A (De)constructive Approach to Program Checking

Shafi Goldwasser
MIT and Weizmann Institute
shafi@theory.csail.mit.edu

Dan Gutfreund
SEAS, Harvard University
danny@eecs.harvard.edu

Alexander Healy
SEAS, Harvard University
ahealy@fas.harvard.edu

Tali Kaufman
CSAIL, MIT
kaufmant@mit.edu

Guy N. Rothblum
CSAIL, MIT
rothblum@csail.mit.edu

Abstract

Program checking, program self-correcting and program self-testing were pioneered by Blum and Kannan and Blum, Luby and Rubinfeld as a new way to gain confidence in software, by considering program correctness on an input by input basis rather than full program verification. Work in the field of program checking focused on designing, for specific functions, checkers, testers and correctors which are more efficient than the best program *known* for the function. These were designed utilizing specific algebraic, combinatorial or completeness properties of the function at hand.

In this work we introduce a novel composition methodology for improving the efficiency of program checkers. We use this approach to design a variety of program checkers that are provably more efficient, in terms of circuit depth, than the *optimal* program for computing the function being checked. Extensions of this methodology for the cases of program testers and correctors are also presented. In particular, we show:

- For all $i \geq 1$, every language in $\mathcal{RNC}^i$ (that is $\mathcal{NC}^1$-hard under $\mathcal{NC}^0$-reductions) has a program checker in $\mathcal{RNC}^{i-1}$. In addition, for all $i \geq 1$, every language in $\mathcal{RNC}^i$ (that is $\mathcal{NC}^1$-hard under $\mathcal{AC}^0$-reductions) has a program corrector, tester and checker in $\mathcal{RAC}^{i-1}$. This is the first time checkers are designed for a *wide* class of functions *characterized only by its complexity*, rather than by algebraic or combinatorial properties. This characterization immediately yields new and efficient checkers for languages such as graph connectivity, perfect matching and bounded-degree graph isomorphism.
- Constant-depth checkers, testers and correctors for matrix multiplication, inversion, determinant and rank. All previous program checkers, testers and correctors for these problems run in nearly logarithmic depth. Moreover, except for matrix multiplication, they all require the use of the library notion of [Blum-Luby-Rubinfeld], in which checkers have access to a library of programs for various matrix functions, rather than only having access to a program for the function being checked.

Important ingredients in these results are new and very efficient checkers for complete languages in low complexity classes (e.g. $\mathcal{NC}^1$). These constructions are based on techniques that were developed in the field of cryptography.

Contents

1 Introduction

This work puts forth a methodology for constructing program checkers, testers, and correctors which are provably more efficient in terms of their *circuit depth* (or alternatively parallel computing time) than *any* algorithm that computes their function. The crux of the new idea is to make these objects more efficient by systematically delegating some of their work to the potentially faulty program being checked. Although our focus is on the circuit depth of these objects, the general delegation methodology may, in principal, also be useful for improving other complexity measures such as (sequential) time and space. We elaborate on our choice of complexity measure in Section 1.4.

We use the methodology in two ways. First, to design highly efficient checkers, testers, and correctors for entire classes of functions characterized by their structural complexity, rather than by their algebraic or combinatorial properties. These classes include functions such as graph connectivity, perfect matching, and bounded-degree graph isomorphism. Second, to give an array of new checkers, correctors and testers for specific matrix functions, giving the first known checkers for some functions, and significantly improving known checkers for others.

We believe that our results and techniques shed new light on important issues in the field of program checking, as discussed throughout this paper. To best state our results and explain their significance, we first give a brief review of definitions and of the previous work on program checking, as well as the impact it has had on complexity theory.

1.1 Program Checking and its Impact

In the mid-eighties, Blum and Kannan [BK95] proposed the methodology of program "result checking" (or "instance checking"), which focuses on correctness of an algorithm *per input* rather than full program verification. The methodology associates every function to be computed with a new algorithm called the *checker*. Then, given *any* possibly buggy program for the function and any input, the checker "checks" whether the program on this input computes the function correctly.

The work of Blum, Luby, and Rubinfeld [BLR93] further introduced the notion of function *testers* and *correctors*. A tester tests whether a given program for a function is correct on random inputs (with relatively high probability). A corrector for a function is given an input to the function and a program (for computing the function) which may be buggy, but is guaranteed to compute the function correctly on random inputs (with relatively high probability over the choice of inputs), and outputs a correct output on the given input with high probability (over the randomness of the corrector).

While the notion of program checking (testing and correcting) may be motivated by "real-life" applications, it has had a fundamental impact on theoretical computer science, as it is related to basic questions regarding the nature of computation. The definition itself captures many notions that are central in theoretical computer science, such as sub-linear time algorithms (as the checker probes the truth table of the program in a few places), and recovering from faulty data (in the case of correctors). It is therefore not surprising that ideas and techniques that were developed in this field have been used extensively in other areas. Indeed, several techniques from the early results on correctors and testers (in particular for the integer multiplication function [BLR93] and for the matrix permanent function by Lipton [Lip91]) were pivotal in showing the expressive power of IP and PCP proof systems, and the notion of testers in and of itself has evolved into the successful field of property testing. In fact, the techniques that we develop in this paper have already born fruit in work on interactive proofs and error-correcting codes [GGH$^+$07]. Further applications of program

checkers can be found in lower bounds proofs [San07], hierarchy theorems [Bar02, FS04, FST05], de-randomization [BFNW93, IKW02, GSTS03, SU07], average-case complexity [TV07] and more. The common theme in all these applications is that often one has a circuit that *should* compute some function, but no guarantee that it indeed does so. For example, such a circuit may be chosen non-deterministically (e.g. as in [BFNW93]), or it may belong to a collection of circuits, only an unknown one of which is correct (e.g. as in [TV07]). This is where program checking comes into play; by running the circuit (on a given input) through the checker, we are guaranteed (w.h.p.) to either get the correct answer or to detect a problem.

The focus of the rich body of work in the result checking field has been the design of efficient checkers (and testers/ correctors) for many *specific* functions, by exploiting either their algebraic or combinatorial properties. Most notably, these functions include arithmetic operations, matrix operations, and certain graph and group operations. By and large, these are function families which possess random and downwards self-reducibility properties.

Interestingly, the connection between the complexity of a problem and its checkability is not very clear. For example, the matrix permanent function, which is notoriously hard to compute, was one of the first to be shown easy to self-correct [Lip91]. In contrast, the matrix determinant function (which is efficiently computable) was not known to have a non-trivial checker in the standard sense.[1] An interesting open question is whether problems of related complexity (say problem π_1 is reducible to problem π_2) have related checkers (or even if the existence of a checker for π_2 implies a checker for π_1). Beigel [BK95] shows that if two decision problems are *equivalent*, then designing a checker for one would immediately provide a checker for the other. However, we do not know how to use a checker for an easy problem to construct a checker for a harder problem. In contrast with many other areas of complexity theory, it is unclear how to leverage the existence of checkers for complete-problems toward the design of checkers for other (not necessarily complete) problems.

Since a correct algorithm for a given function is also trivially a checker for the function, [BK95] required, in order to avoid triviality, that checkers will have the *little-oh time property*: the running time of the checker must be little-oh of the running time of the most efficient *known* program that computes the function. This ensures a quantifiable difference between the checker and the programs it checks. An analogue *little-oh parallel time property* was considered by Rubinfeld [Rub96]: a checker's parallel running time should be little-oh of the parallel running time of the most efficient known program that computes the function.[2] (Throughout, the standard complexity measure of oracle algorithms is used, where the complexity of the algorithm is measured *without* the complexity of the oracle's computations.)

An even more ambitious goal than constructing checkers that beat the best algorithm known for the function, is to construct checkers that beat *any* algorithm for the function (or alternatively the *optimal* algorithm). Currently no checkers are known to be more efficient (in terms of sequential running time) than the optimal programs for the functions being checked since **no super-linear lower bounds are known for any explicit function**. The work of [Rub96] does exhibit constant-depth (i.e., $\mathcal{AC}^0$) checkers for (specific) functions that have a super-constant lower bound on their circuit depth, thus provably separating the complexity of checking and computing in terms of circuit depth.

The work presented here addresses and sheds light on several of the issues mentioned:

[1] A checker was known in the library model of [BLR93]. See section 1.2 for more details on the model.

[2] More precisely if a program can be computed by $p(n)$ processors in depth $d(n)$, [Rub96] requires the checker to run either in depth $o(d(n))$ or in depth $O(d(n))$ with $o(p(n))$ processors.

- *Computing versus Checking*: can we relate the computational complexity of a function and the complexity of checking it? Can we design checkers for functions classified by their complexity, rather than by their algebraic and combinatorial properties?
- *General Tools:* give generally applicable systematic tools for designing and improving the complexity of program checkers.
- *How to Meaningfully Distinguish Checkers from Programs*: is it a sufficient distinction to assure quantifiable difference between the running time of a checker for a function and the running time of the optimal program for computing the function?

1.2 New Checkers, Testers & Correctors

Checkers for Complexity Classes. We construct checkers that are provably more efficient than computing the functions they check (in terms of circuit depth) for entire complexity classes, and not just specific functions with special algebraic or combinatorial properties.

Theorem 1.1. *For every $i \geq 1$, every language in $\mathcal{RNC}^i$ that is $\mathcal{NC}^1$-hard under $\mathcal{NC}^0$-reductions has a checker in $\mathcal{RNC}^{i-1}$. Every language in $\mathcal{RNC}^i$ that is $\mathcal{NC}^1$-hard under $\mathcal{AC}^0$ reductions has a tester and corrector (and checker) that are in $\mathcal{RAC}^{i-1}$.*

See Section 2 for the definitions of the complexity classes involved, and Section 5 for a discussion and full proofs.

The requirement of being $\mathcal{NC}^1$-hard under $\mathcal{NC}^0$ reductions may seem restrictive, but in fact this is not the case. Barrington's [Bar89] characterization of $\mathcal{NC}^1$ as languages that have small width branching programs allows one to capture $\mathcal{NC}^1$ computations by many graph theoretic and algebraic functions. Examples of natural functions and languages that satisfy the theorem requirements, and for which no provably better checkers were previously known, include *graph connectivity* (in its many variants), deciding whether a given graph has a *prefect matching* and *bounded-degree graph isomorphism*.[3] Theorem 1.1 gives provably better checkers for all these functions (see below). Other examples include computing the determinant of a matrix, matrix exponentiation, and more.

Finally, we note that the proof of the theorem is constructive: it shows how to transform any program in $\mathcal{RNC}^i$ for a language into a checker in $\mathcal{RNC}^{i-1}$ (or a tester/corrector in $\mathcal{RAC}^{i-1}$) (for all programs) for that language. Languages satisfying the requirements of Theorem 1.1 have checkers that are provably more efficient (in terms of circuit depth) than their optimal program. Indeed, for a language L satisfying the theorem conditions, let $i \geq 1$ be such that L is in $\mathcal{RNC}^i$ but not in $\mathcal{RNC}^{i-1}$ (i is well defined, as $\mathcal{RNC}^0$ is strictly contained in $\mathcal{RNC}^1$). By the theorem, this language has a checker in $\mathcal{RNC}^{i-1}$. Note that even if we currently do not know the best algorithm for the language, the theorem (or rather the proof) still yields a checker that satisfies the little-oh parallel time property with respect to the best algorithm that is currently known. In the future, any algorithmic improvement on the language (placing it in a lower $\mathcal{RNC}$ class), will immediately give rise to an even better checker (placing it in an even lower $\mathcal{RNC}$ class).

Constant Depth Checkers for Matrix Problems. [BLR93] consider the problem of testing and correcting matrix functions such as multiplication, inverse, determinant and rank. They suggested a non-standard model in which the checker/ tester/corrector can access (at unit cost) not only the

[3]Note that [BK95] gave a checker for (unbounded degree) graph isomorphism, but this checker is not known to be provably better, especially for the efficiently solvable case of bounded degree graphs.

program to be checked, but also a **library** of (possibly faulty) programs that allegedly compute other related functions. Within this extended model, they show how to test and correct (and thus check) programs for the above matrix functions. For example, their determinant checker uses access to a program that allegedly computes matrix determinant (as in the usual setting), but it also has access to programs that allegedly compute matrix multiplication and inversion.

We present *standard* checkers, testers and correctors for matrix multiplication, inversion, rank and determinant, removing altogether the need for the matrix library introduced in [BLR93]. These checkers/testers/correctors can be implemented in $\mathcal{AC}^0$, and for some ranges of parameters even in $\mathcal{NC}^0$.[4] Except for the matrix multiplication function, they are the first checkers that are *provably* more efficient than the optimal program for computing these functions in terms of circuit depth.[5] Previous known checkers/testers/correctors for matrix problems both relied on a program library and had high parallel complexity. Furthermore, we note that the checkers we build for matrix multiplication and matrix inversion are *optimal* up to constant factors in every parameter: depth (or parallel time), size (or number of processors) and number of queries. A summary is given in Table 1 of Section 6.

Theorem 1.2. *Matrix multiplication, inversion, determinant and rank have all probabilistic $\mathcal{AC}^0$ checkers, testers and correctors.*[6] *Over a field of cardinality 2^s for a constant s, matrix multiplication and inversion have probabilistic $\mathcal{NC}^0$ checkers, testers and correctors that perform a constant number of calls to the program oracle .*

The checkers (and tester/correctors) for the various matrix operations are constructed by starting from high depth checkers inspired by (though sometimes quite different from) the library checkers of [BLR93]. We then apply the general method (i.e. the composition theorem stated in Section 1.3), along with other algebraic ideas, to improve their circuit depth. Often the theorem is applied more than once, gradually improving the checker until it reaches constant depth. See Section 3 for discussion and proofs.

1.3 New Techniques and Tools

Checker Composition and Delegating Computation. The guiding principle of this work is to design checkers that work as little as possible. We observe that a checker has access to a potentially powerful resource: the (allegedly correct) program it is checking, which can often compute a complex function. Our goal is thus to *delegate* computations from the checker to the program being checked, all the while verifying that the results returned for these delegated computations are correct.

To realize this goal, we propose a composition methodology for improving the complexity of checkers. The crux of the idea is to start with a checker C for the function at hand – this C may be a previously designed checker, or even just a correct program for the function (which trivially gives a checker) – and then decompose this checker into *sub-computations*. The work of these sub-computations is in turn replaced by calls to P, the potentially faulty program being checked, on

[4] An $\mathcal{NC}^0$ checker is a constant depth fan-in 2 checker with oracle gates to the program. Naturally, these oracle gates have unbounded fan-in (as the checker runs the program on growing inputs). Previous work did not present checkers in this low complexity class.

[5] [BLR93, Rub96] give an $\mathcal{AC}^0$ tester and corrector for matrix multiplication that make $O(\log n)$ program oracle calls. These are somewhat non-standard in that the corrector needs the given program to work well on *each input length*, from the length of its input and down, *simultaneously*. The tester may reject a program even if it is good on all input lengths but one, and in particular even if it is perfectly correct on the input length being tested.

[6] For rank the result holds only over fields that are of size polynomial in the input length.

appropriate inputs. This is done by applying a reduction that maps sub-computations to instances of the function that P allegedly computes. In other words, we delegate the computations of these sub-tasks to P. The correctness of these delegated sub-computations performed by P is finally verified by checkers for the sub-computations.[7] When the checkers for the sub-computations are more efficient than the sub-computations themselves, this results in a new checker with improved efficiency.

We capture this approach in a checker composition theorem. Take $L_{external}$ to be the language being checked, and consider the case where the checker's sub-computations are of a language $L_{internal}$.

Theorem 1.3 (Checker Composition)**.** *Let $L_{internal}$ and $L_{external}$ be two languages that satisfy the following conditions:*

1. *There exists an efficient*[8] *reduction from $L_{internal}$ to $L_{external}$.*
2. *$L_{external}$ has an efficient checker of depth $d_{external}$ that has oracle access to $L_{internal}$ (note that by definition it also has oracle access to a program that allegedly computes $L_{external}$.)*
3. *$L_{internal}$ has an efficient checker of depth $d_{internal}$.*

Then there exists an efficient checker for $L_{external}$, of depth $O(d_{external} \cdot d_{internal})$.

The improvement comes from the fact that the depth of the resulting checker does not depend on the depth of *computing* $L_{internal}$, only on the depth $d_{internal}$ of *checking* it, which may be much smaller. We mention that a similar idea, albeit in a very specific setting, was used in the work of Arvind, Subrahmanyam and Vinodchandran [ASV02].

Note that while the Composition Theorem as stated here only explicitly considers checking languages (i.e. boolean functions), it extends naturally (and is used in this work) for checkers of non-boolean functions. See Section 3 for a formal statement of the theorem, the proof and extensions, as well as an analogous composition theorem for testers and correctors.[9]

The composition methodology provides a simple way to design checkers that is very similar to the top-down approach of algorithm design: first decompose the problem into smaller (and easier) sub-problems, solve them and then combine these solutions to solve more the complex problem, all the while ensuring errors are kept under control. This approach can be used iteratively to get better and better checkers (for example see our checkers, testers and correctors for matrix determinant in Section 6). Moreover, this approach enables us to construct checkers for functions that do not necessarily have the type of self-reducibility or completeness properties exploited in previous works of [BK95, BLR93, Lip91, Sha92, BFL91].

It is illuminating to compare Theorem 1.3, in this regard, to Beigel's Theorem [BK95] that informally says that a checker for a decision problem π_1 can be used to design a checker for a decision problem π_2, if π_1 and π_2 are "computationally equivalent" (reducible to each other). The

[7] More precisely, we can consider the composed program, P', that applies the reduction and then applies P on the result of the reduction. Thus, P' can be viewed as a program that allegedly computes the sub-computation, and we can then check P' with a checker for this sub-computation.

[8] Throughout, efficient means poly-time (or poly-size when talking about circuits). In addition, we require the reduction to have constant-depth. See Section 3 for the exact statement.

[9] We mention that while the proof of the composition theorem for program checkers is reasonably straightforward, for testers and correctors the argument is more delicate and involved.

idea is that to design a checker for π_2, we can reduce any π_2 instance to a π_1 instance, run the checker for π_1 on this instance, while translating π_1 queries that this checker makes to π_2 instances and feeding them to the program being checked (which allegedly computes π_2). The conceptual difference in our approach is that instead of translating back and forth between instances of the two "external" equivalent languages, we delegate the computations of the *checker itself* to the program being checked. It turns out that this simple idea is surprisingly powerful, both in the checking setting (as we show here) and in other settings [GGH+07]. In particular it is this difference that allows one to use a checker for an easy problem to construct a checker for a harder problem, as well as allowing the whole top-down approach (which is impossible using Beigel's theorem, as the two languages must be computationally equivalent). Finally, we want to point out that while Beigel's theorem uses the checker for π_1 as a black-box, our approach of decomposition is *inherently non-black-box.*

Main Building Blocks: Checkers for Complete Languages. To apply our methodology in a general manner (rather than only working on checkers for specific problems), we look for sub-computations (as described above), that on one hand capture many functionalities, and are thus helpful in the design of checkers, and on the other hand are themselves not very complex, so we can delegate them to programs for many functions that we may want to check. Furthermore, and just as importantly, these functions must have very efficient checkers, testers and correctors, so that we gain in efficiency when replacing the task of computing these functions with the task of checking them. We find such functions in the form of complete languages for low complexity classes such as $\mathcal{NC}^1$. For example, we build an $\mathcal{NC}^0$ checker, and an $\mathcal{AC}^0$ tester and corrector for the $\mathcal{NC}^1$-complete problem given by Barrington [Bar89]. The efficiency of the checkers for this language (as well as other useful languages such as Parity) are based on techniques that were developed in the field of cryptography by Kilian [Kil88], Feige, Kilian and Naor [FKN94] and Ishai and Kushilevitz [IK02].

Theorem 1.4. *There is an $\mathcal{NC}^0$ checker for the parity function, as well as for problems that are complete (under $\mathcal{NC}^0$ reductions) in the class $\mathcal{NC}^1$ and classes that contain $\mathcal{NL}$ (such as $\oplus$-$\mathcal{L}$ and mod_k-$\mathcal{L}$).*

For definitions of these complexity classes see Section 2, for proofs and full statements of results, see Section 4. The Composition Theorem enables us to use these checkers for languages complete for weak classes (such as $\mathcal{NC}^1$) to construct efficient checkers for languages in higher complexity classes. We emphasize that unlike other properties of functions (or languages), the existence of checkers for complete languages did not previously seem to imply or be related to the existence of checkers for non-complete languages (although, by Beigel's Theorem, it does imply checkers for other languages that *are* complete). Indeed, (likely for this reason) past work was more concerned with checkers for useful and practical functions, and less with checkers for complete languages. This is in contrast to many other areas of complexity theory, where demonstrating properties of complete languages has direct implications for an entire complexity class.[10]

1.4 Other Contributions and Comments

On Circuit Depth as a Checker Complexity Measure. In this work we focus on the circuit depth (or parallel time) of checkers. To some extent this choice is because our main building

[10] One such example is interactive proofs, where exhibiting an interactive proof for a complete language immediately implies interactive proofs for the entire, and thus research on interactive proofs focused on treating hard (non-polynomial time) complete languages and whole complexity classes.

blocks (i.e. the checkers for complete functions) are more efficient in terms of circuit depth than computing their functions. It is this fact that enables us to achieve one of the primary goals of this work, namely to construct a variety of checkers that are *more efficient* (in terms of depth) than *any* algorithm for the functions being checked. Obtaining similar results for sequential time seems currently beyond our reach, as there are no known non-trivial lower bounds on the time complexity of computing any explicit function (this is in contrast to the circuit depth measure [FSS84]). We do emphasize again though, that in principal the composition methodology can be used to improve the time complexity of checkers, if we can identify sub-computations of the checker ($L_{internal}$ in Theorem 1.3) for which checking is more efficient, in terms of time complexity, than computing. Thus the depth measure, interesting on its own, also serves here as a test-bed for a general program checking paradigm.

The Little-Oh property and Independence of Errors. Blum and Kannan [BK95], followed by Blum and Wasserman [WB97], argue that since the little-oh time requirement gives assurance that the checker is different than the program itself, then "heuristically it must be doing something essentially different from what P (the program) does and so if buggy may reasonably be expected to make different errors" ([WB97], page 8), which intuitively will decrease the likelihood of "correlated errors" and a bug going undetected. Interestingly, the proof of Theorem 1.1 suggests that the above intuition is not sound, at least as far as the little-oh parallel time property is concerned. The idea used in the proof of Theorem 1.1 is to come up with an efficient checker by starting with a correct program for the function. This checker has the little-oh parallel time property with respect to any program (and not just the best known one), yet its description is based on the best algorithm for the function being checked. In fact, if one has bugs in the implementation of the particular correct algorithm from which the checker is derived, then these bugs are likely to also show up in the checker!

Provably Beating the Best Program without Knowing A Lower Bound for the Function. The works of [BK95] and [Rub96] focus on designing checkers that are more efficient than the best *known* program for the function, rather than the *optimal* program for the function.[11] As stated above, the first examples of checking that is provably easier than computing are from [Rub96], which exhibits constant-depth ($\mathcal{AC}^0$) checkers for functions (such as parity) that have a nearly logarithmic circuit depth lower bound (see [FSS84]). The separation between checking and computing in [Rub96] is due to known lower bounds on the parallel complexity of the function in question.

In fact, it is tempting to conjecture that proving the existence of a checker that is more efficient than *any* program for the function *requires* presenting an explicit checker that beats a *specific and known* lower bound for the function. This can be "hand waived" as follows: to prove that a checker is faster than all programs for a function, we must both know a lower bound on the complexity of computing the function, and then design a checker that beats that lower bound (as in [Rub96]).

Theorem 1.1 shows that this "hand waiving" argument is misleading. It shows the existence of checkers that are more efficient (in circuit depth) than any program for their function, without

[11] To see the significance of this distinction, consider the matrix multiplication function. The best known algorithm for it runs in time roughly $n^{2.37}$ (for $n \times n$ matrices) [CW90]. While we do have a $O(n^2)$ time checker for this function [Fre79], and while this checker satisfies the little-oh time property, it is not known to run in little-oh time of *all algorithms* for matrix multiplication. It is possible that one day an $O(n^2)$ matrix multiplication algorithm will be found, and Freivalds' checker (and all other known checkers for matrix multiplication) will cease to satisfy the little-oh property.

proving that they beat any specific known lower bound. This is because it uses the code of the optimal program (without necessarily knowing what it is) to construct a more efficient checker. The complexity of these checkers varies with the complexity of the optimal program.

1.5 Other Related Work

As discussed above, our work benefits from a long line of beautiful results on program checking, interactive proofs and cryptography. It is instructive to compare our work in detail to recent works.

Applebaum, Ishai and Kushilevitz [AIK06]. Some of our techniques are inspired by the breakthrough work of Applebaum, Ishai and Kushilevitz [AIK06] on improving the efficiency of cryptographic primitives. The work of [AIK06] can also be viewed as improving the efficiency of players participating in a (cryptographic) protocol by pushing computation from one of the participants to another (e.g. improving the efficiency of encryption at the expense of adding to the complexity of decryption). The main difference between this approach and ours is that they consider protocols or objects in which the *goal* of a sender is to reveal the results of its computation to a receiver, so there is no issue of a malicious party that may corrupt the computation. The main concern in [AIK06] is privacy, or in other words, how can one party (while conducting very efficient computations) reveal the outcome of its computation without revealing how the computation was conducted. In our case, on the other hand, the program is *un-trusted* but the checker still wants it to perform its computations. Our concern is how to perform this (very efficiently) while still maintaining soundness. While we use the same strong random self-reducability properties of complete languages that [AIK06] used to guarantee privacy, we do so in a different way to both test and correct programs computing these languages (see Section 4 for the details).

Goldwasser et al. [GGH$^+$07]. The recent work of Goldwasser *et al.* [GGH$^+$07] uses the methodology of delegating computation (originally developed here) as a way to (1) improve the efficiency of verifiers in interactive proofs and (2) improve the efficiency of decoding algorithms for error correcting codes. The settings of program checking, interactive proofs, and error correcting codes, each present a different set of challenges for the delegation paradigm, as we discuss below.

In the program checking domain, in contrast to the standard interactive proof setting, one needs to address the additional challenge of efficient proof verification by delegation to a *restricted prover*: the (honest) prover can only answer queries to the language being proved (as opposed to being computationally unbounded in the interactive proof setting). The fact that the (honest) prover is restricted to answering queries about the language being proved necessitates careful design of protocols that typically use very specific properties of the functions being proved (checked). In fact, it is not at all well understood which languages have such proof systems. On the other hand, in the setting of interactive proofs [GGH$^+$07] one must deal with the challenge that the (dishonest) prover may change its answers according to the messages exchanged in the interaction.

The results of [GGH$^+$07] on constructing efficient error-correcting codes are related to program correctors. Again, in the error-correcting code setting one can design decoders that delegate work to an encoder that can perform arbitrary (efficient) computations, whereas in the program correction setting the restricted encoder can only compute membership in a specific language. The challenge in the coding setting is recovering from (potentially very large) errors introduced by the noisy channel.

2 Definitions and Preliminaries

For a string $x \in \Sigma^*$ (where Σ is some finite alphabet) we denote by $|x|$ the length of the string, and by x_i or $x[i]$ the i'th symbol in the string. For a finite set S we denote by $y \in_R S$ that y is a uniformly distributed sample from S. For $n \in \mathbb{N}$, we denote by $[n]$ the set $\{1, 2, \ldots, n\}$. For a finite alphabet Γ we denote by Δ_Γ the relative (or fractional) Hamming distance between strings over Γ. That is, let $x, y \in \Gamma^n$ then $\Delta_\Gamma(x, y) = \Pr_{i \in_R [n]}[x[i] \neq y[i]]$, where $x[i], y[i] \in \Gamma$. Typically, Γ will be clear from the context, we will then drop it from the subscript.

2.1 Circuit and Complexity Classes

We assume that the reader is familiar with standard complexity classes such as $\mathcal{NP}$, $\mathcal{EXP}$ and $\mathcal{NEXP}$. For a positive integer $i \geq 0$, $\mathcal{AC}^i$ circuits are Boolean circuits (with AND, OR and NOT gates) of size $\text{poly}(n)$, depth $O(\log^i n)$, and unbounded fan-in AND and OR gates (where n is the length of the input). $\mathcal{NC}^i$ circuits are boolean circuits of size $\text{poly}(n)$ and depth $O(\log^i n)$ where the fan-in of AND and OR gates is 2. We use the same notations to denote the classes of functions computable by these circuit models. We denote by $\mathcal{AC}$ the class $\bigcup_{i \in \mathbb{N}} AC^i$, and by $\mathcal{NC}$ the class $\bigcup_{i \in \mathbb{N}} NC^i$. $\mathcal{RNC}^i$, $\mathcal{RAC}^i$, $\mathcal{RNC}$ and $\mathcal{RAC}$ are the (one-sided) randomized analogs of the above classes. In particular, $\mathcal{AC}^0$ circuits are boolean circuits (with AND, OR and NOT gates) of constant-depth, polynomial size, and unbounded fan-in AND and OR gates. $\mathcal{NC}^1$ circuits are boolean circuits of fan-in 2, polynomial size and logarithmic (in the input size) depth. $\mathcal{NC}^0$ circuits are similar to $\mathcal{NC}^1$, but have constant-depth. Note that in $\mathcal{NC}^0$ circuits, every output bit depends on a constant number of input bits. $\mathcal{AC}^0$, $\mathcal{NC}^1$ and $\mathcal{NC}^0$ are the classes of languages (or functions) computable (respectively) by $\mathcal{AC}^0/\mathcal{NC}^1/\mathcal{NC}^0$ circuits. $AC^i[q]$ (for a prime q) are similar to $\mathcal{AC}^i$ circuits, but augmented with $\mod q$ gates.

Throughout, circuits may have many output bits (we specify the exact number when it is not clear from the context). Also, often we consider uniform circuit classes. Unless we explicitly note otherwise, circuit families are log-space uniform, i.e. each circuit in the family can be described by a Turing machine that uses a logarithmic amount of space in the size of the circuit (for non-polynomial size circuit families the default uniformity is using space logarithmic in the circuit family size). Thus $\mathcal{NC}^0$ (resp. $\mathcal{NC}^1$) computations in this work are equivalent to constant (resp. logarithmic) parallel time in the CREW PRAM model.

Finally, we extensively use oracle circuits: circuits that have (unit cost) access to an oracle computing some function. We sometimes interpret this function as a string, in which case the circuit queries and index and receives from the oracle the symbol in that location in the string.

2.2 Definitions: Checkers, Testers and Correctors

Definition 2.1 (Checker)**.** A checker for a function f with (one-sided) error parameter $0 \leq \delta \leq 1$, is a probabilistic algorithm C with oracle access to a program oracle P that supposedly computes f. A program checker gives the following guarantees for *every* instance x:

1. (Completeness) If P computes f correctly (on every input), then $\Pr[C^P(x) = f(x)] = 1$.
2. (Soundness) For *any* P, $\Pr[C^P(x) \in \{f(x), \perp\}] > 1 - \delta$, where $\perp$ is a special error symbol, and where the probability is over the internal coin tosses of the checker C.

Throughout this work, the error parameter δ is $1/3$ unless we explicitly note otherwise.[12]

When considering distributions over instances of functions, it is convenient to consider separately distributions on instances of the same description length. We define:

Definition 2.2 (Distribution over instances)**.** A distribution D over instances from $\{0,1\}^*$, is an ensemble of probability distributions $D = \{D_n\}_{n\in\mathbb{N}}$, such that D_n is a distribution over $\{0,1\}^n$.

Definition 2.3 (Average-case error)**.** Let f be some function over $\{0,1\}^*$, P a program, and D a distribution over instances from $\{0,1\}^*$. We define $err_{f,P,D} : \mathbb{N} \to (0,1)$ as $err_{f,P,D}(n) = Pr_{x\leftarrow D_n}[P(x) \neq f(x)]$.

We say that P is δ-*good* for the function f with respect to D if $err_{f,P,D}(n) \leq \delta$ for every $n \in \mathbb{N}$.

Definition 2.4 (Tester)**.** Let $0 \leq \varepsilon_1 < \varepsilon_2 \leq 1$ and $0 < \delta < 1$. An $(\varepsilon_1, \varepsilon_2)$-*tester with error* δ *for a function* f *with respect to a distribution* D, is a probabilistic algorithm T that has an oracle access to a program P, such that the following holds. If $err_{f,P,D}(n) \leq \varepsilon_1$, $T^P(1^n)$ outputs "accept" with probability at least $1-\delta$. If $err_{f,P,D}(n) \geq \varepsilon_2$, $T^P(1^n)$ outputs "reject" with probability at least $1-\delta$. The default value of δ is $1/6$.

Definition 2.5 (Corrector)**.** A corrector for a function f is a probabilistic algorithm Cor, that has an oracle access to a program P. We say that,

1. P is *correctable by* Cor *with error* δ (for some $0 < \delta < 1$) if for every $x \in \{0,1\}^*$, $Pr[Cor^P(x) = f(x)] > 1 - \delta$ (where the probability is over the randomness of Cor).
2. Cor is an ε-*corrector with respect to* D *and with error* δ (for some $0 < \varepsilon < 1, 0 < \delta < 1$), if for every P for which $err_{f,P,D} \leq \varepsilon$, P is correctable with error δ.

The default value of δ is $1/6$.

Note that in all the definitions above, the error parameter δ can be reduced to be an arbitrarily small constant without increasing the depth of the checker/tester/corrector by more than a constant factor, assuming that we start with a δ that is bounded away from $1/2$ by a constant. Typically, the desired distance parameters $\varepsilon_1, \varepsilon_2$ from Definition 2.4 and ε from Definition 2.5 are constants that are bounded away from 0 and 1.

Definition 2.6 (Tester-Corrector Pair)**.** A tester-corrector pair for f with threshold $\delta > 0$, is a pair of probabilistic algorithms (T, Cor), such that there are constants $0 \leq \varepsilon_1 < \varepsilon_2 \leq 1$, $0 < \varepsilon < 1$ and a distribution D, such that T is an $(\varepsilon_1, \varepsilon_2)$-tester for f with respect to D, Cor is an ε-corrector for f with respect to D, and there is a promise that if the tester T accepts a program P with probability at least $1-\delta$ then P is correctable by Cor. The default value of δ is $1/3$.

Remark 2.7. *Note that the requirement in Definition 2.6 is that there exist* some *ensemble of distributions for which T is a tester and Cor is a corrector. This ensemble may be very unnatural or even not efficiently sampleable (although typically it will be). The mere existence of such an ensemble ensures that whenever T "thinks" that a program is good enough for the corrector to correct, this is indeed the case. Thus if our goal is ultimately to detect (with the tester) which*

[12] For $\mathcal{NC}^0$ checkers this error parameter can be reduced to any desired constant. For all the other checkers presented in this work (i.e. "$\mathcal{AC}^0$ and up"), the soundness can be made exponentially small.

programs are correctable and then correct them (using the corrector), we can construct testers and correctors that work properly with very peculiar distributions. The point is that once the tester decides that a program is correctable, then the corrector works properly on every input, regardless of the distribution with respect to which the tester came to this conclusion.

Remark 2.8. *Also note that we set the default value of δ in Definition 2.6 to be 1/3 while in Definition 2.4 it is 1/6. This gives robustness to the notion of tester-corrector pair: programs that are very close to the function should be accepted by the tester with very high probability, however even programs that are accepted with a decent probability (but not as high as really good programs) are correctable. I.e. the corrector is able to correct programs that the tester thinks are good but not with very high confidence. This robustness property is both natural and essential for proving useful program checking results such as constructing a program checker from a tester-corrector pair and proving the composition theorem for testers and correctors. Typically this property holds for natural testers and correctors.*

3 Composing Checkers, Testers and Correctors

In the introduction we stated the composition theorem for checkers and described the main ideas of the proof. In this section we prove that theorem and present the Composition Theorems for program checkers, testers and correctors. These theorems serve as the primary tools we use to improve the efficiency of these objects. The principle behind the Composition Theorems is simple: if the checker contains some functionality that can be accessed through the (potentially faulty) program's interface, and moreover this functionality is itself checkable (by a "more efficient" checker), then computing the functionality is delegated from the checker to the program. Every computation of the given functionality is replaced with a call to the program (via some reduction), and the program's answers are run through the simpler checker for this functionality. The same principle holds also for program testers and correctors (though the analysis is more involved).

3.1 Composing Program Checkers

We now restate more formally and prove Theorem 1.3. In the statement below, unless we state otherwise, the circuits involved are of *bounded* fan-in. So, for example, a reduction computable in depth d refers to a reduction that can be computed by a family of circuits (one for each input length) of depth d and bounded fan-in AND and OR gates. We will later discuss extensions.

Theorem 3.1 (Composition Theorem for Program Checkers, Theorem 1.3 restated)**.** *Let $L_{internal}$ and $L_{external}$ be two languages that satisfy the following conditions:*

1. *Hardness of the external language for the internal language:*

   There exists an efficient constant-depth (Turing) reduction from $L_{internal}$ to $L_{external}$.

2. *The internal language "helps" to check the external language:*

   $L_{external}$ has an efficient checker of depth $d_{external}$ with access to oracle $L_{internal}$ (note that by definition it also has oracle access to a program that allegedly computes $L_{external}$.)

3. *Checkability of the internal language:*

   $L_{internal}$ has an efficient checker of depth $d_{internal}$.

Then there exists an efficient checker for $L_{external}$, of depth $O(d_{external} \cdot d_{internal})$, with a single polynomial fan-in AND gate at the top.

Proof. We construct a checker C for $L_{external}$, starting from the checker C_{external} that uses oracle gates to $L_{internal}$, whose existence is guaranteed by Condition 2 of the theorem. We assume C_{external} has success probability at least $\frac{5}{6}$ (otherwise we amplify its success probability). Let $P_{external}$ be the program that C (and C_{external}) checks. Our goal is to replace every oracle call that C_{external} makes to $L_{internal}$ with a (probabilistic) circuit B, of depth $O(d_{internal})$, that computes the language $L_{internal}$ using oracle calls to $P_{external}$. We base B on the program checker for $L_{internal}$ (guaranteed by Condition 3). This checker expects to have oracle access to a program that allegedly computes $L_{internal}$ (and not $L_{external}$). To that end we define the program $P_{internal}$ as follows: on instance x of $L_{internal}$ run the reduction from $L_{internal}$ to $L_{external}$ given in Condition 1 to produce a instances $x_1, \ldots, x_q$ of $L_{extrenal}$. Then run $P_{extrenal}$ on $x_1, \ldots, x_q$, and return its answers to complete the computation of the reduction.

Specifically, the circuit B is constructed as follows.

The circuit B: On input x (an instance of $L_{internal}$), B runs C_{internal}, the checker for $L_{internal}$ given in Condition 3, with oracle access to $P_{internal}$. I.e. on every query y that C_{internal} makes to the program it checks, B runs the reduction from $L_{internal}$ to $L_{external}$, and then runs $P_{external}$ on the outputs of the reduction. The output of the reduction is then returned to C_{internal} as the answer to the query y. In this way B obtains the output of C_{internal} which gives a prediction regarding the membership of x in $L_{internal}$. B then runs $P_{internal}$ on x and compares its output with the answer of C_{internal}. If these two answers agree, then B's answer is the same as them, otherwise B outputs $\perp$. The total depth of B is indeed $O(d_{internal})$ (since the reduction from $L_{internal}$ to $L_{external}$ can be computed in constant depth).

The key properties of this construction are the following completeness and soundness.

- Completeness:

  If $P_{external}$ computes $L_{external}$ correctly (on every input), then so does $P_{internal}$. In this case both C_{internal} and $P_{internal}$ will agree on the correct answer (on every input) with probability 1, and B will be correct on every input.

- Soundness:

  - If $P_{internal}(x) = L_{internal}(x)$ then B cannot output $1 - L_{internal}(x)$. This is because it only gives a prediction regarding the membership of x if both $P_{internal}$ and C_{internal} agree on it (otherwise it returns $\perp$).
  - If $P_{internal}(x) \neq L_{internal}(x)$, the only event that will cause B to output $1 - L_{internal}(x)$ is if C_{internal} outputs $1 - L_{internal}(x)$. But this happens with probability at most $1/6$ by the fact that C_{internal} is a checker with this soundness (which we assume w.l.o.g.).

Given the construction of this circuit B, the checker C runs as follows (checking a program $P_{external}$):

1. Run the checker C_{external}, replacing every oracle call to $L_{internal}$ with a computation of the circuit B.
2. If any of B's runs returned the symbol $\perp$, then output $\perp$. This step is implemented using an AND gate of polynomial fan-in.
3. Otherwise, output the same answer as C_{external}.

Thus C is of total depth $O(d_{external} \cdot d_{internal})$, with a single polynomial fan-in AND gate. We now prove that C is indeed a program checker for the language $L_{external}$ (given the completeness and soundness properties of the circuit B).

Claim 3.2 (Completeness)**.** *If $P_{external}$ computes $L_{external}$ correctly (on every input), then for every x, $Pr[C(x) = L_{external}(x)] = 1$.*

Proof. When $P_{external}$ computes $L_{external}$ correctly, then by the completeness property of B, it perfectly simulates the oracle for $L_{internal}$ and then by the fact that C_{external} is itself a program checker (with oracle calls to $L_{internal}$), we conclude that C correctly outputs $L_{external}(x)$ on its input x. ∎

Claim 3.3 (Soundness)**.** *If $P_{external}$ does not compute $L_{external}$ correctly, then for every x, $Pr[C(x) = 1 - L_{external}(x)] \leq 1/3$*

Proof. We say that a program or a checker that attempts to decide a language L makes an error on an instance x if it declares that x is in L when it is not or vice versa. If the program outputs any other symbol (e.g. $\perp$) we do not consider this as an error.

We know that for every x, the probability that C_{external} makes an error, is bounded by $1/6$ when it is given oracle access to $L_{internal}$. C simulates C_{external} by replacing $L_{internal}$ with B. We want to bound the probability that B causes C_{external} to make an error that it would not have made had he given access to $L_{internal}$.

Fix random coins $\bar{r}$ for C_{external} on which it does not make an error (when given oracle access to $L_{internal}$). Look at the execution of C_{external} with these random coins. If during the execution, for every query y that C_{external} makes to $L_{internal}$, it holds that $P_{internal}(y) = L_{internal}(y)$, then by the first soundness property of B, replacing $L_{internal}$ with B never causes C_{external} to make an error; At the very most it causes it to replace a correct answer with $\perp$.

Otherwise, let y be the first query that C_{external} makes for which $P_{internal}(y) \neq L_{internal}(y)$. By the second soundness property of B, with probability at least $5/6$, B on input y, will output $\perp$, and thus will cause C to output $\perp$, i.e. C will not make an error.

We conclude, by the union bound over the errors of C_{external} and the errors of B, that C makes an error with probability at most $1/3$. ■

■

3.2 Extensions

We now present some useful extensions of the Composition Theorem for program checkers. In some of our applications, the checker for $L_{external}$ has access to additional oracles (beyond the program it checks and the oracle to $L_{internal}$). We point out that the theorem holds even with these additional oracle gates. More formally,

Lemma 3.4. *Let $L_{internal}$ and $L_{external}$ be two languages satisfying the conditions of Theorem 3.1. Let G be the set of gates used by the program checkers involved and the reduction. Then the program checker in the conclusion of the theorem has the same properties and it uses gates from the set G.*

This lemma allows us to iterate the Composition Theorem, by gradually removing oracles, or alternatively replacing one oracle with another. This will prove itself to be a very useful tool.

The additional unbounded fan-in AND gate at the top of the checker from Theorem 3.1, prevents us from using this lemma to construct checkers in $\mathcal{NC}^0$. We now show how it can be removed.

Lemma 3.5. *Let $L_{internal}$ and $L_{external}$ be two languages satisfying the conditions of Theorem 3.1, and furthermore, suppose there is a constant-depth reduction from the Parity function[13] to $L_{external}$. Then there exists an efficient checker for $L_{external}$, of depth $O(d_{external} \cdot d_{internal})$ (without the unbounded fan-in AND gate at the top).*

[13]Recall that the Parity function from $\{0,1\}^*$ to $\{0,1\}$ is defined as $Parity(b_1, \ldots, b_n) = \sum_{i=1}^{n} b_i$ with addition over $GF(2)$.

Proof. By the proof of Theorem 3.1, all we need to show is how to remove the unbounded fan-in AND gate. We do that by first replacing it with a constant number of unbounded fan-in Parity gates, and then use the Composition Theorem with Parity as the internal function.

Consider the following randomized reduction from AND to Parity: on n bits input to the AND function, $(b_1 \dots b_n)$, generate n uniformly random bits $(r_1, \dots, r_n)$, and compute $Parity(r_1 \cdot (1 - b_1), \dots, r_n \cdot (1 - b_n))$ (where all operations are over $GF(2)$). If the AND of the bits is 1 then this Parity is 0 with probability 1. On the other hand, if the AND is 0 then the parity is a uniformly random bit. If we repeat this a constant number of times, we can compute AND with an arbitrarily small constant probability of error by using a constant number of Parity gates (and further notice that this reduction can be done in constant depth).

So we have replaced the AND gate with a constant number of Parity gates without increasing the depth of the checker C. Next, we want to remove these Parity gates, and we do that by applying Theorem 3.1 on $L_{external}$ as the external language and Parity as the internal language. Condition 1 is given by the hypothesis of this lemma, Condition 2 holds by our construction, and Condition 3 is given by Lemma 4.9, where a constant depth (i.e. $\mathcal{NC}^0$) checker for the Parity function is presented. Note that here we composed only a constant number of oracle gates, and thus there is no need for another AND gate of large fan-in. ■

On Checking Functions versus Languages: Our Composition Theorem for checkers (as well as the ones for testers and correctors, see below) only considered checking *languages* (i.e. boolean functions). In many cases, and also within this work, we check, test and correct non-boolean functions. The Composition Theorems hold also for checkers, testers and correctors of non-boolean functions.

3.3 Composing Program Testers and Correctors

In this section we state and prove our composition theorem for testers and correctors. We begin by considering reductions between language and their influence on the average success probability of programs computing these languages.

Definition 3.6 ($(\varepsilon_1, \varepsilon_2)$-reduction)**.** Let L_1 and L_2 be two languages, and let D_1 and D_2 be ensembles of distributions on instances of L_1 and L_2 (respectively).

We say that a reduction R is an $(\varepsilon_1, \varepsilon_2)$-reduction from (L_1, D_1) to (L_2, D_2) if, when it is given oracle access to an ε_2-good program P_2 for L_2 with respect to D_2, R^{P_2} is an ε_1-good program for L_1 with respect to D_1.

Theorem 3.7. *Composition Theorem for Testers/Correctors*

Let $L_{internal}$ and $L_{external}$ be two languages. Suppose that there exist parameters (which may depend on the input length n) $0 \leq \alpha_1, \alpha_2, \alpha, \varepsilon_1, \varepsilon_2, \varepsilon, \beta \leq 1$, such that the following conditions hold:

1. *Hardness of the external language for the internal language:*

   There exists a distribution ensemble D and an efficient constant-depth (α, β)-reduction R from $L_{internal}$ with the uniform distribution[14] to $L_{external}$ with the distribution D.

[14]For simplicity we assume in all there conditions of the theorem that the reduction, testers and correctors are with respect to the uniform distribution. However, the proof can easily be carried through to general distributions (and indeed the theorem is used later with distributions other than uniform).

2. *The internal language "helps" to test/correct the external language:*

   $L_{external}$ has an efficient $(\varepsilon_1, \varepsilon_2)$-tester and ε-corrector, both have depth at most $d_{external}$, and use oracle gates to $L_{internal}$.

3. *Testability and Correctability of the internal language:*

   $L_{internal}$ has an efficient (α_1, α_2)-tester and α-corrector, both have at most depth $d_{internal}$ and they form a tester-corrector pair (with default threshold 1/3).

Then $L_{external}$ has a $(\frac{1}{2} \cdot \min(\varepsilon_1, \beta), \frac{1}{2} + \frac{\varepsilon_2}{2})$-tester and $(\frac{1}{2} \cdot \min(\varepsilon, \beta))$-corrector with respect to the distribution D' generated by sampling with probability $\frac{1}{2}$ from D and $\frac{1}{2}$ from the uniform distribution. The tester and corrector can be implemented by depth $O(d_{external} \cdot d_{internal})$ circuits. Finally, if the tester and corrector for $L_{external}$ in condition 2 are a tester-corrector pair with threshold δ, then the new tester and corrector for $L_{external}$ form a pair with threshold $\frac{11 \cdot \delta}{12}$.

The fan-in of the composed tester and corrector depends (logarithmically) on the number of oracle calls that the external tester and corrector (in Condition 2) make to $L_{internal}$. Thus, if the number of these oracle calls is constant, then the composed tester and corrector are of bounded fan-in.[15]

We now give a proof sketch, followed by a full proof.

Proof Sketch. The basic idea is similar to the checker composition theorem, but the analysis is more involved. We consider a distribution D' over the instances of $L_{external}$ which is the "average" (a convex combination) of the uniform distribution and the distribution D on $L_{external}$ instances from the reduction R.

The new corrector simulates the corrector for $L_{external}$ that uses $L_{internal}$ as an oracle. Whenever the corrector makes an oracle query x to $L_{internal}$, we do the following: run the corrector for $L_{internal}$ on x. For every query y that this corrector makes to a program that allegedly computes $L_{internal}$ (we do not have such a program), run the reduction R on y. Use $P_{external}$, the real program being checked (a program for $L_{external}$), to answer the query y (to $L_{external}$) made by the reduction. We think of this procedure of running R on an $L_{internal}$ instance using $P_{external}$ as an oracle as a "virtual program", $P_{internal}$, for $L_{internal}$.

Now, if $P_{external}$ solves $L_{external}$ correctly with high probability with respect to D', then it follows that the virtual program $P_{internal}$ solves $L_{internal}$ with high probability with respect to the uniform distribution on $L_{internal}$ instances (this follows from the way we constructed D', which guaranteed that $P_{external}$ solves $L_{external}$ correctly w.h.p. on in stances drawn from D). This means that with high probability, the corrector for $L_{internal}$ successfully corrects $P_{internal}$ on all the queries made by the corrector for $L_{external}$. It follows that with high probability, our new corrector is the same as running the old corrector for $L_{external}$ with a perfect oracle for $L_{internal}$, which is a good corrector by Condition 2 of the theorem.

Let us turn our attention to the tester for $L_{external}$. We could construct the new tester as we did in the corrector case. I.e., simulate the old tester, obtaining the answers for $L_{internal}$-queries via correcting the virtual program $P_{internal}$ (using the corrector for $L_{internal}$). When $P_{external}$ is good with respect to D' we will indeed accept it by an argument similar to the corrector case. This, however, would not work if $P_{external}$ is bad with respect to D'. In this case we have no guarantee on the behavior of $P_{internal}$ and thus no guarantee on its correctability. This means that potentially

[15] We use this fact to get testers and correctors in $\mathcal{NC}^0$.

we are simulating the tester with a bad $L_{internal}$-oracle, in which case all bets are off, and the tester may accept the bad program $P_{external}$ (even though it should reject it). To address this problem, we add a test that checks $P_{internal}$ (with respect to the uniform distribution), by running it through the tester for $L_{internal}$. If it rejects, then it follows that $P_{external}$ is bad with respect to D' and we correctly reject it. If it accepts, then it follows by hypothesis that the corrector for $L_{internal}$ indeed corrects $P_{internal}$ and we obtain a simulation identical to running the old tester for $L_{external}$ with a perfect oracle for $L_{internal}$, which is a good tester by Condition 2 of the theorem. ■

Full proof of Theorem 3.7. The proof follows ideas similar to the ones introduced in the proof of Theorem 3.1, but the analysis is different. To avoid separate notation for testers and correctors we make (w.l.o.g) the simplifying assumption that $\alpha = \alpha_1$ and $\varepsilon = \varepsilon_1$.

We construct a tester T and corrector Cor for $L_{external}$. Our starting point is (again) the tester and corrector for $L_{external}$, whose existence is guaranteed by Condition 2 of the Theorem. Let $P_{extrenal}$ be the program to be tested and corrected. Our goal is to replace every oracle call to $L_{internal}$ with a circuit B that computes the language $L_{internal}$ using oracle calls to $P_{external}$, has depth $O(d_{internal})$, and has the property that if $P_{external}$ is "good enough" on the distribution D' (from the proof statement), then (simultaneously) *all* of B's activations by T and Cor give the correct answers with probability at least $\frac{11}{12}$.

The Circuit B. Let Cor_{external} and T_{external} be the corrector and tester for $L_{external}$ (Condition 2), and Cor_{internal} and T_{internal} be the corrector and tester for $L_{internal}$ (Condition 3). Let $p(n) \leq poly(n)$ be a bound on the number of oracle calls Cor_{external} and T_{external} make to $L_{internal}$.

B simulates Cor_{internal} $O(\log(p(n)))$ times in parallel, and outputs the majority of these simulations' answers. Note that B cannot directly activate Cor_{internal} because it requires access to a program that allegedly computes $L_{internal}$, whereas B only has access to $P_{external}$ (that allegedly computes $L_{external}$). Thus B *simulates* runs of Cor_{external} using the reduction R from $L_{internal}$ to $L_{external}$, whose existence is guaranteed by Condition 1 of the theorem. During the simulation of Cor_{internal}, whenever it makes a call to its program (i.e. queries an instance of $L_{internal}$), B runs R on that instance with $P_{external}$ as its oracle, and answers as R does. Thus the total depth of B is indeed $O(d_{internal})$, and it uses majority gates of fan-in $O(\log(p(n))) = O(\log n)$ (we note that actually B only needs to compute *approximate* majority, see [AB84]). In the case that the external tester and corrector make only a constant number of $L_{internal}$ oracle calls, these majorities are only on a constant number of items and thus can be replaced by $\mathcal{NC}^0$ circuits.

Next we argue that B computes $L_{internal}$ correctly with high probability.

Claim 3.8. *If $P_{external}$ is $(\frac{1}{2} \cdot \min(\varepsilon_1, \beta))$-good for $L_{external}$ with respect to the distribution D' then for every $x \in \{0,1\}^n$,*

$$\Pr[B(x) \neq L_{internal}(x)] \leq \frac{1}{12 \cdot p(n)}$$

Proof. B runs Cor_{internal} which corrects programs that are α_1-good for $L_{internal}$ with respect to the uniform distribution. B uses a "program" for $L_{internal}$ that is obtained by running the reduction R using $P_{external}$ as its oracle . Since $P_{external}$ is $(\frac{1}{2} \cdot \min(\varepsilon_1, \beta))$-good for $L_{external}$ with respect to the distribution D', it is also β-good for $L_{external}$ with respect to the distribution D (by the construction of D'). We know that when R is run with a β-good program for $L_{external}$ as its oracle , it gives an α_1-good program for $L_{internal}$ with respect to the uniform distribution.

By the fact that Cor_{internal} corrects programs that are α_1-good with respect to the uniform distribution, on every instance of $L_{internal}$, every execution of Cor_{internal} by B gives a correct answer with probability at least $\frac{5}{6}$. By taking the majority of $O(\log(p(n)))$ executions of Cor_{internal}, the probability that B errs is at most $\frac{1}{12 \cdot p(n)}$. ■

The Corrector Cor**.** The corrector Cor runs the "old" corrector Cor_{external}, replacing every oracle call to $L_{internal}$ with a call to the circuit B. Thus Cor has depth $O(d_{external} \cdot d_{internal})$ and the only gates of unbounded fan-in are the majority gates used by B, which have fan-in $O(\log(p(n)))$.

Claim 3.9. *Cor is a $(\frac{1}{2} \cdot \min(\varepsilon_1, \beta))$-corrector for $L_{external}$ with respect to the distribution D'.*

Proof. Let $P_{external}$ be $(\frac{1}{2} \cdot \min(\varepsilon_1, \beta))$-good with respect to the distribution D'. By the construction of D', it follows that $P_{external}$ is ε-good with respect to the uniform distribution (recall that $\varepsilon = \varepsilon_1$). This means that Cor_{external} computes every instance of $L_{external}$ correctly with probability at least $\frac{5}{6}$ when given oracle access to $L_{internal}$ and $P_{external}$. When Cor simulates Cor_{external}, it replaces calls to $L_{internal}$ by an execution of B. By Claim 3.8, B gives a correct answer with probability at least $1 - \frac{1}{12 \cdot p(n)}$ on every execution. Since Cor runs B at most $p(n)$ times, by union bound, B is correct on all the executions with probability at least $\frac{11}{12}$. We conclude that for every instance of $L_{external}$, Cor computes it correctly with probability at least $\frac{2}{3}$. ■

The Tester T**.** Let $P_{external}$ be the program to be tested. The tester T runs as follows:

1. Repeat the following c times (c is a constant that will be determined later): Run the internal tester T_{internal}, replacing each oracle call to its program, by executing the reduction R on the $L_{internal}$ instance, and using $P_{external}$ as the oracle for the reduction.

   If T_{internal} rejects in more than a $\frac{1}{4}$-fraction of the c executions, reject immediately, otherwise proceed to the next step.

2. Run the external tester T_{external}, replacing every one of its oracle calls to $L_{internal}$ by running the circuit B. Output whatever answer this simulation of T_{external} gives.

T has depth $O(d_{external} \cdot d_{internal})$ and the only gates of unbounded fan-in are the majority gates used by B, which have fan-in $O(\log(p(n)))$. Next we prove that T is a $(\frac{1}{2} \cdot \min(\varepsilon_1, \beta), \frac{1}{2} + \frac{\varepsilon_2}{2})$-tester for $L_{external}$ with respect to the distribution D' (this is established in Claims 3.10 and 3.12 below) and that together with Cor it forms a tester-corrector pair, assuming the original tester and corrector formed a pair (this is established in Claim 3.13).

Claim 3.10. *T accepts programs that are $(\frac{1}{2} \cdot \min(\varepsilon_1, \beta))$-good w.r.t. the distribution D', with probability at least $\frac{2}{3}$.*

Proof. Assume $P_{external}$ is $(\frac{1}{2} \cdot \min(\varepsilon_1, \beta))$-good w.r.t. the distribution D', thus it is also β-good w.r.t. the distribution D. It follows that when $P_{external}$ is used as the oracle of the reduction R, we get a α_1-good program for $L_{internal}$. So in the first step, in each execution, T_{internal} rejects with probability at most $\frac{1}{6}$. By the Chernoff bound, if we take c to be a large enough constant, with probability at least $\frac{11}{12}$, not more than $\frac{1}{4}$ fraction of the c executions will reject and we will proceed to the next step.

From the construction of D', it follows that $P_{extrenal}$ is also ε_1-good with respect to the uniform distribution on $L_{external}$ instances. T_{external} is an $(\varepsilon_1, \varepsilon_2)$-tester with success probability $\frac{5}{6}$, and thus when B gives correct answers in *all* its executions, T_{external} accepts with probability at least $\frac{5}{6}$, even if we replace its $L_{internal}$ oracle with B. By Claim 3.8 and the union bound, the probability that B indeed gives the correct answer in all its executions (in the second step) is at least $\frac{11}{12}$.

We conclude (by taking a union bound over the error probabilities) that T accepts $P_{external}$ with probability at least $\frac{2}{3}$. ■

Claim 3.11. *Let P' be a program (supposedly computing $L_{internal}$) obtained by running the reduction R on instances of $L_{internal}$ and taking $P_{external}$ to be the oracle for the reduction. If P' is not correctable by Cor_{internal}, then T rejects in its first step with probability at least $\frac{11}{12}$.*

Proof. T_{internal} and Cor_{internal} are a tester-corrector pair with threshold 1/3 (the default value). So if P' is not correctable by Cor_{internal} it must hold that T_{internal} rejects P' in each one of its executions (in the first step of T) with probability at least 1/3. By Chernoff bound, if we take c to be a large enough constant, the probability that more than 1/4 of the executions reject is at least $\frac{11}{12}$. ■

Claim 3.12. *T rejects programs that are* not *$(\frac{1}{2}+\frac{\varepsilon_2}{2})$-good w.r.t. the distribution D', with probability at least $\frac{2}{3}$.*

Proof. First, if the internal corrector Cor_{internal}, when it is run with the program (where answers are computed via the reduction R), does not give a correct answer (on every input) with high probability (say more than 5/6), then the program is rejected (in the first step) with probability at least $\frac{11}{12}$ (see Claim 3.11).

If Cor_{internal} does compute $L_{internal}$ correctly with high probability, then with probability $\frac{11}{12}$ all of B's executions in the second step give correct answers. Now observe that a program that is not $(\frac{1}{2} + \frac{\varepsilon_2}{2})$-good w.r.t the distribution D', is also not ε_2-good w.r.t. the uniform distribution on $L_{external}$ inputs. When B always gives correct answers, T_{external} will reject with probability at least $\frac{5}{6}$. Taking a Union Bound over all error probabilities, we conclude that a program that is not $(\frac{1}{2} + \frac{\varepsilon_2}{2})$-good w.r.t. D' is rejected with probability at least $\frac{2}{3}$. ■

Claim 3.13. *If T_{external} and Cor_{external} form a tester-corrector pair with threshold δ (for some $0 < \delta < 1/2$), then T and Cor form a pair with threshold $\delta' = \frac{11 \cdot \delta}{12}$.*

Proof. If T accepts a program $P_{external}$ with probability at least $1 - \delta' > 1/12$ then in particular it passes the first step with this probability. This implies by Claim 3.11 that the program P (for $L_{internal}$) obtained by running the reduction R with $P_{external}$ as its oracle , is correctable by Cor_{internal}.

If the above program P is correctable by Cor_{internal}, then with probability at least $\frac{11}{12}$ all of B's executions in the second step give correct answers. Condition on the event that all the calls to B give correct answers. (Which means that the behavior of T_{external} with oracle to B is identical to its behavior with oracle to $L_{internal}$.) The probability that T rejects in this conditional space is at most $\frac{12 \cdot \delta'}{11} = \delta$. By the fact that T_{external} and Cor_{external} form a tester-corrector pair with threshold δ (when they are given oracle access to $L_{internal}$) it follows that $P_{external}$ is correctable by Cor_{external} when the latter is given oracle access to $L_{internal}$. By replacing oracle calls to $L_{internal}$ with calls to B we increase the error probability of Cor_{external} by at most $\frac{1}{12}$. ■

■

We now present a useful claim for quantifying the parameters of reductions between language-distribution pairs.

Claim 3.14. *Let L_1 and L_2 be two languages such that there exists a non-adaptive (Turing) reduction R from L_1 to L_2 that makes at most q queries to L_2. Then for any ε_1 and distribution D_1, there exists a distribution D_2 such that R is an $(\varepsilon_1, \frac{\varepsilon_1}{q^2})$-reduction from (L_1, D_1) to (L_2, D_2).*

Proof. The distribution D_2 is obtained by picking a random L_1 instance using D_1, computing R's q (non-adaptive) queries (these are L_2 instances), and outputting one of them uniformly at random.

Consider each of the q distributions on the q queries that R makes when run on a random sample from D_1. Any program P that is $\frac{\varepsilon_1}{q^2}$-good for L_2 with respect to D_2 is also $\frac{\varepsilon_1}{q}$-good on each of these distributions. Now if we run the reduction R on a random instance sampled by D_1, P answers each of R's queries correctly with probability at least $1 - \frac{\varepsilon_1}{q}$. Taking a Union Bound, the probability that P makes an error on at least one of the q queries made by the reduction R is at most ε_1. We conclude that if P is $\frac{\varepsilon_1}{q^2}$-good for L_2 with respect to D_2, then R using P as its oracle is ε_1-good for L_1 w.r.t. D_1. ■

Remark 3.15. *The above claim implies that whenever the languages of the composition theorem have a reduction that only makes a constant number of oracle calls, the β parameter in the composition theorem is constant, and so are the composed tester and corrector's distance parameters.*

In most cases where we use the Composition Theorem this will be the case. In fact, whenever we refer to using the Tester/Corrector Composition Theorem in this work we implicitly refer to using it with such reductions, using Claim 3.14. The only exceptions occur in Section 6, where we use the composition theorem with languages between which there do not appear to be reductions that only make a constant number of queries. In these cases we construct amplified reductions *for the specific pairs of languages at hand (see Claims 6.6,6.10).*

Remark 3.16. *When using Theorem 3.7 we often apply it recursively. Namely, we compose some external language with an internal one, and then use the external language with its new tester/corrector, as an internal language to be composed with a tester/corrector of an even more complex language (which now plays the role of the external language). We therefore want to point out what happens to the parameters of the testers/correctors after the composition, in order to make sure we can then use the new testers/correctors in the next application of the theorem.*

When all the parameters involved in the statement of Theorem 3.7 are constants (i.e. the ε's, the α's and β), and the distance parameters of the original testers/correctors (i.e. the ε's and the α's) are bounded away from 0 and 1, then the distance parameters of the composed tester/corrector are all constants that are bounded away from 0 and 1.

Also, we choose the threshold for the tester-corrector pair of the internal language to be the default value 1/3. We would like to point out that any threshold that is bounded away (from above) from 1/4 by a constant will do. Also note that if the threshold for the original tester-corrector pair of the external language is δ then the threshold for the new pair is $(1 - \lambda) \cdot \delta$ with $\lambda = 1/12$. We want to point out that λ can be set to be an arbitrarily small constant.

We conclude that if we apply Theorem 3.7 recursively a constant number of times (each time using the external language from the previous round as an internal language), then we end up with testers/correctors that have distance parameters that are bounded away by constants from 0 and

1, assuming that the testers/correctors along the way have such distance parameters and that the reductions along the way use a constant number of queries.

4 Checkers, Testers and Correctors for Complete Languages

In this section we show several languages that are complete for certain complexity classes, and have very efficient checkers, testers, and correctors. In particular we prove Theorem 1.4.

4.1 Randomized images

We start by defining the properties of functions and languages that we need for our approach. The first property says that we can easily generate a random instance together with the evaluation of the function on this input.

Definition 4.1 (Solved instance generator)**.** Let $f : \{0,1\}^* \to \{0,1\}^*$ be a function. We say that a randomized algorithm G is a *solved instance generator* for f if, given 1^n, it generates a pair (x, y), where x is a uniformly random element of $\{0,1\}^n$ and $y = f(x)$.

The second property is a reduction from one function to another that says, roughly, that we can evaluate the first function on every instance by querying the second function in a random location.

Definition 4.2 (Random instance reduction)**.** Let $f : \{0,1\}^* \to \{0,1\}^*$ and $g : \{0,1\}^* \to \{0,1\}^*$ be two functions. We say that a pair of algorithms $(\mathcal{R}, \mathcal{E})$ is a *random instance reduction* from f to g if $\mathcal{R}$ is a randomized algorithm that given $x \in \{0,1\}^n$, generates a pair (x', τ), where x' is a uniformly random element of $\{0,1\}^{m(n)}$ and $\tau \in \{0,1\}^*$ and it holds that $\mathcal{E}(g(x'), \tau) = f(x)$.

If $m(n) = n$ we say that the random instance reduction is *length-preserving*. If f and g are the same function, we say that it is a random instance self-reduction.[16] We call $\mathcal{R}$ the *Randomizer* and $\mathcal{E}$ the *Evaluator*.

The objects that we will be interested in are pairs of functions that have the above two properties.

Definition 4.3 (Randomized image)**.** Let $f : \{0,1\}^* \to \{0,1\}^*$ and $g : \{0,1\}^* \to \{0,1\}^*$ be two functions. We say that g is a *randomized image* of f, if there is a random instance reduction from f to g, and g has a solved instance generator.

We say that it is length-preserving if the random instance reduction is length-preserving, and that it is a randomized self-image if $f = g$. Finally, we say that the randomized image can be implemented in some complexity class $\mathcal{C}$, if the algorithms $G, \mathcal{R}$ and $\mathcal{E}$ (from Definitions 4.1 and 4.2) can be implemented in this class.

Next we present several languages that have extremely efficient randomized images. In the sequel we will abuse the term by saying that a language has a randomized image, meaning that its characteristic function has a randomized image. Our constructions use techniques that were developed in the field of cryptography, with some appropriate modifications.

One important family of functions that have randomized images are word problems over finite groups.

Definition 4.4. Let $(G, \odot)$ be a group. We define the word problem of G to be the function $L_G : G^* \to G$, where $L_G(a_1, \ldots, a_n) = a_1 \odot a_2 \odot \cdots \odot a_n$.

[16] Random instance self-reductions are a special form of what is called in the literature *random self-reductions*. The word instance in our terminology, should emphasize the fact that the reduction is from one instance to another (random) instance. General random self-reductions can make many *self*-queries to the function in order to compute its value on a given instance.

Claim 4.5. *Let $(G, \odot)$ be a finite group. Then L_G has a length-preserving randomized self-image that can be implemented by $\mathcal{NC}^0$ circuits given that they can sample random elements from G.*

Proof. Our constructions are based on a randomization technique from [Bab87, Kil88]. Details follow.

Solved instance generator: We would like to sample a random instance $x \in G^n$ together with $y = L_G(x)$. Given 1^n and a random string $\bar{a} = (a_1, \ldots, a_n) \in_R G^n$, define x to be $(a_1, a_1^{-1} \odot a_2, a_2^{-1} \odot a_3, \ldots, a_{n-1}^{-1} \odot a_n)$, and y to be a_n. Since $a_1, \ldots, a_n$ are independently and uniformly distributed, so are $a_1, a_1^{-1} \odot a_2, a_2^{-1} \odot a_3, \ldots, a_{n-1}^{-1} \odot a_n$. Clearly, $L_G(x) = a_1 \odot a_1^{-1} \odot a_2 \odot a_2^{-1} \odot a_3 \cdots a_{n-1}^{-1} \odot a_n = a_n$. Finally, note that every element in x' is a function of two elements in $\bar{a}$, therefore the procedure can be implemented by an $\mathcal{NC}^0$ circuit over the alphabet G.

Random instance self-reduction: The randomizer $\mathcal{R}$, given $x \in G^n$ and a random string $\bar{a} = (a_1, \ldots, a_n) \in_R G^n$, outputs $x' \in G^n$ which is $(x_1 \odot a_1, a_1^{-1} \odot x_2 \odot a_2, a_2^{-1} \odot x_3 \odot a_3, \ldots, a_{n-1}^{-1} \odot x_n \odot a_n)$, and τ which is a_n^{-1}. Define $\mathcal{E}(\sigma, \tau) = \sigma \odot \tau$. Since $a_1, \ldots, a_n$ are independently and uniformly distributed, so are $x_1 \odot a_1, a_1^{-1} \odot x_2 \odot a_2, a_2^{-1} \odot x_3 \odot a_3, \ldots, a_{n-1}^{-1} \odot x_n \odot a_n$. Clearly, $\mathcal{E}(f(x'), a_n^{-1}) = x_1 \odot a_1 \odot a_1^{-1} \odot x_2 \odot a_2 \odot a_2^{-1} \odot x_3 \odot a_3 \cdots a_{n-1}^{-1} \odot x_n \odot a_n \odot a_n^{-1} = x_1 \odot x_2 \odot \cdots \odot x_n = L_G(x)$. Finally, note that every element in y is a function of one element in x and two elements in $\bar{a}$, therefore $\mathcal{R}$ can be implemented by an $\mathcal{NC}^0$ circuit over the alphabet G ($\mathcal{E}$ is over a finite domain so it is clearly in $\mathcal{NC}^0$). ∎

Corollary 4.6. *The parity function: $Parity(x_1, \ldots, x_n) = \sum_{i=1}^{n} x_i$ where $x_i \in \{0, 1\}$ and the sum is over $GF(2)$ has a length-preserving randomized self-image that can be implemented by $\mathcal{NC}^0$ circuits.*

Barrington [Bar89] has shown that L_{S_5} is complete for the class $\mathcal{NC}^1$ under $\mathcal{NC}^0$ reductions[17] (S_5 is the symmetric group over five elements). We conclude:

Corollary 4.7. *There is an $\mathcal{NC}^1$-complete function under $\mathcal{NC}^0$ reductions that has a length-preserving randomized self-image that can be implemented by $\mathcal{NC}^0$ circuits* that are given access to a source of random elements in S_5.

In particular, using the fact that a randomized $\mathcal{AC}^0$ circuit can sample uniformly from the group S_5 (with a small probability of failure), we conclude that the randomized self-image can be implemented by $\mathcal{AC}^0$ circuits.

4.2 From randomized self-images to program checkers

Next we prove that functions with efficient randomized self-images also have efficient checkers, testers and correctors.

Theorem 4.8. *Let f be a function with a solved instance generator that can be computed by bounded fan-in circuits of depth d, and a random instance self-reduction where the randomizer and evaluator can both be computed by bounded fan-in circuits of depth d.*

Then f has a $(\frac{1}{3}, \frac{2}{3})$-tester-corrector pair and a program checker, where all of them can be implemented by bounded fan-in circuits of depth $d + O(1)$.

[17] While [Bar89] only considers $\mathcal{AC}^0$ reductions, it is clear that the reduction is in fact $\mathcal{NC}^0$ as every element of S_5 in the resulting word problem depends on exactly one input bit of the original instance.

Proof. The tester runs the solved instance generator to generate a solved pair (x, y), where x is uniformly distributed and $f(x) = y$. It then runs the program P on x and accepts if and only if $P(x) = y$.

The corrector receives an input x and runs the randomizer to generate a pair (y, τ) where y is uniformly distributed. It then runs P on y, and uses the evaluator on input $(P(y), \tau)$. If $P(y) = f(y)$ then this recovers $f(x)$. The distance parameters and soundness of the tester and corrector can be amplified by repeating the above procedures $O(1)$ times in parallel.

The program checker runs the tester O(1) times in parallel and also runs the corrector O(1) times in parallel on its input. If the tester rejects in even one of its runs it outputs BUG, otherwise it outputs the majority answer of the corrector's answers.

The correctness and soundness of the tester, corrector and checker follow directly from the properties of solved instance generator and random instance self-reduction.

■

As with the Composition Theorems, if the solved instance generator or the random instance self-reduction require additional gates (beyond bounded fan-in AND, OR and NOT gates) then the Theorem still holds with checkers, testers, and correctors that use these gates.

4.3 Checkers, testers and correctors for complete languages

We now apply Theorem 4.8 to obtain extremely efficient checkers, testers and correctors for complete languages.

Lemma 4.9. *The Parity function has a $(\frac{1}{3}, \frac{2}{3})$-tester, $\frac{1}{3}$-corrector and a program checker all implementable in $\mathcal{NC}^0$.*

Proof. The proof is immediate from Theorem 4.8, as the parity language has an $\mathcal{NC}^0$ solved instance generator and random instance self-reduction (see Corollary 4.6). ■

Note that the parity function is a linear function, and in fact the results of [BLR93] give an NC^0 tester and corrector for the parity function. Our tester and corrector make fewer calls to the program being checked: they each only make a single query (as opposed to 3 and 2 respectively in the tester and corrector of [BLR93]). Next we prove the following lemma:

Lemma 4.10. *There is an $\mathcal{NC}^1$-complete language under $\mathcal{NC}^0$ reductions that has an $\mathcal{NC}^0$ program checker and an $\mathcal{AC}^0$ $(\frac{1}{3}, \frac{2}{3})$-tester and $\frac{1}{3}$-corrector pair.*

Proof. Applying Corollary 4.7 and Theorem 4.8 (with $\mathcal{AC}^0$ gates), we obtain a checker, tester and corrector that are implementable in $\mathcal{AC}^0$ for the $\mathcal{NC}^1$-complete language L_{S_5}. They contain unbounded fan-in AND gates which are used to sample uniform elements in S_5, the rest of the computations can be done with bounded fan-in gates.

In order to obtain checkers in $\mathcal{NC}^0$, we use the Composition Theorem for checkers (Theorem 3.1). We replace the AND and OR gates with Parity gates as it is done in the proof of Lemma 3.5. We then use the Composition Theorem to remove the Parity gates (again, we refer the reader to the proof of Lemma 3.5). ■

We now show checkers, testers and correctors for languages that are complete for classes higher than NL (nondeterministic log-space). We start with the definitions of these classes and the complete languages for them.

Definition 4.11. The class $\oplus$-$\mathcal{L}$ contains all the languages that are decidable by a nondeterministic log-space Turing machine with the acceptance criteria that the number of accepting paths is even.

Definition 4.12. The class mod_k-$\mathcal{L}$ (for an integer $k > 1$) contains all the languages that are decidable by a nondeterministic log-space Turing machine with the acceptance criteria that the number of accepting paths is zero modulo k.

Definition 4.13. The language $\oplus$-connectivity is the language of tuples (G, s, t), such that G is a directed graph containing the vertices s and t, and the number of paths from s to t is even.

Definition 4.14. The language mod_k-connectivity (for an integer $k > 1$) is the language of tuples (G, s, t), such that G is a directed graph containing the vertices s and t, and the number of paths from s to t is zero modulo k.

Theorem 4.15. *$\oplus$-$\mathcal{L}$ and mod_k-$\mathcal{L}$ are complete for $\oplus$-connectivity and mod_k-connectivity respectively under $\mathcal{NC}^0$ reductions.*

We proceed with presenting checkers, testers and correctors for these languages.

Corollary 4.16. *$\oplus$-connectivity has an $\mathcal{NC}^0$ program checker, and an $\mathcal{AC}^0$ $(\frac{1}{3}, \frac{2}{3})$-tester and $\frac{1}{3}$-corrector pair.*

Proof. The work [IK02] shows that $\oplus$-connectivity has a solved instance generator and random instance self-reduction in $\mathcal{NC}^0[\oplus]$. This immediately gives a checker, tester and corrector in $\mathcal{NC}^0[\oplus]$ (by Theorem 4.8); i.e. they are in $\mathcal{NC}^0$ with unbounded fan-in parity oracle gates. We will use the Composition Theorems to "collapse" the checker to $\mathcal{NC}^0$ and the tester and corrector to $\mathcal{AC}^0$. Taking the $\oplus$-connectivity language to be the "external" language, and the parity language as the "internal" language, the three conditions of the Composition Theorems (Theorems 3.1 and 3.7) hold:

1. Hardness of the external language for the internal language:

   Clearly given an oracle to the $\oplus$-connectivity language, one can compute parities of vectors. The simple Karp reduction is in $\mathcal{NC}^0$(e.g. using the $\oplus$-$\mathcal{L}$-machine for computing the parity language).

2. The internal language helps to check/test/correct the external language:

   This is simply because is was shown above how to construct a checker, tester and corrector for $\oplus$-connectivity that are in $\mathcal{NC}^0[\oplus]$.

3. Check/test/correct-ability of the internal language:

   By Lemma 4.9.

By the Composition Theorems (Theorems 3.1 and 3.7) we get a tester and a corrector for $\oplus$-connectivity in $\mathcal{AC}^0$, and a program checker in $\mathcal{NC}^0$ (note that for this we use the fact that $\oplus$-connectivity is hard for Parity under $\mathcal{NC}^0$ reductions, as required by Lemma 3.5). ■

Corollary 4.17. *Let k be a prime. Then* mod_k*-connectivity has an* $\mathcal{NC}^0$ *program checker, and an* $\mathcal{AC}^0$ $(\frac{1}{3}, \frac{2}{3})$*-tester and* $\frac{1}{3}$*-corrector pair.*

Proof. The work of [IK02] shows that mod_k-connectivity has a solved instance generator and random instance self-reduction in $\mathcal{AC}^0$ with oracle gates for multiplication over $GF[k]$ and for addition of n numbers over $GF[k]$. By Theorem 4.8 this gives a checker, tester, and corrector for mod_k-connectivity that are in $\mathcal{AC}^0$ with oracle gates for multiplication over $GF[k]$ and for addition of n numbers over $GF[k]$. Note that the checker, tester and corrector also need to generate (almost) random field elements, but this can be done in $\mathcal{AC}^0$.

We obtain a checker, tester, and corrector in $\mathcal{AC}^0$ using the Composition Theorems (Theorems 3.1 and 3.7). We first note that both multiplication over $GF[k]$ and addition of n numbers over $GF[k]$ are reducible (under $\mathcal{NC}^0$ reductions) to mod_k-connectivity . These two functions are themselves both checkable in $\mathcal{AC}^0$: a checker/tester/corrector for multiplication was given by [BLR93], the checker/tester/corrector for adding n numbers $\mod k$ is similar to the one for Parity given in Lemma 4.9. All these checkers/testers/correctors need to be able to add two numbers over $GF[k]$ and to generate random field elements, both of these can be done in $\mathcal{AC}^0$. This gives an $\mathcal{AC}^0$ checker,tester, and corrector for mod_k-connectivity .

To further obtain a program checker in $\mathcal{NC}^0$, we use the checker Composition Theorem (Theorem 3.1). This uses the fact that both AND and Parity are reducible to mod_k-connectivity under $\mathcal{NC}^0$ reductions. ■

5 Checkers, Testers and Correctors for a Complexity Class

In this section we use the Composition Theorems to prove Theorem 1.1. We begin with a proof sketch for the theorem followed by a discussion. We then present and prove fully a more general theorem and conclude with a proof of Theorem 1.1. The generalized theorem (Theorem 5.1 in this section) uses the composition methodology to obtain checkers whose depth is related to the depth of the circuits that compute the language being checked. As with the Composition Theorems, all the circuits involved here contain only bounded fan-in gates, unless stated otherwise.

Proof sketch for Theorem 1.1. Let L be a language in $\mathcal{RNC}^{\mathsf{i}}$ that is $\mathcal{NC}^{\mathbf{1}}$-hard under $\mathcal{NC}^{\mathbf{0}}$-reductions. To build the $\mathcal{RNC}^{\mathsf{i-1}}$ checker for L, we begin with the trivial checker for L, i.e. simply a (correct) $\mathcal{RNC}^{\mathsf{i}}$ circuit for L. We then decompose this circuit C into $O(\log^{i-1} n)$ "layers" of depth $\log n$ each. Put another way, we view this checker as a circuit of depth $O(\log^{i-1} n)$ that has oracle gates for evaluating depth-$\log n$ sub-computations. (Specifically, the oracle gates take as input a circuit D of depth $\log n$ and an input x to this circuit, and output $D(x)$.) Since this oracle computes an $\mathcal{NC}^{\mathbf{1}}$ language, we can replace it with an oracle that computes the $\mathcal{NC}^{\mathbf{1}}$-complete language from Theorem 1.4 (without paying more then a constant factor in depth). We now use the Composition Theorem to eliminate these oracle gates. We take the $\mathcal{NC}^{\mathbf{1}}$-complete language accepted by the oracle gates to be our internal language $L_{internal}$, and take L to be $L_{extrenal}$. Clearly, all the conditions of the theorem hold: (a) $L_{internal}$ is in $\mathcal{NC}^{\mathbf{1}}$, and therefore is reducible to L (which we assumed is $\mathcal{NC}^{\mathbf{1}}$-hard), (b) $L_{extrenal}$ has a checker of depth $O(\log^{i-1} n)$ that uses oracle calls to $L_{internal}$, and (c) $L_{internal}$ has a $\mathcal{NC}^{\mathbf{0}}$ checker (Theorem 1.4). We therefore conclude that L has an $\mathcal{RNC}^{\mathsf{i-1}}$ checker. A similar approach (based on the Composition Theorem for testers/correctors) yields the statement regarding testers and correctors. ■

Using a Correct Program to Construct Checkers of Correctness. The idea used in the proof of Theorem 1.1 is to build an efficient checker by starting with a correct program for the function. At first, this may seem strange: if we have a correct program, what do we need a checker for? The answer is that the checker will check *all programs* for this function, including programs which may have more desirable features than the correct one we used for the design of the checker. In fact, we find the idea of starting with a correct program for the function as a way to design checkers is in itself interesting. In practice, *testing* software for correctness is often done by comparing on well-chosen test cases to an existing correct (although possibly very slow) program. What we show here is that the approach of starting with a correct program is beneficial also for designing efficient checkers. Moreover, it provides a first handle on the design of a checker for a function without "nice" structural properties (i.e., for which all that is known is some program defining it).

Theorem 5.1 (Generalization of Theorem 1.1)**.** *Let L be a language computable by circuits of depth d, and let $d_{collapse} > 0$ be some integer (that could be a function of the input length n). If there exists a language $L_{internal}$ such that:*

1. *(L is "harder" than $L_{internal}$) There is a constant depth reduction from $L_{internal}$ to L.*
2. *($L_{internal}$ is complete for depth $d_{collapse}$ computations) There is a constant depth reduction from any language computable by circuits of depth $d_{collapse}$ to $L_{internal}$.*

3. ($L_{internal}$ is checkable) $L_{internal}$ has a constant-depth checker/tester-corrector.

Then L has a depth $O(d/d_{collapse})$ checker with a single unbounded fan-in AND gate at the top, and depth $O(d/d_{collapse})$ tester-corrector with (possibly many) unbounded fan-in gates.

Proof. The proof follows from Theorems 3.1 (for programs checkers) and 3.7 (for program testers and correctors). Details follow.

Program Checkers: Take L to be the "external" language, and $L_{internal}$ to be the "internal" language. We show that the three conditions of Theorem 3.1 are satisfied:

- Hardness of the external language for the internal language:

  This condition is immediately satisfied by Condition 1 of the theorem. Note that the depth of this reduction is indeed constant.

- The internal language "helps" to check the external language:

  To see this, observe that a trivial program checker for any language is the circuit that correctly computes that language (and ignores the program oracle). Starting with such a circuit C for L, we can use oracle gates for $L_{internal}$, together with the fact that $L_{internal}$ is hard for depth $d_{collapse}$ circuits, to construct an efficient depth $O(d/d_{collapse})$ checker for L that uses oracle gates to $L_{internal}$. This checker divides the circuit C into $O(d/d_{collapse})$ layers, each of depth $d_{collapse}$. It then "collapses" each such layer to constant depth with an oracle call to $L_{internal}$ (this can be done using the constant-depth reduction from any depth-$d_{collapse}$ computation to $L_{internal}$ guaranteed by Condition 2 of the theorem).

  The depth of this new checker is $O(d/d_{collapse})$. In the terms of Theorem 3.1, this is the value $d_{external}$.

- Checkability of the internal language:

  This condition is immediately satisfied by Condition 3 of the theorem. The depth of the checker for the internal language is constant, and thus in the terms of Theorem 3.1 $d_{internal} = O(1)$.

Now we apply Theorem 3.1, and we conclude that L has a depth $O(d_{external} \cdot d_{internal}) = O(d/d_{collapse})$ checker with a single unbounded fan-in AND gate at the top.

Testers and Checkers: Using a similar observation, that a circuit computing a language also gives a trivial tester and corrector for that language, we conclude (similarly) by Theorem 3.7 that L has a tester and corrector of depth $O(d/d_{collapse})$. The distance parameters of the tester and corrector are affected by success probability of the reduction from depth $d_{collapse}$ circuits to $L_{internal}$ (or by its number of queries, see Claim 3.14). We note that if the number of queries made by this reduction is constant, then so are the composed tester and corrector's distance parameters. ∎

We can now prove Theorem 1.1.

Proof of Theorem 1.1. Let L be the language that is in $\mathcal{RNC}^{\mathsf{i}}$, and is $\mathcal{NC}^{\mathbf{1}}$-hard under $\mathcal{NC}^{\mathbf{0}}$-reductions. We take the $\mathcal{NC}^{\mathbf{1}}$-complete language from Lemma 4.10 to be the internal language. Since this language is hard for circuits of bounded fan-in and logarithmic depth (the Karp reduction makes only a single query), and because it has a constant depth ($\mathcal{AC}^{\mathbf{0}}$) tester and corrector (by Lemma 4.10), we conclude (by Theorem 5.1) that L has a tester and corrector of depth $O(\log^{i-1}(n))$, that uses unbounded fan-in gates. In other words, L has a tester and corrector in $\mathcal{RAC}^{\mathsf{i}-\mathbf{1}}$.

Note that since L (the external language here) is $\mathcal{NC}^{\mathbf{1}}$-hard under $\mathcal{NC}^{\mathbf{0}}$ reductions, there are $\mathcal{NC}^{\mathbf{0}}$ reductions from the Parity function as well as the AND function to L. Also, the internal language has an $\mathcal{NC}^{\mathbf{0}}$ program checker by Lemma 4.10. By using the Composition Theorem for program checkers (Theorem 3.1), and an argument similar to the one given in the proof of Lemma 3.5, we conclude that L has a checker in $\mathcal{RNC}^{\mathsf{i}-\mathbf{1}}$ (without the additional unbounded fan-in AND gate). ■

6 Program Libraries Revisited: Checkers for Matrix Functions

In this section we present new checkers, testers and correctors for the following matrix functions: multiplication, inversion, determinant and rank. These constructions are provably more efficient (in terms of circuit depth) than the optimal algorithms computing these functions, and they are checker/tester/correctors in the standard sense, i.e. they do not use a program library. We begin with an outline in Section 6.1, followed by detailed constructions and analyses. A summary of the parameters that we achieve and comparison to previous constructions appear in Table 1 below.

	Depth	Time	Program Calls	Previously Known
Multiplication over GF(2^s), $s = O(1)$	$\mathcal{NC}^0$	$O(n^2)$	$O(1)$	$\mathcal{AC}^0$, $O(\log n)$ program calls [BLR93, Rub96]
Multiplication over any finite field	$\mathcal{AC}^0$	$O(n^2)$	$O(1)$	$\mathcal{AC}^0$, $O(\log n)$ program calls [BLR93, Rub96]
Inversion over GF(2^s), $s = O(1)$	$\mathcal{NC}^0$	$O(n^2)$	$O(1)$	Library only, poly depth [BLR93]
Inversion over any finite field	$\mathcal{AC}^0$	$O(n^2)$	$O(1)$	Library only, poly depth [BLR93]
Determinant over any finite field	$\mathcal{AC}^0$	poly(n)	poly(n)	Library only, poly depth [BLR93]
Rank over poly(n) size fields	$\mathcal{AC}^0$	poly(n)	poly(n)	Library only, poly depth [BLR93]

Table 1: Complexity of Tester/Correctors (with constant error) for $n \times n$ Matrix Operations.

6.1 Outline

In this subsection we give an overview of some of the constructions and the ideas that are used in the checkers, testers and correctors for the matrix functions. In subsequent subsections we give the full details.

A Checker for Matrix Multiplication. We now present a constant-depth checker for matrix multiplication (over $GF(2)$). The starting point for our checker is Freivalds' checker for this function [Fre79]. Its input is two $n \times n$ matrices A, B, and a confidence parameter β, and it is given access to a program P that allegedly computes matrix multiplication. The checker first runs P on input (A, B). It then chooses a random vector $\vec{r}$ in $\{0,1\}^n$, and verifies that: $A \times (B \times \vec{r}) = P(A, B) \times \vec{r}$. If not, the checker outputs $\perp$. This test is repeated $O(\log(1/\beta))$ times, and the checker accepts only if all tests pass. A simple analysis shows that if $A \times B = P(A, B)$, the checker accepts with probability 1, otherwise it outputs $\perp$ with probability at least $1 - \beta$.

The advantage of this checker over the trivial checker that computes the multiplication of A and B is that it runs in *time* $O(n^2 \cdot \log(1/\beta))$, better than any known algorithm for matrix multiplication (for, say, constant β). We note however, that its parallel complexity is high: it multiplies matrices with vectors, which requires logarithmic depth. That is, each such operation (over $GF(2)$) consists of computing (in parallel) n inner products, which in turn boils down to computing n parities of n-bit vectors, and each such parity requires logarithmic depth [FSS84].

We now want to construct a constant depth checker based on Freivalds' checker, by using the composition theorem (Theorem 1.3). We take $L_{internal}$ to be Parity, and $L_{external}$ to be matrix multiplication. We now observe that the conditions of Theorem 1.3 hold: (1) there is a constant

depth reduction from Parity to Matrix multiplication: let $\vec{v}$ be the vector of bits whose parity we wish to compute; construct a matrix whose first row is $\vec{v}$ and multiply it by the all-ones matrix; the top-left item in the result is the parity of $\vec{v}$; (2) there is a constant depth checker for matrix multiplication that uses an oracle to Parity; (3) Parity has a constant depth checker (Theorem 1.4). Applying Theorem 1.3, we conclude that matrix multiplication has a constant-depth checker.

A Checker for Matrix Inversion. Below we present a constant-depth checker for matrix inversion (the function that, given a matrix, say it is singular or finds its inverse).

At first glance, constructing a constant-depth checker for matrix inversion seems challenging. Indeed, we do not know how to verify correctness of a given inverse or apply any form of random self-reduction on this function without using matrix multiplication (random self-reduction is a common tool in checker design), and computing matrix multiplication is too costly. The composition approach provides a way around this: first construct a non-efficient checker that *does* use matrix multiplication, then use composition to "remove" the matrix multiplication computations.

We now sketch the construction of an $\mathcal{AC}^0$ checker for matrix inversion (over GF(2)). The checker is given access to a potentially faulty program P for matrix inversion, and an arbitrary $n \times n$ matrix M to invert (or to output "not invertible" if M is singular). In addition we give the checker access to a matrix multiplication oracle. The checker proceeds in two stages: first it *tests* the (faulty) inversion program P to make sure that it correctly inverts random matrices with high probability. It then transforms the instance M into a random instance M' (that is invertible if and only if M is) and from $(M')^{-1}$ deduces M^{-1}. Details follow.

The testing stage. In the testing phase, the checker repeats the following several times in parallel: **(1)** Generate a random matrix A and ask P to invert A. **(2)** If P returned a matrix $P(A)$, verify that $P(A) \times A = I$. **(3)** If $P(A) \times A \neq I$, output $\perp$.

Throughout, the checker keeps track of the fraction of queries that resulted in $P(A)$ returning a matrix (rather than "not invertible"). This fraction should be close to the (constant) fraction of $n \times n$ matrices that are in fact invertible; if it is not, then the checker should declare that P is buggy. The point is that the program P can never trick the checker into believing that a non-invertible matrix is invertible, since the checker always verifies P's response by multiplying $P(A)$ and A. (Here one can already see how matrix multiplication is useful for us: it allows us to check the answers of the program, and force it to have only one-sided errors). The program must provide correct inverses for a fraction of matrices that is very close to the expected fraction of invertible matrices! This mean that, with high probability, any program that passes the test correctly inverts most invertible matrices.

Using random self-reduction. With this in mind, we continue to the second stage. The checker multiplies the instance M by a random matrix R, and asks P to invert $M \times R$. With constant probability, R is invertible, and thus if M is invertible then $M \times R$ is a random invertible matrix; therefore, P will return $(M \times R)^{-1} = R^{-1} \times M^{-1}$ w.h.p., and then the checker can multiply this on the left by R and obtain M^{-1}. (Here the reader can observe the second use we get from matrix multiplication: it gives a random self-reduction between invertible matrices). The checker verifies that $I = M \times (R \times P(M \times R))$ and if so outputs the (always correct) inverse $R \times P(M \times R)$. If, however, M is not invertible, then P can never return a correct inverse of M. By repeating the above $O(1)$ times (in parallel), the checker can be assured that it either has a correct inverse of M or that M is not invertible.

Removing the Multiplication Oracle. Up to this point, we have constructed a checker for matrix inversion that uses matrix multiplication as a sub-routine. Multiplication is the only non-$\mathcal{AC}^0$ "sub-

computation" performed by this checker, so we would like to remove it to obtain an $\mathcal{AC}^0$ checker. Above, we saw that matrix multiplication has an $\mathcal{AC}^0$ checker. So all we need to show, in order to apply the Composition Theorem and to obtain an $\mathcal{AC}^0$ checker for matrix inversion, is a constant depth reduction from multiplication to inversion. Such a reduction follows from the identity:

$$\begin{pmatrix} I_n & A & 0 \\ 0 & I_n & B \\ 0 & 0 & I_n \end{pmatrix}^{-1} = \begin{pmatrix} I_n & -A & A \times B \\ 0 & I_n & -B \\ 0 & 0 & I_n \end{pmatrix}$$

Checkers for Determinant and Rank. We also present checkers, testers and correctors for the determinant and rank functions. These are significantly more involved, and require several new ideas, as outlined below.

Amplified Reductions. A significant obstacle comes up when constructing the determinant and rank testers/correctors. The problem is that we want these testers and correctors to delegate internal sub-computations of matrix inversion to the program they check, using a reduction from inversion to determinant or rank. However, these reductions involve a polynomial number of oracle calls. Even if we are given a program for determinant (or rank) that works well on all but a small constant fraction of matrices, when we plug this program into the reduction that uses it polynomially many times, the result may have errors with very high (1 - 1/poly) probability. To overcome this problem, we present *amplified* reductions from matrix inversion to determinant and rank, these reductions show that a program that computes determinant or rank correctly on all but a small constant fraction of matrices can be used to compute matrix inversions correctly on *every* matrix with high probability.

Different Testers and Correctors. The library testers and correctors of [BLR93] use a procedure of Randall [Ran93] for generating random invertible matrices of known determinant. This procedure is recursive (and thus highly sequential), and also is more complex than other components of the testers/correctors. Since we want constant-depth testers and correctors, we cannot rely on this procedure. Instead, when building the initial tester/correctors (even before applying composition), we use different ideas from those of [BLR93]. The most significant example is our tester for the determinant function. The tester of [BLR93] simply checks that the program is correct on random matrices with known determinant (using [Ran93]). Our tester, on the other hand, first checks that the program is close to computing some homomorphism from the group of invertible $n \times n$ matrices over the finite field F to the multiplicative group of F (using the homomorphism test of [BLR93, BCLR04]). In the second stage, the tester verifies that the program is close to computing the one homomorphism that we care about, namely the determinant. This is achieved by exploiting the fact that the determinant is the *only* non-constant homomorphism that is multi-linear in the entries of the matrix.

Repeated Composition. The constructions for determinant and rank exploit, more than any other construction, the top-down approach that the Composition Theorem enables. In both cases we start with checkers that use two oracles: one for matrix multiplication and one for inversion. The Composition Theorem is then applied several times, gradually removing the oracles (or replacing them by weaker ones), until we get standard and efficient (in terms of circuit depth) testers and correctors (i.e. ones that do not use oracles to other functions).

Removing Libraries via Composition. In fact, the composition theorem is tailored to removing the need for program libraries when building checkers, testers and correctors. Simply use the program for the function being checked to compute other functions in the library, and use testers/correctors for these functions to check the correctness of these computations and correct them (if necessary). The only requirement is that a tester/corrector for a library function f only calls other library programs for functions that reduce to f.

6.2 Matrix Multiplication

We begin by using the Composition Theorem to simplify the tester and corrector for matrix multiplication of [BLR93]. Similar techniques can be used (directly) to simplify Freivalds' well known checker for matrix multiplication (see [Fre79]).

Lemma 6.1. *The matrix multiplication function over any field whose size is a constant power of 2 has an optimal tester and corrector (i.e. these run in linear-time, are in $\mathcal{NC}^0$, and make $O(1)$ program oracle calls).*

Over other (finite) fields, matrix multiplication has a tester and corrector that run in linear time, are in $\mathcal{AC}^0$, and for constant-size fields they make only $O(1)$ program oracle calls.

Proof. We examine matrix multiplication over a field F, and assume F's size is a constant power of 2 so we can add, multiply, and sample random members of F using an $\mathcal{NC}^0$ circuit (if F is of a different size, we get a tester and corrector in $\mathcal{AC}^0$ instead of $\mathcal{NC}^0$). Recall the tester and corrector for matrix multiplication presented by [BLR93]. At the heart of their constructions is Freivalds' checker for matrix multiplication [Fre79]. It takes as input three $n \times n$ matrices A, B and C, and a confidence parameter β. If $A \times B = C$ the checker accepts with probability 1, otherwise it rejects with probability at least $1 - \beta$. The specification of the checker is presented in Figure 1.

FREIVALDS_CHECKER(n, A, B, β)

1. $C \leftarrow P(A, B)$
2. repeat $O(\log(1/\beta))$ times:
   (a) Pick a random column vector $\vec{r} \in \{0,1\}^n$
   (b) If $A \times (B \times \vec{r}) \neq C \times \vec{r}$, then output $\perp$.
3. Output C.

Figure 1: Freivalds' Checker

The advantage of this checker over the trivial checker that computes the multiplication of A and B is that it runs in *time* $O(n^2 \cdot \log(1/\beta))$, better than any known algorithm for matrix multiplication. Note, however, that its *parallel time complexity* is high, as it needs to compute multiplications of matrices over F with $\{0,1\}$-vectors. This involves computing long sums over the field (e.g. Parity in the case of $GF(2)$), and thus requires depth that is nearly logarithmic in n. The tester and corrector for matrix multiplication given by [BLR93] (shown in Figures 2 and 3), are based on Frievalds' checker, and thus they are not in constant depth. We transform them to be of constant

depth (while maintaining their running time and number of program oracle calls) by using the Composition Theorem (Theorem 3.7).

MATRIX MULTIPLICATION CORRECTOR(A,B,β)

1. repeat $O(\log(1/\beta))$ times *in parallel*:
   (a) $A_1, B_1 \leftarrow$ random $n \times n$ matrices
   (b) $A_2 \leftarrow A - A_1$
   (c) $B_2 \leftarrow B - B_1$
   (d) $C \leftarrow P(A_1, B_1) + P(A_1, B_2) + P(A_2, B_1) + P(A_2, B_2)$
   If $Freivalds\_Checker(A, B, C, \frac{\beta}{2})$ accepts, then output C
2. If $Freivalds\_Checker$ never accepted, then output $\perp$

Figure 2: Matrix Multiplication Corrector

MATRIX MULTIPLICATION TESTER(β)

1. Repeat $O(\log(1/\beta))$ times *in parallel*:
   (a) Generate two random matrices A and B
   (b) $C \leftarrow P(A, B)$
   (c) If $Freivalds\_Checker(A, B, C, \frac{\beta}{2})$ accepts, then the answer from this step is 0. Otherwise the answer is 1
2. If the fraction of steps answering 0 is at least $\frac{1}{16}$, then accept. Otherwise, reject.

Figure 3: Matrix Multiplication Tester

Our Matrix Multiplication Corrector and Tester: To apply Theorem 3.7, we take matrix multiplication to be the external function, and *matrix-row-sums* to be the internal function. The *matrix-row-sums* function receives a matrix and returns the (column) vector whose i-th entry is the sum (over F) of the i-th row of the matrix. We show that the conditions of the theorem hold:

1. A reduction from the internal function to the external function:
   The $\mathcal{NC}^0$ reduction from computing the sums of the rows of an $n \times n$ matrix to multiplication of $n \times n$ matrices is simple: given a matrix A, multiply it with the all-1 $n \times n$ matrix, output the first column of the result.
2. The internal function "helps" to test/correct the external function:

An oracle that computes *matrix-row-sums* can be used to compute matrix-vector multiplications in $\mathcal{NC}^0$:

$$A \times \vec{v} = \begin{pmatrix} a_{1,1} & \dots & a_{1,n} \\ a_{2,1} & \dots & a_{2,n} \\ \dots & & \\ a_{n,1} & \dots & a_{n,n} \end{pmatrix} \times \begin{pmatrix} v_1 \\ . \\ . \\ v_n \end{pmatrix} = \begin{pmatrix} \sum_{i=1}^{n} a_{1,i} \cdot v_i \\ . \\ . \\ \sum_{i=1}^{n} a_{n,i} \cdot v_i \end{pmatrix}$$

This final vector can be computed (in $\mathcal{NC}^0$) by computing *matrix-row-sums* on the matrix whose (i,j)-th entry is $a_{i,j} \cdot v_j$ (where all $a_{i,j}$-s are field elements, and v_j-s are in $\{0,1\}$).

Given this $\mathcal{NC}^0$ procedure to compute matrix-vector multiplications results in an $\mathcal{NC}^0$ tester and corrector for matrix multiplication (using an oracle to *matrix-row-sums*).

3. Testability and Correctability of the internal function:

   The tester and corrector for the *matrix-row-sums* function are generalizations of the tester and corrector for computing products over finite groups given by Claim 4.5 and Theorem 4.8 (both appear in Section 4. The tester and corrector work with the additive group over F, and apply on each matrix row (independently) the randomization technique given in the proof of Claim 4.5. This randomization technique is then used, as in the proof of Theorem 4.8, to test and correct the program that allegedly computes the *matrix-row-sums* function.

We can now use the Composition Theorem to construct a *standard* constant-depth tester and corrector for matrix multiplication (i.e., one that only uses a program oracle that allegedly computes matrix multiplication). Moreover, since (for a constant β) the tester and corrector that we start with (before applying the composition), as well as all reductions and the *matrix-row-sums* tester and corrector, all run in linear time and $\mathcal{NC}^0$ (and only make a constant number of oracle calls), we conclude that the composed tester and corrector are *optimal*: they only make a constant number of calls to the program they check, run in linear time, and are in $\mathcal{NC}^0$ (for F of size a constant power of 2).

For other finite fields, addition, multiplication by 0 or 1, and generating (almost) random field elements are all in $\mathcal{AC}^0$, and so are the composed tester and corrector (they still run in linear time and make only a constant number of program oracle calls though). ■

6.3 Matrix Inversion

A tester and corrector for the function that computes whether a matrix is invertible or not were given by [BLR93]. Their tester and corrector used the concept of a *Library* to get access to a matrix-multiplication oracle . We present a standard tester and corrector (that do not use a library) for the matrix inversion function.

Lemma 6.2. *The matrix inversion function over any field whose size is a constant power of 2 has an optimal tester and corrector (i.e. these run in linear-time, are in $\mathcal{NC}^0$, and make $O(1)$ program oracle calls).*

Over other (finite) fields, matrix inversion has a tester and corrector that run in linear time, are in $\mathcal{AC}^0$, and for constant-size fields they make only $O(1)$ program oracle calls.

Proof. We examine matrix inversion over a field F, and assume F's size is a constant power of 2 (similarly to the case of matrix multiplication, for fields different size we get a tester and corrector in $\mathcal{AC}^0$ instead of $\mathcal{NC}^0$). We begin by presenting the corrector (Figure 4) and tester (Figure 5) as if they have access to a (correct) matrix multiplication oracle , we will later remove this oracle using the Tester/Corrector Composition Theorem. Note that we analyze the behavior of this tester and corrector on the uniform distribution over invertible matrices with entries in the field F.[18] We use P to denote the matrix inversion program being checked, and $Mult$ to denote the (always correct) matrix multiplication oracle .

MATRIX INVERSION CORRECTOR(A,β)

1. repeat $O(\log(1/\beta))$ times *in parallel*:
   (a) Generate a random matrix R
   (b) $R' \leftarrow Mult(A, R)$
   (c) Use P to try to invert R': $R'_{Inv} \leftarrow P(R')$.
   If P cannot invert, proceed to the next loop iteration
   (d) $A' \leftarrow Mult(R, R'_{Inv})$
   (e) If $I_n = Mult(A, A')$ (where I_n is the identity matrix of dimension n), then output A'
2. Output that A is not invertible.

Figure 4: Matrix Inversion Corrector

Corrector Analysis: We begin by noting that when A is *not* invertible, this corrector *always* outputs $\perp$ (it outputs $\perp$ unless it actually finds A's inverse), thus we restrict our attention to the program's behavior for invertible As. With some constant probability the random matrix R will be invertible (this probability grows with the size of the field F, but even for $GF(2)$ it is at least 1/4). In loop iterations when R is invertible, computing the inverse of $A \times R$ using P is actually inverting a totally random invertible matrix. If this inversion succeeds, the corrector always outputs the inverse of A. Thus if P is $\frac{1}{8}$-close to being correct (for a random invertible matrix), then in each loop iteration the probability that *both* R is invertible (probability at least 1/4) *and* P succeeds (probability at least $1 - \frac{1}{8}$) is greater than 0.1. The probability that in at least one of the $O(\log(1/\beta))$ loop iterations this constant probability event occurs at least once is at least $1 - \beta$. Note that this corrector is constant-depth (using matrix multiplication oracles).

Tester Analysis: Let γ be the probability that a random matrix over F is invertible (can be hard-wired into the tester, γ is at least 1/4). Note that the errors of the program can only be "one-sided", in the sense that in any iteration of the tester, if the random matrix A is not invertible, then the answer from that iteration is always 1. Thus the probability that the answer from any iteration

[18] Recall that we are allowed to choose any distribution over the instances of the function, as long of course that we prove the tester-corrector pair to be correct with respect to this distribution. See Remark 2.7.

MATRIX INVERSION TESTER(β)

1. Let γ be the probability that a random matrix over F is invertible (can be hard-wired into the tester)
2. repeat $O(\log(1/\beta))$ times *in parallel*:
   (a) Generate a random matrix A
   (b) Compute $A_{Inv} \leftarrow P(A)$
   (c) If $I_n = Mult(A, A_{Inv})$ then the answer from this step is 0, otherwise the answer is 1
3. Let η be the fraction of 0-answers in the loop. If $|\eta - \gamma| \leq \frac{\gamma}{16}$, then accept. Otherwise reject.

Figure 5: Matrix Inversion Tester

is 1 is at least $1-\gamma$. If the program is at least $\frac{1}{32}$-close to being correct (on the *invertible* matrices), then the probability that the answer from the iteration is 0 is at least $\frac{31\gamma}{32}$, and the probability that the fraction of $O(\log(1/\beta))$ independent iterations that give 0 answers is $\frac{\gamma}{16}$-distant from γ is at most β (by a Chernoff Bound). If the program is at least $\frac{1}{8}$-far from being correct (on a random invertible matrix), then the probability that the answer from the iteration is 0 is at most $\frac{7\gamma}{8}$, and the probability that the fraction of $O(\log(1/\beta))$ independent iterations that give 0 answers is $\frac{\gamma}{16}$-distant from γ is at least $1-\beta$ (again by a Chernoff Bound). Note that this tester is constant depth (again, with matrix multiplication oracle gates).

Tester-Corrector Pair: The tester and corrector are a tester-corrector pair because the tester accepts $\frac{1}{32}$-good programs w.h.p., and rejects programs that are not $\frac{1}{8}$-good w.h.p. The corrector corrects using any program that is at least $\frac{1}{8}$-good.

The Final Tester and Corrector: We have presented a constant depth tester and corrector using oracle gates to the matrix multiplication function. We note that for a constant β and F whose size is a constant power of 2 the tester and corrector are in $\mathcal{NC}^0$. We now want to use these, together with the Composition Theorem (Theorem 3.7), to construct a constant-depth tester and corrector in the standard sense (i.e. *without* oracle gates to the matrix multiplication function). To do this, we need to show that the conditions of the Composition Theorem hold when the external function is matrix inversion, and the internal function is matrix multiplication. Condition 2 (the "internal" language helps check the "external" function) is satisfied by the construction of a tester and corrector above. Condition 3 (testability and correctability of the internal language) is satisfied by the constant-depth ($\mathcal{NC}^0$ for the proper β and field size) tester and corrector for matrix multiplication given in Section 6.2.

To show Condition 1 (hardness of the external language for the internal language), we need a reduction from the internal language to the external language. To see this reduction, observe that

when computing the matrix multiplication $A \times B$, it suffices to examine the block matrix:

$$M = \begin{pmatrix} I_n & A & 0 \\ 0 & I_n & B \\ 0 & 0 & I_n \end{pmatrix}$$

And observe that:

$$M^{-1} = \begin{pmatrix} I_n & -A & A \times B \\ 0 & I_n & -B \\ 0 & 0 & I_n \end{pmatrix}$$

Now the upper-right block of M^{-1} is the multiplication of A and B. Thus to multiply two $n \times n$ matrices, it suffices to compute a single inversion of a $3n \times 3n$ matrix. Alternatively, this can be done with 27 non-adaptive inversions of $n \times n$ matrices by breaking A and B to 3×3 block matrices, where each block is of dimension $n/3$, and then applying the reduction to the blocks.

Now using the Composition Theorem, we can remove the matrix multiplication oracle gates. The resulting tester and corrector are a tester-corrector pair, and do not require using a library. Moreover, since (for a constant β) the checker and corrector that we start with (before applying the composition), as well as all reductions and the matrix multiplication tester and corrector, all run in linear time and $\mathcal{NC}^0$ (and only make a constant number of oracle calls), we conclude that the composed tester and corrector are *optimal*: they only make a constant number of calls to the program they check, run in linear time, and are in $\mathcal{NC}^0$ (for F of size a constant power of 2).

Similarly to the case of matrix multiplication, for other fields the composed tester and corrector run in $\mathcal{AC}^0$, linear time, and make only $O(1)$ program oracle calls. ■

Finally, we note that the Composition Theorem also gives a (standard) tester and corrector for the (boolean) function that checks whether a matrix is invertible or not over polynomial-size fields. The tester and corrector run in $\mathcal{AC}^0$, but require polynomial time and make polynomially many oracle calls.

6.4 Matrix Determinant

A tester and corrector for matrix determinant were given by [BLR93]. They used a library with matrix-multiplication and matrix-inversion functions. In this section we construct a standard constant-depth tester and corrector for matrix determinant, i.e. ones that do not use a program library (we do note, however, that the number of parallel calls to the program being checked is larger than in the library tester and corrector of [BLR93]). Our corrector is based on the construction of [BLR93], however our tester is different. Unlike [BLR93], we avoid the use of Randall's [Ran93] (sequential) procedure for generating random invertible matrices with known determinant.

In this section and in the next, we use a procedure that generates (w.h.p.) a random invertible matrix over a finite field F, given an oracle Inv that correctly inverts any invertible matrix (and returns $\perp$ for non-invertible matrices). A description of such a procedure appears in Figure 6.

Claim 6.3. *The procedure in Figure 6 outputs $\perp$ with probability at most β. Furthermore, conditioned on the event that it does not output $\perp$, its output is uniformly distributed over invertible matrices with entries in the field F. If β is a constant and the size of F is a constant power of 2, the procedure can be implemented in $\mathcal{NC}^0$, otherwise it can be implemented in $\mathcal{AC}^0$.*

$Random - Invertible(n,\ \beta)$

1. For i going from 1 to $O(\log(1/\beta))$ do the following *in parallel*:
   (a) Choose a random $n \times n$ matrix A_i with entries in F.
   (b) Let $A_i^{-1} \leftarrow Inv(A_i)$.
2. If for every i, $A_i^{-1} = \perp$, output $\perp$. Otherwise, output A_i for the minimal i for which it is not $\perp$.

Figure 6: Generating a random invertible matrix

Proof. A random matrix is invertible with (at least) constant probability bounded away from 0. By running the loop in the procedure for $O(\log(1/\beta))$ iterations, with probability at least β one of the iterations will generate an invertible matrix. Whenever this event occurs, the procedure outputs the first invertible matrix it generated, which is indeed a random invertible matrix. ∎

We now turn to the problem of testing and correcting programs for matrix determinant.

Lemma 6.4. *The matrix determinant function over a finite field F has an $\mathcal{AC}^0$ tester and corrector.*

Proof. The corrector (Figure 7) and tester (Figure 8) are presented as if they have access to a (correct) oracle for matrix multiplication and matrix inversion. The oracles will later be removed using the Tester/Corrector Composition Theorem (Theorem 3.7). We denote by P the matrix determinant program being checked, and by $Mult$ and Inv the (always correct) matrix multiplication and inversion oracles (respectively). The behavior of the oracle tester and oracle corrector is analyzed with respect to the uniform distribution over *invertible* matrices with entries in the field F.

Corrector Analysis: We begin by observing that the corrector always outputs 0 on non-invertible matrices (by using its oracle Inv to the inversion function). It remains to analyze its behavior on invertible matrices. Assume P is a $\frac{1}{16}$-good program for matrix determinant w.r.t the uniform distribution on invertible matrices. By Claim 6.3, the probability that $Random - Invertible$ fails in any iteration is at most $\frac{1}{8}$. When this doesn't happen, R is a random invertible matrix. The corrector multiplies A and R to get the matrix R', which is again a uniformly random invertible matrix. The probability that P errs on a uniformly random matrix such as R or R' is at most $\frac{1}{16}$. By taking a union bound over the events that $Random - Invertible$ returns $\perp$, and the determinants of R or R' are not computed correctly, we conclude that the correct determinant of A is computed in each loop iteration with probability at least $\frac{3}{4}$. After computing the approximate majority of all loop iterations answers, the total error probability of the corrector is at most β (by a Chernoff bound).

Tester Analysis: The first loop tests that the program P computes a function that is close (with respect to the uniform distribution on invertible matrices) to a homomorphism from the (non-abelian) group of invertible matrices over F (denoted $GL_n(F)$) to the (abelian) multiplicative

MATRIX DETERMINANT CORRECTOR(A,β)

1. If $\perp = Inv(A)$ then output 0 and exit.
2. Otherwise, repeat $O(\log(1/\beta))$ times *in parallel*:
   (a) Generate a random invertible matrix R using $Random - Invertible(n, \frac{1}{8})$ (see Figure 6). If it fails the answer from this iteration is 0.
   (b) $R' \leftarrow Mult(A, R)$.
   (c) $d_R \leftarrow P(R)$.
   (d) If $d_R = 0$ skip this iteration.
   (e) $d_{R'} \leftarrow P(R')$.
   (f) The answer from this iteration is $d_{R'}/d_R$.

   And output the majority among the answers for all iterations.

Figure 7: Matrix Determinant Corrector

group over the elements of F (denoted F^*). Note that the determinant is such a homomorphism. We use the generic homomorphism tester of [BLR93] (with its analysis for non-Abelian groups given in [BCLR04]) to analyze the tester.

If the program is $\frac{1}{256}$-good on a random invertible matrix, then in particular it is close to a homomorphism from $GL_n(F)$ to F^* (the determinant is such a homomorphism), and we can proceed by following the analysis of [BLR93]. The probability that the program does not compute the homomorphism correctly on even one of the random (but not independent) matrices $R_1, R_2, R_1 \times R_2$ is at most $\frac{3}{256}$. The probability that $Random - Invertible$ fails in one of its two activations is at most $\frac{1}{50}$. Taking a union bound, the total probability of the answer in each iteration being 0 is less than $\frac{1}{24}$. Thus, the program is rejected after the loop with probability at most $\frac{\beta}{2}$.

Now consider the case that the program is $\frac{1}{16}$-far from computing any homomorphism from $GL_n(F)$ to F^*. The probability that $Random - Invertible$ fails in one of its two activations is at most $\frac{1}{50}$. When this does not happen R_1 and R_2 are random invertible matrices, and the probability that $P(R_1) \times P(R_2) \neq P(R_1 \times R_2)$ is at least $\frac{1}{8}$ (see [BCLR04]). Taking a union bound, the probability of the answer in each iteration being 0 is at least $\frac{1}{10}$. Thus the program is rejected after the loop with probability at least $1 - \frac{\beta}{2}$.

The second loop distinguishes between the determinant function and other homomorphisms from $GL_n(F)$ to F^*. To analyze it we need the following claim.

Claim 6.5. *For every homomorphism $h : GL_n(F) \to F^*$, there exists an integer $0 \leq k \leq |F| - 1$, such that for every $M \in GL_n(F)$, $h(M) = det(M)^k$.*

Proof. Consider the group G of diagonal matrices that have an arbitrary elements of F^* along the diagonal. G is clearly isomorphic to $(F^*)^n = F^* \times F^* \times \ldots \times F^*$ (n times). Next, consider the restriction $h : G \to F*$. This is a homomorphism from $(F^*)^n$ to F^*, and such homomorphisms are easily seen to all be of the form $h(M) = a_1^{k_1} a_2^{k_2} \cdots a_n^{k_n}$ (for $0 \leq k_1, \ldots, k_n \leq |F| - 1$), where the a_i's

MATRIX DETERMINANT TESTER(n, β)

1. Repeat $O(\log(1/\beta))$ times *in parallel*:

   (a) Run twice $Random - Invertible(n, \frac{1}{100})$ to generate two matrices R_1 and R_2. If either execution outputs $\perp$, then the answer from this iteration is 0.

   (b) If $P(R_1) \cdot P(R_2) \neq P(Mult(R_1 \cdot R_2))$ then the answer from this iteration is 0, otherwise the answer is 1.

   If the fraction of 0-answers (out of the iterations that we didn't skip) is at least $\frac{1}{16}$ then reject.

2. Otherwise, repeat $O(\log(1/\beta))$ times *in parallel*:

   (a) Run $Random - Invertible(n, \frac{1}{32})$ to generate a matrix R. If the execution outputs $\perp$, then the answer from this iteration is 0.

   (b) Choose a uniformly distributed non-zero element c in F.

   (c) Let R' be the matrix R with every entry in the first row multiplied by c. If $c \cdot P(R) \neq P(R')$ then the answer from this iteration is 0, otherwise the answer is 1.

   If the fraction of 0-answers is at least $\frac{1}{8}$ then reject, otherwise accept.

Figure 8: Matrix Determinant Tester

are the diagonal entries of M. This follows from the fact that F^* is cyclic and homomorphisms of product groups are just products of the homomorphisms on each component.

Now we want to show that $k_1 = k_2 = \ldots = k_n = k$ to prove that $h(M) = det(M)^k$ for these specific matrices. Let S be a permutation matrix that swaps rows i and j upon left multiplication and swaps columns i and j upon right multiplication. Clearly, $S^2 = I$, so $h(S)^2 = 1$. Thus $h(SMS) = h(S)h(M)h(S) = h(M)$ and so $k_i = k_j$, since SMS just swaps a_i and a_j and the function remains unchanged. This is true for any i and j, so all the k's must be the same.

Now consider the row/column-operation matrices, i.e. matrices with 1's on the diagonal, a single 1 elsewhere, and 0's everywhere else. These, together with the elements of G generate all of $GL_n(F)$, since Gaussian elimination allows us to transform any non-singular matrix to the identity, and moreover, when the matrix is non-singular one can do Gaussian elimination without any swaps, so these operations do indeed suffice. Conveniently, $h(M) = 1$ for all of these matrices; indeed, let p be the characteristic of F, then for any row/column operation matrix M, we have $M^p = I$ (note how we use here the fact that F is finite) and so $h(M)^p = h(M^p) = h(I) = 1$ and so $h(M) = 1 = 1^k = det(M)^k$.

So to conclude, we exhibited a set of generators of $GL_n(F)$ such that every matrix T in this set has $h(T) = det(T)^k$ for some global, fixed k, and therefore $h(M) = det(M)^k$ for every matrix in $GL_n(F)$. ■

We now proceed with the analysis of the tester. In each iteration of the second loop, unless the $Random - Invertible$ call fails (probability at most $\frac{1}{32}$), the matrices R and R' are uniformly distributed in $GL_n(F)$ (though not independent). If the program is $\frac{1}{256}$-close to the determinant function (with respect to the uniform distribution on invertible matrices), then with probability at least $\frac{254}{256}$ the program agrees with the determinant on both matrices. Taking a union bound, the answer from each iteration will be 0 with probability at most $\frac{1}{16}$. By the Chernoff bound, the program is accepted in Step 2 with probability at least $1 - \frac{\beta}{2}$.

On the other hand, if the program is $\frac{1}{16}$-close to some other homomorphism $h : GL_n(F) \to F^*$, then by Claim 6.5, $h(M) = det(M)^k$ for some fixed $0 \leq k \leq |F| - 1$ ($k \neq 1$). With probability at least $\frac{7}{8}$, the program evaluated on both R and R' agrees with h. In this case we will have, $c \cdot P(R) = c \cdot h(R) = c \cdot det(R)^k$, and on the other hand $P(R') = det(R')^k = c^k \cdot det(R)^k$. With probability at least $1/2$ over the choice of c, $c \neq c^k$ (since $k \neq 1$). Taking a union bound over the probability that $Random - Invertible$ fails, we conclude that the answer from each iteration is 0 with probability at least $\frac{1}{4}$. Therefore, by the Chernoff bound the program will be rejected with probability at least $1 - \frac{\beta}{2}$.

In conclusion, if the program is $\frac{1}{256}$-good on invertible matrices, it is rejected in any of the two steps with probability at most $\frac{\beta}{2}$. The total rejection probability is at most β. If the program is not $\frac{1}{16}$-good, then either it is not $\frac{1}{16}$ close to any homomorphism, and rejected in the first loop with probability at least $1 - \frac{\beta}{2}$, or it is $\frac{1}{16}$-close to some homomorphism $h \neq det$, and then it is rejected with probability at least $1 - \frac{\beta}{2}$ in the second loop. Thus the probability that the tester rejects a $\frac{1}{16}$-far program is at least than $1 - \beta$.

Tester-Corrector Pair: The (oracle) tester and corrector are a tester-corrector pair because the tester rejects any program that is not $\frac{1}{16}$-good w.h.p, and the corrector corrects $\frac{1}{16}$-good programs.

Composing the Tester and Corrector: The tester and corrector presented above are constant depth using oracle gates to the matrix multiplication and inversion functions.[19] We want to use the Composition Theorem (Theorem 3.7), with matrix determinant as an external function, and inversion as the internal function, to construct a constant-depth tester and corrector for the determinant function that does need the inversion oracle. To do this, we need to show that the conditions of the theorem hold. By the above, Condition 2 (the internal language "helps" to check the external language) holds. Condition 3 (testability and correctability of the internal language) holds by Claim 6.2.

Condition 1, hardness of the external language for the internal language, also holds. Indeed, Cramer's rule states that each coordinate of the inverse is the corresponding cofactor (the signed determinant of the corresponding minor) divided by the determinant of the matrix. For a matrix A, denote by $M_{i,j}(A)$ the (i,j)-th minor of A (i.e. A with the i-th row and j-th column removed). Cramer's rule states that:

$$(A^{-1})_{i,j} = (-1)^{i+j} \cdot \frac{det(M_{i,j}(A))}{det(A)}$$

Thus we can use the determinant program oracle to compute the inverse. While this suffices for applying the Composition Theorem, it is somewhat unsatisfying because the reduction from

[19] Note that multiplication and division of field elements over large fields cannot be done in $\mathcal{AC}^0$. They are, however, easily doable in $\mathcal{AC}^0$ with an oracle to matrix multiplication.

inversion to determinant needs to compute the *entire* inverse of a matrix, and thus makes $O(n^2)$ program oracle queries (one call per matrix entry to get $det(M_{i,j}(A))$, plus another "global" call for getting $det(A)$). This implies that when we apply the Composition Theorem it will give a composed tester and corrector with only polynomially small distance parameters (see Claim 3.14). Roughly speaking, the reason is that in the composition step, we want the reduction from the internal language to the external language, to succeed with high probability when its oracle is the program being tested and corrected. For this to happen we want that with high probability, simultaneously all the oracle calls are correct. We therefore require that the success probability of each call (or in other words, the distance of the program from the function it allegedly computes) is smaller than inverse the number of oracle calls the reduction makes (so that we can apply a union bound over the oracle calls).

In what follows this obstacle is overcome. We show how to amplify the success probability of each oracle call to the program being tested and corrected, so that it works even with programs that are only constant close to the function. This results in an *amplified* reduction from matrix inversion to determinant, that uses a program for determinant that is $\Omega(1)$-good to get a program for inversion that is $\Omega(1)$-good. We use this amplified reduction in the Composition Theorem to get a corrector and tester with constant distance parameters (that still make polynomially many calls to the program oracle).

Claim 6.6. *For any constant ε, there exists a constant depth $(\varepsilon, \frac{1}{64})$-reduction from matrix inversion on* any *distribution D_1 to matrix determinant on the uniform distribution. The reduction uses oracle gates to matrix multiplication.*

Proof. We begin with the standard reduction that uses Cramer's rule. As a simplification first step, suppose that the reduction could generate random invertible matrices in constant depth. I.e suppose the reduction has access to an oracle that outputs random invertible matrices. Now observe that the standard reduction using Cramer's Rule makes many oracle calls to determinant only on *invertible* matrices. On singular matrices the reduction makes only one call (because if the determinant is 0 we already know that there is no inverse).

To amplify the success probability of the reduction when computing determinants of *invertible* matrices, we will use the following modification of the corrector for the matrix determinant function given in Figure 7: first, start the execution of the corrector from step 2 in Figure 7 (step 1 is not necessary since we know that the matrix is invertible). Second, we will use our oracle that generates random invertible matrices instead of the procedure $Random - Invertible$. Note that this modified corrector does not use an oracle to inversion, but its correction properties (for invertible matrices) remain the same.

We now use this corrector to amplify the success probability of each call that the standard (Cramer's Rule) reduction makes to compute the determinant of an invertible matrix. Every time the reduction wants to call the program on some matrix A, it will instead run the corrector on A with the same program, setting β (the confidence parameter) to be $O(1/n^2)$. If the program for determinant is $\frac{1}{64}$-good on random matrices, then it is at least $\frac{1}{16}$-good on random *invertible* matrices, and with high probability (more than $1 - 1/n^2$) $det(A)$ will be computed correctly by the corrector. Thus with high (constant) probability the whole inverse matrix is computed correctly.

Finally, the reduction still needs a method for generating random invertible matrices. To overcome this difficulty, observe that if the program oracle *is* a $\frac{1}{64}$-good program for computing determinant on random matrices, then it *can* compute *almost* random invertible matrices in constant

depth. To do this, generate (in parallel) several ($O(1)$) random matrices, use the determinant program oracle to check whether or not their determinant is zero, and output the first matrix whose determinant (according to the program oracle) is non-zero. The output of this procedure is an almost random invertible matrix (the statistical distance between the output and the distribution of random invertible matrices is less than $\frac{1}{32}$). The determinant corrector, as it is used in the amplified reduction above, works even when activated with such almost-random invertible matrices.

This reduction works with high probability for computing the inverse of *any* matrix, and thus in particular it works for any distribution on matrices. ■

The amplified reduction (which uses only the determinant program oracle and an oracle to matrix multiplication) can be used in the Composition Theorem to get a constant-depth tester and corrector for matrix determinant, using an oracle for matrix multiplication.

Corollary 6.7. *The matrix determinant function has an $\mathcal{AC}^0$ tester and corrector, using an oracle to matrix multiplication.*

Proof. Use the Composition Theorem with inversion as the internal language, determinant as the external language and the amplified reduction from Claim 6.6. ■

Proposition 6.1. The matrix determinant function has an $\mathcal{AC}^0$ tester and corrector without any additional oracles (i.e. a tester and corrector in the standard sense).

Proof. By Corollary 6.7 the determinant function has a constant-depth tester and corrector using a matrix multiplication oracle . While it was already shown above that good programs for determinant can compute inversions, and thus also multiplications (see the proof of Claim 6.2), again the Composition Theorem cannot be directly applied because the reduction from matrix multiplication to determinant makes too many oracle calls. Moreover, the "amplified" reduction from inversion to determinant won't help because it itself uses a matrix multiplication oracle.

To overcome these obstacles, observe that one can replace the matrix multiplication oracle with a vector-sum oracle while maintaining constant depth (computing all the entries of a matrix multiplication in parallel). The vector-sum function over F has a constant depth tester and corrector (similar to the tester and corrector for the parity function in Lemma 4.9 and the *matrix-row-sums* tester and corrector in the proof of Claim 6.1). Moreover, there is a one-to-one $\mathcal{NC}^0$ reduction from vector-sum to determinant. The reduction on a vector $\vec{v} = (x_1, x_2, \ldots, x_n)$ proceeds as follows:

$$\sum_{i=1}^{n} x_i = det \begin{pmatrix} x_1 & -x_2 & x_3 & \ldots & -1^{n-1} \cdot x_n \\ 1 & 1 & 0 & \ldots & \ldots \\ 0 & 1 & 1 & 0 & \ldots \\ \cdot & & & & \cdot \\ \cdot & & & & \cdot \\ \cdot & & & & \cdot \\ 0 & 0 & \ldots & 1 & 1 \end{pmatrix}$$

Applying the Composition Theorem (Theorem 3.7), we get a constant-depth tester and corrector for matrix determinant without any additional oracle gates. ■

■

6.5 Matrix Rank

A tester and corrector for matrix rank were given by [BLR93]. They used a library with matrix-multiplication and matrix-inversion functions. In this section, we eliminate the need for a library, using the Composition Theorem to give a standard constant-depth tester and corrector for matrix rank over fields of polynomial size (we do note, however, that the number of parallel calls to the rank program oracle made by our tester and corrector is larger than in the library tester and corrector of [BLR93]).

Lemma 6.8. *The matrix rank function over a prime field F of polynomial size (in the dimension of the matrix), has an $\mathcal{AC}^0$ tester and corrector.*

Proof. The corrector (Figure 9) and tester (Figure 10) are similar to the ones given by [BLR93], but they are presented as if they have access to (always correct) matrix multiplication and matrix inversion oracles . We follow the notation of [BLR93], using $I^r_{n\times n}$ to denote the $n \times n$ matrix that is all zero, except for r 1's in the first r entries of the main diagonal.

We use P to denote the matrix rank program being checked, and $Mult$ and Inv to denote the (always correct) matrix multiplication and inversion oracles (respectively). The behavior of the tester and corrector is analyzed on the distribution on $2n \times 2n$ matrices generated by uniformly choosing a random rank r in $\{0 \dots 2n\}$ and then generating a random $2n \times 2n$ matrix of rank r.

MATRIX RANK CORRECTOR(A,β)

1. Repeat $O(\log(1/\beta))$ times *in parallel*:
   (a) Use $Random - Invertible(2n, O(\beta))$ to generate two random $2n \times 2n$ invertible matrices R and Q. If $Random - Invertible$ returns $\perp$, the answer from this iteration is $\perp$.
   (b) Choose a random $t \in \{0 \dots n\}$
   (c) $A_1 \leftarrow \begin{pmatrix} A & 0_{n\times n} \\ 0_{n\times n} & I^t_{n\times n} \end{pmatrix}$
   (d) $A_2 \leftarrow Mult(A_1, R)$
   (e) $A_3 \leftarrow Mult(Q, A_2)$
   (f) The answer from this iteration is $P(A_3) - t$
2. Output the majority among the answers for all iterations.

Figure 9: Matrix Rank Corrector

Corrector Analysis: The analysis follows that of [BLR93]. The matrix A_1 (of size $2n \times 2n$) is of rank $rank(A) + t$, and thus $A_3 = Q \times A_1 \times R$ is a *random* matrix of rank $rank(A) + t$. This fact is stated in the following claim:

Claim 6.9. *Let A be a $2n \times 2n$ matrix of rank r, and let R, Q be random invertible matrices of size $2n \times 2n$, where all matrices are over F. Then the matrix $Q \times A \times R$ is a random uniformly distributed matrix of rank r over F.*

Proof. There is a bijection from the set of pairs of invertible matrices that take any A to I_r, to the set that takes I_r to itself. ■

MATRIX RANK TESTER(β)

1. Repeat $O(\log(1/\beta))$ times *in parallel*:
   (a) Use $Random - Invertible(2n, O(\beta))$ to generate two random $2n \times 2n$ invertible matrices R and Q. If $Random - Invertible$ returns $\perp$, the answer from this iteration is 0.
   (b) Choose a random $r \in \{0 \dots 2n\}$
   (c) $A_1 \leftarrow I^t_{2n \times 2n}$
   (d) $A_2 \leftarrow Mult(A_1, R)$
   (e) $A_3 \leftarrow Mult(Q, A_2)$
   (f) If $P(A_3) = r$ the the answer from this iteration is 1, otherwise the answer is 0.
2. If the fraction of 0-answers is at most $\frac{1}{32}$, then accept. Otherwise reject.

Figure 10: Matrix Rank Tester

Tester Analysis: The analysis follows that of [BLR93]. The matrix A_1 (of size $2n \times 2n$) is of rank r, and thus $A_3 = R^{-1} \times A_1 \times R$ is a *random* matrix of rank r. The tester tests whether the program correctly computes ranks for a random matrices of randomly selected rank.

Composing the Tester and Corrector: First, note that the (perfect) matrix multiplication and inversion oracles that the tester and corrector use can both be replaced by a (perfect) oracle to determinant, while maintaining the (constant) depth of the tester and corrector.[20] We now use the Composition Theorem (Theorem 3.7), to construct a constant-depth tester and corrector in the standard sense (i.e. *without* oracle gates). To do this, we need to show that the conditions of the theorem hold when the external function is matrix rank, and the internal function is matrix determinant. Condition 2 (the internal function "helps" test/correct the external function) is satisfied by the construction of a tester and corrector we just presented. Condition 3 (testability and correctability of the internal language) is satisfied by the constant-depth tester and corrector for matrix determinant given in Claim 6.4.

Condition 1 (hardness of the external language for the internal language) requires more work, and in fact we only know of a reduction from determinant to rank for fields of polynomial size

[20] Recall that the matrix rank tester and corrector work for fields of polynomial size, and thus multiplication and division of field elements can be done in $\mathcal{AC}^0$.

(polynomial in the matrix size n). The reduction uses the fact that computing whether the determinant of a matrix over $F = GF(k)$ (for a prime k) is equal to some value a or not is in the complexity class $mod_k - L$ (see e.g. [BDHM91]). Furthermore, any boolean $mod_k - L$ computation on a polynomial size input can be transformed (in $\mathcal{NC}^0$) into a polynomial size $n^c \times n^c$ matrix whose rank is full if and only if the the result of the computation is 1. This leads to a reduction from matrix determinant to matrix rank over $GF(k)$. For a matrix A:

1. For $a \leftarrow 0 \ldots k-1$, do the following in parallel:

   Construct the matrix D_a whose rank is full if and only if the determinant of A equals a. Use the matrix rank oracle to determine whether the rank of D_a is full.

2. Return the a for which the determinant of D_a was non-zero.

Note that while this reduction is constant depth, its size and number of oracle calls are polynomial in the field size k. For small constant field sizes the reduction makes a constant number of oracle queries, and we can immediately use the Composition Theorem to obtain a constant-depth tester and corrector for matrix rank with constant distance parameters. For polynomial field sizes, however, the reduction makes a polynomial number of oracle calls and the distance parameters become polynomially small (as was the case when composing the matrix determinant tester and corrector). However, as was the case for matrix determinant, we can again do better by "amplifying" the reduction.

Claim 6.10. *For any $\varepsilon > 0$, there exists a constant depth $(\varepsilon, \frac{1}{64})$-reduction from matrix determinant of $n \times n$ matrices on any distribution, to matrix rank on the distribution on $n^c \times n^c$ matrices obtained by choosing at random a rank r between 0 and n^c, and then generating a random matrix of that rank. The reduction uses an oracle for matrix multiplication.*

Proof. The problem again with the basic reduction outlined above is that even if the program oracle for matrix rank is reasonably good on average (i.e. a constant distance from perfectly correct), it could always be bad for at least one of the matrices D_a used in the reduction, and the reduction would fail with very high probability. To overcome this difficulty (using only an oracle for matrix multiplication), we amplify the success probability in each computation of $rank(D_a)$. This is done, similarly to the amplified reduction from inversion to determinant of Claim 6.6, using the matrix rank corrector. The amplified reduction computes the rank of each D_a by running the corrector, using two completely random matrices R and Q (this is because unlike the corrector outlined above, we cannot assume the reduction has access to $Random - Invertible$). If R and Q were random invertible matrices, then after taking the majority of many such calls, the reduction computes the rank of a correctly with all but polynomially small error probability. But now observe that if the field size is at least large enough constant (recall that for small constant size fields we can directly apply the composition theorem with the "simple" unamplified reduction), then with very high probability the random matrices R and Q are, in fact, invertible! The computation of each D_a's rank is successful with all but polynomially small probability, and the computation of A's determinant is correct w.h.p. Thus, if the rank program is a small enough (constant) distance from being correct, then the amplified reduction succeeds with all but an arbitrarily (polynomially) small error probability. The reduction computes the determinant of any matrix correctly with high probability, and thus it works for any distribution on matrices. ■

Applying the Composition Theorem, this gives a constant-depth tester and corrector for matrix rank using matrix multiplication oracles .

Corollary 6.11. *The matrix rank function over prime fields of polynomial size has an $\mathcal{AC}^0$ tester and corrector with a matrix multiplication oracle .*

Proposition 6.2. The matrix rank function over prime fields of polynomial size has an $\mathcal{AC}^0$ tester and corrector (ones that do not use any non standard oracle calls).

By Corollary 6.11 we get a constant depth tester and corrector that use an oracle for matrix multiplication. Again, the matrix multiplication oracles are easily replaced by vector-sum oracles (maintaining constant depth, as in Proposition 6.1). Furthermore, these vector-sum oracles can be replaced (while maintaining constant depth) by oracles to the (boolean) *vector-sum-equal* function. This function, on input a vector $\vec{v} = (v_1, \ldots, v_n)$ and a field element a, outputs 1 if the sum of $\vec{v}$'s entries is exactly a. To replace a vector-sum oracle with an oracle to *vector-sum-equal*, simply call *vector-sum-equal* (in parallel) with all possible values a, and output the one correct a for which *vector-sum-equal*'s output is 1. The number of calls to to *vector-sum-equal* required to replace each vector-sum oracle is linear in the field size (and thus polynomial in n).

We now apply the Composition Theorem with *vector-sum-equal* as the internal function and matrix rank as the external function. Condition 2 (the internal function "helps" test/correct the external function) is satisfied by the construction above.

Condition 1 (hardness of the external language for the internal language) is satisfied by the following (constant depth) reduction. For a vector $\vec{v} = (v_1, \ldots, v_n)$ and a field element a (in $GF(k)$), construct the $(n+1) \times (n+1)$ matrix $M_a^{\vec{v}}$ as follows:

$$M_a^{\vec{v}} = \begin{pmatrix} 1 & 0 & 0 & \ldots & 0 & v_1 \\ 0 & 1 & 0 & \ldots & 0 & v_2 \\ \vdots & & & & & \\ 0 & 0 & 0 & \ldots & 1 & v_n \\ 1 & 1 & 1 & \ldots & 1 & a \end{pmatrix}$$

It is not hard to verify that the rank of $M_a^{\vec{v}}$ is not full if and only if the sum of $\vec{v}$'s entries is a.

Condition 3 (testability and correctability of the internal language) is satisfied by the construction of a constant-depth tester and corrector for the *vector-sum-equal* function. Applying the Composition Theorem results in a (standard) constant-depth tester and corrector for matrix rank. ■

7 Acknowledgements

We thank Swastik Kopparty, Ronitt Rubinfeld, and Salil Vadhan for helpful and insightful conversations. Finally, we thank anonymous referees for many helpful comments.

References

[AB84] Miklós Ajtai and Michael Ben-Or. A theorem on probabilistic constant depth computation. In *Proceedings of the 16th Annual ACM Symposium on Theory of Computing*, pages 471–474, 1984.

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

[ASV02] Vikraman Arvind, K. V. Subrahmanyam, and N. V. Vinodchandran. The query complexity of program checking by constant-depth circuits. *Chicago J. Theor. Comput. Sci.*, 2002, 2002.

[Bab87] László Babai. Random oracles separate pspace from the polynomial-time hierarchy. *Inf. Process. Lett.*, 26(1):51–53, 1987.

[Bar89] David A. Mix Barrington. Bounded-width polynomial-size branching programs recognize exactly those languages in nc^1. *J. Comput. Syst. Sci.*, 38(1):150–164, 1989.

[Bar02] Boaz Barak. A probabilistic-time hierarchy theorem for "slightly non-uniform" algorithms. In *Proceedings of the 10th International Conference on Random Structures and Algorithms*, pages 194–208, 2002.

[BCLR04] Michael Ben-Or, Don Coppersmith, Michael Luby, and Ronitt Rubinfeld. Non-abelian homomorphism testing, and distributions close to their self-convolutions. In *Proceedings of APPROX-RANDOM*, pages 273–285, 2004.

[BDHM91] Gerhard Buntrock, Carsten Damm, Ulrich Hertrampf, and Christoph Meinel. Structure and importance of logspace-MOD-classes. In *Symposium on Theoretical Aspects of Computer Science*, pages 360–371, 1991.

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

[BFNW93] László Babai, Lance Fortnow, Noam Nisan, and Avi Wigderson. Bpp has subexponential time simulations unless exptime has publishable proofs. *Computational Complexity*, 3:307–318, 1993.

[BK95] Manuel Blum and Sampath Kannan. Designing programs that check their work. *Journal of the ACM*, 42(1):269–291, 1995.

[BLR93] Manuel Blum, Michael Luby, and Ronitt Rubinfeld. Self-testing/correcting with applications to numerical problems. *Journal of Computer and System Sciences*, 47(3):549–595, 1993.

[CW90] Don Coppersmith and Shmuel Winograd. Matrix multiplication via arithmetic progressions. *J. Symb. Comput.*, 9(3):251–280, 1990.

[FKN94] Uriel Feige, Joe Kilian, and Moni Naor. A minimal model for secure computation (extended abstract). In *STOC*, pages 554–563, 1994.

[Fre79] Rusins Freivalds. Fast probabilistic algorithms. In *MFCS*, pages 57–69, 1979.

[FS04] Lance Fortnow and Rahul Santhanam. Hierarchy theorems for probabilistic polynomial time. In *FOCS*, pages 316–324, 2004.

[FSS84] Merrick L. Furst, James B. Saxe, and Michael Sipser. Parity, circuits, and the polynomial-time hierarchy. *Mathematical Systems Theory*, 17(1):13–27, April 1984.

[FST05] Lance Fortnow, Rahul Santhanam, and Luca Trevisan. Hierarchies for semantic classes. In *STOC*, pages 348–355, 2005.

[GGH+07] Shafi Goldwasser, Dan Gutfreund, Alexander Healy, Tali Kaufman, and Guy N. Rothblum. Verifying and decoding in constant depth. In *STOC*, pages 440–449, 2007.

[GSTS03] Dan Gutfreund, Ronen Shaltiel, and Amnon Ta-Shma. Uniform hardness vs. randomness tradeoffs for arthur-merlin games. In *IEEE Conference on Computational Complexity*, pages 33–47, 2003.

[IK02] Yuval Ishai and Eyal Kushilevitz. Perfect constant-round secure computation via perfect randomizing polynomials. In *ICALP*, pages 244–256, 2002.

[IKW02] Russell Impagliazzo, Valentine Kabanets, and Avi Wigderson. In search of an easy witness: exponential time vs. probabilistic polynomial time. *J. Comput. Syst. Sci.*, 65(4):672–694, 2002.

[Kil88] Joe Kilian. Founding cryptography on oblivious transfer. In *STOC*, pages 20–31, 1988.

[Lip91] Richard J. Lipton. New directions in testing. *Proceedings of DIMACS workshop on distributed computing and cryptography*, 2:191–202, 1991.

[Ran93] Dana Randall. Efficient generation of random nonsingular matrices. *Random Struct. Algorithms*, 4(1):111–118, 1993.

[Rub96] Ronitt Rubinfeld. Designing checkers for programs that run in parallel. *Algorithmica*, 15(4):287–301, 1996.

[San07] Rahul Santhanam. Circuit lower bounds for merlin-arthur classes. In *STOC*, pages 275–283, 2007.

[Sha92] Adi Shamir. IP = PSPACE. *Journal of the ACM*, 39(4):869–877, 1992.

[SU07] Ronen Shaltiel and Christopher Umans. Low-end uniform hardness vs. randomness tradeoffs for am. In *STOC*, pages 430–439, 2007.

[TV07] Luca Trevisan and Salil P. Vadhan. Pseudorandomness and average-case complexity via uniform reductions. *Computational Complexity*, 16(4):331–364, 2007.

[WB97] Hal Wasserman and Manuel Blum. Software reliability via run-time result-checking. *Journal of the ACM*, 44(6):826–849, 1997.