Combinatorial PCPs with efficient verifiers[*]

Or Meir[†]

Abstract

The PCP theorem asserts the existence of proofs that can be verified by a verifier that reads only a very small part of the proof. The theorem was originally proved by Arora and Safra (J. ACM 45(1)) and Arora et al. (J. ACM 45(3)) using sophisticated algebraic tools. More than a decade later, Dinur (J. ACM 54(3)) gave a simpler and arguably more intuitive proof using alternative combinatorial techniques.

One disadvantage of Dinur's proof compared to the previous algebraic proof is that it yields less efficient verifiers. In this work, we provide a combinatorial construction of PCPs with verifiers that are as efficient as the ones obtained by the algebraic methods. The result is the first *combinatorial* proof of the PCP theorem for **NEXP** (originally proved by Babai et al., STOC 1991), and a combinatorial construction of super-fast PCPs of Proximity for **NP** (first constructed by Ben-Sasson et al., CCC 2005).

1 Introduction

1.1 Background and Our Results

The PCP theorem [AS98, ALM$^+$98] is one of the major achievements of complexity theory. A PCP (Probabilistically Checkable Proof) is a proof system that allows checking the validity of a claim by reading only a constant number of bits of the proof. The PCP theorem asserts the existence of PCPs of polynomial length for any claim that can be stated as membership in an **NP** language. The theorem has found many applications, most notably in establishing lower bounds for approximation algorithms.

The original proof of the PCP theorem by Arora et al. [AS98, ALM$^+$98] was based on algebraic techniques: Given a claim to be verified, they construct a PCP for the claim by "arithmetizing" the claim, i.e., reducing the claim to a related "algebraic" claim about polynomials over finite fields, and then asking the prover to prove this algebraic claim. Proving the algebraic claim, in turn, requires an arsenal of tools that employ the algebraic structure of polynomials. While those algebraic techniques are very important and useful, it seems somewhat odd that one has to go through algebra in order to prove the PCP theorem, since the theorem itself says nothing about algebra. Furthermore, those techniques seem to give little intuition as to why the theorem holds.

Given this state of affairs, it is an important goal to gain a better understanding of the PCP theorem and the reasons for which it holds. In her seminal paper, Dinur [Din07] has made a big step toward achieving this goal by giving an alternative proof of the PCP theorem using a combinatorial approach. Her proof is not only considerably simpler than the original proof, but also seems to shed more light on the intuitions that underlay the theorem. However, her PCP construction is

[*]This research was partially supported by the Israel Science Foundation (grants No. 460/05 and 1041/08).

[†]Department of Computer Science and Applied Mathematics, Weizmann Institute of Science, Rehovot 76100, Israel. Email: or.meir@weizmann.ac.il

still inferior to the algebraic constructions in two aspects that will be discussed below. We believe that it is important to try to come up with a combinatorial construction of PCPs that matches the algebraic constructions in those aspects, as this will hopefully advance our understanding of the PCP theorem.

Recall that **NP** can be viewed as the class of languages $L \subseteq \{0,1\}^*$ for which one can efficiently verify the claim that $x \in L$ by reading some witness $w \in \{0,1\}^*$ whose length is polynomial in $|x|$. The first aspect in which the algebraic PCPs surpass Dinur's PCP concerns the length of the proofs used by the PCP verifier, and its relation to the length of w. The original PCP theorem asserts that in order to verify that $x \in L$, the PCP verifier needs to use a proof of length $\text{poly}(|x| + |w|)$. However, using algebraic techniques, one can construct PCP verifiers that verify the claim that $x \in L$ using a proof of length only $(|x| + |w|) \cdot \text{poly}\log(|x| + |w|)$ [BSS05]. It is not known whether one can construct such a PCP using a combinatorial approach such as Dinur's[1], and we view it as an important challenge.

The second aspect, which is the focus of this work, concerns the running time of the verification procedure, and its dependence on the proof length. Note that, while in order to verify that $x \in L$ the verifier must run in time which is at least linear in the length of x (since the verifier has to read x), the effect of the proof length on the verifier's running time may be much smaller. Using the algebraic techniques, one can construct PCP verifiers whose running time depends only poly-logarithmically on the proof length. On the other hand, the verifiers obtained from Dinur's proof of the PCP theorem are not as efficient, and depend polynomially on the proof length. While this difference does not matter much in the context of standard PCPs for **NP**, it is very significant in two related settings that we describe below.

PCPs for NEXP While the PCP theorem is most famous for giving PCP systems for languages in **NP**, it can be scaled to higher complexity classes, up to **NEXP**. Informally, the PCP theorem for **NEXP** states that for every language $L \in \mathbf{NEXP}$, the claim that $x \in L$ can be verified by reading a constant number of bits from an exponentially long proof, where the verifier runs in *polynomial time*. Note that in order to meet the requirement that the verifier will run in polynomial time, one needs to make sure the verifier's running time depends only *poly-logarithmically on the proof length*.

The PCP theorem for **NEXP** can be proved by combining the algebraic proof of the PCP theorem for **NP** (of [AS98, ALM$^+$98]) with the ideas of Babai et al. [BFL91]. Dinur's proof, on the other hand, is capable of proving the PCP theorem for **NP**, but falls short of proving the theorem for **NEXP** due to the running time of its verifiers. Our first main result is the first combinatorial proof of the PCP theorem for **NEXP**:

Theorem 1.1 (PCP theorem for **NEXP**, informal)**.** *For every $L \in \mathbf{NEXP}$, there exists a probabilistic polynomial time verifier that verifies claims of the form $x \in L$ by reading only a constant number of bits from a proof of length* $\exp(\text{poly}(|x|))$

Indeed, Theorem 1.1 could already be proved by combining the works of [AS98, ALM$^+$98] and [BFL91], but we provide a combinatorial proof of this theorem.

PCPs of Proximity PCPs of Proximity ([BSGH$^+$06, DR06]) are a variant of PCPs that allows a super-fast verification of claims while compromising on their accuracy. Let $L \in \mathbf{NP}$ and suppose we wish to verify the claim that $x \in L$. Furthermore, suppose that we are willing to compromise on the accuracy of the claim, in the sense that we are willing to settle with verifying that x is *close* to

[1]We mention that the construction of PCPs that have proof length $T(n) \cdot \text{poly}\log(T(n))$ uses Dinur's combinatorial techniques in addition to the algebraic techniques. Still, the main part of this construction is algebraic.

some string in L. PCPs of Proximity (abbreviated PCPPs) are proofs that allow verifying that x is *close* to L by reading only a constant number of bits from *both* x and the proof. Using the algebraic methods, one can construct PCPPs with verifiers that run in time which is *poly-logarithmic in* $|x|$ (see, e.g., [BSGH^{+}05]). Note that this is highly non-trivial even for languages L that are in $\mathbf{P}$.

One can also construct PCPPs using Dinur's techniques, but the resulting verifiers are not as efficient, and run in time $\text{poly}(|x|)$. While those verifiers still have the property that they only read a constant number of bits from x, they seem to lose much of their intuitive appeal. Our second main result is a combinatorial construction of PCPPs that allow super-fast verification:

Theorem 1.2 (PCPPs with super-fast verifiers, informal). *For every $L \in \mathbf{NP}$, there exists a probabilistic verifier that verifies claims of the form "x is close to L" by reading only a constant number of bits from x and from a proof of length* $\text{poly}(|x|)$*, and that runs in time* $\text{poly}(\log |x|)$.

Again, Theorem 1.2 could already be proved by combining the works of [AS98, ALM^{+}98] and [BFL91], but we provide a combinatorial proof of this theorem.

1.2 Our Techniques

Our techniques employ ideas from both the works of Dinur [Din07] and Dinur and Reingold [DR06]. In this section we review these works and describe the new aspects of our work. For convenience, we focus on the construction of super-fast PCPPs (Theorem 1.2).

1.2.1 On Dinur's proof of the PCP theorem

We begin by taking a more detailed look at Dinur's proof of the PCP theorem, and specifically at her construction of PCPP verifiers. The crux of Dinur's construction is a combinatorial amplification technique for increasing the probability of PCPP verifiers to reject false claims. Specifically, given a PCPP verifier that uses a proof of length ℓ, and that rejects false claims with probability ε, the amplification transforms the verifier into a new verifier that rejects false claims with probability $2 \cdot \varepsilon$, but needs to be given a proof of length $\beta \cdot \ell$ for some constant $\beta > 1$.

Using the amplification technique, we can construct PCPP verifiers that use proofs of polynomial length as follows. Let $L \in \mathbf{NP}$. Then, L has a trivial PCPP verifier that uses proofs of length $\text{poly}(n)$ and has rejection probability $\frac{1}{\text{poly}(n)}$ - for example, consider the verifier that reduces L to the problem of 3SAT, then asks to be given as a proof the satisfying assignment of the corresponding formula, and then verifies that the assignment satisfies a random clause. Next, we apply the amplification to the trivial verifier iteratively, until we obtain a PCPP verifier that rejects false claims with constant probability (which does not depend on n). Clearly, the number of iterations required is $O(\log n)$, and therefore the final PCPP verifier uses proofs of length $\beta^{O(\log n)} \cdot \text{poly}(n) = \text{poly}(n)$, as required.

As we mentioned before, this proof yields PCPP verifiers that run in time $\text{poly}(n)$, while we would have wanted our verifiers to be super-fast, i.e., run in time $\text{poly}(\log n)$. The reason for the inefficiency of Dinur's PCPP verifiers is that the amplification technique increases the running time of the verifier to which it is applied by at least a constant factor. Since the amplification is applied for $O(\log n)$ iterations, the resulting blow-up in the running time is at least $\text{poly}(n)$.

1.2.2 On Dinur and Reingold's construction of PCPPs

In order to give a construction of PCPPs with super-fast verifiers, we consider another combinatorial construction of PCPPs, which was proposed by Dinur and Reingold [DR06] prior to Dinur's proof of the PCP theorem. We refer to this construction as the "DR construction". Like Dinur's construction, the DR construction is an iterative construction. However, unlike Dinur's construction, the DR construction uses only $O(\log \log n)$ iterations. This means that if their construction can be implemented in a way such that each iteration incurs a linear blow-up to the running time of the verifiers, then the final verifiers will run in time $\text{poly} \log n$ as we desire. *Our first main technical contribution is showing that such an implementation is indeed possible.* In order to do so, we revisit several known techniques from the PCP literature and show that they have super-fast implementations.

However, the DR construction has a significant shortcoming: its verifiers use proofs that are too long; specifically, this construction uses proofs of length $n^{\text{poly} \log n}$. *Our second main technical contribution is showing how to modify the DR construction so as to have proofs of length* $\text{poly}(n)$ *while maintaining the high efficiency of the verifiers.*

1.2.3 Our construction

Following Dinur and Reingold, it is more convenient to describe our construction in terms of "assignment testers" (ATs). Assignment testers are PCPPs that allow verifying that an assignment is close to a satisfying assignment of a given circuit. Any construction of ATs yields a construction of PCPs and PCPPs, and therefore our goal is to construct ATs whose running time is poly-logarithmic in the size of the given circuit.

The crux of the DR construction is a reduction that transforms an AT that acts on circuits of size k to an AT that acts on circuits of size k^c (for some constant $c > 0$). Using such a reduction, it is possible to construct an AT that works on circuits of size n by starting from an AT that works on circuits of constant size and applying the reduction for $O(\log \log n)$ times. However, the DR reduction also increases the proof length from ℓ to $\ell^{c'}$ (for some constant $c' > c$), which causes the final ATs to have proof length $n^{\text{poly} \log n}$. Moreover, the reduction runs in time that is polynomial in the given circuit, rather than poly-logarithmic. We turn to discuss the issues of improving the proof length and improving the running time separately, but of course, in the actual construction these aspects need to be combined.

The proof length A close examination of the DR reduction shows that its superfluous blow-up stems from two sources. The first source is the use of a "parallel repetition"-like error reduction technique, which yields a polynomial blow-up to the proof length. This blow-up can be easily reduced by using the more efficient amplification technique from Dinur's work.

The second source of the blow-up is the use of a particular circuit decomposition technique. The DR reduction uses a procedure that decomposes a circuit into an "equivalent" set of smaller circuits. This part of the reduction yields a blow-up that is determined by the parameters of the decomposition. The DR reduction uses a straightforward method of decomposition that incurs a polynomial blow-up. In our work, we present an alternative decomposition method that is based on packet-routing ideas and incurs a blow-up of only a poly-logarithmic factor, as required. We mention that in order to analyze our circuit decomposition method we prove a lemma on sorting networks, which may be of independent interest.

The running time For the rest of the discussion, it would be convenient to view the DR reduction as constructing a "big" AT that acts on "big" circuits from a "small" AT that acts on "small" circuits. The big AT works roughly by decomposing the given circuit to an equivalent set of smaller circuits, invoking the small AT on each of the smaller circuits, and combining the resulting residual tests in a sophisticated way. However, if we wish the big AT to run in time which is linear in the running time of the small AT, we can not afford invoking the small AT on each of the smaller circuits since the

the number of those circuits is super-constant. We therefore modify the reduction so that it does not invoke the small AT on each of the smaller circuits, but rather invoke it once on the "universal circuit", and use this single invocation for testing the smaller circuits. When designed carefully, the modified reduction behaves like the original reduction, but has the desired poly-logarithmic running time.

In addition to the foregoing issue, we must also show that our decomposition method and Dinur's amplification technique have sufficiently efficient implementations. This is easily done for the decomposition. However, implementing Dinur's error reduction is non-trivial, and can be done only for PCPs that possess a certain property. The efficient implementation of Dinur's error reduction method is an additional contribution of our work.

Organization of the paper In Section 2 we cover the relevant background for our work and give a formal statement of our main results. In Section 3 we give a high-level overview of our work.

2 Preliminaries and Our Main Results

2.1 Notational Conventions

For any $n \in \mathbb{N}$, we denote $[n] \overset{\text{def}}{=} \{1, \ldots, n\}$. For any $S \subseteq [n]$ and $x \in \{0,1\}^n$, we denote by $x_{|S}$ the projection of x to S. That is, if $S = \{i_1, \ldots, i_s\}$ for $i_1 < \ldots < i_s$, then $x_{|S} = x_{i_1} \ldots x_{i_s}$

For any two strings $x, y \in \{0,1\}^n$, we denote by $\delta(x, y)$ the relative Hamming distance between x and y, i.e., $\delta(x, y) \overset{\text{def}}{=} \Pr_{i \in [n]}[x_i \neq y_i]$. For any string $x \in \{0,1\}^*$ and a set $S \subseteq \{0,1\}^*$, we denote by $\delta(x, S)$ the relative Hamming distance between x and the nearest string of length $|x|$ in S, and we use the convention that $\delta(x, S) = 1$ if no string of length $|x|$ exists in S. In particular, we define $\delta(x, \emptyset) = 1$.

Whenever we discuss circuits in this paper, we always refer to circuits that have fan-in and fan-out at most 2, unless stated explicitly otherwise. For any circuit φ, we denote by $\mathrm{SAT}(\varphi)$ the set of satisfying assignments of φ.

2.2 PCPs

As discussed in the introduction, the focus of this work is on proofs that can be verified by reading a small number of bits of the proof *while running in a short time*. While we are also interested in having short proofs, it is usually more convenient to upper bound the number of coin tosses used by the verifier, and note that this number can be used to bound the length of the proofs (see Remark 2.2 below). The following definition captures the notion of verifiers we are interested in.

Definition 2.1. Let $r, q, t : \mathbb{N} \to \mathbb{N}$. A (r, q, t)-PCP verifier V is a probabilistic oracle machine that when given input $x \in \{0,1\}^*$, runs for at most $t(|x|)$ steps, tosses at most $r(|x|)$ coins, makes at most $q(|x|)$ *non-adaptive* queries to its oracle, and outputs either "accept" or "reject". We refer to r, q, and t, as the randomness complexity, query complexity and time complexity of the verifier respectively.

Remark 2.2. Note that for an (r, q, t)-PCP verifier V and an input x, we can assume without loss of generality that the oracle is a string of length at most $2^{r(|x|)} q(|x|)$, since this is the maximal number of different queries that V can make.

Now that we have defined the verifiers, we can define the languages for which membership can be verified.

Definition 2.3. Let $r, q, t : \mathbb{N} \to \mathbb{N}$, let $L \subseteq \{0,1\}^*$ and let $\varepsilon \in (0,1]$. We say that $L \in \mathbf{PCP}_\varepsilon[r,q,t]$ if there exists an (r,q,t)-PCP verifier V that satisfies the following requirements:

- **Completeness:** For every $x \in L$, there exists $\pi \in \{0,1\}^*$ such that $\Pr[V^\pi(x) \text{ accepts}] = 1$.
- **Soundness:** For every $x \notin L$ and for every $\pi \in \{0,1\}^*$ it holds that $\Pr[V^\pi(x) \text{ rejects}] \geq \varepsilon$.

Remark 2.4. Note that Definition 2.3 specifies the rejection probability, i.e., the probability of false claims to be rejected. We warn that it is more common in PCP literature to specify the error probability, i.e., the probability that false claims are *accepted*.

Remark 2.5. Note that for any two constants $0 < \varepsilon_1 < \varepsilon_2 < 1$, it holds that $\mathbf{PCP}_{\varepsilon_1}[r,q,t] = \mathbf{PCP}_{\varepsilon_2}[O(r), O(q), O(t)]$ by a standard amplification argument. Thus, as long as we do not view constant factors as significant, we can ignore the exact constant ε and refer to the class $\mathbf{PCP}[r,q,t]$.

Remark 2.6. The standard notation usually omits the running time t, and refers to the class $\mathbf{PCP}[r,q]$, which equals $\mathbf{PCP}[r,q,\mathrm{poly}(n)]$.

The PCP theorem is usually stated as $\mathbf{NP} \subseteq \mathbf{PCP}[O(\log n), O(1)]$, but in fact, the original proof of [AS98, ALM$^+$98], when combined with earlier ideas of [BFL91], actually establishes something stronger. In order to state the full theorem, let us say that a function $f : \mathbb{N} \to \mathbb{N}$ is admissible if it can be computed in time $\mathrm{poly}\log f(n)$ (this definition can be extended for functions f of many variables).

Theorem 2.7 (implicit in the PCP theorem of [BFL91, AS98, ALM$^+$98])**.** *For any admissible function $T(n) = \Omega(n)$, it holds that*

$$\mathbf{NTIME}(T(n)) \subseteq \mathbf{PCP}[O(\log T(n)), O(1), \mathrm{poly}(n, \log T(n))]$$

In her paper [Din07], Dinur presented a combinatorial proof of the PCP theorem. However, while her proof matches the randomness and query complexity of Theorem 2.7, it only yields a weaker guarantee on the time complexity of the verifier:

Theorem 2.8 (Implicit in Dinur's PCP theorem [Din07])**.** *For any admissible function $T(n) = \Omega(n)$, it holds that*

$$\mathbf{NTIME}(T(n)) \subseteq \mathbf{PCP}[r(n) = O(\log T(n)), q(n) = O(1), t(n) = \mathrm{poly}T(n)]$$

Note that due to the difference in the time complexity, Theorem 2.7 implies Theorem 1.1 (PCP theorem for **NEXP**) as a special case for $T(n) = \exp(\mathrm{poly}(n))$, while Theorem 2.8 does not. *One contribution of this work is a combinatorial proof for Theorem 2.7.* This proof is actually an immediate corollary of our proof of Theorem 2.15 to be discussed in the next section.

2.3 PCPs of Proximity

2.3.1 The definition of PCPPs

We turn to formally define the notion of PCPs of Proximity (PCPPs). We use a definition that is more general than the one discussed in Section 1. In Section 1, we have described PCPPs as verifiers that verify the claim $x \in L$ for some language $L \in \mathbf{NP}$ by reading a constant number of bits from x and from an additional proof. For example, if L is the language of graphs with a clique of size $\frac{n}{4}$ (where n is the number of vertices in the graph), one can use a PCPP verifier to verify that x represents such a graph. We can consider a more general example, in which we wish to verify that x is a graph of n vertices that contains a clique of size m, where m is a parameter given as an input to the verifier. In such a case, we would still want the PCPP verifier to read only a constant number of bits of x, but we would want to allow the verifier to read all of m, which may be represented using $\log n$ bits. It therefore makes sense to think of PCPP verifiers that are given two inputs:

1. An *explicit input* that is given on their input tape, and which they are allowed to read entirely.
2. An *implicit input* to which they are given oracle access, and of which they are only allowed to read a constant number of bits.

In order to define the languages that such verifiers accept, we need to consider languages of pairs (w, x), where w is the explicit input and x is the implicit input. This motivates the following definition:

Definition 2.9. A pair-language is a relation $L \subseteq \{0,1\}^* \times \{0,1\}^*$. For every $x \in \{0,1\}^*$, we denote $L(w) \overset{\text{def}}{=} \{x : (w, x) \in L\}$.

Using Definition 2.9, we can describe the task of PCPP verifiers as follows: Given $w, x \in \{0,1\}^*$, verify that x is close to $L(w)$ by reading all of w and a constant number of bits from x and from an additional proof. For super-fast verifiers, we would also require a running time of $\text{poly}(|w|, \log |x|)$. In the foregoing example of cliques of size k, the explicit input w will be of the form (n, m) and $L(w)$ will be the set of all graphs of n vertices that contain a clique of size m. Note that in this example w can be represented using $O(\log n)$ bits, so super-fast verifiers will indeed run in time poly-logarithmic in the size of the graph.

PCPs of Proximity were defined independently by Ben-Sasson et al. [BSGH$^+$06] and by Dinur and Reingold [DR06][2], where the latter used the term "Assignment Testers". The question of the efficiency of the verifiers is more appealing when viewed using the definition of [BSGH$^+$06], and therefore we chose to present the foregoing intuitive description of PCPPs in the spirit of [BSGH$^+$06]. Below, we present the definition of PCPPs of [BSGH$^+$05], which is in the spirit of [BSGH$^+$06] but better suits our purposes. We then state our results according to this definition.

Definition 2.10 (PCPP verifier, following [BSGH$^+$05])**.** Let $r, q : \mathbb{N} \to \mathbb{N}$ and let $t : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$. An (r, q, t)-PCPP verifier is a probabilistic oracle machine that has access to two oracles, and acts has follows:

1. The machine expects to be given as an explicit input a pair (w, m), where $w \in \{0,1\}^*$ and $m \in \mathbb{N}$. The machine also expects to be given access to a string $x \in \{0,1\}^m$ in the first oracle as well as to a string $\pi \in \{0,1\}^*$ in the second oracle.

2. The machine runs for at most $t(|w|, m)$ steps, tosses at most $r(|w| + m)$ coins and makes at most $q(|w| + m)$ queries to both its oracles non-adaptively.

3. Finally, the machine outputs either "accept" or "reject".

We refer to r, q and t as the randomness complexity, query complexity and time complexity of the verifier respectively. Note that $t(n, m)$ depends both on $|w|$ and on $|x|$, while $r(n)$ and $q(n)$ depend only on their sum. The reason is that we want the time complexity to depend differently on $|w|$ and on $|x|$ (e.g., to depend polynomially on $|w|$ and poly-logarithmically on $|x|$).

For a PCPP verifier V and strings $w, x, \pi \in \{0,1\}^*$, we denote by $V^{x,\pi}(w)$ the output of V when given $(w, |x|)$ as explicit input, x as the first oracle and π as the second oracle. That is, $V^{x,\pi}(w)$ is a short hand for $V^{x,\pi}(w, |x|)$.

Definition 2.11 (PCPP)**.** Let $L \subseteq \{0,1\}^* \times \{0,1\}^*$ be a pair-language and let $\varepsilon > 0$. We say that $L \in \mathbf{PCPP}_\varepsilon[r(n), q(n), t(n, m)]$ if there exists an $(r(n), q(n), t(n, m))$-PCPP verifier V that satisfies the following requirements:

[2]We mention that PCPs of Proximity are related to the previous notion holographic proofs of [BFLS91] and to the work of [Sze99], see [BSGH$^+$06] for further discussion.

- **Completeness:** For every $(w, x) \in L$, there exists $\pi \in \{0,1\}^*$ such that $\Pr\left[V^{x,\pi}(w) \text{ accepts}\right] = 1$.
- **Soundness:** For every $w, x \in \{0,1\}^*$ and for every $\pi \in \{0,1\}^*$ it holds that $\Pr\left[V^{x,\pi}(w) \text{ rejects}\right] \geq \varepsilon \cdot \delta\left(x, L(w)\right)$.

Similarly to the PCP case, when we do not view constant factors as significant we will just refer to the class $\mathbf{PCPP}\left[r, q, t\right]$, since for every $0 < \varepsilon_1 < \varepsilon_2$ it holds that $\mathbf{PCPP}_{\varepsilon_1}\left[r, q, t\right] = \mathbf{PCPP}_{\varepsilon_2}\left[O(r), O(q), O(t)\right]$.

We turn to discuss two important features of Definition 2.11.

The soundness requirement Note that the probability that V rejects a pair (w, x) depends on the distance of x to $L(w)$. The reason is that if V is given access to some x that is very close to $x' \in L(w)$, then since V may read only a very small part of x, the probability that it queries a bit on which x and x' differ may very small.
We mention that the requirement that the rejection probability would be *proportional to* $\delta\left(x, L(w)\right)$ is a fairly strong requirement, and that PCPPs that satisfy this requirement are sometimes referred to in the literature as "Strong PCPPs". One may also consider weaker soundness requirements. However, we use the stronger requirement since we can meet it, and since it is very convenient to work with.

PCPPs versus PCPs The following corollary shows that PCPPs are in some sense a generalization of PCPs:

Corollary 2.12 ([BSGH$^+$06]). *Let $PL \in \mathbf{PCPP}_{\varepsilon}\left[r(n), q(n), t(n, m)\right]$ be a pair-language, let $p : \mathbb{N} \to \mathbb{N}$ be such that for every $(w, x) \in L$ it holds that $|x| \leq p(n)$, and define a language $L \overset{\text{def}}{=} \{w : \exists x \text{ s.t. } (w, x) \in PL\}$. Then it holds that $L' \in \mathbf{PCP}_{\varepsilon}\left[r(n), q(n), t\left(n, p(n)\right)\right]$.*

Proof Let V be the PCPP verifier for PL. We construct a verifier V' for L as follows. For any $w \in L$, a proof π' that convinces V' to accept w will consist of a string x such that $(w, x) \in L$ and a proof π that convinces V to accept (w, x). We now construct V' in the straightforward way. The analysis of V' is trivial. ■

Remark 2.13. The proof of Corollary 2.12 is based on a different perspective on PCPPs than the one we used throughout this section. So far we have treated the implicit input x as a claim to be verified, and the explicit input w as auxiliary parameters. However, one can also view w as the claim to be verified and x as the witness for the claim w. In this perspective, the role of the PCPP verifier is to verify that x is close to being a valid witness for the claim w. While this perspective is often more useful for working with PCPPs, we feel that it is less appealing as a motivation for the study of PCPPs, and therefore did not use it in our presentation.

2.3.2 Constructions of PCPPs and our results

The following notation is useful for stating the constructions of PCPPs.

Notation 2.14. Let $T : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ let $PL \subseteq \{0,1\}^* \times \{0,1\}^*$ be a pair-language. We say that L is decidable in time T if there exists a Turing machine M such that when M us given as input a pair (w, x), the machine M runs in time at most $T\left(|x|, |w|\right)$, and accepts if and only if $(w, x) \in PL$, Although PCPPs were only defined after the works of [AS98, ALM$^+$98], their algebraic techniques can be modified and combined with earlier ideas from [BFL91] to yield the following PCPPs[3] :

Theorem 2.15 (PCPPs that can be obtained from [BFL91, AS98, ALM$^+$98])**.** *For any admissible function $T(n,m)$ and a pair-language PL that is decidable in time T it holds that*

$$PL \in \mathbf{PCPP}\left[O\left(\log T(n,k)\right), O(1), \mathrm{poly}\left(n, \log T(n,m)\right)\right]$$

In her work [Din07], Dinur has also given a construction of PCPPs. Her focus in this part of the work was giving PCPPs with short proofs, and in order to construct them she combined her combinatorial techniques with the previous algebraic techniques (i.e., [BSS05]). However, one can also obtain the following PCPPs using her combinatorial techniques alone:

Theorem 2.16 (PCPPs that can be obtained from [Din07])**.** *For any admissible function $T(n,m)$ and a pair-language PL that is decidable in time T it holds that*

$$PL \in \mathbf{PCPP}\left[O\left(\log T(n,m)\right), O(1), \mathrm{poly} T(n,m)\right]$$

Again, note that due to the difference in the time complexity, Theorem 2.15 implies Theorem 1.2 as a special case for $T(n) = \mathrm{poly}\,(n)$, while Theorem 2.16 does not. *Our main result is a combinatorial proof of Theorem 2.15.* Observe that Theorem 2.15 implies Theorem 2.7 using Corollary 2.12. We thus focus on proving Theorem 2.15.

We conclude the discussion in PCPPs by showing that Theorem 2.15 indeed implies a formal version of Theorem 1.2.

Corollary 2.17 (Special case of [BSGH$^+$06])**.** *Let $L \in \mathbf{NP}$, and let PL be the pair-language $\{(\lambda, x) : x \in L\}$ (where λ is the empty string). Then $PL \in \mathbf{PCPP}[O(\log m), O(1), \mathrm{poly}\log m]$ (note that here m denotes the length of x).*

Proof sketch The main difficulty in proving Corollary 2.17 is that PL may not be decidable in polynomial time (unless $\mathbf{P} = \mathbf{NP}$), and therefore we can not use Theorem 2.15 directly. The naive solution would be to use Theorem 2.15 to construct a PCPP verifier V_1 for the *efficiently decidable* pair-language

$$PL_1 = \{(\lambda, x \circ y) : y \text{ is a valid witness for the claim that } x \in L\}$$

and then construct a verifier V for PL by asking the prover to provide a witness y in the proof oracle and emulating the action of V_1 on x and y. The problem is that is x is significantly shorter than y, it might be the case that x is far from L and yet $x \circ y$ is close to $PL_1(\lambda)$. Instead, we use Theorem 2.15 to construct a PCPP verifier V_2 for the *efficiently decidable* pair-language

$$PL_2 = \left\{\left(\lambda, \underbrace{x \circ x \circ \ldots \circ x}_{|y|/|x|} \circ y\right) : y \text{ is a valid witness for the claim that } x \in L\right\}$$

and then construct a verifier V for PL by emulating V_2 as before. For more details, see [BSGH$^+$06, Proposition 2.5]. ■

Remark 2.18. We have stated Theorems 2.15 and 2.16 only for pair-languages that are decidable in *deterministic* time, but in fact, one can use them to construct PCPPs for pair-languages that are decidable in *non-deterministic* time, using the same proof idea of Corollary 2.17. For details, see [BSGH$^+$06, Proposition 2.5].

[3]See discussion in [BSGH$^+$06, Sections 1.3 and 2.2] and in [BSGH$^+$05]. We mention that [BSGH$^+$05] was the first time super-fast PCPPs were constructed explicitly. The main improvement of [BSGH$^+$05] over [AS98, ALM$^+$98] is in the proof length, which is not the focus of the current work.

3 Overview

In this section we give a high-level overview of our construction of PCPPs (which, in turn, implies the construction of PCPs). For the most of this overview we focus on describing the construction itself while ignoring the issues of efficient implementation. Then, in Section 3.4, we describe how this construction can be implemented efficiently.

3.1 The structure of the construction

Our construction and the DR construction share a similar structure. In this section we describe this structure and discuss the differences between the constructions.

3.1.1 Assignment testers

Assignment Testers are an equivalent[4] variant of PCPPs that was introduced by [DR06]. Both our construction and the DR construction can be described more conveniently as constructions of assignment testers rather than constructions of PCPP. We therefore start by describing the notion of assignment testers.

An assignment tester is an algorithm that is defined as follows. The assignment tester takes as an input a circuit φ of size n over a set X of Boolean variables. The output of the assignment tester is a collection of circuits $\psi_1, \ldots, \psi_R$ of size $s \ll n$ whose inputs come from the set $X \cup Y$, where Y is a set of auxiliary variables. The circuits $\psi_1, \ldots, \psi_R$ should satisfy the following requirements:

1. For any assignment x to X that satisfies φ, there exists an assignment y to Y such that the assignment $x \circ y$ satisfies all the circuits $\psi_1, \ldots, \psi_R$.
2. For any assignment x to X that is far (in Hamming distance) from any satisfying assignment to φ, and every assignment y to Y, the assignment $x \circ y$ violates at least ε fraction of the circuits $\psi_1, \ldots, \psi_R$.

There is a direct correspondence between assignment testers and PCPPs: An assignment tester can be thought as a PCPP that checks the claim that x is a satisfying assignment to φ. The auxiliary variables Y correspond to the proof string of the PCPP, and the circuits $\psi_1, \ldots, \psi_R$ correspond to the various tests that the verifier performs on the R possible outcomes of its random coins. In particular, the query complexity and the proof length of the PCPP can be upper bounded by s and $|Y| \leq R \cdot s$ respectively. Furthermore, note that the fraction ε corresponds to the rejection probability of the PCPP, and we therefore refer to ε as the rejection probability of the assignment tester. With a slight abuse of notation, we will also say that the circuits $\psi_1, \ldots, \psi_R$ have rejection probability ε.

Our main technical result is a combinatorial construction of an assignment tester that has $R(n) = \text{poly}(n)$, $s(n) = O(1)$ and that runs in time $\text{poly} \log n$. Note that this is impossible when using the foregoing definition of assignment testers, since the assignment tester needs time of at least $\max \{n, R \cdot s\}$ only to read the input and to write the output. However, in this overview we ignore this problem, and in the actual proof we work with a definition of assignment testers that uses implicit representations of the input and the output (see discussion in Section 3.4).

3.1.2 The iterative structure

Our construction and the DR construction are iterative constructions: The starting point of those constructions is an assignment tester for circuits of constant size, which is trivial to construct.

[4]The equivalence can be established via the ideas introduced in [BSGH$^+$05, Sec. 7].

Then, in each iteration, those constructions start from an assignment tester for circuits of size[5] at most k, and use it to construct an assignment tester for circuits of size $\approx k^{c_0}$ (for some constant $c_0 > 1$). Thus, if we wish to construct an assignment tester for circuits of size n, we use $O(\log \log n)$ iterations. As discussed in Section 1.2.2, one major ingredient of this work is showing a construction that can be implemented such that each iteration increases the running time of the assignment tester by a a constant factor. Therefore, the final assignment testers run in time poly $\log n$.

The key difference between our construction and the DR construction is in the effect of a single iteration on the number of output circuits R. In the DR construction, each iteration increases the number of output circuits from R to $R^{c'}$ for some constant $c' > c_0$ (where c_0 is the foregoing constant). Thus, after $O(\log \log n)$ iterations the final assignment testers have $n^{\mathrm{poly}\log n}$ output circuits, which in turn implies a PCPP that uses proofs of length $n^{\mathrm{poly}\log n}$. In contrast, in our construction a single iteration increases the number of output circuits from R to $R^{c_0}\mathrm{poly}\log R$, and thus the final assignment testers have $\mathrm{poly}(n)$ output circuits, as desired.

3.1.3 The structure of a single iteration

We proceed to describe the structure of a single iteration. For the purpose of this description, it is convenient to assume that we wish to construct an assignment tester for circuits of size n using an assignment tester for circuits of size n^{γ} for some constant $\gamma < 1$ (we take $\gamma = 1/c_0$, where c_0 us the constant from Section 3.1.2).

Circuit decompositions In the following description, we use a notion that we call "circuit decomposition". A circuit decomposition is an algorithm that takes as input a circuit φ over Boolean variables X and "decomposes" it to set of smaller circuits $\psi_1, \ldots, \psi_m$ over Boolean variables $X \cup Y$, such that an assignment x to X satisfies φ if and only if there exists an assignment y to Y such that $x \circ y$ satisfies all the circuits ψ_i. Note that a circuit decomposition can be viewed as an assignment tester with the trivial rejection probability $\frac{1}{m}$. Alternatively, a circuit decomposition can be viewed as a generalization of the Cook-Levin reduction that transforms a circuit into a 3SAT formula.

The general structure of an iteration We begin with a general description of an iteration that fits both our construction and the DR construction. Suppose that we wish to construct an assignment tester A for circuits of size n, and assume that we already have a "small" assignment tester A_S that can take as input any circuit of size $n' \leq n^{\gamma}$, and outputs $R(n')$ output circuits of constant size. Let ε denote the rejection probability of A_S . When given as input a circuit φ of size n over a set of Boolean variables X, the assignment tester A proceeds in three main steps:

1. The assignment tester A applies to φ a circuit decomposition D (to be specified later), resulting in a set of circuits $\psi_1, \ldots, \psi_{m(n)}$ of size $s(n)$ for some $m(n), s(n) < n^{\gamma}$ (to be specified later). The decomposition D is required to have an additional property that is needed for the next step, and we discuss this property in Section 3.2.
2. The assignment tester A combines the circuits $\psi_1, \ldots, \psi_{m(n)}$ with the assignment tester A_s in some sophisticated way that is somewhat similar to the tensor product of error-correcting codes (see Section 3.3 for details). The result of this operation is $R' = R(O(m(n))) \cdot R(s(n))$ circuits $\tau_1, \ldots \tau_{R'}$ over variables $X \cup Y \cup Z$, that have rejection probability ε^2.
3. The assignment tester A applies an error-reduction transformation (to be specified later) to the circuits $\tau_1, \ldots, \tau_{R'}$ obtained in Step 2 in order to increase their rejection probability back to ε, and outputs the resulting circuits.

[5]I.e., an assignment tester that can only take as in input a circuit of size at most k

Our construction versus the DR construction Our construction differs from the DR iteration in the circuit decomposition used in Step 1 and in the error-reduction transformation used in Step 3. We begin by discussing the latter. The DR construction uses a variant of the parallel repetition technique in order to do the error reduction. This technique incurs a polynomial blow-up in the number of output circuits of A, while in order to have the desired number of output circuits we can only afford a constant factor blow-up.

In our construction, we replace the parallel repetition technique with Dinur's amplification technique (outlined in Section 1.2.1), which only incurs a constant factor blow-up. This is indeed a fairly simple modification, and the reason that Dinur's technique was not used in the original DR construction is that it did not exist at that time. However, we note that in order to use Dinur's amplification in our context, we need to show that it can be implemented in a super-fast way, which was not proved in the original work of Dinur [Din07] (see further discussion in Section 3.4).

We turn to discuss the circuit decomposition used in Step 1. Recall that Step 2 generates a set of $R(O(m(n))) \cdot R(s(n))$ circuits. Thus, the choice of the functions $m(n)$ and $s(n)$ is crucial to the number of output circuits of A. In particular, it can be verified that the recurrence relation $R(n) = R(O(m(n))) \cdot R(s(n))$ can be solved to a polynomial only if the product $m(n) \cdot s(n)$ is upper bounded by approximately n. However, since the decomposition used in Step 1 must satisfy an additional property that is needed for Step 2, it is not trivial to find a decomposition for a good choice of $m(n)$ and $s(n)$.

The original DR construction uses a straightforward decomposition method that decomposes a circuit of size n into $m(n) = O(n^{3\alpha})$ circuits of size $s(n) = O(n^{1-\alpha})$, where α is a constant arbitrarily close to 0. Thus, $m(n) \cdot s(n) = O(n^{1+2\alpha})$, which causes the final assignment testers of the DR construction to have $n^{\text{poly}\log n}$ output circuits. Our technical contribution in this regard is devising an alternative decomposition method that decomposes a circuit of size n into $m(n) = \sqrt{n}\text{poly}\log n$ circuits of size $s(n) = \sqrt{n}\text{poly}\log n$. Thus, $m(n) \cdot s(n) = n\text{poly}\log n$, which is good enough to make the whole construction have a polynomial number of output circuits.

3.2 Our circuit decomposition method

In this section we describe the circuit decomposition we use in Step 1 (of Section 3.1.3). We begin by describing the additional property that the decomposition is required to have in order to be useful for the construction. Recall that a circuit decomposition takes as an input a circuit φ of size n over Boolean variables X and outputs circuits $\psi_1, \ldots, \psi_{m(n)}$ of size $s(n)$ over Boolean variables $X \cup Y$. In order for a decomposition to be useful for Step 2 (of Section 3.1.3), the decomposition also needs to be "matrix-based": We say that a decomposition is matrix-based if it is possible to arrange the variables $X \cup Y$ in a matrix such that the rows of the matrix are of length at most $s(n)$, and such that each circuit ψ_i reads variables *only from a constant number of rows*. The property of a decomposition being matrix-based is reminiscent of the parallelization technique used in the PCP literature, and we refer the reader to Section 3.3.2 for more details regarding how it is used.

3.2.1 The DR decomposition

Before describing our decomposition, we briefly sketch the DR decomposition. Given a circuit φ of size n over a variables set X, they transform φ into a 3-CNF formula by adding $O(n)$ auxiliary variables Y. Next, they choose some arbitrarily small constant $\alpha > 0$, and arrange the variables in $X \cup Y$ arbitrarily in an $O(n^{\alpha})$-by-$O(n^{1-\alpha})$ matrix. Finally, they construct for every triplet of rows of the matrix a circuit ψ_i that verifies all the clauses that depend on variables that reside only in those three rows (relying on the fact that each clause depends on three variables). This results in $m = O(n^{3\alpha})$ circuits of size $O(n^{1-\alpha})$, as described in Section 3.1.3.

3.2.2 Our decomposition

The inefficiency of the DR decomposition results from the fact that we construct a circuit ψ_i for every possible triplet of rows, since we do not know in advanc9e which variables will be used by each clause. Prior works in the PCP literature have encountered a similar problem in the context of efficient arithmetization of circuits, and solved the problem by embedding the circuit into a deBrujin graph using packet-routing techniques (see, e.g., [BFLS91, PS94]). While we could also use an embedding into a deBrujin graph in our context, we actually use a much simpler solution, taking advantage of the fact that the requirements that we wish to satisfy are weaker. We do mention, however, that our solution is still in the spirit of "packet-routing" ideas.

We turn to sketch the way our decomposition method works. Fix a circuit φ of size n. The decomposition acts on φ roughly as follows:

1. For each wire $w = (g_1, g_2)$ in φ (where g_1 and g_2 are gates of φ), the decomposition adds two auxiliary variables $v_w^{g_1}$ and $v_w^{g_2}$ representing the bit coming from g_1 and going into g_2 through the wire w respectively.
2. The decomposition arranges the variables in an $O(\sqrt{n}) \times O(\sqrt{n})$ matrix M_1 such that each pair of variables $v_w^{g_1}$ and $v_w^{g_2}$ are in the same row of M_1. Then, for each row of M_1, the decomposition outputs a circuit that checks for each pair $v_w^{g_1}, v_w^{g_2}$ in the row that $v_w^{g_1} = v_w^{g_2}$.
3. The decomposition outputs additional circuits that "sort" the variables in a new order, by routing the variables through a sorting network, while using additional auxiliary variables to represent the order of the variables at each intermediate layer of the sorting network. After the sorting, the variables are arranged in an $O(\sqrt{n}) \times O(\sqrt{n})$ matrix M_2 that has the following property: For each gate g of φ whose adjacted wires are $w_1, \ldots, w_k$, the variables $v_{w_1}^g, \ldots, v_{w_k}^g$ are in the same row of M_2 (note that without loss of generality we may assume that $k \leq 4$).
4. Finally, for each row of M_2, the decomposition outputs a circuit that checks for each gate g in the row that the variables $v_{w_1}^g, \ldots, v_{w_k}^g$ constitute a legal computation of g.

The straightforward way for implementing the sorting network in Step 3 is to implement each comparator of the network by an output circuit of the decomposition. However, this will result in the decomposition outputting $n \operatorname{poly}\log n$ circuits of size $O(1)$, while we wish it to output $\sqrt{n} \operatorname{poly}\log n$ circuits of size $O(\sqrt{n})$. In order to solve this problem, we view the wires of the sorting network as carrying whole rows of M_1 rather than single variables, and modify each comparator so that it sorts two whole rows of M_1 rather than sorting two single variables. After this modification, the sorting network uses $\sqrt{n} \operatorname{poly}\log n$ comparators, where each comparator acts on $O(\sqrt{n})$ variables. Such a network can be implemented in the decomposition in a straightforward way. To complete the analysis of this construction, we prove the following lemma on sorting networks, which may be of independent interest:

Lemma 3.1. *Let $m, k \in \mathbb{N}$, and let N be a sorting network over m wires. Let N' be the network that is obtained by modifying N such that each wire of N carries k elements instead of a single element, and by modifying the comparators of N as follows: When a comparator is given as input two lists of k elements, it outputs on its first wire the sorted list of the k smaller elements and outputs on its second wire the sorted list of the k greater elements. Then, whenever N' is invoked on any m lists of k elements, the concatenation of the lists that are placed on the output wires of N' forms a sorted list of mk elements.*

Remark 3.2. We note that the way the decomposition is described above it outputs circuits of size $O(\sqrt{n})$, while in Section 3.1.3 we stated that our decomposition outputs circuits of size $\sqrt{n}\text{poly}\log n$. The reason for this difference is that the foregoing description of the decomposition ignores the issue of implementing the decomposition in a super-fast way. In order to implement this decomposition efficiently, we need to use output circuits of size $\sqrt{n}\text{poly}\log n$ rather than $O(\sqrt{n})$.

3.3 The tensor product lemma

In this section we outline the proof of the following lemma, which is used in Step 2 (of Section 3.1.3).

Lemma 3.3 (Tensor Product Lemma, simplified)**.** *Let D be a matrix-based circuit decomposition that when given a circuit of size n outputs $m(n)$ circuits of size $s(n)$. Let A_S be an assignment tester that can take as input circuits of any size $n' \leq \max\{O(m(n)), s(n)\}$, outputs $R(n')$ circuits of size $O(1)$ and has rejection probability ε. Then, we can use D and A_S to construct an assignment tester A that when given as input a circuit of size n, outputs $R(O(m(n))) \cdot R(s(n))$ and has rejection probability $\Omega(\varepsilon^2)$.*

The construction of the assignment tester A from A_S and D is somewhat similar to the tensor product of error correcting codes, hence the name of the lemma. We note that the proof of this lemma is implicit in [DR06], although they only proved it for their specific choice of D and A_S. We stress that the proof of [DR06] does not maintain the super-fast running time of the assignment tester, and that one of our main contributions is proving the lemma for super-fast assignment testers (see discussion in Section 3.4).

3.3.1 Warm-up: Ignoring issues of robustness

As a warm-up, we consider the following thought-experiment: Let A' be an assignment tester that has rejection probability ε. We say that A' is idealized if when given an input circuit φ, at least an ε fraction of the output circuits of A' reject an assignment $x \circ y$ for *every unsatisfying assignment* x to φ (and not just for every assignment x that is *far from any satisfying assignment to* φ). Of course, it is impossible to construct an idealized assignment tester with the parameters we desire, but for the purpose of this warm-up discussion we ignore this fact.

We now show how to prove the tensor product lemma when both A and A_S are idealized (we note that in this case the assumption that D is matrix-based is not needed). When given as input a circuit φ of size n, the assignment tester A acts as follows:

1. The assignment tester A applies the circuit decomposition D to φ, resulting in a variable set Y and in $m(n)$ circuits $\psi_1, \ldots, \psi_{m(n)}$ of size $s(n)$ over $X \cup Y$.
2. The assignment tester A applies the assignment tester A_S to each of the circuits ψ_i, each time resulting in a variables set Z_i and in $R(s(n))$ circuits $\xi_{i,1}, \ldots, \xi_{i,R(s(n))}$ of size $O(1)$ over $X \cup Y \cup Z_i$.
3. The assignment tester A constructs circuits $\eta_1, \ldots, \eta_{R(s(n))}$ of size $O(m(n))$ over $X \cup Y \cup \bigcup_i Z_i$ by defining $\eta_j = \bigwedge_{i=1}^{m(n)} \xi_{i,j}$. Note that those circuits correspond to the columns of the matrix whose elements are the circuits $\xi_{i,j}$.
4. The assignment tester A applies the assignment tester A_S to each of the circuits η_j, each time resulting in a variables set W_j and in $R(O(m(n)))$ circuits $\tau_{1,j}, \ldots, \tau_{R(O(m(n))),j}$ of size $O(1)$ over $X \cup Y \cup \bigcup_i Z_i \cup W_j$.
5. Finally, A outputs the $R(O(m(n))) \cdot R(s(n))$ circuits $\tau_{1,1}, \ldots, \tau_{R(O(m(n))),R(s(n))}$ of size $O(1)$ over $X \cup Y \cup \bigcup_i Z_i \cup \bigcup_j W_j$.

Clearly, the assignment tester A has the correct number and size of output circuits. It remains to show that it has rejection probability $\Omega(\varepsilon^2)$. Let $Y' = Y \cup \bigcup_i Z_i \cup \bigcup_j W_j$ be the variables set of A. Fix an assignment x to X that does not satisfy φ and fix some assignment y' to Y'. Since x does not satisfy φ, there must exist some circuit ψ_i that rejects $x \circ y'$. This implies that at least an ε fraction of the circuits $\xi_{i,1}, \ldots, \xi_{i,R(s(n))}$ reject $x \circ y'$, let us denote those circuits by $\xi_{i,j_1}, \ldots, \xi_{i,j_k}$. Now, observe that since $\xi_{i,j_1}, \ldots, \xi_{i,j_k}$ reject $x \circ y'$, the circuits $\eta_{j_1}, \ldots, \eta_{j_k}$ must reject $x \circ y'$ as well, and that $\eta_{j_1}, \ldots, \eta_{j_k}$ form ε fraction of the circuits $\eta_1, \ldots, \eta_{R(s(n))}$. Finally, for each circuit η_{j_h} that rejects $x \circ y'$, it holds that at least ε fraction of the circuits $\tau_{1,j_h}, \ldots, \tau_{R(O(m(n))),j_h}$ reject $x \circ y'$, and it therefore follows that at least ε^2 fraction of the circuits $\tau_{1,1}, \ldots, \tau_{R(O(m(n))),R(s(n))}$ reject $x \circ y'$.

3.3.2 The actual proof

We turn to discuss the proof of the tensor product lemma for the actual definition of assignment testers, i.e., non-idealized assignment testers. In this case, the analysis of Section 3.3.1 breaks down. In order to see it, fix an assignment x to X that is far from satisfying φ and an assignment y' to Y'. As argued in Section 3.3.1, we are guaranteed that there exists a circuit ψ_i that is not satisfied by $x \circ y'$. However, we can no longer conclude that ε fraction of the circuits $\xi_{i,1}, \ldots, \xi_{i,R(s(n))}$ reject $x \circ y'$: This is only guaranteed when $x \circ y'$ is far from any satisfying assignment to ψ_i, which may not be the case.

The analysis of Section 3.3.1 does go through, however, provided that the circuits $\psi_1, \ldots, \psi_{m(n)}$ and $\eta_1, \ldots, \eta_{R(s(n))}$ have a property called "robustness". The PCP literature contains few techniques for "robustizing" the output of an assignment tester, provided that the assignment tester has specific properties. In this work, we define and analyze a generalization of the robustization technique of [DR06], which requires the assignment tester to have the following property:

Definition 3.4. We say that an assignment tester A that outputs circuits of size s is block-based if the variables in $X \cup Y$ can be partitioned to blocks of size at most s such that each output circuit of A queries variables from a constant number of blocks.

For example, every matrix-based assignment tester is also block-based, with the blocks being the rows of the corresponding matrix. We now have the following lemma:

Lemma 3.5 (Robustization)**.** *Any block-based assignment tester can be transformed into a robust one, with a linear blow-up in the number and size of output circuits and a decrease of a constant factor in the rejection probability. Furthermore, the same holds for any block-based* circuit decomposition.

In order to prove the tensor product lemma, we also need the assignment tester A_S to be oblivious. An assignment tester is said to be oblivious if for every i, the variables in $X \cup Y$ that the i-th output circuit ψ_i queries depend only on i, and in particular do not depend on the input circuit φ. The work of [DR06] has showed that every assignment tester can be transformed into an oblivious one with an almost-linear loss in the parameters.

We return to the analysis of Section 3.3.1. As said, this analysis goes through provided that the circuits $\psi_1, \ldots, \psi_{m(n)}$ and $\eta_1, \ldots, \eta_{R(s(n))}$ are robust. We deal with each set of circuits separately:

1. Making the circuits $\psi_1, \ldots, \psi_{m(n)}$ robust is easy: Recall that D is matrix-based, and is thus in particular block-based. We can therefore use Lemma 3.5 to robustize its output circuits.
2. In order to make the circuits $\eta_1, \ldots, \eta_{R(s(n))}$ robust, we act as follows. Recall that D is matrix-based, so let M be the corresponding matrix that consists of the variables in $X \cup Y$.

For every $i \in [m(n)]$, let $U_i \subseteq X \cup Y$ denote the set of variables queried by ψ_i. Without loss of generality, we may assume that U_i consists of whole rows of M, and that ψ_i reads each row consecutively (otherwise modify ψ_i appropriately). We also assume without loss of generality that all the circuits $\psi_1, \ldots, \psi_{m(n)}$ query the same number of rows. We now make the following observations.

(a) Each circuit $\xi_{i,j}$ only queries variables from $U_i \cup Z_i$.

(b) Furthermore, $\xi_{i,j}$ queries only a constant number of variables of $U_i \cup Z_i$ (since $\xi_{i.j}$ is of size $O(1)$).

(c) Moreover, since A_S is oblivious, the indices of the variables that $\xi_{i,j}$ queries inside $U_i \cup Z_i$ depend only on j, and do not depend on i. For example, if $\xi_{1,j}$ reads the first, third and fifth variables of $U_1 \cup Z_1$, then the circuit $\xi_{2,j}$ reads the first, third and fifth variables of $U_2 \cup Z_2$, the circuit $\xi_{3,j}$ reads the first, third and fifth variables of $U_3 \cup Z_3$, etc.

Let us define the matrix N whose (i, k)-th element is the k-th variable of Z_i. Now, combining the foregoing observations with our assumptions on the input structure of the circuits $\psi_1, \ldots, \psi_{m(n)}$ imply that the each circuit η_j queries variables only from a *constant number of columns* of M and N. Thus, the circuits $\eta_1, \ldots, \eta_{R(s(n))}$ are *block-based with the blocks being the columns of M and N*. We can therefore apply Lemma 3.5 to robustize the circuits $\eta_1, \ldots, \eta_{R(s(n))}$.

Thus, the construction of A described in Section 3.3.1 can be modified so as to make sure that the circuits $\psi_1, \ldots, \psi_{m(n)}$ and $\eta_1, \ldots, \eta_{R(s(n))}$ are robust. After this modification, the analysis outlined in Section 3.3.1 may proceed essentially as before.

3.4 Efficiency issues

We turn to discuss the modifications that need to be made to the foregoing construction in order to implement it efficiently.

3.4.1 Modifying the formalism

In Section 3.1.1 we defined an assignment tester as an algorithm that takes as input a circuit φ of size n and outputs R circuits $\psi_1, \ldots, \psi_R$ of size s. Clearly, such an algorithm must run in time $\min\{n, R \cdot s\}$. However, we want our assignment testers to run in time $\operatorname{poly}\log n$, which is much smaller than both n and $R \cdot s$. We therefore have to use a different notion of assignment tester in order to have the desired running time.

In order to resolve this issue, we use succinct representations of both the input circuit and the output circuits. The key observation is that in order to construct a PCPP for **NP**, we only need to run our assignment tester on circuits that are obtained from the standard reduction from **NP** to Circuit-Value, and that those circuits can be succinctly represented using $\operatorname{poly}\log n$ bits. An assignment tester can therefore be defined[6] as an algorithm that takes as input a succinct representation of a circuit φ and an index i, and outputs a succinct representation of a circuit ψ_i. Indeed, such an algorithm can run in time $\operatorname{poly}(\log n, \log(R \cdot s))$.

Using the new definition of assignment testers adds an additional level of complexity to our construction, since we implement all the ingredients of our construction (and in particular, the decomposition method, the tensor product lemma, and Dinur's amplification theorem) so as to work with succinct representations.

[6]We mention that this definition of assignment testers can be viewed as a variant of the notion of "verifier specifications" of [BSGH$^+$05].

3.4.2 Efficient implementation of the tensor product lemma

Working with succinct representation of circuits instead of with the circuits themselves is especially difficult in the implementation of tensor product lemma. The main difficulty comes from the need to generate succinct representations of the circuits $\eta_1, \ldots, \eta_{R(s(n))}$ (as defined in Section 3.3.1). Recall that those circuits are defined by $\eta_j = \bigwedge_{i=1}^{m(n)} \xi_{i,j}$. While each of the circuits $\xi_{i,j}$ has a succinct representation, this does not imply that their conjunction has a succinct representation. In particular, we can not allow the size of a representation of a circuit η_j to depend linearly on $m(n)$.

In order to solve this problem, we observe that the output circuits $\psi_1, \ldots, \psi_{m(n)}$ of D must be "similar", since they are all generated by the same super-fast decomposition. We then show that for each j, the similarity of the circuits $\psi_1, \ldots, \psi_{m(n)}$ can be translated into a similarity of the circuits $\xi_{1,j}, \ldots, \xi_{m(n),j}$, and that this similarity of $\xi_{1,j}, \ldots, \xi_{m(n),j}$ can be used to construct a succinct representation of η_j.

To be more concrete, we sketch a simplified version of our solution[7]. Consider the "universal circuit" U that is given as input a circuit C and a string x, and outputs $C(x)$. Now, for each i, instead of invoking A_S on ψ_i, we invoke A_S on U, and whenever one of the output circuits $\xi_{i,j}$ makes a query to an input bit of U that corresponds to the circuit C, we redirect the query to the corresponding bit of ψ_i. The circuits $\xi_{i,j}$ that are constructed in this manner should simulate the circuits ξ_{ij} that were constructed in Section 2. The point is that for any fixed j, the circuits $\xi_{1,j}, \ldots, \xi_{m(n),j}$ are identical to one another, and differ only in the way the queries are redirected. Using this fact, and the fact that the representation of each of the circuits ψ_i is generated by the super-fast decomposition, it is easy to construct a succinct representation of the circuit η_j. We note that in the actual construction, the circuit U is not given the circuit C but rather an error-correcting encoding of the succinct representation of C, and that U takes as an input an additional proof string.

3.4.3 A finer analysis of Dinur's amplification theorem

In order for our assignment testers to run in time poly log n, we need to make sure that all the steps taken in a single iteration incur only a constant factor blow-up in the running time. In particular, we need to show this holds for Dinur's amplification theorem, since this was not proved in [Din07].

It turns out that in order to analyze the running time of Dinur's amplification theorem, one should make additional requirements from the assignment testers. Specifically, recall that the proof of the amplification theorem works by representing the assignment testers as "constraint graphs", and by applying various transformations to those graphs. The running time of those transformations depends on the explicitness of those graphs. Thus, in order to be able to present a super-fast implementation of Dinur's amplification technique, we must make sure that the corresponding constraint graphs are strongly-explicit.

In the context of assignment testers, a strongly-explicit constraint graph corresponds to an assignment tester that has a super-fast "reverse lister" (a.k.a "reverse sampler"[8]). A reverse lister for an assignment tester A is an algorithm that behaves as follows: Suppose that on input φ over variables set X, the assignment tester A outputs circuits $\psi_1, \ldots, \psi_R$ over variables set $X \cup Y$. Then, given the name of a variable $x \in X \cup Y$, the reverse lister allows retrieving the list of all circuits $\psi_{i_1}, \ldots, \psi_{i_m}$ that take x as input. We therefore make sure that all the assignment testers

[7]We mention that a similar technique was used in [DR06] in order to transform an assignment tester to an oblivious one.

[8]The term "reverse sampler" was used in the context of PCPs [BG02]. However, we feel that the term "reverse lister" is more natural in the context of assignment testers.

we construct in this work have corresponding super-fast reverse listers.

3.4.4 Revisiting known PCP techniques

Our construction uses known PCP techniques such as composition (see [BSGH$^+$06, DR06]) and robustization (see Lemma 3.5). However, when using those techniques in our construction, we have to make sure that those techniques preserve the super-fast running time of the assignment testers, as well as super-fast running time of their corresponding reverse listers (needed for the analysis of Dinur's amplification theorem). We thus present new implementations of those techniques that meet both the latter conditions. In particular, we make the following contributions:

1. **Composition**: While a super-fast implementation of the compoisition technique has been proposed in [BSGH$^+$06], their implementation did not preserve the running time of the corresponding reverse listers. In this work, we give a more sophisticated implementation that does preserve the running time of the reverse listers.

2. **Robustization**: As mentioned in Section 3.3.2, in this work we present a generalization of the robustization technique of [DR06]. In particular, the robustization technique of [DR06] works only for block-based assignment testers whose blocks are all of the same size, and contain only proof bits, while the bits of the tested assignment a read separately. Waiving those restrictions supports a cleaner proof of the Tensor Product lemma (Section 3.3).
   Implementing the robustization technique for super-fast assignment testers requires the block structure of the block-based assignment tester to have a strongly explicit representation. We define this representation and provide a super-fast robustization theorem.

3. **Proof length:** We revisit the connection between the randomness complexity of a PCP and its proof length, which is more complicated in the settings of super-fast PCPs than in the common settings. In the PCP literature it is common to assume that the proof length of PCPs is bounded by an exponential function in the randomness complexity. It is not clear that this can be assumed without loss of generality in the setting of super-fast PCPs. However, we show that this assumption can indeed be made for assignment testers that have super-fast reverse listers.

Acknowledgement. The author is grateful to Oded Goldreich for many useful discussions and ideas.

References

[ALM$^+$98] Sanjeev Arora, Carsten Lund, Rajeev Motwani, Madhu Sudan, and Mario Szegedy. Proof verification and intractability of approximation problems. *Journal of ACM*, 45(3):501–555, 1998. Preliminary version in FOCS 1992.

[AS98] Sanjeev Arora and Shmuel Safra. Probabilistic checkable proofs: A new characterization of NP. *Journal of ACM volume*, 45(1):70–122, 1998. Preliminary version in FOCS 1992.

[BFL91] László Babai, Lance Fortnow, and Carsten Lund. Non-deterministic exponential time has two-prover interactive protocols. *Computational Complexity*, 1:3–40, 1991.

[BFLS91] László Babai, Lance Fortnow, Leonid A. Levin, and Mario Szegedy. Checking computations in polylogarithmic time. In *STOC*, pages 21–31, 1991.

[BG02] Boaz Barak and Oded Goldreich. Universal arguments and their applications. In *IEEE Conference on Computational Complexity*, pages 194–203, 2002.

[BSGH$^+$05] Eli Ben-Sasson, Oded Goldreich, Prahladh Harsha, Madhu Sudan, and Salil Vadhan. Short PCPs verifiable in polylogarithmic time. pages 120–134, 2005.

[BSGH$^+$06] Eli Ben-Sasson, Oded Goldreich, Prahladh Harsha, Madhu Sudan, and Salil Vadhan. Robust PCPs of proximity, shorter PCPs and applications to coding. *SIAM Journal of Computing*, 36(4):120–134, 2006.

[BSS05] Eli Ben-Sasson and Madhu Sudan. Simple PCPs with poly-log rate and query complexity. In *STOC*, pages 266–275, 2005. Full version can be obtained from Eli Ben-Sasson's homepage at `http://www.cs.technion.ac.il/~eli/`.

[Din07] Irit Dinur. The PCP Theorem by gap amplification. *Journal of ACM*, 54(3):241–250, 2007. Preliminary version in STOC 2006.

[DR06] Irit Dinur and Omer Reingold. Assignment testers: Towards combinatorial proof of the PCP theorem. *SIAM Journal of Computing*, 36(4):155–164, 2006.

[PS94] Alexander Polishchuk and Daniel A. Spielman. Nearly-linear size holographic proofs. In *STOC*, pages 194–203, 1994.

[Sze99] Mario Szegedy. Many-valued logics and holographic proofs. In *ICALP*, pages 676–686, 1999.