma 5.4.1 to get an **ePCP**, A', with:

1. Verifier time $O((\log(T) + n)\mathbf{polylog}(|\mathbb{F}|))$ and space $O(\log(T)\log(|\mathbb{F}|))$.
2. Randomness $O(\log(T)\log(|\mathbb{F}|))$.
3. Prover space $O(\log(|\mathbb{F}|)\log(T) + S)$.
4. $O(\log(T))$ queries.
5. Alphabet $\mathbb{F}$.
6. Extrapolation time $O(\log(T)\mathbf{polylog}(|\mathbb{F}|))$.
7. Degree $O(\log(T))$ and $O(\log(T))$ variables.
8. Perfect completeness.
9. Low degree soundness 0.1.
10. Log of proof length $O(\log(T)\log(|\mathbb{F}|))$.

Then run Theorem 4.4.1 to get a **rPCP**, A with:

1. Verifier time polynomial in $\log(T), n$, and $\mathbf{polylog}(|\mathbb{F}|)$.
2. Verifier space $O(\log(|\mathbb{F}|)\log(T) + S)$.
3. Randomness $r(n) = O(\log(T)\log(|\mathbb{F}|))$.
4. Alphabet $\mathbb{F}$.
5. $O(|\mathbb{F}|)$ queries.
6. Query time $O(\log(T)\mathbf{polylog}(|\mathbb{F}|))$.
7. It has log of proof length $O(\log(T)\log(|\mathbb{F}|))$ since it has the same prover as A'.
   And prover space $O(\log(|\mathbb{F}|)\log(T) + S)$.

8. Perfect completeness and soundness 0.99.

Let V be the verifier for A. Now let $L' = \{((x,r),p) : V(x,r,p) = 1\}$. Then L' is a pair language, where for length $n' = n + r(n) = O(n + \log(T)\log(|\mathbb{F}|))$ first inputs, there is a length $m = O(|\mathbb{F}|) = \mathbf{poly}(\log(T)) = \mathbf{poly}(n')$ second input with symbols from $\mathbb{F}$. Further language L' is decided by a Turing machine running in time $\mathbf{poly}(n')$, and space $O(n' + S)$

Note that Lemma 6.3.1 only holds for algorithms with time and space bounds at least n. So we bound the time and space of our verifier V by $\mathbf{poly}(n')$. See that $\log(|\mathbb{F}|) = O(\log(\log(T))) = O(\log(n'))$ and for any α', for sufficiently large α, we have $|\mathbb{F}| > \alpha \log(T)^2 \geq \alpha' \log(n')^2$.

Then by Lemma 6.3.1, there is a **dPCP** for L', B, such that B has:

1. Decoder time
$$\begin{aligned} &O((\log(n') + n')\mathbf{polylog}(|\mathbb{F}|)) \\ =&O((n + \log(T)\log(|\mathbb{F}|))\mathbf{polylog}(|\mathbb{F}|)). \end{aligned}$$
2. Randomness
$$O(\log(n')\log(|\mathbb{F}|)) = O(\log(T)\log(|\mathbb{F}|)).$$
3. Encoder space
$$O(\log(n')\log(|\mathbb{F}|) + S) = O(\log(T)\log(|\mathbb{F}|) + S).$$
4. $O(\log(n')) = O(\log(n) + \log(\log(T)))$ queries.
5. Alphabet $\mathbb{F}$.
6. Query time
$$O(\log(n')\mathbf{polylog}(|\mathbb{F}|)) = O((\log(n) + \log(\log(T)))\mathbf{polylog}(|\mathbb{F}|)).$$
7. Perfect completeness and soundness 0.005.
8. Log of proof length
$$O(\log(n')\log(|\mathbb{F}|)) = O((\log(n) + \log(\log(T)))\log(|\mathbb{F}|)).$$

Then by Theorem 6.1.2, we have a PCP for L, C, such that C has:

1. Verifier time $O((n + \log(T)\log(|\mathbb{F}|))\mathbf{polylog}(|\mathbb{F}|))$
2. Query time $O(\log(T)\mathbf{polylog}(|\mathbb{F}|))$.
3. $O(\log(n) + \log(\log(T)))$ queries.
4. Alphabet $\mathbb{F}$.
5. Log of proof length $O(\log(T)\log(|\mathbb{F}|))$
6. Prover space $O(\log(T)\mathbf{polylog}(|\mathbb{F}|) + S)$.
7. Perfect completeness and soundness 0.995.

Then, by repeating this for $200\ln(1/\delta)$ times, we get a **PCP** that uses within a constant factor the same time, space, and number of queries, and has soundness δ.

□

We note that our **PCP** construction only ever uses the space assumption to make our prover efficient. Thus the same proofs also can be used to prove Theorem 1.1.4.

6.5 Fine Grained MIP = NEXP

For completeness, we also prove our fine grained equivalence between **MIP** and **NEXP**: Corollary 1.1.5.

Corollary 1.1.5 (Fine Grained Equivalence of **MIP** = **NEXP**). *For any time constructable function $p(n) = \Omega(n)$, language $L \in$ **NTIME**$[2^{\tilde{O}(p(n))}]$ if and only if there is a two prover, one round* **MIP** *protocol for L whose verifier runs in time $\tilde{O}(p(n))$.*

Proof. First, there is a simple nondeterministic algorithm for a language with an **MIP** protocol: nondeterministically guess the entire prover strategy, then run the verifier for every choice of randomness on that strategy. So any language L with an **MIP** protocol with a time $\tilde{O}(p(n))$ verifier has a nondeterministic algorithm running in time $2^{\tilde{O}(p(n))}$.

For a language $L \in$ **NTIME**$[2^{\tilde{O}(p(n))}]$, we first transform it to a two query **MIP** with high error, than apply a parallel repetition theorem [Raz98; Hol07; Rao08] to get back to constant soundness.

For the high error **MIP**, the verifier chooses the randomness for the **PCP** in Theorem 1.1.4 and asks one prover for the value of the **PCP** at every location it would query, and the second prover for one of those locations randomly. This has soundness $1 - \Omega\left(\frac{1}{\log(p(n))}\right)$, since the **PCP** has constant soundness, and $O(\log(p(n)))$ queries.

Finally, we notice that this **MIP** is in fact a projection game, and thus by [Rao08], repeating this protocol k times in parallel gives soundness

$$\left(1 - \Omega\left(\frac{1}{\log(p(n))}\right)\right)^{\Omega(k/\log(p(n)))}.$$

Setting $k = O(\log(p(n))^2)$ amplifies the high error **MIP** to constant soundness.

This only requires running the time $\tilde{O}(p(n))$ verifier from the **PCP** for **poly**$(\log(p(n)))$ many times. Thus this **MIP** verifier also runs in time $\tilde{O}(p(n))$. $\square$

7 Open Problems

There are several ways we would like to improve the circuit lower bounds.

1. Remove the advice bit.

   We still had to use advice, a limitation from the original Santhanam result. It would be nice if we could get lower bounds on **MA** with no non-uniformity.

2. Prove tight bounds for all k.

   Another limitation of our circuit lower bound is that it does not prove this tight bound for all $k > 1$, just for some k.

   The major barrier is in the case that **SPACE**$[n]$ algorithms may require super linear, but polynomial, sized circuits. Then the circuit size required for any given space may change in a strange way. For example, suppose for some $a > 1$

   $$\mathbf{SPACE}[n] \subseteq \mathbf{SIZE}[O(n^a)] \setminus \mathbf{SIZE}[o(n^a)].$$

   What we would like, but this does not obviously imply, is that for all $b > 1$:

   $$\mathbf{SPACE}[n^b] \subseteq \mathbf{SIZE}[O(n^{ab})] \setminus \mathbf{SIZE}[o(n^{ab})].$$

   While a padding argument gives **SPACE**$[n^b] \subseteq$ **SIZE**$[O(n^{ab})]$, it does not give **SPACE**$[n^b] \not\subseteq$ **SIZE**$[o(n^{ab})]$. We may even have something weird, like

   $$\mathbf{SPACE}[n^a] \subseteq \mathbf{SIZE}[O(n^a)] \setminus \mathbf{SIZE}[o(n^a)].$$

   That is, even if space n algorithms require circuit size n^a, we may not need larger circuits until our algorithms use more space than n^a.

In this case, to get circuit lower bounds greater than n^a, we need to use an algorithm with space greater than n^a. Unfortunately, our verifier uses queries to the prover of the same length as the space of the algorithm being verified. Then the prover needs to use linear space in its input length, and may require size $(n^a)^a = n^{a^2}$ circuits.

One way to try to solve this problem is to show that if

$$\mathbf{SPACE}[n] \subseteq \mathbf{SIZE}[O(n^a)] \setminus \mathbf{SIZE}[o(n^a)]$$

for some $a > 1$, then for all $b > 1$:

$$\mathbf{SPACE}[n^b] \subseteq \mathbf{SIZE}[O(n^{ab})] \setminus \mathbf{SIZE}[o(n^{ab})].$$

This seems plausible, but hard to prove.

Another direction is to find an efficient **PCP** for $\mathbf{SPACE}[n^b]$ with prover queries shorter than n^b (or equivalently, proof length less than 2^{n^b}). But this seems hard as shorter **PCP** proofs imply more efficient algorithms.

For instance, for constant c, if L has a **PCP** with polynomial time verifier and proof length 2^{n^c}, then $L \in \mathbf{MATIME}[O(2^{n^c})]$ just by guessing the whole proof string, and verifying it. So if every language in $\mathbf{NTIME}[O(2^{n^b})]$ had a **PCP** with proof length $O(2^{n^c})$, then we would have

$$\mathbf{NTIME}[O(2^{n^b})] \subseteq \mathbf{MATIME}[O(2^{n^c})].$$

If $c < b$, this would contradict a derandomization conjecture that

$$\mathbf{MATIME}[f(n)] \subseteq \mathbf{NTIME}[\mathbf{poly}(f(n))].$$

Thus any more efficient **PCP** either must not apply to nondeterministic algorithms (ours does), or **MA** cannot be efficiently derandomized. This does not rule out this approach, but is a major challenge.

3. Make lower bound more frequent.

   Another direction is improving the infinitely often separation to a more frequently often separation. Murray and Williams [MW18] gave a refinement of the Santhanam circuit lower bounds that is incomparable to ours. In it they proved that for some $L \in \mathbf{MA}/O(\log(n))$ and constant c, for almost every n, either L on length n inputs wouldn't have circuits with size n^k, or L on length n^{ck} inputs wouldn't have circuits with size $n^{c^2k^2}$. One might want to strengthen their results.

   We conjecture that there exists some constant $k > 1$, function $f(n) = o(1)$, gap function $g(n) = \mathbf{poly}(n)$, and language $L' \in \mathbf{MATIME}[O(n^{k+f(n)})]/O(\log(n))$ such that for all n there is some $m \in [n, g(n)]$ such that language L' on length m inputs does not have circuits of size m^k.

   The Murray and Williams result produces a language L that for every input length n will either be the downward self reducible language from Santhanam's result (Y in Lemma 3.2.1), or a circuit found with exhaustive search (like in Lemma 3.3.1). If the prover circuit for the exhaustive search is small enough, then L is exhaustive search. Otherwise, L is (possibly padded) Y.

   The idea is that if exhaustive search on length n inputs doesn't have small prover circuits, than for the input length of the prover circuits, we have a hard problem (specifically, Y). Unfortunately, provers have input length about n^{ck} for some constant c. For that prover to be hard enough for our circuit lower bound, length n^{ck} inputs must require size n^{ck^2} circuits. So to make sure the provers are hard enough, length n inputs for exhaustive search may have to use prover circuits as large as n^{ck^2}!

   Our **PCP** can improve the constant c in the Murray and Williams result, but improving the approximately n^{k^2} verifier time to near n^k requires new ideas.

4. Prove exponential lower bounds for **MAEXP**.

   A similar problem is to prove exponential circuit lower bounds for the exponential version of **MA**, known as **MAEXP**. The best circuit lower bounds known for **MAEXP** are "half-exponential" by Miltersen, Vinodchandran, and Watanabe [MVW99]. Loosely, a function is half exponential if that function composed with itself is exponential.

One could also look to improve our **PCP**. In particular, one could try to replicate other existing results while maintaining the $\tilde{O}(n + \log(T))$ runtime. Standard techniques can reduce the number of queries, or improve the soundness. We suspect these techniques can be used to give $\mathbf{poly}(T)$ proof length. Can we construct **PCP**s with length $\tilde{O}(T)$ proofs while having a $\tilde{O}(n+\log(T))$ verifier runtime? Known **PCP**s with proof length $\tilde{O}(T)$ have verifier time $\Omega(n + \log(T)^2)$.

In particular, we suspect ideas from theorem 2.6 in [Ben+05][6] (which extends the results of [BS+04] from **NP** to **NEXP**) may give a **PCP** with verifier time near $\tilde{O}(n + \log(T))$ while giving a proof of length $T^{1+o(1)}$. But it would not have proof length $\tilde{O}(T) = T\mathbf{polylog}(T)$. We stress that such an analysis has not been performed, and the **PCP** of [Ben+05] is much more complex than ours.

References

[AB09] Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach*. 1st. USA: Cambridge University Press, 2009. ISBN: 0521424267.

[AS97] Sanjeev Arora and Madhu Sudan. "Improved Low-Degree Testing and Its Applications". In: *Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing*. STOC '97. El Paso, Texas, USA: Association for Computing Machinery, 1997, 485–495. ISBN: 0897918886. DOI: 10.1145/258533.258642. URL: https://doi.org/10.1145/258533.258642.

[AS98] Sanjeev Arora and Shmuel Safra. "Probabilistic Checking of Proofs: A New Characterization of NP". In: *J. ACM* 45.1 (Jan. 1998), 70–122. ISSN: 0004-5411. DOI: 10.1145/273865.273901. URL: https://doi.org/10.1145/273865.273901.

[Aro+98] Sanjeev Arora, Carsten Lund, Rajeev Motwani, Madhu Sudan, and Mario Szegedy. "Proof Verification and the Hardness of Approximation Problems". In: *J. ACM* 45.3 (May 1998), 501–555. ISSN: 0004-5411. DOI: 10.1145/278298.278306. URL: https://doi-org.ezproxy.lib.utexas.edu/10.1145/278298.278306.

[BFL90] L. Babai, L. Fortnow, and C. Lund. "Nondeterministic exponential time has two-prover interactive protocols". In: *Proceedings [1990] 31st Annual Symposium on Foundations of Computer Science*. 1990, 16–25 vol.1. DOI: 10.1109/FSCS.1990.89520.

[BS+04] Eli Ben-Sasson, Oded Goldreich, Prahladh Harsha, Madhu Sudan, and Salil Vadhan. "Robust Pcps of Proximity, Shorter Pcps and Applications to Coding". In: *Proceedings of the Thirty-Sixth Annual ACM Symposium on Theory of Computing*. STOC '04. Chicago, IL, USA: Association for Computing Machinery, 2004, 1–10. ISBN: 1581138520. DOI: 10.1145/1007352.1007361. URL: https://doi.org/10.1145/1007352.1007361.

[BV14] Eli Ben-Sasson and Emanuele Viola. "Short PCPs with Projection Queries". In: *Automata, Languages, and Programming - 41st International Colloquium, ICALP 2014, Copenhagen, Denmark, July 8-11, 2014, Proceedings, Part I*. Ed. by Javier Esparza, Pierre Fraigniaud, Thore Husfeldt, and Elias Koutsoupias. Vol. 8572. Lecture Notes in Computer Science. Springer, 2014, pp. 163–173. DOI: 10.1007/978-3-662-43948-7_14. URL: https://doi.org/10.1007/978-3-662-43948-7_14.

[Ben+05] E. Ben-Sasson, O. Goldreich, P. Harsha, M. Sudan, and S. Vadhan. "Short PCPs verifiable in polylogarithmic time". In: *20th Annual IEEE Conference on Computational Complexity (CCC'05)*. 2005, pp. 120–134. DOI: 10.1109/CCC.2005.27.

[Ben+13] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, and Eran Tromer. "On the Concrete Efficiency of Probabilistically-Checkable Proofs". In: *Proceedings of the Forty-Fifth Annual ACM Symposium on Theory of Computing*. STOC '13. Palo Alto, California, USA: Association for Computing Machinery, 2013, 585–594. ISBN: 9781450320290. DOI: 10.1145/2488608.2488681. URL: https://doi.org/10.1145/2488608.2488681.

[6] We emphasize this is the theorem labeled "Efficient PCPPs with small query complexity", which does not have quasi-linear proof length.

[Coo72] Stephen A. Cook. “A Hierarchy for Nondeterministic Time Complexity”. In: *Proceedings of the Fourth Annual ACM Symposium on Theory of Computing*. STOC ’72. Denver, Colorado, USA: Association for Computing Machinery, 1972, 187–192. ISBN: 9781450374576. DOI: 10.1145/800152.804913. URL: https://doi.org/10.1145/800152.804913.

[DH09] Irit Dinur and Prahladh Harsha. “Composition of Low-Error 2-Query PCPs Using Decodable PCPs”. In: *2009 50th Annual IEEE Symposium on Foundations of Computer Science*. 2009, pp. 472–481. DOI: 10.1109/FOCS.2009.8.

[DR04] Irit Dinur and Omer Reingold. “Assignment testers: towards a combinatorial proof of the PCP-theorem”. In: *45th Annual IEEE Symposium on Foundations of Computer Science*. 2004, pp. 155–164. DOI: 10.1109/FOCS.2004.16.

[Dor+20] Dean Doron, Dana Moshkovitz, Justin Oh, and David Zuckerman. “Nearly Optimal Pseudorandomness From Hardness”. In: *61st IEEE Annual Symposium on Foundations of Computer Science, FOCS 2020, Durham, NC, USA, November 16-19, 2020*. IEEE, 2020, pp. 1057–1068.

[FS11] Lance Fortnow and Rahul Santhanam. “Robust Simulations and Significant Separations”. In: *Proceedings of the 38th International Colloquim Conference on Automata, Languages and Programming - Volume Part I*. ICALP’11. Zurich, Switzerland: Springer-Verlag, 2011, 569–580. ISBN: 9783642220050.

[FS95] Katalin Friedl and Madhu Sudan. “Some Improvements to Total Degree Tests”. In: *In Proceedings of the 3rd Annual Israel Symposium on Theory of Computing and Systems*. 1995, pp. 190–198.

[FST05] Lance Fortnow, Rahul Santhanam, and Luca Trevisan. “Hierarchies for Semantic Classes”. In: *Proceedings of the Thirty-Seventh Annual ACM Symposium on Theory of Computing*. STOC ’05. Baltimore, MD, USA: Association for Computing Machinery, 2005, 348–355. ISBN: 1581139608. DOI: 10.1145/1060590.1060642. URL: https://doi.org/10.1145/1060590.1060642.

[FSW09] Lance Fortnow, Rahul Santhanam, and Ryan Williams. “Fixed-Polynomial Size Circuit Bounds”. In: *2009 24th Annual IEEE Conference on Computational Complexity*. 2009, pp. 19–26. DOI: 10.1109/CCC.2009.21.

[HR18] Justin Holmgren and Ron Rothblum. “Delegating Computations with (Almost) Minimal Time and Space Overhead”. In: *2018 IEEE 59th Annual Symposium on Foundations of Computer Science (FOCS)*. 2018, pp. 124–135. DOI: 10.1109/FOCS.2018.00021.

[HS65] J. Hartmanis and R. E. Stearns. “On the Computational Complexity of Algorithms”. In: *Transactions of the American Mathematical Society* 117 (1965), pp. 285–306. ISSN: 00029947. URL: http://www.jstor.org/stable/1994208.

[Hol07] Thomas Holenstein. “Parallel Repetition: Simplifications and the No-Signaling Case”. In: *Proceedings of the Thirty-Ninth Annual ACM Symposium on Theory of Computing*. STOC ’07. San Diego, California, USA: Association for Computing Machinery, 2007, 411–419. ISBN: 9781595936318. DOI: 10.1145/1250790.1250852. URL: https://doi.org/10.1145/1250790.1250852.

[Lun+92] Carsten Lund, Lance Fortnow, Howard Karloff, and Noam Nisan. “Algebraic Methods for Interactive Proof Systems”. In: *J. ACM* 39.4 (Oct. 1992), 859–868. ISSN: 0004-5411. DOI: 10.1145/146585.146605. URL: https://doi.org/10.1145/146585.146605.

[MP06] D. van Melkebeek and K. Pervyshev. “A generic time hierarchy for semantic models with one bit of advice”. In: *21st Annual IEEE Conference on Computational Complexity (CCC’06)*. 2006, 14 pp.–144. DOI: 10.1109/CCC.2006.7.

[MR08] Dana Moshkovitz and Ran Raz. “Two Query PCP with Sub-Constant Error”. In: *2008 49th Annual IEEE Symposium on Foundations of Computer Science*. 2008, pp. 314–323. DOI: 10.1109/FOCS.2008.60.

[MVW99] Peter Bro Miltersen, N. V. Vinodchandran, and Osamu Watanabe. “Super-Polynomial versus Half-Exponential Circuit Size in the Exponential Hierarchy”. In: *Proceedings of the 5th Annual International Conference on Computing and Combinatorics*. COCOON’99. Tokyo, Japan: Springer-Verlag, 1999, 210–220. ISBN: 3540662006.

[MW18] Cody Murray and Ryan Williams. “Circuit Lower Bounds for Nondeterministic Quasi-Polytime: An Easy Witness Lemma for NP and NQP”. In: *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing*. STOC 2018. Los Angeles, CA, USA: Association for Computing Machinery, 2018, 890–901. ISBN: 9781450355599. DOI: 10.1145/3188745.3188910. URL: https://doi.org/10.1145/3188745.3188910.

[Mei09] Or Meir. “Combinatorial PCPs with Efficient Verifiers”. In: *2009 50th Annual IEEE Symposium on Foundations of Computer Science*. 2009, pp. 463–471. DOI: 10.1109/FOCS.2009.10.

[Rao08] Anup Rao. “Parallel Repetition in Projection Games and a Concentration Bound”. In: *Proceedings of the Fortieth Annual ACM Symposium on Theory of Computing*. STOC ’08. Victoria, British Columbia, Canada: Association for Computing Machinery, 2008, 1–10. ISBN: 9781605580470. DOI: 10.1145/1374376.1374378. URL: https://doi.org/10.1145/1374376.1374378.

[Raz98] Ran Raz. “A Parallel Repetition Theorem”. In: *SIAM Journal on Computing* 27.3 (1998), pp. 763–803. DOI: 10.1137/S0097539795280895. eprint: https://doi.org/10.1137/S0097539795280895. URL: https://doi.org/10.1137/S0097539795280895.

[SFM78] Joel I. Seiferas, Michael J. Fischer, and Albert R. Meyer. “Separating Nondeterministic Time Complexity Classes”. In: *J. ACM* 25.1 (1978), 146–167. ISSN: 0004-5411. DOI: 10.1145/322047.322061. URL: https://doi.org/10.1145/322047.322061.

[San07] Rahul Santhanam. “Circuit Lower Bounds for Merlin-Arthur Classes”. In: *Proceedings of the Thirty-Ninth Annual ACM Symposium on Theory of Computing*. STOC ’07. San Diego, California, USA: Association for Computing Machinery, 2007, 275–283. ISBN: 9781595936318. DOI: 10.1145/1250790.1250832. URL: https://doi.org/10.1145/1250790.1250832.

[Sha92] Adi Shamir. “IP = PSPACE”. In: *J. ACM* 39.4 (Oct. 1992), 869–877. ISSN: 0004-5411. DOI: 10.1145/146585.146609. URL: https://doi.org/10.1145/146585.146609.

[Wil11] Ryan Williams. “Non-uniform ACC Circuit Lower Bounds”. In: *2011 IEEE 26th Annual Conference on Computational Complexity*. 2011, pp. 115–125. DOI: 10.1109/CCC.2011.36.

[Žá83] Stanislav Žák. “A Turing machine time hierarchy”. In: *Theoretical Computer Science* 26.3 (1983), pp. 327–333. ISSN: 0304-3975. DOI: https://doi.org/10.1016/0304-3975(83)90015-4. URL: https://www.sciencedirect.com/science/article/pii/0304397583900154.

A PCP Composition Proof

Here is the proof of **PCP** composition: Theorem 6.1.2.

Theorem 6.1.2 (**PCP** Composition)**.** *Suppose L is a language with an* **rPCP***, A, with verifier V, prover P, query function Q, and index function I such that*

1. *Q runs in time $t(n)$.*
2. *P run in space $s(n)$.*
3. *V uses $r(n)$ bits of randomness.*
4. *A uses alphabet Σ.*
5. *A has robust soundness δ.*
6. *A has perfect completeness.*
7. *A has proof length $l(n)$.*

Suppose $L' = \{((x, r), y) : V(x, r, y) = 1\}$. Let $n' = n + r(n)$. Suppose L' has a **dPCP** *protocol, B, with decoder D and encoder E such that*

1. *E runs in space $s'(n')$.*

2. *D runs in time $t'(n')$.*

3. *B uses $q'(n')$ queries.*

4. *B has query time $t^*(n')$.*

5. *B uses alphabet Σ'.*

6. *B has soundness δ'.*

7. *B has perfect completeness.*

8. *B has proof length $l'(n')$.*

Then there is a **PCP** *protocol for L, C, such that*

1. *C has verifier time $O(t'(n'))$*

2. *C uses $O(q'(n'))$ queries.*

3. *C has prover space $O(s(n) + t(n) + s'(n'))$.*

4. *C uses alphabet $\Sigma' \cup \Sigma$.*

5. *C has query time $O(t(n) + t^*(n'))$.*

6. *C has soundness $\delta + \delta'$.*

7. *C has perfect completeness.*

8. *C has proof length $l(n) + 2^{r(n)}l'(n')$.*

Proof. At a high level, our new **PCP** will essentially use the **dPCP** to prove that a query would pass the **rPCP**. The decoding property of the **dPCP** forces the proof to almost commit to a single accepting result for these query locations. And the robustness property of the **rPCP** means that on average, the proof is far away from the accepting one the **dPCP** claimed. So the two will disagree most of the time.

Suppose B uses $r'(n)$ bits of randomness. Then our new verifier, V', will expect for its randomness (r, r', i) where r is $r(n)$ bits of randomness for V, r' is $r'(n)$ bits of randomness for D, and i is a uniformly random element of $[q(n)]$ where $q(n)$ is the number of queries for A. Then C has a proof length of $l(n) + 2^{r(n)}l'(n)$ which we write, for proof π, as one substring π' of length $l(n)$, and for each possible value of r, a substring π^r of length $l'(n)$.

Let I' be the index function for B. Then finally, our new **PCP** just checks if

$$D((x,r), r', \pi^r_{I'((x,r),r',i)}, i) = \pi'_{Q(x,r,i)}.$$

The proof is expected to have π' as the proof for the **rPCP**, and then for each r, π^r should be the proof for the inner verifier that $V(x, r, \pi'_{I(x,r)}) = 1$, where I is the index function for A.

1. The time of the new verifier, is the time to run the **dPCP** decoder, $t'(n')$, plus the time to compare the result to a symbol in the **rPCP** proof. This comparison takes time linear in the symbol size, which since the decoder decodes a symbol, is at most $O(t'(n'))$ time. Thus the composed verifier takes time $O(t'(n'))$.

2. The total number of queries are just the number for the inner **dPCP**, $q'(n')$, plus one to check consistency with the outer **rPCP**.

3. The proof only requires space $s'(n')$ to compute the symbols from **dPCP** when given query access to symbols from the **rPCP** for a choice of randomness.

   To calculate the index in the **rPCP** proof of one of the symbols given to the verifier for a choice of randomness requires time, and space, $t(n)$.

   To calculate a symbol of the proof for the **rPCP** only requires space $s(n)$. So computing the symbols from the inner **dPCP** only require space $O(s'(n') + t(n) + s(n))$. And of course, queries to the outer **rPCP** only require space $s(n)$.

4. The symbols for π' are in Σ, and the symbols for π^r are in Σ', so the alphabet is $\Sigma' \cup \Sigma$.

5. A query location for C will either be to a query location of A or B, which can be computed in time $t(n)$ or $t^*(n')$. In either case, it can be computed in time $O(t(n) + t^*(n'))$.

6. For soundness, suppose $x \notin L$. Then by robust soundness of A, for any proof π', for $Y_r = \{y : V(x, r, y) = 1\}$:
$$\mathrm{E}_r[\Delta(\pi'_{I(x,r)}, Y_r)] \geq 1 - \delta.$$
Then for any r, for any π^r, by the soundness of the **dPCP**, either
$$\Pr_{r',i}[D((x,r), r', \pi^r_{I'((x,r),r',i)}, i) \neq \perp] \leq \delta',$$
or there is some y^r where $V(x, r, y^r) = 1$ and
$$\Pr_{r',i}[D((x,r), r', \pi^r_{I'((x,r),r',i)}, i) \notin \{y^r_i, \perp\}] \leq \delta'.$$
Then the probability that we accept is the probability that
$$D((x,r), r', \pi^r_{I'((x,r),r',i)}, i) = \pi'_{Q(x,r,i)}.$$
This can happen in 3 ways. If y_r doesn't exist, then we accept only if $D((x,r), r', \pi^r_{I'((x,r),r',i)}, i) \neq \perp$. If y_r does exist, then either $y^r_i = \pi'_{Q(x,r,i)}$ or $D((x,r), r', \pi^r_{I'((x,r),r',i)}, i) \notin \{\perp, y^r_i\}$.
Then we can bound the probability of acceptance by:
$$\begin{aligned}\Pr[\text{accept}] \leq &\Pr_{r,i}[y_r \text{ exists} \wedge D((x,r), r', \pi^r_{I'((x,r),r',i)}, i) \notin \{y^r_i, \perp\}] \\ &+ \Pr_{r,i}[y_r \text{ exists} \wedge y^r_i = \pi'_{I(x,r)}(i)] \\ &+ \Pr_{r,i}[y_r \text{ doesn't exist} \wedge \Big(D((x,r), r', \pi^r_{I'((x,r),r',i)}, i) \neq \perp\Big)] \\ \leq &\Pr_{r,i}[y_r \text{ exists}]\delta' + \delta + \Pr_{r,i}[y_r \text{ doesn't exist}]\delta' \\ \leq &\delta' + \delta.\end{aligned}$$

7. The protocol has completeness since if $x \in L$, there is some proof $\pi^x = \pi'$ so that for any of the chosen r, then $V(x, r, \pi^x_{I(x,r)}) = 1$. Then from completeness of the **dPCP**, there is a π^r so that for any i, the inner **PCP** will always return $\pi^x_{I(x,r)}(i)$.

□

B Automata Proofs

In this section, we show how to construct the cellular automata for an algorithm, and how to arithmetize a formula for its rules. First, let us show how to construct the cellular automata for a RAM algorithm: Lemma 5.2.1.

Lemma 5.2.1 (RAM algorithms have simple cellular automata)**.** *Let A be a RAM algorithm recognizing L, running in time T and space S where $S = \Omega(\log(n))$ and $T = \Omega(S)$. Further, A uses input coming from a read only space of n bits.*

Then there is a 1 dimensional cellular automata, B, simulating A, such that

1. *B runs in time $T' = \mathbf{poly}(T, n)$, and space $S' = O(n + S)$.*

2. *B has a constant size alphabet, Σ, where for some k, we have $|\Sigma| = 2^{2^k}$. That is, Σ is represented by a power of 2 number of bits.*

3. *For any input x for A, there is a corresponding input for B, y_x, of length S'. And we also have that $y_x = (y^1, y_x^2, y^3)$ where*
   (a) *y^1 has length $O(\log(S'))$ and is independent of the specific x, only the length of x, and y^1 is computable in time $O(|y^1|)$.*
   (b) *y_x^2 is exactly n symbols where for some $f : \{0, 1\} \to \Sigma$, for each $i \in [n]$, $(y_x^2)_i = f(x_i)$, where f is computable in constant time.*
   (c) *y^3 is exactly S copies of a specific symbol in Σ.*
4. *Not all transitions for B will be defined, and A accepts on x if and only if after time T' starting on y_x, B reaches a steady state. Similarly, A rejects on x if and only if there is no sequence of T' valid transitions in B starting from y_x.*
5. *If B has a starting state that is (y^1, z) for any z that is not (y_x^2, y^3) for some $x \in L$, then B will not have T' valid transitions.*
6. *Let $x \in L$ be an input for A, with transformed input for B, y_x. Given a time $t \in [T']$ and a memory location $s \in [S']$, there is a RAM algorithm C that can compute the symbol in cell s at time t in B's computation history on y_x in time $O(T)$ and space $O(S)$ given read only access to x.*

Proof. The idea is simple: First convert the RAM machine into an input oblivious, single tape Turing machine, B', where there is $O(\log(S + n))$ space for the registers, n space for the read only input, followed by S working space reserved on the tape. Notably, this input oblivious Turing machine may temporarily modify the contents of this read only space. In fact, it needs to. But these will always be temporary since we are simulating a RAM machine where these are read only.

Then we create our cellular automata, B, by encoding this Turing machine's state into the cell the head is on, alongside that cell's symbol. Then the state transitions will come from whether the current state has the head, or a neighbor does. On accepting, the cellular automata will just remove the head and remain constant. On rejecting, there will just be an undefined transition. This makes accepting equivalent to the existence of a valid computation history.

For more details, first, we take our input RAM algorithm A, and make a new RAM algorithm A' that does the same thing, but starts by making sure its working space is all 0.

Turing machine B' can be made from A' by first adding $O(\log(S'))$ bits before the first bit to hold the current memory configuration of the registers. Then the Turing Machine starts at the beginning, goes through the motions it would need to do on any register to register operation and any state change. Then it goes from the beginning forward, looking for the index it wants to operate on for any register memory operation.

Each time it moves, it copies the bit in front of the head behind it, and shifts all its registers 1 forward. At each potential bit, it moves the tape head as if it was going to do every operation, but doesn't actually do it unless the indexes match for a register memory operation. Once it gets to the far side, it returns the registers to the start.

This Turing Machine runs in time $T' = O(T(S + n)\log(S')^2)$ and only needs size $S' = O(S + n)$ to hold bits of the computation, and the registers. In particular, at time 0, it has $O(\log(S + n)$ space reserved for the registers, exactly n cells reserved for the read only input, and S cells reserved for the working space.

Then B' can be made into a cellular automata, B^*, by expanding the alphabet to be a pair of an entry from the alphabet of the Turing Machine, and an entry of the state of the current TM or empty.

Then the rules B^* follow from the state transitions of B', where the cell contents and Turing Machine state indicate how the cell contents should change, and the state of it's neighbors and itself indicate which Turing Machine state it should move to next (that is, should the head move from a neighboring state to this one).

We need to do one more thing to B^* to get B. Before we start, we want to verify the format. B will do this by sweeping from from the beginning to the end and back, making sure each symbol is of the appropriate format. This will tell us that the provided y_x^2 and y^3 encode binary inputs. Such a procedure will work if y^1 is correct. Than after that, the simulation of A' will further make sure y^3 actually encodes all 0.

Automata B does this by having a special sweeping state that sweeps from left to right that makes sure it only encounters binary inputs, that it then changes to be activated, makes it all the way to the end, and then sweeps back to the beginning where it turns into the head for the Turing machine B' simulating A'.

For a time t and a space s, the state of cell s in the history of B at time t can be computed by an algorithm C which does the following:

1. If the time t is in the preprocessing part of B before B^* starts, we can just return the bit from y_x directly, if it is after the head, the head if it is on the head, or the bit from y_x activated if it is before.

   If t is not in the time for preprocessing, just subtract the amount of time to do the preprocessing from t and move on.

2. A time t in B is part of the simulation of some step at some time t' in A'. Since the Turing Machine is input oblivious, we know exactly how many operations a step in A' is in B and can calculate t'. Run A up to time t'.

3. Calculate the current cell the Turing machine should be visiting at time t'. We can straightforwardly calculate how long it takes to simulate all the register operations, and then each cell takes the same amount of time.

   (a) If we are in the middle of a register register operation: If s is in a register, simulate B on the registers till time t, and then directly output it. If s is not a register, output that cell's value from the previous time, as nothing happened.

   (b) If we are looking at another cell: If s is a cell before the register's location during this operation, then just return the value of this cell at the time step after this. If s is a cell inside the register, then simulate B on this register and this one bit right up till time t, then output s at time t. Otherwise, output the value of this cell now. It hasn't been modified yet by this step in the RAM algorithm.

This can easily be done since the algorithm is input oblivious, we know exactly how many steps in B one step in A will take. There is a direct, simple way to translate from a state in A to a state in B. And each step from one bit in memory to the next as it seeks the appropriate index takes the same amount of time, so we can skip right to the correct one. Thus we can easily compute if the sought index is before, or after the one being checked and change the state appropriately. □

This gives us a version of the original RAM algorithm with a locally checkable computation history, since cellular automata is a local model of computation. This is essentially the definition of a cellular automata. Remember that we are assuming the states in the cellular automata are in some convenient binary encoding.

Lemma B.0.1 (Cellular Automata have Constant Size Consistency Checks)**.** *For any cellular automata with S bounded cells in memory, for any $i \in [S]$ there is a constant size Boolean function on the states of the $i-1, i$, and $i+1$ cell, and a new proposed value for cell i that outputs whether that would be the new state of cell i after a time step.*

We want to use the multilinear extension of the computation history of the B in Lemma 5.2.1 to get an arithmetization of Lemma B.0.1. As part of this, we need another arithmetization.

Lemma B.0.2 (Successor Arithmetization)**.** *For field $\mathbb{F}$, $l \geq 1$, there is a $O(l\,\mathbf{polylog}(|\mathbb{F}|))$ time algorithm computing the multilinear extension of $u+1=v$ for l bit numbers, u and v.*

Proof. For this proof, we will assume that u and v have their high order bits first, so v_1 is the bit with the largest magnitude, and v_l is the bit with the smallest magnitude. For all $l \geq 1$, define $f_l : \mathbb{F}^l \times \mathbb{F}^l \to \mathbb{F}$ inductively by

1. If $l = 1$, $f_l(u, v) = (1 - u_1) \cdot v_1$,

2. If $l > 1$,
$$f_l(u, v) = (1 - u_l) \cdot v_l \cdot \mathrm{equ}(u_{[l-1]}, v_{[l-1]}) + u_l \cdot (1 - v_l) f_{l-1}(u_{[l-1]}, v_{[l-1]}).$$

Then by induction, each f_l is multilinear and consistent with the check that $v = u + 1$.

We can calculate every equ term together with $O(l)$ field operations, by starting with $\text{equ}(u_1, v_1)$, then multiplying it by $\text{equ}(u_2, v_2)$ to get $\text{equ}(u_{[2]}, v_{[2]})$, and so on. Then using each of these, f can be calculated inductively in a straightforward way using only $O(l)$ field operations. □

Now we can construct the inconsistency function actually used in our **PCP**. The idea is to take 2 times, 3 spaces, and a claimed computation history, and output if the cells at these times and spaces violate Lemma B.0.1. For technical reasons, we will further ask for the states of those spaces at that time in the input, and only do the check if these states agree with the computation history. Of course, we will actually get an arithmetization of such a boolean function.

So now we can prove Lemma 5.2.3.

Lemma 5.2.3 (Inconsistency Function)**.** *Let k be a constant, s, t be integers, and $\mathbb{F}$ be a field. Let B be a cellular automata with 2^{2^k} different states per cell running in $S = 2^s$ cells, and time $T = 2^t$. Then there is a function Γ_B taking any function $X : \mathbb{F}^s \times \mathbb{F}^k \times \mathbb{F}^t \to \mathbb{F}$ and returning a function $Y : \mathbb{F}^{3s+2t+4(2^k)} \to \mathbb{F}$ such that:*

1. *If X is Boolean on Boolean inputs, Y is Boolean on Boolean inputs.*
2. *If X is Boolean on Boolean inputs, then Y is 0 on all Boolean inputs if and only if X on Boolean inputs encodes a valid computation history for B.*
3. *If X is degree d, then Y is degree $O(s + t + d)$. If X is degree d in every variable individually, Y is degree $O(d)$ in every variable individually.*
4. *Given oracle access to X, Y can be computed in time $O((t + s)\mathbf{polylog}(|\mathbb{F}|))$ with a constant number of calls to X.*
5. *If $Y = \Gamma_B(X)$ is 0 on all Boolean inputs, then $\Gamma_B(MLB(X))$ is also 0 on all Boolean inputs.*

Proof. From Lemma B.0.1, there is a function, $\phi : \{0,1\}^{4(2^k)} \to \{0,1\}$, which takes $a_0, a_1, a_2, a_1' \in \{0,1\}^{2^k}$ and outputs $\phi(a_0, a_1, a_2, a_1') = 0$ if in the cellular automata with a_0, a_1, a_2 adjacent, in that order, in the cellular automata at one time, replaces a_1 with a_1' in the next time, and outputs 1 otherwise.

Let $\hat{\phi}$ be the multilinear extension of ϕ. Since ϕ has constant size, $\hat{\phi}$ is a constant size arithmetic expression that can be computed in time $O(\mathbf{polylog}(|\mathbb{F}|))$.

Let $\theta : \{0,1\}^{3s} \times \{0,1\}^{2t} \to \{0,1\}$ be the function that takes $s_0, s_1, s_2 \in \{0,1\}^s$ and $t_0, t_1 \in \{0,1\}^t$ and outputs 1 if $s_0 + 1 = s_1$, $s_1 + 1 = s_2$ and $t_0 + 1 = t_1$, and 0 otherwise. By Lemma B.0.2, in time $O((t + s)\mathbf{polylog}(|\mathbb{F}|))$ we can compute a function $\tilde{\theta} : \mathbb{F}^{3s} \times \mathbb{F}^{2t} \to \mathbb{F}$ that has constant degree in each variable and is consistent with θ on boolean values.

For $s_0, s_1, s_2 \in \mathbb{F}^s, t_0, t_1 \in \mathbb{F}^t, a_0, a_1, a_2, a_1' \in \mathbb{F}^{2^k}$, define Y by:

$$\begin{aligned} Y(s_0, s_1, s_2, t_0, t_1, a_0, a_1, a_2, a_1') =& \tilde{\theta}(s_0, s_1, s_2, t_0, t_1) \cdot \\ & \hat{\phi}(a_0, a_1, a_2, a_1') \cdot \\ & \prod_{j \in \{0,1,2\}} \prod_{i \in \{0,1\}^k} \text{equ}(X(s_j, i, t_0), (a_j)_i) \cdot \\ & \prod_{i \in \{0,1\}^k} \text{equ}(X(s_1, i, t_1), (a_1')_i). \end{aligned}$$

1. If X is binary on binary inputs, Y is binary on binary inputs since it is just a product of functions that are binary on binary inputs.
2. If X is binary on binary inputs, then for binary inputs, Y is 1 if and only if s_0, s_1, s_2 are adjacent states, a_0, a_1, a_2 are the states of s_0, s_1, s_2 at time t_0, a_1' is the state of s_1 at time t_1, and the transition from a_1 to a_1', given neighbors a_0 and a_2 is invalid.

   Thus, if X on binary inputs is a valid computation history, no constraints are ever violated, and Y is 0 on binary inputs. If X is not a valid computation history, it has an improper transition at some point, and at that point, Y would be 1.

3. Y is the product of only constantly many terms (since k is constant), all of which, but potentially X, have degree at most $O(s+t)$, and the X has degree d. Products only add degrees, and we only take constantly many products. So we have degree $O(s+t+d)$.

   For individual variable degree, the input to each of the constantly many X all have degree 1 in distinct variables. So if X has degree d in every variable individually, each call to X is degree at most d in each variable individually. Y is only a product of constantly many calls to X times functions with constant degree. So Y has degree $O(d)$ in each variable individually.

4. Function $\tilde{\theta}$ runs in time $O((s+t)\mathbf{polylog}(|\mathbb{F}|))$, and function $\hat{\phi}$ runs in time $O(\mathbf{polylog}(|\mathbb{F}|))$, and each of the constantly many equ only take $O(\mathbf{polylog}(|\mathbb{F}|))$ time with constantly many calls to X. So we only take $O((s+t)\mathbf{polylog}(|\mathbb{F}|))$ time overall with constantly many oracle calls to X.

5. Suppose Y is 0 on all boolean inputs. Let $Y' = \Gamma_A(\mathrm{MLB}(X))$.

   In particular, suppose for some boolean values for $s_0, s_1, s_2, t_0, t_1, a_0, a_1, a_2$, and a'_1, function Y is 0. This happens if and only if one of the products making up Y is 0.

   $\tilde{\theta} = 0$ or $\tilde{\phi} = 0$: Neither of these terms involve X, so they are still 0 no matter what we switch X to.

   $\mathrm{equ}(X(s_j, i, t_0), (a_j)_i)$ for some i, j: We know $(a_j)_i$ is binary, so by the definition of equ, this expression simplifies to either $X(s_j, i, t_0) = 0$, or $X(s_j, i, t_0) = 1$. In either case, this implies $X(s_j, i, t_0)$ is binary. Thus on this input, $\mathrm{MLB}(X) = X$, and this term is still 0 in $\Gamma_A(\mathrm{MLB}(X))$.

   $\mathrm{equ}(X(s_1, i, t_1), (a'_1)_i)$ for some i: Same as above.

□

C Sum Check Proofs

Here we prove that sum check works: Lemma 5.3.1. For reference, here is how we defined our sum check protocol:

Definition 5.3.2 (Sum Check Protocol Definition)**.** *Let $n, d \in \mathbb{N}$, and $\mathbb{F}$ be a field with $|\mathbb{F}| > \max\{d, n\} + 1$. Suppose $f : \mathbb{F}^n \times \mathbb{F} \to \mathbb{F}$. Then the degree d Sum Check Protocol on f is the following randomized algorithm.*

1. *Get $2n$ random field elements, $R = (r_1, \ldots, r_n$ and $r'_1, \ldots, r'_n)$.*
2. *Reject if $f((r_1, \ldots, r_n), 1) \neq 0$.*
3. *For i from 1 to n:*
   (a) *For $j \in [d+1]$, query*
   $$a_i^j = f((r'_1, \ldots, r'_{i-1}, j, r_{i+1}, \ldots r_n), i+1).$$
   Using these, let $g_i : \mathbb{F} \to \mathbb{F}$ be the degree d polynomial so that for all $j \in [d+1]$, $g_i(j) = a_i^j$.
   (b) *If*
   $$f((r'_1, \ldots, r'_{i-1}, r_i, \ldots r_n), i) \neq (1 - r_i)g_i(0) + r_i g_i(1)$$
   reject.
   (c) *If*
   $$f((r'_1, \ldots, r'_i, r_{i+1}, \ldots r_n), i+1) \neq g_i(r'_i)$$
   reject.
4. *If all checks pass, accept.*

We often refer to the ability of some function $g : \mathbb{F}^n \to \mathbb{F}$ to pass a sum check. The sum check on function f checks whether $g(x) = f(x, n+1)$ on binary inputs is the constant 0 function. It can be useful to refer to the probability of g passing the sum check, assuming the rest of f is defined optimally.

Definition C.0.1 (Passing A Sum Check)**.** *Let $n, d \in \mathbb{N}$, $\mathbb{F}$ be a field with $|\mathbb{F}| > \max\{n, d+1\}$, $\delta \in [0,1]$, and $g : \mathbb{F}^n \to \mathbb{F}$. Then we say g passes the degree d sum check with probability δ if there exists some function $f : \mathbb{F}^n \times \mathbb{F} \to \mathbb{F}$ so that for all $x \in \mathbb{F}$, $g(x) = f(x, n+1)$, and f passes the degree d sum check protocol with probability δ.*

If g is low degree, then the sum check does a good job checking if g is 0 on all binary inputs.

Lemma C.0.2 (Sum Check Of Low Degree Polynomial)**.** *Let $n, d \in \mathbb{N}$, $\mathbb{F}$ be a field with $|\mathbb{F}| > n(d+1)$, $\delta \in [0,1]$, and $g : \mathbb{F}^n \to \mathbb{F}$. Then:*

1. *If for all $x \in \{0,1\}^n$ we have $g(x) = 0$ and the max degree of g in any variable is at most d, then g passes the degree d sum check with probability 1.*
2. *If there exists $x \in \{0,1\}^n$ such that $g(x) \neq 0$ and the max degree of g in any variable is at most d', then g passes the degree d sum check with probability at most $\frac{(d'+1)n}{|\mathbb{F}|}$.*

Proof. First we define f in the format a sum check expects (whether or not the multilinear extension of g actually is 0).

$$\begin{aligned}
f_{n+1}((x_1, \dots, x_n), n+1) =& g(x_1, \dots, x_n) \\
f((x_1, \dots, x_n), i) =& (1 - x_i) f((x_1, \dots, x_{i-1}, 0, x_{i+1}, \dots, x_n), i+1) \\
&+ x_i f((x_1, \dots, x_{i-1}, 1, x_{i+1}, \dots, x_n), i+1) \\
f_i(x) =& f(x, i).
\end{aligned}$$

For $i \notin [n+1]$, how we define $f(x,i)$ is arbitrary since it is never queried. By induction, see that for $n \geq j \geq i \geq 1$, then f_i is linear in variable j. In particular, f_1 is multilinear. Further, each f_i agree on boolean inputs.

1. Suppose for all $x \in \{0,1\}^n$ we have $g(x) = 0$ and the max degree of g in any variable is at most d. Then by induction, for $i \in [n+1]$ and $j \in [n]$, function f_i has degree at most d in variable j.

   Then choose randomness, $R = (r_1, \dots, r_n$ and $r'_1, \dots, r'_n)$. See that f_1 is multilinear, and 0 on all binary inputs, so it must be the 0 function. Thus $f((r_1, \dots, r_n), 1) = 0$.

   For every $i \in [n]$, by definition

   $$\begin{aligned}
   & f((r'_1, \dots, r'_{i-1}, r_i, r_{i+1}, \dots, r_n), i) \\
   =& (1 - r_i) f((r'_1, \dots, r'_{i-1}, 0, r_{i+1}, \dots, r_n), i+1) \\
   &+ r_i f((r'_1, \dots, r'_{i-1}, 1, r_{i+1}, \dots, r_n), i+1).
   \end{aligned}$$

   Since f_{i+1} is degree at most d in variable i, function g_i in the sum check is a degree at most d polynomial, and g_i agrees with f_{i+1} on $d+1$ points, then $f_{i+1} = g_i$ as a function of variable i. So

   $$f((r'_1, \dots, r'_{i-1}, r'_i, r_{i+1}, \dots, r_n), i+1) = g_i(r'_i)$$

   and

   $$f((r'_1, \dots, r'_{i-1}, r_i, r_{i+1}, \dots, r_n), i) = (1 - r_i) g_i(0) + r_i g_i(1).$$

   So all tests pass.

2. Suppose there exists $x \in \{0,1\}^n$ such that $g(x) \neq 0$ and the max degree of g in any variable is at most d'. Then by induction, for $i \in [n+1]$ and $j \in [n]$, function f_i has degree at most d' in variable j. Now take any candidate function, $f' : \mathbb{F}^n \times \mathbb{F} \to \mathbb{F}$, so that $f'(x, n+1) = g(x)$. Define $f'_i(x) = f(x, i)$.

   Our goal is to show that if f'_1 is not equal to f_1, then with low probability will f' be able to change to f on the values we are evaluating without the sum check catching it. Since f' must be f at the last step, due to how we defined them, function f' must fail the sum check with high probability.

   Since $f_1(x) \neq 0$, function f_1 is not the constant 0 function. Since f_1 is multilinear, f_1 has degree at most n. Thus the probability that $f_1(r_1, \dots, r_n) = 0$ is at most $\frac{n}{|\mathbb{F}|}$ by Schwartz-Zippel.

Suppose $f_1(r_1, \ldots, r_n) \neq 0$. Then either $f'_1(r_1, \ldots, r_n) = f_1(r_1, \ldots, r_n)$ or not. If they are equal, sum check will fail. Now we will perform induction.

Suppose for $i \leq n$, with probability at most $\frac{n+d'(i-1)}{|\mathbb{F}|}$ has f' not failed the sum check by step i and

$$f_i(r'_1, \ldots, r'_{i-1}, r_i, r_{i+1}, \ldots, r_n) = f'_i(r'_1, \ldots, r'_{i-1}, r_i, r_{i+1}, \ldots, r_n).$$

So suppose

$$f_i(r'_1, \ldots, r'_{i-1}, r_i, r_{i+1}, \ldots, r_n) \neq f'_i(r'_1, \ldots, r'_{i-1}, r_i, r_{i+1}, \ldots, r_n).$$

Define degree d' function g^*_i and degree d function g'_i so that for $j^* \in [d'+1]$ and $j' \in [d+1]$ we have

$$\begin{aligned} g^*_i(j^*) =& f_{i+1}(r'_1, \ldots, r'_{i-1}, j^*, r_{i+1}, \ldots, r_n) \\ g'_i(j') =& f'_{i+1}(r'_1, \ldots, r'_{i-1}, j', r_{i+1}, \ldots, r_n). \end{aligned}$$

Since f_{i+1} is degree d' in variable i and agrees with g^*_i on $d'+1$ places, for any r'_i we have

$$\begin{aligned} f_{i+1}(r'_1, \ldots, r'_{i-1}, r'_i, r_{i+1}, \ldots, r_n) =& g^*_i(r'_i) \\ f_i(r'_1, \ldots, r'_{i-1}, r_i, r_{i+1}, \ldots, r_n) =& (1-r_1)g^*_i(0) + r_i g^*_i(1). \end{aligned}$$

If $g^*_i = g'_i$, then the sum check fails because

$$f_i(r'_1, \ldots, r'_{i-1}, r_i, r_{i+1}, \ldots, r_n) \neq f'_i(r'_1, \ldots, r'_{i-1}, r_i, r_{i+1}, \ldots, r_n).$$

So suppose $g^*_i \neq g'_i$. By Schwartz-Zippel, the probability $g^*_i(r'_i) = g'_i(r'_i)$ is at most $\frac{d'}{|\mathbb{F}|}$.

So suppose $g^*_i(r'_i) \neq g'_i(r'_i)$. Then if

$$f'_{i+1}(r'_1, \ldots, r'_{i-1}, r'_i, r_{i+1}, \ldots, r_n) \neq g'_i(r'_i)$$

the sum check fails. So suppose they are equal. Then we have

$$\begin{aligned} f'_{i+1}(r'_1, \ldots, r'_{i-1}, r'_i, r_{i+1}, \ldots, r_n) =& g'_i(r'_i) \\ \neq& g^*_i(r'_i) \\ =& f_{i+1}(r'_1, \ldots, r'_{i-1}, r'_i, r_{i+1}, \ldots, r_n). \end{aligned}$$

So by a union bound, the probability that we haven't failed by step $i+1$ and

$$f_{i+1}(r'_1, \ldots, r'_{i-1}, r'_i, r_{i+1}, \ldots, r_n) = f'_{i+1}(r'_1, \ldots, r'_{i-1}, r'_i, r_{i+1}, \ldots, r_n)$$

is at most $\frac{n+d'i}{|\mathbb{F}|}$.

Finally, for $i = n+1$, we know $f'_{n+1} = f_{n+1}$, since they are equal to function g. So with probability at most $\frac{n(d'+1)}{|\mathbb{F}|}$ has the sum check not failed.

□

Now we need a prover to actually carry out this protocol in small space. But this can be done following the expected definition for f in the obvious way.

Lemma C.0.3 (Sum Check Proofs Require Low Space)**.** *Suppose $g : \mathbb{F}^n \to \mathbb{F}$ has degree d in each variable and can be computed in space S and is 0 on Boolean inputs. Then for some $f : \mathbb{F}^n \times \mathbb{F} \to \mathbb{F}$ so that for all $x \in \mathbb{F}$, $g(x) = f(x, n+1)$ and f passes the degree d sum check protocol with probability 1, f can be calculated in space $O(n \log(|\mathbb{F}|) + S)$.*

Further, if g has degree d in variable i, so does f.

Proof. First, we define f inductively in the usual way:

$$\begin{aligned}f((x_1,\ldots,x_n),n+1) =& g(x_1,\ldots,x_n)\\ f((x_1,\ldots,x_n),i) =& (1-x_i)f((x_1,\ldots,x_{i-1},0,x_{i+1},\ldots,x_n),i+1)\\ &+ x_i f((x_1,\ldots,x_{i-1},1,x_{i+1},\ldots,x_n),i+1)\\ f_i(x) = f(x,i).\end{aligned}$$

Now one may observe that we didn't define f for $i \notin [n+1]$. We can just assume they are all 0 for now, this will be addressed in our **ePCP**. Note this is the same f used in Lemma C.0.2, so passes the sum check.

We can then rewrite each f_i in a more convenient format of

$$f_i(x_1,\ldots,x_n) = \sum_{y\in\{0,1\}^{n+1-i}} g(x_1,\ldots,x_{i-1},y_1,\ldots,y_{n+1-i})\cdot \prod_{j\in[n+1-i]} \mathrm{equ}(y_j,x_{i-1+j}).$$

This can be shown to be correct by induction. Then this can be calculated for any f_i in a straightforward way keeping track of a constant number of field elements, and a pointer for y and j, requiring only $O(n\log(|\mathbb{F}|))$ bits. ☐

Given all of these, Lemma 5.3.1 follows.

Lemma 5.3.1 (Sum Check Protocol)**.** *Let $n,d\in\mathbb{N}$, and $\mathbb{F}$ be a field with $|\mathbb{F}| > (d+1)n$. Then there is some protocol, A, so that for any $f:\mathbb{F}^n\times\mathbb{F}\to\mathbb{F}$:*

1. *For some $m = O(nd)$ and $R = 2n$, there is a verifier $V : \mathbb{F}^m \to \{0,1\}$ and query function $Q : \mathbb{F}^R\times[m]\to\mathbb{F}^n\times\mathbb{F}$ so that*

$$A(f,r) = V(f(Q(r,1)),f(Q(r,2)),\ldots,f(Q(r,m))).$$

2. *V runs in time $O(nd\,\mathbf{polylog}(|\mathbb{F}|))$ and space $O(nd\log(|\mathbb{F}|))$.*
3. *For any $r\in\mathbb{F}^R$, for $Q_r(i) = Q(r,i)$, Q_r is time $O(nd\,\mathbf{polylog}(|\mathbb{F}|))$ extrapolatable.*
4. *For any $r\in\mathbb{F}^R$, the last coordinate of Q is always an element of $[n+1]$. That is, for all $i\in[m]$, $Q(r,i)_{n+1}\in[n+1]$*

   Further, the last coordinate of Q is only equal to $n+1$ at most $O(d)$ times.

Completeness *For any $g:\mathbb{F}^n\to\mathbb{F}$ where g has max degree d in any individual variable, if for all $x\in\{0,1\}^n, g(x) = 0$, then there is some $f:\mathbb{F}^n\times\mathbb{F}\to\mathbb{F}$ so that:*

- *For all $x\in\mathbb{F}^n$ we have $g(x) = f(x,n+1)$.*
- *Sum check succeeds on f:*
$$\Pr_r[A(f,r)=1] = 1.$$
- *Function f has degree at most d in each of its first n variables.*
- *If function g is computable in space S, then function f is computable in space $O(n\log(|\mathbb{F}|)+S)$.*

Soundness *for any $g:\mathbb{F}^n\to\mathbb{F}$ where g has max degree d', if there exists $x\in\{0,1\}^n$ such that $g(x)\neq 0$, then for any $f:\mathbb{F}^n\times\mathbb{F}\to\mathbb{F}$ so that for all $x\in\mathbb{F}^n, g(x) = f(x,n+1)$, sum check fails with high probability:*

$$\Pr_r[A(f,r)=1] \le \frac{(d'+1)n}{|\mathbb{F}|}.$$

Proof. The verifier and query function are implicit in Definition 5.3.2. As are the verifier runtime, and where the queries are made. Extrapolatability of the queries is shown in Lemma 5.3.3. Completeness and soundness is shown in Lemma C.0.2. The low space in the completeness case is shown in Lemma C.0.3. ☐

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