Delegating Computation Reliably: 
Paradigms and Constructions 

by 

Guy N. Rothblum 

Submitted to the Department of Electrical Engineering and Computer Science 
on September 4, 2009, in partial fulfillment of the 
requirements for the degree of 
Doctor of Philosophy 

Abstract 

In an emerging computing paradigm, computational capabilities, from processing power to storage capacities, are offered to users over communication networks as a service. This new paradigm holds enormous promise for increasing the utility of computationally weak devices. A natural approach is for weak devices to delegate expensive tasks, such as storing a large file or running a complex computation, to more powerful entities (say servers) connected to the same network. While the delegation approach seems promising, it raises an immediate concern: when and how can a weak device verify that a computational task was completed correctly? This practically motivated question touches on foundational questions in cryptography and complexity theory.

The focus of this thesis is verifying the correctness of delegated computations. We construct efficient protocols (interactive proofs) for delegating computational tasks. In particular, we present:

- A protocol for delegating any computation, where the work needed to verify the correctness of the output is linear in the input length, polynomial in the computation’s depth, and only poly-logarithmic in the computation’s size. The space needed for verification is only logarithmic in the computation size. Thus, for any computation of polynomial size and poly-logarithmic depth (the rich complexity class $\mathcal{NC}$), the work required to verify the correctness of the output is only quasi-linear in the input length. The work required to prove the output’s correctness is only polynomial in the original computation’s size. This protocol also has applications to constructing one-round arguments for delegating computation, and efficient zero-knowledge proofs.

- A general transformation, reducing the parallel running time (or computation depth) of the verifier in protocols for delegating computation (interactive proofs) to be constant.

Next, we explore the power of the delegation paradigm in settings where mutually distrustful parties interact. In particular, we consider the settings of checking the correctness
of computer programs and of designing error-correcting codes. We show:

- A new methodology for checking the correctness of programs (program checking), in which work is delegated from the program checker to the untrusted program being checked. Using this methodology we obtain program checkers for an entire complexity class (the class of $\mathcal{NC}$-computations that are $\mathcal{NC}^1$-hard), and for a slew of specific functions such as matrix multiplication, inversion, determinant and rank, as well as graph functions such as connectivity, perfect matching and bounded-degree graph isomorphism.

- A methodology for designing error-correcting codes with efficient decoding procedures, in which work is delegated from the decoder to the encoder. We use this methodology to obtain constant-depth ($\mathcal{AC}^0$) locally decodable and locally-list decodable codes. We also show that the parameters of these codes are optimal (up to polynomial factors) for constant-depth decoding.

Thesis Supervisor: Shafi Goldwasser
Title: RSA Professor of Electrical Engineering and Computer Science
To my parents, Naomi and Uriel.
Acknowledgments

I am forever grateful to the mentors, colleagues, friends and family that have led, accompanied and supported me in my journey through graduate school.

First and foremost, I am unbelievably lucky to have had Shafi Goldwasser as my thesis advisor. I cannot imagine a more exciting, innovative or inspiring mentor. Shafi’s relentlessly creative thinking, her uncanny ability to produce fascinating research problems, broad and deep knowledge, perspective, and enthusiasm made research a real joy. Beyond being such an inspiring scientist, Shafi’s warmth, encouragement, friendship and caring about any problem I encountered, big or small, were invaluable throughout graduate school. I am forever indebted to her for her inspiration, influence and support. They are in every page of this thesis.

When Shafi was away from MIT, Salil Vadhan agreed to serve as my co-advisor; in particular, he hosted me for a wonderful semester in Berkeley. Every time I talk with Salil, whatever problem comes up, the issues crystallize and their fundamentals come sharply into focus. I have benefited enormously from his profound and clear thinking and from his thorough approach to research. I thank him for all that he has taught me and for his unending support and generosity.

Moni Naor, as my M.Sc. advisor at the Weizmann Institute and throughout my Ph.D. studies, taught me how to approach and enjoy research. It is impossible to speak with Moni without learning something new and exciting. I am grateful for the many enjoyable and enlightening conversations we had and for all that I learned from him.

In my last year of graduate school, I had the privilege and pleasure of working with Cynthia Dwork. Cynthia opened for me the doors into the world of privacy research and repeatedly amazed me with her clear and creative thinking and her unerring grasp of the big picture. I thank her for all that she has taught me and for all the advice she has shared.

I thank Silvio Micali for teaching me about cryptography, game theory, and computer science, and for his generous advice and assistance throughout graduate school. I am grateful to Ron Rivest for many enlightening conversations and comments. Special thanks to Silvio and Ron also for serving on my thesis committee. I am grateful to the many other teachers
at MIT, Weizmann, Berkeley and elsewhere who have shaped my scientific development, especially to Ran Canetti, Oded Goldreich, Piotr Indik, Yuval Ishai, Erez Petrank, Ran Raz, Omer Reingold, Ronitt Rubinfeld, Madhu Sudan and Luca Trevisan.

None of the work that is included in this thesis would have happened without the truly remarkable students and postdocs I was privileged to interact with in graduate school. I am most indebted to my co-authors on works included in this thesis. The results in Chapter 3 are joint work with Shafi and Yael Tauman Kalai. From day one, working with Yael was fruitful, fascinating, and a real pleasure. I am grateful for all that I have learned from her, and for her friendship and help. The results in Chapters 4, 5 and 6 are based on joint works with Shafi, Danny Gutfreund, Alex Healy and Tali Kaufman. I thank Danny, Alex and Tali for numerous enlightening conversations and countless enjoyable lunches.

Here an apology is due, as these pages are too short to do justice to the unsung friends and colleagues without whom graduate school would never have been so exciting or enjoyable and this thesis would never have been written. These include the members of the theory and cryptography groups at MIT, Weizmann, Berkeley and Microsoft Research, and my dear friends, who keep up with me through thick, thin and endless relocations. Know that you have made all the difference. I am endlessly grateful.

Most importantly, as I made my journey through graduate school, often as intense physically as it was intellectually, my anchor was always my family. I cannot thank them enough for their unconditional love, their unwavering support, and their unshakable faith in me. Roni, for always making me laugh and always making sure I am enjoying myself. There was no one that I missed more when I was away. Alex and Modi, for always caring so much and always welcoming me into their home with open arms, no matter how late or how frequently I arrive. My embarrassingly proud grandfather Tedi, I wish he could read these pages. More than anyone else, my parents, Naomi and Uri, to whom my gratitude is beyond expression.
## Contents

1 Introduction .............................................. 15  
   1.1 Verifying Interactive Proofs for Efficient Computations ........ 19  
       1.1.1 1-round Arguments for Certifying Computations ........ 25  
       1.1.2 Succinct Zero-Knowledge Proofs .................. 26  
       1.1.3 Public-Coin Log-Space Verification ................. 27  
       1.1.4 Technical Highlights ............................ 27  
   1.2 Verifying Interactive Proofs in Constant Depth ............... 30  
       1.2.1 Technical Overview and Comparison to Related Work .... 32  
   1.3 Delegation in Program Checking and Correcting ................ 36  
       1.3.1 Further Remarks .................................. 40  
       1.3.2 More Technical Tools ............................ 42  
   1.4 Delegation in Error Correcting Codes ..................... 43  
       1.4.1 Technical Highlights ............................ 47  
   1.5 Thesis Roadmap ...................................... 48  

2 Definitions and Preliminaries ......................... 51  
   2.1 Circuit and Complexity Classes ........................ 51  
   2.2 Interactive Proofs .................................. 52  
   2.3 Randomized images .................................. 54  

3 Verifying Interactive Proofs for Efficient Computations .... 61  
   3.1 Overview ........................................... 61
4.1.1 Interactive Proofs: New Definitions ................................. 153
4.2 Verification in Constant Parallel Time ............................... 154
  4.2.1 General Proof Systems ........................................... 158
4.3 Negative Results ....................................................... 163

5 Delegation in Program Checking ........................................ 167
  5.1 Introduction .......................................................... 167
    5.1.1 Program Checking and its Impact ............................. 168
    5.1.2 New Checkers, Testers & Correctors ......................... 171
    5.1.3 New Techniques and Tools .................................... 171
    5.1.4 Other Contributions and Comments ............................ 173
    5.1.5 Other Related Work ............................................ 174
  5.2 Definitions and Preliminaries ..................................... 174
    5.2.1 Definitions: Checkers, Testers and Correctors ............. 174
  5.3 Composing Checkers, Testers and Correctors ..................... 177
    5.3.1 Composing Program Checkers ................................ 177
    5.3.2 Extensions ..................................................... 181
    5.3.3 Composing Program Testers and Correctors ................ 182
  5.4 Checkers, Testers and Correctors for Complete Languages ..... 192
    5.4.1 From randomized self-images to program checkers ......... 192
    5.4.2 Checkers, testers and correctors for complete languages 193
  5.5 Checkers, Testers and Correctors for a Complexity Class ..... 196
  5.6 Program Libraries Revisited: Checkers for Matrix Functions 200
    5.6.1 Outline ......................................................... 200
    5.6.2 Matrix Multiplication ........................................ 205
    5.6.3 Matrix Inversion ............................................... 209
    5.6.4 Matrix Determinant ............................................ 213
    5.6.5 Matrix Rank .................................................... 223
6 Delegation in Error Correcting Codes

6.1 Introduction ............................................. 229
   6.1.1 This Work ........................................... 230
   6.1.2 Related Work ....................................... 232
   6.1.3 On the Choice of Parameters ....................... 234

6.2 Preliminaries ......................................... 235
   6.2.1 Locally Decodable and Locally List-Decodable Codes .......... 235
   6.2.2 Majority and Related Functions ....................... 237

6.3 Transformations: Reducing the Complexity of Decoding ........ 238
   6.3.1 Unique Local Decoding ............................... 238
   6.3.2 Local List Decoding ................................. 241

6.4 Explicit Constructions ................................ 248
   6.4.1 A (Uniquely) Locally Decodable Binary Code .............. 249
   6.4.2 A List-Locally Decodable Non-Binary Code ................ 254
   6.4.3 A Locally List-Decodable Binary Code .................. 256

6.5 Local-List-Decoding Requires Computing Majority ............. 264

6.6 Hardness Amplification ................................ 274
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3-1</td>
<td>Low degree test ((P_{LDT}(\pi), V_{LDT}^\pi))</td>
<td>83</td>
</tr>
<tr>
<td>3-2</td>
<td>Sum-check protocol ((P_{SC}(f), V_{SC}^f)) [LFKN92, Sha92]</td>
<td>85</td>
</tr>
<tr>
<td>5-1</td>
<td>Freivalds’ Checker</td>
<td>206</td>
</tr>
<tr>
<td>5-2</td>
<td>Matrix Multiplication Corrector</td>
<td>206</td>
</tr>
<tr>
<td>5-3</td>
<td>Matrix Multiplication Tester</td>
<td>207</td>
</tr>
<tr>
<td>5-4</td>
<td>Matrix Inversion Corrector</td>
<td>210</td>
</tr>
<tr>
<td>5-5</td>
<td>Matrix Inversion Tester</td>
<td>211</td>
</tr>
<tr>
<td>5-6</td>
<td>Generating a random invertible matrix</td>
<td>213</td>
</tr>
<tr>
<td>5-7</td>
<td>Matrix Determinant Corrector</td>
<td>215</td>
</tr>
<tr>
<td>5-8</td>
<td>Matrix Determinant Tester</td>
<td>216</td>
</tr>
<tr>
<td>5-9</td>
<td>Matrix Rank Corrector</td>
<td>224</td>
</tr>
<tr>
<td>5-10</td>
<td>Matrix Rank Tester</td>
<td>224</td>
</tr>
<tr>
<td>6-1</td>
<td>Algorithm (A_1)</td>
<td>270</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

In a wide array of modern computing setting, parties want to accomplish tasks (computational or otherwise) that they cannot or do not want to perform themselves. A natural approach in such situations is for such a party, the delegator, to delegate the task to another entity, a delegatee, which will perform it for them. One increasingly prevalent such setting is that of cloud computing, where users (delegators) outsource their data and even their computations to remote servers (delegatees). This effective approach faces serious obstacles. In this thesis we focus on the challenge of guaranteeing reliability. The question we ask is how can a delegator ensure that a delegatee performed computational tasks correctly, without performing them itself?

This question is of interest both from a foundational and a practical perspective. We study it not only as it pertains to today's emerging technology, but also in a host of computational settings. We focus on settings where mutually distrustful or error-prone parties interact and computational tasks may be outsourced from one party to another. Our interest lies in exploring both how reliability can be guaranteed for delegated computations, and when the approach of delegating computation can be useful for increasing the efficiency of computational systems. The settings we study include (i) Interactive proofs, where a malicious prover (delegatee) may be arbitrarily powerful, as well as interactive arguments, where malicious provers are assumed to be computationally bounded, (ii) Checking the correctness
of programs. We view the (untrusted) program whose correctness is being checked as a computational resource, and delegate work from the checker to the program being checked (while ensuring reliability), and (iii) Efficient decoding of error correcting codes, where we delegate work from the decoder to the encoder using the unreliable channel.

This thesis contains the following results:

- In Section 1.1 we study interactive proofs for delegating computation. In this setting the verifier is the delegator who wants to be convinced of an output’s correctness, and the prover is the delegatee who wants to generate a proof of correctness. We ask which polynomial-time computations have such interactive proofs where the prover’s running time is polynomial and the proof can be verified super-efficiently (in terms of running time and space).

Our main contribution is a new interactive proof protocol for general uniform computations. The running time of the (honest) prover in this protocol is polynomial in the computation’s circuit-size. The running time of the verifier is linear in the input length, polynomial in the computation’s circuit depth, and only poly-logarithmic in the circuit’s size. The space used by the verifier is logarithmic in the computation’s size. The communication is polynomial in the computation’s depth and poly-logarithmic in its size.

This new protocol has applications to many several questions in the theory of interactive proofs and arguments. We use it, together with a transformation of Kalai and Raz [KR09], to construct efficient one-round computationally sound arguments for general computations under standard cryptographic intractability assumptions. Previously such results were known only by the work of Micali [Mic94] in the random oracle model. The new protocol also allows us to build new succinct zero-knowledge proofs for $NP$ languages, where the communication complexity is linear in the witness length and the complexity of verifying is dominated by the depth of the language’s $NP$ relation (rather than its size). In addition, we obtain a complete characterization of the power of interactive proofs with public-coin logarithmic-space verifiers.
In Section 1.2 we consider another notion of efficiently verifying that tasks were performed correctly: constant depth (or constant parallel running time) verification. We study the power of interactive proofs with constant-depth verifiers, considering general (not necessarily polynomial-time) computations, but also focusing on polynomial-time computations as a special case.

Our main contribution here is a general transformation from an arbitrary interactive proof protocol, into one where the verifier runs in constant depth. Thus, we show that interactive proofs with constant-depth verifiers are as powerful as general interactive proofs. This transformation preserves (up to polynomial factors) the running time of the (honest) prover, and in particular it allows us to obtain an interactive proof with constant-depth verification and an efficient (honest) prover for any efficiently computable language.

The settings of interactive proofs and arguments are perhaps the most immediate ones for studying the question of delegating computation reliably. However, this question arises naturally in many computational settings where mutually distrustful parties interact. We re-examine two central scenarios in the theory of computing through the lens of delegating computation: (i) the question of checking, testing and correcting programs, where we delegate computation form the checker (or tester or corrector) to the program being checked, and (ii) the question of designing error correcting codes, where we delegate computation from the decoder to the encoder. We show:

- Program Checking: Here the goal is to check the correctness of an untrusted program on a specific input. There are two components to the system: a weak checker that tries to verify correctness, and a “stronger” program, which allegedly computes a complex function. This program might be faulty or malicious and compute some other arbitrary function. We introduce the idea of delegating computation (in a reliable manner) from the checker to the (potentially faulty) program being checked. The challenge is first finding useful computations that can be computed by the program (which, even if it
is not faulty, only computes a fixed function), and then finding reliable methods for delegating these computations (even if the program is malicious).

This approach yields new checkers (and testers and correctors, see below). In particular, we obtain program checkers for an entire complexity class (the class of \( \mathcal{NC} \)-computations that are \( \mathcal{NC}^1 \)-hard), and for a slew of specific functions such as matrix multiplication, inversion, determinant and rank, as well as graph functions such as connectivity, perfect matching and bounded-degree graph isomorphism. Moreover, this approach and the results obtained shed new light on long-standing questions in the field of program checking. See Section 1.3 for details.

- **Error-Correcting Codes**: Here the goal is to encode a message so that even if many of its bits are corrupted (by a noisy channel) a decoding procedure can still recover the original message. We again examine the two components of this system: the encoder and decoder, which communicate over the noisy channel. We introduce the idea of delegating computation from the decoder to the encoder. The challenge is that any helpful information that the encoder wishes to convey to the decoder might be corrupted by the noisy channel, and so we need error-resilient methods for conveying computationally helpful information.

Using this methodology we obtain new locally decodable and locally list-decodable codes with constant-depth decoders (decoding known codes with similar parameters required implementing more complex operations that provably cannot be implemented in constant depth). We also show that these codes are optimal up to polynomial factors. See Section 1.4 for further discussion.

We proceed with an overview of results for each of the above contributions.
1.1 Verifying Interactive Proofs for Efficient Computations

We study interactive proofs that can be used for delegating efficient (i.e. polynomial-time) computations. The general setting is of several computational devices of differing computational abilities interacting with each other over a network. Some of these devices are computationally weak due to various resource constraints. As a consequence there are tasks, which potentially could enlarge a device’s range of application, that are beyond its reach. A natural solution is to delegate computations that are too expensive for one device, to other devices which are more powerful or numerous and connected to the same network. This approach comes up naturally in today’s and tomorrow’s computing reality as illustrated in the following two examples.

1. Large Scale Distributed Computing. The idea of Volunteer Computing is for a server to split large computations into small units, send these units to volunteers for processing, and reassemble the result (via a much easier computation). The Berkeley Open Infrastructure for Network Computing (BOINC) [And03, And04] is such a platform whose intent is to make it possible for researchers in fields as diverse as physics, biology and mathematics to tap into the enormous processing power of personal computers around the world. A famous project using the BOINC platform is SETI@home [SET07, SET99], where large chunks of radio transmission data are scanned for signs of extraterrestrial intelligence. Anyone can participate by running a free program that downloads and analyzes radio telescope data. Thus, getting many computers to pitch into the larger task of scanning space for the existence of extraterrestrial intelligence, and getting people interested in science at the same time. Another example of a similar flavor is the Great Internet Mersenne Prime Search [Mer07], where volunteers search for Mersenne prime numbers and communicate their findings to a central server.

2. Weak Peripheral Devices. More and more, small or cheap computational devices with limited computational capabilities, such as cell-phones, printers, cameras, security access-
cards, music players, and sensors, are connected via networks to stronger remote computers whose help they can use. Consider, for example, a sensor that is presented with an access-card, sends it a random challenge, and receives a digital signature of the random challenge. The computation required to verify the signature involves public-key operations which are too expensive both in time and space for the sensor to run. Instead, it could interact with a remote mainframe (delegatee), which can do the computation.

The fundamental problem that arises is: how can a delegator verify that the delegatees performed the computation correctly, without running the computation itself? For example, in the volunteer computing setting, an adversarial volunteer may introduce errors into the computation, by claiming that a chunk of radio transmissions contains no signs of extraterrestrial intelligence. In the Mersenne Prime search example, an adversary may claim that a given range of numbers does not contain a Mersenne prime. Or in the sensor example, the communication channel between the main-frame and the sensor may be corrupted by an adversary.

All would be well if the delegatee could provide the delegator with a proof that the computation was performed correctly. The challenge is that for the whole idea to pay off, it is essential that the time to verify such a proof of correctness be significantly smaller than the time needed to run the entire computation. At the same time, the delegatee should not invest more than a reasonable amount of time in this endeavor. Interactive proofs with efficient provers (the delegatees) and super-efficient verifiers (the delegators) provide a natural solution to the problem of delegating computation reliably.

**Interactive Proofs for Delegating Computation.** Efficient proof verification lies at the heart of complexity theory. Classically, this was captured by the class \( \mathcal{NP} \), where a deterministic polynomial time verification procedure which receives the proof (witness), a certificate of polynomial length, and verifies its validity. For example, to prove that a graph contains a Hamiltonian cycle, the proof is the cycle and the (deterministic, non-interactive,

---

\(^1\)With regard to the Mersenne Prime example, we note that current methods for verifying the output of polynomial time deterministic primality tests [AKS04] are not significantly faster than running the test itself.
polynomial time) verification procedure verifies that it is indeed a Hamiltonian cycle in the
input graph. Interactive proof systems, introduced by [GMR89, Bab85], extend the classic
notion of proof verification by considering randomized and interactive (polynomial time)
verification procedures. The proof, rather than being written down non-interactively, is an
interactive protocol with a prover. Dishonest provers might employ an arbitrarily malicious
adaptive strategy, and soundness is still required to hold against such malicious dishonest
provers (i.e. the verifier should, with high probability over its coins, reject inputs that are
not in the language).

We want to use interactive proofs to prove the correctness of delegated computations.
Namely, the statement to be proved is that the delegated computation was executed correctly
(i.e. that the given output is the correct one); the delegator is the verifier in the interactive
proof; the delegatee is the prover in the interactive proof, who convinces the delegatee that
he performed the computation correctly (and runs in polynomial time). In these interactive
proofs the (honest) prover should run in polynomial time (as we want efficient parties to be
able to use them), and the verification should be super-efficient; in particular, verification
should be much more efficient that running the computation (as the point is to farm out the
computation).

Previous work studied interactive proofs through the lenses of both cryptography and
complexity theory. These settings both differ from ours as discussed below.

**Complexity-theoretic setting:** Most work has focused on studying the expressive power
of interactive proofs under various resource restrictions (e.g. verification time, space, depth,
rounds or randomness). The complexity of proving has received less attention. Indeed, since
research focused on proofs for intractable languages, the honest prover is often\(^2\) assumed
to be able to perform intractable computations in the interest of efficient verifiability. In
Arthur-Merlin games, the honest prover is accordingly named after Merlin, a computationally
unbounded magician.

\(^2\)We note that there are important exceptions to the above, e.g. the work of Beigel, Bellare, Feigenbaum
and Goldwasser [BBFG91] on competitive proof systems.
Cryptographic setting: In the cryptographic study of interactive proofs, most works consider protocols where all parties must run in polynomial time. The focus remains, however, on intractable (usually $\mathcal{NP}$) languages, such as deciding quadratic-residuosity modulo a composite number. To allow the honest prover to perform computations otherwise impossible in polynomial time, he or she can use auxiliary secrets, e.g. the factorization of the input modulus in the quadratic residuosity example. This model is reasonable in protocol settings where the input is generated by the prover himself. The prover can generate the input along with an auxiliary secret which enables him prove non-$\mathcal{BPP}$ properties. However, in settings where the input is generated by an external source, an efficient prover does not have access to auxiliary information about the input.

Our Setting: We embark on the study of interactive proofs for the real world, where all parties (both the verifier and the prover) are efficient and the proofs are for efficiently computable languages. Thus, we replace the unbounded (honest) prover with a bounded prover who is limited to running probabilistic polynomial-time computations. We think of the input to the interactive proof as dictated by an outside source, possibly even by the verifier. Thus, the prover has no auxiliary information to help him in the proving task.

Clearly, if both the prover and the verifier are efficient, then the language is tractable (in $\mathcal{BPP}$). This may seem puzzling at first glance, since usually one allows the verifier to run arbitrary polynomial-time computations, and thus it could compute on its own whether or not the input is in the language! This obviously is not very interesting. Indeed, we want verification to be considerably faster than computing.

The question we ask in this work is which polynomial-time computable languages have interactive proofs with a super-efficient verifier and an efficient prover. We emphasize that although we aim for the honest prover to be efficient, we still require the soundness of the proof system to hold unconditionally. Namely, we make no assumptions on the computational power of a dishonest prover.
Overview of Our Results: Our main technical theorem gives a public coin interactive proof for general uniform computations:

**Theorem 1.1.1.** Let $L$ be a language that can be computed by a family of $O(\log(S(n)))$-space uniform boolean circuits of size $S(n)$ and depth $d(n)$. $L$ has an interactive proof where:

1. The prover runs in time $\text{poly}(S(n))$. The verifier runs in time $n \cdot \text{poly}(d(n), \log S(n))$ and space $O(\log(S(n)))$. Moreover, if the verifier is given oracle access to the low-degree extension of its input, then its running time is only $\text{poly}(d(n), \log S(n))$.

2. The protocol has perfect completeness and soundness $1/2$. 4

3. The protocol is public-coin, with communication complexity $d(n) \cdot \text{polylog}(S(n))$.

In particular, for languages computable by log-space uniform NC (circuits of polylog($n$) depth), the prover is efficient, the verifier runs in time $n \cdot \text{polylog}(n)$ and space $O(\log(n))$, and the communication complexity is polylog($n$). The protocol of Theorem 1.1.1 can be used for reliably delegating computations.

We note also that the above protocol immediately yields a method for delegating a collection of many computations with low amortized cost. This is an immediate consequence of the protocol’s complexity being proportional to the computation depth. Consider a set of $m$ computations whose correctness we wish to verify. Potentially these computations are $m$ different circuits evaluated on $m$ different inputs and with $m$ different outputs. Viewing these $m$ computations as a single larger circuit, the circuit depth is only increased by a log $m$ additive factor beyond the depth of the deepest computation in the set. Thus, the cost of verifying $m$ computations (once all $m$ computations and their inputs and outputs are specified) is only polylog($m$) times higher than the cost of verifying the most expensive computation in the set.

---

3 A circuit family is $s(n)$-space uniform if there exists a Turing Machine that on input $1^n$ runs in space $O(s(n))$ and outputs the circuit for inputs of length $n$. A circuit family is $L$-uniform if it is log-space uniform.

4 Throughout this work we work with constant soundness for interactive proof systems. This is easily amplified via parallel or sequential repetition.
Comparison to Prior Work on Interactive Proofs. We emphasize that Theorem 1.1.1 improves previous work on interactive proofs, including the works of Lund, Fortnow, Karloff and Nissan [LFKN92], Shamir [Sha92], and Fortnow and Lund [FL93] in terms of the honest prover’s running time. In particular, Corollary 3.1.1 gives efficient honest provers, whereas the honest provers in previous results run in super-polynomial time (even for log-space languages). Both of the works [LFKN92, Sha92] address complete languages for $\#P$ and for $PSPACE$, and thus naturally the honest prover needs to perform non-polynomial time computation. Scale-down of the protocols to $P$ or even to $L$ retains the non-polynomial time provers. In particular, for a time $t(n)$ and space $s(n)$ computation, prior work gives interactive proofs where the (honest) provers running time is $t(n)\text{poly}(s(n))$. The work of Fortnow and Lund [FL93], using algebraic methods extending [LFKN92, Sha92], on the other hand, does explicitly address the question of interactive proofs for polynomial time languages and in particular $NC$. They show how to improve the space complexity of the verifier, in particular achieving log-space and poly-time verifiers for $NC$ computations. Their protocol, however, has a non-polynomial time prover as in [LFKN92, Sha92].

Our work puts severe restrictions on the sequential runtime (and space) of the verifier (and, in Section 1.2, on its computation depth). This continues a sequence of works which investigated the power of interactive proofs with weak verifiers (but often with unbounded provers). Dwork and Stockmeyer [DS92a, DS92b] investigated the power of finite state verifiers (with and without zero-knowledge). Condon and Ladner [CL88], Condon and Lipton [CL89], and Condon [Con91] studied space (and time) bounded verifiers. Kilian [Kil88b] considered zero-knowledge for space-bounded verifiers. Fortnow and Sipser (see results in [For89]) and Fortnow and Lund [FL93] focused on public-coin restricted space verifiers.

Prior Work in Other Models. Variants of the question of checking the correctness of computations have been studied previously in several settings outside of the interactive proof model. These studies include the work of Babai, Fortnow, Levin, and Szegedy [BFLS91] in the Holographic Proofs model (or alternatively the PCP model), and the works of Micali [Mic94] and Kilian [Kil95] on computationally sound argument systems. These works
raise similar goals to ours, but in their respective models, requiring super-efficient verifiability, and efficient provability (polynomial time in the non-deterministic time complexity of accepting the input). We elaborate on these seminal related works in Section 3.1.1. In contrast, our work is in the standard interactive proof model [GMR89] where soundness is achieved unconditionally, making no assumptions on the power of the dishonest prover (as in [Mic94, Kil95]), nor making assumptions on the non-adaptivity of the dishonest prover (as in [BFLS91]). Prior work has also considered delegating specific useful computations (such as, say, modular exponentiation), especially in cryptographic settings. See for example Hohenberger and Lysyanskaya [HL05] for a discussion of concerns that arise and prior work.

**Further Applications:** Beyond its direct applications to delegating computation, Theorem 1.1.1 also allows us to make progress on several questions, as described below. As further applications, we also obtain new Interactive PCPs [KR08] and Probabilistically Checkable Arguments [KRO9]. See Chapter 3 for details.

### 1.1.1 1-round Arguments for Certifying Computations

The above protocol of Theorem 1.1.1 requires multiple rounds of communication (the number of rounds is polynomial in the depth and poly-logarithmic in the size of the computation). We find it very interesting to consider non-interactive protocols for delegating computation, where the prover only needs to send a single message to the verifier. This message can contain the output of the computation and a proof, and no interaction is necessary (beyond perhaps some offline setup phase). More generally, we seek to reduce the round complexity of protocols for delegating computation. The only known non-interactive (or 1-round) general protocol for delegating computation is in Micali’s work [Mic94] on CS proofs. These results, however, hold only in the random oracle model, and it is not known how to translate such results to protocols that are secure in the real world (see [CGH04, GK03]). We seek instead to base protocols on standard cryptographic intractability assumptions.

We obtain a one-round protocol using the protocol of Theorem 1.1.1 together with a recent
transformation of Kalai and Raz [KR09] from public-coin interactive proofs to one-round arguments. Furthermore, this protocol is “almost” non-interactive (see below). The protocol has computational soundness against polynomial-time cheating provers. This computational soundness is based on a standard cryptographic assumption, the existence of a PIR scheme with polylog communication (see e.g. the work of Cachin, Micali and Stadler [CMS99]).

The 1-round protocol we obtain is a computationally sound argument system for general (uniform) computations. The (honest) prover’s work is polynomial in the computation size, the verifier’s work is linear in the input length and polynomial in the security parameter, the computation depth and only polylogarithmic in the computation size. The communication is polynomial in the computation’s depth and the security parameter and polylogarithmic in the computation size. Moreover, the verifier’s message in this protocol is completely independent of the input and the computation whose correctness is being proven. Thus, the protocol is “almost” non-interactive in the sense that the verifier can compute this message in advance and independently of the computation, and no further interaction is needed once the computation becomes known.

See Theorem 3.1.2 of Section 3.1.2 for a precise statement. We note that, as in the main result of Theorem 1.1.1, this protocol can be used to amortize the cost of delegating many computations in a natural way.

1.1.2 Succinct Zero-Knowledge Proofs

Zero-knowledge interactive proofs are central in the study of cryptography and also in complexity theory. In this setting we have an \( \mathcal{NP} \) relation \( R \) which takes as input an instance \( x \) and a witness \( w \). A prover (who knows both \( x \) and \( w \)) wants to convince a verifier (who knows only \( x \) and does not know \( w \)) in zero-knowledge that \( R(x, w) = 1 \). The verifier should learn nothing from this interaction beyond the validity of the prover’s assertion.

We improve known results on zero-knowledge proofs significantly. We show that any \( \mathcal{NP} \) language whose relation \( R \) can be verified by depth \( d \) circuits has a zero-knowledge proof where the communication complexity is linear in the witness length and polynomial
in the security parameter and the depth \( d \). Previously zero-knowledge proofs with communication proportional to the witness length were known only for languages whose relation \( R \) is computable in \( \mathcal{AC}^0 \) (constant depth) [IKOS07, KR08]. For general \( \mathcal{NP} \) languages outside this class, the best protocols have communication polynomial in the circuit size of \( R \) [GMW91, Blu87]. Our protocol can be based on any one-way function. Moreover, for \( \mathcal{NP} \) languages whose relations can be verified in \( (L-) \) uniform depth \( d \), the verifier in the new protocol runs in time that is linear in the input size and polynomial in \( d \) and the security parameter. Previous work on zero knowledge proofs for general classes of \( \mathcal{NP} \) languages did not achieve verifier computation time that is lower than the size of \( R \). See Theorems 3.1.5 and 3.1.6 in Section 3.1.5 for precise statements.

1.1.3 Public-Coin Log-Space Verification

Finally, we consider the power of interactive proofs with public-coin, log-space, poly-time verifiers. We show such a proof system for every language in \( \mathcal{P} \). This gives a tight characterization for the expressive power of interactive proof systems with such verifiers, settling an open question [Con91, For89, For89, FL93]. This result is a corollary of Theorem 1.1.1, using the fact that languages in \( \mathcal{P} \) have \( (L\text{-uniform}) \) poly-size circuits. See Corollary 3.1.3 in Section 3.1.3 for a precise statement and further discussion.

1.1.4 Technical Highlights

We conclude with an overview of the techniques used to obtain the main result of Theorem 1.1.1. In a nutshell, our goal is to reduce the verifier’s runtime to be proportional to the depth of the circuit \( C \) being computed, rather than its size, while maintaining (honest) prover running time that is proportional to the circuit’s size.

In previous work, spanning both the single and multi prover models [LFKN92, Sha92, BFL91, KR08], the entire computation of the underlying machine is arithmetized and turned into an algebraic expression whose value is claimed and proved by the prover.

We assume without loss of generality that the circuit \( C \) is a depth \( d \) arithmetic circuit
in a layered form where there are as many layers as the depth of the circuit. Departing from previous work, here we instead employ an interactive protocol that closely follows the (parallelized) computation of $C$, layer by layer, from the output layer to the input layer, numbering the layers in increasing order from the top (output) of the circuit to the bottom (input) of the circuit.\footnote{I.e., layer 0 is the output layer, and layer $d$ is the input layer.} We consider a redundant high-distance encoding (a low-degree extension) of the values of circuit’s gates in each layer. The verifier has no time to compute points in the low-degree extension of the computation on input $x$ in layer $i$: this is the low-degree extension (a high distance encoding) of the vector of values that the gates in the circuit’s $i$-th layer take on input $x$, and to compute it one needs to actually evaluate $C$, which we want to avoid! Thus, the low-degree extension of the $i$-th layer, will be instead supplied by the prover. Of course, the prover may cheat. Thus, each phase of the protocol lets the verifier reduce verification of a single point in the low-degree extension of an advanced step (layer) in the parallel computation, to verification of a single point in the low-degree extension of the previous step (layer). This process is repeated iteratively (for as many layers as the circuit has), until at the end the verification has been reduced to verifying a single point in the extension of the first step in the computation. In the first step of the computation (the input layer), the only information “computed” is the input $x$, the verifier can compute the low degree extension of the input $x$ on its own in nearly-linear time.

**Going from Layer to Layer:** Given the outline above, the main remaining challenge is how to reduce verification of a single point in the low degree extension of a layer in the circuit, to verification of a single point in the low degree extension of the previous layer. One of our main technical contributions is a protocol that achieves this task.

We observe that every point in the low degree extension (LDE) of the advanced layer (layer $i$) is a linear combination, or a weighted sum, of the values of that layer’s gates. The circuit has fan-in 2, so the value of each gate is a function of the values of two gates in the layer below (layer $i + 1$). We can express the value of each gate $g$ in layer $i$ as a sum, over
all possible gate-pairs $(k, \ell)$ in layer $i + 1$, of a low degree function of: (i) the values of gates $k$ and $\ell$, and (ii) a predicate that indicates whether gates $k$ and $\ell$ are indeed the “children” of gate $g$. Arithmetizing this entire sum of sums, we run a sum-check protocol [LFKN92] to verify the value of one point in the low-degree extension of layer $i$. Modulo many details, we conclude that, given access to (a low-degree extension of) the predicate that says whether a pair of gates $(k, \ell)$ are the children of the gate $g$, at the end of this sum-check protocol the verifier only needs to verify the values of a pair of points in the LDE of layer $i + 1$. This is still not enough, as we need to reduce the verification of a single point in the LDE of layer $i$ to the verification of a single point in layer $i + 1$ and not of a pair of points. We finally use an interactive protocol to reduce verifying two points in the LDE of layer $i + 1$ to verifying just one.

The verifier’s running time in each of these phases is poly-logarithmic in the circuit size. In the final phase, computing one point in the low-degree extension of the input requires only nearly-linear time, independent of the rest of the circuit. Another important point is that the verifier does not need to remember anything about earlier phases of the verification, at any point in time it only needs to remember what is being verified about a certain point in the computation. This results in very space-efficient verifiers. The savings in the prover’s running time comes (intuitively) from the fact that the prover does not need to arithmetize the entire computation, but rather proves statements about one (parallel) computation step at a time.

**Utilizing Uniformity:** It remains then to show how the verifier can compute (a low-degree extension of) a predicate that decides whether circuit gates are connected, without looking at the entire circuit. To do this, we proceed in two steps. First, we examine low space computations, e.g. uniform log-space Turing Machines (deterministic or non-deterministic). A log-space machine can be transformed into a family of boolean circuits with poly-logarithmic depth and polynomial size. We show that in this family of circuits, it is possible to compute the predicate that decides whether circuit gates are connected in poly-logarithmic time and constant ($\mathcal{AC}^0$) depth. This computation can itself be arithmetized, which allows the verifier
to compute a low-degree extension of the predicate in poly-logarithmic time. Thus we obtain an interactive proof with an efficient prover and super-efficient verifier for any \( L \) or \( \mathcal{NL} \) computation.

Still, the result above took advantage of the (strong) uniformity of very specific circuits that are constructed from log-space Turing Machines. We want to give interactive proofs for general log-space uniform circuits, and not only for the specific ones we can construct for log-space languages. How then can a verifier compute even the predicate that decides whether circuit gates in a log-space uniform circuit are connected (let alone its low degree extension)? In general, computing this predicate might require nearly as much time as evaluating the entire circuit. We overcome this obstacle by observing that the verifier does not have to compute this predicate on its own: it can ask the prover to compute the predicate for it. Of course, the prover may cheat, but the verifier can use the above interactive proof for log-space computations to force the prover to prove that it computed the (low degree extensions of) the predicate correctly. This final protocol gives an interactive proof for general log-space uniform circuits with low depth.

See Section 3.1.7 for an expanded and more detailed technical overview of the protocol.

1.2 Verifying Interactive Proofs in Constant Depth

We proceed to study a different notion of super-efficient verification, and consider the task of proof verification by constant-depth verifiers. Whereas in Chapter 1.1 we focused mostly on interactive proofs for efficient computations, and required all parties to run in polynomial time, here we study interactive proofs for general (not necessarily polynomial-time) computations. We are interested in exploring the expressive power of interactive proofs with very restricted constant-depth verifiers, i.e. we ask which languages/computations have such interactive proofs. As a special case we also consider the power of protocols where all parties run in polynomial time.
Main Results: We show that interactive proofs remain surprisingly powerful even with very restricted verifiers. In particular, we show general transformations from any \((p\text{-prover})\) interactive proof with \(k\) rounds of interaction, to a \((p\text{-prover})\) interactive proof with \(k + O(1)\) rounds and a constant depth \((\mathcal{NC}^0)\) verifier. By this we mean that the verifier’s strategy at each interaction round can be computed in constant depth \((\mathcal{NC}^0)\) when given access to the input, the randomness, and the messages exchanged in the previous rounds. This is captured by the following Lemma:

Lemma 1.2.1. Every language that has a \(k\)-round simple interactive proof system with \(p \geq 1\) provers, completeness \(c\) and soundness \(s\), has an interactive proof system with \(p\) provers, \(k+2\) rounds, completeness \(c\) and soundness \(s + \delta\), where \(\delta > 0\) is an arbitrarily small constant, and the verifier’s entire strategy (both interaction and decision) is in \(\mathcal{NC}^0\).

See Section 4.2 for proof intuition and a full proof. Here by a “simple” proof system we mean one in which the verifier’s messages (but not necessarily its decision bit) can be computed in \(\mathcal{NC}^0\) (see Definition 4.1.3, in particular any public coin proof system is “simple” in this sense). Building on Lemma 1.2.1, we obtain the following characterizations (see Section 4.2 for details):

Corollary 1.2.2. A language is in \(\mathcal{AM}\) if and only if it can be verified in constant parallel time with one prover, two rounds of interaction, and arbitrarily small constant soundness. In particular, every language in \(\mathcal{NP}\) has such a proof system.

Theorem 1.2.3. A language is in \(\mathcal{NEXP}\) if and only if it can be verified in constant parallel time with two provers, five rounds, perfect completeness and soundness \(\delta\), where \(\delta > 0\) is an arbitrarily small constant.

---

6 We observe in Section 4.3 that by adding \(O(\log n)\) communication rounds it is not hard to transform any protocol into one with an \(\mathcal{NC}^0\) verifier. However, we achieve this while only adding a constant number of communication rounds. This is what enables us to obtain constant parallel time verification for languages in \(\mathcal{AM}\) and \(\mathcal{NEXP}\).

7 Here we follow the standard convention that measures the complexity of the protocol only in terms of the resources used by the verifier, i.e. it is assumed that the prover’s messages are generated instantly. Thus, our statement about constant parallel-time verification follows from the fact that the protocols we construct have a constant number of communication rounds and that each round the verifier’s strategy can be implemented in constant parallel time.
Negative results: We complement our positive results with two negative results. First, we show that constant-round proof systems with an $\mathcal{NC}^0$ verifier cannot have sub-constant soundness (unless the language itself is in $\mathcal{NC}^0$). Second, we show that there is no public-coin proof system with an $\mathcal{NC}^0$ verifier (again, unless the language itself is in $\mathcal{NC}^0$). This result sheds light on private vs. public coins proof systems and in particular on our protocols (which, naturally, use private coins). In particular, it shows that both interaction and private randomness are provably essential for non-trivial $\mathcal{NC}^0$ verification. See Section 4.3 for details.

Related Work: See Section 1.1 for an overview of past work on interactive proofs. We emphasize that previous proof systems for complete languages in $\mathcal{NP}$, $\mathcal{IP}$ and $\mathcal{NEXP}$ require, at the very least, the verification of an $\mathcal{NP}$ statement at the end of the protocol (even to achieve constant soundness). By the Cook-Levin reduction, this verification is very efficient (i.e. in $\mathcal{AC}^0$); indeed, a key point in the Cook-Levin theorem is that computation can be verified by making many local consistency checks and ensuring that they all hold. However, while the local checks are of constant size, the verifier still needs to verify that all of them hold by computing an AND of large fan-in, and therefore is not in $\mathcal{NC}^0$. We show that, perhaps surprisingly, a verifier can use interaction with the prover together with its (private) random coins to avoid performing a global test on its entire input and proof. This is done by replacing the global test with a test that the prover is not cheating. The size (fan-in) of this new test is only a function of the soundness, independent of the size of the input.\footnote{Although we emphasize the locality of the verifier, one should not confuse our verifiers with PCP verifiers. While the latter do look at a constant number of bits in the proof, they still need to check consistency with the whole input (e.g. when computing the PCP reduction) and therefore are not in $\mathcal{NC}^0$ (as a function of the input, the proof and the randomness). In fact, our lower bounds (see Section 4.3) show that non-trivial languages cannot have $\mathcal{NC}^0$ PCP verifiers (roughly speaking, this is because PCPs are not interactive).}

1.2.1 Technical Overview and Comparison to Related Work

We conclude with an overview of some of the techniques used to prove the transformation of Lemma 1.2.1. We improve the receiver’s (verifier’s) efficiency by delegating some of its
computation to the (possibly malicious) sender (the prover). As throughout this thesis, the idea of delegating computation from the receiver to the untrusted sender seems dubious at first glance: the receiver's computation is the only reliable part in the whole interaction! Indeed, this seems to leave the receiver very vulnerable to malicious behavior of the sender. To give the receiver a better guarantee, we ask more from the sender: we ask the sender to convince the receiver that he has performed the computations correctly (in the case of proof verification), or to send the results of the computations with redundancy that will allow the receiver to easily recover the correct results even from a corrupted word (in the case of codes). This may seem to bring us back to square one; namely, the receiver again needs to verify a proof or to decode a code. So where do we gain in efficiency? The key point is that we are not trying now to verify an arbitrary claim, or to recover arbitrary information, but rather we are trying to make sure that a certain computation was conducted correctly. Specifically, we show that if the receiver's computations have certain properties, which we discuss shortly, then the tasks of verifying their correctness or decoding the correct results of the computations can be done extremely efficiently – much more efficiently than the receiver's original computation. To develop our approach we look at functions that the receiver needs to compute and require them to have two properties (these properties will be useful in the subsequent sections as well):

1. (Random instance reduction) One can compute the function on any given instance by querying another function (say $g$) at a completely random location. This property allows us to “mask” the receiver’s computation as a random instance and to correct the sender’s computations.

2. (Solved instance generator) We can generate efficiently a random instance of the function $g$ together with $g$’s value on this instance. This property allows us to check the correctness of the sender’s computations.

Combining these properties allows us to ensure (w.h.p.) that the computations that the sender conducts for the receiver are indeed correct. Of course, this approach would not
give us much if we could not show that the above properties can be implemented more efficiently than the original computations. To that end we show, using techniques that were developed in the field of cryptography [Bab87, Kil88a, FKN94, IK02, AIKO6], that for functions computable in \( \mathcal{NC}^1 \) these properties can be implemented in probabilistic constant parallel time. Thus, we can take any \( \mathcal{NC}^1 \) receiver and transform it into one that runs in constant parallel time or constant depth. This reduces our task to finding a sender-receiver protocol for the required task in which the receiver is in \( \mathcal{NC}^1 \).

**Comparison with [AIKO6]:** Our approach for improving the receiver's efficiency by delegating some of its computation to the (possibly malicious) sender, is related to and inspired by the breakthrough work of Applebaum, Ishai and Kushilevitz [AIKO06] on improving the efficiency of cryptographic primitives. A discussion about the similarities and differences between these works and the results presented here follows.

The work of [AIKO06] can also be viewed as improving the efficiency of players participating in a 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 the 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 [AIKO06] 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 the settings considered in this thesis, on the other hand, the sender is untrusted or error-prone, but the receiver still wants it to perform computations for him. Our concern (in various settings, see below) is how to verify (very efficiently) the correctness of outputs computed by the untrusted or error-prone sender. Given these differences, the tools we use are somewhat different from those used in [AIKO06]. Nonetheless, some of the techniques we employ are similar.
Comparison with Settings in Subsequent Sections: In Section 1.3 and Chapter 5 we develop a methodology of delegating computation as a way to increase the efficiency of program checkers (see [BK95]) and program testers/correctors (see [BLR93]). In fact, one can view program checking as an interactive proof setting where the prover is analogous to the program and the checker is analogous to a verifier. The prover in the program checking setting is fixed in advance and restricted to computing only the language being proved, as opposed to being computationally unbounded and dynamic (according to the messages exchanged in the protocol) in the usual proof verification setting. Moreover, a program checker for a language gives such a proof system both for the language and for its complement. These differences give rise to different challenges in the design of such protocols, and in particular in the implementation of the delegation methodology. The fact that the prover is restricted to answering queries about the language being proved, in the case of program checkers, requires careful design of such 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. In Chapter 5 we give a methodology for constructing such systems with very efficient verification (checking), and a family of results for a wide variety of languages. While in this section we consider the (easier) setting of an unbounded prover, we (as opposed to the program checking setting) must deal with the challenge that the prover may change its answers according to the messages exchanged in the interaction. For example, in our interactive proof setting one cannot design proof systems that first test the prover on random inputs, and then correct it (which is a common methodology for constructing program checkers).

The results in Section 1.4 and Chapter 6 on constructing efficient error-correcting codes are also related to our results on interactive proofs and program checkers. In Section 1.4 we design decoders that delegate work to an encoder that can perform arbitrary (efficient) computations. The challenge is for the encoder to relay to the decoder computationally helpful information over a noisy channel in an error-robust way. There is a guarantee, though, that the channel is not arbitrarily malicious and the noise rate is bounded. The main challenge
is then recovering from very large fractions of errors (especially in the list-decoding setting).

See Chapter 4 and the subsequent sections for further details.

1.3 Delegation in Program Checking and Correcting

Program checking, program self-correcting and program self-testing were pioneered by Blum and Kannan [BK95] and Blum, Luby and Rubinfeld [BLR93] as a new way to gain confidence in software, by considering program correctness on an input by input basis rather than full program verification. In the setting of program checking, functions to be computed are associated with a new algorithm, called the checker. The checker, given any possibly buggy program for the function and any input, “checks” whether the program on this input computes the function correctly. 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.

We revisit program checking, taking a new approached based on delegating computation. In the program checking scenario there are again two main components: a “powerful”, but potentially faulty, program that purportedly computes some complex function, and the “weak” checker (presumably more efficient than any program for computing the function). The approach we take is delegating computation from the checker to the program being checked. Taking the approach of delegating computation in the program checking setting faces new challenges. First, while the program being checked is supposed to compute a complex function, it is not able to run arbitrary computations for us (unlike the interactive proof setting we considered in previous sections). The challenge here is identifying “helpful” (to the checker) computations that are reducible to the function that the program claims to compute. The other challenge is, of course, that the program may be faulty and compute an arbitrary different function. Thus, we need to delegate our computations in a reliable way.
**Checker Composition and Delegating Computation:** To realize the idea of designing better program checkers by delegating computation, 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 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. When the checkers for the sub-computations are more efficient than the sub-computations themselves, this results in a new checker with improved efficiency.

One of our main contributions is a composition theorem that captures this approach. We focus on reducing the computation depth of program checkers, but Take $L_{\text{external}}$ to be the language being checked, and consider the case where the checker’s sub-computations are of a language $L_{\text{internal}}$.

**Theorem 1.3.1 (Checker Composition, Informal).** Let $L_{\text{internal}}$ and $L_{\text{external}}$ be two languages that satisfy the following conditions:

1. There exists an efficient\(^9\) reduction from $L_{\text{internal}}$ to $L_{\text{external}}$.

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

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

\(^9\)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.

\(^{10}\)Throughout, efficient means poly-time (or poly-size when talking about circuits). In addition, we require the reduction to have constant-depth. See Section 5.3 for the exact statement.
Then there exists an efficient checker for $L_{\text{external}}$, of depth $O(d_{\text{external}} \cdot d_{\text{internal}})$.

The improvement comes from the fact that the depth of the resulting checker does not depend on the depth of computing $L_{\text{internal}}$, only on the depth $d_{\text{internal}}$ of checking it, which may be much smaller. See Section 5.3 for a formal statement of the theorem, the proof and extensions, as well as an analogous composition theorem for testers and correctors.\footnote{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.}

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 5.6). 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].

We proceed with an overview of some of the main results we obtain from the composition theorem.

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

See Section 5.2 for the definitions of the complexity classes involved, and Section 5.5 for a discussion and full proofs. 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.
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, see Section 2.3.

Finally, we note that the proof of the theorem is constructive: it shows how to transform any program in $\mathcal{RN}C^i$ for a language into a checker in $\mathcal{RN}C^{i-1}$ (or a tester/corrector in $\mathcal{RA}C^{i-1}$) (for all programs) for that language. Languages satisfying the requirements of Theorem 1.3.2 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 $\mathcal{RN}C^i$ but not in $\mathcal{RN}C^{i-1}$ ($i$ is well defined, as $\mathcal{RN}C^0$ is strictly contained in $\mathcal{RN}C^1$). By the theorem, this language has a checker in $\mathcal{RN}C^{i-1}$. 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 $\mathcal{RN}C$ class), will immediately give rise to an even better checker (placing it in an even lower $\mathcal{RN}C$ 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 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 use the composition theorem to construct 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 $\mathcal{NC}^0$. 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. 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 5.1 of Section 5.6.

**Theorem 1.3.3.** Matrix multiplication, inversion, determinant and rank have all probabilistic $\mathcal{AC}^0$ checkers, testers and correctors. 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 5.1.3), 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 5.3 for discussion and proofs.

### 1.3.1 Further Remarks

The results above shed new light on the theory of program checking, we conclude this overview of results by highlighting two of these contributions.

---

12 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.

13 [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.

14 For rank the result holds only over fields that are of size polynomial in the input length.
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 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 ($L_{\text{internal}}$ in Theorem 1.3.1) 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 1.3.2 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 1.3.2 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!
1.3.2 More Technical Tools

We end this section by highlighting another of the technical tools we introduce and use.

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 [Kil88a], Feige, Kilian and Naor [FKN94] and Ishai and Kushilevitz [IK02]. See the discussion in Section 1.2.1.

**Theorem 1.3.4.** 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 $\text{mod}_k$-$\mathcal{L}$).

For definitions of these complexity classes see Section 5.2, for proofs and full statements of results, see Section 5.4. 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.\textsuperscript{15}

See Section 1.2.1 for a comparison to the work of [AIK06] and to the challenges presented in the settings of Sections 1.2 and 1.4.

See Chapter 5 for further details and discussions of our results on program checkers.

1.4 Delegation in Error Correcting Codes

Error correcting codes are highly useful combinatorial objects that have found numerous applications both in practical settings as well as in many areas of theoretical computer science and mathematics. In the most common setting of error-correcting codes we have a message space that contains strings over some finite alphabet $\Gamma$ (for simplicity we assume that all strings in the message space are of the same length). The goal is to design a function, which we call the \textit{encoder}, that encodes every message in the message space into a \textit{codeword} such that even if a fairly large fraction of symbols in the codeword are corrupted it is still possible to recover from it the original message. The procedure that recovers the message from a possibly corrupted codeword is called the \textit{decoder}.

We examine this well-studied setting in light of our methodology of delegating computation, focusing on the goal of efficient decoding. We again examine two components of the system: the “strong” encoder (which we are not trying to optimize), and the “weaker” decoder (whose efficiency we aim to improve). Our approach is to delegate computation from the decoder to the encoder. The main challenge is that the only “help” the encoder can offer the decoder is via a single message (the codeword) sent over a noisy and unreliable channel. Thus, we view the codeword as a computational resource that can be used by the decoder, and we aim to design noise-resilient methods for conveying computationally useful

\textsuperscript{15}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.
information.

This general approach could, potentially, be applied in a wide variety of settings for reducing different complexity measures of the decoding process. Here we apply it towards reducing the circuit depth of the decoder in the settings of locally-decodable and locally list-decodable codes. We proceed with a brief overview of the setting.

**Locally Decodable and List-Decodable Codes:** It is well known that beyond a certain fraction of errors, it is impossible to recover the original message, simply because the relatively few symbols that are not corrupted do not carry enough information to specify (uniquely) the original message. Still, one may hope to recover a list of candidate messages, one of which is the original message. Such a procedure is called *list-decoding.*

Typically, the goal of the decoder is to recover the entire message (or list of candidate messages) by reading the entire (possibly corrupted) codeword. There are settings, however, in which the codeword is too long to be read as a whole. Still, one may hope to recover any given individual symbol of the message, by reading only a small number of symbols from the corrupted codeword. This setting is called *local-decoding,* and both the unique and list decoding variants (as discussed above) can be considered. See Trevisan’s survey [Tre04] for a more detailed discussion.

Locally decodable codes, both in the unique and list decoding settings, have found many applications in theoretical computer science, most notably in private information retrieval [CKGS98, KT00], and worst-case to average-case hardness reductions [STV01] (see below). Furthermore, they have the potential of being used for practical applications, such as reliably storing a large static data file, only small portions of which need to be read at a time.

We note that all previously known decoders for locally (list-)decodable codes with similar parameters, e.g. the codes of [STV01], compute majorities or finite field operations that (provably) cannot be implemented in $\mathcal{AC}^0$.

**Transformations for Reducing Decoding Depth:** We show general transformations for reducing the circuit depth of the decoders in error-correcting codes:
Theorem 1.4.1 (Transformation for binary locally decodable codes). Let \( C : \{0, 1\}^M \rightarrow \{0, 1\}^N \) be an explicit locally-decodable binary code that can be non-adaptively decoded from some constant distance \( \delta < 1/4 \) by a probabilistic \( NC^1 \) circuit of size \( polylog(N) \). Then there is an explicit binary code \( C' : \{0, 1\}^M \rightarrow \{0, 1\}^{2N} \) that is locally-decodable from distance \( \delta/2 \) by a probabilistic \( AC^0 \) circuit of size \( polylog(N) \).

Theorem 1.4.2. (Transformation for non-binary codes) Let \( \Sigma \) be a finite alphabet and let \( C : \Sigma^M \rightarrow \Sigma^N \) be an explicit code that is non-adaptively locally list-decodable by probabilistic \( NC^1 \) circuits from agreement \( \varepsilon \) and with list size \( \ell \). Then there is an explicit code \( C' : \Sigma^M \rightarrow \Gamma^N' \) that is locally list-decodable by probabilistic \( AC^0 \) circuits, from agreement \( \varepsilon \) and with list size \( 2\ell \), where \( |\Gamma| = |\Sigma|\cdot O(1/\varepsilon) \), and \( N' = \max\{N, 2^{(\log(N)/\varepsilon)^4}\} \), for arbitrarily small constant \( \delta > 0 \). In particular, for \( \varepsilon \geq 1/poly(\log N) \), we obtain \( N' = N \). Finally, if \( C \) was uniquely locally decodable (with \( \ell = 1 \)), then so is \( C' \).

The proofs and further discussion can be found in Sections 6.1.1 and 6.3.

Explicit Codes with Constant-Depth Decoding: Using the above transformations, we obtain explicit locally decodable and locally list-decodable codes. Our first construction is an explicit locally-decodable binary code:

Theorem 1.4.3 (Locally decodable binary code). There is an explicit code \( C : \{0, 1\}^M \rightarrow \{0, 1\}^{poly(M)} \) that can be locally decoded from distance \( 1/25 \) by probabilistic \( AC^0 \) circuits of size \( poly(\log M) \).

This code has roughly the same parameters (up to polynomial factors) as the canonical example of a locally-decodable binary code with polynomial rate [STV01]. See Section 6.4 for more details. We also obtain an explicit family of locally list-decodable binary codes:

Theorem 1.4.4 (Locally list-decodable binary code). For every \( 2^{-\Theta(\sqrt{\log M})} \leq \varepsilon = \varepsilon(M) < 1/2 \), there exists a \((1/2-\varepsilon, poly(1/\varepsilon))\)-locally-list-decodable code \( \{C_M : \{0, 1\}^M \rightarrow \{0, 1\}^{poly(M)}\}_{M \in \mathbb{N}} \) with a local-decoder that can be implemented by a family of constant depth circuits of size \( poly(\log M, 1/\varepsilon) \) using majority gates of fan-in \( \Theta(1/\varepsilon) \) (and AND/OR gates of unbounded fan-in).
Remark 1.4.5. In the statement of Theorem 1.4.4, we view the decoder as a constant-depth circuit with majority gates of fan-in $\Theta(1/\varepsilon)$. This serves to demonstrate the connection between local list-decoding and computing majority. For distance bounded away from 1/2 by polylogarithmically small fraction, the code obtained is simply in $\mathcal{AC}^0$.

In particular, for $\varepsilon = 1/polylog \log(M)$ the decoder can be implemented in $\mathcal{AC}^0$ with size $polylog(M)$. See Section 6.4 for further details.

Lower Bounds: We characterize the complexity of locally list-decoding binary codes, showing that the codes of Theorem 1.4.4 are essentially optimal. The proof of the following theorem is in section 6.5.

Theorem 1.4.6 (Informal). If there exists a binary code with a $(1/2 - \varepsilon, poly(1/\varepsilon))$-local-list-decoder of size $s$ and depth $d$, then there exists a circuit of size $poly(s)$ and depth $O(d)$ that computes majority on $\Theta(1/\varepsilon)$ bits.

In particular, using known lower bounds for computing majority, this means that constant-depth decoders cannot recover from large errors.

The question of lower bounding the complexity of local-list-decoders was raised by Viola [Vio06]. He conjectured that $(1/2 - \varepsilon, \ell)$-locally-list-decodable codes require computing majority over $O(1/\varepsilon)$ bits, even when the list size $\ell$ is exponential in $1/\varepsilon$. Theorem 1.4.6 proves the conjecture for the case of sub-exponential size lists. Previous and independent works gave results for other (incomparable) special cases: Viola [Vio06] gave a proof (which he attributed to Madhu Sudan) of the conjecture for the special case of the standard non-local list-decoding setting (this result does not imply, however, a strong lower bound for the local list-decoding setting). Viola [Vio06] also proved that there are no constant-depth decoders (with polynomial-size lists) for specific codes (Theorem 1.4.6, on the other hand, shows that there are no such decoders for any code).

Recently (independently of our work), Shaltiel and Viola [SV08] gave a beautiful proof of the conjecture for the local list-decoding setting, with $\ell$ exponential in $1/\varepsilon$, but for the special case that the decoder is restricted to have non-adaptive access to the received word.
(I.e., they give a lower bound for decoders that make all their queries to the received word simultaneously.) Our result is incomparable to [SV08]: we prove Viola’s conjecture only for the case that $\ell$ is sub-exponential in $1/\varepsilon$, but do so for any decoder, even an adaptive one.

We emphasize that for important ranges of parameters the best codes known to be decodable in constant depth use adaptive decoders. In particular, the constant depth decoders of this work are adaptive. See Section 6.1.2 for a fuller discussion of related work.

**Black-Box Reductions:** Finally, we consider the consequences of our positive and negative results on local list-decoding to (fully black-box) hardness amplification for low complexity classes. See Section 6.6 for a full discussion.

### 1.4.1 Technical Highlights

We seek to improve the decoder’s efficiency by delegating some of its computation to the encoder, in an error-robust way. The idea of delegating computation from the decoder to the encoder seems dubious at first glance, as any information sent from the encoder might be corrupted by the noisy channel. This seems to leave the decoder very vulnerable to malicious behavior of the sender. To give the decoder a better guarantee, we ask more from the encoder: to send the results of the computations with redundancy that will allow the decoder to easily recover the correct results even from a corrupted word. This may seem to bring us back to square one; namely, the decoder again needs to decode a code. So where do we gain in efficiency? The key point is that we are not trying now to recover arbitrary information, but rather we are trying to recover the results of a certain computation. Specifically, we show that if the decoder’s computations have certain properties (see Section 1.2.1 and below) then the task of decoding the correct results of the computations can be done extremely efficiently – much more efficiently than the decoder’s original computation. The properties we utilize are Properties 1, namely Random instance reduction, and 2, namely Solved instance generator, as described in Section 1.2.1.

Combining these properties allows us to ensure (w.h.p.) that the decoder can recover
from error introduced by the channel into the results of computations that were delegated to the encoder. As in Section 1.2.1, we use techniques from cryptography to show that for functions computable in \( \mathcal{NC}^1 \) these properties can be implemented in probabilistic constant parallel time. Thus, we can take any \( \mathcal{NC}^1 \) decoder and transform it into one that runs in constant parallel time or constant depth. We obtain transformations for binary locally decodable codes and also for non-binary locally decodable and locally list decodable codes see the theorem statement of Theorems 1.4.1 and 1.4.2 in Section 1.4 for theorem statements for these transformations, the proofs are in Section 6.3. See Section 1.2.1 for a comparison with work in cryptography such as that of [AIK06] and with the techniques in Section 1.2 and 1.3.

See Chapter 6 for further discussion of all our results on error-correcting codes.

### 1.5 Thesis Roadmap

This thesis combines research done in several works. Detailed descriptions of all the results highlighted above can be found in the corresponding chapters. Chapter 2 contains common definitions, preliminaries and technical tools used throughout the thesis. Beyond these common tools, each chapter is a self-contained description of its results and each can be read on its own.

Chapter 3 contains results on verifying interactive proofs for efficient computations and applications, see the overview in Section 1.1. The results are joint work with Goldwasser and Kalai, originally presented in [GKR08].

Chapter 4 contains results on verifying interactive proofs in constant depth, see the overview in Section 1.2. The results are joint work with Goldwasser, Gutfreund, Healy and Kaufman, and were originally presented in [GGH+07].

Chapter 5 contains results on delegating computation in the setting of program checking, testing and correcting, see the overview in Section 1.3. The results are join work with Goldwasser, Gutfreund, Healy and Kaufman, and were originally presented in [GGH+08].
Chapter 6 contains results on delegating computation in the setting of error-correcting codes, see the overview in Section 1.4. These results are a combination of two works. The first is joint work with Gutfreund, Healy and Kaufman on designing efficient decoding procedures that was originally presented in [GGH+07]. The second is later work with Gutfreund, improving the earlier codes and proving optimality of the final constructions, this work was originally presented in [GR08].
Chapter 2

Definitions and Preliminaries

For a string $x \in \Sigma^*$ (where $\Sigma$ 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 $\Gamma$ we denote by $\Delta_\Gamma$ the relative (or fractional) Hamming distance between strings over $\Gamma$. That is, let $x, y \in \Gamma^n$ then $\Delta_\Gamma(x, y) = \Pr_{i \in [n]}[x[i] \neq y[i]]$, where $x[i], y[i] \in \Gamma$. Typically, $\Gamma$ 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 $NP$, $EXP$ and $NEXP$. For a positive integer $i \geq 0$, $AC^i$ circuits are Boolean circuits (with AND, OR and NOT gates) of size $\operatorname{poly}(n)$, depth $O(\log^i n)$, and unbounded fan-in AND and OR gates (where $n$ is the length of the input). $NC^i$ circuits are boolean circuits of size $\operatorname{poly}(n)$ 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 $AC$ the class $\bigcup_{i \in \mathbb{N}} AC^i$, and by $NC$ the class $\bigcup_{i \in \mathbb{N}} NC^i$. $RNC^i$, $RAC^i$, $RNC$ and $RAC$ are the (one-sided) randomized analogs of the above classes. In particular, $AC^0$ 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. $\mathcal{AC}^i[q]$ (for a prime $q$) are similar to $\mathcal{AC}^i$ circuits, but augmented with $\text{mod } q$ 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 Interactive Proofs

We give here standard definitions for interactive proof systems.

**Definition 2.2.1.** An *interactive proof system* for a language $L$ with completeness $c$ and soundness $s$, is a two party game between a probabilistic polynomial-time verifier $V$ and a computationally unbounded prover $P$. The system has two stages: First, in the *interaction* stage, $V$ and $P$ are given a common input $x$ and they exchange messages to produce a transcript $t = (V(r), P)(x)$ (the entire messages exchange) where $r$ are the internal random coins of $V$. Then, in the *decision* stage, $V$ decides according to $x, t$ and $r$, whether to accept
or reject. The following should hold:

1. (Completeness) There exist a (honest) prover strategy $P$, such that for every $x \in L$, $\Pr_r[V(x, t, r) = \text{accept}] \geq c$, where $t = (V(r), P)(x)$.

2. (Soundness) For every $x \not\in L$ and every prover $P^*$, $\Pr_r[V(x, t, r) = \text{accept}] \leq s$, where $t = (V(r), P^*)(x)$.

If we do not specify $c$ and $s$ then their respective default values are $2/3$ and $1/3$.

A *multi-prover interactive proof system* with $p$ provers for a language $L$ is the same as an interactive proof system with the only difference that at the interaction stage $V$ exchange messages with $p$ different provers that have no communication between them. Thus the transcript $t$ contains $p$ separate sub-transcripts $t_1, \ldots, t_p$ each with a different prover.

A round of interaction is an exchange of $p$ messages ($p \geq 1$) that are sent in parallel from the verifier to the $p$ provers (one to each prover) and $p$ messages that are sent in parallel back from the provers to the verifier (one from each prover). Note that the number of rounds may depend on the length of the input to the protocol. We denote by $\mathcal{AM}$ (i.e., Arthur-Merlin games) the class of languages that have protocols with one prover and a constant number of interaction rounds.

We denote by $IP_{c,s}(k)$ the class of languages that have an interactive proof system with completeness $c$, soundness $s$ and $k$ rounds of interaction. It is well known that $IP_{2/3,1/3}(k) = IP_{1-2^{-n},2^{-n}}(k)$ (see, e.g., [Go99]). We denote by $MIP_{c,s}(p,k)$ the class of languages that have a multi-prover interactive proof system with completeness $c$, soundness $s$, $p$ provers and $k$ rounds of interaction.

We use the following characterization of $\mathcal{NEXP}$ by Feige and Lovasz:

**Theorem 2.2.2.** ([FL92] $\mathcal{NEXP} = MIP_{1,2^{-n}}(2,1)$)
2.3 Randomized Images

We start by defining the properties of functions and languages that we need for our approach in Chapters 4, 5 and 6. The first property says that we can easily generate a random instance together with the evaluation of the function on this input.

**Definition 2.3.1** (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 2.3.2** (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 \( (x', \tau) \), 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.\(^1\) 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 2.3.3** (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.

---

\(^1\)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.
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 2.3.1 and 2.3.2) 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 2.3.4.** Let \((G, \odot)\) be a group. We define the word problem of \( G \) to be the function \( L_G : G^* \rightarrow G \), where \( L_G(a_1, \ldots, a_n) = a_1 \odot a_2 \odot \cdots \odot a_n \).

**Claim 2.3.5.** 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, Kil88a]. 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 \( \tau \) 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, \( E(f(x'), a_n^{-1}) = x_1 \odot a_1 \odot a_1^{-1} \odot x_2 \odot a_2 \odot a_2^{-1} \odot x_3 \odot a_3 \cdots a_{n-1}^{-1} \odot x_n \odot a_n \odot a_n^{-1} = x_1 \odot x_2 \odot \cdots \odot x_n = L_G(x) \). Finally, note that every element in \( y \) is a function of one element in \( x \) and two elements in \( \bar{a} \), therefore \( R \) can be implemented by an \( \mathcal{NC}^0 \) circuit over the alphabet \( G \) (\( E \) is over a finite domain so it is clearly in \( \mathcal{NC}^0 \)).

**Corollary 2.3.6.** The parity function: \( \text{Parity}(x_1, \ldots, x_n) = \sum_{i=1}^{n} x_i \) 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\(^2\) (\( S_5 \) is the symmetric group over five elements). We conclude:

**Corollary 2.3.7.** 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 \( S_5 \).

Having access to a source of random elements in \( S_5 \) is a non-standard requirement. The reason that we need it is that standard (Boolean) \( \mathcal{NC}^0 \) circuits cannot sample uniformly from a set of size \( |S_5| = 120 \) (given access to a source of random bits). This motivates (as in [AIK06]) the following construction, based on ideas from [IK02], which gives complete languages in the (higher) class \( \oplus \mathcal{L} \) that have efficient randomized images that can be implemented by \( \text{Boolean} \ \mathcal{NC}^0 \) circuits. Consider the following language:

**Definition 2.3.8.** The language \( CMD \) (for Connectivity Matrix Determinant\(^3\)) is defined as follows: an instance of the language is a \( n \times n \) matrix \( A \) that has 0/1 entries on the main diagonal and above it, -1 on the second diagonal (one below the main), and 0 below this diagonal. \( A \) is represented by the list of \( n(n+1)/2 \) 0/1 entries on and above the main diagonal. Define the characteristic function of \( L \) to be \( \text{det}(A) \) where the determinant is computed over GF(2).

\(^2\)While [Bar89] only considers \( \mathcal{NC}^0 \) 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.

\(^3\)The origin of the name comes from the fact that the determinant of matrices we consider represents information about the number of paths between two nodes in a directed acyclic graph.
Claim 2.3.9. \([IK02]\) CMD is \(\oplus\)-\(\mathcal{L}\)-complete under \(\mathcal{NC}^0\) reductions.

Next, building on ideas from [Bab87, Kil88a, FKN94, IK02, AIKO6], we show that CMD has a a randomized self-image that can be implemented in \(\mathcal{AC}^0[\oplus]\). We will then modify the language to obtain a randomized image (but not a self-image) for CMD that can be implemented in \(\mathcal{NC}^0\).

Claim 2.3.10. CMD has a length-preserving randomized self-image that can be implemented by \(\mathcal{AC}^0[\oplus]\) circuits.

Proof. For every integer \(n\), Consider the family of \(n \times n\) matrices \(\mathcal{M}_1\) with 1’s on the main diagonal, 0’s below it and 0/1 above it. Similarly consider the family of \(n \times n\) matrices \(\mathcal{M}_2\) with 1’s on the main diagonal, 0/1 in the last column (except for entry \((n, n)\) which is 1), and 0’s in all other entries. Notice that every matrix in \(\mathcal{M}_1 \cup \mathcal{M}_2\) has determinant 1. It is shown in [IK02] that for every instance \(A \in CMD\) (respectively \(A \notin CMD\)), the matrix \(C_1 \times A \times C_2\) (represented by the entries on and above the main diagonal) is uniformly distributed over the elements in CMD (respectively not in CMD) of the same length, when \(C_1\) and \(C_2\) are uniformly distributed in \(\mathcal{M}_1\) and \(\mathcal{M}_2\) respectively. Furthermore, CMD halves the space, that is, for every input length, exactly half the instances are in CMD.

Given these properties, we can construct a solved instance generator and a random instance self-reduction for (the characteristic function of) CMD. The solved instance generator on \(1^n\) chooses at random \(b \in_R \{0, 1\}\) and constructs the matrix \(H_b\) that has -1 on the second diagonal, \(b\) in entry \((1, n)\) and 0’s elsewhere. It then chooses \(C_1 \in_R \mathcal{M}_1\) and \(C_2 \in_R \mathcal{M}_2\) and outputs the pair \((C_1 \times H_b \times C_2, b)\). By the above properties \(L(x) = \det(C_1 \times H_b \times C_2) = b\), and \(C_1 \times H_b \times C_2\) is uniformly distributed (over the choice of \(b, C_1, C_2\)).

Notice that,

\[
(C_1 \times H_b \times C_2)_{ij} = \sum_{\ell=1}^{n} \sum_{k=1}^{n} ((C_1)_{ik} \cdot (H_b)_{k\ell} \cdot (C_2)_{\ell j})
\]

Thus every entry in the output of the solved instance generator is a degree 3 polynomial.
over \(GF(2)\) of the bits representing \(C_1, C_2\) and \(H_b\) and it is therefore computable by \(\mathcal{AC}^0[\oplus]\) circuits.

For the random instance self-reduction, given an \(n \times n\) matrix \(A\) as above, the randomizer \(R\) chooses \(b \in_R \{0, 1\}\) and constructs the matrix \(A'\) which is equal to \(A\) everywhere except for entry \((1, n)\) defined to be \(A'_{1,n} = A_{1,n} \oplus b\). Next \(R\) chooses \(C_1 \in_R \mathcal{M}_1\) and \(C_2 \in_R \mathcal{M}_2\) and outputs the pair \((C_1 \times A' \times C_2, b)\). Finally, for \(E\) to retrieve the value of \(\det(A)\) from \(\det(C_1 \times A' \times C_2)\) and \(b\), we notice that \(\det(A) = \det(A') \oplus b\) (when the determinants are computed over \(GF(2)\)). This is because the entry \((1, n)\) appears in exactly one summand in the determinant of \(A\) and \(A'\). So \(\det(C_1 \times A' \times C_2) \oplus b = \det(A') \oplus b = \det(A)\). As in the case of the solved instance generator, the random instance self-reduction is computable in \(\mathcal{AC}^0[\oplus]\).

As in [AIK06], the reason the above construction is not in \(\mathcal{NC}^0\), is that every entry in the product matrix is a sum (over \(GF(2)\)) of many elements, each is a product (over \(GF(2)\)) of three elements. Thus it does not depend on a constant number of input and randomness bits. To overcome this, as in [AIK06], we randomize this sum using the randomizing technique for the word problem over the group \(Z^+_2\) (Corollary 2.3.5).

Let us start by defining the following variant of \(CMD\).

**Definition 2.3.11.** The language \(DCMD\) (for Decomposed Connectivity Matrix Determinant) is defined as follows: for every integer \(n > 0\), an instance (of length \(\frac{n(n+1)}{2}\)) of \(DCMD\) is described by \(\frac{n(n+1)}{2}\) \(n^2\)-tuples of bits. The \(n^2\)-tuples are indexed by \(1 \leq j \leq i \leq n\). The \((i, j)\)'th tuple is denoted \((b_{i,j}^{(1)}, \ldots, b_{i,j}^{(n^2)})\). We think of such an instance as a \(n \times n\) matrix \(A\) with -1 on the second diagonal, 0's below the second diagonal, and the \(\frac{n(n+1)}{2}\) entries above the second diagonal are given by

\[
A_{i,j} = \bigoplus_{k=1}^{n^2} b_{i,j}^{(k)}
\]

We say that such an instance is in \(DCMD\) if and only if the determinant over \(GF(2)\) of the matrix it represents is 1.
Claim 2.3.12. DCMD is a randomized image of CMD that can be implemented by $\mathcal{NC}^0$ circuits.

Proof. We describe the solved instance generator, the random instance reduction is obtained similarly. The first step of the solved instance generator is as in the proof of Claim 2.3.10. That is, let $A = C_1 \times H_b \times C_2$ where $b \in R \{0, 1\}$, $C_1 \in R \mathcal{M}_1$ and $C_2 \in R \mathcal{M}_2$. Now consider the entries of $A$ as given in Equation 2.1. For every entry, $(i, j)$, the solved instance generator chooses $n^2$ random bits, $(b_1^{(1,1)}, \ldots, b_1^{(n,n)})$. It then generates the DCMD instance whose $(k, \ell)$‘th entry (here we change to indexing in $[n] \times [n]$ instead of $[n^2]$) in the $(i,j)$’th $n^2$-tuple is

$$b_{k,\ell-1}^{i,j} \oplus (C_1)_{ik} \cdot (H_b)_{k\ell} \cdot (C_2)_{\ell j} \oplus b_{i,j}^{(k,\ell)}$$

Where we define $b_{i,j}^{(k,0)} = b_{i,j}^{(k-1,n)}$, and $b_{i,j}^{(0,0)} = b_{i,j}^{(n,n)} = 0$ (note that we do not use our random choice for $b_{i,j}^{(n,n)}$, it is included here only to simplify the indexing).

by applying the argument of Claim 2.3.5 w.r.t. the group $Z_2^+$ on each entry, we see that this instance is a DCMD representation of the matrix $A$. Furthermore, For every entry on and above the main diagonal of $A$, the $n^2$-tuple associated with it is uniformly distributed among all tuples whose parity equals to that entry. Since the value of every entry on and above the main diagonal is uniformly and independently distributed (as argued in the proof of Claim 2.3.10), we obtain a uniformly distributed instance of DCMD whose determinant is $b$. Finally, we note that every bit of the DCMD instance is a function of five input/randomness bits, and thus the solved instance generator can be implemented by an $\mathcal{NC}^0$ circuit.

Now that we have randomized images for several complete problems, we observe that the property is preserved by reductions.

Claim 2.3.13. If a language $L$ is hard for some complexity class $\mathcal{C}_1$ under some class $\mathcal{C}_2$ of Karp-reductions and if $L$ has a randomized image that can be implemented in $\mathcal{C}_2$, then every language $L' \in \mathcal{C}_1$ has a randomized image that can be implemented in $\mathcal{C}_2$. 

59
Proof. The randomized image of \( L' \) will be the one of \( L \). To obtain a random instance reduction, given an instance \( x \) for the language \( L' \), reduce it to an instance \( y \) for \( L \). Then apply on \( y \) the random instance reduction from \( L \) to its randomized image. 

Another useful property of functions is the following strong form of downward self-reducibility.

**Definition 2.3.14.** We say that a language \( L \) is strongly downward self-reducible if, for every constant \( \delta > 0 \), \( L \) can be decided by a family of polynomial-size constant-depth oracle circuits, such that the circuit for length \( n \) makes queries to an oracle that solves \( L \) at input length \( n^\delta \).

Using associativity of group operations, the following is straightforward.

**Claim 2.3.15.** Let \( (G, \odot) \) be a finite group. Then \( L_G \) (the word problem over \( G \)) is strongly downwards self-reducible.

We conclude this section with the following lemma which says that there is an \( NC^1 \)-complete language that has all the properties that we need.

**Lemma 2.3.16.** There is an \( NC^1 \)-complete language under \( NC^0 \) reductions, that is strongly downwards self-reducible, and has a randomized image that can be implemented by \( NC^0 \) circuits.

Proof. Barrington [Bar89] showed that the decisional version of \( L_{S_5} \) (in which one has to decide whether the resulting permutation in \( S_5 \) is the identity or not) is \( NC^1 \) complete under \( NC^0 \) reductions. By Claim 2.3.15 it is strongly downwards self-reducible. Uniform \( NC^1 \) is contained in \( \oplus L \), so by Claims 2.3.9, 2.3.12 and 2.3.13, \( DCMD \) is a randomized image of the decisional version of \( L_{S_5} \) that can be implemented by \( NC^0 \) circuits. 

Chapter 3

Verifying Interactive Proofs for Efficient Computations

3.1 Overview

Roadmap for Section 3.1. We begin with an overview and discussion of results. Our main result in this chapter is described in Section 3.1.1. We further use our techniques to obtain several other results: constructing computationally-sound one-round argument systems for any (L-uniform) NC computation, under computational assumptions (Section 3.1.2); Characterizing public-coin log-space interactive proofs (Section 3.1.3); Constructing low communication zero-knowledge proofs (Section 3.1.5); Constructing Interactive PCPs (IPCP) and Probabilistically Checkable Arguments (PCA), improving on [KR08, KR09] (Section 3.1.6). A high-level overview of our techniques is given Section 3.1.7. We proceed in Section 3.2 with preliminaries. The full protocols, proofs and technical details are presented in the subsequent sections.

3.1.1 Main Result

Our most general result is a public-coin interactive proof for any language computable by a family of boolean circuits that is L-uniform (or, more generally, can be generated using a low-
space Turing Machine).\footnote{A circuit family is \(s(n)n\)-space uniform if there exists a Turing Machine that on input \(1^n\) runs in space \(O(s(n))\) and outputs the circuit for inputs of length \(n\). A circuit family is \(L\)-uniform if it is log-space uniform.} We view this as a very relaxed notion of uniformity. In particular, it captures logarithmic space uniform computations and uniform parallel computing classes, see the discussion following Corollary 3.1.1. The communication complexity is polynomial in the depth of the computation and poly-logarithmic (rather than polynomial) in its size; the running time of the verifier is linear in the input length, polynomial in the depth and poly-logarithmic in its size; and the prover’s running time is polynomial in the computation size. See Theorem 1.1.1 in Section 1.1 for the formal statement.

An overview of the proof idea is given in Section 3.1.7, see Section 3.4 (and 3.3) for a full proof. The interactive proofs constructed in Theorem 1.1.1 provide a natural solution to the delegating computation problem mentioned above. Namely, the statement to be proved is that the delegated computation was executed correctly; the delegator is the verifier in the interactive proof; the delegatee is the prover in the interactive proof, who convinces the delegatee that he performed the computation correctly (and runs in polynomial time).

As a primary implication, we get that any computation with low parallel time (significantly smaller than the computation’s total size) has a very efficient interactive proof. In particular, for languages in \(L\)-uniform \(NC\), we have:

**Corollary 3.1.1.** Let \(L\) be a language in \(L\)-uniform \(NC\), i.e. computable by a family of \(O(\log(n))\)-space uniform circuits of size \(\text{poly}(n)\) and depth \(\text{polylog}(n)\). \(L\) has an interactive proof where:

1. The prover runs in time \(\text{poly}(n)\), the verifier runs in time \(n \cdot \text{polylog}(n)\) and space \(O(\log(n))\).

2. The protocol has perfect completeness and soundness \(1/2\).

3. The protocol is public-coin, with communication complexity \(\text{polylog}(n)\).

A natural question is how can this be done when the verifier cannot even take the circuit in question as an additional input (it has no time to read it!). This is where the condition on the
log-space uniformity of the circuit family comes in. For such circuit families, the circuit has a “short” implicit representation which the verifier can use without ever constructing the entire circuit. We view log-space-uniformity as a relaxed notion of uniformity for polynomial-sized circuits (though admittedly less relaxed than poly-time-uniformity). In particular, Corollary 3.1.1 applies to any language in $\mathcal{NL}$, and even to any language computable by a $PRAM$ in poly-logarithmic parallel time.

Alternatively, by modifying the model (to include an on-line and an off-line stage of computation) we also obtain results for the non-uniform setting. See Sections 3.1.4 and 3.1.7 for more details. See Section 1.1 for an overview of prior work on interactive proofs.

**Comparison to Prior Work in Other Models.** The goal of the work of Babai, Fortnow, Lund and Szegedy [BFLS91] on Holographic Proofs for $NP$ (i.e., PCP-proofs where the input is assumed to be given to the verifier in an error-correcting-code format), was to extend Blum and Kannan’s program checking model [BK95] to checking the results of executions (the combination of software and hardware) of long computations. They show how to achieve checking time that is poly-logarithmic in the length of the computation (on top of the time taken to convert the input into an error correcting code format). Their proof-string has length that is polynomial in the computation time (the verifier has random access to this proof string). However the soundness of proofs in this PCP like model (as well as its more efficient descendants [PS94, BGH+06, BGH+05, Din07]) requires that the verifier/delegator either “posses” the entire PCP proof string (though only a few of its bits are read), or somehow have a guarantee that the prover/delegatee cannot change the PCP proof string after the verifier has started requesting bits of it. Such guarantees seem difficult to achieve over a network as required in the delegation setting.

Kilian [Kil92, Kil95] gives an argument system for any $NP$ computation, with communication complexity that is polylogarithmic, and verifier runtime which is linear in the input length (up to polylogarithmic factors). This is achieved by a constant round protocol, in which the prover first constructs a PCP for the correctness of the computation, and then Merkle-hashes it down to a short string and sends it to the verifier. To do this, one must
assume the existence of strong collision-intractable hash functions with poly-logarithmic output size.\(^2\) We emphasize, that an argument system achieves only computational soundness (soundness with respect to a computationally bounded dishonest prover). In the interactive proof setting soundness is guaranteed against any cheating prover.

Finally, Micali raises similar goals to ours in his work on Computationally Sound (CS) proofs [Mic94]. His results are however obtained in the random oracle model. This allows him to achieve CS-proofs for the correctness of general time computations with a nearly linear time verifier, a prover whose runtime is polynomial in the time complexity of the computation, and a poly-log length non-interactive (“written down” rather than interactive) proof. Alternatively viewed, Micali’s work gets non-interactive CS-proofs under the same assumption as [Kil92], and assuming the existence of Fiat-Shamir-hash-functions [FS86] to remove interaction. The plausibility of realizing Fiat-Shamir-hash-functions by any explicit function ensemble has been shown to be highly questionable [Bar01, CGH04, DNRS03, GK03].

Finally, we note that all of the above [BFLS91, Kil92, Kil95, Mic94] use the full PCP machinery, and in fact this use of PCPs is to some extent inherent [RV09]. Our results, on the other hand, do not use the full PCP machinery. In particular, we do not use low-degree tests for our main results.

### 3.1.2 One-Round Arguments

So far, we mostly considered interactive settings with multiple rounds. We find it very interesting to pursue the question of delegating computation in the non-interactive or single-round setting as well. One may envision a delegator farming out computations to a computing facility (say by renting computer time at a super-computer facility during the night hours), where the result is later returned via e-mail with a fully written-down “certificate” of correctness.

Thus, we further ask: for which polynomial time computations can a polynomial time

\(^2\)With standard intractability assumptions, one could get arguments of small but polynomial communication complexity (using universal arguments [BG02]).
prover, after receiving a challenge from the verifier, write down a certificate of correctness that is super-efficiently verifiable, and in particular is significantly shorter than the time of computation (otherwise the verifier cannot even receive the certificate!). I.e. we envision a one-round protocol, where the verifier sends the prover a (potentially private-coin) challenge, and gets back a certificate of correctness for some claim. Micali’s CS-proofs result [Mic94] is the only solution known to this problem, and it is in the random oracle model. We note that in the random oracle model CS proofs are fully non-interactive: there is no need for the verifier to even send a challenge to the prover.

We address this problem and construct one-round arguments for any \(\mathcal{L}\)-uniform \(\mathcal{NC}\) computation, assuming the existence of a computational private information retrieval (PIR) scheme with \(\text{poly}(\kappa)\)-communication, where \(\kappa\) is the security parameter. We note that such a PIR scheme exists for any \(\kappa \geq \log |DB|\) (where \(|DB|\) is the database size) under the \(N\)-th Residuosity Assumption [Lip05, IP07], and under the \(\Phi\)-Hiding Assumption [CMS99]. For a polynomially small security parameter, such PIR schemes exist under a variety of computational assumptions (see e.g. [KO97]). Moreover, this argument has the property that the verifier’s challenge is independent of the language or the input whose membership is being proved. This means, for example, that the challenge can be prepared in advance and posted on the verifier’s webpage (note, however, that we make no claims for soundness if the verifier uses the same challenge more than once).

For security parameter \(\kappa\), the size of the certificates is \(\text{poly}(\kappa, \log n)\), the (honest) prover runs in polynomial time, and the verifier runs in time \(n \cdot \text{poly}(\kappa, \log n)\) to verify a certificate (as in [Mic94]). Soundness holds only against computationally bounded cheating provers (see Section 3.2.4 for a definition and more details about the computational assumption).

Theorem 3.1.2. Let \(L\) be a language computable by a family of \(O(\log(S(n)))\)-space uniform boolean circuits of size \(S(n)\) and depth \(d(n)\). Let \(\kappa \geq \log(S(n))\) be a security parameter. Assume the existence of a secure PIR scheme, with communication \(\text{poly}(\kappa)\), receiver work \(\text{poly}(\kappa)\), and sender work \(\text{poly}(n, \kappa)\) (where \(n\) is the database size). The language \(L\) has a 1-round (private coin) argument system with the following properties:
1. The prover runs in time \( \text{poly}(S(n)) \), the verifier runs in time \( n \cdot \text{poly}(\kappa, d(n), \log(S(n))) \).\(^3\)

2. The protocol has perfect completeness and computational soundness \( \frac{1}{2} \) (can be made arbitrarily small): for any input \( x \notin L \) and for any cheating prover running in time \( \leq 2^\kappa^3 \), the probability that the verifier accepts is \( \leq \frac{1}{2} \).

3. The sizes of the certificate (the prover’s message) and the verifier’s challenge are \( \text{poly}(\kappa, d(n)) \). The verifier’s short challenge depends only on the parameters \( n \) and \( \kappa \), and is independent of the language \( L \) and the input \( x \).

The idea of the proof is as follows. We apply to the protocol of Theorem 1.1.1 a new transformation due to Kalai and Raz in their paper on probabilistically checkable arguments [KR09]. They use a computational PIR scheme to transform any public-coin interactive proof into a one-round argument system (where the verifier’s first and only message can be computed independently and ahead of the input).

More specifically, [KR09] show how to convert any public-coin interactive proof system \((P, V)\) (for a language \( L \)), with communication complexity \( \ell \), completeness \( c \), and soundness \( s \), into a one-round (two-message) argument system \((P', V')\) (for \( L \)), with communication complexity \( \text{poly}(\ell, \kappa) \), completeness \( c \), and soundness \( s + 2^{-\kappa^2} \) against malicious provers of size \( \leq 2^\kappa^3 \), where \( \kappa \geq \log(S(n)) \) is the security parameter. The verifier \( V' \) runs in time \( t_V \cdot \text{poly}(\kappa) \), where \( t_V \) is \( V \)'s running time. The prover \( P' \) runs in time \( t_P \cdot \text{poly}(\kappa, 2^\lambda) \), where \( t_P \) is \( P \)'s running time, and \( \lambda \) satisfies that each message sent by the prover \( P \) depends only on the \( \lambda \) previous bits sent by \( V \).

Note that if \( \lambda \) is super-logarithmic, then the resulting prover \( P' \) is inefficient. Fortunately, the protocol of Theorem 1.1.1 has the property that \( \lambda = O(\log(S(n))) \), and thus the resulting \( P' \) runs in time \( \text{poly}(S(n)) \).

As discussed above, the resulting one-round argument system \((P', V')\) has the property that the first message, sent by \( V' \), depends only on the random coin tosses of \( V' \) (and is independent of the language \( L \) and the instance \( x \)), and can be computed in time \( \text{poly}(\kappa) \).

\(^3\)Moreover, if the verifier is given oracle access to the low-degree extension of its input, then its running time is only \( \text{poly}(\kappa, d(n), \log(S(n))) \).
Thus, this challenge message can be computed by the verifier in advance and say posted on its webpage. For further details, see Section 3.6.

**Applying [KR09] to Other Interactive Proofs.** We note that the [KR09] transformation can in principle be applied to any interactive proof for a PSPACE language (as IP=PSPACE). This gives polynomial-communication 1-round arguments for PSPACE computations, where the verifier runs in polynomial time, the honest prover runs in exponential time, say $2^{p(n)}$ for a polynomial $p(\cdot)$ (this is the time required to produce the certificate). Choosing a security parameter $\kappa = \text{poly}(n)$, soundness can be made to hold against dishonest provers that run in time $2^{p^3(n)}$. However, in this case the honest prover runs in super-polynomial time. Scaling known interactive proofs (such as [LFKN92, Sha92]) to efficiently computable languages and applying the [KR09] transformation still results in an inefficient prover.

### 3.1.3 Public-Coin Log-Space Verifiers

Theorem 1.1.1 as presented above, leads to the resolution of an open problem on characterizing public-coin interactive proofs for log-space verifiers.

The power of interactive proof systems with a log-space verifier has received significant attention (see Condon [Con91] and Fortnow and Sipser [For89]). It was shown that any language that has a public-coin interactive proof with a log-space verifier is in P. Fortnow and Sipser [For89] showed that such proof systems exist for the class LOGCFL. Fortnow and Lund [FL93] improved this result, showing such protocols for any language in NC. In fact, for the class \( \mathcal{P} \), [FL93] achieve $\frac{\log^2(n)}{\log \log(n)}$-space public-coin verifiers.

We resolve this question. As a corollary of Theorem 1.1.1, and using the fact that languages in \( \mathcal{P} \) have \( \mathcal{L} \)-uniform poly-size circuits, we show the following theorem (see Section 3.4.2 for details):

**Corollary 3.1.3.** Let \( L \) be a language in \( \mathcal{P} \), i.e. one that can be computed by a deterministic Turing machine in time $\text{poly}(n)$. \( L \) has an interactive proof where:
1. The prover runs in time $\text{poly}(n)$, the verifier runs in time $\text{poly}(n)$ and space $O(\log(n))$.

2. The protocol has perfect completeness and soundness $\frac{1}{2}$.

3. The protocol is public-coin, with communication complexity $\text{poly}(n)$.

### 3.1.4 Non-Uniform Circuit Families

We also obtain results for non-uniform circuits. In the non-uniform setting the verifier must read the entire circuit, which is as expensive as carrying out the computation. Thus, we separate the verification into an off-line (non-interactive) pre-processing phase, and an on-line interactive proof phase. In the off-line phase, before the input $x$ is specified, the verifier is allowed to run in $\text{poly}(S)$ time, but retains only $\text{poly}(d, \log(S))$ bits of information about $C$ (where $d$ is the depth and $S$ is the size of the circuit $C$). These bits are retained for the on-line interactive proof phase, where the verifier gets the input $x$ and interacts with the prover who tries to prove $C(x) = 1$. A similar distinction between on-line and off-line computation for interactive proofs was made in the work of Dwork and Stockmeyer [DS02] on provers that are resource-bounded during a protocol’s execution. There the separation is with respect to the prover. The prover (honest or malicious) is given a bounded amount of advice from an offline stage, and it is shown how to construct secure protocols under the assumption that the length of advice given to a dishonest prover is bounded (the honest prover makes do with very short advice). In our case, the (short) advice is given to the verifier. We emphasize that the information computed in the off-line phase can only be used once, if it is used twice (even for different inputs) then soundness is compromised.

**Theorem 3.1.4.** Let $L$ be a language computable by a (non-uniform) circuit family $C$ of size $S(n)$ and depth $d(n)$. There exists an on-line/off-line interactive proof $(P(C, x), V(x, data), V_{\text{pre}}(C))$ for $L$. This protocol has completeness 1, and soundness $\frac{1}{2}$ (can be made arbitrarily small).

The complexity of the protocol is as follows:

1. The (randomized) pre-processing computation $V_{\text{pre}}(C)$ takes time $\text{poly}(S(n))$. The output data is of length $|data| = \text{poly}(d(n), \log(S(n)))$.  

68
2. The prover $P(C, x)$ runs in time $\text{poly}(S(n))$.

3. The on-line verifier $V(x, \text{data})$ runs in time $n\text{-poly}(d(n), \log(S(n)))$ and space $O(\log(S(n)))$.

4. The communication complexity of the (on-line) interactive protocol is $\text{poly}(d(n), \log(S(n)))$.

See Section 3.4.3 for the details.

### 3.1.5 Succinct Zero Knowledge Proofs

Aside from the primary interest (to us) of delegating computation, Theorem 1.1.1 above, and more importantly the techniques used, enable us to improve previous results on communication efficient zero-knowledge interactive proofs. The literature on zero-knowledge interactive proofs and interactive arguments for $\mathcal{NP}$ is immense. In this setting we have an $\mathcal{NP}$ relation $R$ which takes as input an $n$-bit instance $x$ and a $k$-bit witness $w$. A prover (who knows both $x$ and $w$) wants to convince a verifier (who knows only $x$ and does not know $w$) in zero-knowledge that $R(x, w) = 1$.

Recently, attention has shifted to constructing zero knowledge interactive proofs with communication complexity that is polynomial or even linear in the length of the witness $w$, rather than in $R$'s worst case running time, as in traditional zero-knowledge proofs [GMW91, Blu87].

Working towards this goal, Ishai, Kushilevitz, Ostrovsky, and Sahai [IKOS07] showed that if one-way functions exist, then for any $\mathcal{NP}$ relation $R$ that can be verified by an $\mathcal{AC}^0$ circuit (i.e., a constant-depth circuit of unbounded fan-in), there is a zero-knowledge interactive proof with communication complexity $k \cdot \text{poly}(\kappa, \log(n))$, where $\kappa$ is a security parameter.\footnote{A similar result (with slightly higher communication: $\text{poly}(k, \kappa, \log(n))$) was obtained independently by Kalai and Raz [KR08].}

We improve the results of [IKOS07, KR08] significantly. We enlarge the set of languages that have zero-knowledge proofs with communication complexity quasi-linear in the witness size, from relations $R$ which can be verified by $\mathcal{AC}^0$ circuits (constant depth) to relations...
which can be verified by $\mathcal{NC}$ (polylog depth) circuits. More generally, we relate the communication complexity to the depth of the relation $R$:

**Theorem 3.1.5.** Assume one-way functions exist, and let $\kappa = \kappa(n) \geq \log(n)$ be a security parameter. Let $L$ be an $\mathcal{NP}$ language whose relation $R$ can be computed on inputs of length $n$ with witnesses of length $k = k(n)$ by Boolean circuits of size $\text{poly}(n)$ and depth $d(n)$. Then $L$ has a zero-knowledge interactive proof as follows:

1. The prover runs in time $\text{poly}(n)$ (given a witness), the verifier runs in time $\text{poly}(n)$ and space $O(\log(n))$.

2. The protocol has perfect completeness and soundness $1/2$.

3. The protocol is public-coin, with communication complexity $k \cdot \text{poly}(\kappa, d(n))$.

In particular, for relations $R$ in $\mathcal{NC}$, the protocol of Theorem 3.1.5 matches the communication complexity achieved by [IKOS07] for $\mathcal{AC}^0$; i.e., the communication complexity is quasi-linear in the witness length.

From an application point of view, enlarging the set of communication efficient protocols from relations verifiable in $\mathcal{AC}^0$ to relations verifiable in $\mathcal{NC}$, is significant. Many typical statements that one wants to prove in zero knowledge involve proving the correctness of cryptographic operations in zero knowledge, such as “The following is the result of proper decryption” or “The following is a result of a pseudorandom function”. Many such operations are generally not implementable in $\mathcal{AC}^0$ (see [LMN93]), but can often be done in $\mathcal{NC}$.

The idea behind this theorem is to use our public-coin interactive protocol from Theorem 1.1.1, and carefully apply to it the (standard) transformation from public-coin interactive proofs to zero knowledge interactive proofs of [BGG+88]. This is done using statistically binding commitments, which can be implemented using one-way functions [Nao89, HILL99]. Details are in Section 3.5.

For $\mathcal{NP}$ languages whose relations can be verified in $\mathcal{L}$-uniform $\mathcal{NC}$, the verifier in our zero knowledge proof runs in time that is quasi-linear in the input size. We note that the
works of [IKOS07, KR08] mentioned above, on zero knowledge interactive proofs for $\mathcal{AC}^0$ computable relations, do not address (nor do they achieve) improvements in the verifier’s computation time. This is captured by the following theorem:

**Theorem 3.1.6.** Assume one-way functions exist, and let $\kappa = \kappa(n) \geq \log(n)$ be a security parameter. Let $L$ be an $\mathcal{NP}$ language whose relation $R$ can be computed on inputs of length $n$ with witnesses of length $k = k(n)$ by a $\mathcal{L}$-uniform family of boolean circuits of size $\text{poly}(n)$ and depth $d(n)$. Then $L$ has a zero-knowledge interactive proof as follows:

1. The prover runs in time $\text{poly}(n)$ (given a witness), the verifier runs in time $n \cdot \text{poly}(k, \kappa, d)$ and space $O(\log(n))$.

2. The protocol has perfect completeness and soundness $1/2$.

3. The protocol is public-coin, with communication complexity $k \cdot \text{poly}(\kappa, d(n))$.

We note that in the setting of arguments and computational soundness, it is known by [Kil92] how to obtain asymptotically very efficient zero-knowledge argument systems with polylogarithmic communication complexity for all of $\mathcal{NP}$. Besides the weaker soundness guarantees, those results require assuming collision-resistant hashing (we assume only one-way functions), and use the full PCP machinery.

### 3.1.6 Results on IPCP and PCA

Building on our interactive proofs, we show constructions, with better parameters and novel features, of two new proof systems introduced by Kalai and Raz [KR08, KR09].

**Low communication and short Interactive PCP.** In [KR08] Kalai and Raz proposed the notion of an interactive PCP (IPCP): a proof system in which a polynomial time verifier has access to a proof-string (a la PCP) as well as an interactive prover. When an $\mathcal{NP}$ relation $R$ is implementable by a constant-depth circuit (i.e., $R \in \mathcal{AC}^0$) they show an IPCP for $R$ with polylog query complexity, where the proof-string is of size polynomial
in the length of the witness to $R$ (rather than the size of $R$) and an interactive phase of communication complexity $\text{polylog}(n)$. We extend this result to $\mathcal{NP}$ relations implementable by poly-size circuits of depth $d$. Namely, we demonstrate an IPCP with a proof-string of length polynomial in the length of the witness and an interactive phase of communication complexity $\text{poly}(\log n, d)$. In particular, this extends the results of [KR08] from relations in $\mathcal{AC}^0$ to relations in $\mathcal{NC}$. Moreover, the work of [KR08] focuses on the communication complexity of the proof system, but not the runtime of the verifier (the complexity of their verifier is proportional to the size of $R$). For relations in $\mathcal{L}$-uniform $\mathcal{NC}$, our techniques yield IPCPs with verifier time complexity that is quasi-linear in the input and witness sizes. See Section 3.7 for details and theorem statements.

**PCA with Efficient Provers.** Another work of Kalai and Raz [KR09] proposes a new proof system model called *probabilistically checkable argument* (PCA). A PCA is a relaxation of a probabilistically checkable proof (PCP): a verifier first specifies a challenge to the prover, and the proof (PCA) is tailored to this verifier challenge. The soundness property is required to hold only *computationally*, i.e. against bounded malicious provers. Other than these differences, the setting is the same as that of PCPs: after specifying the challenge and receiving the proof, the probabilistic polynomial time verifier only reads a few bits of the proof string in order to verify. A PCA is said to be *efficient* if the honest prover, given a witness, runs in time $\text{poly}(n)$.

Using the assumption that (computational) PIR schemes with polylog communication exist, [KR09] show a transformation from any IPCP with certain properties to a short PCA. Applying this transformation to our IPCP (the conditions of the transformation are met) yields an efficient PCA with proof-string length $\text{poly}(\text{witness size}, \log n, d)$ and query complexity $\text{poly}(\log n, d)$ for any language in $\mathcal{NP}$ whose relation can be computed by depth $d$ and poly-size circuits. We note that the efficiency of the prover is derived from a special property of our proof system. In particular, previous PCAs (obtained when one starts with the IPCPs of [KR08]) require non-polynomial time provers. See Section 3.8 for details and

\footnote{In both this work and in [KR08], the prover always runs in polynomial time.}
3.1.7 Bird’s Eye View of the Protocol

The Big Picture. In a nutshell, our goal is to reduce the verifier’s runtime to be proportional to the depth of the circuit \( C \) being computed, rather than its size, without increasing the prover’s runtime by too much.

To do this we use many of the ideas developed for the MIP and PCP setting, starting with the works of [BGKW88, BFL91, BFLS91, AS98, ALM+98, FGL+96]. We apply these ideas to the problem of proving that the computation of a (uniform) circuit \( C \) is progressing properly, without the verifier actually performing it or even looking at the entire circuit. Applying the ideas pioneered in the MIP/PCP setting to our setting, however, runs into immediate difficulties. The MIP/PCP constructions require assuming that the verifier somehow has access to a committed string (usually the string should contain a low degree extension—a high-distance encoding—of \( C \)'s computation on the input \( x \)). This assumption is built into the PCP model, and is implicitly achieved in the MIP model by the fact that the provers cannot communicate. Our challenge is that in our setting we cannot assume such a commitment! Instead, we force the prover to recursively prove the values he claims for this low-degree extension, and do this while preserving the prover’s time complexity.

Elaborating on the above, we proceed to give the idea of the proof of our main theorem. Assume without loss of generality that the circuit \( C \) is a depth \( d \) arithmetic circuit in a layered form where there are as many layers as the depth of the circuit.\(^6\)

In previous work, spanning both the single and multi prover models [LFKN92, Sha92, BFL91, KR08],\(^7\) the entire computation of the underlying machine is arithmetized and turned into an algebraic expression whose value is claimed and proved by the prover.

Departing from previous work, here we instead employ an interactive protocol that closely follows the (parallelized) computation of \( C \), layer by layer, from the output layer to the input layer, numbering the layers in increasing order from the top (output) of the circuit to the

\(^6\) Every circuit can be converted into this format by at most squaring its size and not changing the depth.

\(^7\) One exception is the work of Feige and Kilian on refereed games [FK97], which is in a different model.
bottom (input) of the circuit. The verifier has no time to compute points in the low-degree extension of the computation on \( x \) in layer \( i \): this is the low-degree extension (a high distance encoding) of the vector of values that the gates in the circuit’s \( i \)-th layer take on input \( x \), and to compute it one needs to actually evaluate \( C \), which we want to avoid! Thus, the low-degree extension of the \( i \)-th layer, will be instead supplied by the prover. Of course, the prover may cheat. Thus, each phase of the protocol lets the verifier reduce verification of a single point in the low-degree extension of an advanced step (layer) in the parallel computation, to verification of a single point in the low-degree extension of the previous step (layer). This process is repeated iteratively (for as many layers as the circuit has), until at the end the verification has been reduced to verifying a single point in the extension of the first step in the computation. In the first step of the computation (the input layer), the only information “computed” is the input \( x \), the verifier can compute the low degree extension of the input \( x \) on its own in nearly-linear time.

**Going from Layer to Layer.** Given the outline above, the main remaining challenge is how to reduce verification of a single point in the low degree extension of a layer in the circuit, to verification of a single point in the low degree extension of the previous layer.

As an aside, we note that at first glance this may seem similar to a problem faced by [FK97] in their work on refereed games: a two-prover setting where one prover is honest and the other is cheating, and the verifier does not know which is which. They also examine the computation step by step or layer by layer, but they do this for a sequential (exponential-time) computation. To grossly simplify their work, they play the provers against each other, running a binary search to find a point \( t \) of the computation where the provers agree on the results of the computation up to time \( t \), but not on the result at time \( t + 1 \). Intuitively, once this is achieved the verifier can run on his own that single computation step and find out which prover is cheating. We, on the other hand, do not have two provers to play against each other for binary search. Also, while we also layer the computation, we deal with a highly parallel computation (i.e. any circuit layout), and so the verifier cannot even run a

\[8\text{i.e., layer 0 is the output layer, and layer } d \text{ is the input layer.}\]
single step of the computation.

So the question remains, how can the verifier go from an advanced layer \( i \) in the computation to an earlier layer \( i + 1 \), without even being able to look at the entire layer? The main ingredient we use to do this is a sum-check protocol (see [LFKN92]) applied to the gates of level \( i \).

We observe that every point in the low degree extension (LDE) of the advanced layer (layer \( i \)) is a linear combination, or a weighted sum, of the values of that layer’s gates. The circuit has fan-in 2, so the value of each gate is a function of the values of two gates in the layer below (layer \( i + 1 \)). We can express the value of each gate \( g \) in layer \( i \) as a sum, over all possible gate-pairs \((k, \ell)\) in layer \( i + 1 \), of a low degree function of: (i) the values of gates \( k \) and \( \ell \), and (ii) a predicate that indicates whether gates \( k \) and \( \ell \) are indeed the “children” of gate \( g \). Arithmetizing this entire sum of sums, we run a sum-check protocol to verify the value of one point in the low-degree extension of layer \( i \). To simplify matters, we assume for now that the verifier has access to (a low-degree extension of) the predicate that says whether a pair of gates \((k, \ell)\) are the children of the gate \( g \). Then (modulo many details) at the end of this sum-check protocol the verifier only needs to verify the values of a pair of points in the LDE of layer \( i + 1 \). This is still not enough, as we need to reduce the verification of a single point in the LDE of layer \( i \) to the verification of a single point in layer \( i + 1 \) and not of a pair of points. We finally use an interactive protocol to reduce verifying two points in the LDE of layer \( i + 1 \) to verifying just one.

We assumed for simplicity of exposition above that the verifier has access to a low degree extension of the predicate describing arbitrary circuit gates. Thus, the (central) remaining question is how the verifier gains access to such LDE’s of predicates that decide whether circuit gates are connected, without looking at the entire circuit (as the circuit itself is much larger than the verifier’s running time). This is where we use the uniformity of the circuit, described below.

The verifier’s running time in each of these phases is poly-logarithmic in the circuit size. In the final phase, computing one point in the low-degree extension of the input requires
only nearly-linear time, independent of the rest of the circuit. Another important point is that the verifier does not need to remember anything about earlier phases of the verification, at any point in time it only needs to remember what is being verified about a certain point in the computation. This results in very space-efficient verifiers. The savings in the prover’s running time comes (intuitively) from the fact that the prover does not need to arithmetize the entire computation, but rather proves statements about one (parallel) computation step at a time.

**Utilizing Uniformity.** It remains then to show how the verifier can compute (a low-degree extension of) a predicate that decides whether circuit gates are connected, without looking at the entire circuit. To do this, we use the uniformity of the circuit. Namely, the fact that it has a very short implicit representation. A similar problem was faced by [BFL91]: There a computation is reduced to an (exponential) 3SAT formula, and the (polynomial-time) verifier needs to access a low-degree extension of a function computing which variables are in a specific clause of the formula. In the [BFL91] setting this can be done because the Cook-Levin reduction transforms even exponential-time uniform computations into formulas where information on specific clauses can be computed efficiently. Unfortunately, we cannot use the Cook-Levin reduction as [BFL91] and other works do, because we need to transform uniform computations into low-depth circuits without blowing up the input size.

To do this, we proceed in two steps. First, we examine low space computations, e.g. uniform log-space Turing Machines (deterministic or non-deterministic). A log-space machine can be transformed into a family of boolean circuits with poly-logarithmic depth and polynomial size. We show that in this family of circuits, it is possible to compute the predicate that decides whether circuit gates are connected in poly-logarithmic time and constant \((\text{AC}^0)\) depth. This computation can itself be arithmetized, which allows the verifier to compute a low-degree extension of the predicate in poly-logarithmic time. Thus we obtain an interactive proof with an efficient prover and super-efficient verifier for any \(\mathcal{L}\) or \(\mathcal{NC}^1\) computation.

Still, the result above took advantage of the (strong) uniformity of very specific circuits that are constructed from log-space Turing Machines. We want to give interactive proofs for
general log-space uniform circuits, and not only for the specific ones we can construct for log-space languages. How then can a verifier compute even the predicate that decides whether circuit gates in a log-space uniform circuit are connected (let alone its low degree extension)? In general, computing this predicate might require nearly as much time as evaluating the entire circuit. We overcome this obstacle by observing that the verifier does not have to compute this predicate on its own: it can ask the prover to compute the predicate for it! Of course, the prover may cheat, but the verifier can use the above interactive proof for log-space computations to force the prover to prove that it computed the (low degree extensions of) the predicate correctly. This final protocol gives an interactive proof for general log-space uniform circuits with low depth.

Finally, we note that even for non-uniform circuits, the only “heavy” computation that the verifier needs to do is computing low-degree extensions of the predicate that decides whether circuit gates are connected. The locations at which the verifier needs to access this predicate are only a function of its own randomness (and not of the input or the prover’s responses). This means that even for a completely non-uniform circuit, the verifier can compute these evaluations of the predicate’s low-degree extension off-line on its own, without knowing the input or interacting with the prover. This off-line phase requires run-time that is proportional to the circuit size. Once the input is specified, the verifier, who has the (poly-logarithmically many) evaluations of the predicate’s low degree extension that it computed off-line, can run the interactive proof on-line with the prover. The verifier will be super efficient in this on-line phase. See Section 3.3 and 3.4 for details.

**Organization of the Exposition.** The full exposition of the main result (Theorem 1.1.1) is organized in two phases. First, to highlight and clarify some of the new technical ideas, we present in Section 3.3 a bare-bones interactive proof protocol. In this protocol (which is an abstraction), we assume that the verifier “magically” gets access to (low-degree extensions of) the predicates that decide whether (and how) triplets of circuit gates are connected (predicates that specify the circuit). Given oracle access to (low degree extensions of) these predicates, we show in Theorem 3.3.1 an interactive proof with the parameters
claimed above. This still is not an interactive proof in the standard model, as the verifier gets these oracle functions “magically”. In Section 3.4 we show how to implement the above bare-bones protocol for uniform circuits. We begin in Section 3.4.1 by showing that (non-deterministic) log-space languages have low-depth polynomial-size circuits for which the verifier can compute low-degree extensions of the predicates on its own in polylogarithmic time. This immediately gives an interactive proof with a super-efficient verifier for $NL$ languages, the result is stated in that section as Theorem 3.4.4. We then proceed in Section 3.4.2 with a general result for $L$-uniform circuits, another implementation of the bare-bones protocol. Here the verifier cannot compute the predicates super-efficiently on its own. Instead, the prover computes the values of the predicates for the verifier. Of course, the prover may cheat, but these are $L$ computations. Thus, we can use the interactive proofs for $NL$ computations (of Theorem 3.4.4) as a sub-protocol, where the prover proves the correctness of his answers. This gives the result of Theorem 1.1.1.

The remainder of the paper is devoted to applications and consequences. In Section 3.5 we construct low-communication (succinct) zero-knowledge proofs. Combining our protocols with a transformation of Kalai and Raz [KR09], we obtain one-round arguments in Section 3.6. Section 3.7 shows constructions of new and improved IPCPs. Finally, in Section 3.8 we present new construction of PCAs.

3.2 Preliminaries

3.2.1 Low Degree Extension

Fix $H$ to be an extension field of $GF[2]$, and fix $F$ to be an extension field of $H$ (and in particular, an extension field of $GF[2]$), where $|F| = poly(|H|)$.9 We always assume (without loss of generality) that field operations can be performed in time that is poly-logarithmic in the field size, and space that is logarithmic in the field size. Fix an integer $m \in \mathbb{N}$. In what

---

9 Usually, when doing low degree extensions, $F$ is taken to be an extension field of $GF[2]$, and $H$ is simply a subset of $F$ (not necessarily a subfield). In this paper, we take $H$ to be a subfield. However, all we actually use is the fact that it of size $2^k$ for some $k$. 

78
follows, we define the low degree extension of a $k$-element string $(w_0, w_1, \ldots, w_{k-1}) \in \mathbb{F}^k$ with respect to $\mathbb{F}, \mathbb{H}, m$, where $k \leq |\mathbb{H}|^m$.

Fix $\alpha : \mathbb{H}^m \to \{0, 1, \ldots, |\mathbb{H}^m| - 1\}$ to be any (efficiently computable) one-to-one function. In this paper, we take $\alpha$ to be the lexicographic order of $\mathbb{H}^m$. We can view $(w_0, w_1, \ldots, w_{k-1})$ as a function $W : \mathbb{H}^m \to \mathbb{F}$, where

$$W(z) \overset{\text{def}}{=} \begin{cases} w_{\alpha(z)} & \text{if } \alpha(z) \leq k - 1 \\ 0 & \text{o.w.} \end{cases} \quad (3.1)$$

A basic fact is that there exists a unique extension of $W$ into a function $\tilde{W} : \mathbb{F}^m \to \mathbb{F}$ (which agrees with $W$ on $\mathbb{H}^m$: $\tilde{W}|_{\mathbb{H}^m} \equiv W$), such that $\tilde{W}$ is an $m$-variate polynomial of degree at most $|\mathbb{H}| - 1$ in each variable. Moreover, as is formally stated in the proposition below, the function $\tilde{W}$ can be expressed as

$$\tilde{W}(t_1, \ldots, t_m) = \sum_{i=0}^{k-1} \tilde{\beta}_i (t_1, \ldots, t_m) \cdot w_i,$$

where each $\tilde{\beta}_i : \mathbb{F}^m \to \mathbb{F}$ is an $m$-variate polynomial, that depends only on the parameters $\mathbb{H}, \mathbb{F},$ and $m$ (and is independent of $w$), of size $\text{poly}(|\mathbb{H}|, m)$ and of degree at most $|\mathbb{H}| - 1$ in each variable.

The function $\tilde{W}$ is called the low degree extension of $w = (w_0, w_1, \ldots, w_{k-1})$ with respect to $\mathbb{H}, \mathbb{F}, m$, and is denoted by $\text{LDE}_{\mathbb{H}, \mathbb{F}, m}(w)$.

**Proposition 3.2.1.** There exists a Turing machine that takes as input an extension field $\mathbb{H}$ of $\text{GF}[2]$, an extension field $\mathbb{F}$ of $\mathbb{H}$, and an integer $m$. The machine runs in time $\text{poly}(|\mathbb{H}|, m)$ and space $O(\log(|\mathbb{H}|) + \log(m))$. It outputs the unique $2m$-variate polynomial $\tilde{\beta} : \mathbb{F}^m \times \mathbb{F}^m \to \mathbb{F}$ of degree $|\mathbb{H}| - 1$ in each variable (represented as an arithmetic circuit of degree $|\mathbb{H}| - 1$ in each variable), such that for every $(w_0, w_1, \ldots, w_{k-1}) \in \mathbb{F}^k$ with $k \leq |\mathbb{H}|^m$,

---

10 Throughout this work, when we refer to a machine that takes as input a field, we mean the machine is given a short (poly-logarithmic in the field size) description of the field, that permits field operations to be computed in time that is poly-logarithmic in the field size and space that is logarithmic in the field size.
and for every $z \in \mathbb{F}^m$,
\[ \tilde{W}(z) = \sum_{p \in \mathbb{H}^m} \tilde{\beta}(z, p) \cdot W(p), \tag{3.2} \]

where $W : \mathbb{H}^m \to \mathbb{F}$ is the function corresponding to $(w_0, w_1, \ldots, w_{k-1})$ as defined in Equation (3.1), and $\tilde{W} : \mathbb{F}^m \to \mathbb{F}$ is its low degree extension (i.e., the unique extension of $W : \mathbb{H}^m \to \mathbb{F}$ of degree at most $|\mathbb{H}| - 1$ in each variable).

Moreover, $\tilde{\beta}$ can be evaluated in time $\text{poly}(|\mathbb{H}|, m)$ and space $O(\log(|\mathbb{H}|) + \log(m))$. Namely, there exists a Turing machine with the above time and space bounds, that takes as input parameters $\mathbb{H}, \mathbb{F}, m$ (as above), and a pair $(z, p) \in \mathbb{F}^m \times \mathbb{F}^m$, and outputs $\tilde{\beta}(z, p)$.

**Proof of Proposition 3.2.1.** Consider the function $\beta : \mathbb{H}^m \times \mathbb{H}^m \to \mathbb{F}$ defined by
\[ \beta(z, p) = \begin{cases} 1 & \text{if } z = p \\ 0 & \text{o.w.} \end{cases} \]

Let $\tilde{\beta} : \mathbb{F}^m \times \mathbb{F}^m \to \mathbb{F}$ be the unique extension of $\beta$, of degree at most $|\mathbb{H}| - 1$ in each variable. It is easy to see that $\tilde{\beta}$ satisfies Equation (3.2), since it satisfies Equation (3.2) for every $z \in \mathbb{H}^m$, and it is of degree at most $|\mathbb{H}| - 1$ in each variable.

It remains to prove that $\tilde{\beta}$ can be both evaluated on an input, and generated (given $(\mathbb{H}, \mathbb{F}, m)$), in time $\text{poly}(|\mathbb{H}|)$ and space $O(\log(|\mathbb{H}|))$.

1. Let $b : \mathbb{F} \times \mathbb{F} \to \mathbb{F}$ be the unique bivariate polynomial of degree $\leq |\mathbb{H}| - 1$ in each variable, such that for every $t, x \in \mathbb{H}$,
\[ b(t, x) = \begin{cases} 1 & \text{if } t = x \\ 0 & \text{o.w.} \end{cases} \]

This function is a polynomial (or arithmetic circuit) of size $\text{poly}(|\mathbb{H}|)$. It can be both evaluated on an input, and generated (given $(\mathbb{H}, \mathbb{F}, m)$), in time $\text{poly}(|\mathbb{H}|)$ and space.
2. Consider the arithmetic circuit $C : \mathbb{F}^m \times \mathbb{F}^m \to \mathbb{F}$ defined by

$$C(z, p) = \prod_{j=1}^{m} b(z_j, p_j).$$

This circuit of size $\text{poly}(|\mathbb{H}|, m)$ and degree $|\mathbb{H}| - 1$ in each of its variables. It can be evaluated on an input, and generated (given $(|\mathbb{H}|, \mathbb{F}, m)$), in time $\text{poly}(|\mathbb{H}|, m)$ and space $O(\log(|\mathbb{H}|) + \log(m))$.

It remains to note that $C$ computes the function $\tilde{\beta}$, since it agrees with $\beta$ on $\mathbb{H}^m \times \mathbb{H}^m$, and is a polynomial of degree $|\mathbb{H}| - 1$ in each variable. ■

**Claim 3.2.2.** There exists a Turing machine that takes as input an extension field $\mathbb{H}$ of $\text{GF}[2]$, an extension field $\mathbb{F}$ of $\mathbb{H}$, an integer $m$, a sequence $w = (w_0, w_1, \ldots, w_{k-1}) \in \mathbb{F}^k$ such that $k \leq |\mathbb{H}|^m$, and a coordinate $z \in \mathbb{F}^m$. It outputs the value $\tilde{W}(z)$, where $\tilde{W}$ is the unique low-degree extension of $w$ (with respect to $\mathbb{H}, \mathbb{F}, m$). The machine’s running time is $|\mathbb{H}|^m \cdot \text{poly}(|\mathbb{H}|, m)$ and its space usage is $O(m \cdot \log(|\mathbb{H}|))$.

**Proof of Claim 3.2.2.** The proof is a direct corollary of Proposition 3.2.1. Let $W : \mathbb{H}^m \to \mathbb{F}$ be the function corresponding to $w$, as defined in Equation (3.1). By Equation 3.2, for every $z \in \mathbb{F}^m$,

$$\tilde{W}(z) = \sum_{p \in \mathbb{H}^m} \tilde{\beta}(z, p) \cdot W(p).$$

By Proposition 3.2.1, we know that $\tilde{\beta}$ can be computed in time $\text{poly}(|\mathbb{H}|, m)$ and space $O(\log(|\mathbb{H}|) + \log(m))$. Thus, computing the entire sum (of products) can be done in time $|\mathbb{H}|^m \cdot \text{poly}(|\mathbb{H}|, m)$ and space $O(m \cdot \log(|\mathbb{H}|))$. ■

In Section 3.4 we refer to the low-degree extension of a $k$-element string, where each element is a *vector*. Namely, we consider the low degree extension of

$$w = (w_0, w_1, \ldots, w_{k-1}) \in (\mathbb{F}^k)^k$$
with respect to $F, H, m$, where again $k \leq |H|^m$.

Similarly to what was done above for the case $\ell = 1$, we view $(w_0, w_1, \ldots, w_{k-1})$ as a (vector valued) function $W : H^m \to \mathbb{F}^\ell$ (in particular, $W$ is again 0 on inputs whose lexicographic order is $|H|^m$ or more). As before, there is a unique extension of $W$ into a function $\hat{W} : \mathbb{F}^m \to \mathbb{F}^\ell$ which agrees with $W$ on $H^m$, and where each of the outputs is a function of degree at most $|H| - 1$ in every input variable. As before, the function $\hat{W}$ is called the low-degree extension of $w$ with respect to $H, F, m$ and denoted (as usual) by $\text{LDE}_{H,F,m}(w)$.

Finally, note that $\hat{W}$ can be expressed as the low degree extensions of $\ell$ standard functions (from $H^m$ to $\mathbb{F}$), each computing one of the $\ell$ items in $W$’s output. By Claim 3.2.2, the function $\hat{W}$ can be expressed as an arithmetic circuit over $F$, that can be generated and evaluated in time $|H|^m \cdot \text{poly}(|H|, m, \ell)$ and space $O(\log(\ell) + m \cdot \log(|H|))$.

### 3.2.2 Low Degree Test

We next explain what a low degree test is. We note that in this work, a low degree test is used only in Section 3.7, the section on interactive PCPs.

Fix a finite field $F$. Suppose that a verifier wishes to test whether a function $\pi : \mathbb{F}^m \to \mathbb{F}$ is close to an $m$-variate polynomial of degree $\leq d$ (think of $d$ as significantly smaller than $|\mathbb{F}|$). We think of a low degree test as an interactive proof for $\pi$ being close to an $m$-variate polynomial of degree $\leq d$. This proof should be short (say, of size $\leq \text{poly}(|\mathbb{F}|, m)$). The verifier has only oracle access to $\pi$, and is allowed to query $\pi$ at only a few points (say, only one point).

In this work, we need a low degree test with sub-constant error. Three such tests exist in the literature: One due to Raz and Safra [RS97], one due to Arora and Sudan [AS03], and one due to Moshkovitz and Raz [MR08]. For the sake of convenience, we use the latter. The low degree test of [MR08] is described in Figure 3-1, and is denoted by $(P_{\text{LDT}}(\pi), V_{\text{LDT}}^*).$

**Lemma 3.2.3.** For any $m \geq 3$ and $1 \leq d \leq |\mathbb{F}|$, the low degree test $(P_{\text{LDT}}(\pi), V_{\text{LDT}}^*)$ described in Figure 3-1 has the following guarantees.
Low Degree Test for $\pi : \mathbb{F}^m \rightarrow \mathbb{F}$

1. The verifier chooses uniformly and independently $z_1, z_2, z_3 \in \mathbb{F}^m$. If they are linearly dependent then he accepts. Otherwise, he sends the prover the triplet $(z_1, z_2, z_3)$.

2. The prover sends $\eta : \mathbb{F}^3 \rightarrow \mathbb{F}$, which is supposedly the function $\pi$ restricted to the subspace $U$ spanned by the vectors $z_1, z_2, z_3$. Namely,

$$\eta(\alpha_1, \alpha_2, \alpha_3) \overset{\text{def}}{=} \pi(\alpha_1 z_1 + \alpha_2 z_2 + \alpha_3 z_3).$$

3. The verifier checks that $\eta$ is of degree at most $d$. If the check fails then the verifier rejects. Otherwise, the verifier chooses a random point $z$ in the subspace $U$, by choosing uniformly $\alpha_1, \alpha_2, \alpha_3 \in \mathbb{F}$ and setting $z = \alpha_1 z_1 + \alpha_2 z_2 + \alpha_3 z_3$. He queries the oracle $\pi$ at the point $z$, and accepts if and only if

$$\eta(\alpha_1, \alpha_2, \alpha_3) = \pi(z).$$

Figure 3-1: Low degree test ($P_{LDT}(\pi), V_{LDT}^\pi$)

- **Completeness:** If $\pi : \mathbb{F}^m \rightarrow \mathbb{F}$ is an $m$-variate polynomial of total degree $\leq d$ then

$$\Pr[(P_{LDT}(\pi), V_{LDT}^\pi) = 1] = 1$$

- **Soundness (decoding):** For every $\pi : \mathbb{F}^m \rightarrow \mathbb{F}$ and every (unbounded) interactive Turing machine $\hat{P}$, if

$$\Pr[(\hat{P}(\pi), V_{LDT}^\pi) = 1] \geq \gamma$$

then there exists an $m$-variate polynomial $f : \mathbb{F}^m \rightarrow \mathbb{F}$ of total degree $\leq d$, such that

$$\Pr_{z \in \mathbb{F}^m} [\pi(z) = f(z)] \geq \gamma - \varepsilon,$$

where

$$\varepsilon \overset{\text{def}}{=} 2^{10} m^8 \sqrt{md/|\mathbb{F}|}.$$ 

- **Complexity:** $P_{LDT}(\pi)$ is an interactive Turing machine, and $V_{LDT}^\pi$ is a probabilistic interactive Turing machine with oracle access to $\pi : \mathbb{F}^m \rightarrow \mathbb{F}$. The prover $P_{LDT}$ runs in time $\leq \text{poly}(|\mathbb{F}|^m)$. The verifier $V_{LDT}^\pi$ runs in time $\leq \text{poly}(|\mathbb{F}|, m)$ and queries the oracle $\pi$ at a single point. The communication complexity is $\leq \text{poly}(|\mathbb{F}|, m)$. 

83
We refer the reader to [MR08] for a proof of Lemma 3.2.3.

3.2.3 Interactive Sum-Check Protocol

Fix a finite field $\mathbb{F}$. In a sum-check protocol, a (not necessarily efficient) prover takes as input an $m$-variate polynomial $f : \mathbb{F}^m \rightarrow \mathbb{F}$ of degree $d$ in each variable (think of $d$ as significantly smaller than $\lvert \mathbb{F} \rvert$). His goal is to convince a verifier that

$$\sum_{z \in H^m} f(z) = 0,$$

for a subset $H \subseteq \mathbb{F}$. The verifier only has oracle access to $f$, and is required to be efficient in both its running time and its number of oracle queries. In Figure 3-2, we review the standard sum-check protocol, as it appeared for example in [LFKN92, Sha92]. We denote this protocol by $(P_{sc}(f), V_{sc}^f)$.

Lemma 3.2.4. Let $f : \mathbb{F}^m \rightarrow \mathbb{F}$ be an $m$-variate polynomial of degree at most $d$ in each variable, where $d < \lvert \mathbb{F} \rvert$. The sum-check protocol $(P_{sc}(f), V_{sc}^f)$, described in Figure 3-2, satisfies the following properties.

- **Completeness:** If $\sum_{z \in H^m} f(z) = 0$ then
  $$\Pr \left[ \left( P_{sc}(f), V_{sc}^f \right) = 1 \right] = 1.$$

- **Soundness:** If $\sum_{z \in H^m} f(z) \neq 0$ then for every (unbounded) interactive Turing machine $\tilde{P}$,
  $$\Pr \left[ \left( \tilde{P}(f), V_{sc}^f \right) = 1 \right] \leq \frac{md}{\lvert \mathbb{F} \rvert}.$$

- **Complexity:** $P_{sc}(f)$ is an interactive Turing machine, and $V_{sc}^f$ is a probabilistic interactive Turing machine with oracle access to $f : \mathbb{F}^m \rightarrow \mathbb{F}$. The prover $P_{sc}(f)$ runs in time $\leq \text{poly}(\lvert \mathbb{F} \rvert^m)$.
  The verifier $V_{sc}^f$ runs in time $\leq \text{poly}(\lvert \mathbb{F} \rvert, m)$ and space $\leq \text{poly}(\lvert \mathbb{F} \rvert^m)$.

Nota 11: Here we assume the prover's input is a description of the function $f$, from which $f$ can be computed (on any input) in time $\leq \text{poly}(\lvert \mathbb{F}^m \rvert)$.
Sum-Check Protocol for $\sum_{t_1,\ldots,t_m \in H} f(t_1,\ldots,t_m) = 0$

- In the first round, $P$ computes the univariate polynomial $g_1 : \mathbb{F} \to \mathbb{F}$ defined by

$$g_1(x) \overset{\text{def}}{=} \sum_{t_2,\ldots,t_m \in H} f(x,t_2,\ldots,t_m),$$

and sends $g_1$ to $V$. Then, $V$ checks that $g_1 : \mathbb{F} \to \mathbb{F}$ is a univariate polynomial of degree at most $d$, and that

$$\sum_{x \in H} g_1(x) = 0.$$  

If not $V$ rejects. Otherwise, $V$ chooses a random element $c_1 \in_R \mathbb{F}$, and sends $c_1$ to $P$.

- In the $i$'th round, $P$ computes the univariate polynomial

$$g_i(x) \overset{\text{def}}{=} \sum_{t_{i+1},\ldots,t_m \in H} f(c_1,\ldots,c_{i-1},x,t_{i+1},\ldots,t_m),$$

and sends $g_i$ to $V$. Then, $V$ checks that $g_i$ is a univariate polynomial of degree at most $d$, and that

$$\sum_{x \in H} g_i(x) = g_{i-1}(c_{i-1}).$$

If not $V$ rejects. Otherwise, $V$ chooses a random element $c_i \in_R \mathbb{F}$, and sends $c_i$ to $P$.

- In the last round, $P$ computes the univariate polynomial

$$g_m(x) \overset{\text{def}}{=} f(c_1,\ldots,c_{m-1},x),$$

and sends $g_m$ to $V$. Finally, $V$ checks that $g_m$ is a univariate polynomial of degree at most $d$, and that

$$\sum_{x \in H} g_m(x) = g_{m-1}(c_{m-1}).$$

If not $V$ rejects. Otherwise, $V$ chooses a random element $c_m \in_R \mathbb{F}$ and checks that

$$g_m(c_m) = f(c_1,\ldots,c_m),$$

by querying the oracle at the point $z = (c_1,\ldots,c_m)$.

Figure 3-2: Sum-check protocol $(P_{SC}(f),V_{SC}^f)$ [LFKN92, Sha92]

$O(\log(|\mathbb{F}|) \cdot m)$, and queries the oracle $f$ at a single point. The communication complexity is $\leq \text{poly}(|\mathbb{F}|, m)$, and the total number of bits sent from the verifier to the prover is $O(m \cdot \log |\mathbb{F}|)$. Moreover, this protocol is public-coin; i.e., all the messages sent by
the verifier are truly random and consist of the verifier’s random coin tosses.

**Proof of Lemma 3.2.4:** The completeness condition and the complexity condition follow immediately from the protocol description. As for the soundness, let $f : \mathbb{F}^m \to \mathbb{F}$ be a polynomial of degree at most $d$ in each variable, such that $\sum_{z \in H^m} f(z) \neq 0$. Assume for the sake of contradiction that there exists a cheating prover $\tilde{P}$ for which

$$s \overset{\text{def}}{=} \Pr \left[ (\tilde{P}(f), V_{SC}^f) = 1 \right] > \frac{md}{|\mathbb{F}|}.$$

Recall that in the sum-check protocol the prover sends $m$ univariate polynomials $g_1, \ldots, g_m$, and the verifier sends $m - 1$ random field elements $c_1, \ldots, c_{m-1} \in \mathbb{F}$. For every $i \in [m]$, let $A_i$ denote the event that

$$g_i(x) = \sum_{t_{i+1}, \ldots, t_m \in H} f(c_1, \ldots, c_{i-1}, x, t_{i+1}, \ldots, t_m).$$

Let $S$ denote the event that $\left( \tilde{P}(f), V_{SC}^f \right) = 1$. Notice that $\Pr[S|A_1 \land \ldots \land A_m] = 0$. We will reach a contradiction by proving that

$$\Pr[S|A_1 \land \ldots \land A_m] \geq s - \frac{md}{|\mathbb{F}|}.$$

To this end, we prove by (reverse) induction that for every $j \in [m]$,

$$\Pr[S|A_1 \land \ldots \land A_m] \geq s - \frac{(m - j + 1)d}{|\mathbb{F}|}.$$  \hfill (3.3)

For $j = m$,

$$s = \Pr[S] \leq \Pr[S \neg(A_m)] + \Pr[S|A_m] \leq \frac{d}{|\mathbb{F}|} + \Pr[S|A_m],$$

where the latter inequality follows from the fact that every two distinct univariate polynomials of degree $\leq d$ over $\mathbb{F}$ agree in at most $\frac{d}{|\mathbb{F}|}$ points. Thus,
\[
\Pr[S|A_m] \geq s - \frac{d}{|F|}.
\]

Assume that Equation (3.3) holds for \( j \), and we will show that it holds for \( j - 1 \).

\[
s - \frac{(m - j + 1)d}{|F|} \leq \Pr[S|A_j \land \ldots \land A_m] \leq \Pr[S|\neg(A_{j-1}) \land A_j \land \ldots \land A_m] + \Pr[S|A_{j-1} \land A_j \land \ldots \land A_m] \leq \frac{d}{|F|} + \Pr[S|A_{j-1} \land \ldots \land A_m],
\]

which implies that

\[
\Pr[S|A_{j-1} \land \ldots \land A_m] \geq s - \frac{(m - (j - 1) + 1)d}{|F|},
\]

as desired. \qed

### 3.2.4 Private Information Retrieval (PIR)

A Private Information Retrieval (PIR) scheme, a concept introduced by Chor, Goldreich, Kushilevitz, and Sudan [CKGS98], allows a user to retrieve information from a database in a private manner. More formally, the database is modeled as an \( N \) bit string \( x = (x_1, \ldots, x_N) \), out of which the user retrieves the \( i \)'th bit \( x_i \), without revealing any information about the index \( i \). A trivial PIR scheme consists of sending the entire database to the user, thus satisfying the PIR privacy requirement in the information-theoretic sense. A PIR scheme with communication complexity smaller than \( N \) is said to be non-trivial.

A PIR scheme consists of three algorithms: \( Q^{PIR} \), \( D^{PIR} \), and \( R^{PIR} \). The query algorithm \( Q^{PIR} \) takes as input a security parameter \( t \), the database size \( N \), and an index \( i \in [N] \) (that the user wishes to retrieve from the database). It outputs a query \( q \), which should reveal no information about the index \( i \), together with an additional output \( s \), which is kept secret by the user and will later assist the user in retrieving the desired element from the database. The database algorithm \( D^{PIR} \) takes as input a security parameter \( t \), the database
and a query $q$, and outputs an answer $a$. This answer enables the user to retrieve $x_i$ by applying the retrieval algorithm $R^{PIR}$, which takes as input a security parameter $t$, the database size $N$, an index $i \in [N]$, a corresponding pair $(q, s)$ obtained from the query algorithm, and the database answer $a$ corresponding to the query $q$. It outputs a value which is supposed to be the $i$'th value of the database.

In this paper we are interested in poly-logarithmic PIR schemes, formally defined by Cachin et al. [CMS99], as follows.\textsuperscript{12}

**Definition 3.2.5.** Let $t$ be the security parameter and $N$ be the database size. Let $Q^{PIR}$ and $D^{PIR}$ be probabilistic circuits, and let $R^{PIR}$ be a deterministic circuit. We say that $(Q^{PIR}, D^{PIR}, R^{PIR})$ is a poly-logarithmic private information retrieval scheme if the following conditions are satisfied:

1. **(Size Restriction:)** $Q^{PIR}$ and $R^{PIR}$ are of size $\leq \text{poly}(t, \log N)$, and $D^{PIR}$ is of size $\leq \text{poly}(t, N)$. The output of $Q^{PIR}$ and $D^{PIR}$ is of size $\leq \text{poly}(t, \log N)$.

2. **(Correctness:)** $\forall N, \forall t, \forall \text{database } x = (x_1, \ldots, x_N) \in \{0, 1\}^N$, and $\forall i \in [N]$,

$$\Pr[R^{PIR}(t, N, i, (q, s), a) = x_i \mid (q, s) \leftarrow Q^{PIR}(t, N, i), a \leftarrow D^{PIR}(t, x, q)] \geq 1 - 2^{-t^3}.$$ 

3. **(User Privacy:)** $\forall N, \forall t, \forall i, j \in [N]$, and $\forall$ adversary $A$ of size at most $2^{t^3}$,

$$| \Pr[A(t, N, q) = 1 \mid (q, s) \leftarrow Q^{PIR}(t, N, i)] - \Pr[A(t, N, q) = 1 \mid (q, s) \leftarrow Q^{PIR}(t, N, j)] | \leq 2^{-t^3}.$$ 

\textsuperscript{12}Definition 3.2.5 is not worded exactly as the one in [CMS99], but was shown to be equivalent to it in [KR06].
3.3 The Bare-Bones Protocol for Delegating Computation

Our goal is constructing a protocol in which a prover, who is given a circuit $C : \{0,1\}^k \rightarrow \{0,1\}$ of size $S$ and of depth $d$, and a string $x \in \{0,1\}^k$, proves to a verifier that $C(x) = 0$. The verifier’s running time should be significantly smaller than $S$ (the time it would take him to evaluate $C(x)$ on his own). At the same time, we want the prover to be efficient, running in time that is polynomial in $S$.

Since we want the verifier to run in time that is smaller than the circuit size, we must utilize the uniformity of the circuit, as discussed in Section 3.1.7. In this section, however, we do not focus on this issue, and we do not directly obtain protocols for delegating computation. Rather, we work around the circuit uniformity issue by giving the verifier oracle access to (an extension of) the function that on input three gates outputs 1 if one gate is the addition (or the multiplication) of the other two gates. The verifier will run in quasi-linear time given this oracle. We call this protocol a bare-bones interactive proof protocol, it should be taken as an abstraction, meant to highlight and clarify some of the new technical ideas in our work. It is not an interactive proof in the standard model. Rather, we slightly change the model. We give the verifier oracle access to (an extension $F$ of) the function that specifies $C$. Namely, he gets oracle access to (an extension $F$ of) the function that on input three gates outputs 1 if one gate is the addition (or the multiplication) of the other two gates in $C$.\textsuperscript{13}

We defer fully specifying the oracle function $F$ to Subsection 3.3.1 below. For the details on how we realize the bare-bones protocol as an interactive proof (removing the oracle), see the overview in Section 3.1.7 and the full Details in Section 3.4.

Let $C : \{0,1\}^k \rightarrow \{0,1\}$ be a boolean circuit. We now proceed to present our first result, the bare-bones protocol, denoted by $(P_1, V_1)$, for efficiently proving that $C(x) = 0$. The prover and the verifier take as input a string $x \in \{0,1\}^k$, and are both given oracle

\textsuperscript{13}We also give the prover oracle access to this function. We note that the prover can compute on his own (an extension of) the function that defines $C$. However, he needs to know which extension is given to $V$, as the function specifying $C$ may have many extensions.
access to the function $\mathcal{F}$ specifying $C$ (as defined in Subsection 3.3.1), where $\mathcal{F}$ is of degree poly-logarithmic in $S$. In the protocol, the verifier $V_1$'s running time (with unit-cost oracle access to $\mathcal{F}$) will be very small (quasi-linear in the input size for the ranges of parameters we focus on), and the prover $P_1$ will remain efficient.

Of course, this result on its own does not imply an interactive proof with a fast verifier in the standard model, as computing the values of $\mathcal{F}$ may take as much time as evaluating the circuit $C$ on an input! Nonetheless, in Section 3.4 we show how to “fill in the blanks” in this bare-bones protocol. This is done by showing that for wide classes of uniform computations, the values of the oracle $\mathcal{F}$ can be computed by the verifier (on its own or with help from the prover) in poly-logarithmic (in $S$) time. This leads to implementations of the bare-bones protocol which are standard interactive proofs (without any oracles).

Giving the verifier (and prover) access to this oracle function $\mathcal{F}$, the properties of the bare-bones protocol for delegating computation are specified in the theorem below.

**Theorem 3.3.1.** Let $C : \{0,1\}^k \rightarrow \{0,1\}$ be a boolean circuit with fan-in 2 of size $S$ and depth $d$. Let $\mathcal{F}$ be an oracle computing (an extension of) the function specifying $C$, as defined in Subsection 3.3.1, of polylog($S$)-degree. Protocol $(P_1^{\mathcal{F}}(x), V_1^{\mathcal{F}}(x))$ has the following properties:

- **Completeness:** If $C(x) = 0$ then
  $$\Pr \left[ (P_1^{\mathcal{F}}(x), V_1^{\mathcal{F}}(x)) = 1 \right] = 1$$

- **Soundness:** If $C(x) \neq 0$ then for every (unbounded) interactive Turing machine $P^*$,
  $$\Pr \left[ (P^*^{\mathcal{F}}(x), V_1^{\mathcal{F}}(x)) = 1 \right] \leq \frac{1}{100}$$

- **Complexity:** The running time of the prover $P_1$ is $\text{poly}(S)$. The running time of the verifier $V_1$ is $k \cdot \text{poly}(d, \log(S))$ and it uses $O(\log(S))$ space. The communication complexity is $\text{poly}(d, \log(S))$. 

90
Moreover, the following four additional properties are satisfied:

1. The protocol is public-com.

2. The verifier makes \(O(d)\) queries to \(F\). Moreover, the points where the verifier queries \(F\) are determined solely by its (public) coins, and are uniformly random.

3. Each message sent by the prover \(P_1\) depends only on the preceding \(O(\log(S))\) bits sent by the verifier (and on the input \(x\) and oracle \(F\)).\(^{14}\)

4. If, instead of the input \(x\), \(V_1\) is given oracle access to the low degree extension of \(x\) (with respect to \(\mathbb{H}, F, m'\) as defined in Subsection 3.3.1), the protocol still satisfies all claims above. Moreover, the verifier runs in time \(\text{poly}(d, \log(S))\) and space \(O(\log(S))\). In this case, \(V_1\) queries the low-degree extension of \(x\) at a single point, which is uniformly random (over his coins).

The rest of this section is devoted to specifying the oracle \(F\) and then proving Theorem 3.3.1. We begin in Subsection 3.3.1 with preliminaries, conventions, and specifications of the protocol’s parameters, including the oracle function \(F\). The bare-bones protocol is given in Subsection 3.3.2. Finally, we prove Theorem 3.3.1 in Subsection 3.3.3.

### 3.3.1 Preliminaries

**Parameters.** Fix any circuit \(C : \{0, 1\}^k \rightarrow \{0, 1\}\). We denote the circuit size by \(S\), and the circuit depth by \(d \leq S\). Let \(\mathbb{H}\) be an extension field of \(\text{GF}[2]\) such that

\[
\max\{d, \log(S)\} \leq |\mathbb{H}| \leq \text{poly}(d, \log(S)).
\]

Let \(m\) be an integer such that

\[
S \leq |\mathbb{H}|^m \leq \text{poly}(S).
\]

\(^{14}\)This fact will be used in Section 3.8, which uses the bare-bones protocol to construct efficient “short” probabilistically checkable arguments.
Let $m' \leq m$ be an integer such that

$$k \leq |\mathbb{H}|^{m'} \leq k + \text{poly}(d, \log(S)).$$

Let $\mathbb{F}$ be an extension field of $\mathbb{H}$, where

$$|\mathbb{F}| \leq \text{poly}(|\mathbb{H}|).$$

And let $\delta \in \mathbb{N}$ be a (degree) parameter such that

$$|\mathbb{H}| - 1 \leq \delta < |\mathbb{F}|.$$

**Assumptions and notations.** Note that any Boolean (or arithmetic) circuit $C : \{0, 1\}^k \rightarrow \{0, 1\}$ can be converted into an arithmetic circuit $C : \mathbb{F}^k \rightarrow \mathbb{F}$ over the field $\mathbb{F}$, while increasing the size and the depth of the circuit by at most a constant factor. Indeed, throughout Section 3.3, we assume (without loss of generality) that $C : \mathbb{F}^k \rightarrow \mathbb{F}$ is an arithmetic circuit over the field $\mathbb{F}$. We also assume for simplicity that the circuit $C : \mathbb{F}^k \rightarrow \mathbb{F}$ is a *layered* arithmetic circuit of *fan-in* 2 (over the gates $\times$ and $+$ and over the field $\mathbb{F}$). This is without loss of generality, since any arithmetic circuit can be converted into a layered arithmetic circuit of fan-in 2, while increasing the size of the circuit by at most a polynomial factor and while increasing the depth of the circuit by at most a factor of $O(\log S)$.

When considering the $d$ layers of $C$, we think of the 0 layer as the output layer, and of the $d$ layer as the input layer. For simplicity of notations, we assume that all the layers in $C$ are of the same size (except for the input layer), and we assume that the size of each layer is $S$.\(^{15}\) We note that any circuit (of size $S$) can be transformed into one with exactly $S$ gates in each level, by adding $< S$ dummy gates (that are the constant zero) to each layer. In

\(^{15}\)Note the discrepancy between the input layer and the other layers. For our results in Sections 3.3 and 3.4 we can assume that the input layer is also of size $S$ (and this will simplify the notations a bit). However, the proof of Theorem 3.7.2 in Section 3.7 makes use of the fact that the input layer is small (of size $k$), whereas the other layers may be large (of size $S$).
particular, the transformed circuit

\[
C' : \mathbb{F}^k \to \mathbb{F}^S
\]
satisfies that for every \((x_1, \ldots, x_k) \in \mathbb{F}^k\),

\[
C'(x_1, \ldots, x_k) = (C(x_1, \ldots, x_k), 0, \ldots, 0).
\]

This increases the size of the circuit by at most a quadratic factor (and does not increase its depth).

For each \(0 < i < d - 1\), we denote the \(S\) gates in the \(i\)'th layer of \(C\) by \((g_{i,0}, g_{i,1}, \ldots, g_{i,S-1})\), and we denote the \(k\) gates in the \(d\)'th layer of \(C\) (i.e., the input layer) by \((g_{d,0}, g_{d,1}, \ldots, g_{d,k-1})\). For each \(i \in \{d - 1\}\), we associate with \(C\) two functions

\[
\text{add}_i, \text{mult}_i : \{0, 1, \ldots, S - 1\}^3 \to \{0, 1\},
\]

defined by

\[
\text{add}_i(j_1, j_2, j_3) = \begin{cases} 
1 & \text{if } g_{i-1,j_1} = g_{i,j_2} + g_{i,j_3}, \\
0 & \text{o.w.}
\end{cases}
\tag{3.4}
\]

and

\[
\text{mult}_i(j_1, j_2, j_3) = \begin{cases} 
1 & \text{if } g_{i-1,j_1} = g_{i,j_2} \times g_{i,j_3}, \\
0 & \text{o.w.}
\end{cases}
\tag{3.5}
\]

Similarly, and we associate with \(C\) two additional functions

\[
\text{add}_d, \text{mult}_d : \{0, 1, \ldots, S - 1\} \times \{0, 1, \ldots, k - 1\}^2 \to \{0, 1\},
\]

defined as in Equations (3.4) and (3.5), respectively.

We say that the functions \(\{\text{add}_i, \text{mult}_i\}_{i \in [d]}\) specify the circuit \(C\).
For each $i \in [d - 1]$, let
\[ \text{add}_i, \text{mult}_i : \mathbb{F}^{3m} \to \mathbb{F} \]
be multivariate polynomials of degree $\leq \delta$ in each variable, that extend the functions $\text{add}_i$ and $\text{mult}_i$, respectively.\(^{16}\) Namely, the functions $\text{add}_i$ and $\text{mult}_i$ satisfy that for every $z_1, z_2, z_3 \in \mathbb{H}^m$, such that $\alpha(z_1), \alpha(z_2), \alpha(z_3) \leq S - 1$ (where $\alpha : \mathbb{H}^m \to \{0, 1, \ldots, |\mathbb{H}|^m - 1\}$ is the lexicographic order),
\[ \text{add}_i(z_1, z_2, z_3) = \text{add}_i(\alpha(z_1), \alpha(z_2), \alpha(z_3)) \]
and
\[ \text{mult}_i(z_1, z_2, z_3) = \text{mult}_i(\alpha(z_1), \alpha(z_2), \alpha(z_3)). \]

If $z_1, z_2, z_3 \in \mathbb{H}^m$ but for one of them, say $z_j$, it is the case that $\alpha(z_j) > S - 1$, then both $\text{add}_i, \text{mult}_i$ return 0. The fact that such functions $\text{add}_i$ and $\text{mult}_i$ exist follows from the fact that $\delta \geq |\mathbb{H}| - 1$. In particular, $\text{add}_i$ (resp. $\text{mult}_i$) could be the low degree extension of $\text{add}_i$ (resp. $\text{mult}_i$),\(^{17}\) though we will sometimes take them to be different extensions (of slightly higher degree).

Similarly, let
\[ \text{add}_d, \text{mult}_d : \mathbb{F}^m \times \mathbb{F}^{m'} \times \mathbb{F}^{m'} \to \mathbb{F} \]
be multivariate polynomials of degree $\leq \delta$ in each variable, that extend the functions $\text{add}_d$ and $\text{mult}_d$, respectively. Namely, the functions $\text{add}_d$ and $\text{mult}_d$ satisfy that for every $z_1 \in \mathbb{H}^m$ such that $\alpha(z_1) \leq S - 1$, and for every every $z_2, z_3 \in \mathbb{H}^{m'}$ such that $\alpha(z_2), \alpha(z_3) \leq k - 1$,
\[ \text{add}_d(z_1, z_2, z_3) = \text{add}_d(\alpha(z_1), \alpha(z_2), \alpha(z_3)) \]

\(^{16}\)See Subsection 3.2.1 for a discussion on low degree extensions.
\(^{17}\)We note that in Section 3.2 we only defined the low degree extension of a string (not of a function). The low degree extension of a function $f : \mathbb{H}^m \to \mathbb{F}$ (for any $m$) can be defined analogously, as the unique function $\tilde{f} : \mathbb{F}^m \to \mathbb{F}$ of degree $\leq |\mathbb{H}| - 1$ in each variable, such that for every $z \in \mathbb{H}^m$, $\tilde{f}(z) = f(z)$.

94
and
\[
\operatorname{mult}_d(z_1, z_2, z_3) = \operatorname{mult}_d(\alpha(z_1), \alpha(z_2), \alpha(z_3)).
\]

And if \( z_1 \in \mathbb{H}^m \) and \( z_2, z_3 \in \mathbb{H}^{m'} \), but either \( \alpha(z_1) > S - 1 \), or \( \alpha(z_2) > k - 1 \), or \( \alpha(z_3) > k - 1 \), then both \( \tilde{\operatorname{add}}_d, \tilde{\operatorname{mult}}_d \) return 0.

We say that the functions \( \{\tilde{\operatorname{add}}_i, \tilde{\operatorname{mult}}_i\}_{i \in [d]} \) are extensions of the functions that specify the circuit \( C \). Note that unlike the functions \( \{\operatorname{add}_i, \operatorname{mult}_i\}_{i \in [d]} \) that specify \( C \), the extensions \( \{\tilde{\operatorname{add}}_i, \tilde{\operatorname{mult}}_i\}_{i \in [d]} \) are not uniquely determined by the circuit \( C \). For \( \delta > |\mathbb{H}| - 1 \) there are many possible extensions of the functions that specify the circuit \( C \), and \( \{\tilde{\operatorname{add}}_i, \tilde{\operatorname{mult}}_i\}_{i \in [d]} \) are some such extensions. We specify \( \{\tilde{\operatorname{add}}_i, \tilde{\operatorname{mult}}_i\} \) separately in each implementation of the bare-bones protocol.

We are now ready to specify the oracle \( \mathcal{F} \) accessed by the prover and verifier in the bare-bones protocol. This oracle consists of the collection of functions \( \{\tilde{\operatorname{add}}_i, \tilde{\operatorname{mult}}_i\}_{i \in [d]} \):
\[
\mathcal{F} = \{\tilde{\operatorname{add}}_i, \tilde{\operatorname{mult}}_i\}_{i \in [d]},
\]
where the prover and verifier can access \( \tilde{\operatorname{add}}_i \) or \( \tilde{\operatorname{mult}}_i \) by querying \( \mathcal{F} \) with the proper \( i \), a bit specifying \( \tilde{\operatorname{add}} \) or \( \tilde{\operatorname{mult}} \), and an input in \((\mathbb{F}^m)^3\) or (for \( i = d \)) in \( \mathbb{F}^m \times (\mathbb{F}^{m'})^2 \).

Finally, for each \( 0 \leq i \leq d - 1 \) we associate a vector \( v_i = (v_{i,0}, \ldots, v_{i,S-1}) \in \mathbb{F}^S \) with the \( i \)'th layer of the circuit \( C \), and we associate a vector \( v_d = (v_{d,0}, \ldots, v_{d,k-1}) \in \mathbb{F}^k \) with the \( d \)'th layer of the circuit \( C \). The vector \( v_0 \) is associated with the output layer of the circuit, and the vector \( v_d \) is associated with the input layer of the circuit. These vectors are functions of the input \( x = (x_1, \ldots, x_k) \in \mathbb{F}^k \), and are defined as follows: For each \( 0 \leq i \leq d \) we let \( v_i \) be the vector that consists of the values of all the gates in the \( i \)'th layer of the circuit on input \( x \). So, the vector \( v_0 \), that corresponds to the output layer, satisfies \( v_0 = (C(x), 0, \ldots, 0) \in \mathbb{F}^S \). Similarly, the vector \( v_d \), that corresponds to the input layer, satisfies \( v_d = (x_1, \ldots, x_k) \in \mathbb{F}^k \).
For each $0 \leq i \leq d - 1$, let

$$
\tilde{V}_i : \mathbb{F}^m \rightarrow \mathbb{F}
$$

be the low degree extension of $v_i$ with respect to $\mathbb{F}, \mathbb{F}, m$ (as defined in Subsection 3.2.1). Claim 3.2.2 implies that the function $\tilde{V}_i$ is of degree $\leq |\mathbb{F}| - 1$ in each of its $m$ variables, and can be computed in time $\leq \text{poly}(|\mathbb{F}|^m) = \text{poly}(S)$.

Let

$$
\tilde{V}_d : \mathbb{F}^m' \rightarrow \mathbb{F}
$$

be the low degree extension of $v_d$ with respect to $\mathbb{F}, \mathbb{F}, m'$. Claim 3.2.2 implies that the function $\tilde{V}_d$ is of of degree $\leq |\mathbb{F}| - 1$ in each of its $m$ variables, and can be computed in time $\leq |\mathbb{F}|^m \cdot \text{poly}(|\mathbb{F}|, m') = k \cdot \text{poly}(d, \log(S))$.

### 3.3.2 The Bare-Bones Protocol

In this subsection, we present the bare-bones protocol $(P_1, V_1)$ for efficiently proving that $C(x) = 0$. In this protocol we give both the verifier $V_1$ and the prover $P_1$ oracle access to the set of functions

$$
\mathcal{F} = \{\text{add}_t, \text{mult}_t\}_{t \in [d]},
$$

as defined in Subsection 3.3.1.\footnote{We note that the functions in $\mathcal{F}$ could have been given to the prover $P_1$ as input (say, via their truth-tables). We decided to give $P_1$ oracle access to these functions only for the sake of simplicity of the exposition (and not because of size constraints). Note also, that given oracle access to these functions, the prover $P_1$ can reconstruct the circuit $C$ in time $O(|C|)$.)\footnote{In Section 3.4 we show how these oracles can be realized in some specific cases (for example, if $C$ is an $L$-uniform circuit).}

The prover and verifier also take as input the string $x \in \{0, 1\}^k$. We begin with a protocol overview, and then proceed with the full protocol.

**Protocol Overview.** The prover wants to prove $C(x) = 0$, or equivalently, that $\tilde{V}_0(0, \ldots, 0) = 0$. This is done in $d$ phases (where $d$ is the depth of $C$). In the $i$'th phase ($1 \leq i \leq d$) the prover reduces the task of proving that $\tilde{V}_{i-1}(z_{i-1}) = r_{i-1}$ to the task of proving that $\tilde{V}_i(z_i) = r_i$, where $z_i$ is a random value determined by the protocol (and $z_0 = (0, \ldots, 0)$, $r_0 = 0$).
$r_0 = 0$). Finally, after the $d$'th phase, the verifier checks on his own that $\tilde{V}_d(z_d) = r_d$. Note that $\tilde{V}_d$ is the low degree extension of the input $x$ with respect to $\mathbb{H}, F, m$. Thus, this last verification task requires computing a single point in the low degree extension of the input $x$. This is the “heaviest” computation run by the verifier, and this final computation is independent of the circuit $C$; it can be done in quasi-linear time in the input length. Moreover, if the verifier is given oracle access to the low-degree extension of $x$, then this only requires a single oracle call.

The Bare-Bones Protocol:

Parameters.

We use the parameters defined in Subsection 3.3.1: circuit size $S$, circuit depth $d$, input size $k$, where $k, d \leq S$. We also defined there the fields $\mathbb{H}, F$, integers $m, m'$ and a degree parameter $\delta$.

The layered arithmetic circuit $C : \mathbb{F}^k \rightarrow \mathbb{F}$ is of fan-in 2 (over the gates $+$ and $\times$), of size $S$ and depth $d$.

Input.

The prover and the verifier take as input a string $x \in \mathbb{F}^k$, and are both given oracle access to a set functions $\mathcal{F} = \{\text{add}_i, \text{mult}_i\}_{i \in [d]}$ corresponding to $C$ (as defined in Subsection 3.3.1), where each function in $\mathcal{F}$ is of degree $\leq \delta$ in each variable.

The protocol $(P_l^\mathcal{F}(x), V_l^\mathcal{F}(x))$.

The prover needs to prove that $C(x) = 0$, or equivalently, that $\tilde{V}_0(0, \ldots, 0) = 0$. This is done in $d$ phases (where $d$ is the depth of $C$). In the $i$'th phase ($1 \leq i \leq d$) the prover reduces the task of proving that $\tilde{V}_{i-1}(z_{i-1}) = r_{i-1}$ to the task of proving that $\tilde{V}_i(z_i) = r_i$, where $z_i$ is a random value determined by the protocol (and $z_0 = (0, \ldots, 0)$, $r_0 = 0$). Finally, after the $d$'th phase, the verifier checks on his own that $\tilde{V}_d(z_d) = r_d$. Note that $\tilde{V}_d$ is the low degree extension of $x$ with respect to $\mathbb{H}, F, m'$. Thus, this last verification task requires computing a single point in the low degree extension of $x$. This is the “heaviest” computation run by the verifier, and this final computation is independent of the circuit $C$; it can be done in quasi-linear time in the input length. Moreover, if the verifier is given oracle access to the low-degree extension of $x$, then this only requires a single oracle call.
the input $x$. This is the “heaviest” computation run by the verifier (for the ranges of parameters we focus on in this work), and this final computation is independent of the circuit $C$. It depends only on the input $x$ (and on the randomness of the verifier and the parameters), and can be done in quasi-linear time in the input length. Moreover, note that if the verifier is given oracle access to the low-degree extension of $x$, then this only requires a single oracle call.

In what follows we describe these phases in more detail. In each phase, the communication complexity is $\text{poly}(d, \log S)$, the running time of the prover is at most $\text{poly}(S)$, and the running time of the verifier is $\text{poly}(d, \log S)$.

**The $i$'th phase ($1 \leq i \leq d - 1$).** In this phase, we reduce the task of proving that

$$\tilde{V}_{i-1}(z_{i-1}) = r_{i-1},$$

to the task of proving that

$$\tilde{V}_i(z_i) = r_i,$$

where $z_i \in \mathbb{F}^m$ is a random value determined by the verifier, and $r_i$ is a value determined by the protocol.

According to Proposition 3.2.1, for every $z \in \mathbb{F}^m$,

$$\tilde{V}_{i-1}(z) = \sum_{p \in \mathbb{H}^m} \tilde{\beta}(z, p) \cdot \tilde{V}_{i-1}(p)$$

where $\tilde{\beta} : \mathbb{F}^m \times \mathbb{F}^m \to \mathbb{F}$ is a polynomial of size $\text{poly}(|\mathbb{H}|, m)$ and of degree at most $|\mathbb{H}| - 1$ in each variable, that can be computed by a Turing machine that runs in time $\leq \text{poly}(|\mathbb{H}|, m)$. 

98
Notice that for every $p \in \mathbb{H}^m$,
\[
\tilde{V}_{i-1}(p) = \sum_{\omega_1, \omega_2 \in \mathbb{H}^m} \text{add}_i(p, \omega_1, \omega_2) \cdot \left( \tilde{V}_i(\omega_1) + \tilde{V}_i(\omega_2) \right) + \text{mult}_i(p, \omega_1, \omega_2) \cdot \tilde{V}_i(\omega_1) \cdot \tilde{V}_i(\omega_2).
\]
Thus, for every $z \in \mathbb{F}^m$,
\[
\tilde{V}_{i-1}(z) = \sum_{p, \omega_1, \omega_2 \in \mathbb{H}^m} \tilde{\beta}(z, p) \cdot \left( \text{add}_i(p, \omega_1, \omega_2) \cdot \left( \tilde{V}_i(\omega_1) + \tilde{V}_i(\omega_2) \right) + \text{mult}_i(p, \omega_1, \omega_2) \cdot \tilde{V}_i(\omega_1) \cdot \tilde{V}_i(\omega_2) \right).
\]
For every $z \in \mathbb{F}^m$, let $f_z : (\mathbb{F}^m)^3 \to \mathbb{F}$ be the function defined by
\[
f_z(p, \omega_1, \omega_2) \overset{\text{def}}{=} \tilde{\beta}(z, p) \cdot \left( \text{add}_i(p, \omega_1, \omega_2) \cdot \left( \tilde{V}_i(\omega_1) + \tilde{V}_i(\omega_2) \right) + \text{mult}_i(p, \omega_1, \omega_2) \cdot \tilde{V}_i(\omega_1) \cdot \tilde{V}_i(\omega_2) \right).
\]
Proposition 3.2.1, together with the definitions of add, mult, and $\tilde{V}_i$, implies that the function $f_z$ is a $3m$-variate polynomial of size $\leq \text{poly}(S)$ and of degree at most $\delta + |\mathbb{H}| - 1 \leq 2\delta$ in each variable. Note that, for every $z \in \mathbb{F}^m$,
\[
\tilde{V}_{i-1}(z) = \sum_{p, \omega_1, \omega_2 \in \mathbb{H}^m} f_z(p, \omega_1, \omega_2).
\]
Thus, proving that $\tilde{V}_{i-1}(z_{i-1}) = r_{i-1}$ is equivalent to proving that
\[
r_{i-1} = \sum_{p, \omega_1, \omega_2 \in \mathbb{H}^m} f_{z_{i-1}}(p, \omega_1, \omega_2).
\]
This is done by running the interactive sum-check protocol, as described in Figure 3-2.\textsuperscript{20}

However, in order to carry out the verification task, the verifier needs to compute on his own the function $f_{z_{i-1}}(p, \omega_1, \omega_2)$, on random inputs $p, \omega_1, \omega_2 \in \mathbb{F}^m$ (chosen by the verifier). Recall that the verifier has oracle access to the functions add, and mult. Moreover, according to Proposition 3.2.1, computing the function $\tilde{\beta}$ requires time $\leq \text{poly}(|\mathbb{H}|, m)$. So, the main

\textsuperscript{20}Note that in the interactive sum-check protocol the prover takes the function $f_z$ as input, whereas our prover $\mathcal{P}_1$ does not take $f_z$ as input. This is not a problem since $\mathcal{P}_1$ can compute the function $f_z$ (as a polynomial or as a truth-table) using his oracles, in time $\text{poly}(S)$.
computational burden in this verification task is computing $\tilde{V}_i(\omega_1)$ and $\tilde{V}_i(\omega_2)$, which requires time $\text{poly}(S)$ (and thus cannot be computed by our computationally bounded verifier).

In the protocol, the prover $\mathcal{P}_1$ now sends both these values, $\tilde{V}_i(\omega_1)$ and $\tilde{V}_i(\omega_2)$, to the verifier. The verifier $V_1$ (who knows $\omega_1$ and $\omega_2$) receives two values $v_1, v_2$ and now wants to verify that $\tilde{V}_i(\omega_1) = v_1$ and $\tilde{V}_i(\omega_2) = v_2$.

Thus, so far, using the sum-check protocol, we reduced task of proving that $\tilde{V}_{i-1}(z_{i-1}) = r_{i-1}$ to the task of proving that both $\tilde{V}_i(\omega_1) = v_1$ and $\tilde{V}_i(\omega_2) = v_2$. However, recall that our goal was to reduce the task of proving that $\tilde{V}_{i-1}(z_{i-1}) = r_{i-1}$ to the task of proving a single equality of the form $\tilde{V}_i(z_i) = r_i$. Therefore, what remains (in the $i$'th phase) is to reduce the task of proving two equalities of the form $\tilde{V}_i(\omega_1) = v_1$ and $\tilde{V}_i(\omega_2) = v_2$ to the task of proving a single equality of the form $\tilde{V}_i(z_i) = r_i$. This is done via the following (standard) interactive process.

1. Let $t_1, t_2 \in \mathbb{F}$ be two distinct fixed elements known to the prover $\mathcal{P}_1$ and the verifier $V_1$.

Let $\gamma : \mathbb{F} \to \mathbb{F}^m$ be the unique line (i.e., polynomial of degree at most 1), such that for every $i \in \{1, 2\}$, $\gamma(t_i) = \omega_i$. It is well known that for any $t_1, t_2, \omega_1, \omega_2$, the conditions $\gamma(t_i) = \omega_i$ determine $\gamma$ uniquely, and that $\gamma$ can be computed (by both $\mathcal{P}_1$ and $V_1$) in time $\text{poly}(|\mathbb{F}|, m)$ and space $O(\log(|\mathbb{F}|) \cdot m)$.

2. The prover $\mathcal{P}_1$ sends the function $\tilde{V}_i \circ \gamma : \mathbb{F} \to \mathbb{F}$ to the verifier $V_1$.

3. Upon receiving a function $f : \mathbb{F} \to \mathbb{F}$ from the prover (supposedly, $f = \tilde{V}_i \circ \gamma$), the verifier $V_1$ checks that $f$ is a polynomial of degree at most $m \cdot (|\mathbb{H}| - 1)$, and that $f(t_1) = v_1$ and $f(t_2) = v_2$. If these tests pass, then $V_1$ chooses a random element $t \in \mathbb{F}$ and sends it to $\mathcal{P}_1$.

4. The prover and verifier continue to Phase $i + 1$ with $z_i \triangleq \gamma(t)$ and $r_i \triangleq f(t)$.

The $d$'th phase. This phase is very similar to the previous phases. The only difference stems from the fact that the $d$'th layer of $C$ is smaller than its previous layers. Namely, it is of size $|\mathbb{H}|^{m'}$ rather than size $|\mathbb{H}|^m$. Thus, in this phase the sum-check protocol is over
$p \in \mathbb{F}^m$ and over $\omega_1, \omega_2 \in \mathbb{H}^m$. Similarly, the proceeding interactive protocol in this phase reduces the task of proving two equalities of the form $\tilde{V}_d(\omega_1) = v_1$ and $\tilde{V}_d(\omega_2) = v_2$ to the task of proving a single equality of the form $\tilde{V}_d(z_d) = r$, where now $\omega_1, \omega_2, z_d \in \mathbb{F}^m$.

The final verification. After the final verification phase, the verifier $V_1$ needs to verify on his own that $\tilde{V}_d(z_d) = r$. This amounts to computing a single point in the low-degree extension of the input $x$ (with respect to $\mathbb{F}, \mathbb{H}, m'$). The verifier runs this computation on its own (or, if given oracle access to the low degree extension of the input $x$, the verifier queries the oracle at point $z_d$ and verifies that the answer returned is $r_d$).

### 3.3.3 Proof of Theorem 3.3.1

**Completeness.** The perfect completeness follows immediately from the protocol description, as well as the perfect completeness of the sum-check protocol (see Lemma 3.2.4).

**Soundness.** For the soundness condition, fix any layered arithmetic circuit $C : \mathbb{F}^k \rightarrow \mathbb{F}$, any $x \in \mathbb{F}^k$ such that $C(x) \neq 0$, and any set of functions $\mathcal{F}$ (as defined in Subsection 3.3.1). Assume that there exists a cheating prover $P^*$ such that

$$\Pr[(P^*(x), V_1(x)) = 1] = s.$$ 

Recall that the protocol $(P^*(x), V_1(x))$ consists of $d$ phases. Each phase consists of a sum-check protocol and an additional short interactive protocol. According to our notations, the sum-check protocol requires the values of $\tilde{V}_i(w_1)$ and $\tilde{V}_i(w_2)$ for verification, and the additional interactive protocol reduces the verification of $\tilde{V}_i(w_1) = v_1$ and $\tilde{V}_i(w_2) = v_2$ to the verification of a single equality $\tilde{V}_i(z_i) = r_i$.

Let $A$ denote the event that $(P^*(x), V_1(x)) = 1$. For every $0 \leq i \leq d$, let $T_i$ denote the event that indeed $\tilde{V}_i(z_i) = r_i$. Thus, assuming $C(x) \neq 0$ is equivalent to assuming $\neg (T_0)$. 

101
Notice that

\[ s = \Pr[A] = \Pr[A \wedge \neg(T_0) \wedge T_d] \leq \Pr[\exists i \in [d] \text{ s.t. } A \wedge \neg(T_{i-1}) \wedge T_i] \leq \sum_{i=1}^{d} \Pr[A \wedge \neg(T_{i-1}) \wedge T_i]. \]

For every \(i \in [d]\), let \(E_i\) denote the event that indeed \(\bar{V}_i(w_1) = v_1\) and \(\bar{V}_i(w_2) = v_2\).\(^{21}\)

Then,

\[ \Pr[A \wedge \neg(T_{i-1}) \wedge T_i] = \Pr[A \wedge \neg(T_{i-1}) \wedge T_i \wedge E_i] + \Pr[A \wedge \neg(T_{i-1}) \wedge T_i \wedge \neg(E_i)] \]

The soundness property of the interactive sum-check protocol implies that

\[ \Pr[A \wedge \neg(T_{i-1}) \wedge T_i \wedge E_i] \leq \Pr[A \wedge \neg(T_{i-1}) \wedge T_i \wedge E_i] \leq \frac{3m \cdot 2\delta}{|F|} \leq \frac{6m\delta}{|F|}. \]

The fact that any two distinct univariate degree \(t\) polynomials agree on at most \(t\) points implies that

\[ \Pr[A \wedge \neg(T_{i-1}) \wedge T_i \wedge \neg(E_i)] \leq \Pr[A \wedge T_i \wedge \neg(E_i)] \leq \frac{m(|\mathbb{H}| - 1)}{|F|} \leq \frac{m\delta}{|F|}. \]

Thus,

\[ \Pr[A \wedge \neg(T_{i-1}) \wedge T_i] \leq \frac{6m\delta}{|F|} + \frac{m\delta}{|F|} = \frac{7m\delta}{|F|}. \]

All in all, we get that

\[ s \leq \frac{7m\delta}{|F|}. \]

Taking \(F\) such that \(|F| \geq 700m\delta = \text{poly}(|\mathbb{H}|)\), we get that \(s \leq \frac{1}{100}\) as desired.

**Complexity.** Recall that the bare-bones protocol proceeds in \(d\) phases (where \(d\) is the depth of \(C\)). In the \(i\)’th phase (1 ≤ \(i\) ≤ \(d\)) the prover reduces the task of proving that \(\bar{V}_{i-1}(z_{i-1}) = r_{i-1}\) to the task of proving that \(\bar{V}_i(z_i) = r_i\). This is done by running a sum-

\(^{21}\)Note that \((w_1, v_1)\) and \((w_2, v_2)\) depend on the phase \(i \in [d]\). For the sake of simplicity, this dependence is not captured in our notations.
check protocol and an additional short interactive protocol.

The complexity of the $i$'th phase of the protocol, $1 \leq i \leq d$, is as follows (we ignore the difference between $m$ and $m'$ that comes into play only in the $d$'th phase):

1. The running time of the prover $P_i$ is $\text{poly}(|\mathbb{F}|^m) = \text{poly}(S)$, both in the sum-check protocol (see Lemma 3.2.4) and in the proceeding interactive process.

2. The running time of the verifier $V_i$ (with oracle access to $\mathcal{F}$) is $\text{poly}(|\mathcal{F}|, m) = \text{poly}(d, \log(S))$, both in the sum-check protocol (see Lemma 3.2.4) and in the proceeding interactive process.

   The space used by $V_i$ is $O(\log(|\mathcal{F}|) \cdot m) = O(\log(S))$, both in the sum-check protocol (see Lemma 3.2.4) and in the proceeding interactive process. Note that the only information that the prover and verifier need to “remember” for the next phase is the values $i, z, r_i$ (and they don’t need to remember any information from previous phases). This implies, in particular, that the total space used by the verifier in all phases is only $O(\log(|\mathcal{F}|) \cdot m) = O(\log(S))$, not much larger than the space used in a single phase.

3. The sum-check protocol has communication complexity $\text{poly}(|\mathcal{F}|, m)$ (see Lemma 3.2.4), and the proceeding interactive process has communication complexity $\text{poly}(|\mathcal{F}|)$. Thus, in total, each phase has communication complexity $\text{poly}(|\mathcal{F}|, m) = \text{poly}(d, \log(S))$. Moreover, the verifier $V_i$ is public-coin, and the number of random bits it sends to the prover $P_i$ in each phase is $O(\log(|\mathcal{F}|) \cdot m)$. This, together with the fact that the only information that the prover needs to “remember” for the next phase is the values $i, z, r_i$ (and does not need to remember any information from previous phases), implies that each message sent by the prover depends only on the preceding $O(\log(|\mathcal{F}|) \cdot m) = O(\log(S))$ random bits sent by the verifier.

4. In each phase, the verifier queries each $\tilde{a}$, and $\tilde{\text{mult}}$, only at a single location. The verifier’s queries to $\tilde{a}$, and $\tilde{\text{mult}}$, are determined by its (public) coin tosses in the sum-check protocol and are thus also uniformly random (over the verifier’s coin tosses).
Finally, the verifier $V_1$ needs to verify on his own that $\tilde{V}_d(z_d) = r_d$. This amounts to computing a single point in the low-degree extension of the input $x$ (with respect to $F, \mathbb{H}, m'$).

This can be done (by Claim 3.2.2) in time $k \cdot \text{poly}(|\mathbb{H}|, m') = k \cdot \text{poly}(d, \log(S))$ and space $O(\log(|\mathbb{H}|) \cdot m') = O(\log(S))$. If the verifier has an oracle to the low degree extension of $x$, then this can instead be accomplished by a single (unit cost) oracle query to point $z_d$, a uniformly random point determined by the verifier’s coin tosses in the $d$'th phase of the protocol.

3.4 Interactive Proofs: Implementing the Bare-Bones Protocol

Recall that our goal is to construct a protocol in which a prover, who is given a circuit $C : \{0, 1\}^k \rightarrow \{0, 1\}$ of size $S$ and of depth $d$, and a string $x \in \{0, 1\}^k$, can prove to a verifier that $C(x) = 0$, while the verifier runs in time significantly less than $S$, which is the time that it would take him to evaluate $C(x)$ on his own. We also want the verifier to use as little space as possible, continuing the study of space-bounded verifiers.

In Section 3.3 we presented the bare-bones protocol, where we gave the verifier oracle access to a set of functions $\mathcal{F} = \{\tilde{\text{add}}_i, \tilde{\text{mult}}_i\}_{i \in [d]}$, which are (extensions of) functions that define $C$. With these oracles the verifier was both time efficient and space efficient. In our results, however, we want to work in the standard model of interactive proofs, where the verifier does not have oracle access to these functions. Thus, our goal in this section is to implement or realize the bare-bones protocol in the standard model of interactive proofs.

We build on the foundations laid in the previous section to construct standard interactive proofs for uniform languages, where the complexity of the verifier and the prover are comparable to those in the bare-bones protocol. In particular, we provide methods for the verifier to reliably obtain the values of $\{\tilde{\text{add}}_i, \tilde{\text{mult}}_i\}_{i \in [d]}$ in a time-efficient and space-efficient manner. This section consists of three parts:

First, in Subsection 3.4.1 we show how to implement the bare-bones protocol for lan-
languages in $\mathcal{NL}$; i.e., languages computable in logarithmic non-deterministic space. To prove this result, we show that such languages have circuits of poly-size and polylog-depth, for which $\{\text{add}, \text{mult}, \}_{r \in \mathbb{Q}}$ can be evaluated by the verifier in polylog-time and log-space (without using any non-standard oracles).

**Second**, in Subsection 3.4.2 we use the above result on delegating $\mathcal{NL}$ computations, to show how to implement the bare-bones protocol for any language in $(\mathcal{L}$-uniform) $\mathcal{NC}$. To this end, we show that for such languages, there exists an interactive sub-protocol that the prover can use to prove to the verifier the values of $\{\text{add}, \text{mult}, \}_{r \in \mathbb{Q}}$. In these subprotocols the verifier runs in poly-logarithmic time and logarithmic space (and the prover runs in polynomial time). We then implement the bare-bones protocol by replacing the verifier’s oracle calls to $\mathcal{F}$ with these interactive sub-protocols, in which the prover provides the verifier with the values of functions in $\mathcal{F}$ and proves their correctness. We also use this idea to obtain interactive public-coin proofs with log-space verifiers for all of $\mathcal{P}$ (see Corollary 3.1.3).

**Finally**, in Subsection 3.4.3 we take an alternate approach that does not rely on the uniformity of the circuit (the computation) being delegated. Instead, we split the delegation process into two phases: an off-line (non-interactive) pre-processing phase, run (only) by the verifier before the input $x$ to the circuit is even specified. In this phase the verifier gets access the entire circuit and works in time that is polynomial in the size of the circuit. The output of the pre-processing phase is a short data string (much shorter than the circuit size). Then, after the input $x$ is specified, the prover and verifier run an on-line interactive proof phase. This on-line protocol is an implementation of the bare-bones protocol. In particular, in this on-line phase the verifier’s and the prover’s running times, as well as the communication complexity, is as in the bare-bones protocol. This result is formally stated in Theorem 3.1.4.

We begin by (briefly) reviewing the notation and conventions introduced in Section 3.3.

**Conventions: a Recap.** Throughout this section, whenever we speak of a circuit $C$ for computing a language or function, we follow the conventions introduced in the bare-bones protocol (Section 3.3.1). Let $\mathbb{H}$ be an extension field of $\mathbb{GF}[2]$, $\mathcal{F}$ an extension field of $\mathbb{H}$ (and
thus also of GF[2]). We always think of the circuit $C$ (which is defined over the field GF[2]), as a \textit{layered} arithmetic circuit with fan-in 2, over the extension field $F$. Further, $C$’s gates are labeled so that $g_{i,z}$ denotes the $z$-th gate in layer $i$, where we alternately treat $i$ and $z$ as boolean strings or values in $\{0, 1, \ldots, d\}$ and $\{0, 1, \ldots, |C| - 1\}$ (respectively). The top or output layer is layer 0, the bottom or input layer is layer $d$. We assume here that the bottom layer includes $n$ input gates and 2 “constant” gates, one for the constant 0 and one for the constant 1.

We define the functions $\text{add}_i, \text{mult}_i$, as in Section 3.3.1: they take as input three labels in $\{0, 1, \ldots, |C| - 1\}$, the first corresponding to a gate in layer $i - 1$ and the other two corresponding to gates in layer $i$. The functions answer 1 if the first gate is an addition or multiplication (respectively) of the other two. Note that, as in Section 3.3.1, $\text{add}_d$ and $\text{mult}_d$ take as input three labels, where the first label in $\{0, 1, \ldots, |C| - 1\}$ corresponds to a gate in layer $d - 1$, and the other two labels in $\{0, 1, \ldots, n + 1\}$ correspond to input gates. Throughout this section we abuse notation and do not distinguish between the functions $\text{add}_i, \text{mult}_i$, as defined above, and the \textit{boolean circuits} we construct to compute them, which we also call $\text{add}_i, \text{mult}_i$.

For $m, m'$ such that $|\mathbb{H}^m| \geq |C|$ and $|\mathbb{H}^{m'}| \geq n + 2$, the functions $\alpha, \alpha'$ are the functions that take a vector in $\mathbb{H}^m$ (respectively $\mathbb{H}^{m'}$) and output its lexicographic order.

As before, let $\tilde{\text{add}}_i, \tilde{\text{mult}}_i : (\mathbb{F}^m)^3 \rightarrow \mathbb{F}$ be some extensions of $\text{add}_i, \text{mult}_i$, (with respect to $(\mathbb{H}, \mathbb{F}, m, m')$). Namely, if all three of their inputs are in $\mathbb{H}^m$, they translate them into three gate labels (using $\alpha, \alpha'$), and answer as $\text{add}_i, \text{mult}_i$. In particular, if the inputs are all in $\mathbb{H}^m$, then the answer is always 0 or 1. If even one of the inputs is an element of $\mathbb{F}^m$ that is not in $\mathbb{H}^m$, $\tilde{\text{add}}_i, \tilde{\text{mult}}_i$ output some value in $\mathbb{F}$. An important property we want from these functions is that they have low degree $\delta$, in particular $\delta$ will be significantly smaller than $|\mathbb{F}|$. Throughout this section we abuse notation and do not distinguish between \textit{functions} $\tilde{\text{add}}_i, \tilde{\text{mult}}_i$, as described above, and the \textit{arithmetic circuits} we construct to compute them, which we also call $\tilde{\text{add}}_i, \tilde{\text{mult}}_i$. 106
3.4.1 Interactive Proofs for $\mathcal{NL}$

In this subsection we show how to implement the bare-bones protocol for any language in $\mathcal{NL}$. The full result is stated in Theorem 3.4.4. We proceed by first showing in Subsection 3.4.1 that languages in $\mathcal{NL}$ are computable by circuits for which $\tilde{\text{add}}, \tilde{\text{mult}}$, can be evaluated in poly-logarithmic time and logarithmic space. In Section 3.4.1 we implement the bare-bones protocol by having the verifier replace its oracle $\mathcal{F}$ with these easy to evaluate $\tilde{\text{add}}, \text{mult}$.

Circuits for $\mathcal{NL}$ Languages with Efficient Low Degree $\tilde{\text{add}}, \text{mult}$

Overview. Our goal in this subsection is to show that every language in $\mathcal{NL}$ has (for every input length) a polylog-depth and poly-size arithmetic circuit that computes it. This circuit has the additional property that $\{\tilde{\text{add}}, \tilde{\text{mult}}\}$ are polylog-size arithmetic circuits that are log log-space uniform. This implies, in particular, that they can be evaluated in polylog-time and log-space, as desired. The degree of these circuits is denoted by $\delta$, and is significantly smaller than $|\mathcal{F}|$. We do this in three steps:

First, we show that every language in $\mathcal{NL}$ has (for every input length) a poly-size and polylog-depth arithmetic circuit, for which $\tilde{\text{add}}, \tilde{\text{mult}}$, are polylog-size, log log-space uniform, constant-depth ($\mathcal{AC}^0$) boolean circuits. This result is stated in Lemma 3.4.1. Second, we show how to compute the low-degree extensions of $\alpha, \alpha'$ using a polylog-size, log log-space uniform and low degree (degree $|\mathcal{H}| - 1$) arithmetic circuit. This result is stated in Claim 3.4.2. Finally, we combine the two results above, to show that every language in $\mathcal{NL}$ has (for every input length) a poly-size and polylog-depth arithmetic circuit, for which $\tilde{\text{add}}, \tilde{\text{mult}}$, are polylog-size, log log-space uniform, low degree ($\delta$) arithmetic circuits. This is the result we need for implementing the Bare-Bones protocol for $\mathcal{NL}$ computations, and it is stated in Lemma 3.4.3.

---

22Throughout this section, when we refer to a circuit as being $s(n)$-space uniform we always refer (implicitly or explicitly) to a family of circuits, one for every input length. The family is $s(n)$-space uniform if there exists a Turing machine that takes as input $1^n$, outputs the entire circuit, and uses only $O(s(n))$ space. It thus also runs in time at most $2^{O(s(n))}$. 

107
Step 1: Small, Uniform, Constant Depth Boolean $add, mult$. We begin by showing that every language in $\mathcal{NC}$ has (for each input length) a poly-size, polylog-depth circuit, for which $add, mult$, are log log-space uniform, polylog-size and constant-depth ($\mathcal{AC}^0$) boolean circuits. We state this result in a more general manner, for any space and time bounds.

Lemma 3.4.1. Let $L$ be any language computed by a non-deterministic Turing Machine $T$ in time $t(n)$ and space $s(n)$ (we assume $s(n) = \Omega(\log(n))$). Let $n$ be any input length.

There exists an arithmetic circuit $C$ over $\mathbb{GF}[2]$ for computing $L$ on inputs of length $n$. The circuit $C$ is of size $\text{poly}(2^{s(n)})$ and depth $d(n) = O(s(n) \cdot \log(t(n)))$ (with fan-in 2).

For all $i \in \{1, \ldots, d(n)\}$, the circuits $add$, and $mult$, are $\text{poly}(s(n))$-size constant depth ($\mathcal{AC}^0$) circuits. These circuits can be generated by a $O(\log(s(n)))$-space Turing machine $G$, that takes $(n, i, b)$ as input (where $i \in \{1, \ldots, d(n)\}, b \in \{0, 1\}$). It outputs the circuit $add$, if $b = 0$, and it outputs the circuit $mult$, if $b = 1$.

Proof of Lemma 3.4.1.

Preliminaries. We begin with notation and preliminaries, reviewing how to translate $T$’s computations into questions about the adjacency matrix of it’s computation graph.

We assume (without loss of generality) that the machine $T$ has an input-tape and a single work-tape, both over a boolean alphabet. Let $Q$ be the (constant size) set of possible machine states. The transition table $R_T$ of $T$ is a (constant-size) collection of pairs of tuples:

$$R_T \subseteq (Q \times \{0, 1\} \times \{0, 1\}, Q \times \{L, R\} \times \{0, 1\} \times \{L, R\}).$$

The first tuple includes a state and two alphabet symbols (the first read by the input-tape reading head, the second by the work-tape reading head). The second tuple includes a new machine state, a direction to move the input-tape reading head, a new value for location just read from the work-tape, and a direction to move the work-tape reading head. Two tuples are in the (non-deterministic) machine’s table $R_T$ if the (non-deterministic) machine may move from the state described by the first item in the tuple in the manner specified by the
second item.

For an input \( x \in \{0,1\}^n \), we encode a configuration of the machine as a vector \( c = (q, i, j, t) \in \{0,1\}^{g(n)} \), where \( g(n) = O(1) + \log(n) + \log(s(n)) + s(n) = O(s(n)) \) is the length of the representation of a configuration. Each configuration includes \( q \), the machine’s internal state \((O(1) \text{ bits})\), the location \( i \in [n] \) of the input-tape reading head, the location \( j \in [s(n)] \) of the work-tape reading head, and the contents \( t \in \{0,1\}^{s(n)} \) of the work-tape. Assume w.l.o.g. that the all 0 vector denotes the machine’s initial configuration (we call this vector \( a \)) and that the machine has a unique accepting configuration, encoded by the vector \( b \) (say this is the all 1’s vector).

One can view the machine’s configurations on an input \( x \in \{0,1\}^n \) as vertices of a directed acyclic graph, where there is an edge from configuration \( u \) to configuration \( v \) if, on the input \( x \), the (non-deterministic) machine \( T \) can move from configuration \( u \) to configuration \( v \). We add self-loops to all the vertices in the graph. \( T \) accepts an input \( x \) if and only if there is a directed path from \( a \) to \( b \) in the graph.

Let \( B_x \) denote the \((0/1)\) adjacency matrix of this graph (with 1’s on the main diagonal denoting the self-loops). We construct a sequence of matrices: \( B_{\log(t(n))}, \ldots, B_1, B_0 \). The \((u, v)\)-th entry of \( B_p \) is 1 iff there is a path of length at most \( 2^{\log(t(n)) - p} \) from \( u \) to \( v \) in the machine’s configuration graph, i.e. iff the machine \( T \) on input \( x \) can go from configuration \( u \) to configuration \( v \) in \( 2^{\log(t(n)) - p} \) steps or less. Otherwise the \((u, v)\)-th entry is 0. Observe that \( B_{\log(t(n))} \) is simply the adjacency matrix \( B_x \). To compute the matrix \( B_{p-1} \) from \( B_p \), we use the fact that there is a path of length at most \( 2 \cdot \ell \) from \( u \) to \( v \) iff there exists \( w \) such that there is a path of length at most \( \ell \) from \( u \) to \( w \) and a path of length at most \( \ell \) from \( w \) to \( v \). In other words (using arithmetic over \( \mathbb{GF}[2] \)):

\[
B_{p-1}[u, v] = 1 + \prod_{w \in \{0,1\}^{g(n)}} (1 + B_p[u, w] \cdot B_p[w, v]).
\] (3.6)

The question of whether \( T \) accepts an input \( x \) is equivalent to asking whether or not \( B_0[a, b] = 1 \).
The Circuit $C$. The layered circuit $C$ computes one after another the matrices $B_{\log(t(n))} = B_2, B_{\log(t(n)-1)}, \ldots, B_1, B_0$. Once $C$ computes $B_0$ it outputs its $(a, b)$-th entry. Each layer of $C$ is made up entirely of either addition or multiplication gates. The computation is done by layered sub-circuits: the bottom sub-circuit, given the input $x \in \{0, 1\}^n$, computes (in constant depth) the adjacency matrix $B_x = B_{\log(t(n))}$. There are $\log(t(n))$ intermediate sub-circuits above the bottom sub-circuit (numbered from top to bottom $0, \ldots, \log(t(n)) - 1$), each of depth $g(n) + O(1) = O(s(n))$ and size $\text{poly}(2^n)$. The $i$-th sub-circuit uses the matrix $B_{i+1}$, computed by the previous sub-circuit, to compute $B_i$ (this is done as specified by Equation (3.6)). Finally the top (0-th) sub-circuit computes $B_0$ and outputs its $(a, b)$-th entry. The depth and size of $C$ are as claimed in the lemma statement.

The input layer includes the $n$ input gates as well as 2 “constant gates” labeled $n, n + 1$, and holding the values 0 and 1 (respectively). Each of the other layers includes at most $2^{3g(n)} + 2$ gates. These include at most $2^{3g(n)}$ “standard” gates (whose values differ between layers), and 2 “constant gates”, as in the input layer, whose values are 0 and 1. As we did for the input layer, we label the constant gates within each layer by $2^{3g(n)}, 2^{3g(n)} + 1$ (respectively). By convention, if an intermediate layer includes less than $2^{3g(n)}$ “standard” gates, then we sometimes use shorter labels, implicitly fixing their least significant bits to be 0 and ignoring them in the exposition below.\(^{23}\)

Thus, each intermediate gate in the circuit is labeled as a tuple $\ell = (p, k, z)$, where $p \in \{0, 1, \ldots, \log(t(n))\}$ denotes the gate’s sub-circuit, $k \in \{1, \ldots g(n) + O(1)\}$ denotes its layer within that sub-circuit and $z \in \{0, 1, \ldots, \text{poly}(2^n)\}$ is its index within that layer.

We proceed with a detailed layer-by-layer construction. As we describe each layer of the circuit (say the $i - 1$-th), we also argue that $\text{add}_i, \text{mult}_i$ are both $\log(g(n))$-space uniform, $\text{poly}(g(n))$-size, constant depth $(\text{AC}^0)$ circuits.

The Constant Gates. We begin by describing how the constant gates are computed. Then, throughout the rest of the exposition we take for granted the fact that in each layer

\(^{23}\)in fact the circuits $\text{add}_i$ and $\text{mult}_i$ constructed below must verify that these bits are all 0, this is easily accomplished.
of the circuit, the values of two constant gates (labeled as above) are computed correctly.

For every layer $i$ except the layer above the input layer (the $(\log(t(n)), 0)$-th layer): if $i$ is a layer of multiplication gates, then its constant gate $2^{3^g(n)} + b$ (computing the constant $b \in \{0, 1\}$) is a multiplication gate, multiplying the gates $2^{3^g(n)} + b$ (computing $b$) and $2^{3^g(n)} + 1$ (computing 1) in layer $i + 1$. For an addition layer, its constant gate $2^{3^g(n)} + b$ is an addition gate, adding the gates $2^{3^g(n)} + b$ (computing $b$) and $2^{3^g(n)}$ (computing 0) in layer $i + 1$. For the layer above the input layer (layer $(\log(t(n)), 0)$, an addition layer in the construction below), its constant gate $2^{3^g(n)} + b$ is an addition gate of the input layer gates: $n + b$ (computing $b$), and $n$ (computing 0).

The output of $\text{add}$, $\text{mult}$, on the constant gates in layer $i - 1$ is simple to compute. It is just a matter of checking whether the labels of the layer $i$ gates (of size $O(g(n))$) are exactly equal to what they should be. For example, for an addition layer, on input $z_1 = 2^{3^g(n)} + 1$ (constant value 1), $\text{add}$, accepts iff $z_2 = 2^{3^g(n)} + 1$ (constant value 1) and $z_2 = 2^{3^g(n)}$ (constant value 0). These computations can be done by $\text{poly}(g(n))$-size $\mathcal{AC}^0$ circuits (one for addition gates, one for multiplication). Moreover, these circuits can be generated in $\log(g(n))$-space (given $n, i$). In the descriptions that follow, when we describe the circuits $\text{add}_i$ and $\text{mult}_i$, we always implicitly mean that they first check (in $\mathcal{AC}^0$) whether the gate label in layer $i$ is of a constant gate, and if so they run the circuits described above. It is easy to verify that this maintains the depth, size and uniformity of the circuits described below (up to constant factors). For simplicity, we do not explicitly note this below.

**The Input Sub-Circuit.** The input (or bottom) sub-circuit of $C$, has as its input $x \in \{0, 1\}^n$. As its $2^{g(n)} \cdot 2^{g(n)}$ (“standard”) outputs it has the adjacency matrix $B_x$ (or $B_{\log(t(n))}$). The sub-circuit has 2 layers: the input layer, with $n + 2$ gates, and the top layer with $2^{2^{g(n)}}$ standard addition gates (and 2 constant gates as described above).

Let us examine the $(u, v)$-th entry of the matrix $B_x$: configuration $u$ reads only one input bit, say the $i$-th ($i \in [n]$) bit from the input $x$. There are only 4 possibilities (arithmetic is over $\mathbb{GF}[2]$):

1. Configuration $u$ can never go to $v$, regardless of $x_i$: $B_x[u, v] = 0$.
2. Configuration $u$ can always go to $v$, regardless of $x$: $B_x[u,v] = 1$. Note that this also includes the case $u = v$, as all vertices in the graph have self-loops.

3. Configuration $u$ can go to $v$ iff $x = 1$: $B_x[u,v] = x$.

4. Configuration $u$ can go to $v$ iff $x = 0$: $B_x[u,v] = 1 + x$.

Thus each entry of the bottom sub-circuit’s output depends on (at most) a single input bit, and certainly this layer’s output can be computed in depth 1.

We now turn to describing the circuits $add$, and $mult$, for the top layer of the input sub-circuit (layer $(\log(t(n)),0)$ of $C$). This layer has only addition gates. Thus for any query about multiplication gates, $mult(\log(t(n)),1)$ simply answers 0. To compute whether the input gates with labels $z_2, z_3 \in [n+2]$ are the children of an addition gate $z_1 \in \{0,1\}^{2g(n)}$ at the top layer of the input sub-circuit, the circuit $add(\log(t(n)),1)$ proceeds as follows:

1. Parse $z_1$ as a pair $(u,v)$ of machine configurations. Parse $u = (q_1 , i_1 , j_1 , t_1 )$ and $v = (q_2 , i_2 , j_2 , t_2 )$. Now the question is what is the value of $B_x[u,v]$.

2. There are four possible cases (as above). The circuit $add(\log(t(n)),1)$ computes which of these four cases occurs. We claim this can be done by a $\log(g(n))$-space uniform $AC^0$ boolean circuit of size $poly(g(n))$.

First, the circuit checks whether $u = v$. If so, then there is a self loop at this graph entry and we treat this gate as a Case 2 gate. Otherwise ($u \neq v$), the circuit first verifies that $t_1$ and $t_2$ are identical everywhere except at location $j_1$. It then checks for each possible transition in the machine’s (constant-size) transition table $R_T$, whether reading either or both of the possible values of $x_i$, could cause configuration $u$ to move to configuration $v$ via that transition. This is equivalent to checking (for each possible transition) whether either or both possible bit values of $x_i$ could make the internal state $q_i$ change to $q_2$, the input-tape reading head move from location $i_1$ to $i_2$, the work-tape reading head move from location $j_1$ to $j_2$, and the $j_1$-th bit of $t_1$ to be overwritten by that of $t_2$. All of these conditions can be verified by a $\log(g(n))$-space uniform $AC^0$ circuit of $poly(g(n))$-size.
3. After running this computation, \( \text{add}_{\log(t(n)),1} \) “knows” which of the four possibilities above is the case for \((u, v)\). In Case 1 neither possible value can cause the transition. In this case, \( \text{add}_{\log(t(n)),1} \) accepts if and only if \( z_2 = n \) and \( z_3 = n \) (i.e., this gate’s value is \( 0 = 0 + 0 \)). In Case 2 both possible values can cause the transition. In this case, \( \text{add}_{\log(t(n)),1} \) accepts if and only if \( z_2 = n + 1 \) and \( z_3 = n \) (i.e., this gate’s value is \( 1 = 1 + 0 \)). In Case 3 only \( x_{z_1} = 1 \) causes the transition. In this case, \( \text{add}_{\log(t(n)),1} \) accepts if and only if \( z_2 = i_1 \) and \( z_3 = n \) (i.e., this gate’s value is \( x_{i_1} = x_{i_1} + 0 \)). Finally, in Case 4 only \( x_{z_1} = 0 \) causes the transition. In this case, \( \text{add}_{\log(t(n)),1} \) accepts if and only if \( z_2 = i_1 \) and \( z_3 = n + 1 \) (this gate’s value is \( x_{i_1} + 1 \)).

The circuit \( \text{add}_{\log(t(n)),1} \) as above is a \( O(\log g(n)) \)-space uniform \( \mathcal{AC}^0 \) boolean circuit of size \( \text{poly}(g(n)) \).

**Intermediate Sub-Circuits.** The \( p \)-th intermediate sub-circuit takes as input the \( 2^g(n) \times 2^g(n) \) 0/1-matrix \( B_{p+1} \), the output layer of the sub-circuit below it, and outputs \( B_p \) (also a \( 2^g(n) \times 2^g(n) \) 0/1 matrix). This is done via the rule stated in Equation 3.6, using \( g(n) + 3 \) layers. The bottom layer computes the \( 2^{3 \cdot g(n)} \) products of pairs of matrix entries needed to compute the large product in Equation 3.6. For each such pair product \( B_{p+1}[u, w] \cdot B_{p+1}[w, v] \), the next layer computes \( 1 + B_{p+1}[u, w] \cdot B_{p+1}[w, v] \). Then, the next \( g(n) \) layers compute \( \prod_{w \in \{0, 1\}^{g(n)}} \left( 1 + B_{p+1}[u, w] \cdot B_{p+1}[w, v] \right) \). Finally, the top layer of the sub-circuit adds 1 to each such product. It computes for each pair \((u, v)\) the value \( 1 + \prod_{w \in \{0, 1\}^{g(n)}} \left( 1 + B_{p+1}[u, w] \cdot B_{p+1}[w, v] \right) \), which is the \((u, v)\)-th entry of \( B_p \) (by Equation 3.6).

A more detailed account follows: let \( p \in \{0, 1, \ldots, \log(t(n)) - 1\} \) be the sub-circuit’s index among all the intermediate sub-circuits. Let \( B_{p+1} \) be the \( 2^{g(n)} \times 2^{g(n)} \) 0/1-matrix which is the output of the sub-circuit below this one. We label the sub-circuit’s layers top to bottom as \((p, 0), (p, 1), \ldots, (p, g(n) + 2)\). The intermediate sub-circuits are all identical, and so we can mostly disregard \( p \) for the rest of this exposition.

The bottom layer (layer \((p, g(n)) + 2\)) has \( 2^{3 \cdot g(n)} \) multiplication gates, each labeled by three configurations \((u, v, w)\). The value of the \((u, v, w)\)-th gate should be \( B_{p+1}[u, w] \cdot B_{p+1}[w, v] \).
For this layer \( \text{add}_i \) is always 0, \( \text{mult}_i \) is easy to compute, as it accepts \( z_1 = (u, v, w) \) iff \( z_2 = (u, w) \) and \( z_3 = (w, v) \).

In the next layer (layer \( (p, g(n) + 1) \)), the gate labeled \( (u, v, w) \) computes the value of gate \( (u, v, w) \) in the bottom layer plus 1. Here \( \text{mult}_i \) is always 0, \( \text{add}_i \) accepts \( z_1 = (u, v, w) \) iff \( z_2 = (u, v, w) \) and \( z_3 = 2^{3^g(n)} + 1 \) (the constant 1 gate).

For \( k \in \{g(n), \ldots, 1\} \), the \( (p, k) \)-th layer has \( 2^{2^g(n)+k-1} \) gates, each labeled by a pair of configurations \( (u, v) \) and a string \( y \in \{0, 1\}^{k-1} \). The value of the \( (u, v, y) \)-th gate in the \( (p, k) \)-th layer is the product of the \( (u, v, y \circ 0) \)-th gate and the \( (u, v, y \circ 1) \)-th gate in the \( (p, k + 1) \)-th layer. The value of the \( (u, v) \)-th gate at layer 2 will indeed be the product \( \prod_{y \in \{0, 1\}^{g(n)}} (1 + B_{p+1}[u, w] \cdot B_{p+1}[w, v]) \), just as required. For these layers, \( \text{add}_{(p,k)} \) always outputs 0, as there are only multiplication gates. \( \text{mult}_{(p,k)} \) for \( z_1 = (u, v, y) \) accepts if and only if \( z_2 = (u, v, y \circ 0) \) and \( z_3 = (u, v, y \circ 1) \); otherwise it outputs 0.

It remains to describe the top layer (layer \( (p, 0) \)) in the sub-circuit, which has \( 2^{2^g(n)} \) gates. Gate \( (u, v) \) in layer \( (p, 0) \) computes 1 plus the value in gate \( (u, v) \) of layer \( (p, 1) \). The value computed by gate \( (u, v) \) in layer \( (p, 0) \) is (as required) \( 1 + \prod_{y \in \{0, 1\}^{g(n)}} (1 + B_{p+1}[u, w] \cdot B_{p+1}[w, v]) \). Here \( \text{mult}_i \) always outputs 0, \( \text{add}_i \) accepts \( z_1 = (u, v) \) iff \( z_2 = (u, v) \) and \( z_3 = 2^{3^g(n)} + 1 \) (the constant 1 gate).

We conclude that each layer in each of the intermediate sub-circuits has, as required, circuits \( \text{add}_i, \text{mult}_i \) that are \( O(\log g(n)) \)-space uniform \( \text{AC}^0 \) boolean circuit of size \( \text{poly}(g(n)) \).

**The Machine G:** It remains to show that there exists a single \( \log(g(n)) \)-space machine \( G \) that generates the circuit \( C \). More precisely, it remains to argue that there exists a \( \log(g(n)) \)-space machine \( G \), that takes as input a triple \( (n, i, b) \), and outputs the circuit \( \text{add}_i \) if \( b = 0 \), and the circuit \( \text{mult}_i \) if \( b = 1 \). The existence of such a machine follows easily from the above constructions of \( \text{add}_i \) and \( \text{mult}_i \) for each layer of the circuit \( C \). Note that all the intermediate sub-circuits are identical, there exists a single \( \log(g(n)) \)-space machine that on input \( (n, i, b) \) generates \( \text{add}_i \) or \( \text{mult}_i \) for each layer of these sub-circuits. The input sub-circuit (which is different from the intermediate ones) has only two layers, for which \( \text{add}_i \) and \( \text{mult}_i \) can be generated in \( \log(g(n)) \)-space as above. Thus there exists a single \( \log(g(n)) \)-space machine
G, such that for every layer $i$ in $C$, on input $(n, i, b)$, $G$ generates $\text{add}_i$ or $\text{mult}_i$ as required.

**Step 2: Small Low-Degree Circuits for $\alpha, \alpha'$.** Recall that our goal in this section is proving that every language in $\mathcal{NL}$ has circuits for which $\text{add}_i$ and $\text{mult}_i$ are themselves small low-degree uniform arithmetic circuits. Lemma 3.4.1 was the first step towards this goal. We now turn our attention to the mappings $\alpha, \alpha'$ that map arithmetic vectors to boolean (or numerical) labels of circuit gates. If we want to create small arithmetic circuits computing $\text{add}_i, \text{mult}_i$, these circuits should themselves be able to compute the mappings $\alpha$ and $\alpha'$. We show that $\alpha, \alpha'$ can be computed by $O(\log(|F|) + \log(m))$-space uniform, $\text{poly}(|F|, m)$-size arithmetic circuits (over $F$) of degree $|H| - 1$.

**Claim 3.4.2.** Fix $H$, an extension field of $GF[2]$, $F$ an extension field of $H$, and $m$ an integer value. Let $\alpha : H^m \to GF[2]^{|\log(|H|^m)|}$ be the function that maps a vector in $H^m$ to its lexicographic order, represented as a sequence of $\log(|H|^m)$ 0's and 1's (we can think of these as elements of $GF[2]$ or of its extension field $F$).

There exists an arithmetic circuit $T_{H,F,m} : F^m \to F^{|\log(|H|^m)|}$ that computes the low degree extension (with respect to $H, F, m$) of $\alpha$. The circuit $T_{H,F,m}$ is of size $\text{poly}(|H|) \cdot m$ and degree $|H| - 1$ in each of its inputs. It can be generated (from $(|H|, |F|, m)$) or evaluated (on an input in $F^m$) by a $O(\log(|F|) + \log(m))$-space uniform Turing machine.

**Proof of Claim 3.4.2.** Let $\alpha_1 : H \to F^{|\log(|H|)|}$ be the function that takes a single element of $H$ and maps it to its lexicographic order, represented as a sequence of $\log(|H|)$ 0's and 1's (elements of $GF[2]$ and thus also of $F$). Let $\tilde{\alpha}_1$ be the unique low-degree extension of $\alpha_1$. The circuit $T_{H,F,m}$ applies $\tilde{\alpha}_1$ to each of its $m$ inputs (elements of $F$), and outputs the concatenation of the $m$ outputs of $\tilde{\alpha}_1$. The reason that $T_{H,F,m}$ indeed computes the low degree extension also applies to functions with multiple outputs, as described in Section 3.2.1.

24 Again, in the past we usually worked with low degree extensions of functions that map multiple $H$-elements to a single $F$ element, whereas $\alpha_1$ maps a single $H$ element to many $F$ elements. This is a special case of a low degree extension (where the function may have multiple outputs). All the general results and construction still hold for this special case.
degree extension $\tilde{\alpha}$ of $\alpha$, follows from the fact that the size of $\mathbb{H}$ is a power of 2, which in turn follows from the fact that it is an extension field of $\mathbb{GF}[2]$.

To compute the low degree extension $\tilde{\alpha}_1$, the circuit uses a lookup table (with $|\mathbb{H}|$ entries) that contains the lexicographic order of each element in $\mathbb{H}$. The lookup table and its low-degree extension can be generated (given $m, \mathbb{F}, \mathbb{H}$), and evaluated on an input, in space $O(\log(|\mathbb{F}|))$ and time $\text{poly}(|\mathbb{H}|)$. This follows from Proposition 3.2.1, and from our assumption that addition and multiplication of field elements can be done in $O(\log(|\mathbb{F}|))$-space. Thus, the entire circuit $T_{\mathbb{H}, \mathbb{F}, m}$ is of size $\text{poly}(|\mathbb{H}|) \cdot m$. It can be generated, and evaluated on an input, by an $O(\log(|\mathbb{F}|) + \log(m))$-space uniform Turing machine.

**Step 3: Small, Uniform, Low Degree Arithmetic $\tilde{\text{add}}, \tilde{\text{mult}}$.** We are now ready to prove the main lemma of this subsection, showing that every language in $\mathcal{NC}$ has (for each input length) a poly-size and polylog-depth circuit, for which $\tilde{\text{add}}$, and $\tilde{\text{mult}}$, are log log-space uniform, polylog-size arithmetic circuits of degree that is significantly smaller than $|\mathbb{F}|$. We state the Lemma for any time and space bounds.

**Lemma 3.4.3.** Let $L$ be any language computed by a non-deterministic Turing Machine $T$ in time $t(n)$ and space $s(n)$ (we assume $s(n) = \Omega(\log(n))$). Fix $\mathbb{H}$ to be an extension field of $\mathbb{GF}[2]$, and $\mathbb{F}$ an extension field of $\mathbb{H}$ (and thus also of $\mathbb{GF}[2]$) of size at most $\text{poly}(s(n))$. Let $n$ be an input length.

There exists an arithmetic circuit $C$ over $\mathbb{GF}[2]$ (and thus also over $\mathbb{F}$) for computing $L$ on inputs of length $n$. The circuit $C$ is of size $\text{poly}(2^s(n))$ and depth $d(n) = O(s(n) \cdot \log(t(n)))$ (with fan-in 2).

For all $i \in \{1, \ldots, d(n)\}$, $\tilde{\text{add}}_i$, and $\tilde{\text{mult}}_i$ are arithmetic circuits over $\mathbb{F}$. All these circuits have degree at most $|\mathbb{H}| \cdot \text{poly}(s(n))$ (independent of $|\mathbb{F}|$), and size $\text{poly}(s(n), m)$. Moreover, they can be evaluated on an input or generated by $O(\log(s(n)) + \log(m))$-space uniform Turing machines. The generating machine $\tilde{G}$ takes $(n, i, b, |\mathbb{H}|, |\mathbb{F}|, m, m')$ as input ($i \in \{1, \ldots, d(n)\}, b \in \{0, 1\}$). If $b = 0$, then $\tilde{G}$ outputs the circuit $\tilde{\text{add}}_i$. If $b = 1$, then $\tilde{G}$ outputs the circuit $\tilde{\text{mult}}_i$.  

116
Proof of Lemma 3.4.3. The circuit $C$ is exactly the same circuit constructed in the proof of Lemma 3.4.1. The (arithmetic) circuits $\tilde{\text{add}}$, and $\tilde{\text{mult}}$, take as input 3 “gate labels” in $\mathbb{F}^m$. They apply the circuit $T_{\mathbb{H},\mathbb{F},m}$ (from Claim 3.4.2), which computes the low degree extension of $\alpha$, to each of these labels. If a “gate label” was in $\mathbb{H}^m$, the output should be a boolean translation: its lexicographic order. The circuits $\tilde{\text{add}}$, and $\tilde{\text{mult}}$, take the result of this translation, and use it as input for an arithmetization of the boolean circuit $\text{add}$, or $\text{mult}$, (respectively) from Lemma 3.4.1. If the original gate labels were all in $\mathbb{H}^m$, then the output should be equal to $\text{add}$’s or $\text{mult}$’s output on their boolean translations. A more detailed description follows.

Let $\{\text{add},\text{mult}\}$ be the boolean circuit families constructed in Lemma 3.4.1, and let $G$ be the machine that generates them. Transforming the boolean circuits into arithmetic circuits over $\mathbb{F}$ is easily done in the (by now) standard way: AND gates become multiplication, and a gate computing the NOT of some wire $w$ is turned into an arithmetic gate computing the value $1 - w$. This does not increase the circuit size, depth or uniformity by more than a constant factor. Since $\{\text{add},\text{mult}\}$ are $\mathcal{AC}^0$ circuits of size $\text{poly}(s(n))$ (independent of $|\mathbb{F}|$), the resulting arithmetic circuits are of size and degree at most $\text{poly}(s(n))$ (also independent of $|\mathbb{F}|$). We note also that since the boolean circuits are $O(\log(s(n)))$-space uniform and constant-depth, they, and their arithmetized versions, can be generated and evaluated in $O(\log(s(n)))$-space.

Note that these new (arithmetic) circuits take as input a boolean representation of gate labels, i.e. where each label is given as $O(\log(|C_n|))$ “boolean” inputs that are all the 0 or the 1 field element. The circuits $\tilde{\text{add}}$, and $\tilde{\text{mult}}$, on the other hand, take as input gate labels represented as arithmetic vectors in $\mathbb{F}^m$. Thus, what remains is to translate the arithmetic labels into boolean ones, by running them through translation circuit $T_{\mathbb{H},\mathbb{F},m}$ as constructed in Claim 3.4.2. This adds a (multiplicative) $|\mathbb{H}|$-factor to the degree, and a poly($|\mathbb{F}|$, $m$) additive factor to the size. When the inputs are all elements in $\mathbb{H}$, the translation circuits output the correct (boolean, i.e. consisting of 0/1 field elements) gate labels, and the circuit

---

26 We ignore here the fact that $\text{add}_{d(n)}$ and $\text{mult}_{d(n)}$ work over $m'$ inputs instead of $m$, this can be handled in a similar manner.
(the “arithmetized” \textit{add}, or \textit{mult},) correctly outputs whether or not the first gate is an addition or multiplication of the other two.

We obtain, as required, \textit{add}, and \textit{mult}, that have degree \(|\mathbb{H}| \cdot \text{poly}(s(n))\) (independent of \(|\mathbb{F}|\)), and size \text{poly}(s(n), m). Moreover, they are (again, as required), \(O(\log(s(n)) + \log(m))\)-space uniform: they can be generated by running a combination of \(G\) and the machine generating \(T_{\mathbb{H},\mathbb{F},m}\), and can be evaluated on an input in \(O(\log(s(n)) + \log(m))\) space.

Realizing the Bare-Bones Protocol

Using the above construction of small, low degree and uniform circuits \textit{add}, and \textit{mult}, we can now proceed to present our first implementation of the bare-bones protocol: a protocol for delegating \(\mathcal{NC}\) computations.

Recall that to implement the bare-bones protocol one must have a way for the verifier to implement \textit{add}, and \textit{mult}, oracles, whose degree is not too large. This is exactly what Lemma 3.4.3 provides! Namely, we now have a way for the verifier to implement the oracles in the bare-bones protocol on its own, where it can compute the answer to each “oracle” query in poly-logarithmic time and logarithmic space. This gives a protocol for delegating \(\mathcal{NC}\) computations. We state this result more generally, for given non-deterministic time and space bounds.

\textbf{Theorem 3.4.4.} \textit{Let} \(L\) \textit{be a language that can be computed by a non-deterministic Turing Machine using space} \(s(n)\) \textit{and time} \(t(n)\) \textit{(we assume} \(s(n) = \Omega(\log(n))\)). \(L\) \textit{has an interactive proof (an implementation of the bare-bones protocol) where:}

\begin{enumerate}
  \item \textit{The prover runs in time} \(\text{poly}(2^{s(n)})\), \textit{the verifier runs in time} \(n \cdot \text{poly}(s(n))\) \textit{and space} \(O(s(n))\).
  \item \textit{The protocol has perfect completeness and soundness} \(1/100\).
  \item \textit{The protocol is public-coin, with communication complexity} \(\text{poly}(s(n))\).
\end{enumerate}
Proof of Theorem 3.4.4. Fix an input length $n$. By Lemma 3.4.3, the language $L$ can be computed by a circuit $C$ of size $\text{poly}(2^{s(n)})$ and depth $d(n) = O(s^2(n))$. Fix $\mathbb{H}, \mathbb{F}, m, m'$ as in the bare-bones protocol (see Subsection 3.3.1). Namely, $\mathbb{H}$ is an extension field of $\mathbb{GF}[2]$ of size $O(s(n)^2)$, $\mathbb{F}$ is an extension field of $\mathbb{H}$ of size $\text{poly}(s(n))$, $m = O(s(n)/\log(s(n)))$, and $m' = O(\log(n)/\log(s(n)))$.

The circuits $\tilde{\text{add}}_t$ and $\tilde{\text{mult}}_t$, constructed in Lemma 3.4.3, are of degree $\delta = \text{poly}(s(n))$, and can be generated and evaluated (over $\mathbb{F}$) in time $\text{poly}(s(n))$ and space $O(\log(s(n)))$.

Now all that remains is to run the bare-bones protocol, replacing oracle calls to $\mathcal{F}$ for computing $\{\tilde{\text{add}}_t, \tilde{\text{mult}}_t\}$ with explicit computations of $\tilde{\text{add}}_t$ and $\tilde{\text{mult}}_t$ constructed above. From Theorem 3.3.1 we get that the protocol has perfect completeness, and soundness $1/100$. The prover’s work is $\text{poly}(2^{s(n)})$. The verifier’s work is only $n \cdot \text{poly}(s(n))$, and his space usage is $O(s(n))$. The protocol is public-coin, and the communication complexity is $\text{poly}(s(n))$.

Plugging in the parameters for languages in $\mathcal{NL}$, i.e. space $O(\log(n))$ and time $\text{poly}(n)$, we get that the prover is efficient, the verifier runs in quasi-linear time, and the communication complexity is $\text{polylog}(n)$. This is stated formally below (the proof is immediate from Theorem 3.4.4):

**Corollary 3.4.5.** Let $L$ be a language in $\mathcal{NL}$, i.e. one that can be computed by a non-deterministic Turing Machine using space $O(\log(n))$ and time $\text{poly}(n)$. $L$ has an interactive proof (an implementation of the bare-bones protocol) where:

1. The prover runs in time $\text{poly}(n)$, the verifier runs in time $n \cdot \text{polylog}(n)$ and space $O(\log(n))$.

2. The protocol has perfect completeness and soundness $1/100$.

3. The protocol is public-coin, with communication complexity $\text{polylog}(n)$. 

119
3.4.2 Interactive Proofs for \( \mathcal{L} \)-Uniform Circuits

In this subsection, we show how to implement the bare-bones protocol for any polynomial-size circuit that is log-space uniform. The complexity of the prover is polynomial in the circuit size, the complexity of the verifier is quasi-linear in the input length and polynomial in the circuit depth. The communication complexity is polynomial in the circuit depth and logarithmic in the circuit size.

This result uses Theorem 3.4.4 from the previous subsection for delegating \( \mathcal{A}/\mathcal{C} \) computations. We proceed in two steps. First we show that for log-space uniform circuits, the functions \( \tilde{\text{add}}, \) and \( \tilde{\text{mult}}, \) which are the unique low degree extensions of \( \text{add}, \) and \( \text{mult},, \) can themselves be computed in log-space (with respect to appropriately chosen fields). Then, applying Theorem 3.4.4 (or rather Corollary 3.4.5), we immediately conclude that a verifier can delegate the computation of \( \tilde{\text{add}}, \) and \( \tilde{\text{mult}}, \) to the prover. In this delegation protocol, the prover’s work is polynomial in the circuit size, but the verifier’s work is only poly-logarithmic in the circuit size (note that the input size of \( \text{add}, \) and \( \text{mult},, \) is itself only logarithmic).

So, given any log-space uniform circuit, the verifier and prover can run the bare-bones protocol. Whenever the bare-bones verifier needs to compute the value of \( \text{add}, \) or \( \text{mult},, \) the prover supplies it with the value, and proves that this value is correct by running, as a sub-protocol, the protocol of Theorem 3.4.4.

We begin by showing that for log-space uniform circuits, \( \tilde{\text{add}}, \) and \( \tilde{\text{mult}},, \) can be computed in log-space. We state the claim for any space bounds:

**Claim 3.4.6.** Let \( \mathcal{C} = \{C_n\} \) be a family of \( s(n) \)-space uniform circuits. Fix fields \( \mathbb{H} \) (an extension field of \( \mathbb{GF}[2] \)) of size \( O(s(n)) \), and \( \mathbb{F} \) (an extension field of \( \mathbb{H} \)) of size \( \text{poly}(s(n)) \). Take \( m = O(s(n)/\log(s(n))) \) and \( m' = O(n/\log(s(n))) \).

There exists an \( O(s(n)) \)-space Turing machine that computes the functions \( \tilde{\text{add}},, \tilde{\text{mult}}, \) for the circuit \( C_n \) (with respect to \( \mathbb{H}, \mathbb{F}, m, m' \). Here we take \( \tilde{\text{add}},, \tilde{\text{mult}},, \) to be the unique low degree extensions of \( \text{add},, \text{mult},, \) of degree \( |\mathbb{H}| - 1 \). The machine takes as input \( 1^n, \mathbb{H}, \mathbb{F}, m, m' \) and the input (in \( \mathbb{F}^m \) or \( \mathbb{F}^{m'} \)) to \( \tilde{\text{add}}, \) or \( \tilde{\text{mult}},. \)
Proof of Claim 3.4.6. Fix an input length \( n \). Every bit of the circuit \( C_n \) can be generated using \( O(s(n)) \) space, and thus the (boolean) functions \( \text{add} \) and \( \text{mult} \) can be evaluated using \( O(s(n)) \) space. By Claim 3.4.2, the function \( \alpha \) (or \( \alpha' \)) that converts a vector in \( \mathbb{F}^m \) (resp. \( \mathbb{F}'^m \)) into its (boolean) lexicographic order can be computed in space \( O(\log(|\mathbb{F}|) + \log(m)) = O(\log(s(n))) \) (the same holds for \( \alpha' \)).

Consider the two functions that take inputs in \( \mathbb{F}^m \), convert them into boolean gate labels using \( \alpha \), and then run \( \text{add} \) or \( \text{mult} \) on the result. By the above, these functions can be evaluated on inputs in \( \mathbb{F}^m \) in space \( O(s(n)) \). Now, note that \( \text{add} \) and \( \text{mult} \) are the unique low-degree extension of these functions. Thus, by Claim 3.2.2, these low-degree extensions can themselves be computed using an additional \( O(m \cdot \log(|\mathbb{F}|)) = O(s(n)) \) bits of space.

The total space needed to compute \( \tilde{\text{add}}, \tilde{\text{mult}} \), is \( O(s(n)) \). Clearly, the above holds also for \( \tilde{\text{add}}_d, \tilde{\text{mult}}_d \), where \( d \) is the bottom layer of the circuit. ■

We now proceed with a result about delegating the computation of languages computable by log-space uniform circuits. This is the result claimed in Theorem 1.1.1 of Section 3.1.1.

Theorem 3.4.7 (Theorem 1.1.1 of Section 3.1.1, restated). Let \( L \) be a language that can be computed by a family of \( O(\log(S(n))) \)-space uniform boolean circuits of size \( S(n) \) and depth \( d(n) \). \( L \) has an interactive proof where:

1. The prover runs in time \( \text{poly}(S(n)) \). The verifier runs in time \( n \cdot \text{poly}(d(n), S(n)) \) and space \( O(\log(S(n))) \). Moreover, if the verifier is given oracle access to the low-degree extension of its input, then its running time is only \( \text{poly}(d(n), S(n)) \).

2. The protocol has perfect completeness and soundness \( \frac{1}{2} \).

3. The protocol is public-coin, with communication complexity \( d(n) \cdot \text{polylog}(S(n)) \).

Proof of Theorem 1.1.1. Fix an input length \( n \). By the conditions of the theorem, on any input length \( n \), the language \( L \) can be computed by a \( O(\log(S(n))) \)-space uniform arithmetic circuit \( C \) over \( \mathbb{GF}[2] \), of size \( S(n) \) and depth \( d(n) \). Assume (without loss of generality) that \( C \) has fan-in 2.

121
Fix $\mathbb{H}, \mathbb{F}, m, m'$ as in the bare-bones protocol (see Subsection 3.3.1). Namely, $\mathbb{H}$ is an extension field of $\mathbb{GF}[2]$ of size $\max\{\log(S(n)), d(n)\}$, $\mathbb{F}$ is an extension field of $\mathbb{H}$ of size $\text{poly}(\|\mathbb{H}\|)$, $m = O(\log(S(n))/\log(\|\mathbb{H}\|))$, and $m' = O(\log(n)/\log(\|\mathbb{H}\|))$.

We run the bare-bones protocol of Section 3.3, taking $\tilde{\text{add}}$, and $\tilde{\text{mult}}$, to be the unique low-degree extensions of $\text{add}$, and $\text{mult}$, respectively (of degree $\|\mathbb{H}\| - 1$ in each variable). The verifier in the bare-bones protocol queries $\tilde{\text{add}}$, and $\tilde{\text{mult}}$, at most $2d(n)$ times: In each phase of the protocol (or layer of the circuit) he queries $\tilde{\text{add}}$, once and queries $\tilde{\text{mult}}$, once. Now, when implementing the bare-bones protocol, the prover will send the verifier the values of $\tilde{\text{add}}$, and $\tilde{\text{mult}}$, at the points the verifier needs. The prover can do this, since these points are specified by the verifier’s public coins in previous rounds. Of course, a dishonest prover may lie, so we run a separate protocol for verifying the correctness of the $\tilde{\text{add}}$, and $\tilde{\text{mult}}$, computations.

By Claim 3.4.6, since $C$ is a $\log(S(n))$-space uniform circuit, $\tilde{\text{add}}$, and $\tilde{\text{mult}}$, can be computed in $O(\log(S(n)))$ space. In turn, by Theorem 3.4.4, there exists an interactive proof for verifying the correctness of each bit of $\tilde{\text{add}}$,'s and $\tilde{\text{mult}}$,'s outputs (the output is a $\log(\|\mathbb{F}\|)$-bit string representing an element in $\mathbb{F}$). For each verifier query, we repeat this interactive proof protocol $O(\log(d(n)) + \log(\log(\|\mathbb{F}\|)))$ times for each bit of the output, to get soundness $\frac{1}{200d(n)}$ for the entire ($\log(\|\mathbb{F}\|)$-bit) answer. In all these invocations, by Theorem 3.4.4, the total prover running time is $\text{poly}(S(n))$, the verifier running time is $d(n) \cdot \text{polylog}(S(n))$ (recall that the input to $\tilde{\text{add}}$, and $\tilde{\text{mult}}$, is only of size $O(\log(S(n)))$), and the verifier uses $O(\log(S(n)))$ total space. The probability that prover cheating in any one of the $O(d(n))$ invocations goes undetected, is (by a Union bound) at most $\frac{1}{100}$.

So, in summary, we run the bare-bones protocol, replacing oracle calls to $\mathbb{F}$ for computing \{\text{add}, \text{mult}\} with an interactive sub-protocol where the prover sends the verifier the value of $\tilde{\text{add}}$, or $\tilde{\text{mult}}$, and then proves its correctness. From Theorem 3.3.1 we get that the protocol has perfect completeness, and the total probability of a cheating prover not being detected is (by a Union bound) $\frac{1}{100} + \frac{1}{100} < \frac{1}{50}$.

The prover’s work is $\text{poly}(S(n))$. The verifier’s work is only $\text{polylog}(S(n)) \cdot (n + d(n))$,
and its space usage is $O(\log(S(n)))$. The protocol is public-coin, and its communication complexity is $d(n) \cdot \text{polylog}(S(n))$.

As an immediate consequence of Theorem 1.1.1 we obtain two main corollaries. The first, stated as Corollary 3.1.1 in Section 3.1.1, gives interactive proofs for languages that are computable by $\mathcal{L}$-uniform $\mathcal{NC}$ circuit families (circuits families of poly-size and polylog-depth). The second corollary, stated as Corollary 3.1.3 in Section 3.1.3, gives a public-coin interactive proofs with a log-space verifier for every language in $\mathcal{P}$. This second corollary is also immediate, using the well known fact that languages in $\mathcal{P}$ have $\mathcal{L}$-uniform poly-size circuits.

Finally, Theorem 1.1.1 also yields a new corollary for interactive proofs for languages computed by uniform Turing Machines (rather than uniform circuits). Using the fact that a language that can be computed by a Turing machine in time $t(n)$ and space $s(n)$ can also be computed by a $O(s(n))$-space uniform circuit of size $\text{poly}(t(n) \cdot 2^{s(n)})$ and depth $s^2(n)$ we conclude that:

**Corollary 3.4.8.** Let $L$ be a language that can be computed by a Turing Machine in time $t(n)$ and space $s(n)$. $L$ has an interactive proof where:

1. The prover runs in time $\text{poly}(t(n) \cdot 2^{s(n)})$. The verifier runs in time $n \cdot \text{poly}(s(n))$ and space $\text{poly}(s(n))$.

2. The protocol has perfect completeness and soundness $1/2$.

3. The protocol is public-coin, with communication complexity $\text{poly}(s(n))$.

It is instructive to compare Corollary 3.4.8 in terms of the honest prover's running time with the well-known $\text{IP} = \mathcal{PSPACE}$ theorem of [LFKN92, Sha92].

**Theorem 3.4.9 (IP = $\mathcal{PSPACE}$ [LFKN92, Sha92]).** Let $L$ be a language that can be computed by a Turing Machine in time $t(n)$ and space $s(n)$. $L$ has an interactive proof where:
1. The prover runs in time $2^{\text{poly}(s(n), \log(t(n)))}$. The verifier runs in time $n \cdot \text{poly}(s(n))$ and space $\text{poly}(s(n))$.

2. The protocol has perfect completeness and soundness $1/2$.

3. The protocol is public-coin, with communication complexity $\text{poly}(s(n))$.

Thus, using Theorem 1.1.1 (via Corollary 3.4.8), we actually obtain a significant reduction in the running time of the honest prover: from $2^{\text{poly}(s(n), \log(t(n)))}$ to $\text{poly}(t(n) \cdot 2^{s(n)})$. In particular for logarithmic space (and polynomial time) computations, this gap means the difference between efficient and inefficient (quasi-polynomial time) honest provers. A fascinating open questions is obtaining a protocol with an honest prover that runs in time $\text{poly}(t(n))$ and space $\text{poly}(s(n))$ (while maintaining the verifier running time and communication complexity of known protocol).

3.4.3 Protocols for Delegating Non-Uniform Computation

So far we have focused on interactive proofs for delegating uniform computations. In the non-uniform setting we cannot escape having the verifier read the entire circuit, so there is no hope for the verifier’s running time to be smaller than the circuit size. As a result, in the non-uniform setting, we do not require the entire computation of the verifier to be super-efficient. Instead, we separate the verification into an off-line (non-interactive) preprocessing phase, which occurs before the input is even specified, and an on-line interactive proof phase, in which the input is known to both the prover and the verifier. We only require that the verifier be super efficient in the on-line interactive phase. In what follows, let $C$ be a boolean circuit family of size $S(n)$ and depth $d(n)$ on inputs of length $n$.

In the off-line phase, before the input $x$ is specified, the verifier is allowed to run a long ($\text{poly}(|C|)$-time) randomized computation $\text{data} \leftarrow \mathcal{V}_{\text{pre}}(C)$, resulting in an output $\text{data}$, which will be retained in the proceeding on-line interactive phase. The output $\text{data}$ of the verifier’s preprocessing computation should be significantly smaller than $|C|$. In our
construction, \textit{data} will be of size \(\text{poly}(d, \log(S))\) (for circuits of polylog depth this is much less than \(|C|\)).

Next, after the input \(x\) is specified, the prover and verifier run an on-line interactive phase. In this phase, the verifier \(V\) takes as input \(x\) and \textit{data} (but not the circuit \(C\)). The prover \(P\) takes as input the (entire) circuit \(C\) and the input \(x\), and proves to the verifier that \(C(x) = b\) for some value \(b\). It is crucial that the prover does not know \textit{data}. Moreover, \textit{data} is only good for a single invocation of the on-line protocol, and cannot be reused for multiple inputs (intuitively, this is because during the interactive phase, the prover may learn information about \textit{data}). We make the usual completeness and (information-theoretic) soundness requirements. We require that the verifier’s running time \textit{in the on-line phase} is significantly smaller than the size of \(C\), that the prover is efficient, and that the communication be small.

We present such an on-line/off-line protocol for delegating the computation of non-uniform circuits, where the size of \textit{data} is polynomial in the depth of the circuit being delegated (and poly-logarithmic in its size), and the verifier’s running time in the on-line phase is linear in the input length and polynomial in the circuit depth (and poly-logarithmic in its size). This protocol (as the protocols for the uniform case) is an implementation of the bare-bones protocol. The idea is for the verifier to choose its random coins in the pre-processing phase. His oracle queries to \(\mathcal{F} = \{\text{add}, \text{mult}\}\) are uniquely determined by these random coins. The verifier can thus compute the oracle answers in the preprocessing phase.

He will then save these answers, together with the random coins, in the \textit{data} string. A formal Theorem follows:

\textbf{Theorem 3.4.10} (Theorem 3.1.4 of Section 3.1.4, restated). Let \(L\) be a language computable by a (non-uniform) circuit family \(C\) of size \(S(n)\) and depth \(d(n)\). There exists an on-line/off-line interactive proof \((P(C, x), V(x, \text{data}), V_{\text{pre}}(C))\) for \(L\). This protocol has completeness 1, and soundness \(\frac{1}{2}\) (can be made arbitrarily small). The complexity of the protocol is as follows:

1. The (randomized) pre-processing computation \(V_{\text{pre}}(C)\) takes time \(\text{poly}(S(n))\). The output \textit{data} is of length \(|\text{data}| = \text{poly}(d(n), \log(S(n)))\).
2. The prover $P(C, x)$ runs in time $\text{poly}(S(n))$.

3. The on-line verifier $V(x, \text{data})$ runs in time $n \cdot \text{poly}(d(n), \log(S(n)))$ and space $O(\log(S(n)))$.

4. The communication complexity of the (on-line) interactive protocol is $\text{poly}(d(n), \log(S(n)))$.

**Proof of Theorem 3.1.4.** As noted above, the protocol is another implementation of the bare-bones protocol of Theorem 3.3.1. Recall, that in the bare-bones protocol, the running times of the prover and verifier, as well as the communication complexity, are exactly as we want. The only problem is that there the verifier is given oracle access to the functions $\{\text{add, mult}\}$. Here we would like to implement these functions.

To avoid ambiguity, we think of $\{\text{add}, \text{mult}\}$ as the (unique) low-degree extensions of $\{\text{add}, \text{mult}\}$. The prover $P(C, x)$ can implement the bare-bones protocol while simulating these oracles on his own, since he is allowed to run in time $\text{poly}(S)$. The verifier, on the other hand, will use the off-line pre-processing phase to “take care” of computing the values of $\{\text{add}, \text{mult}\}$ that will be needed in the on-line interactive phase. Note that these $O(d)$ oracle queries are a function of the $\text{poly}(d \cdot \log(S))$ public coins chosen by the verifier throughout the bare-bones protocol.

Thus, the pre-processing algorithm $V_{\text{pre}}(C)$, chooses the $\text{poly}(d \cdot \log(S))$ public coins. These immediately specify the verifier’s $O(d)$ queries to the functions $\{\text{add}, \text{mult}\}$. Then, $V_{\text{pre}}(C)$ computes the answers to all these queries. To do this, it computes the truth table (of size $\text{poly}(S)$) of the boolean functions $\text{add}, \text{mult}$, and then computes their low-degree extensions (note that this can be done without knowing the input $x$). By Claim 3.2.2, computing the low-degree extension takes time $\text{poly}(S)$. Finally, $V_{\text{pre}}(C)$ outputs the string $\text{data}$ consisting of the $\text{poly}(d \cdot \log(S))$ random coins it chose, as well as the $O(d)$ oracle function values (each of size $\text{poly}(d, \log(S)))$. The total length of the $\text{data}$ string is thus $\text{poly}(d, \log(S))$. In the on-line interactive phase, the verifier $V(x, \text{data})$ simulates the verifier of the bare-bones protocol, while using the random coins and the oracle answers, as specified in $\text{data}$.

126
The completeness, soundness and complexity properties of the interactive phase are inherited directly from the bare-bones protocol for delegating computation (Theorem 3.3.1).

As a final note, observe that indeed once the data string is used in an interactive protocol, the prover knows the random coins chosen in the pre-processing phase, and thus if the same data string is used again (with this prover), even for a different input, the protocol is no longer sound.

3.5 Low Communication Zero-Knowledge Interactive Proofs

In this section, we construct succinct zero-knowledge proofs for many \( \mathcal{NP} \) languages: In particular, the communication complexity of these proofs is quasi-linear in the witness size for any language whose \( \mathcal{NP} \) relation is computable by a \( \mathcal{NC} \) circuit. We consider both the \((L)\)-uniform setting, and the non-uniform setting.

In the non-uniform setting, we use the bare-bones protocol (described in Subsection 3.3.2) to show that (based on the existence of one-way functions) every \( \mathcal{NP} \) language \( L \), verifiable by a depth \( d \) Boolean circuit, has a zero knowledge proof with communication complexity \( k \cdot \text{poly}(d, \kappa) \), where \( k \) is the witness size, and \( \kappa \) is the security parameter. Note that the communication complexity may be significantly smaller than the instance size.

In the uniform setting, we show that every \( \mathcal{NP} \) language \( L \), verifiable by a log-space uniform Boolean circuit of depth \( d \), has a zero knowledge proof with communication complexity as above, and moreover, the runtime of the verifier is very efficient: it is only linear in the input size, and polynomial in the witness size, the circuit depth and the security parameter.

**Notations.** In what follows let \( L = \{ x : \exists w \text{ s.t. } R_L(x, w) = 1 \} \) be an \( \mathcal{NP} \) language. We think of the relation \( R_L \) as a Boolean circuit (rather than a function). We denote by \( d \) the depth of \( R_L \), by \( n = |x| \) the instance size, by \( k = |w| \) the witness size, and by \( \kappa \) the security parameter. The reader should think of \( k, d, \kappa \) as functions of \( n \).
We start by recalling formally our two theorems, starting with the theorem for the non-uniform setting.

**Theorem 3.5.1** (Theorem 3.1.5 of Section 3.1.5, restated). Assume one-way functions exist, and let \( \kappa = \kappa(n) \geq \log(n) \) be a security parameter. Let \( L \) be an \( \mathcal{NP} \) language whose relation \( R \) can be computed on inputs of length \( n \) with witnesses of length \( k = k(n) \) by Boolean circuits of size \( \text{poly}(n) \) and depth \( d(n) \). Then \( L \) has a zero-knowledge interactive proof

1. The prover runs in time \( \text{poly}(n) \) (given a witness), the verifier runs in time \( \text{poly}(n) \) and space \( O(\log(n)) \).

2. The protocol has perfect completeness and soundness \( 1/2 \).

3. The protocol is public-coin, with communication complexity \( k \cdot \text{poly}(\kappa, d(n)) \).

The theorem for the uniform setting is similar, with the additional property that the verifier is very efficient.

**Theorem 3.5.2** (Theorem 3.1.6 of Section 3.1.5, restated). Assume one-way functions exist, and let \( \kappa = \kappa(n) \geq \log(n) \) be a security parameter. Let \( L \) be an \( \mathcal{NP} \) language whose relation \( R \) can be computed on inputs of length \( n \) with witnesses of length \( k = k(n) \) by a \( \mathcal{L} \)-uniform family of boolean circuits of size \( \text{poly}(n) \) and depth \( d(n) \). Then \( L \) has a zero-knowledge interactive proof

1. The prover runs in time \( \text{poly}(n) \) (given a witness), the verifier runs in time \( n \cdot \text{poly}(k, \kappa, d) \) and space \( O(\log(n)) \).

2. The protocol has perfect completeness and soundness \( 1/2 \).

3. The protocol is public-coin, with communication complexity \( k \cdot \text{poly}(\kappa, d(n)) \).
Proof idea of Theorems 3.1.5 and 3.1.6. The idea is to use the bare-bones protocol of Theorem 3.3.1, together with the (by now) standard transformation of [BGG+88], that converts public-coin interactive proofs into zero-knowledge ones. More specifically, we first consider the (not zero-knowledge) interactive proof, where the prover first sends the verifier the witness \( w \), and then they both run the bare-bones protocol. This interactive proof is public-coin, and has the desired complexity parameters.

Next we use the transformation of [BGG+88], to convert this protocol into a zero-knowledge one: The prover does not send his messages in the clear, but instead commits to them. The prover then proves using a (standard) zero-knowledge proof (e.g. that of [GMW91], though we will use a more efficient proof), that the underlying verifier would have accepted this transcript. It may seem that we are right back where we started, as we need again to prove a statement in zero-knowledge. The point (and the reason we make progress) is that the bare-bones protocol guarantees that this final statement involves only a very small verifier computation, and thus this final zero-knowledge proof is very efficient with low communication complexity.

However, recall that in the bare-bones protocol, the verifier gets access to oracle functions specifying the circuit. So, in order to use the bare-bones protocol, we need to implement these oracles. In the non-uniform case we implement these oracles (as in Theorem 3.1.4) by having the verifier compute the oracle answers by himself (an efficient computation). In the uniform case (as in Theorem 1.1.1), we solve this by having the prover give these oracle answers to the verifier, and prove that he computed these values correctly. Note that this computation is polynomial time, so this does not violate zero-knowledge. We proceed with formal proofs.

Proof of Theorem 3.1.5: Fix an \( \mathcal{NP} \) language \( L \), as above. Fix a security parameter \( \kappa = \kappa(n) \leq n \), and assume the existence of a (one-way) function \( f : \{0,1\}^\kappa \to \{0,1\}^\kappa \). This implies that there exists a statistically binding and computationally hiding bit commitment scheme, with sender work, receiver work, and communication that are all poly(\( \kappa \)) [Nao89, HILL99] (see Goldreich’s book [Gol01] for the definition of a commitment scheme and for
the proof method).

Our zero-knowledge interactive proof \((P_{ZK}(x,w), V_{ZK}(x))\) for \(L\), makes use of the bare-bones protocol (presented in Section 3.3) to prove that \(R_L(x,w) = 1\), while assuming that the bare-bones verifier, instead of taking the pair \((x,w)\) as input, has oracle access to the low degree extension of \((x,w)\), denoted by \(\tilde{G} : \mathbb{F}^{m'} \rightarrow \mathbb{F}\). It was shown in Theorem 3.3.1, that in this case the runtime of the verifier is \(\leq \text{poly}(k,d)\). In conclusion, in the bare-bones protocol, both the prover and the verifier have oracle access to a set of functions \(\mathcal{F}\); and the verifier, in addition, has oracle access to \(\tilde{G}\). We denote this bare-bones protocol by

\[
(P_{1}^{\mathcal{F}}(x,w), V_{1}^{\mathcal{F},\tilde{G}}).
\]

According to the notation in Section 3.3,

\[
\mathcal{F} = \{\tilde{\text{add}}_i, \tilde{\text{mult}}_i\}_{i \in [d]},
\]

where \(\tilde{\text{add}}_i, \tilde{\text{mult}}_i\) are some (low degree) extensions of \(\text{add}_i, \text{mult}_i\), respectively. We take \(\tilde{\text{add}}_i\) and \(\tilde{\text{mult}}_i\) to be the unique low-degree extensions of \(\text{add}_i\) and \(\text{mult}_i\), respectively, so as to ensure that they are uniquely defined (and can be computed in polynomial time). See Section 3.3 for the details.

The protocol \((P_{ZK}(x,w), V_{ZK}(x))\) proceeds as follows:

1. The prover \(P_{ZK}(x,w)\) first sends the verifier a bit-by-bit commitment to \(w\).

2. The prover \(P_{ZK}(x,w)\) and verifier \(V_{ZK}(x)\) run the bare-bones protocol \((P_{1}, V_{1})\) (see Theorem 3.3.1) with the following difference: The prover \(P_{ZK}\), rather than sending his messages “in the clear”, will send commitments to all his messages. Namely, if the bare-bones protocol consists of a transcript of the form:

\[(r_1, m_1, r_2, m_2, \ldots, r_{\ell}, m_{\ell}),\]

then in the zero-knowledge protocol \((P_{ZK}(x,w), V_{ZK}(x))\), this transcript will be con-
verted to a transcript of the form:

\[(r_1, \text{com}(m_1), r_2, \text{com}(m_2), \ldots, r_\ell, \text{com}(m_\ell)).\]

The prover \(P_{\text{ZK}}(x, w)\) emulates the prover \(\mathcal{P}_1\) of the bare-bones protocol (simulating the oracle calls to \(\mathcal{F}\) on his own). Recall that this can be done in time \(\text{poly}(n)\). The verifier \(V_{\text{ZK}}(x)\) emulates the verifier \(\mathcal{V}_1\). Recall that the bare-bones protocol is public-coin, and so the verifier \(V_{\text{ZK}}(x)\) does not need to “know” the messages \(m_1, \ldots, m_\ell\), nor does he need to use the oracle \(\mathcal{F}\) (or the oracle \(\tilde{G}\) to a low-degree extension of the input), in order to generate \(r_1, \ldots, r_\ell\).

3. By Theorem 3.3.1, the verifier \(\mathcal{V}_{1,\mathcal{G}}\) queries the oracle \(\mathcal{F}\) at \(O(d)\) points, determined uniquely by the verifier’s randomness \(r_1, \ldots, r_\ell\). Moreover, these points, as well as the oracle’s answers, can be computed in time \(\text{poly}(n)\) given the verifier’s randomness.

Both the prover \(P_{\text{ZK}}(x, w)\) and the verifier \(V_{\text{ZK}}(x)\) compute these oracle queries, and simulate the oracle’s answer on each of these queries. We denote the answers by \(v_1, \ldots, v_{O(d)} \in \mathbb{F}\).

4. Also, according to Theorem 3.3.1, the verifier \(\mathcal{V}_{1,\mathcal{G}}\) queries his oracle \(\tilde{G}\) (i.e., the low-degree extension of the input) at a single point. This point depends only on his randomness \(r_1, \ldots, r_\ell\) (and can be computed in polynomial time given the verifier’s randomness).

Both the prover \(P_{\text{ZK}}(x, w)\) and the verifier \(V_{\text{ZK}}(x)\) compute this oracle query, denoted by \(z \in \mathbb{F}^m\). This can be done in time \(\text{poly}(n)\).

5. According to Proposition 3.2.1,

\[\tilde{G}(z) = \sum_{p \in \mathbb{F}^m} \tilde{\beta}(z, p) \cdot \tilde{G}(p).\]

Moreover, \(\tilde{\beta}\) can be evaluated in time \(\text{poly}(d, \log n)\) (with respect to the parameters
chosen by the bare-bones protocol). Denote by \( p_1, \ldots, p_n \in \mathbb{H}^m \) the \( n \) points that satisfy \( \tilde{G}(p_i) = x_i \), where \( x = (x_1, \ldots, x_n) \). Denote by \( p_{n+1}, \ldots, p_{n+k} \in \mathbb{H}^m \) the \( k \) points that satisfy \( \tilde{G}(p_{n+i}) = w_i \), where \( w = (w_1, \ldots, w_k) \).

Both the prover \( P_{ZK}(x, w) \) and the verifier \( V_{ZK}(x) \) compute the value \( t \triangleq \sum_{i=1}^{n} \tilde{\beta}(z, p_i) \cdot \tilde{G}(p_i) \). This can be done in time \( \text{poly}(n) \).

6. Next, the prover and verifier run a previously known (but communication-efficient) zero knowledge proof, say that of [CD97] or [IKOS07]. The statement being proved is that:

\[
(com(w), r_1, com(m_1), \ldots, r_\ell, com(m_\ell), v_1, \ldots, v_{O(d)}, t) \in L',
\]

where the language \( L' \) is defined as follows. Equation (3.7) holds if the verifier \( V_{1}^{F, \tilde{G}} \), with randomness \( r_1, \ldots, r_\ell \), accepts the transcript

\[
(r_1, m_1, \ldots, r_\ell, m_\ell),
\]

assuming that \( v_1, \ldots, v_{O(d)} \) are the answers obtained by the oracle \( F \), and

\[
t + \sum_{i=1}^{k} \tilde{\beta}(z, p_{n+i}) \cdot w_i,
\]

is the answer obtained by the oracle \( \tilde{G} \), where \( z \) is the point that \( V_{1}^{F, \tilde{G}} \) queries \( \tilde{G} \).

We use the zero knowledge proof of [CD97] (or, alternatively, an even further optimized construction of [IKOS07]). The properties we use are that the proof has perfect completeness, soundness \( 1/3 \) and communication that is linear in the size of the verifying circuit (and polynomial in the security parameter \( \kappa \)). In our case the circuit size is \( k \cdot \text{poly}(\kappa, d) \) (see below), and so the communication complexity is also \( k \cdot \text{poly}(\kappa, d) \).

7. The verifier \( V_{ZK}(x) \) accepts if and only if he accepts this zero-knowledge proof.

We next show that this protocol is zero-knowledge, has perfect completeness, soundness \( 1/2 \), and communication complexity \( k \cdot \text{poly}(\kappa, d) \), as desired.
The fact that this protocol is zero-knowledge follows from the fact that the underlying commitment scheme is computationally hiding and from the fact that the underlying zero-knowledge proof is indeed zero-knowledge (see [BGG+88] for details). Perfect completeness follows from the fact that the underlying zero-knowledge proof used has perfect completeness. The fact that the soundness is $1/2$ follows (by a union bound) from the soundness of the bare-bones protocol, from the fact that the underlying zero-knowledge proof has soundness at most $1/3$, and from the fact that the commitment scheme is statistically binding. It remains to argue that the communication complexity is $k \cdot \text{poly}(\kappa, d)$.

To prove that the communication complexity of $(P_{ZK}(x, w), V_{ZK}(x))$ is $k \cdot \text{poly}(\kappa, d)$, it suffices to prove that $L'$ is an $NP$ language with a verification circuit of size $k \cdot \text{poly}(\kappa, d)$. To this end, we consider the witness consisting of all the de-commitment values, and show that it can be verified in time $k \cdot \text{poly}(\kappa, d)$.

Given these de-commitment values, the value of $w$ and $m_1, \ldots, m_t$ can be computed in time $k \cdot \text{poly}(\kappa, d)$. Moreover, given:

$$(w, r_1, m_1, \ldots, r_t, m_{t,1}, v_1, \ldots, v_{O(d)}, t),$$

checking whether $V'_{1,F,G}$ accepts the transcript $(r_1, m_1, \ldots, r_t, m_t)$, assuming that the oracle answers of $F$ are $v_1 \ldots, v_{O(d)}$, and given the value $G(z)$, can be done in time $\text{poly}(d, \log n)$. Finally, given $w$ and $t$ (the part of the low-degree extension of $(x, w)$ at point $z$ that depends on $x$), the value of the oracle $G$ at point $z$ can be computed in time $k \cdot \text{poly}(d, \log n)$. So the total size of the verification circuit is $k \cdot \text{poly}(d, \kappa)$. ■

Proof of Theorem 3.1.6: The proof of Theorem 3.1.6 is almost identical to the proof (and protocol) of Theorem 3.1.5. The only differences are that now the circuit computing $R_L$ is uniform, and we want to leverage this fact to reduce the verifier’s running time. The running time of the verifier presented in the above proof of Theorem 3.1.5 is as required, except for its computation of the $O(d)$ values of the functions $\tilde{add}_i, \tilde{mult}_i$, which takes polynomial time in the circuit size.
However, recalling Claim 3.4.6, if the circuit computing $\mathcal{R}_L$ is itself $\mathcal{L}$-uniform, then $\tilde{\text{add}}, \tilde{\text{mult}}$, can be computed in $O(\log(n))$-space (the inputs to these functions are themselves of size $O(\log(n))$). We used this fact in Theorem 1.1.1 of Section 3.4 to show that the prover can simply send to the verifier the values of $\tilde{\text{add}}, \tilde{\text{mult}}$, at the points that the verifier needs. The prover then proves that it sent the correct values using the protocol of Theorem 3.4.4 (with soundness $O(1/d)$). The verifier’s running time to verify the values of $\tilde{\text{add}}, \tilde{\text{mult}}$, on the desired points is only polylog($n$).

We modify the above protocol of Theorem 3.1.5 in a similar manner. Instead of computing the values $v_1, \ldots, v_{O(d)}$ of $\tilde{\text{add}}, \tilde{\text{mult}}$, on its own, the verifier asks the prover to send him these values and prove that they were computed correctly. This is done after running the bare-bones protocol (i.e. after the prover has already committed to all its messages). Note that the function value being proven here is efficiently computable, and so zero-knowledge is not violated (the simulator can run this computation itself). The above protocol has all the properties of the protocol in Theorem 3.1.5 (in particular soundness is maintained), and also the verifier’s work is $n \cdot \text{poly}(k, d, \kappa)$. We note that by using (in the last step of the protocol) a zero-knowledge proof with a verifier whose running time is linear in the size of the verification circuit, we can get the verifier’s running time down to $(n + k) \cdot \text{poly}(d, \kappa)$, the details are omitted. ■

3.6 One-Round Arguments for Delegating Computation

In this section we are concerned with the question of reducing the amount of interaction in protocols for delegating computation. As discussed in Section 3.1.2, we seek to construct one round protocols, where a verifier can issue a challenge to a polynomial-time prover, and get back a (computationally sound) certificate of correctness for a claim made by the prover. In this setting, the verifier with a Turing machine $M$ computing a language $L$ wants to verify that $x \in L$, but without taking the time to run the Turing machine $M$ on the
input $x$ (the case of verifying that $x \notin L$ or certifying computation of functions rather than languages can be done similarly). Towards this end, the verifier wants to send $M$ and $x$ to an un-trusted prover, who will then provide a short (non-interactive) computationally sound certificate that $x \in L$. We only allow the verifier to send the prover (together with $x$), a single challenge message $m_V$, that may help guarantee the soundness of the certificate. The certificate is thus a function of the machine $M$, the input $x$, and the verifier’s challenge $m_V$. Ideally, the challenge $m_V$ is independent of the input $x$ and the language being proved, in which case the verifier can compute the challenge $m_V$ in advance and, say, publish it on his webpage (this will be the case in the scheme we present below).

Following the exposition above, we view a system for certifying computations as a 1-round computationally sound argument system for a language $L$. The verifier and prover know a machine $M$ computing $L$ and an input $x$. The verifier sends a challenge message $m_V$, the prover replies with a certificate, and the verifier accepts or rejects. Completeness is the guarantee that if $x \in L$, the verifier should accept when it interacts with the (honest) prover. Soundness is the guarantee that if $x \notin L$, no efficient prover can make the verifier accept. The main complexity measures we are interested in bounding are the running time of the verifier and the prover (as a function of the complexity of computing $L$) and the length of the certificate and the challenge.

Our main result in this section is a system for certifying computation (a one-round argument system) for a language computed by a family of ($L$-uniform) $\mathcal{NC}$ circuits. The prover’s running time is polynomial in the circuit size. The verifier’s running time is quasi-linear. The lengths of the certificate and the verifier’s challenge are poly-logarithmic (using a poly-logarithmic security parameter).

This result uses our interactive proof for delegating computation (Theorem 1.1.1), together with a recent result of Kalai and Raz [KR09] on transforming interactive proof systems into a one-round (two-message) computationally sound argument systems. The soundness of the certificate relies on the privacy of a (computational) PIR scheme with poly-logarithmic communication (see Section 3.2.4 for a definition of PIR schemes and more details).
We first re-state the result we use from [KR09], and then present our main theorem about certifying efficient computations.

**Theorem 3.6.1.** [KR09] Let $\kappa \geq \log n$ be a security parameter. Assume the existence of a secure PIR scheme (as defined in Definition 3.2.5), with communication $\text{poly}(\kappa)$, receiver work $\text{poly}(\kappa)$, and sender work $\text{poly}(n, \kappa)$ (where $n$ is the database size).

Assume that there exists an interactive proof system $(P, V)$ for proving membership in some language $L$, with the following properties:

1. Completeness $c$, soundness $s$ and communication complexity $\ell$.
2. Verifier running time $t_V$ and prover running time $t_P$.
3. Each message sent by the prover depends only on the $\lambda$ previous bits sent by $V$.
4. The verifier’s messages depend only on the verifier’s random coin tosses (and are independent of the interaction and the input).

Then there exists a one-round (two-message) argument system $(P’, V’)$ for $L$, with communication complexity $\ell’ = \text{poly}(\ell, \kappa)$, completeness $c’ \geq c - 2^{-n^2}$, and soundness $s’ \leq s + 2^{-\kappa}$ against provers of size $\leq 2^\kappa$. The verifier $V’$ runs in time $\leq t_V \cdot \text{poly}(\kappa)$. The prover $P’$ runs in time $\leq \text{poly}(t_P, \kappa, 2^\kappa)$.

Moreover, the resulting one-round argument system $(P’, V’)$ has the property that the first message, sent by $V’$, depends only on the random coin tosses of $V’$, and is independent of the instance $x$ or of the language being proven.

Applying the transformation of the above theorem to our efficient interactive proofs from Theorem 1.1.1, we directly obtain efficient one-round arguments for delegating computation:

**Theorem 3.6.2** (Theorem 3.1.2 of Section 3.1.2, restated). Let $L$ be a language computable by a family of $O(\log(S(n)))$-space uniform boolean circuits of size $S(n)$ and depth $d(n)$. Let $\kappa \geq \log(S(n))$ be a security parameter. Assume the existence of a secure PIR scheme, with communication $\text{poly}(\kappa)$, receiver work $\text{poly}(\kappa)$, and sender work $\text{poly}(n, \kappa)$ (where $n$ is the
The language $L$ has a 1-round (private coin) argument system with the following properties:

1. The prover runs in time $\text{poly}(S(n))$, the verifier runs in time $n \cdot \text{poly}(\kappa, d(n), \log(S(n)))$.\textsuperscript{27}

2. The protocol has perfect completeness and computational soundness $1/2$ (can be made arbitrarily small): for any input $x \notin L$ and for any cheating prover running in time $\leq 2^m$, the probability that the verifier accepts is $\leq 1/2$.

3. The sizes of the certificate (the prover’s message) and the verifier’s challenge are $\text{poly}(\kappa, d(n))$. The verifier’s message depends only on the parameters $n$ and $\kappa$, and is independent of the language $L$ and the input $x$.

We conclude with a few Remarks:

1. Since the verifier’s challenge depends only on the parameters (and is independent of the computation being certified), the verifier can prepare the challenge in advance, before he knows the language or input whose membership is proved. We note, however, that a fresh challenge must be used for every invocation of the argument system, otherwise soundness might break down.

2. The protocol is private coins. Moreover, a certificate (which is verifier dependent) cannot be verified without the verifier’s (private) random coins. This means that our certificates cannot be used to convince anyone that an input is in the language, except the verifier, who knows his private coins, and knows that they were generated randomly.

3. It is instructive to compare this result to two previous works on providing certificates for efficient computation (i.e. for languages in $\mathcal{P}$). The results of [BFLS91] give long certificates, of size polynomial in the circuit size (even for languages in $\mathcal{NC}$). Though these results are efficiently probabilistically checkable. The result of Micali [Mic94] on CS Proofs, requires the use of a random oracle, a primitive whose realization is

\textsuperscript{27}Moreover, if the verifier is given oracle access to the low-degree extension of its input, then its running time is only $\text{poly}(\kappa, d(n), \log(S(n)))$.}
by now notoriously questionable (though he obtains short certificates for any efficient computation, not only for $\mathcal{NC}$).

### 3.7 An Interactive PCP

In this section, we use the bare-bones protocol (described in Subsection 3.3.2) to construct an interactive PCP scheme, as introduced in [KR08]. An interactive PCP (say for membership of an input $x$ in a language $L$) is a combination of a PCP and a short interactive proof. Roughly speaking, an interactive PCP is a proof that can be verified by reading only a small number of its bits, with the help of a short interactive proof. We begin in Subsection 3.7.1 with a brief introduction to interactive PCPs, and then present our new construction in Subsection 3.7.2.

#### 3.7.1 Preliminaries

More precisely, let $L = \{x : \exists w \text{ s.t. } (x, w) \in R_L\}$ be an $\mathcal{NP}$ language, described by a polynomial-time computable relation $R_L$. Let $p, q, \ell, c, s$ be parameters as follows: $p, q, \ell$ are integers and $c, s$ are reals, s.t. $0 < s < c < 1$ (informally, $p$ is the size of the PCP string, $q$ is the number of queries allowed to the PCP string, $\ell$ is the communication complexity of the interactive proof, $c$ is the completeness parameter and $s$ is the soundness parameter). The reader should think of the parameters $p, q, \ell, c, s$ as functions of the instance size $n$. An interactive PCP with parameters $(p, q, \ell, c, s)$ for membership in $L$ is an interactive protocol between an (efficient) prover $P$ and an (efficient) verifier $V$.\footnote{One could also consider a model with a prover that is not necessarily efficient. Originally in [KR08] interactive PCPs were defined with efficient provers, and we will also focus on efficient provers throughout this work.} We assume that both the prover and the verifier know the language $L$ and get as input an instance $x$ of size $n$. The prover gets an additional input $w$ (supposedly a witness for the membership $x \in L$). In the first round of the protocol, the prover generates a (PCP) string $\pi$ of $p$ bits (think of $\pi$ as an encoding of the witness $w$). The verifier is still not allowed to access $\pi$. The prover and
the verifier then apply an interactive protocol, where the total number of bits communicated is \( \ell \). During the protocol, the verifier is allowed to access at most \( q \) bits of the string \( \pi \). After the interaction, the verifier decides whether to accept or reject the statement \( x \in L \).

**Definition 3.7.1.** [KR08] A pair \((P, V)\) of probabilistic polynomial time interactive Turing machines is an interactive PCP for \( L \) with parameters \((p, q, \ell, c, s)\), if for every \((x, w) \in \mathcal{R}_L\) the prover \(P(x, w)\) generates a bit string \( \pi \) (known as the PCP string) of size at most \( p(n) \) (where \( n = |x| \)), such that the following properties are satisfied.

- **Completeness:** For every \((x, w) \in \mathcal{R}_L\),

\[
\Pr[(P(x, w), V^\pi(x)) = 1] \geq c(n)
\]

(where \( n = |x| \), and the probability is over the random coin tosses of \( P \) and \( V \)).

- **Soundness:** For every \( x \notin L \), every (unbounded) interactive Turing machine \( \tilde{P} \), and every string \( \tilde{\pi} \in \{0, 1\}^* \),

\[
\Pr[(\tilde{P}(x), V^{\tilde{\pi}}(x)) = 1] \leq s(n)
\]

(where \( n = |x| \), and the probability is over the random coin tosses of \( V \)).

- **Complexity:** The communication complexity of the protocol \((P(x, w), V^\pi(x))\) is at most \( \ell(n) \), and \( V \) reads at most \( q(n) \) bits of \( \pi \).

Let \( L = \{ x : \exists w \ s.t. \ (x, w) \in \mathcal{R}_L \} \) be any \( \mathcal{NP} \) language. It was shown in [KR08], that if \( \mathcal{R}_L \) can be computed by a constant depth Boolean circuit (over the basis \( \land, \lor, \neg, \oplus \)) then \( L \) has an interactive PCP with the following parameters: the length of the PCP string is polynomial in the witness size (i.e., \( p = \text{poly}(|w|) \)), it makes only a single query to the PCP oracle (i.e., \( q = 1 \)), and it has poly-logarithmic communication complexity (i.e., \( \ell = \text{polylog}(|x|) \)).
3.7.2 New Improved Interactive PCPs

We extend the results of [KR8]. We show that for every $\mathcal{NP}$ language $L = \{x : \exists w \text{ s.t. } (x, w) \in R_L\}$, if the relation $R_L$ can be computed by a polynomial size circuit of depth $d$, then $L$ has an interactive PCP with the following parameters: the length of the PCP is polynomial in $d$ and the witness size (i.e., $p = \text{poly}(d, |w|)$), it makes only a single query to the PCP oracle (i.e., $q = 1$), and it has communication complexity $\ell = \text{poly}(d, \log |x|)$. In particular, we match the parameters of [KR8] (up to polynomial factors) for any $R_L$ that can be computed in $\mathcal{NC}$ (poly-logarithmic depth). Moreover, our interactive PCP has the additional property that each message sent by the prover, during the interactive phase, depends only on $O(\log |x|)$ bits sent by the verifier (and on the input and the randomness of the prover). This property (which previous interactive PCP’s do not have) will be used in Section 3.8 to construct “short” efficient probabilistically checkable arguments.

**Theorem 3.7.2.** Let $C : \{0, 1\}^k \rightarrow \{0, 1\}$ be a Boolean circuit of size $S$ and depth $d$. Then, for any $\varepsilon \geq 1/S$,$^{29}$ the satisfiability of $C$ can be proven by an interactive PCP with the following parameters: $p = \text{poly}(k, d, \log S, \frac{1}{\varepsilon})$, $q = 1$, $\ell = \text{poly}(d, \log S, \frac{1}{\varepsilon})$, $c = 1$ and $s \leq \frac{1}{2} + O(\varepsilon)$.

Moreover, the interactive PCP has the following two properties:

1. The PCP string $\pi$ (generated by the prover in the first round of the protocol) depends only on the witness $w \in \{0, 1\}^k$ and the parameters $S, d, \varepsilon$, and not on the circuit $C$.

2. The interactive phase is public coin, and each message sent by the prover depends only on the preceding $O(\log S)$ bits sent by the verifier.

The following is an immediate corollary of Theorem 3.7.2:

**Corollary 3.7.3.** Let $L = \{x : \exists w \text{ s.t. } R_L(x, w) = 1\}$ be any $\mathcal{NP}$ language. Let $n = |x|$ denote the instance size, let $k = |w|$ denote the witness size, and let $d$ denote the circuit size.

---

$^{29}$We require $\varepsilon > 1/S$ in order to ensure that the prover runs in time $\text{poly}(|C|)$. We could take $0 < \varepsilon < 1/S$ and then the prover’s running time is polynomial in $1/\varepsilon$.  

140
depth of $R_L$. Then, for any $\varepsilon > 1/n$,\(^{30}\)

$$L \in \text{IPCP}(p, q, \ell, c, s),$$

with $p = \text{poly}(k, d, \frac{1}{\varepsilon})$, $q = 1$, $\ell = \text{poly}(d, \log n, \frac{1}{\varepsilon})$, $c = 1$, and $s \leq \frac{1}{2} + O(\varepsilon)$. Moreover, this interactive PCP has the two additional properties stated in Theorem 3.7.2.

**Remark.** Notice that if we allow “many” queries to the PCP string then we can reduce the soundness to be any parameter $s$, as follows: First omit the parameter $\varepsilon$ in Theorem 3.7.2 by setting the soundness parameter to be a constant, and then improve the soundness parameter via parallel repetition. This will increase the query complexity to $O(\log \frac{1}{\varepsilon})$ and will increase the communication complexity $\ell$ by a factor of $O(\log \frac{1}{s})$.

Rather than proving Theorem 3.7.2 directly, as was done in [KR08], we prove a weaker version (stated in Theorem 3.7.4), that allows “many” queries to the PCP string. In [KR08] it was shown how to convert an interactive PCP with many queries into interactive PCP with one query (and in particular, how to get Theorem 3.7.2 from Theorem 3.7.4). We note that this weaker version (Theorem 3.7.4) is interesting on its own, and in particular it is this weaker version that we use in order to construct “short” efficient probabilistically checkable arguments in Section 3.8.

**Theorem 3.7.4.** Let $C : \{0, 1\}^k \rightarrow \{0, 1\}$ be a Boolean circuit of size $S$ and depth $d$. Then, for any soundness parameter $s > 2^{-S}$,\(^{31}\) the satisfiability of $C$ can be proven by an interactive PCP with the following parameters: $p = \text{poly}(k, d, \log S)$, $q = \text{poly}(\log d, \log \log S, \log \frac{1}{s})$, $\ell = \text{poly}(d, \log S, \log \frac{1}{s})$, completeness $c = 1$, and soundness $s$. Moreover, the interactive PCP has the following two properties:

1. The PCP string $\pi$ (generated by the prover in the first round of the protocol) depends

---

\(^{30}\)Again, we require $\varepsilon > 1/n$ in order to ensure that the prover runs in polynomial time. We could take $0 < \varepsilon < 1/n$ and then the prover’s running time is polynomial in $1/\varepsilon$.

\(^{31}\)We require $s > 2^{-S}$ in order to ensure that the prover and verifier run in time $\text{poly}(|C|)$. We could take $0 < s < 2^{-S}$ and then the running time would be polynomial in $\log \frac{1}{s}$.
only on the witness \( w \in \{0,1\}^k \) and the parameters \( S, d \), and not on the circuit \( C \).

2. The interactive phase is public coin, and each message sent by the prover depends only on the preceding \( O(\log S) \) bits sent by the verifier.

We now proceed with a proof of Theorem 3.7.4, starting with a high-level overview. Suppose we want an interactive PCP for proving that there exists a string \( w \in \{0,1\}^k \) such that \( C(w) = 0 \), where \( C : \{0,1\}^k \rightarrow \{0,1\} \) is a circuit as in the theorem statement.

**Proof Outline.** Roughly speaking, the interactive PCP consists of the following steps:

1. The PCP string \( \pi \) is simply the low-degree extension of the witness \( w \).
2. Verify that \( \pi \) is close to a low-degree polynomial, by running a low degree test of [MR08] (described for completeness in Subsection 3.2.2).
3. Verify that the string \( w \), encoded in the oracle \( \pi \), satisfies \( w \in \{0,1\}^k \).
4. Verify that the string \( w \), encoded in the oracle \( \pi \), satisfies \( C(w) = 0 \).

We use our delegation protocol to execute Steps (3) and (4).

**Comparison with the scheme of [KR08].** The interactive PCP of [KR08] also follows steps (1)-(4) as above. The main difference between our protocol and the one in [KR08] is in the execution of Steps (3) and (4): More specifically, we reduce the task of verifying that \( w \in \{0,1\}^k \) to the task of verifying that \( g(w) = 0 \), where \( g \) is some arithmetic circuit of size \( \text{poly}(k) \) and depth \( \text{polylog}(k) \). Then we prove that \( g(w) = 0 \) and that \( C(w) = 0 \) using our delegation protocol.

On the other hand, in [KR08], they first use a linear error-correcting-code to reduce the task of verifying that \( w \in \{0,1\}^k \) to the task of verifying that \( g(w) = 0 \), where \( g \) is some arithmetic formula of size \( \text{poly}(k) \), constant depth, and constant degree. Then they use a method due to Razborov and Smolenski to convert (the constant depth Boolean circuit) \( C \).
into an arithmetic formula $f$ of degree $d = \text{polylog}(k)$. Finally, they use an “efficient sum-check protocol” for proving that $g(w) = 0$ and for proving that $f(w) = 0$. We note that the communication complexity of their sum-check protocol depends polynomially on the degree $d$, whereas in our delegation protocol the communication complexity depends polynomially on the depth $d$. This is why we can get an interactive PCP for all of $NC$ (with polylog($k$) communication complexity) while [KR08] cannot go beyond $AC^0$.

Proof of Theorem 3.7.4. In what follows, we prove Theorem 3.7.4 with soundness parameter $s = \frac{11}{12}$. This suffices since for any $s > 0$, by repeating the interactive phase $O(\log \frac{1}{s})$ times we get an interactive PCP with the desired parameters and soundness $s$. We assume $k \geq \log S$. This is without loss of generality since we could always increase $k$ to be $\log S$ by adding dummy variables. Note that this does not change the guarantees in the statement of Theorem 3.7.4. Consider the following interactive PCP protocol $(P, V)$ for proving the satisfiability of $C : \{0, 1\}^k \rightarrow \{0, 1\}$.

Parameters:

1. $k, S, d$, where

   $$k, d < S \leq 2^k.$$

2. Parameters $\mathbb{H}, \mathbb{F}, m, m'$, that together with $k, S, d$, are valid parameters for the bare-bones protocol given is Section 3.3.2. In particular, $\mathbb{H}$ is an extension field of $\text{GF}[2]$, $\mathbb{F}$ is an extension field of $\mathbb{H}$, and $m, m'$ are integers, such that

   $$d \leq |\mathbb{H}| \leq \text{poly}(d, \log S),$$

   $$S \leq |\mathbb{H}|^m \leq \text{poly}(S),$$

   $$k \leq |\mathbb{H}|^{m'} \leq \text{poly}(d, k),$$

   143
and

$$|F| \leq \text{poly}(|\mathbb{H}|).$$

Moreover, the parameters $\mathbb{H}, F, m, m'$ should satisfy the following additional properties:

(a) $m' \geq 3$.
(b) $|F| \geq m'(|\mathbb{H}| - 1)$.
(c) $2^{10} m' \sqrt{\frac{(m')^2 (|\mathbb{H}| - 1)}{|F|}} \leq \frac{1}{12}$.

These properties guarantee that we can apply Lemma 3.2.3 with respect to $F, m'$, and $d = m'(|\mathbb{H}| - 1)$, and get $\varepsilon \leq \frac{1}{12}$.

**Input:**

Both the prover and the verifier take as input a Boolean circuit $C : \{0, 1\}^k \rightarrow \{0, 1\}$ of size $S$ and depth $d$. The prover takes an additional input

$$w = (w_0, w_1, \ldots, w_{k-1}) \in \{0, 1\}^k,$$

such that $C(w) = 0$.

**The protocol** $(P(C, w), V^*(C))$.

1. **Computing the PCP string $\pi$.**

The PCP string $\pi$ is the low-degree-extension of $w$ w.r.t. the parameters $\mathbb{H}, F, m'$. Namely,

$$\pi \overset{\text{def}}{=} \text{LDE}_{\mathbb{H}, F, m'}(w_0, w_1, \ldots, w_{k-1}).$$

The verifier is given oracle access to $\pi$. Note that $\pi : \mathbb{F}^{m'} \rightarrow \mathbb{F}$ is a multivariate polynomial of degree $|\mathbb{H}| - 1$ in each variable, and thus is of total degree $\leq m' \cdot (|\mathbb{H}| - 1)$. 

144
2. Running the low degree test on \( \pi \).

The verifier \( V \) checks that \( \pi \) is close to an \( m' \)-variate polynomial \( f : F^{m'} \rightarrow F \) that has total degree \( \leq m'(|H| - 1) \). This is done by running the low degree test \((P_{\text{LDT}}(\pi), V_{\text{LDT}}^{\pi})\) described in Subsection 3.2.2. If the test fails then the verifier rejects.

Note that so far the protocol depends only on \( w \) and on the parameters \( S, d \), and does not depend on the circuit \( C \).

3. Proving that \( C(w) = 0 \).

Interpret \( C \) as a layered arithmetic circuit (of fan-in 2 over \( F \)). Let \( \mathcal{F} = \{\text{add}_i, \text{mult}_i\}_{i \in [d]} \) be a set of functions corresponding to \( C \) (as defined in Subsection 3.3.1), such that for every \( i \in [d] \), \( \text{add}_i \) is the unique low degree extension of \( \text{add}_i \), and \( \text{mult}_i \) is the unique low degree extension of \( \text{mult}_i \). Note that both the prover and the verifier of the interactive PCP protocol can compute the functions in \( \mathcal{F} \) on their own (in time \( \text{poly}(S) \)).

The prover and the verifier run the bare-bones protocol \((P_{\text{IF}}^{\mathcal{F}}(w), V_{\text{IF}}^{\mathcal{F}}(w))\) described in Section 3.3.2 for proving that \( C(w) = 0 \) (with respect to \( \delta = |H| - 1 \)). The prover \( P(C, w) \) (of the interactive PCP system) emulates \( P_{\text{IF}}^{\mathcal{F}}(w) \) by computing the functions in \( \mathcal{F} \) on his own (and thus simulating the oracle \( \mathcal{F} \)). The verifier \( V(C) \) (of the interactive PCP system) emulates \( V_{\text{IF}}^{\mathcal{F}}(w) \) by computing the functions in \( \mathcal{F} \) on his own, and using his oracle \( \pi \) instead of \( w \).

Recall that according to the third (additional) property of Theorem 3.3.1, the verifier \( V_1 \) can run the bare-bones protocol, even if he is not given \( w \) as input, but is only given oracle access to the low degree extension of \( w \) (with respect to \( H, F, m' \)). In this case, \( V_1 \) queries the low degree extension of \( w \) at a single random point corresponding to a field element, or alternatively, at \( O(\log d + \log \log S) \) bit points (since each element in \( F \) can be represented by \( O(\log d + \log \log S) \) bits). Since with high probability (assuming the low degree test passes), the oracle \( \pi \) of the interactive PCP is close to the low degree extension of \( w \) (with respect to \( H, F, m' \)), the verifier can use the oracle \( \pi \) of the
interactive PCP as an oracle to the low degree extension of $w$.

If the verifier $V^x_f(w)$ of the bare-bones protocol rejects then the verifier $V^x(C)$ of the interactive PCP protocol also rejects.

4. Restricting all satisfying assignments to bit strings.

In order to ensure soundness, the verifier should verify that $w \in \{0, 1\}^k$. To this end, consider the function $\Psi : \mathbb{F}^k \rightarrow \mathbb{F}$, defined as follows:

$$\Psi(t_1, \ldots, t_k) \overset{\text{def}}{=} \prod_{\beta \in \mathbb{F}\backslash\{0\}} \left( \beta - \prod_{i=1}^{k} \left( \prod_{\gamma \in \mathbb{F}\backslash\{0, 1\}} (t_i - \gamma) \right) \right).$$

Note that $\Psi(t_1, \ldots, t_k) = 0$ if and only if $t_1, \ldots, t_k \in \{0, 1\}$. Moreover, $\Psi$ can be implemented by a layered arithmetic circuit of fan-in 2 (over $\mathbb{F}$), of size $\text{poly}(k, |\mathbb{F}|) \leq \text{poly}(k, d)$ and of depth $\leq \text{poly}(\log |\mathbb{F}|, \log k) \leq \text{poly}(\log k, \log d)$. For the simplicity of the analysis (and without loss of generality), we assume that $\Psi$ is of size $\leq S$.

The prover will prove that $\Psi(w) = 0$, as was done in Step 3.

Analysis of the protocol $(P(C, w), V^x(C))$. The fact that $\pi = \text{LDE}_{\mathbb{F}, m'}(w)$ implies that the PCP string is of size $p = |\mathbb{F}|^{m'} \cdot (\log |\mathbb{F}|) \leq \text{poly}(k, d)$. Theorem 3.3.1 and Lemma 3.2.3 imply that the protocol $(P(C, w), V^x(C))$ has communication complexity $\ell = \text{poly}(d, \log S)$ and completeness $c = 1$. Note that our protocol queries $\pi$ at three points (each corresponding to a field element): Once during the run of the low degree test (in Step 2), and once during each run of the delegation protocol (in Step 3 and Step 4). Thus, the query complexity is $q = O(\log |\mathbb{F}|) = O(\log d, \log \log S)$. We next prove that its soundness is $s \leq \frac{11}{12}$.

Fix any $C : \{0, 1\}^k \rightarrow \{0, 1\}$, of size $S$ and depth $d$, which is not satisfiable. Fix any unbounded (cheating) prover $\tilde{P}$, and any function $\tilde{x} : \mathbb{F}^{m'} \rightarrow \mathbb{F}$. Let $E$ denote the event that $\tilde{P}, V^x(C) = 1$, and let $s \overset{\text{def}}{=} \Pr[E]$. Assume for the sake of contradiction that $s > \frac{11}{12}$. According to Lemma 3.2.3, there exists
an \( m' \)-variate polynomial \( f : \mathbb{F}^{m'} \to \mathbb{F} \) of degree \( \leq m' \cdot (|\mathbb{H}| - 1) \) such that
\[
\Pr_{z \in \mathbb{F}^{m'}}[\tilde{\pi}(z) = f(z)] \geq s - \varepsilon,
\]
where \( \varepsilon \) is defined in Lemma 3.2.3. Let
\[
\gamma \overset{\text{def}}{=} \Pr_{z \in \mathbb{F}^{m'}}[\tilde{\pi}(z) \neq f(z)] \leq 1 - (s - \varepsilon).
\]
Our contradiction assumption (that \( s > \frac{11}{12} \)), together with our assumption that \( \varepsilon \leq \frac{1}{12} \), implies that
\[
\gamma \leq \frac{1}{6}. \tag{3.8}
\]
Define \( (\bar{w}_0, \bar{w}_1, \ldots, \bar{w}_{k-1}) \in \mathbb{F}^k \) by
\[
\bar{w}_i \overset{\text{def}}{=} f(\alpha^{-1}(i)),
\]
where \( \alpha : \mathbb{H}^{m'} \to \{0, 1, \ldots, k' - 1\} \) \( (k' \overset{\text{def}}{=} |\mathbb{H}|^{m'}) \) is the lexicographic order of \( \mathbb{H}^{m'} \).

Recall that both times when emulating the verifier \( V_1 \) of the bare-bones protocol, the verifier \( V \) queries the oracle at a single random point (corresponding to a field element). Let \( B \) denote the event that on these two points \( \tilde{\pi} \) is consistent with \( f \). Note that
\[
\Pr[\neg B] \leq 2\gamma \leq \frac{1}{3}. \tag{3.9}
\]
Let \( A \) denote the event that \( \bar{w}_0, \bar{w}_1, \ldots, \bar{w}_{k-1} \in \{0, 1\} \). Theorem 3.3.1 implies that
\[
\Pr[E|A \wedge B] \leq \frac{1}{100},
\]
and
\[
\Pr[E|\neg A \wedge B] \leq \frac{1}{100}.
\]

147
using some basic facts from probability theory, we conclude that

\[ s = \Pr[E] \leq \Pr[E|B] + \Pr[\neg B] \leq \]

\[ \Pr[E|A \land B] + \Pr[E|\neg A \land B] + \Pr[\neg B] \leq \frac{1}{100} + \frac{1}{100} + \frac{1}{3} < \frac{5}{12} , \]

contradicting our assumption that \( s > \frac{11}{12} \).

It remains to show that the two additional properties, required by the statement in Theorem 3.7.2, are attained.

1. The fact that \( \pi \) depends only on the witness \( w = (w_0, w_1, \ldots, w_{k-1}) \) and on the parameters \( S \) and \( d \), follows immediately from the definition of \( \pi \overset{\text{def}}{=} \text{LDE}_{H,F,m'}(w_0, w_1, \ldots, w_{k-1}) \), and from the fact that the parameters \( H,F,m' \) depend only on the parameters \( S, k, d \).

2. Recall that the interactive phase consists of a low degree test, and two runs of the bare-bones protocol, one with circuit \( C \) and one with circuit \( \Psi \).

The fact that the interactive phase is public-coin follows from the fact that both the low degree test, and the bare-bones protocol, are public coin. The fact that each message sent by the prover depends only on the preceding \( O(\log S) \) bits sent by the verifier, follows from the fact that in the low degree test the verifier sends a total of at most \( O(\log S) \) bits, and in the bare-bones protocol each message of the prover depends only on the preceding \( O(\log S) \) bits sent by the verifier.

\[ \square \]

### 3.8 A Probabilistically Checkable Argument

In this section, we give an efficient and short probabilistically checkable argument (PCA) system for many \( \mathcal{NP} \) languages. To this end, we use our interactive PCP system described
in Section 3.7, together with a general method given in [KR09], for converting interactive PCP systems into PCA systems.

A probabilistically checkable argument (PCA), a notion introduced in [KR09], is a relaxation of the notion of probabilistically checkable proof (PCP). It is defined analogously to PCP, with two differences: (1) the verifier first specifies a challenge to the prover, and the proof (PCA) is tailored to this verifier challenge. The soundness property is required to hold only computationally, i.e. against bounded malicious provers. Other than these differences, the setting is the same as that of PCPs: after specifying the challenge and receiving the proof, the probabilistic polynomial time verifier only reads a few bits of the proof string in order to verify. A PCA is said to be efficient if the honest prover, given a witness, runs in time poly(n).

More specifically, each PCA system is associated with three algorithms: a challenge generation algorithm $G$, a proof generation algorithm $P$, and a verification algorithm $V$. It is also associated with five parameters $t, p, q, c, s$, where $t, p, q$ are integers and $c, s$ are reals, s.t. $0 < s < c < 1$. (Informally, $t$ is the security parameter, $p$ is the size of the PCA, $q$ is the number of queries allowed to the PCA, $c$ is the completeness parameter and $s$ is the soundness parameter). We think of the parameters $t, p, q, c, s$ as functions of the instance size $n$.

Let $L$ be an $NP$ language, defined by $L = \{x : \exists w \text{ s.t. } (x, w) \in R_L\}$. Suppose that Alice wishes to prove to Bob that $x \in L$. Assume that Bob applied in the past the challenge generation algorithm $G$, and thus is associated with a pair of secret key and public challenge $(SK, PK) \leftarrow G(1^t)$. Bob’s public challenge, $PK$, is sent to Alice. We assume that both Alice and Bob know $L$ and that both get as input an instance $x$ of size $n$. Alice gets an additional input $w$ (supposedly a witness for the membership of $x \in L$). A PCA system allows Alice to generate a string $\pi \leftarrow P(x, w, PK)$ of $p$ bits. Bob is allowed to access at most $q$ bits of the string $\pi$, and based on these bits he decides whether to accept or reject the statement $x \in L$.

**Definition 3.8.1.** [KR09] A triplet $(G, P, V)$ of probabilistic Turing machines is a PCA.
system for $L$ with parameters $(t, p, q, c, s)$, if the following holds:

- $G$ is a probabilistic Turing machine that runs in time $\text{poly}(t)$, and $V$ is a probabilistic oracle machine that runs in time $\text{poly}(t, n)$.

- For every $(x, w) \in \mathcal{R}_L$ (where $|x| = n$) and every $(SK, PK) \leftarrow G(1^{t(n)})$, the algorithm $P(x, w, PK)$ generates a bit string $\pi$ of size at most $p(n)$, and the oracle machine $V^\pi(x, SK, PK)$ reads at most $q(n)$ bits of $\pi$.

- **Completeness:** For every $(x, w) \in \mathcal{R}_L$ (where $|x| = n$),

  $$\Pr[V^\pi(x, SK, PK) = 1] \geq c(n)$$

  (where the probability is over $(SK, PK) \leftarrow G(1^{t(n)})$, over $\pi \leftarrow P(x, w, PK)$, and over the randomness of $V$).

- **Soundness:** For every $x \notin L$ (where $|x| = n$), and every cheating prover $\tilde{P}$ of size $\leq 2^t(n)$,

  $$\Pr[V^{\tilde{\pi}}(x, SK, PK) = 1] \leq s(n)$$

  (where $\tilde{\pi} = \tilde{P}(PK)$, and the probability is over $(SK, PK) \leftarrow G(1^{t(n)})$ and over the randomness of $V$).

**Remark.** Note that in Definition 3.8.1 we did not specify the complexity of $P$. We say that a PCA system $(G, P, V)$ is **efficient** if $P$ runs in time $\text{poly}(t, n)$.

It was shown in [KR09], that any interactive PCP system (with certain properties) can be transformed into a PCA system.

**Theorem 3.8.2.** [KR09] Assume the existence of a (uniform) poly-logarithmic PIR scheme.$^{32}$ Assume that there exists an interactive PCP system $(P, V)$ with parameters $(p, q, t, c, s)$ for some $NP$ language $L$, with the following properties:

---

$^{32}$The definition of a (uniform) poly-logarithmic PIR scheme can be found in Section 3.2.4.
1. For every input $x \in L$, every auxiliary input $w \in \{0,1\}^*$, and for every $i \in [\ell]$, the message sent by the (honest) prover $P(x,w)$ in the $i$'th round of the protocol $(P(x,w), V(x))$ depends only on the message sent by $V$ in the $i$'th round of the protocol (and on $x,w$ and the random coin tosses of $P$), and does not depend on the messages sent by $V$ before the $i$'th round.

2. Each message sent by the verifier in this phase depends only on the verifier’s random coin tosses (and is independent of the interaction, the PCP string $\pi$, and the input $x$), and can be computed in time $\leq \text{poly}(\ell)$.

Then, for any security parameter $t \geq \max\{\ell, \log n\}$ there exists a PCA system $(G', P', V')$ with parameters $(t, p', q', c', s')$ for the language $L$, where $p' = \text{poly}(p, t)$, $q' = \text{poly}(q, t)$, $c' \geq c - 2^{-t^2}$, and $s' \leq s + 2^{-t^2}$. The prover $P'$ runs in time $\leq \text{poly}(t, n, 2^A)$, where $A$ is the length of the longest message sent from $V$ to $P$ in the interactive phase of the interactive PCP system $(P,V)$.

Applying Theorem 3.8.2 to our interactive PCP system, results with a PCA system that is both efficient and short for many N\text{P} languages, as stated in the following theorem.

**Theorem 3.8.3.** Assume the existence of a (uniform) poly-logarithmic PIR scheme. Fix any N\text{P} language $L = \{x : \exists w \text{ s.t. } (x,w) \in R_L\}$. Let $n = |x|$ denote the instance size, let $k = |w|$ denote the witness size, and let $d$ denote the depth of $R_L$. Then for any soundness parameter $s > 2^{-n}$ and for any security parameter $t \geq \text{poly}(d, \log n, \log \frac{1}{s})$ there exists a PCA system for $L$ with parameters $(t, p', q', c', s')$, where $p' = \text{poly}(k, t)$, $q' \leq \text{poly}(t)$, $c' \geq 1 - 2^{-t^2}$, and $s' \leq s + 2^{-t^2}$. Moreover, the prover of this PCA system runs in time $\leq \text{poly}(n, t)$.

**Remark.** Applying Theorem 3.8.2 to the interactive PCP systems given in [KR08], results with inefficient PCA systems; i.e., with PCA systems where the prover runs in super

---

33 As is common, we allow the prover in the interactive proof system to use an auxiliary input, supposedly a witness for $x \in L$.

34 We think of each round as consisting of a message sent by the verifier $V$ followed by a message sent by the prover $P$. 

151
polynomial time. This follows from the fact that in these systems, the length of the longest
message sent from the verifier to the prover is of size $\text{polylog}(n)$. 
Chapter 4

Verifying Interactive Proofs in Constant Depth

In this section we explore the power of interactive proofs with constant depth verifiers, see Section 1.2 for an overview.

Organization. In Section 4.1 we give some general preliminaries and specific background about interactive proofs. Section 4.2 shows how to transform general proof systems to ones with verifiers in $\mathcal{NC}^0$. In Section 4.3 we present our negative results.

4.1 Preliminaries and main tools

4.1.1 Interactive Proofs: New Definitions

We proceed with new definitions and terminology that that will be useful for us later. We first formally define the notion of $\mathcal{NC}^0$ (or constant parallel time) verifiers.

Definition 4.1.1. We say that round $i$ of a proof system (with one or more provers) can be computed in $\mathcal{NC}^0$ if the computation of the verifier in this round can be performed by an $\mathcal{NC}^0$ circuit (that may depend on the round $i$) that is given the input $x$ to the protocol, the randomness $r$ and the partial transcript from the previous $i - 1$ rounds.
We assume that the circuit may depend on the number of the round because $NC^0$ circuits cannot even increment an integer by 1.\footnote{We could consider a model with the same $NC^0$ verifier in all rounds. The models are equivalent for protocols with $O(1)$ communication rounds. For other protocols results carry through, except that we can only bound the expected number of communication rounds when interacting with a malicious prover.}

**Definition 4.1.2.** We say that a language can be verified (interactively) in constant parallel time, if it has an interactive proof system (with one or more provers) with a constant number of rounds, and the entire verifier’s strategy can be implemented in $NC^0$, i.e. every round of the interaction stage as well as the decision stage.

We define a special type of proof systems in which “most” of the verifier’s computation is pushed to the decision stage, keeping the computation during the interaction stage extremely efficient.

**Definition 4.1.3.** We say that a proof system (with one or more provers) is *simple* if in every round of the interaction stage the verifier’s computation can be performed by an $NC^0$ circuit.

A special case of simple proof systems is public-coin proof systems.

**Definition 4.1.4.** An interactive proof system is called *public-coin*, if at each round of the interaction stage the verifier simply tosses new random coins and sends them to the prover (while preforming no other computation).

**Theorem 4.1.5.** ([GS86]) For every $k > 0$, every language in IP($k$) has a public-coin interactive proof system with $k + 2$ rounds.

### 4.2 Verification in Constant Parallel Time

We start by showing how to transform any simple proof system into one in which the entire strategy of the verifier (interaction and decision) is in $NC^0$. This is the Result of Lemma 1.2.1 in Section 1.2. Here we give the proof intuition and then a proof.
Proof Intuition. We first notice that any simple proof system can be transformed into a simple proof system (with one more round) in which the decision stage can be implemented in \( \mathcal{AC}^0 \) (this essentially boils down to evaluating a CNF formula via the Cook-Levin reduction) and hence also in \( \mathcal{NC}^1 \). We now want to simulate this \( \mathcal{NC}^1 \) computation by an \( \mathcal{NC}^0 \) verifier with the help of the prover. Let \( L \) be the \( \mathcal{NC}^1 \)-complete language given by Lemma 2.3.16 and let \( I \) be its randomized image. The verifier creates a uniformly distributed \( \ell \)-tuple of instances \( c_1, \ldots, c_\ell \) where for each \( i \), \( c_i \) is either: (with probability 1/2) a solved instance for which the verifier knows \( I(c_i) \) (this can be done by using the solved instance generator for \( I \), see Definition 2.3.1), or (with probability 1/2) a randomized instance such that given \( I(c_i) \) the verifier can compute the output of the \( \mathcal{NC}^1 \) computation of the original verifier (this can be done using the random instance reduction from \( L \) to \( I \) and the fact that \( L \) is \( \mathcal{NC}^1 \)-complete).

The verifier asks the prover to provide the values \( I(c_1), \ldots, I(c_\ell) \). It then checks for every solved instance \( c_i \) (for which it knows the value \( I(c_i) \)) whether the prover answered correctly. The verifier also checks that for the randomized instances the answers that the prover gave all evaluate (via the evaluator) to the same answer for the original instance. If both of these checks pass for all the \( c_i \)'s, it is a good indication that the prover also provided the correct answer for all the randomized instances. Here we use the fact that the prover cannot distinguish between the two types of instances since they are all uniformly distributed. The verifier can then extract the value of the \( \mathcal{NC}^1 \) computation from any correct answer to a randomized instance, and use it as its output.

Full Proof of Lemma 1.2.1. We first consider the case of a single prover, i.e. \( p = 1 \). We proceed by showing that any simple proof system can be transformed into a simple proof system (with one more round) in which the decision stage can be implemented in \( \mathcal{AC}^0 \) (this essentially boils down to evaluating a CNF formula via the Cook-Levin reduction) and hence also in \( \mathcal{NC}^1 \). Then we add one more round to enable the verification to be performed in \( \mathcal{NC}^0 \). Details follow.

Let \( V' \) be the verifier and \( P' \) the honest prover in the original (simple) protocol and
let $V, P$ be these entities in the new protocol. By the definition of a simple proof system (Definition 4.1.3), the computations of $V'$ during the interaction stage are in $\mathcal{NC}^0$. Thus, $V$ and $P$ run the first $k$ rounds as in the original protocol, and we then add two rounds as follows.

**Round $k + 1$** In the original protocol, given the input $x$ (of length $n$), the transcript $t$ and the verifier’s random coins $r$, $V'$ can decide in polynomial time whether to accept or reject. Let $|(x, t, r)| = m(n) = \text{poly}(n)$. This round is as follows: $V$ sends all its random coins to $P$ and $P$ sends back the tableau of the computation $V'(x, t, r)$.

If $V$ were an $\mathcal{AC}^0$ circuit, it could at this stage verify that the tableau is correct and deduce the output of $V'$ (accept/reject). This is because checking the validity of the tableau amounts to verifying that $x, t, r$ is written in the first row, and that all the local transitions are legal. However $V$ is not an $\mathcal{AC}^0$ circuit. So we proceed to the next round.

**Round $k + 2$** Consider the language $L$ associated with the above $\mathcal{AC}^0$ computation. That is, an instance contains a tableau $T$ and $x, t, r$ as above, and it belongs to $L$ if the tableau $T$ is consistent with the computation $V'(x, t, r)$, and if this computation accepts. In particular $L \in \mathcal{NC}^1$ and therefore, by Lemma 2.3.16, it has a randomized image $I$.

Let $\ell$ be an integer that will be determined later. Given an instance $a = (T, x, t, r)$, $V$ does the following: for each $i \in [\ell]$ (in parallel and with independent random coins), choose uniformly $v_i \in_R \{0, 1\}$. If $v_i = 0$, then $V$ runs the $\mathcal{NC}^0$ solved instance generator for $I$ on input length $m'(n)$, to obtain a pair $(c_i, y_i)$. If $v_i = 1$, then $V$ runs on $a$ the $\mathcal{NC}^0$ random instance reduction from $L$ to $I$, to obtain a pair $(c_i, \tau_i)$. Here $|c_i| = m'(n) = \text{poly}(m(n)) = \text{poly}(n)$. $V$ then sends to $P$ the message $(c_1, \ldots, c_\ell)$, and $P$ sends back answers $(b_1, \ldots, b_\ell) \in \{0, 1\}^\ell$.

**Decision** $V$ accepts if and only if the following holds:

1. For every $i \in [\ell]$ for which $v_i = 0$, $y_i = b_i$.  

156
2. For every $i \in [\ell]$ for which $v_i = 1$, $\mathcal{E}(b_i, \tau_i) = 1$ (recall that $\mathcal{E}$ is the evaluator in the random instance reduction from $L$ to $I$).

**Correctness** Clearly, if $\ell$ is a constant (independent of $n$) the entire strategy of $V$ can be implemented in $\mathcal{NC}^0$. We proceed to prove completeness and soundness.

**Claim 4.2.1.** The protocol has completeness $c$.

**Proof.** The honest prover $P$ plays the first $k$ rounds like the honest prover $P'$ of the original protocol. It then sends the correct tableau, and the correct values $b_1, \ldots, b_\ell$, which are the membership values (0/1) of the instances $c_1, \ldots, c_\ell$ in $I$. By the definition of solved instance generator, this implies that with probability 1, the verifier passes the first test. By the definition of random instance reduction, for every $i \in [\ell]$ for which $v_i = 1$, $\mathcal{E}(b_i, \tau_i) = 1$ if and only if $(T, x, t, r) \in L$. This happens exactly when the original verifier $V'$ accepts the original protocol, and the probability for that is at least $c$. ■

**Claim 4.2.2.** The protocol has soundness $s + 2^{-\ell}$.

**Proof.** Let $x$ be an instance not in the language. Consider the event:

$$A: \mathcal{E}(b, \tau) = 1 \text{ for every } i \in [\ell] \text{ for which } v_i = 1$$

By the soundness of the original protocol and the definition of random instance reduction, $\Pr[A] \leq s$. If event $A$ does not occur, the only way that a cheating prover, $P^*$, can convince $V$ to accept is by cheating on $c_i$ for every $i$ for which $v_i = 1$. If the prover cheats on $c_j$ where $v_j = 0$, then by the definition of solved instance generator, $V$ will reject with probability 1. In other words, in order to cheat and not get caught, $P^*$ must cheat on every $i$ for which $v_i = 1$ and give the correct answer on every $i$ for which $v_i = 0$. By the definitions of solved instance generator and random instance reduction, the $v_i$'s are independent of $(c_1, \ldots, c_\ell)$. Thus $P^*$ has to guess exactly the value of $\ell$ independent unbiased coin tosses which he can do with probability at most $2^{-\ell}$. We conclude that the probability that $P^*$ can convince $V$ to accept is bounded by $s + 2^{-\ell}$. ■
Let $\delta > 0$ be an arbitrarily small constant. By setting $\ell = \log(1/\delta)$ we conclude the proof for single-prover systems. For multi-provers, the same arguments apply where the last two rounds $(k + 1, k + 2)$ are played only with the first prover $P_1$. That is, in round $k + 1$, $V$ sends to $P_1$ its random coins as well as the transcripts of messages exchanged with all the other provers, then $P_1$ and $V$ proceed as above.

**Remark 4.2.3.** In the proof above, the “hardest” computation that the verifier is performing is an AND of fan-in $\log(1/\delta)$. In terms of parallel computing time this amounts to $\log \log(1/\delta)$. Generalizing the argument to non-constant $\delta$ we can obtain proof systems with negligible soundness (e.g. $n^{-\log n}$) with a verifier that runs in $O(\log \log n)$ parallel time.

**Remark 4.2.4.** Vadhan [Vad06] has suggested an alternative implementation of round $k + 2$: the prover wants to convince $V'$ that $V(x, t, r) = 1$. Let $b = V(x, t, r)$, and for $c \in \{0, 1\}$ denote $I_c = \{z : I(z) = c\}$. The verifier generates an instance $y$ that is uniformly distributed in $I_b$ restricted to the relevant input length. It also generates an instance $y'$ that is uniformly distributed in $I_0$ restricted to the same input length. The ability to sample such $y, y'$ follows directly from the techniques used to prove the results in Section 2.3. The verifier then chooses at random one of $y$ and $y'$ and the prover has to say whether it is from $I_1$ or $I_0$. Note that this is very similar to the protocol for Graph Non-Isomorphism [GMW91].

### 4.2.1 General Proof Systems

Next we want to use Lemma 1.2.1 to obtain our results about general proof systems. For proof systems with a single prover, we can first apply the transformation of Goldwasser and Sipser [GS86] to obtain a public-coin protocol (which is clearly also a simple protocol). Then, by applying Lemma 1.2.1 to the resulting protocol we obtain a general transformation from any interactive single-prover proof system to a one in which the verifier is in $\mathcal{NC}^0$ with an addition of $O(1)$ rounds. In particular, we obtain the following theorem and corollary:

**Theorem 4.2.5.** Every language in $\mathcal{NP}(k)$ has an interactive protocol with $k + 4$ rounds, and soundness $\delta$, where $\delta > 0$ is an arbitrarily small constant and the verifier is in $\mathcal{NC}^0$. 

158
Proof. Let \( L \in \mathcal{IP}(k) \). By Theorem 4.1.5, \( L \) has a public-coin proof system with \( k + 2 \) rounds. We can amplify the completeness and soundness of the protocol to be \( 1 - 2^{-n} \) and \( 2^n \) respectively, while still having \( k + 2 \) rounds. By definition, a public-coin proof system is simple, so we can apply Theorem 1.2.1 to obtain a proof system with an \( \mathcal{NC}^0 \) verifier and the stated parameters. ■

As a consequence, we obtain the characterization of Corollary 1.2.2 (see Section 1.2).

Next we want to apply similar arguments to obtain \( \mathcal{NC}^0 \) verifiers for multi-prover proof systems. Unfortunately, the concept of public-coins does not exist in this context, as clearly the queries to the different provers depend on a common random string that the verifier must hide from the provers. Thus we have to take a different approach.

Feige and Lovasz [FL92] showed how to transform any one-round proof system with many provers (and exponentially small soundness) to a one-round proof system with two provers (and exponentially small soundness). We show that a modification of their ideas can transform any one-round, two-prover protocol to a three-round, two-prover simple protocol (both having exponentially small soundness).

**Lemma 4.2.6.** *Every language in \( \mathcal{MIP}_{c,2^{-n}}(2,1) \), has a simple multi-prover proof system with two provers, three rounds of interaction, completeness \( c \) and soundness \( 2^{-n} \).*

Proof. We use a modification of the transformation of [FL92]. Let \( L \in \mathcal{MIP}_{c,2^{-n}}(2,1) \). Let \( V', P_1', P_2' \) be the players in the original protocol and \( V, P_1, P_2 \) be the players in the new protocol. In the new protocol there are three threads of exchanges that take place at the same time. For the convenience of the reader, we will state for each message which thread it belongs to.

In the new protocol, \( V \) plays with \( P_1 \) the original protocol, where \( P_1 \) plays the role of \( P_1' \) and \( P_2' \). Exchanges that are part of the simulation of the original protocol belong to thread I. Clearly, there is no reason that this protocol will be sound since \( P_1 \) can correlate the answers of the two provers. We therefore need to somehow force \( P_1 \) to run independent strategies and for that we will need \( P_2 \). That is, we will run certain correlation tests between \( P_1 \) and \( P_2 \). Roughly speaking, the idea is that \( P_1 \) will be so occupied with passing these tests, that it
will be forced to play independent strategies for $P_1'$ and $P_2'$. Exchanges that are part of these tests belong to thread II. In addition, we will use $P_1$ to help $V$ perform its computations in $\mathcal{NC}^0$ during the interaction stage. Exchanges that help achieve this goal belong to thread III. The last thread helps $V$ perform the computations for the other threads in $\mathcal{NC}^0$, thus typically messages in this thread also have a role in either thread I or II.

Assume w.l.o.g. that every message exchanged in the original protocol is of the same length $\ell = \ell(n) \geq n$ (where $n$ is the input length). Let $F$ be a field of characteristic 2 of size at least $2^{9\ell}$ (This size of the field is as required by [FL92]). We can assume that $V$ and the provers hold the same representation of this field. (One can use a canonic representation as in [HV06] or use $P_1$ to provide such a representation and at the decision stage prove that it is indeed an irreducible polynomial (of the relevant degree) over $GF(2)$.)

Let $g_j : \{0, 1\}^\ell \rightarrow \mathbb{F}$ be a function that represents the optimal strategy for prover $P_j'$ (for $j \in \{1, 2\}$). Let $\hat{g}_j$ be the following unique multi-linear representation of $g_j$ over $F$: for $b \in \{0, 1\}$ and a formal variable $x$ let $s(b, x) = x$ if $b = 1$ and $s(b, x) = 1 - x$ if $b = 0$; then define the $\ell$-variate multi-linear polynomial over $F$:

$$\hat{g}_j(u_1, \ldots, u_\ell) = \sum_{b_1, \ldots, b_\ell \in \{0, 1\}} g_j(b_1, \ldots, b_\ell) \Pi_{i=1}^\ell s(b_i, u_i)$$

We now describe the protocol on input $x \in \{0, 1\}^n$:

**Round 1:**

$V$: Threads I & III - Toss coins $r'$ for $V'$ and sends them to $P_1$.

Thread II - choose uniformly and independently $h_1, h_2 \in R F$ and send them to $P_1$. (The $\mathcal{NC}^0$ verifier can choose random elements in the field since we assume it is of characteristic 2.)

$P_1$: Threads I & III - send $q_1, q_2 \in \{0, 1\}^\ell$ (the queries $V'$ generates on input $x$ and randomness $r'$), and $a_1, a_2 \in \{0, 1\}^\ell$ (the answers that $P_1', P_2'$ provides on input $x$ and queries $q_1, q_2$).

Thread II - send two univariate polynomials over $F$ (represented by the list of coefficients), $f_1, f_2$, each of degree $\ell$ (these are the restrictions of $\hat{g}_1, \hat{g}_2$ to the lines $d_1 = \{q_1 + th_1 : t \in \mathbb{F}\}$ and $d_2 = \{q_2 + th_2 : t \in \mathbb{F}\}$ respectively, where we view $q_1, q_2$ as points in $F^\ell$).
Round 2:
V: Threads I & III - choose uniformly $t_1, t_2 \in_R \mathbb{F}$ and send them to $P_1$ (indices of random points on the lines).

$P_1$: Threads I & III - send $v_1 = d_1(t_1) = q_1 + t_1 h_1$ and $v_2 = d_2(t_2) = q_2 + t_2 h_2$ (the names in $\mathbb{F}^\ell$ of the random points on the lines).

Round 3:
V: Thread II - send $v_1, v_2$ to $P_2$.
$P_2$: Thread II - send $y_1 = \hat{g}_1(v_1), y_2 = \hat{g}_2(v_2)$.

Decision: V checks that the following holds:

1. $q_1, q_2$ are the queries that $V'$ asks $P_1', P_2'$ given input $x$ and randomness $r'$ (threads I & III).

2. $V'$ accepts given $x, r'$ and the transcripts $q_1, a_1, q_2, a_2$ (thread I).

3. $f_1, f_2$ are univariate polynomials of degree $\ell$ over $\mathbb{F}$ (threads II & III).

4. $f_1(0) = q_1$ and $f_2(0) = q_2$ (threads II & III).

5. The values for $v_1, v_2 \in \mathbb{F}^\ell$ that $P_1$ sent are indeed $d_1(t_1) = q_1 + t_1 h_1$ and $d_2(t_2) = q_2 + t_2 h_2$ respectively (threads II & III).

6. $y_1 = f_1(t_1)$ and $y_2 = f_2(t_2)$ (thread II).

Clearly, the proof system is simple. Completeness follows easily from [FL92]. We now explain why it is sound. Note that all the above tests can be performed deterministically and in polynomial time given the view of $V$ (which includes $x$, its randomness and the transcript). Thus, the tests themselves do not induce errors. The only errors can come from errors made by the simulation of the original protocol (thread I), or by the consistency tests between the restriction of the functions to the lines and the random points on the line (i.e. thread II). Thus the analysis reduces (with some modifications, see below) to the analysis of
Feige and Lovasz [FL92], who prove that the new protocol has completeness and soundness as claimed.

There are two points in which our protocol differs from [FL92]. First, $P_1$ receives right at the beginning all the random coins of $V'$. To see why this is not a problem, consider the following mental experiment. Look at the 3-provers protocol, in which $V'$ plays with $P'_1$ and $P'_2$ the original protocol, and with $P'_3$ plays the following protocol: $V'$ sends to $P'_3$ its random coins $r'$, and $P'_3$ has to reply with $q_1, q_2$, which are the queries $V'$ asks $P'_1, P'_2$ respectively on randomness $r'$. $V'$ accepts if the original protocol with $P'_1, P'_2$ accepts and $P'_3$ sends the correct queries. Clearly, this protocol has the same completeness and soundness because $V'$ can check the correctness of the answer sent by $P'_3$. Now the proof of [FL92] actually shows how to transform a three (or in general polynomially many) provers protocol to a two provers protocol, by letting $P_1$ play all the provers and run the line vs. point test separately on the simulation of each prover. The idea is that this test forces $P_1$ to play an independent strategy for each prover in the simulation. This means that the new protocol must be sound because the original protocol is sound (against independent provers strategies). Now in our mental experiment $P'_3$ has only one possible strategy that he can play without being caught. So he is already forced to play this (independent) strategy and we therefore do not need to run the line vs. point test for $P'_3$, resulting in the protocol that we give above.

The second point in which our protocol differs from [FL92] is that $P_1$ who provides the restriction of $g_j$ to the line $d_j$, knows on which (random) point on the line we are going to query $P_2$. However this information is revealed to him only after he commits to some low-degree polynomial. This prevents him from correlating the polynomial with the point on the line. Given this, the analysis of [FL92] goes through.

By Lemmas 1.2.1 and 4.2.6 we obtain,

**Theorem 4.2.7.** Every language in $\text{MIP}(2, 1)$ has a $\text{MIP}(2, 5)$ protocol with an $\text{NC}^0$ verifier and arbitrarily small constant soundness.

Combining this with Theorem 2.2.2 gives us the result of Theorem 1.2.3 (see Section 1.2).
4.3 Negative Results

In this section we prove that the use of private coins in our protocol is inherent. We also show that constant soundness is the best one can hope for in proof systems that have a constant number of rounds and an $\mathcal{NC}^0$ verifier. These statements hold unless the language is already in $\mathcal{NC}^0$. We start with a more refined definition of $\mathcal{NC}^0$:

**Definition 4.3.1.** For $k \in \mathbb{N}$, $\mathcal{NC}^0_k$ is the class of $\mathcal{NC}^0$ circuits in which every output bit depends on at most $k$ input bits. We say that a language belongs to the class $\mathcal{NC}^0_k$ if for every $n \in \mathbb{N}$, there is an $\mathcal{NC}^0_k$ circuit that decides $L^n = L \cap \{0, 1\}^n$.

Note that if a language is in $\mathcal{NC}^0_k$ then its characteristic function (at every input length) is influenced by at most $k$ variables.

**Theorem 4.3.2.** Let $L \subseteq \{0, 1\}^*$ be an arbitrary language, then $L$ does not have a public-coin interactive protocol with an $\mathcal{NC}^0$ verifier, unless $L$ is in $\mathcal{NC}^0$.

**Proof.** Suppose $L$ is not in $\mathcal{NC}^0_k$ for any constant $k$ and yet it has a public-coin protocol with an $\mathcal{NC}^0$ verifier. In particular, this means that the verifier decides whether to accept its input using an $\mathcal{NC}^0$ circuit that runs on its input, randomness and the transcript. The number of input bits that influence the verifier’s decision is constant. Let $k$ be the overall number of input bits that influence the verifier’s decision bit. Let $n$ be an input length for which $L^n$ does not have an $\mathcal{NC}^0_k$ circuit. Let $x_1, x_2 \in \{0, 1\}^n$ be such that the $k$ bits that the verifier reads are the same in $x_1$ and $x_2$, yet $x_1 \in L$ and $x_2 \notin L$. By the fact that $L^n$ does not have an $\mathcal{NC}^0_k$ circuit (and hence its characteristic function is influenced by more than $k$ variables), such a pair of instances exist. Consider the dishonest prover $P^*$, that on input $x_2$, for any verifier randomness, plays the strategy of the honest prover on input $x_1$. Because the protocol is public-coin, the verifier’s view in both cases is exactly the same, i.e. for any verifier randomness, the prover’s messages on inputs $x_1$ and $x_2$ are identical, and thus the bits that the verifier uses to make its decision are also identical. By the protocol’s completeness on $x_1$, the soundness of the protocol on $x_2$ is violated and we get a contradiction.

\[\blacksquare\]
Next we state our negative result regarding sub-constant soundness.

**Theorem 4.3.3.** Let $L \subseteq \{0, 1\}^*$ be an arbitrary language, then $L$ does not have a constant-round interactive protocol with sub-constant soundness and an $\mathcal{NC}^0$ verifier, unless $L$ is in $\mathcal{NC}^0$.

**Proof.** (sketch) Suppose $L$ is not in $\mathcal{NC}^0_k$ for any constant $k$, and yet it has a constant-round interactive protocol with an $\mathcal{NC}^0$ verifier and soundness $s$. We consider two cases. First suppose that $s = 0$. Then the interactive protocol can be transformed into a public-coin protocol as follows: the prover sends a sequence of random coins on which the original verifier accepts, as well as the transcript of the entire original protocol, which deterministically depends on these bits. The verifier then checks that the original verifier accepts the protocol with these “random” bits. By the fact that $s = 0$, the prover cannot send a false proof. Moreover, this proof system is trivially public-coin, and Theorem 4.3.2 says that there is no public-coin proof system with $\mathcal{NC}^0$ verifiers for any language that is not already in $\mathcal{NC}^0_k$ for some constant $k$.

Next, suppose that $s > 0$. Let $x \not\in L$ be an instance that together with a fixed dishonest prover strategy $P^*$ exhibits $s > 0$. Fix some randomness $\bar{r}$ for the verifier that causes it to accept, and define a new static prover $P^{**}$ that sends the same messages as $P^*(x)$ does when interacting with $V(x, \bar{r})$, regardless of the verifier’s actual messages.

Now consider the verifier’s output bit: it depends on only a constant number of bits from the input $x$, from the verifier’s randomness and from the messages sent by $P^{**}$. Note that the decision bit may depend on messages that $V$ itself sends in the protocol, but since $V$ is $\mathcal{NC}^0$ in every round of the protocol, each of the bits sent by $V$ depends in turn only on a constant number of bits from the input, the randomness, and messages sent in earlier rounds. Thus, since there is only a constant number of rounds, the total number of random bits that affect any $V$-message bit that affects the verifier’s decision bit is constant (though we note it is exponential in the number of rounds).

Let $d$ be the number of randomness bits that influence the decision bit, and denote their values in $\bar{r}$ by $r_{i_1}, \ldots, r_{i_d}$. Then for every random string for which the bits at positions
$i_1, \ldots, i_d$ are $r_1, \ldots, r_d$, it holds that the verifier accepts the protocol on $x$. We conclude that the soundness on input $x$ with this fixed strategy $P^{**}$ is at least $2^{-d}$, which is a constant.

**Discussion.** At first glance, it may seem that the proof of Theorem 4.3.2 should also rule out protocols with private coins (at least for constant-round protocols). We want to explain why this is not the case. We believe that this explanation sheds some interesting light on public versus private coins in the context of $NC^0$ verifiers, and specifically on our protocol (from Lemma 1.2.1). The idea in the proof of Theorem 4.3.2 is that we can let the prover choose its strategy regardless of the input. And then we can argue that since the verifier reads only a constant number of bits from the input before he makes his decision, we can change one input with another without the verifier noticing the change. This cannot be done when the verifier has private coins. Now the prover cannot decide on an arbitrary strategy, because it does not know the private coins of the verifier (i.e. different inputs with the same randomness will not give the same view anymore). This means that if we design our protocol properly, we can force the prover’s bits to depend on the entire input. In this case, the decision bit also depends on the entire input via the prover’s messages. I.e. even though the decision bit depends on a constant number of prover’s bits, each one of them may depend on the entire input. We therefore cannot replace the input without the verifier’s noticing the change.

To see how this works in practice, consider the protocol we give in the proof of Lemma 1.2.1. The last message of the prover contains only a constant number of bits. Let $i \in [\ell]$ be such that $v_i = 1$, and consider the prover’s bit $b_i$. This bit depends on the entire input via the instance $c_i$ that was generated by applying the random instance reduction on the instance $a = (T, x, t, r)$. The protocol, by using private coins, forces the prover to give the correct answer on $c_i$. The dependency of $b_i$ on the input is then revealed to the verifier by computing $\mathcal{E}(b_i, \tau_i)$.

Moving to our result about sub-constant soundness, we want to point out that if we allow a non-constant number of rounds, we can achieve sub-constant soundness. In fact with an
addition of $O(\log n)$ rounds we can achieve the soundness of the original protocol (which can be as small as $2^{-n}$). This is because we can spread the $\mathcal{AC}^0$ computation at the decision stage of the simple proof system, over $O(\log n)$ rounds of the protocol. That is, consider the $\mathcal{NC}^1$ circuit that computes this $\mathcal{AC}^0$ computation. The $\mathcal{NC}^0$ verifier computes at each round another level of the $\mathcal{NC}^1$ circuit. It sends the prover the results of the computation. The prover sends a dummy message, and the verifier continues the computation by reading from the transcript the results from the previous layer of the circuit.

A key point in our results is that we only add a constant number of rounds to obtain an $\mathcal{NC}^0$ verifier. This is what allows us to obtain constant parallel time proof systems for $\mathcal{AM}$ and $\mathcal{NEXP}$.
Chapter 5

Delegation in Program Checking

5.1 Introduction

In this chapter we put 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 5.1.4.

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.

5.1.1 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. An interesting open question is whether problems of related complexity (say problem $\pi_1$ is reducible to problem $\pi_2$) have related checkers (or even if the existence of a checker for $\pi_2$ implies a checker for $\pi_1$). 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

\footnote{A checker was known in the library model of [BLR93]. See section 5.1.2 for more details on the model.}
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:

- **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

²More precisely if a program can be computed by $p(n)$ processors in depth $d(n)$, [Rub96] requires the checker to run either in depth $\omega(d(n))$ or in depth $O(d(n))$ with $o(p(n))$ processors.
and the running time of the optimal program for computing the function?

5.1.2 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.

See Theorem 1.3.2 in Section 1.3 for a statement and discussion of this result, and Section 5.5 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 perfect matching and bounded-degree graph isomorphism.\textsuperscript{3} Theorem 1.3.2 gives provably better checkers for all these functions (see below). Other examples include computing the determinant of a matrix, matrix exponentiation, and more.

Constant Depth Checkers for Matrix Problems. We also use the composition theorem to construct checkers for matrix functions. See Section 1.3 and Theorem 1.3.3 for a statement and overview of these results, and Section 5.3 for discussion and proofs.

5.1.3 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

\textsuperscript{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.
from the checker to the program being checked, all the while verifying that the results returned for these delegated computations are correct. See the discussion in Section 1.3 for more details on this approach and the informal statement of the Checker Composition Theorem (Theorem 1.3.1). See Section 5.3 for a formal statement of the theorem, the proof and extensions, as well as an analogous composition theorem for testers and correctors. We mention that a similar idea, albeit in a very specific setting, was used in the work of Arvind, Subrahmanya 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 5.3 for a formal statement of the theorem, the proof and extensions, as well as an analogous composition theorem for testers and correctors.4

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 5.6). 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 1.3.1, in this regard, to Beigel’s Theorem [BK95] that informally says that a checker for a decision problem π₁ can be used to design a checker for a decision problem π₂, if π₁ and π₂ are “computationally equivalent” (reducible to each other). The idea is that to design a checker for π₂, we can reduce any π₂ instance to a π₁ instance, run the checker for π₁ on this instance, while translating π₁ queries that this checker makes to π₂ instances and feeding them to the program being checked (which allegedly computes π₂). 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

4We 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.
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 $\pi_1$ as a black-box, our approach of decomposition is inherently non-black-box.

**Main Building Blocks: Checkers for Complete Languages.** See Section 1.2.1 for more details about technical building blocks and checkers for complete languages.

### 5.1.4 Other Contributions and Comments

See the discussion of further contributions in Section 1.3.1.

**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. As stated above, the first examples of checking that is provably easier than computing are from [Rub96], which exhibits constant-depth ($\text{AC}^0$) 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

---

5To 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.
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 1.3.2 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 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.

5.1.5 Other Related Work

As discussed above, our work benefits from a long line of beautiful results on program checking, interactive proofs and cryptography. See the related work and discussion in Section 1.2.1.

5.2 Definitions and Preliminaries

5.2.1 Definitions: Checkers, Testers and Correctors

Definition 5.2.1 (Checker). A checker for a function \( f \) with (one-sided) error parameter \( 0 < \delta < 1 \), 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 \( P \), \( \Pr[C^P(x) \in \{f(x), \bot\}] > 1 - \delta \), where \( \bot \) 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 \( \delta \) is 1/3 unless we explicitly note otherwise.\(^6\)

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

**Definition 5.2.2 (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 5.2.3 (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 \( err_{f,P,D}(n) = Pr_{D \leftarrow D^n}[P(x) \neq f(x)] \).

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

**Definition 5.2.4 (Tester).** Let \( 0 < \varepsilon_1 < \varepsilon_2 \leq 1 \) and \( 0 < \delta < 1 \). An \((\varepsilon_1, \varepsilon_2)\)-tester with error \( \delta \) 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 \( \delta \) is 1/6.

**Definition 5.2.5 (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 \( \delta \) (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 \( \varepsilon \)-corrector with respect to \( D \) and with error \( \delta \) (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 \( \delta \).

The default value of \( \delta \) is 1/6.

\(^6\)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.
Note that in all the definitions above, the error parameter $\delta$ 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 $\delta$ that is bounded away from $1/2$ by a constant. Typically, the desired distance parameters $\varepsilon_1, \varepsilon_2$ from Definition 5.2.4 and $\varepsilon$ from Definition 5.2.5 are constants that are bounded away from 0 and 1.

**Definition 5.2.6** (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 $\varepsilon$-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 $\delta$ is $1/3$.

**Remark 5.2.7.** Note that the requirement in Definition 5.2.6 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 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 5.2.8.** Also note that we set the default value of $\delta$ in Definition 5.2.6 to be $1/3$ while in Definition 5.2.4 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.

5.3 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).

5.3.1 Composing Program Checkers

We now restate more formally and prove Theorem 1.3.1. 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 5.3.1** (Composition Theorem for Program Checkers, Theorem 1.3.1 restated).

Let \( L_{\text{internal}} \) and \( L_{\text{external}} \) 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 \( L_{\text{internal}} \) to \( L_{\text{external}} \).
2. The internal language "helps" to check the external language:

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

3. Checkability of the internal language:

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

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

Proof. We construct a checker $C$ for $L_{\text{external}}$, starting from the checker $C_{\text{external}}$ that uses oracle gates to $L_{\text{internal}}$, whose existence is guaranteed by Condition 2 of the theorem. We assume $C_{\text{external}}$ has success probability at least $\frac{5}{6}$ (otherwise we amplify its success probability). Let $P_{\text{external}}$ be the program that $C$ (and $C_{\text{external}}$) checks. Our goal is to replace every oracle call that $C_{\text{external}}$ makes to $L_{\text{internal}}$ with a (probabilistic) circuit $B$, of depth $O(d_{\text{internal}})$, that computes the language $L_{\text{internal}}$ using oracle calls to $P_{\text{external}}$. We base $B$ on the program checker for $L_{\text{internal}}$ (guaranteed by Condition 3). This checker expects to have oracle access to a program that allegedly computes $L_{\text{internal}}$ (and not $L_{\text{external}}$). To that end we define the program $P_{\text{internal}}$ as follows: on instance $x$ of $L_{\text{internal}}$ run the reduction from $L_{\text{internal}}$ to $L_{\text{external}}$ given in Condition 1 to produce instances $x_1, \ldots, x_q$ of $L_{\text{external}}$. Then run $P_{\text{external}}$ 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 $L_{\text{internal}}$), $B$ runs $C_{\text{internal}}$, the checker for $L_{\text{internal}}$ given in Condition 3, with oracle access to $P_{\text{internal}}$. I.e. on every query $y$ that $C_{\text{internal}}$ makes to the program it checks, $B$ runs the reduction from $L_{\text{internal}}$ to $L_{\text{external}}$, and then runs $P_{\text{external}}$ on the outputs of the reduction. The output of the reduction is then returned to $C_{\text{internal}}$ as the answer to the query $y$. In this way $B$ obtains the output of $C_{\text{internal}}$ which gives a prediction regarding the membership of $x$ in $L_{\text{internal}}$. $B$ then runs
$P_{\text{internal}}$ on $x$ and compares its output with the answer of $C_{\text{internal}}$. If these two answers agree, then $B$'s answer is the same as them, otherwise $B$ outputs ⊥. The total depth of $B$ is indeed $O(d_{\text{internal}})$ (since the reduction from $L_{\text{internal}}$ to $L_{\text{external}}$ can be computed in constant depth).

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

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

- **Soundness:**
  
  - If $P_{\text{internal}}(x) = L_{\text{internal}}(x)$ then $B$ cannot output $1 - L_{\text{internal}}(x)$. This is because it only gives a prediction regarding the membership of $x$ if both $P_{\text{internal}}$ and $C_{\text{internal}}$ agree on it (otherwise it returns ⊥).
  
  - If $P_{\text{internal}}(x) \neq L_{\text{internal}}(x)$, the only event that will cause $B$ to output $1 - L_{\text{internal}}(x)$ is if $C_{\text{internal}}$ outputs $1 - L_{\text{internal}}(x)$. But this happens with probability at most 1/6 by the fact that $C_{\text{internal}}$ 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 $P_{\text{external}}$):

1. Run the checker $C_{\text{external}}$, replacing every oracle call to $L_{\text{internal}}$ with a computation of the circuit $B$.

2. If any of $B$'s runs returned the symbol ⊥, then output ⊥. This step is implemented using an AND gate of polynomial fan-in.

3. Otherwise, output the same answer as $C_{\text{external}}$. 

179
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 5.3.2** (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 5.3.3** (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. $\bot$) 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 $\tilde{r}$ 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 $\bot$.

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 $\bot$, and thus will cause $C$ to output $\bot$, i.e. $C$ will not make an error.
We conclude, by the union bound over the errors of \( C_{\text{external}} \) and the errors of \( B \), that \( C \) makes an error with probability at most 1/3.

\[ \]

5.3.2 Extensions

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

**Lemma 5.3.4.** Let \( L_{\text{internal}} \) and \( L_{\text{external}} \) be two languages satisfying the conditions of Theorem 5.3.1. 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 5.3.1, prevents us from using this lemma to construct checkers in \( \mathsf{NC}^0 \). We now show how it can be removed.

**Lemma 5.3.5.** Let \( L_{\text{internal}} \) and \( L_{\text{external}} \) be two languages satisfying the conditions of Theorem 5.3.1, and furthermore, suppose there is a constant-depth reduction from the Parity function\(^7\) to \( L_{\text{external}} \). Then there exists an efficient checker for \( L_{\text{external}} \), of depth \( O(d_{\text{external}} \cdot d_{\text{internal}}) \) (without the unbounded fan-in AND gate at the top).

**Proof.** By the proof of Theorem 5.3.1, 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

---

\(^7\)Recall that the Parity function from \( \{0,1\}^n \) to \( \{0,1\} \) is defined as \( \text{Parity}(b_1, \ldots, b_n) = \sum_{i=1}^n b_i \) with addition over \( GF(2) \).
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, $(b_1 \ldots b_n)$, generate $n$ uniformly random bits $(r_1, \ldots, r_n)$, and compute $\text{Parity}(r_1 \cdot (1 - b_1), \ldots, r_n \cdot (1 - b_n))$ (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 5.3.1 on $L_{\text{external}}$ 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 5.4.2, 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.

**5.3.3 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 5.3.6** ((ε₁, ε₂)-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 \((ε₁, ε₂)\)-reduction from \((L_1, D_1)\) to \((L_2, D_2)\) if, when it is given oracle access to an \(ε₂\)-good program \(P_2\) for \(L_2\) with respect to \(D_2\), \(RP_2\) is an \(ε₁\)-good program for \(L_1\) with respect to \(D_1\).

**Theorem 5.3.7.** Composition Theorem for Testers/Correctors

Let \(L_{\text{internal}}\) and \(L_{\text{external}}\) be two languages. Suppose that there exist parameters (which may depend on the input length \(n\)) \(0 < α_1, α_2, α, ε_1, ε_2, ε, β < 1\), 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 \(L_{\text{internal}}\) with the uniform distribution\(^8\) to \(L_{\text{external}}\) with the distribution \(D\).

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

   \(L_{\text{external}}\) has an efficient \((ε₁, ε₂)\)-tester and \(ε\)-corrector, both have depth at most \(d_{\text{external}}\), and use oracle gates to \(L_{\text{internal}}\).

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

   \(L_{\text{internal}}\) has an efficient \((α₁, α₂)\)-tester and \(α\)-corrector, both have at most depth \(d_{\text{internal}}\) and they form a tester-corrector pair (with default threshold 1/3).

Then \(L_{\text{external}}\) has a \((\frac{1}{2} \cdot \min(ε₁, β), \frac{1}{2} + \frac{ε₂}{2})\)-tester and \((\frac{1}{2} \cdot \min(ε, β))\)-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 \(O(d_{\text{external}} \cdot d_{\text{internal}})\) circuits. Finally, if the tester and corrector for \(L_{\text{external}}\) in condition 2 are a tester-corrector

\(^8\)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).
pair with threshold $\delta$, then the new tester and corrector for $L_{\text{external}}$ form a pair with threshold $\frac{11\delta}{12}$.

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 $L_{\text{internal}}$. Thus, if the number of these oracle calls is constant, then the composed tester and corrector are of bounded fan-in.\(^9\)

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 $L_{\text{external}}$ which is the “average” (a convex combination) of the uniform distribution and the distribution $D$ on $L_{\text{external}}$ instances from the reduction $R$.

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

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

\(^9\)We use this fact to get testers and correctors in $\mathcal{NC}^6$.

184
Let us turn our attention to the tester for $L_{\text{external}}$. We could construct the new tester as we did in the corrector case. I.e., simulate the old tester, obtaining the answers for $L_{\text{internal}}$-queries via correcting the virtual program $P_{\text{internal}}$ (using the corrector for $L_{\text{internal}}$). When $P_{\text{external}}$ 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 $P_{\text{external}}$ is bad with respect to $D'$. In this case we have no guarantee on the behavior of $P_{\text{internal}}$ and thus no guarantee on its correctability. This means that potentially we are simulating the tester with a bad $L_{\text{internal}}$-oracle, in which case all bets are off, and the tester may accept the bad program $P_{\text{external}}$ (even though it should reject it). To address this problem, we add a test that checks $P_{\text{internal}}$ (with respect to the uniform distribution), by running it through the tester for $L_{\text{internal}}$. If it rejects, then it follows that $P_{\text{external}}$ is bad with respect to $D'$ and we correctly reject it. If it accepts, then it follows by hypothesis that the corrector for $L_{\text{internal}}$ indeed corrects $P_{\text{internal}}$ and we obtain a simulation identical to running the old tester for $L_{\text{external}}$ with a perfect oracle for $L_{\text{internal}}$, which is a good tester by Condition 2 of the theorem. 

**Full proof of Theorem 5.3.7.** The proof follows ideas similar to the ones introduced in the proof of Theorem 5.3.1, 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 $L_{\text{external}}$. Our starting point is (again) the tester and corrector for $L_{\text{external}}$, whose existence is guaranteed by Condition 2 of the Theorem. Let $P_{\text{external}}$ be the program to be tested and corrected. Our goal is to replace every oracle call to $L_{\text{internal}}$ with a circuit $B$ that computes the language $L_{\text{internal}}$ using oracle calls to $P_{\text{external}}$, has depth $O(d_{\text{internal}})$, and has the property that if $P_{\text{external}}$ 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 $Cor_{\text{external}}$ and $T_{\text{external}}$ be the corrector and tester for $L_{\text{external}}$ (Condition 2), and $Cor_{\text{internal}}$ and $T_{\text{internal}}$ be the corrector and tester for $L_{\text{internal}}$ (Condition 3). Let $p(n) \leq poly(n)$ be a bound on the number of oracle calls $Cor_{\text{external}}$ and $T_{\text{external}}$ make to
$L_{\text{internal}}$.  

$B$ simulates $\text{Cor}_{\text{internal}} \circ O(\log(p(n)))$ times in parallel, and outputs the majority of these simulations' answers. Note that $B$ cannot directly activate $\text{Cor}_{\text{internal}}$ because it requires access to a program that allegedly computes $L_{\text{internal}}$, whereas $B$ only has access to $P_{\text{external}}$ (that allegedly computes $L_{\text{external}}$). Thus $B$ simulates runs of $\text{Cor}_{\text{external}}$ using the reduction $R$ from $L_{\text{internal}}$ to $L_{\text{external}}$, whose existence is guaranteed by Condition 1 of the theorem. During the simulation of $\text{Cor}_{\text{internal}}$, whenever it makes a call to its program (i.e. queries an instance of $L_{\text{internal}}$), $B$ runs $R$ on that instance with $P_{\text{external}}$ as its oracle, and answers as $R$ does. Thus the total depth of $B$ is indeed $O(d_{\text{internal}})$, 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 $L_{\text{internal}}$ 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 $L_{\text{internal}}$ correctly with high probability.

**Claim 5.3.8.** If $P_{\text{external}}$ is $(\frac{1}{2} \cdot \min(\varepsilon_1, \beta))$-good for $L_{\text{external}}$ with respect to the distribution $D'$ then for every $x \in \{0, 1\}^n$,

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

**Proof.** $B$ runs $\text{Cor}_{\text{internal}}$ which corrects programs that are $\alpha_1$-good for $L_{\text{internal}}$ with respect to the uniform distribution. $B$ uses a “program” for $L_{\text{internal}}$ that is obtained by running the reduction $R$ using $P_{\text{external}}$ as its oracle. Since $P_{\text{external}}$ is $(\frac{1}{2} \cdot \min(\varepsilon_1, \beta))$-good for $L_{\text{external}}$ with respect to the distribution $D'$, it is also $\beta$-good for $L_{\text{external}}$ with respect to the distribution $D$ (by the construction of $D'$). We know that when $R$ is run with a $\beta$-good program for $L_{\text{external}}$ as its oracle, it gives an $\alpha_1$-good program for $L_{\text{internal}}$ with respect to the uniform distribution.

By the fact that $\text{Cor}_{\text{internal}}$ corrects programs that are $\alpha_1$-good with respect to the uniform distribution, on every instance of $L_{\text{internal}}$, every execution of $\text{Cor}_{\text{internal}}$ by $B$ gives a correct answer with probability at least $\frac{5}{6}$. By taking the majority of $O(\log(p(n)))$ executions of
$\textbf{The Corrector Cor.}$ The corrector $\text{Cor}$ runs the “old” corrector $\text{Cor}_{\text{external}}$, replacing every oracle call to $L_{\text{internal}}$ with a call to the circuit $B$. Thus $\text{Cor}$ has depth $O(d_{\text{external}} \cdot d_{\text{internal}})$ and the only gates of unbounded fan-in are the majority gates used by $B$, which have fan-in $O(\log(p(n)))$.

\textbf{Claim 5.3.9.} $\text{Cor}$ is a $\left(\frac{1}{2} \cdot \min(\epsilon_1, \beta)\right)$-corrector for $L_{\text{external}}$ with respect to the distribution $D'$.

Proof. Let $P_{\text{external}}$ be $\left(\frac{1}{2} \cdot \min(\epsilon_1, \beta)\right)$-good with respect to the distribution $D'$. By the construction of $D'$, it follows that $P_{\text{external}}$ is $\epsilon$-good with respect to the uniform distribution (recall that $\epsilon = \epsilon_1$). This means that $\text{Cor}_{\text{external}}$ computes every instance of $L_{\text{external}}$ correctly with probability at least $\frac{5}{6}$ when given oracle access to $L_{\text{internal}}$ and $P_{\text{external}}$. When $\text{Cor}$ simulates $\text{Cor}_{\text{external}}$, it replaces calls to $L_{\text{internal}}$ by an execution of $B$. By Claim 5.3.8, $B$ gives a correct answer with probability at least $1 - \frac{1}{12 p(n)}$ on every execution. Since $\text{Cor}$ 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 $L_{\text{external}}$, $\text{Cor}$ computes it correctly with probability at least $\frac{2}{3}$.

$\textbf{The Tester T.}$ Let $P_{\text{external}}$ 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 $T_{\text{internal}}$, replacing each oracle call to its program, by executing the reduction $R$ on the $L_{\text{internal}}$ instance, and using $P_{\text{external}}$ as the oracle for the reduction. If $T_{\text{internal}}$ 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 $T_{\text{external}}$, replacing every one of its oracle calls to $L_{\text{internal}}$ by running the circuit $B$. Output whatever answer this simulation of $T_{\text{external}}$ gives.
T has depth $O(d_{\text{external}} \cdot d_{\text{internal}})$ 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 $L_{\text{external}}$ with respect to the distribution $D'$ (this is established in Claims 5.3.10 and 5.3.12 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 5.3.13).

**Claim 5.3.10.** $T$ 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 $P_{\text{external}}$ is $(\frac{1}{2} \cdot \min(\varepsilon_1, \beta))$-good w.r.t. the distribution $D'$, thus it is also $\beta$-good w.r.t. the distribution $D$. It follows that when $P_{\text{external}}$ is used as the oracle of the reduction $R$, we get a $\alpha_1$-good program for $L_{\text{internal}}$. So in the first step, in each execution, $T_{\text{internal}}$ 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 $P_{\text{external}}$ is also $\varepsilon_1$-good with respect to the uniform distribution on $L_{\text{external}}$ instances. $T_{\text{external}}$ is an $(\varepsilon_1, \varepsilon_2)$-tester with success probability $\frac{5}{6}$, and thus when $B$ gives correct answers in all its executions, $T_{\text{external}}$ accepts with probability at least $\frac{5}{6}$, even if we replace its $L_{\text{internal}}$ oracle with $B$. By Claim 5.3.8 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 $P_{\text{external}}$ with probability at least $\frac{2}{3}$. $\blacksquare$

**Claim 5.3.11.** Let $P'$ be a program (supposedly computing $L_{\text{internal}}$) obtained by running the reduction $R$ on instances of $L_{\text{internal}}$ and taking $P_{\text{external}}$ to be the oracle for the reduction. If $P'$ is not correctable by $\text{Cor}_{\text{internal}}$, then $T$ rejects in its first step with probability at least $\frac{11}{12}$.

**Proof.** $T_{\text{internal}}$ and $\text{Cor}_{\text{internal}}$ are a tester-corrector pair with threshold $1/3$ (the default value). So if $P'$ is not correctable by $\text{Cor}_{\text{internal}}$ it must hold that $T_{\text{internal}}$ 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 5.3.12.** \( T \) 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 \( Cor_{\text{internal}} \), 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 5.3.11).

If \( Cor_{\text{internal}} \) does compute \( L_{\text{internal}} \) 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 \( \varepsilon_2 \)-good w.r.t. the uniform distribution on \( L_{\text{external}} \) inputs. When \( B \) always gives correct answers, \( T_{\text{external}} \) 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 5.3.13.** If \( T_{\text{external}} \) and \( Cor_{\text{external}} \) form a tester-corrector pair with threshold \( \delta \) (for some \( 0 < \delta < 1/2 \)), then \( T \) and \( Cor \) form a pair with threshold \( \delta' = \frac{11 - \delta}{12} \).

**Proof.** If \( T \) accepts a program \( P_{\text{external}} \) with probability at least \( 1 - \delta' > 1/12 \) then in particular it passes the first step with this probability. This implies by Claim 5.3.11 that the program \( P \) (for \( L_{\text{internal}} \)) obtained by running the reduction \( R \) with \( P_{\text{external}} \) as its oracle, is correctable by \( Cor_{\text{internal}} \).

If the above program \( P \) is correctable by \( Cor_{\text{internal}} \), 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 \( T_{\text{external}} \) with oracle to \( B \) is identical to its behavior with oracle to \( L_{\text{internal}} \).) The probability that \( T \) rejects in this
conditional space is at most $\frac{12\delta'}{11} = \delta$. By the fact that $T_{\text{external}}$ and $C_{\text{external}}$ form a tester-corrector pair with threshold $\delta$ (when they are given oracle access to $L_{\text{internal}}$) it follows that $P_{\text{external}}$ is correctable by $C_{\text{external}}$ when the latter is given oracle access to $L_{\text{internal}}$. By replacing oracle calls to $L_{\text{internal}}$ with calls to $B$ we increase the error probability of $C_{\text{external}}$ by at most $\frac{1}{12}$. ■

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

**Claim 5.3.14.** 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 $\varepsilon_1$ and distribution $D_1$, there exists a distribution $D_2$ such that $R$ is an $(\varepsilon_1, \frac{\varepsilon_1}{q})$-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 $\frac{\varepsilon_1}{q}$-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 $\varepsilon_1$. We conclude that if $P$ is $\frac{\varepsilon_1}{q}$-good for $L_2$ with respect to $D_2$, then $R$ using $P$ as its oracle is $\varepsilon_1$-good for $L_1$ w.r.t. $D_1$. ■

**Remark 5.3.15.** 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 $\beta$ 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 5.3.14. The only exceptions occur in Section 5.6, 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 5.6.6, 5.6.10).

Remark 5.3.16. When using Theorem 5.3.7 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 5.3.7 are constants (i.e. the $\varepsilon$'s, the $\alpha$'s and $\beta$), and the distance parameters of the original testers/correctors (i.e. the $\varepsilon$'s and the $\alpha$'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 $\delta$ then the threshold for the new pair is $(1 - \lambda) \cdot \delta$ with $\lambda = 1/12$. We want to point out that $\lambda$ can be set to be an arbitrarily small constant.

We conclude that if we apply Theorem 5.3.7 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.

5.4 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 1.3.4. Before reading this section, the reader should be somewhat familiar with the definitions and results of Section 2.3

5.4.1 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 5.4.1.** 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 $(1, \frac{2}{3})$-tester-corrector pair and a program checker, where all of them can be implemented by bounded fan-in circuits of depth $d + O(1)$.

**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 $(y, \tau)$ 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.

5.4.2 Checkers, testers and correctors for complete languages

We now apply Theorem 5.4.1 to obtain extremely efficient checkers, testers and correctors
for complete languages.

Lemma 5.4.2. The Parity function has a \((1, \frac{2}{3})\)-tester, \(\frac{1}{3}\)-corrector and a program checker
all implementable in \(NC^0\).

Proof. The proof is immediate from Theorem 5.4.1, as the parity language has an \(NC^0\) solved
instance generator and random instance self-reduction (see Corollary 2.3.6).

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

Proof. Applying Corollary 2.3.7 and Theorem 5.4.1 (with \(AC^0\) gates), we obtain a checker,
tester and corrector that are implementable in \(AC^0\) for the \(NC^1\)-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 5.3.1). We replace the AND and OR gates with Parity gates as it is done in the proof of Lemma 5.3.5. We then use the Composition Theorem to remove the Parity gates (again, we refer the reader to the proof of Lemma 5.3.5).

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 5.4.4.** 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 5.4.5.** The class $\text{mod}_k$-$\mathcal{L}$ (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 5.4.6.** 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 5.4.7.** The language $\text{mod}_k$-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 5.4.8.** $\oplus$-$\mathcal{L}$ and $\text{mod}_k$-$\mathcal{L}$ are complete for $\oplus$-connectivity and $\text{mod}_k$-connectivity respectively under $\mathcal{NC}^0$ reductions.

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

**Corollary 5.4.9.** $\oplus$-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 5.4.1); 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 5.3.1 and 5.3.7) 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 \( \oplus \)-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 5.4.2.

   By the Composition Theorems (Theorems 5.3.1 and 5.3.7) 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 5.3.5).

\[ \blacklozenge \]

**Corollary 5.4.10.** Let \( k \) be a prime. Then \( \text{mod}_k \)-connectivity has an \( \mathcal{NC}^0 \) program checker, and an \( \mathcal{AC}^0 \left( \frac{1}{3}, \frac{2}{3} \right) \)-tester and \( \frac{1}{3} \)-corrector pair.

*Proof. The work of [IK02] shows that \( \text{mod}_k \)-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 5.4.1 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 5.3.1 and 5.3.7). 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 5.4.2. 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 5.3.1). This uses the fact that both AND and Parity are reducible to mod_{k}-connectivity under \mathcal{NC}^0 reductions. 

\section{Checkers, Testers and Correctors for a Complexity Class}

In this section we use the Composition Theorems to prove Theorem 1.3.2. 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 1.3.2. The generalized theorem (Theorem 5.5.1 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 1.3.2. Let \( L \) be a language in \mathcal{RNC}^i that is \mathcal{NC}^1-hard under \mathcal{NC}^0-reductions. To build the \mathcal{RNC}^{i-1} checker for \( L \), we begin with the trivial checker for \( L \), i.e.
simply a (correct) $\mathcal{RNC}^i$ circuit for $L$. We then decompose this circuit $C$ into $O(\log^{-1} n)$ “layers” of depth $\log n$ each. Put another way, we view this checker as a circuit of depth $O(\log^{-1} n)$ 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 $\mathcal{NC}^1$ language, we can replace it with an oracle that computes the $\mathcal{NC}^1$-complete language from Theorem 1.3.4 (without paying more then a constant factor in depth). We now use the Composition Theorem to eliminate these oracle gates. We take the $\mathcal{NC}^1$-complete language accepted by the oracle gates to be our internal language $L_{\text{internal}}$, and take $L$ to be $L_{\text{external}}$. Clearly, all the conditions of the theorem hold: (a) $L_{\text{internal}}$ is in $\mathcal{NC}^1$, and therefore is reducible to $L$ (which we assumed is $\mathcal{NC}^1$-hard), (b) $L_{\text{external}}$ has a checker of depth $O(\log^{-1} n)$ that uses oracle calls to $L_{\text{internal}}$, and (c) $L_{\text{internal}}$ has a $\mathcal{NC}^0$ checker (Theorem 1.3.4). We therefore conclude that $L$ has an $\mathcal{RNC}^{i-1}$ 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 1.3.2 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 5.5.1 (Generalization of Theorem 1.3.2). Let $L$ be a language computable by circuits of depth $d$, and let $d_{\text{collapse}} > 0$ be some integer (that could be a function of the input
length $n$). If there exists a language $L_{\text{internal}}$ such that:

1. (L is “harder” than $L_{\text{internal}}$) There is a constant depth reduction from $L_{\text{internal}}$ to $L$.

2. ($L_{\text{internal}}$ is complete for depth $d_{\text{collapse}}$ computations) There is a constant depth reduction from any language computable by circuits of depth $d_{\text{collapse}}$ to $L_{\text{internal}}$.

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

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

Proof. The proof follows from Theorems 5.3.1 (for programs checkers) and 5.3.7 (for program testers and correctors). Details follow.

**Program Checkers:** Take $L$ to be the “external” language, and $L_{\text{internal}}$ to be the “internal” language. We show that the three conditions of Theorem 5.3.1 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 $L_{\text{internal}}$, together with the fact that $L_{\text{internal}}$ is hard for depth $d_{\text{collapse}}$ circuits, to construct an efficient depth $O(d/d_{\text{collapse}})$ checker for $L$ that uses oracle gates to $L_{\text{internal}}$. This checker divides the circuit $C$ into $O(d/d_{\text{collapse}})$ layers, each of depth $d_{\text{collapse}}$. It then “collapses” each such layer to constant depth with an oracle call to $L_{\text{internal}}$ (this can be done using the constant-depth reduction from any depth-$d_{\text{collapse}}$ computation to $L_{\text{internal}}$ guaranteed by Condition 2 of the theorem).
The depth of this new checker is \( O(d/d_{\text{collapse}}) \). In the terms of Theorem 5.3.1, this is the value \( d_{\text{external}} \).

- 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 5.3.1 \( d_{\text{internal}} = O(1) \).

Now we apply Theorem 5.3.1, and we conclude that \( L \) has a depth \( O(d_{\text{external}} \cdot d_{\text{internal}}) = O(d/d_{\text{collapse}}) \) 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 5.3.7 that \( L \) has a tester and corrector of depth \( O(d/d_{\text{collapse}}) \). The distance parameters of the tester and corrector are affected by success probability of the reduction from depth \( d_{\text{collapse}} \) circuits to \( L_{\text{internal}} \) (or by its number of queries, see Claim 5.3.14). 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 1.3.2.

*Proof of Theorem 1.3.2.* Let \( L \) be the language that is in \( \mathcal{RNC}^i \), and is \( \mathcal{NC}^1 \)-hard under \( \mathcal{NC}^0 \)-reductions. We take the \( \mathcal{NC}^1 \)-complete language from Lemma 5.4.3 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 (\( \mathcal{AC}^0 \)) tester and corrector (by Lemma 5.4.3), we conclude (by Theorem 5.5.1) that \( L \) has a tester and corrector of depth \( O(\log^{-1}(n)) \), that uses unbounded fan-in gates. In other words, \( L \) has a tester and corrector in \( \mathcal{RAC}^{i-1} \).

Note that since \( L \) (the external language here) is \( \mathcal{NC}^1 \)-hard under \( \mathcal{NC}^0 \) reductions, there are \( \mathcal{NC}^0 \) reductions from the Parity function as well as the AND function to \( L \). Also, the internal language has an \( \mathcal{NC}^0 \) program checker by Lemma 5.4.3. By using the Composition
Theorem for program checkers (Theorem 5.3.1), and an argument similar to the one given in the proof of Lemma 5.3.5, we conclude that \( L \) has a checker in \( \mathcal{RNC}^{i-1} \) (without the additional unbounded fan-in AND gate). □

5.6 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 5.6.1, followed by detailed constructions and analyses. A summary of the parameters that we achieve and comparison to previous constructions appear in Table 5.1 below.

<table>
<thead>
<tr>
<th>Function</th>
<th>Depth</th>
<th>Time</th>
<th>Program Calls</th>
<th>Previously Known</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiplication over ( \text{GF}(2^s) ), ( s = O(1) )</td>
<td>( \mathcal{NC}^{0} )</td>
<td>( O(n^2) )</td>
<td>( O(1) )</td>
<td>( \mathcal{AC}^{0} ), ( O(\log n) ) program calls [BLR93, Rut]</td>
</tr>
<tr>
<td>Multiplication over any finite field</td>
<td>( \mathcal{AC}^{0} )</td>
<td>( O(n^2) )</td>
<td>( O(1) )</td>
<td>( \mathcal{AC}^{0} ), ( O(\log n) ) program calls [BLR93, Rut]</td>
</tr>
<tr>
<td>Inversion over ( \text{GF}(2^s) ), ( s = O(1) )</td>
<td>( \mathcal{NC}^{0} )</td>
<td>( O(n^2) )</td>
<td>( O(1) )</td>
<td>Library only, poly depth [BLR93]</td>
</tr>
<tr>
<td>Inversion over any finite field</td>
<td>( \mathcal{AC}^{0} )</td>
<td>( O(n^2) )</td>
<td>( O(1) )</td>
<td>Library only, poly depth [BLR93]</td>
</tr>
<tr>
<td>Determinant over any finite field</td>
<td>( \mathcal{AC}^{0} )</td>
<td>( \text{poly}(n) )</td>
<td>( \text{poly}(n) )</td>
<td>Library only, poly depth [BLR93]</td>
</tr>
<tr>
<td>Rank over ( \text{poly}(n) ) size fields</td>
<td>( \mathcal{AC}^{0} )</td>
<td>( \text{poly}(n) )</td>
<td>( \text{poly}(n) )</td>
<td>Library only, poly depth [BLR93]</td>
</tr>
</tbody>
</table>

Table 5.1: Complexity of Tester/Correctors (with constant error) for \( n \times n \) Matrix Operations.

5.6.1 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 $\beta$, 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 $\beta$). 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 1.3.1). We take $L_{\text{internal}}$ to be Parity, and $L_{\text{external}}$ to be matrix multiplication. We now observe that the conditions of Theorem 1.3.1 hold: (1) there is a constant depth reduction from Parity to Matrix multiplication: let $\vec{U}$ be the vector of bits whose parity we wish to compute; construct a matrix whose first row is $\vec{U}$ and multiply it by the all-ones matrix; the top-left item in the result is the parity of $\vec{U}$; (2) there is a constant depth checker for matrix multiplication that uses an oracle to Parity; (3) Parity has a constant depth checker (Theorem 1.3.4). Applying Theorem 1.3.1, 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 chal-
lenging. 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 $\text{GF}(2)$). 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 $\bot$.

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 means 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 \textit{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 \textit{every} matrix with high probability.

\textit{Different Testers and Correctors.} The library testers and correctors of \cite{BLR93} use a procedure of Randall \cite{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 \cite{BLR93}. The most significant example is our tester for the determinant function. The tester of \cite{BLR93} simply checks that the program is correct on random matrices with known determinant (using \cite{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 \cite{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 \textit{only} non-constant homomorphism that is multi-linear in the entries of the matrix.

\textit{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 \( f \).

5.6.2 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 5.6.1. 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 \( \beta \). 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 5-1.

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 5-2 and 5-3), 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 5.3.7).

Our Matrix Multiplication Corrector and Tester: To apply Theorem 5.3.7, we take matrix multiplication to be the external function, and matrix-row-sums to be the internal
MATRIX MULTIPLICATION TESTER($\beta$)

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 $\text{Freivalds.Checker}(A, B, C, \frac{\beta}{2})$ 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 5-3: Matrix Multiplication Tester

function. The $\text{matrix-row-sums}$ 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 $\text{matrix-row-sums}$ can be used to compute matrix-vector multiplications in $\mathcal{NC}^0$:

   $$A \times \vec{v} = \begin{pmatrix} a_{1,1} & \ldots & a_{1,n} \\ a_{2,1} & \ldots & a_{2,n} \\ \vdots \\ a_{n,1} & \ldots & a_{n,n} \end{pmatrix} \times \begin{pmatrix} v_1 \\ \vdots \\ v_n \end{pmatrix} = \begin{pmatrix} \sum_{i=1}^{n} a_{1,i} \cdot v_i \\ \sum_{i=1}^{n} a_{2,i} \cdot v_i \\ \vdots \\ \sum_{i=1}^{n} a_{n,i} \cdot v_i \end{pmatrix}$$
This final vector can be computed (in $\mathcal{NC}^0$) by computing $\text{matrix-row-sums}$ 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 $\text{matrix-row-sums}$).

3. Testability and Correctability of the internal function:

The tester and corrector for the $\text{matrix-row-sums}$ function are generalizations of the tester and corrector for computing products over finite groups given by Claim 2.3.5 and Theorem 5.4.1 (both appear in Section 5.4. 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 2.3.5. This randomization technique is then used, as in the proof of Theorem 5.4.1, to test and correct the program that allegedly computes the $\text{matrix-row-sums}$ 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 $\beta$) the tester and corrector that we start with (before applying the composition), as well as all reductions and the $\text{matrix-row-sums}$ 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).
5.6.3 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 5.6.2.** 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 5-4) and tester (Figure 5-5) 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 \( F \).\(^{10}\) We use \( P \) to denote the matrix inversion program being checked, and \( \text{Mult} \) to denote the (always correct) matrix multiplication oracle.

**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 \( A \)s. 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 \( 1/8 \)-close to

\(^{10}\)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 5.2.7.
**Matrix Inversion Corrector** $(A, \beta)$

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

2. Output that $A$ is not invertible.

---

Figure 5-4: Matrix Inversion Corrector

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 $1 - 1/4$) 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 $\gamma$ be the probability that a random matrix over $F$ is invertible (can be hard-wired into the tester, $\gamma$ 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 is 1 is at least $1 - \gamma$. If the program is at least $1/32$-close to being correct (on the invertible matrices), then the probability that the answer from the iteration is 0 is at least $31\gamma/32$, and the probability that the fraction of $O(\log(1/\beta))$ independent iterations that give 0 answers is $1 - \gamma/16$-distant from $\gamma$ is at most $1 - \beta$ (by a Chernoff Bound). If the program is at least $1/5$-far from being correct (on a random invertible matrix), then the probability that the answer from the iteration is 0 is at most $7\gamma/8$, and the probability that the fraction of $O(\log(1/\beta))$ independent iterations that give 0 answers is $7\gamma/16$-distant from $\gamma$ is
1. Let $\gamma$ 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 $\eta$ be the fraction of 0-answers in the loop. If $|\eta - \gamma| \leq \frac{\beta}{16}$, then accept. Otherwise reject.

Figure 5-5: Matrix Inversion Tester

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 $\beta$ 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 5.3.7), 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 \beta and field size) tester and corrector for matrix multiplication given in
Section 5.6.2.

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,
observed 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
\( 3 \times 3 \) 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 \( \beta \)) 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 \textit{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.

**5.6.4 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 ⊥ for non-invertible matrices). A description of such a procedure appears in Figure 5-6.

```
Random - Invertible(n, β)

1. For $i$ going from 1 to $O(\log(1/β))$ 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 $i$, $A_i^{-1} = ⊥$, output ⊥. Otherwise, output $A_i$ for the minimal $i$ for which it is not ⊥.
```

Figure 5-6: Generating a random invertible matrix
Claim 5.6.3. The procedure in Figure 5-6 outputs $\bot$ with probability at most $\beta$. Furthermore, conditioned on the event that it does not output $\bot$, its output is uniformly distributed over invertible matrices with entries in the field $F$. If $\beta$ 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 $\mathcal{AC}^0$.

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 $\beta$ 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 5.6.4. The matrix determinant function over a finite field $F$ has an $\mathcal{AC}^0$ tester and corrector.

Proof. The corrector (Figure 5-7) and tester (Figure 5-8) 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 5.3.7). We denote by $P$ the matrix determinant program being checked, and by $\text{Mult}$ and $\text{Inv}$ 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 $\text{Inv}$ 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 5.6.3, the probability that $\text{Random} - \text{Invertible}$ 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
MATRIX DETERMINANT CORRECTOR($A, \beta$)

1. If $\bot = Inv(A)$ 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 5-6). 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 5-7: Matrix Determinant Corrector

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 $\bot$, 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 $\beta$ (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 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
**Matrix Determinant Tester**

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

   (a) Run twice `Random - Invertible(n, 1/100)` to generate two matrices $R_1$ and $R_2$. If either execution outputs ⊥, 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 $1/16$ then reject.

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

   (a) Run `Random - Invertible(n, 1/32)` to generate a matrix $R$. If the execution outputs ⊥, 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 $1/8$ then reject, otherwise accept.

---

Figure 5-8: Matrix Determinant Tester

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 $3/256$. The probability that `Random - Invertible` fails in one of its two activations is at most $1/50$. Taking a union bound, the total probability of the answer in each iteration being 0 is less than $1/24$. Thus, the program is rejected after the loop with probability at most $9/2$.

Now consider the case that the program is $1/16$-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 $1/50$. 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 $1/8$ (see [BCLR04]). Taking
a union bound, the probability of the answer in each iteration being 0 is at least $\frac{1}{16}$. Thus the program is rejected after the loop with probability at least $1 - \frac{\theta}{2}$.

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 5.6.5.** For every homomorphism $h : GL_n(F) \rightarrow F^*$, 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 $h : G \rightarrow F^*$. This is a homomorphism from $(F^*)^n$ to $F^*$, and such homomorphisms are easily seen to all be of the form $h(M) = a_{k_1}a_{k_2} \cdots a_{k_n}$ (for $0 \leq k_1, \ldots, k_n \leq |F| - 1$), where the $a_i$'s 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 \( h : GL_n(F) \rightarrow F^* \), then by Claim 5.6.5, \( h(M) = \det(M)^k \) 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 \( \beta \). 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.\textsuperscript{11} We want to use the Composition Theorem (Theorem 5.3.7), 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 5.6.2.

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 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 5.3.14). 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

\textsuperscript{11}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.
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 5.6.6.** For any constant \( \varepsilon \), there exists a constant depth \((\varepsilon, \frac{1}{\varepsilon})\)-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 5-7: first, start the execution of the corrector from step 2 in Figure 5-7 (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 $\beta$ (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 5.6.7.** 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 5.6.6.

**Proposition 5.6.1.** 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 5.6.7 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 5.6.2), 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 5.4.2 and the matrix-row-sums tester and corrector in the proof of Claim 5.6.1). Moreover, there is a one-to-one $NC^0$ reduction from vector-sum to determinant. The reduction on a vector $\vec{v} = (x_1, x_2, \ldots, x_n)$ proceeds as follows:

$$\begin{vmatrix}
    x_1 & -x_2 & x_3 & \ldots & -1^{n-1} \cdot x_n \\
    1 & 1 & 0 & \ldots & \ldots \\
    0 & 1 & 1 & 0 & \ldots \\
    \vdots & \vdots & \vdots & \ddots & \vdots \\
    0 & 0 & \ldots & 1 & 1
\end{vmatrix}$$

Applying the Composition Theorem (Theorem 5.3.7), we get a constant-depth tester and corrector for matrix determinant without any additional oracle gates. 


5.6.5 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 5.6.8.** 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 5-9) and tester (Figure 5-10) 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 $I_{n \times n}$ 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 $\text{Mult}$ and $\text{Inv}$ 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 $\{0 \ldots 2n\}$ and then generating a random $2n \times 2n$ matrix of rank $r$.

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

**Claim 5.6.9.** 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 Corrector**$(A, \beta)$

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 ⊥, the answer from this iteration is ⊥.

   (b) Choose a random $t \in \{0 \ldots n\}$

   (c) $A_1 \leftarrow \begin{pmatrix} A & 0_{n \times n} \\ 0_{n \times n} & I_{n \times n}^t \end{pmatrix}$

   (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 5-9: Matrix Rank Corrector**

**Matrix Rank Tester**$(\beta)$

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 ⊥, the answer from this iteration is 0.

   (b) Choose a random $r \in \{0 \ldots 2n\}$

   (c) $A_1 \leftarrow I_{2n \times 2n}$

   (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 5-10: 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.\(^{12}\) We now use the Composition Theorem (Theorem 5.3.7), 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 5.6.4.

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 (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 \(\text{mod}_k - L\) (see e.g. [BDHM91]). Furthermore, any boolean \(\text{mod}_k - L\) 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.

\(^{12}\)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 \(\mathcal{AC}^0\).
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 5.6.10.** For any $\varepsilon > 0$, there exists a constant depth $(\varepsilon, \frac{1}{6^4})$-reduction from matrix determinant of $n \times n$ matrices on any distribution, to matrix rank on the distribution on $n^\varepsilon \times n^\varepsilon$ matrices obtained by choosing at random a rank $r$ between 0 and $n^\varepsilon$, 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 $\text{rank}(D_a)$. This is done, similarly to the amplified reduction from inversion to determinant of Claim 5.6.6, 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 $\text{Random - Invertible}$). 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 5.6.11.** The matrix rank function over prime fields of polynomial size has an \( \mathcal{AC}^0 \) tester and corrector with a matrix multiplication oracle.

**Proposition 5.6.2.** 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 5.6.11 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 5.6.1). Furthermore, these vector-sum oracles can be replaced (while maintaining constant depth) by oracles to the (boolean) \textit{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 \textit{vector-sum-equal}, simply call \textit{vector-sum-equal} (in parallel) with all possible values $a$, and output the one correct $a$ for which \textit{vector-sum-equal}’s output is 1. The number of calls to \textit{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 \textit{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.  

\[\blacksquare\]
Chapter 6

Delegation in Error Correcting Codes

6.1 Introduction

**Background.** We study the complexity of locally decoding and locally list decoding codes, we consider both the binary case (i.e. where the code’s alphabet is \{0, 1\})\(^1\) and the non-binary case.

Let us proceed more formally. Let \(C : \Gamma^M \to \Gamma^N\) be the encoding function of an error-correcting code.\(^2\) A local list-decoder \(D\) for a code \(C\) gets oracle access to a corrupted codeword, and outputs a “list” of \(\ell\) local-decoding circuits \(D_1, \ldots, D_\ell\). Each \(D_a\) is itself a probabilistic circuit with oracle access to the corrupted codeword. On input an index \(j \in [M]\), a circuit \(D_a\) from the list tries to output the \(j\)-th symbol of the message. We say that the decoder is a \((\delta, \ell)\)-local-list-decoder, if for every \(y \in \{0, 1\}^N\) and \(m \in \{0, 1\}^M\), such that the fractional Hamming distance between \(C(m)\) and \(y\) is at most \(\delta\), with high probability at least one of the \(D_a\)’s successfully decodes every symbol of the message \(y\).\(^3\)

\(^1\)Binary locally list decodable codes are of particular interest for their applications to hardness amplification.

\(^2\)Formally, we consider a family of codes one for each message length \(M\). The parameters listed above and below, e.g. \(N, \varepsilon, \ell\), should all be thought of as functions of \(M\). For the exact definition of locally list-decodable codes see Definition 6.2.1.

\(^3\)We would like to point out that for binary codes we use \((1/2 - \varepsilon, \ell)\) to denote the relative distance and list size, whereas previous work (e.g. [STV01]) used \((\varepsilon, \ell)\) to denote the same quantities (again, for binary codes). We find this notation more useful, especially when we work with non-binary codes.
When the list size \( \ell \) equals 1, i.e. there is a single decoding algorithm that is guaranteed to decode every bit of the message (w.h.p.), we say that the code is *locally decodable* (without a list). The quantity \( N/M \) which measures the amount of redundancy in the code is called the *rate* of the code.

Throughout this paper, when we speak of locally list decodable codes (rather than just locally decodable codes), we think of a local list-decoder as receiving an “advice” index \( a \in [\ell] \), running \( D \) to output \( D_a \), and then running \( D_a \) to retrieve the \( j \)-th message symbol. Note that by giving both \( D \) and the \( D_a \)'s oracle access to the received word, and requiring them to decode individual symbols, we can hope for decoders whose size is much smaller than \( N \) (in particular we can hope for size that is poly-logarithmic in \( N \)). See section 6.2.1 for formal definitions of locally decodable and locally list decodable codes.

### 6.1.1 This Work

This work studies the complexity of local and local-list decoders. Specifically, we ask whether it is possible to decode in constant depth. We answer this question in the affirmative, presenting transformations that reduce the depth of decoders and obtaining explicit locally decodable and locally list-decodable codes with constant-depth decoders.

**Our approach.** See Section 1.4.1 for an overview of our approach to constructing more efficient decoders.

**Explicit Constructions.** Using the above transformations, we obtain explicit codes that can be decoded very efficiently. Of course, to apply the transformations we need to begin with a decoder whose computations are in \( \mathsf{NC}^1 \)! We design such codes, and in particular we obtain a binary locally decodable code, and both binary and non-binary locally list-decodable codes. These are described in Theorems 1.4.3 and 1.4.4 in Section 1.4, and the following non-binary construction:

**Theorem 6.1.1** (Locally list-decodable non-binary code). For every \( \varepsilon > 0 \), there is an explicit code \( C : \{0,1\}^M \rightarrow \Sigma^N \) that is locally list-decodable by probabilistic \( \mathsf{AC}^0 \) circuits of
size \( \text{poly}(\log M/\varepsilon) \) from agreement \( \varepsilon \) and with list size \( \text{poly}(1/\varepsilon) \). Where \( |\Sigma| = 2^{\text{poly}(1/\varepsilon)} \), and \( N = M^\text{poly}(1/\varepsilon) \).

**Lower Bounds.** Finally, we characterize the complexity of locally list-decoding binary codes. We show that the codes of Theorem 1.4.4 are essentially optimal. See Theorem 1.4.6 in Section 1.4 for an informal statement, the proof is in Section 6.5.

From Theorems 1.4.4 and 1.4.6 we conclude that computing the majority function on \( \Theta(1/\varepsilon) \) bits is essentially equivalent to \( (1/2 - \varepsilon, \text{poly}(1/\varepsilon)) \)-local-list-decoding binary codes: any circuit for a local-decoder of such a code can be used to construct a circuit of roughly the same size and depth that computes majority on \( \Theta(1/\varepsilon) \) bits. In the other direction, there is an explicit \( (1/2 - \varepsilon, \text{poly}(1/\varepsilon)) \)-locally-list-decodable code with a very efficient (in terms of size and depth) local-decoder that uses majority gates of fan-in \( \Theta(1/\varepsilon) \).

By known lower bounds on the size of constant-depth circuits that compute majority [Raz87, Smo87], we obtain the following corollary.

**Corollary 6.1.2 (Informal).** Any constant-depth \( (1/2 - \varepsilon, \text{poly}(1/\varepsilon)) \)-local list decoder for a binary code, must have size almost exponential in \( 1/\varepsilon \). This holds even if the decoder is allowed \( \text{mod } q \) gates, where \( q \) is an arbitrary prime number.

In particular, we get an exact characterization of what is possible with constant-depth decoders: up to radius \( 1/2 - 1/\text{poly log log } M \) locally-list-decodable codes with constant-depth decoders and good parameters exist, and beyond this radius they do not. We note that in fact we prove a stronger result in terms of the list size. We show that \( (1/2 - \varepsilon, \ell) \)-local-list-decoding with a decoder of size \( s \) and depth \( d \), implies a circuit of size \( \text{poly}(s, \ell) \) and depth \( d \) that computes majority on \( O(1/\varepsilon) \) bits. This means that even if the list size is sub-exponential in \( 1/\varepsilon \), the size of the decoder still must be nearly exponential in \( 1/\varepsilon \) (even if the decoder is allowed \( \text{mod } q \) gates).

**Hardness amplification.** Hardness amplification is the task of obtaining from a Boolean function \( f \) that is somewhat hard on the average, a Boolean function \( f' \) that is very hard on
the average. By a beautiful sequence of works [STV01, TV07, Tre03, Vio05], it is well known that there is a tight connection between binary locally (list) decodable codes and hardness amplification. Using this connection, we obtain both new positive and negative (in the spirit of Corollary 6.1.2) results on (black-box) hardness amplification procedures. We defer the statement of these results and a discussion to Section 6.6.

6.1.2 Related Work

As discussed above, our work benefits from a long line of beautiful results on program checking, interactive proofs and cryptography. See the related work and discussion in Section 1.2.1.

Local Decoding. It is well known that for every (non-trivial) \((1/2 - \varepsilon, \ell)\)-locally-list-decodable code, it must hold that \(\ell = \Omega(1/\varepsilon^2)\) [Bli86, GV05] (in fact this bound holds even for standard, non-local, list decoding). Thus, aiming to stay within polynomial factors of the best possible information theoretic parameters, our primary goal is to understand the complexity of decoding \((1/2 - \varepsilon, \text{poly}(1/\varepsilon))\)-locally-list-decodable binary codes that have polynomial rate (i.e. where \(N(M) = \text{poly}(M)\)). We consider such codes to have “good” parameters (we elaborate on this choice below).

An explicit code with good parameters was given by Sudan, Trevisan and Vadhan [STV01]. The local-decoder for this code (namely the algorithm \(D\) as well as the circuits \(D_a\)) is in the complexity class \(NC^2\) (i.e. its depth is poly-logarithmic in its input length). Explicit codes with local-decoders in the (strictly lower) class \(AC^0\) (i.e. constant depth unbounded fan-in decoders) are also known [GL89]. However, these codes do not have good parameters.\(^4\) Specifically the Hadamard code has such a decoder [GL89], but its rate is exponential in \(M\).

\(^4\)We note that for non-binary codes, i.e. codes with large alphabets, one can construct codes with constant-depth local list-decoders and “good” parameters, see [GGH+07].
**Lower Bounds for Local List-Decoding.** The question of lower bounding the complexity of local-list-decoders was raised by Viola [Vio06]. He conjectured that \((1/2-\varepsilon, 2\varepsilon)\)-locally-list-decodable codes require computing majority over \(O(1/\varepsilon)\) bits, even when the list size \(\ell\) is exponential in \(1/\varepsilon\). Note that while exponential lists are not commonly considered in the coding setting (the focus instead is on polynomial or even optimal list sizes), they do remain interesting for applications to (non-uniform) worst-case to average-case hardness reductions. In particular, lower bounds for local-list-decoding with exponential lists, imply impossibility results for non-uniform black-box worst-case to average-case hardness reductions (see Section 6.6). In this paper we prove the conjecture for the case of sub-exponential size lists. While a proof of the full-blown conjecture remains elusive, there are results for other (incomparable) special cases:

Viola [Vio06] gave a proof (which he attributed to Madhu Sudan) of the conjecture for the special case of the standard non-local list-decoding setting. It is shown that a list-decoder from distance \(1/2-\varepsilon\) can be used to compute majority on \(\Theta(1/\varepsilon)\) bits, with only a small blow-up in the size and depth of the decoder. This result rules out, for example, constant-depth list-decoders whose size is \(\text{poly}(1/\varepsilon)\). Note, however, that in the non-local list decoding setting the size of the decoder is at least \(N\) (the codeword length) because it takes as input the entire (corrupted) codeword. This means that the bound on the size of constant-depth decoders does not have consequences for fairly large values of \(\varepsilon\). For example, when \(\varepsilon \geq 1/\log N\), the only implication that we get from [Vio06], is that there is a constant-depth circuit of size at least \(N \geq 2^{1/\varepsilon}\) that computes majority on instances of size \(1/\varepsilon\). But this is trivially true, and thus we do not get any contradiction. In the local-decoding setting the decoders’ circuits are much smaller and thus we can obtain limitations for much larger \(\varepsilon\)’s. Indeed in this paper we rule out constant-depth decoders for \((1/2-\varepsilon, \text{poly}(1/\varepsilon))\)-local-list-decoders for any \(\varepsilon\) smaller than \(1/\text{poly log log } N\) (and recall that this matches the construction of [GGH+07]).

Viola [Vio06] also proved that there are no constant-depth decoders (with polynomial-
size lists) for specific codes, such as the Hadamard and Reed-Muller codes. We, on the other hand, show that there are no such decoders for any code (regardless of the code’s rate, and even with sub-exponential list size).

Recently (independently of our work), Shaltiel and Viola [SV08] gave a beautiful proof of the conjecture for the local-decoding setting, with \( \ell \) exponential in \( 1/\varepsilon \), but for the special case that the decoder is restricted to have non-adaptive access to the received word. (I.e., they give a lower bound for decoders that make all their queries to the received word simultaneously.) Our result is incomparable to [SV08]: we prove Viola’s conjecture only for the case that \( \ell \) is sub-exponential in \( 1/\varepsilon \), but do so for any decoder, even an adaptive one. We emphasize that for important ranges of parameters the best codes known to be decodable in constant depth use adaptive decoders. In particular, the constant depth decoder of [GGH+07], as well as its improvement in this work, are adaptive. In light of this, it is even more important to show lower bounds for adaptive decoders.

6.1.3 On the Choice of Parameters

In this work binary codes with polynomial-rate are considered to have “good” parameters. Usually in the standard coding-theory literature, “good” codes are required to have constant rate.\(^6\) We note that, as far as we know, there are no known locally-decodable codes (both in the unique and list decoding settings) with constant rate (let alone codes that have both constant rate and have decoders that are in the low-level complexity classes that we consider here). The best binary locally decodable codes known have polynomial rate [STV01]. It is an interesting open question to find explicit codes with constant or even polylogarithmic rate.

Finally, we note that in this work we do not (explicitly) consider the query complexity of the decoder. The only bound on the number of queries the decoder makes to the received word comes from the bound on the size of the decoding circuit. The reason is that known codes with much smaller query complexity than the decoder size (in particular constant

\(^6\)We do remark that for applications such as worst-case to average-case reductions, polynomial or even quasi-polynomial rates suffice.
query complexity) have a very poor rate (see e.g. [Yek08]). Furthermore, there are negative results that suggest that local-decoding with small query complexity may require large rate [KT00, DJK+02, Oba02, KdWO4, WdWO05, GKST06].

6.2 Preliminaries

6.2.1 Locally Decodable and Locally List-Decodable Codes

The definition of locally list-decodable codes follows the formulation of Sudan, Trevisan and Vadhan [STV01] (with some small modifications). (Uniquely) locally decodable codes are defined as a special case (with list size 1).

Definition 6.2.1 (Locally decodable and locally list-decodable codes). Let $\Gamma$ be a finite alphabet. An ensemble of functions $\{C_M : \{0,1\}^M \to \Gamma^{N(M)}\}_{M \in \mathbb{N}}$ is a $(\delta(M), \ell(M))$-locally-list-decodable code, if there is an oracle Turing machine $D[\cdot, \cdot, \cdot, \cdot]$ that takes as input an index $i \in [M]$, an “advice” string $a \in [\ell(M)]$ and two random strings $r_1, r_2$ and the following holds: for every $y \in \Gamma^{N(M)}$ and $x \in \{0,1\}^M$ such that $\Delta_\Gamma(C_M(x), y) < \delta(M), \Pr_{r_1, r_2} \left[ \exists a \in [\ell] \text{ s.t. } \forall i \in [M] \Pr_{r_2} [D^y(a, i, r_1, r_2) = x[i]] > 3/4 \right] > 3/4$ (6.1)

If $|\Gamma| = 2$ we say that the code is binary. If $\ell = 1$ we say that the code is a (uniquely) locally decodable code. We say that the code is explicit if $C_M$ can be computed in time $\text{poly}(N(M))$.

Remark 6.2.2. One should think of the decoder’s procedure as having two stages: first it tosses coins $r_1$ and generates a sequence of $\ell$ circuits $\{F_a(\cdot, \cdot, \cdot)\}_{a \in [\ell]}$, where $F_a(i, r_2) = D(a, i, r_1, r_2)$. In the second stage, it uses the advice $a$ to pick the probabilistic circuit $F_a$ and use it (with randomness $r_1$) to decode the message symbol at index $i$. In [STV01] the

---

7 The length of these random strings lower-bounds $D$’s running time. Later in this work, when we consider $D$’s with bounded running time, the length of these random strings will also be bounded.
two-stage process is part of the definition, for us it is useful to encapsulate it in one machine (D).

In the sequel it will be convenient to simplify things by ignoring the first stage, and consider D as a probabilistic circuit (taking randomness r2) with two inputs: the advice a and the index to decode i, with the property that (always) for at least one \( a \in [\ell] \), \( D(a, \cdot) \) decodes correctly every bit of the message (with high probability over \( r_2 \)). Indeed if we hardwire any “good” \( r_1 \) (chosen in the first stage) into D then we are in this situation. This happens with probability at least \( 99/100 \). Thus in our proofs we will assume that this is the case, while (implicitly) adding \( 1/100 \) to the bound on the overall probability that the decoder errs. This simplification makes our proofs much clearer (since we do not have to deal with the extra randomness \( r_1 \)).

Remark 6.2.3 (Complexity of decoding). We often restrict the complexity of D. When we say, for example, that C is locally list-decodable (with the specified parameters) by \( \text{AC}^0 \) circuits, we mean that D (and all of the circuits \( \{F_a\} \)) are probabilistic \( \text{AC}^0 \) circuits of size \( \text{poly}(\log(N)/\varepsilon) \).

As a stepping stone in our constructions we will also use a relaxed variant: approximate locally-decodable codes, see Trevisan [Tre03].

Definition 6.2.4. [approximate locally list-decodable codes [Tre03]] Let \( \Gamma \) be a finite alphabet. We say that a code \( \{C_M : \{0,1\}^M \rightarrow \Gamma^{N(M)}\}_{M \in \mathbb{N}} \) is \( \delta \)-approximate \( (d, \ell) \)-locally-list-decodable, if it is the same as in Definition 6.2.1 with the following relaxation of (6.2.1):

\[
\Pr_{r_1} \left[ \exists a \in [\ell] \text{ s.t. } \Pr_{r_2 \in [M]} \left[ \Pr_{r_1}[D^\ell(a, i, r_1, r_2) = x[i]] > 9/10 \right] \geq 1 - \delta \right] > 99/100
\]

Less formally, in approximate codes the requirement is that for at least one advice string, the decoder decodes at least a \( 1 - \delta \) fraction of the bits of the message (but not necessarily all the bits of the message as in Definition 6.2.1). In our description of the decoders, we will use the two-stage process view of this definition as discussed in Remark 6.2.2.
6.2.2 Majority and Related Functions

We frequently use the two following results of Ajtai [Ajt93] (which are based on [AB84]).

**Lemma 6.2.5.** For all constants $c > 0$, there is a family of \( \mathcal{AC}^0 \) circuits that approximates the weight of \( x \) to within a factor of \( 1 \pm 1/\log^c(n) \): i.e., given \( x \in \{0, 1\}^n \), output a value \( \delta \) such that if the fraction of 1’s in \( x \) is \( \delta' \), then \( (1 - 1/\log^c(n))\delta' \leq \delta \leq (1 + 1/\log^c(n))\delta' \).

In particular,

**Lemma 6.2.6.** For all constants $c > 0$, there is a family of \( \mathcal{AC}^0 \) circuits that computes the following approximate majority promise problem: given \( x \in \{0, 1\}^n \), decide whether the fraction of 1’s in \( x \) is \( \delta' \) greater than \( 1/2 + 1/\log^c(n) \) or less than \( 1/2 - 1/\log^c(n) \) (and give arbitrary answer if none of the two is the case).

We use the promise problem \( \Pi \), defined in [Vio06] as follows:

\[
\Pi_{\text{yes}} = \{ x : x \in \{0, 1\}^{2k} \text{ for some } k \in \mathbb{N} \text{ and } \text{weight}(x) \leq k - 1 \} \\
\Pi_{\text{no}} = \{ x : x \in \{0, 1\}^{2k} \text{ for some } k \in \mathbb{N} \text{ and } \text{weight}(x) = k \}
\]

where \( \text{weight}(x) \) is the number of bits in \( x \) which are 1.

We will extensively use the fact, proven in [Vio06], that computing the promise problem \( \Pi \) on \( 2k \) bit inputs is (informally) “as hard” (in terms of circuit depth) as computing majority of \( 2k \) bits. This is stated formally in the claim below:

**Claim 6.2.7 ([Vio06]).** Let \( \{C\}_{M \in \mathbb{N}} \) be a circuit family of size \( S(M) \) and depth \( d(M) \) that solves the promise problem \( \Pi \) on inputs of size \( M \). Then, for every \( M \in \mathbb{N} \), there exists a circuit \( B_M \) of size \( \text{poly}(S(M)) \) and depth \( O(d(M)) \) that computes majority on \( M \) bits. The types of gates used by the \( B_M \) circuit are identical to those used by \( C_M \). E.g., if \( C_M \) is an \( \mathcal{AC}^0[q] \) circuit, then so is \( B_M \).
6.3 Transformations: Reducing the Complexity of Decoding

In this section we show how to transform locally decodable and locally list-decodable codes with expensive decoding (say in $\mathcal{NC}^1$), into ones that have constant depth $\mathcal{AC}^0$ decoders.

6.3.1 Unique Local Decoding

We first prove Theorem 1.4.1 (see Section 1.4), showing how to transform a locally (and uniquely) decodable binary code that has an $\mathcal{NC}^1$ decoder to a locally decodable code (with almost the same parameters) that has an $\mathcal{AC}^0$ decoder. See the next section for transformations on (uniquely and list) locally decodable codes over general alphabets.

Proof Intuition for Theorem 1.4.1. This application of our general approach is the simplest one. The idea is that the new encoding of $x \in \{0,1\}^M$ has two (equally long) parts appended together. The first part is the original encoding of $x$ and the second part is the truth-table of the randomized image $I$ of the $\mathcal{NC}^1$-complete language $L$ (given by Lemma 2.3.16) on instances of length $\log N$. Given a word that is close enough to a codeword in the new code, we know that both parts are close to what they should be. We then simulate in $\mathcal{AC}^0$ the original $\mathcal{NC}^1$ decoder (on the first half of the received word) as follows: every time we need to compute something (in $\mathcal{NC}^1$) that we cannot do in $\mathcal{AC}^0$, we reduce this computation to an instance $y$ of the language $L$. We then use the random instance reduction from $L$ to $I$ (computable in $\mathcal{NC}^0$) to compute this $y$ by querying the truth-table of $I$ (i.e. the second half of the received word) at a random location. Since the second half is a string that is close to the truth-table of $I$, with relatively high probability we compute $L(y)$ correctly. We can increase the success probability (in $\mathcal{AC}^0$) by repeating many times in parallel and taking the approximate majority of the answers (this can be done in $\mathcal{AC}^0$ by [AB84, Ajt93]). Thus with high probability, the new $\mathcal{AC}^0$ decoder decodes the first half exactly as the old $\mathcal{NC}^1$ decoder does.

A subtle issue that we need to deal with is the fact that the $I$-instances we reduce to
need to be of length exactly \( \log N \) (so that the two parts of the code are equally long). To that end we use the strong downwards self reducibility property of \( L \) (see Definition 2.3.14 and Lemma 2.3.16) to adjust the lengths of the instances we work with.

\[ \text{Proof. (of Theorem 1.4.1)} \] Let \( L \) be the \( \mathcal{NC}^1 \)-complete language from Lemma 2.3.16, and let \( I \) be its randomized image. Let \( x \in \{0, 1\}^M \). We define:

\[ C'(x)_i = C(x)_i, \text{ if } 1 \leq i \leq N \]

and

\[ C'(x)_i = I(i - N), \text{ if } N < i \leq 2N \]

That is, half of the codeword is the original codeword and the other half is the truth-table of \( I \) for input length \( t = \log N \) (we assume w.l.o.g. that \( N \) is a power of 2). The approach to decoding \( C' \) is to use the random instance reduction from \( L \) to \( I \) to correct the \( I \)-half of \( C'(x) \), and then to use the corrected \( I \)-half to help in decoding the half of \( C'(x) \) corresponding to \( C(x) \). Details follow.

We explain how to “compile” the \( \mathcal{NC}^1 \) decoder \( D \) for \( C \), to create the \( \mathcal{AC}^0 \) decoder \( D' \) for \( C' \) that has the same output as \( D \) (w.h.p.). The key component in \( D' \) is a probabilistic \( \mathcal{AC}^0 \) procedure \( A \), that tries to predict whether a given instance is in \( L \) or not, by querying the (possibly corrupted) codeword \( y \). Let \( (\mathcal{R}, \mathcal{E}) \) be the random instance reduction from \( L \) to \( I \). Recall from Definition 2.3.2, that \( \mathcal{R} \) is the randomizer that maps \( L \)-instances to random \( I \)-instances, and \( \mathcal{E} \) is the evaluator that computes the membership status of an \( L \)-instance given the membership status of the random \( I \)-instance it was mapped to (by \( \mathcal{R} \)). Let \( t' = t' \alpha \) (for some constant \( \alpha > 0 \)) be such that \( \mathcal{R} \) maps instances of length \( t' \) to instances of length \( t \). We now describe the algorithm \( A \):

\[ \text{Input : } a \in \{0, 1\}^{t'} \]

\[ \text{Output : A prediction for } L(a). \]
The algorithm:

1. Run $R$ on $a, d = \Theta(\log N)$ times in parallel and with independent random coins to obtain $(c_1, \tau_1), \ldots, (c_d, \tau_d)$, where $c_j \in \{0, 1\}^t$ and $\tau_j \in \{0, 1\}$.

2. Query (in parallel) $y$ at locations $c_1 + N, \ldots, c_d + N$ (where we view $c_j$ as a binary representation of an instance in $[N]$) to obtain the bits $b_1, \ldots, b_d$. For each $1 \leq j \leq d$, let $w_j = \mathcal{E}(b_j, \tau_j)$.

3. Run the $\mathcal{A}C^0$ procedure from Lemma 6.2.6 that computes the approximate majority on $w_1, \ldots, w_d$ and output the outcome of the procedure.

**Claim 6.3.1.** If $y$ is at distance at most $\delta/2$ from a codeword in $C'$ (where $\delta < 1/4$), then for every $a \in \{0, 1\}^t$, the probability that $A$ predicts correctly the membership of $a$ in $L$ is at least $1 - 1/N$.

**Proof.** Since $y$ is at distance at most $\delta/2$ from a codeword in $C'$, then if we consider only the $I$-half of $y$, it must be correct on at least a $1 - \delta > 3/4$ of the entries. By the properties of the random instance reduction, $c_1, \ldots, c_d$ are uniformly and independently distributed in the $I$-half of $y$. So for each $j \in [d]$, $\Pr[b_j = I(c_j)] \geq 3/4$. When $b_j = I(c_j)$, $w_j = L(a)$ (again by the properties of random instance reduction). By Chernoff bound, with probability at least $1/N$ at least $2/3$ of the $w_j$'s hold the correct prediction for $L(a)$. And then the approximate majority procedure gives the correct answer. 

Before describing $D'$, let us consider the computation of $D$ (given oracle access to a word $y' \in \{0, 1\}^N$). Let $s = \text{poly} \log(N)$ be the size of $D$ as an $\mathcal{NC}^1$ circuit. We can divide $D$’s computation as follows: first in parallel it computes the (bits of the) positions it is going to query $y'$, and then given the values of $y'$ in these locations it computes the output bit. Thus the execution of $D$ amounts to running at most $s$ different $\mathcal{NC}^1$ circuits, each of size at most $s$. Let $e_1, \ldots, e_k$ be the $\mathcal{NC}^1$ computations that compute the query locations, and $e_{k+1}$ computes the output bit. By the fact that $L$ is $\mathcal{NC}^1$-complete under $\mathcal{NC}^0$ reductions...
and it is strongly downwards self-reducible, we can solve every $e_i$ by a constant depth circuit making queries to instances of $L$ of length $(\log N)^a = t'$.

We are now ready to describe $D'$. It is given randomness for the circuit $D$ (and additional auxiliary randomness), as well as oracle access to a word $y \in \{0, 1\}^{2N}$. It then simulates the execution of $D$ when it is given oracle access to the first half of $y$ (which we should think about as the original received word). As explained above, this computation can be divided to the $NC^1$ computations $e_1, \ldots, e_{k+1}$. In parallel for each $e_i (1 \leq i \leq k)$ $D'$ runs the constant-depth circuit that computes $e_i$ by making queries to instances of $L$ of length $t'$. This circuit makes $\text{poly} \log (N)$ queries to $L$ on instances of length $t'$. On each one of these queries, run the procedure $A$ and use its answer as the oracle answer in the strong downwards self-reducibility circuit. If $A$ does not make mistakes, then $D'$ holds at this stage the query positions that $D$ would have made to the original word in $\{0, 1\}^N$. So $D'$ makes the queries in the same locations $D$ makes them (in the first half of $y$). After making all these queries, $D'$ holds the answers that $D$ would have read from the original word. Given these values and the randomness, $D'$ can compute $e_{k+1}$ in the same way it computed $e_1, \ldots, e_k$. If $A$ does not make a mistake, then $D'$ holds at this stage the output of $D$ which we also define to be the output of $D'$.

We conclude that $D'$ does exactly the same as $D$ as long as $A$ does not make any mistake during the execution. By claim 6.3.1, $A$ makes a mistake with probability at most $1/N$. Since $D'$ runs $A$ at most $\text{poly} \log (N)$ times, by the union bound, with probability almost 1, $D'$ outputs the same output as $D$. Now, since $y$ is at most $\delta/2$-far from a codeword in $C'$, its first half is at most $\delta$-far from a codeword in $C$. Therefore with high probability $D$, and hence $D'$, retrieves the correct value of the message in the given location. By construction $D'$ is an $AC^0$ circuit of size $\text{poly} \log (N)$.

6.3.2 Local List Decoding

In this section we prove Theorem 1.4.2, showing how to transform a locally list-decodable code that has an $NC^1$ decoder to a locally list-decodable code that has an $AC^0$ decoder. In
particular, this theorem also implies a similar transformation for (uniquely) locally decodable codes with non-binary alphabets.

Proof Intuition for Theorem 1.4.2. The approach that we used to prove Theorem 1.4.1 cannot be used to recover from distance more than 1/4 (even if we start with a code that can be list-decoded from a large distance), because we need the truth-table part of the word to be more than 1/2-close to the respective half in the codeword. One thing we can do to improve the distance that we can recover from, at the price of doubling the alphabet size, is to append the truth-table component-wise. That is, we append to each bit of the original codeword an entry of the truth-table (recall that they are of the same length). This allows us to list-decode from distance $1/2 - 1/\log \log(N)$ (assuming the original code has a locally list-decoder in $NC^1$), and the alphabet size is 4 (two bits per symbol). However, to recover from distance more than 1/2 requires a different technique. The idea now is to append (again, componentwise) the direct-product of $I$’s truth table ($I$ is, as before, the randomized image of the $NC^1$-complete language $L$ from Lemma 2.3.16). That is, every symbol in the new encoding contains a symbol from the old encoding concatenated with the binary string $I(i_1), \ldots, I(i_s)$, where $(i_1, \ldots, i_s)$ is a tuple of binary strings of some length that is determined in the proof. For every possible tuple we will have a different entry in the codeword.

As in the proof of Theorem 1.4.1, the new decoder simulates the old decoder. When it needs to compute some $NC^1$ computation, it creates a uniformly distributed s-tuple where in each entry it either (with probability 1/2) puts a random instance for which it knows its correct membership status in $I$ (this can be done by using the solved instance generator for $I$, see Definition 2.3.1), or (with probability 1/2) puts a random instance from which it can conclude the value of the $NC^1$ computation given the correct membership status of that instance in $I$ (this can be done using the random instance reduction from $L$ to $I$ and the fact that $L$ is $NC^1$-complete). The new decoder now reads from the corresponding location in the received word the (possibly corrupted) membership status of every instance in the s-tuple. It then checks whether on the entries for which it knows the correct answer, the received
word is correct, and that all the other (randomized) instances evaluate back to the same answer. If not, it declares this location in the received word to be corrupted. Otherwise it is a good indication that the symbol is not corrupted. The decoder assumes that the values it reads from it are all correct and infers from them the correct value of the $\mathcal{NC}^1$ computation. Given this procedure we can continue the simulation as in the proof of Theorem 1.4.1.

A subtle issue is the fact that the length of the original codeword and the length of the extra information we append to it (i.e. the number of $s$-tuples) are not necessarily the same. To solve this, we write many copies of the original codeword in the new one, so that the repeated original codeword is of the same length as the number of $s$-tuples. We proceed with the proof.

Proof. (Of Theorem 1.4.2) We start by describing the code. Let $s = s(N, \varepsilon)$ and $t = t(N, \varepsilon)$ be two integers that will be determined later. Consider all the $s$-tuples $(i_1, \ldots, i_s)$ where $i_j \in \{0, 1\}^t$ (for every $1 \leq j \leq s$). We can assume, and will indeed make sure by our choice of $s$ and $t$, that $2^t/N$ is an integer (and we assume without loss of generality that $N$ is a power of 2). Let $L$ be the $\mathcal{NC}^1$-complete language from Lemma 2.3.16 and let $I$ be its randomized image. Let $t' = \text{poly}(t)$ be such that the random instance reduction from $L$ to $I$ on $L$-instances of length $t'$ generates $I$-instances of length $t$.

Let $x \in \Sigma^M$, and let us define the codeword $C'(x)$. Every entry in this codeword is indexed by a different $s$-tuple as above. Thus the length $N'$ of $C'(x)$ is $2^st$. Let $F$ be a (natural) bijection from $s$-tuples to $[2^st]$. Let $(i_1, \ldots, i_s)$ be an $s$-tuple and $k = (F(i_1, \ldots, i_s) \mod N)$. The entry in $C'(x)$ indexed by $(i_1, \ldots, i_s)$ is defined to be:

$$C'(x)_{i_1, \ldots, i_s} = C(X)_k \circ (I(i_1), \ldots, I(i_s))$$

where $\circ$ denotes concatenation of strings. That is, every entry contains two parts, one is an entry in the original encoding of $x$, and one is the $s$-bit characteristic vector of $i_1, \ldots, i_s$ as instances of the language $I$. The original encoding is repeated $2^st/N$ times in the new encoding. It is easy to see that if $C$ is explicit then so is $C'$. Also note that every symbol in
The key component in $D'$ is a probabilistic $\mathcal{AC}^0$ procedure that tries to learn whether a given instance is in $L$ or not, by querying the (possibly corrupted) word $y$. We start by describing a procedure $A_1$ that makes one query to the codeword and has a very low probability of success, we later show how to amplify its success probability. $A_1$ either outputs a prediction regarding the membership of the given instance in $L$ or declares the location in which it queries the membership of the given instance to be corrupted:

**Input**: $a \in \{0,1\}^t$

**Output**: A prediction for $L(a)$ or “corrupted”.

**The algorithm:**

1. For each $i \in [s]$ (in parallel and with independent random coins), do the following:
   Choose uniformly $v_i \in \{0,1\}$. If $v_i = 0$, run the $\mathcal{NC}^0$ solved instance generator for $I$ on input length $t$, to obtain a pair $(c_i, y_i)$ where $y_i \in \{0,1\}$ and $c_i \in \{0,1\}^t$. If $v_i = 1$, run on $a$ the $\mathcal{NC}^0$ random instance reduction from $L$ to $I$, to obtain a pair $(c_i, \tau_i)$ where $\tau_i \in \{0,1\}$ and $c_i \in \{0,1\}^t$.

2. Query the codeword $y$ at the position indexed by the $s$-tuple $(c_1, \ldots, c_s)$, to retrieve the symbol $\sigma \circ (b_1, \ldots, b_s)$ (where $\sigma \in \Sigma$ and $b_j \in \{0,1\}$).

3. For every $i \in [s]$ for which $v_i = 0$, check that $b_i = y_i$. If this equality does not hold (for at least one of these $i$'s) declare the position indexed by $(c_1, \ldots, c_s)$ to be corrupted and abort.
4. For every \( i \in [s] \) for which \( v_i = 1 \), compute the value \( w_i = \mathcal{E}(b_i, \tau_i) \) (recall that \( \mathcal{E} \) is the evaluator in the random instance reduction from \( L \) to \( I \)), and check that for all these \( i \)'s the \( w_i \)'s are equal. If they are, set \( w = w_i \). Otherwise, declare the position indexed by \( (c_1, \ldots, c_s) \) to be corrupted and abort.

5. Output \( w \).

It is easy to verify that \( A_1 \) is in \( \mathcal{AC}^0 \). We now give bounds on the probabilities that it gives correct and wrong predictions.

**Claim 6.3.2.** If \( y \) is \( \varepsilon \)-close to a codeword in \( C' \), then for every \( a \in \{0, 1\}^t' \), the probability that \( A_1 \) predicts correctly the membership of \( a \) in \( L \) is at least \( \varepsilon - 2^{-s} \).

**Proof.** By the definitions of solved instance generator and random instance reduction, \( c_1, \ldots, c_s \) are independently and uniformly distributed. Thus we query \( y \) at a random location (uniformly distributed). Since \( y \) is \( \varepsilon \)-close to a codeword, with probability at least \( \varepsilon \) we query a non-corrupted location. When this happens, we get the correct prediction as long as not all the \( v_i \)'s are 0 (which happens with probability \( 2^{-s} \)). The reason is that \( b_1, \ldots, b_s \) are the true (0/1) membership values of the instances \( c_1, \ldots, c_s \) in \( I \). By the definition of solved instance generator, this implies that with probability 1, for every \( i \) for which \( v_i = 0 \), \( b_i = y_i \). And by the definition of random instance reduction, for every \( i \) for which \( v_i = 1 \), \( w_i = L(a) \).

**Claim 6.3.3.** For every \( a \in \{0, 1\}^t' \) (and every \( y \)), the probability that \( A_1 \) gives the wrong prediction regarding the membership of \( a \) in \( L \) (i.e. says that it is in \( L \) when it is not or vice versa) is at most \( 2^{-s} \).

**Proof.** By the definitions of solved instance generator and random instance reduction, \( c_1, \ldots, c_s \) are independently and uniformly distributed, and they are independent of the \( v_i \)'s. Now suppose \( A_1 \) does not declare location \( (c_1, \ldots, c_s) \) to be corrupted while giving a wrong prediction regarding the membership of \( a \) in \( L \). This can only happen if \( b_i = y_i = I(c_i) \) for every \( i \) for which \( v_i = 0 \) (by the definition of solved instance generator), and \( b_i \neq I(c_i) \) for every \( i \) for which \( v_i = 1 \).
which \( v_i = 1 \) (for otherwise, by the definition of random instance reduction, \( w_i \) will be equal to \( L(a) \) for at least one of these \( i \)'s). In other words, location \((c_1, \ldots, c_s)\) must be corrupted in such a way that exactly the bits indexed by \( i \)'s for which \( v_i = 1 \) are flipped while the rest stay intact. Or, equivalently, every corruption pattern has exactly one choice of \( v_i \)'s that will cause \( A_1 \) to give the wrong prediction (and not to detect the corruption). Since the \( v_i \)'s are independent of \((c_1, \ldots, c_s)\), we conclude that the probability that this event happens is \( 2^{-s} \).

We now amplify the success probability of \( A_1 \). Consider the following \( \mathcal{AC}^0 \) procedure \( A_2 \):

**Input** : \( a \in \{0,1\}^t' \)

**Output** : A prediction for \( L(a) \).

**The algorithm:**

1. Run \( d/\varepsilon \) times in parallel and with independent random coins the algorithm \( A_1 \) on input \( a \) (\( d \) will be determined later).

2. Look at the string of length \( d/\varepsilon \) over \( \{0,1,*\} \) of the answers \( A_1 \) gives in the previous step (where * stands for “corrupted”). Run the procedure from Lemma 6.2.5 on that string, to estimate the number of 0's and 1's in the string. I.e. first change every * to 1 and estimate the number of 0's and then change every * to 0 and estimate the number of 1's.

3. Output the value that gets the higher estimate.

Next we want to prove that \( A_2 \) gives the correct prediction with very high probability. This is true if we can ensure that (with high probability) the ratio between right and wrong predictions of \( A_1 \) is large (and then the procedure from Lemma 6.2.5 gives the right prediction). To that end we set \( s \) to be \( \log(1/\varepsilon) + 2 \), and prove:

**Claim 6.3.4.** If \( y \) is \( \varepsilon \)-close to a codeword in \( C' \), then for every \( a \in \{0,1\}^t' \), the probability that \( A_2 \) predicts correctly the membership of \( a \) in \( L \) is at least \( 1 - 2^{-\Omega(d)} \).
Proof. Let \( z \) be the string of length \( d/\varepsilon \) over \( \{0, 1, *\} \) of \( A_1 \)'s answers. Assume w.l.o.g. that \( a \in L \) (otherwise replace 1's with 0's in the argument below). By Claims 6.3.2, 6.3.3 and our choice of \( s \), the expected number of 1's in \( z \) is at least \( (d/\varepsilon)(\varepsilon - 2^{-s}) = 3d/4 \), and the expected number of 0's is at most \( (d/\varepsilon)2^{-s} = d/4 \). By Chenoff bound, with probability at least \( 2^{-\Omega(d)} \), the actual number of 1's will be at least \( 2d/3 \) while the number of 0's at most \( d/3 \). The \( \mathcal{AC}^0 \) procedure from Lemma 6.2.5 estimates the number of 1's and 0's upto 0.01 (multiplicative) precision. This is enough for \( A_2 \) to conclude correctly that \( a \in L \).

Before we describe \( D' \), let us consider the computation of \( D \) (given oracle access to a word \( y \in \Sigma^N \)). Let \( w = \text{poly}(\log(N)/\varepsilon) \) be the size of \( D \) as an \( \mathcal{NC}^1 \) circuit. As in the proof of Theorem 1.4.1, we can divide \( D \)'s computation to the \( \mathcal{NC}^1 \) computations, \( e_1, \ldots, e_K \), that compute the query locations. And the \( \mathcal{NC}^1 \) computations, \( e_{K+1}, \ldots, e_w \), that compute the output bits (unlike the proof of Theorem 1.4.1, now \( D \) outputs many bits which are the descriptions of the \( M_j \)'s). Each \( e_i \) is of size at most \( w \). By the fact that \( L \) is \( \mathcal{NC}^1 \)-complete and it is strongly downwards self-reducible, we can solve every \( e_i \) by a constant depth circuit making queries to instances of \( L \) of length \( (\log(N)/\varepsilon)^\alpha \) for an arbitrarily small constant \( \alpha > 0 \). We set \( t' \) to be \( (\log(N)/\varepsilon)^\alpha \), and set \( \alpha \) such that \( t = (\log(N)/\varepsilon)^\delta(\log(1/\varepsilon) + 2)^{-1} \) where \( \delta \) is the arbitrarily small constant from the statement of the theorem. Recall that \( t \) and \( t' \) are polynomially related, so there is a setting of the parameters that achieve this. Also in our choice of parameters, we make sure that \( 2^{st}/N \) is an integer (we have enough freedom in our choice of \( t \) to do this).

We are now ready to describe \( D' \). It is given randomness for the circuit \( D \) (and additional auxiliary randomness). Then in parallel for each \( e_i \) (\( 1 \leq i \leq K \)) it runs the constant-depth circuit that computes \( L \) at length \( w \) by querying on instances of length \( t' \) (via strong downwards self-reducibility). This circuit makes \( \text{poly}(\log(N)/\varepsilon) \) queries to \( L \) on smaller instances. On each one of these queries, run the procedure \( A_2 \) and use its answer as the oracle answer. If \( A_2 \) does not make mistakes, then \( D' \) holds at this stage the query positions that \( D \) would have made to the original codeword in \( \Sigma^N \). For each such position, \( D' \) makes the query in a random copy of the original codeword. That is, it chooses \( p \in_R [2^{st}/N] \).
and queries \( y \) in the position \( p \) plus the address computed for the original codeword. After making all these queries, \( D' \) holds the list of \( \Sigma \)-symbols that \( D \) would have read from the original codeword (At this stage \( D' \) ignores the \( s \) bits appended to each \( \Sigma \)-symbol). Given these values and the randomness, \( D' \) can compute \( e_{K+1}, \ldots, e_t \) in the same way it computed \( e_1, \ldots, e_K \). If \( A_2 \) does not make mistakes, then \( D' \) holds at this stage the output of \( D \) which we also define to be the output of \( D' \).

We conclude that \( D' \) does exactly the same as \( D \) as long as \( A_2 \) does not make any mistake during the execution. The number of times we invoke \( A_2 \) is \( \ell = \text{poly}(\log(N)/\varepsilon) \). We set \( d \) (the parameter from the description of \( A_2 \)) to be \( \Theta(\log(\log(N)/\varepsilon)) \) with a leading constant that ensures that the probability (specified in Claim 6.3.3) that \( A_2 \) makes an error is at most \( \ell^{-1}/100 \). So the probability that \( D' \) fails to decode correctly is bounded by the error probability of \( D \) (at most 1/4), plus the probability that \( A_2 \) makes an error (at most 1/100 by union bound). By running \( D' \) twice with independent random coins we can drive the error down to below 1/4. If the code was uniquely decodable it remains uniquely decodable. If it was locally list-decodable (with \( \ell > 1 \)) then this incurs a price of doubling the list size.

It is easy to verify that \( D' \) can be implemented by \( \mathcal{AC}^0 \) circuits of size \( \text{poly}(\log(N)/\varepsilon) \).

Going back to the parameters of the code we see that the length \( N' \) of a codeword is

\[
2^{st} = 2^{\log(1/\varepsilon+2)(\log(N)/\varepsilon)^s(\log(1/\varepsilon)+2)^{-1}} = 2^{(\log(N)/\varepsilon)^s}
\]

and the size of the alphabet \( \Gamma \) is \( |\Sigma| \cdot 2^s = |\Sigma| \cdot O(1/\varepsilon) \).

### 6.4 Explicit Constructions

In this section we apply our general theorems from the previous sections to construct codes with \( \mathcal{AC}^0 \) local-decoders. We do that by first constructing codes with \( \mathcal{NC}^1 \) decoders and then applying the general transformations to them. Previous locally-decodable codes with the parameters that we need are not known to be in \( \mathcal{NC}^1 \) (in particular decoding the code given in [STV01] involves solving a system of linear equations). We therefore construct new
explicit codes with $\mathcal{NC}^1$ decoders, and then apply our transformations to them.

### 6.4.1 A (Uniquely) Locally Decodable Binary Code

We prove theorem 1.4.3, constructing a binary code that can be (uniquely) locally-decoded from a constant distance.

**Proof of Theorem 1.4.3.** In order to apply Theorem 1.4.1 we need to show an explicit code that can be non-adaptively locally decoded from distance $2/25$ by $\mathcal{NC}^1$ circuits of polylogarithmic size. To the best of our knowledge the only known locally-decodable code with the parameters that we need is the so called low-degree extension code concatenated with Hadamard [STV01]. Unfortunately we do not know how to implement the decoder for this code in $\mathcal{NC}^1$. The reason is that it involves decoding the Reed-Solomon code which is done via solving a system of linear equations. We therefore use a combination of known constructions and techniques to construct a new code that does have an $\mathcal{NC}^1$ local decoder (and which avoids decoding the Reed-Solomon code). Our construction has three stages.

**Stage 1 - Decoding the low-degree extension from small distances.** The first stage shows how to decode the low-degree extension code from a very small (sub-constant) relative distance.

Given a string $x \in \{0, 1\}^M$ we associate with it a multi-variate polynomial over a finite field $\mathbb{F}$ as follows: fix $\mathbb{F}$ to be a field of cardinality $(\log M)^2$. Fix a subset $H$ of the field of cardinality $\log M$. Let $m$ be $(\log M)/(\log |H|)$. Let $b : [M] \rightarrow H^m$ be an injective map. To encode $x$, we find the unique $m$-variate polynomial $\tilde{x} : \mathbb{F}^m \rightarrow \mathbb{F}$ of degree at most $|H| - 1$ in each of the $m$ variables, such that for every $i \in [M]$, $\tilde{x}(b(i)) = x_i$. Such a polynomial can be efficiently found by means of (multi-variate) interpolation. The encoding of $x$ is the evaluation of $\tilde{x}$ on every point in $\mathbb{F}^m$. Let us call this code $C_1$ and it maps $\{0, 1\}^M$ to $\mathbb{F}^{N_1}$, where $N_1 = |\mathbb{F}|^m$. Note that $N_1 = M^2$, this is because,

$$\log N_1 = m \log |\mathbb{F}| = \frac{(\log M)(\log |\mathbb{F}|)}{\log |H|} = 2 \log M$$
We now explain how to locally-decode $C_1$ from distance $\delta_1 = 1/(10|\mathbb{F}|) = 1/(10(\log M)^2)$. We are given a string $y \in \mathbb{F}^m$ such that there exist a string $x \in \{0,1\}^M$ that satisfies $\Delta(C_1(x), y) \leq 1/(10|\mathbb{F}|)$, and an index $i \in [M]$. We associate $i$ (via $b$) with a point in $H^m \subseteq \mathbb{F}^m$ on which we want to evaluate the polynomial $\bar{x}$. We choose a random line in $\mathbb{F}^m$ that passes through $b(i)$ (i.e. for $h \in \mathbb{F}^m$, we look at the line $\{b(i) + hz : z \in \mathbb{F}\}$), and we read from $y$ the $\mathbb{F}$-values associated with the locations of all the $|\mathbb{F}|$ points on that line. By pairwise independence, every point on the line in itself is uniformly distributed (independent of $b(i)$), so by the union bound, with probability at least $9/10$, all the values that we read agree with $\bar{x}$ restricted to the line. Note that the restriction to the line is a univariate polynomial over $\mathbb{F}$ of degree

$$m(|H| - 1) \leq (\log M)^2 / \log \log M < |\mathbb{F}|$$

So by interpolation we can retrieve $\bar{x}(b(i))$. By using Lagrange’s formula and [HV06], this non-adaptive decoder can be implemented by $\mathcal{NC}^1$ circuits of size $\text{poly}|\mathbb{F}| = \text{poly}(\log M)$.

**Stage 2 - Boosting the distance with expanders.** We now show how to obtain from $C_1$ a code with large alphabet that can be decoded from a constant distance. To that end we use the expanders based technique of [ABN+92]. Specifically, let $G = (L, R, E)$ be a $d$-regular bipartite graph where $|L| = |R| = N_1$, and $G$ has the property that for every $B \subseteq R$ with $|B| \leq 2N_1/5$, there are at most $\delta_1N_1$ vertices $v \in L$, such $|\Gamma(v) \cap B| \geq d/2$ (where $\Gamma(v)$ is the set of neighbors of $v$).

**Claim 6.4.1.** There exist a family of graphs $G$ with the specified parameters, where $d = \text{poly}(1/\delta_1)$, and given a name of a vertex $v$ in $G$, the list of its $d$ neighbors can be computed by $\mathcal{NC}^1$ circuits of size $\text{poly}(\log N_1, 1/\delta_1)$.

**Proof.** Let $H$ be the regular constant-degree Gabber-Galil expander on $N_1$ nodes [GG81]. We take $G$ to be the bipartite $c \log(1/\delta_1)$-th power of $H$ for some constant $c$ that will be chosen later: specifically, we join $x \in L$ and $y \in R$ by an edge for each walk of length $c \log(1/\delta_1)$ from
x to y in H. Since H has constant degree, the degree of G is \( d = O(1^{\log(1/\delta)}) = (1/\delta)^{O(c)} \). Moreover, it has been shown [GG81] that H has (normalized) second-largest eigenvalue \( \lambda = 1 - \Omega(1) \) and it is well known that \( \lambda(H^k) = \lambda(H)^k \) for regular undirected graphs H. Therefore, our graph G has \( \lambda(G) = (1 - \Omega(1))^{\log(1/\delta)} = \delta_1^{\Theta(c)} \).

We now prove that G has the required decoding property. Fix a subset \( B \subseteq R \) with \(|B| \leq 2N_1/5\) and let \( A \) be the set of vertices \( v \in L \), such that \(|\Gamma(v) \cap B| \geq d/2\) (where \( \Gamma(v) \) is the set of neighbors of \( v \)). We wish to show that \(|A| \leq \delta_1 N_1 \). By the expander mixing lemma (cf. [AS00]), we have:

\[
\left| E(A, B) - d \cdot \frac{|A| \cdot |B|}{N_1} \right| \leq d \lambda \sqrt{|A| \cdot |B|}.
\]

By the definition of \( A \) and \( B \), \( E(A, B) \geq \frac{d}{2} \cdot |A| \), and so we have:

\[
\frac{d}{2} \cdot |A| - d \cdot \frac{|A| \cdot |B|}{N_1} \leq d \lambda \sqrt{|A| \cdot |B|},
\]

which implies that \(|A| \leq \lambda^2 \cdot |B|/(1/2 - |B|/N_1)^2 \leq 40 \cdot \lambda^2 \cdot N_1 \) (where the last inequality follows from the assumption that \(|B| \leq 2N_1/5\)). Since \( \lambda = \delta_1^{\Theta(c)} \), we have \(|A| \leq O(\delta_1^{\Theta(c)})N_1 \leq \delta_1 N_1 \) by an appropriate choice of the constant \( c \).

Furthermore, it has been shown [GV04] that walks of length \( \ell \) on the \( N \)-node Gabber-Galil expander can be computed by \( \mathcal{AC}^0 \) circuits of size \( \text{poly}(\log N, 2^\ell) \), so the neighbors of a given node in G can actually be computed by \( \mathcal{AC}^0 \) circuits of size \( \text{poly}(\log N_1, 1/\delta_1) \) (and therefore, by \( \mathcal{NC}^1 \) circuits of the same size).

We define the following code \( C_2 : \mathbb{F}^{N_1} \rightarrow (\mathbb{F}^d)^{N_1} \):

\[
C_2(x)_i = x_{\Gamma_1(i)}, x_{\Gamma_2(i)}, \ldots, x_{\Gamma_d(i)}
\]

where \( \Gamma_j(i) \) is the \( j \)’th neighbor of vertex \( i \in R \).

We present a local-decoder \( D_2 \) for \( C_2 \) that given a string \( y \in (\mathbb{F}^d)^{N_1} \) for which there exist \( x \in \mathbb{F}^{N_1} \) that satisfies \( \Delta(y, C_2(x)) \leq 2N_1/5 \), the decoder computes every entry in \( x \) correctly.
except for a \(\delta_1\) fraction of the entries. The decoder is as follows: Given oracle access to \(y\) as above and an index \(i \in N_1\), we think of \(i\) as a left vertex in \(G\). The decoder computes the list of right neighbors \(j_1, \ldots, j_d\) of \(i\). It then query \(y\) in positions \(j_1, \ldots, j_d\), to retrieve \(d\) \(d\)-tuples of \(F\)-symbols. Each such tuple contains a prediction of \(x_i\). That is, if \(i\) is the \(k\)'th neighbor of \(j_1\) then the \(k\)'th entry in the \(d\)-tuple that we read from location \(j_1\) in \(y\) is \(x_i\) (supposedly, if \(j\) is not a corrupted location). The decoder takes \(x_i\) to be the prediction that appears more often than others in the \(d\) queries (in fact taking the majority is enough). Note that \(D_2\) can be implemented by \(\mathcal{NC}^1\) circuits of size \(\text{poly}(\log N_1, d) = \text{poly}(\log N_1, 1/\delta_1)\), this is because the neighbors in \(G\) can be computed in \(\mathcal{NC}^1\) by Claim 6.4.1, and computing the plurality amounts to comparing in parallel all the \(d\) predictions and counting (in \(\mathcal{NC}^1\)) which one appears most often. We now prove correctness.

**Claim 6.4.2.** \(\Pr[D_2^d(i) = x_i] \geq 1 - \delta_1\)

**Proof.** Let \(B \subseteq R\) be the set of right vertices that are associated with the corrupted locations in \(y\). By the hypothesis, \(|B| \leq 2N_1/5\). Let \(A \subseteq L\) be the set of vertices \(v \in L\) such that \(|\Gamma(v) \cap B| > d/2\). By the properties of \(G\), \(|A| \leq \delta_1 N_1\). For every \(i \not\in A\), the majority of \(y r_1(i), \ldots, y r_d(i)\) are not corrupted, and therefore will give the right prediction for \(x_i\).

Combining the codes \(C_1\) and \(C_2\), we get a code \(C' : \{0, 1\}^M \rightarrow (\mathbb{F}^d)^N\) that can be non-adaptively locally-decoded from a constant distance by \(\mathcal{NC}^1\) circuits. On index \(i \in [M]\), the decoder \(D'\) for \(C'\), first runs the decoder \(D_1\) for \(C_1\) to produce the query locations. These locations are then passed to the decoder \(D_2\) for \(C_2\). \(D_2\) retrieves the values in the requested locations and passes them back to \(D_1\), who then computes the \(i\)'th bit of the message. \(D'\) is in \(\mathcal{NC}^1\) because \(D_1\) and \(D_2\) are (and all the queries are done in parallel). Informally, it corrects from distance \(2/5\) because \(D_2\) corrects all but \(\delta_1\) fraction of errors, and \(D_1\) corrects the remaining \(\delta_1\).

**Stage 3 - Back to binary code.** The code \(C'\) can be locally-decoded from a constant distance, however the alphabet of the code is not binary but rather contains \(|\mathbb{F}|^d = 252\).
(log M)^{poly log M} symbols. To get back to a binary code we use the technique of concatenating codes. More precisely, note that every symbol in $\mathbb{F}^d$ has a description (by bits) of size $poly log M$. We encode every symbol separately by a binary code of polynomial rate. Thus the size of the new codeword grows only by a polylogarithmic factor. The code that we are going to use is the low-degree extension code concatenated with Hadamard [STV01]. Recall that previously we said that we do not know of an $NC^1$ decoder for this code, however now we need to encode and decode strings of exponentially smaller length (i.e. instead of length $M$, length $poly log M$), and this can easily be done by $NC^1$ circuits of size $poly log M$.

Let $M' = d \log |\mathbb{F}|$, consider the binary code $C_3 : \{0, 1\}^{M'} \rightarrow \{0, 1\}^{N'}$ given in [STV01] (i.e. the low-degree extension code concatenated with Hadamard), that has $N' = poly(M')$ and can be decoded from distance 1/5 (here we do not even need the code to be locally-decodable, i.e. the circuit of size $poly log M$ can read the whole $(poly log M)$-bit codeword).

**Claim 6.4.3.** $C_3$ can be decoded from distance 1/5 by $NC^1$ circuits of size $poly log(M)$.

We define the concatenated code $C : \{0, 1\}^M \rightarrow \{0, 1\}^{N'.N_1}$ as follows. Let $f : [N'.N_1] \rightarrow [N'] \times [N_1]$ be a bijection. Let $i \in [N'.N_1]$, and $f(i) = (j, k)$. We define,

$$C(x)_i = C_3(C'(x)_k)_j$$

Note that $N' = poly(d \log |\mathbb{F}|) = poly log M$, and $N_1 = M^2$. So $C$ maps $M$ bits to $poly(M)$ bits. We now describe the local decoder $D$ for $C$. Given oracle access to a string $y$ that is 2/25 close to a codeword, and $i \in [M]$, we first run the decoder $D'$ for $C'$ to compute the query locations. For every such query, we read the $N'$ bits that encodes (via $C_3$) the relevant symbol in the code $C'$. We run the decoder $D_3$ for $C_3$ to retrieve the symbol in the code $C'$. We then pass these symbols to $D'$ who returns the $i$'th bit of the message. Clearly, $D$ can be implemented by $NC^1$ circuits of size $poly log M$, because $D'$ and $D_3$ can (and all the queries are done in parallel). We now prove correctness.

**Claim 6.4.4.** $D$ decodes $C$ from distance 2/25.

**Proof.** Let $y \in \{0, 1\}^{N'.N_1}$ be a string such that there is $x \in \{0, 1\}^M$ satisfying $\Delta(y, C(x)) \leq 2/25$. Then...
We think of the locations of bits in $y$ as indexed by $(j, k) \in [N'] \times [N_1]$ (as in the definition of $C$). Let $B \subseteq [N_1]$ be such that for every $k \in B$,

$$\Pr_{j \in [N']} [y_{jk} \text{ is corrupted}] \geq 1/5$$

By Markov inequality,

$$\Pr_{k \in [N_1]} [k \in B] \leq 2/5$$

Since the decoder $D_3$ for $C_3$ decodes from distance 1/5, it can decode correctly at least 3/5 of the symbols (those that their indices are not in $B$) to obtain a string $y' \in (\mathbb{R}^d)^{N_1}$ satisfying $\Delta(C'(x), y') \leq 2/5$. The decoder $D'$ for $C'$ can decode from distance 2/5, and therefore can retrieve (w.h.p.) every bit in the message $x$.

Finally, we apply on the code $C$ the transformation given in Theorem 1.4.1, to obtain an explicit binary code that maps $M$ bits to poly($M$) bits, and can be locally-decoded from distance 1/25 by probabilistic $\mathcal{AC}^0$ circuits of size poly(log $M$).

### 6.4.2 A List-Locally Decodable Non-Binary Code

Next we prove Theorem 6.1.1, showing how to explicitly construct a code with large alphabet that can be locally list decoded from very large distances by $\mathcal{AC}^0$ circuits. This code was originally presented in [GGH+07].

**Proof of Theorem 6.1.1.** As in the proof of Theorem 1.4.3, we first show a code with similar parameters that can be locally list-decoded by an $\mathcal{AC}^1$ decoder, and then we apply Theorem 1.4.2. Our code is given by applying the “direct product” construction of Impagliazzo et. al. [IJK06] on the code from Theorem 1.4.3. We start with some definitions. Recall the definition of an approximate locally list-decodable code [Tre03] (Definition 6.2.4). The following definition appears in [IJK06]:

**Definition 6.4.5.** [direct product codes] Let $C : \{0, 1\}^M \rightarrow \{0, 1\}^N$ be a code. The $k$-direct-product of $C$ is the code $C^k : \{0, 1\}^M \rightarrow (\{0, 1\}^k)^N$, defined as follows (for $x \in \{0, 1\}^M$ and
So far, to keep the statements and proofs simple, we only considered non-adaptive decoders. In order to use [IJKO6] we need to consider adaptive decoders.

**Definition 6.4.6.** We say that a decoder is k-adaptive (or has adaptivity k), if the queries that it makes to the received word can be partitioned into k sets (layers) where all the queries in layer i can be done simultaneously (given the answers to the queries from layers $j < i$).

It is not difficult to see that Theorems 1.4.1 and 1.4.2 can be generalized to decoders with constant adaptivity. This is done by applying the same arguments, from one layer to the next. Since the number of layers is constant, the new decoder is still in $\mathcal{AC}^0$.

The following is implicit in [IJKO6].

**Lemma 6.4.7.** Let $C : \{0,1\}^M \rightarrow \{0,1\}^N$ be an arbitrary code. Then for any $\varepsilon = \Omega(\text{poly}(1/k))$: the k-direct-product of C is $(1/25)$-approximate locally and list-decodable from agreement $\varepsilon$ with list size $\text{poly}(1/\varepsilon)$ by constant-adaptivity probabilistic $\mathcal{NC}^1$ circuits of size $\text{poly}(\log(N)/\varepsilon)$.

**Claim 6.4.8.** Let $C : \{0,1\}^M \rightarrow \{0,1\}^{\text{poly}(M)}$ be the code from Theorem 1.4.3. For $\varepsilon > 0$, let $k = \text{poly}(1/\varepsilon)$ satisfy the condition of Lemma 6.4.7. Then $C^k$, the k-direct-product of C, is an explicit code that is locally list-decodable with constant adaptivity by probabilistic $\mathcal{NC}^1$ circuits of size $\text{poly}(\log(M)/\varepsilon)$ from agreement $\varepsilon$ and with list size $\ell = \text{poly}(1/\varepsilon)$.

**Proof.** Let $x \in \{0,1\}^M$, and let $N = \text{poly}(M)$ be the length of a codeword in $C$. Given oracle access to a string $y \in (\{0,1\}^k)^N$ satisfying $\Delta(y, C^k(x)) \leq 1 - \varepsilon$, the decoder $D^k$ for $C^k$, runs the $(1/25)$-approximate local and list decoder for $C^k$. By Lemma 6.4.7, this produces a list, $M_1, \ldots, M_t$, of machines such that at least one of them, $M_j$, computes $C(x)$ correctly on at least a $24/25$ fraction of entries. We combine each $M_i$ with the decoder for $C$. By Theorem 1.4.3, the latter can locally decode $C$ from distance $1/25$. So together with $M_j$ it computes $x$ correctly on every entry. The decoder and the $M_i$’s are in $\mathcal{NC}^1$ because
so are the approximate decoder for \( C_k \) (by Lemma 6.4.7) and the decoder for \( C \) (Theorem 1.4.3). 

Finally, we apply Theorem 1.4.2 (generalized to decoders with constant adaptivity) on \( C_k \) to obtain a code with the specified parameters that is locally list-decodable by probabilistic \( \mathcal{AC}^0 \) circuits of size \( \text{poly}(\log(M)/\varepsilon) \).

### 6.4.3 A Locally List-Decodable Binary Code

We now prove Theorem 1.4.4, giving a construction of locally list-decodable binary codes with efficient decoders. We obtain an essentially optimal (up to polynomial factors) construction, see Section 6.5 for the matching lower bounds.

**Remark 6.4.9.** The construction of Theorem 1.4.4 only applies for \( \varepsilon \geq 2^{-\Theta(\sqrt{\log M})} \). Thus we fall slightly short of covering the whole possible range (since one can hope to get such codes for \( \varepsilon = 1/M^c \) for a small constant \( c \)). We note, however, that the range of \( \varepsilon \) which is most interesting for us is between \( 1/\text{poly} \log M \) and \( 1/\text{poly} \log \log M \) (see the discussion in the introduction) which we do cover. We also mention that if one insists on codes with \( \varepsilon = 1/M^c \), then we can construct such codes with quasi-polynomial rate.

To prove Theorem 1.4.4, we concatenate three codes. The first is the binary locally-decodable code constructed in Section 6.4.1 that can be uniquely decoded from a constant relative distance. I.e., the code of Theorem 1.4.3: \( \{ C_M : \{0,1\}^M \rightarrow \{0,1\}^{\text{poly}(M)} \}_{M \in \mathbb{N}} \) that can be locally decoded (uniquely, i.e. with list size 1) from distance \( 1/25 \) by probabilistic \( \mathcal{AC}^0 \) circuits of size \( \text{poly}(\log M) \).

The second code that we need is a non-binary approximate locally-list-decodable code. Recall the definition of approximate locally list-decodable codes (Definition 6.2.4). We construct an approximate locally list-decodable code: a modification of the code of [IJKW08], that allows local decoding in \( \mathcal{AC}^0 \).

**Theorem 6.4.10.** For every \( \delta = O(1) \) and every \( 2^{-\Theta(\sqrt{\log M})} \leq \varepsilon = \varepsilon(M) < \delta \), there exists a \( \delta \)-approximate \( (\varepsilon,\text{poly}(1/\varepsilon)) \)-locally-list-decodable code \( \{ C_M : \{0,1\}^M \rightarrow \Gamma^{\text{poly}(M)} \}_{M \in \mathbb{N}} \).
Where \(|\Gamma| = \text{poly}(1/\varepsilon)\). The code has a local decoder that can be implemented by constant depth circuits of size \(\text{poly}(\log M, 1/\varepsilon)\).

**Proof.** The code of [IJKW08] is a concatenation of two approximate locally list-decodable codes. The “outer” code \(C_{\text{out}}\), maps \(\{0, 1\}^M\) to \(\Sigma^\text{poly}(M)\), where \(|\Sigma| = 2^\text{poly}(1/\varepsilon)\). To reduce the alphabet size, each \(\text{poly}(1/\varepsilon)\)-bit symbol of this code is then itself encoded by an “inner” code \(C_{\text{in}}\), mapping \(\{0, 1\}^\text{poly}(1/\varepsilon)\) to \(\Gamma^{N'}\) where \(\Gamma\) is as stated in the theorem (of size \(\text{poly}(1/\varepsilon)\)), and \(N' = (\text{poly}(1/\varepsilon))^{\log(1/\varepsilon)}\). We note that for the range of \(\varepsilon\) that we consider we get that \(N' \leq \text{poly}(M)\). Thus the concatenated code maps \(\{0, 1\}^M\) to \(\Gamma^{\text{poly}(M)}\), and by [IJKW08], its approximate list-decoding parameters are as stated.

In terms of complexity, the decoder for \(C_{\text{in}}\) can be implemented in \(\mathcal{AC}^0\) (this is shown in [IJKW08]). We do not, however, know how to implement the decoder for \(C_{\text{out}}\) in \(\mathcal{AC}^0\). Therefore, to get a code decodable in \(\mathcal{AC}^0\), we modify their outer code (we abuse notation and still call the modified code \(C_{\text{out}}\)) and present an \(\mathcal{AC}^0\) decoder for the modified code. The main reason we need to modify their code is that we don’t know of a concise and unique representation of low-degree affine sub-spaces that can be computed and manipulated in \(\mathcal{AC}^0\). We instead represent such subspace using some basis vectors and a shift vector (a concise, but not unique representation). This changes the code and allows decoding in \(\mathcal{AC}^0\).

**Modified Outer Code.** Let \(k = \log |\Sigma|\). We associate the set of message indices \([M]\) with a \(m\)-dimensional vector-space over a finite field \(\mathcal{F}_q\), where \(q^m = M\), and \(q^8 = k = \log |\Sigma| = \text{poly}(1/\varepsilon)\). The codeword is indexed by tuples of 9 vectors in \(\mathcal{F}_q^m\), and thus each codeword is of length \(O(|\mathcal{F}_q^m|^9) = O(M^9) = \text{poly}(M)\).

Take \(msg \in \{0, 1\}^M\). Its encoding is defined as follows. Let \((\vec{v}_1, \ldots, \vec{v}_8, \vec{s})\) be an index into the codeword (i.e. \(\vec{v}_1, \ldots, \vec{v}_8, \vec{s} \in \mathcal{F}_q^m\)). Consider the affine subspace \(B\) spanned by \(\vec{v}_1, \ldots, \vec{v}_8\) and shifted by \(\vec{s}\)

\[
B = \{a_1 \vec{v}_1 + \cdots + a_8 \vec{v}_8 + \vec{s} : a_1, \ldots, a_8 \in \mathcal{F}_q\}
\]

Recalling that entries in the message are associated with vectors in \(\mathcal{F}_q^m\) (using some canonical
ordering), we take the \((\vec{v}_1, \ldots, \vec{v}_8, \vec{s})\)-th codeword entry to be all the message bits whose indices are in the affine subspace \(B\) (in some order). I.e.,

\[
C_{out}(msg)[\vec{v}_1, \ldots, \vec{v}_8, \vec{s}] = \{msg[u] : u \in B\}
\]

Each alphabet symbol's length is the size of the 8-dimensional subspace which is \(q^8 = \text{poly}(1/\varepsilon)\), and thus the alphabet size is exponential in \(1/\varepsilon\), as claimed above. We think of each (9-vector) index in the codeword as an 8-dimensional affine subspace of \(F_q^m\), where the first 8 vectors in the index are basis vectors and the last one is a shift vector. First, note that each such affine subspace appears multiple times with different representations. Also note that not every index represents an 8-dimensional subspace (since the first 8 vectors may be linearly dependent). However, the next claim says that the number of indices that do not represent 8-dimensional subspaces is very small.

**Claim 6.4.11.** Let \(\vec{v}_1, \ldots, \vec{v}_9\) be chosen uniformly and independently from \(F_q^m\). The probability that they are linearly dependent is at most \(\frac{2^8}{q^m} \leq 1/\sqrt{M}\).

**Proof.** Examine the process of iteratively choosing 8 uniformly random vectors. After choosing the first \(i \geq 0\) vectors, and assuming they are independent, they span a subspace of size \(q^i\). The \(i + 1\)-th vector is in the subspace they span only with probability \(\frac{q^i}{q^m}\), and otherwise the \(i + 1\) vectors are linearly independent. Taking a Union Bound, the total probability the 8 vectors are dependent is at most \(\frac{2^8}{q^m}\). □

Note that in our range of parameters, \(1/\sqrt{M} << \varepsilon\). So we can ignore the locations that are indexed by subspaces that are not 8-dimensional, this will not have a meaningful effect on the success probability of decoding or on Hamming distances between received words and codewords (since we only care about received words that have agreement at least \(\varepsilon\) with codewords).

We now describe the local decoder \(D\) for this code (viewed as a two-stage process as discussed in Remark 6.2.2). Given access to a string \(y\) that has Hamming distance at most \(\varepsilon\) from some codeword \(C_{out}(msg)\), \(D\) produces \(O(1/\varepsilon^2)\) probabilistic circuits such that at
least one of them decodes at least $1 - \delta$ fraction of the bits in $msg$ (with high probability over the random coins of the circuit). Each one of the circuits in the list is generated independently as follows: $D$ chooses uniformly at random 9 vectors $v_1, \ldots, v_8, s \in \mathbb{F}_q^m$. By Claim 6.4.11, with high probability these vectors represent an 8-dimensional affine subspace $B$ (otherwise consider this a failure). $D$ then chooses a random 4-dimensional affine subspace $A \subseteq B$. Next, $D$ reads the received word $y$ in the entry indexed by $(v_1, \ldots, v_8, s)$. (For an uncorrupted location this should give the message values at all indices in the subspace $B$.)

Let $v$ be the values in this entry that are associated with the subspace $A$. $D$ generates the circuit $C_{A,v}$ that does the following: on input $j \in [M]$, $C_{A,v}$ checks whether $j \in A$ (where we think of $j$ as a vector in $\mathbb{F}_q^m$). If so, then $C_{A,v}$ outputs the bit in $v$ that is associated with the vector $j$. Otherwise, $C_{A,v}$ repeatedly (in parallel) for $O(\frac{\log(1/\delta)}{\epsilon})$ iterations does the following: chooses $(w_1, \ldots, w_8, \bar{t})$ that are randomly distributed under the constraint that they span an 8-dimensional affine subspace $B$, such that $A \cup \{j\} \subseteq B$. It then reads $y$ at index $(w_1, \ldots, w_8, \bar{t})$, and compares the bits in $v$ with the bits in this location associated with the elements of $A$. If for none of the iterations there is a full agreement between $v$ and the bits associated with $A$, then $C_{A,v}$ outputs some error message. Otherwise, it takes the first iteration in which there is an agreement and outputs the bit in this location associated with $j$.

The proof that this decoder has the desired list-decoding properties follows exactly the proof of [IJKW08]. (Ignoring the tiny fraction of indices that are not 8-dimensional subspaces, our code is their code with the only difference that each bit in the codeword is repeated many times, as many as the number of representations of an 8-dimensional affine subspace.) We only need to verify that the complexity of the decoder is as stated, which amounts to verifying the following three easy claims. Note throughout that addition and multiplication over $\mathbb{F}_q$ can be done by $\mathcal{AC}^0$ circuits of size $\text{poly}(q) = \text{poly}(1/\epsilon)$.

**Claim 6.4.12.** Given an 8-dimensional affine subspace $B \subseteq \mathbb{F}_q^m$, represented by $(v_1, \ldots, v_8, s)$. A probabilistic $\mathcal{AC}^0$ circuit of size $\text{poly}(\log M, 1/\epsilon)$ can sample a uniformly distributed 4-dimensional subspace $A \subseteq B$.  

259
Proof. Every 4-dimensional subspace in $B$ is spanned by 4 linearly independent vectors, and all these sub-spaces are of equal size. Thus, to sample the subspace $A$ we first sample 4 random vectors $(\vec{w}_1, \ldots, \vec{w}_4)$ in $B$, by taking random linear combinations of $\vec{v}_1, \ldots, \vec{v}_8$. With high probability (more than say $1/2$), these vectors will be linearly independent. This can be verified in $\mathcal{AC}^0$ by enumerating all $q^4 = \text{poly}(1/\varepsilon)$ linear combinations, and thus the success probability can be amplified (in parallel time) to $1 - 1/	ext{poly}(M, 1/\varepsilon)$. This gives us a random basis for the 4-dimensional subspace $A$. Now, to choose the shift vector $\vec{t}$ we just choose another random vector in $B$ (linearly independent or not) and obtain the random 4-dimensional subspace $A$ spanned by $(\vec{w}_1, \ldots, \vec{w}_4)$ and shifted by $\vec{t}$. All of these operations can be done by an $\mathcal{AC}^0$ circuit of size $\text{poly}(\log M, 1/\varepsilon)$. ■

Claim 6.4.13. Given a 4-dimensional affine subspace $A \subseteq \mathbb{F}_q^m$, represented by $(\vec{v}_1, \ldots, \vec{v}_4, \vec{s})$ and a vector $\vec{j} \in \mathbb{F}_q^m$. An $\mathcal{AC}^0$ circuit of size $O(\log 1/\varepsilon)$ can check whether $\vec{j} \in A$.

Proof. Again, this is easily done by enumerating all $q^4 = \text{poly}(1/\varepsilon)$ possible linear combinations of $(\vec{v}_1, \ldots, \vec{v}_4)$, and checking whether any of them give the vector $\vec{j} - \vec{s}$. ■

Claim 6.4.14. Given a 4-dimensional affine subspace $A \subseteq \mathbb{F}_q^m$, represented by $(\vec{v}_1, \ldots, \vec{v}_4, \vec{s})$ and a vector $\vec{j} \in \mathbb{F}_q^m \setminus A$, a probabilistic $\mathcal{AC}^0$ circuit of size $\text{poly}(\log M, 1/\varepsilon)$ can sample vectors $(\vec{w}_1, \ldots, \vec{w}_8, \vec{t})$ that are uniform under the condition that they span an 8-dimensional subspace $B$, such that $A \cup \{\vec{j}\} \subseteq B$.

Proof. First, we take a fifth basis vector, which is $\vec{j} - \vec{s}$, to get the (unique) 5-dimensional subspace $A'$ that contains $A$ and $\{\vec{j}\}$. Now, we want to choose a random 8-dimensional subspace that contains $A'$, and to do this we choose 3 more uniformly random vectors that are independent of $(\vec{v}_1, \ldots, \vec{v}_4, \vec{j} - \vec{s})$ (as above, this can be done in $\mathcal{AC}^0$).

We now have 8 basis vector and the shift vector $\vec{s}$. These define the 8-dimensional subspace $B$. Note that this representation of $B$ depends on the representation of $A$ (e.g. in particular the shift vector is still $\vec{s}$), while we need to choose an independent representation of $B$. To solve this, we "randomize" the basis vectors by choosing a new basis consisting of 8 random linearly independent linear combinations of the basis vectors (this randomizes the
basis vectors but doesn’t change the subspace), and then choosing a random vector in the subspace as the new shift. All of this is easily done in $\mathcal{AC}^0$, and we indeed obtain a random representation of a random subspace containing $A'$. ■

The third code in our construction is the well known Hadamard code with its local list-decoder given by Goldreich and Levin [GL89].

**Theorem 6.4.15.** For every $0 < \varepsilon(m) < 1/2$, The Hadamard code, $\text{Had} : \{0,1\}^m \to \{0,1\}^{2^m}$ is a $(1/2 - \varepsilon, \frac{1}{\varepsilon})$-local-list-decodable-code. The decoder can be implemented by an $\mathcal{AC}^0$ circuit of size $\text{poly}(m,1/\varepsilon)$ that uses majority gates of fan-in $\Theta(1/\varepsilon)$.

We can now put everything together and prove Theorem 1.4.4.

**Proof of Theorem 1.4.4.** Fix $\varepsilon(M) = \varepsilon$ in the specified range. Our code $C$ is a combination of the codes in Theorem 6.1.1 (we denote it here by $C_1$), the code in Theorem 6.4.10 (we denote it here by $C_2$) with $\varepsilon_2 = \varepsilon^3/2$ and $\delta = 1/25$, and the code in Theorem 6.4.15 ($C_3$), with $\varepsilon_3 = \varepsilon/2$. Given a message $x \in \{0,1\}^M$, we first encode it using $C_1$ to obtain a binary string $x'$ (of length $N_1 = \text{poly}(M)$). We then encode $x'$ using $C_2$ to obtain a string of length $N_2 = \text{poly}(M)$ over the alphabet $\Gamma$ (of size $\text{poly}(1/\varepsilon)$). We then concatenate this code (i.e. encode every symbol of it) with $C_3$. Let $k = \log(|\Gamma|)$. The length $N$ of the final code is $N_2$ multiplied by $N_3 = 2^k = \text{poly}(1/\varepsilon) \leq \text{poly}(M)$, which is $\text{poly}(M)$.

We now turn to the decoding properties of this code. We start by showing that the concatenation of $C_2$ and $C_3$, denoted by $C'$, is an approximate locally list-decodable (binary) code.

**Lemma 6.4.1.** $C' : \{0,1\}^{N_1} \to \{0,1\}^N$ is a $1/25$-approximate $(1/2 - \varepsilon, \ell)$-locally-list-decodable code, where $\ell = \text{poly}(1/\varepsilon)$. The local decoder for $C'$ can be implemented by constant-depth circuits of size $\text{poly}(\log M, 1/\varepsilon)$, with majority gates of fan-in $O(1/\varepsilon)$ (and AND/OR gates of unbounded fan-in).
Proof. By Theorem 6.4.10 (and the simplification assumption from Remark 6.2.2), there is a locally-list-decoder $D_2$ taking advice of size $\log(\ell_2)$, where $\ell_2 = \text{poly}(1/\varepsilon)$, such that for every $y \in \Gamma^{N_2}$ and for every $m \in \{0,1\}^{N_1}$ for which $\Delta_F(C_2(m), y) \leq 1 - \varepsilon^3/2$,

$$\exists a \in [\ell_2] \text{ s.t. } \Pr_{i \in \Omega_1} [D_2^y(a, i) = m[i]] > 9/10 \geq 24/25$$

Furthermore, $D_2$ can be implemented by a constant depth circuit of size $\text{poly}(\log M, 1/\varepsilon)$.

By Theorem 6.4.15, there is a locally-list-decoder $D_3$ taking advice of size $\log(\ell_3)$, where $\ell_3 = O(1/\varepsilon^2)$, such that for every $y \in \{0,1\}^{N_3}$ and for every $m \in \{0,1\}^k$ for which $\Delta(C_3(m), y) \leq 1/2 - \varepsilon/2$,

$$\exists a \in [\ell_3] \text{ s.t. } \forall i \in [k] \Pr[D_3^y(a, i) = m[i]] > 9/10$$

Furthermore, $D_3$ can be implemented by a constant depth circuit of size $\text{poly}(\log M, 1/\varepsilon)$, that uses majority gates of fan-in $O(1/\varepsilon)$.

The fact that the local-list-decoders for the two codes can be combined to obtain a local-list-decoder for the concatenated code (with list size that is the product of the two list sizes) is quite a standard argument. We refer the reader to [STV01] for the formal details. Here we just sketch the argument.

The decoder $D'$ for the concatenated code $C'$ roughly works as follows: it takes advice $(a_2, a_3) \in [\ell_2] \times [\ell_3]$. Given an index $i$, $D'$ runs $D_2(a_2, i)$. Whenever the latter needs a $(k$-bit$)$ symbol from its received word, $D'$ runs $D_3(a_3, \cdot)$ to retrieve the whole symbol.

To analyze the correctness we argue as follows. For a received word $y \in \{0,1\}^N$ and a message $x \in \{0,1\}^{N_1}$ for which $\Delta(C'(x), y) \leq 1/2 - \varepsilon$, there are at least $\varepsilon/2$ symbols of $C_2(x)$ for which their $C_3$ encoding has $1/2 + \varepsilon/2$ agreement with the corresponding bits in $y$. Each one of these gives rise to a list of $\ell_3$ possible symbols one of which is the correct one.

By an averaging argument, there is a $a_3 \in [\ell_3]$, for which at least $\varepsilon/2 \cdot \varepsilon^2 = \varepsilon^3/2$ fraction of the symbols of $C_2(x)$ are such that the $a_3$'th element in the list produced by $D_3$ (with advice $a_3$) agrees with the corresponding symbol of $C_2(x)$. Since $D_2$ (with an appropriate advice
$a_2$) can $1/25$-approximately recover from agreement $\varepsilon^3/2$, we get that the combined decoder with advice $(a_2, a_3)$ recovers a string that has agreement $24/25$ with $x$.

The size of the decoders $D_2, D_3$ is $\text{poly}(\log M, 1/\varepsilon)$. Both are of constant depth where the latter uses majority gates of fan-in $\Theta(1/\varepsilon)$. Combining the two we get a constant-depth $(1/2 - \varepsilon, \text{poly}(1/\varepsilon))$-locally-list-decoder for the concatenated code of size $\text{poly}(\log M, 1/\varepsilon)$ with majority gates of fan-in $\Theta(1/\varepsilon)$.

We now can describe the decoder $D$ for $C$. On a received word $y \in \{0, 1\}^N$, we run the local-decoder $D_1$ for $C_1$. Whenever it requires a bit from its received word (in $\{0, 1\}^{N_1}$), we run the approximate local-decoder $D'$ for $C'$, with some advice string in $[\ell]$ (where $\ell = \ell_2 \cdot \ell_3$), to obtain a candidate for that symbol. If the received word has $1/2 - \varepsilon$ agreement with $C(x)$ (for some $x \in \{0, 1\}^M$), then there exist an advice string $a \in [\ell]$ such that $D_2(a, \cdot)$ decodes correctly at least $24/25$ fraction of the symbols of $C_2(x)$. Thus, when $D_1$ receives symbols from $D_2(a, \cdot)$, it gets access to a word that has $24/25$ agreement with $C_1(x)$ and hence it correctly decodes every symbol in $x$.

Since the sizes of the two decoders is $\text{poly}(\log M, 1/\varepsilon)$, and their depth is constant, then so are the size and depth of the combined decoder, and it uses majority gates of fan-in $\Theta(1/\varepsilon)$ because so does the decoder $D_2$.

As mentioned in Remark 6.4.9, we can obtain codes with quasi-polynomial rate that work for $\varepsilon = 1/M^\delta$. These are obtained by replacing the code $C'_M$ in the proof of Theorem 6.4.10, which is a de-randomized direct-product code by [IJKW08], with their (not de-randomized) direct-product code. We state the parameters of these codes without a proof.

**Theorem 6.4.16.** For every $1/M^\delta \leq \varepsilon = \varepsilon(M) < 1/2$ (where $\delta > 0$ is a constant), there exist a $(1/2 - \varepsilon, \text{poly}(1/\varepsilon))$-locally-list-decodable code $\{C_M : \{0, 1\}^M \rightarrow \{0, 1\}^{M^{O(\log(1/\varepsilon))}}\}_{M \in \mathbb{N}}$ with a local-decoder that can be implemented by a family of constant depth circuits of size $\text{poly}(\log M, 1/\varepsilon)$ that use majority gates of fan-in $\Theta(1/\varepsilon)$ (and AND gates of unbounded fan-in).
6.5 Local-List-Decoding Requires Computing Majority

In this section we prove Theorem 1.4.6, showing that local list-decoding of binary error correcting codes from relative distance $1/2 - \varepsilon$ essentially requires computing the majority function on $\Theta(1/\varepsilon)$ bits. In particular, this means that there are no constant-depth decoders for polynomial $\varepsilon$. This negative result is essentially tight (up to polynomial factors) given the construction of Theorem 1.4.4. We begin with a formal statement of the theorem:

**Theorem 6.5.1 (Formal statement of Theorem 1.4.6).** Let $\{C_M : \{0, 1\}^M \rightarrow \{0, 1\}^{N(M)}\}_{M \in \mathbb{N}}$ be a $(1/2 - \varepsilon(M), \ell(M))$-locally-list-decodable code, such that $\ell(M) \leq 2^{cM}$, and $1/N^6 \leq \varepsilon(M) \leq \delta_2$ for universal constants $\kappa, \delta_1, \delta_2$. Let $D$ be the local decoding machine, of size $S(M)$ and depth $d(M)$.

Then, for every $M \in \mathbb{N}$, there exists a circuit $A_M$ of size $\text{poly}(S(M), \ell(M))$ and depth $O(d(M))$, that computes majority on $\Theta(1/\varepsilon(M))$ bits. The types of gates used by the circuit $A_M$ are identical to those used by $D$. E.g., if $D$ is an $AC^0[1]$ circuit, then so is $A_M$.

**Proof Intuition for Theorem 1.4.6.** Fix a message length $M$ and $\varepsilon = \varepsilon(M)$. We will describe a circuit $B$ with the stated parameters that decides the promise problem $\Pi$ on inputs of length roughly $1/\varepsilon$. By Lemma 6.2.7 this will also give a circuit for computing majority.

We start with a simple case: assume that the (local) decoder $D$ makes only non-adaptive queries to the received word. In this case we proceed using ideas from the proof of Theorem 6.4 in [Vio06]. Take $m$ to be a message that cannot be even approximately decoded\(^8\) from random noise with error rate $1/2$. Such a word exists by a counting argument. Let $C(m)$ be the encoding of $m$. Let $x \in \Pi_{Y_{\varepsilon s}} \cup \Pi_{N_{o}}$ be a $\Pi$-instance of size $1/2\varepsilon$ (we assume w.l.o.g. throughout that $1/\varepsilon$ is an integer). $B$ uses $x$ to generate a noisy version of $C(m)$, by XORing each one of its bits with some bit of $x$ that is chosen at random. It then uses $D$ to decode.

---

\(^8\)By this we mean that no decoder can recover (w.h.p.) a string that is, say, $1/3$-close to $m$.  

264
this noisy version of $C(m)$. If $x \in \Pi_{No}$, this adds random noise (error rate 1/2), and the decoding algorithm cannot recover most of $m$'s bits. If $x \in \Pi_{Yes}$, then each bit is noisy with probability less than $1/2 - 2\varepsilon$, which means that w.h.p. the fraction of errors is at most $1/2 - \varepsilon$, and the decoding algorithm successfully recovers every bit of $m$.

By comparing the answers of the decoding algorithm (or more precisely, every decoding algorithm in the list, by trying every possible advice) and the real bits of $m$ in a small number of random locations, the algorithm $B$ distinguishes w.h.p. whether $x \in \Pi_{Yes}$ or $x \in \Pi_{No}$.

Note, however, that $B$ as described above is not a standard algorithm for $\Pi$. This is because we gave $B$ access to the message $m$ as well as its encoding. Both of these are strings that are much larger than we want $B$ itself to be. So our next goal is to remove (or at least minimize) $B$'s access to $m$ and $C(m)$, making $B$ a standard circuit for $\Pi$. Observe that $B$ as described above distinguishes whether $x$ is in $\Pi_{Yes}$ or in $\Pi_{No}$ with high probability over the choices of $D$'s random coins, the random locations in which we compare $D$’s answers against $m$, and the random noise generated by sampling bits from $x$. In particular, there exists a fixing of $D$’s random string as well as the (small number of) testing locations of $m$ that maintains the advantage in distinguishing whether $x$ comes from $\Pi_{Yes}$ or $\Pi_{No}$, where now the probability is only over the randomness used to sample bits from $x$. So now we can hardwire the bits of $m$ used to test whether $D$ decodes the noisy version of $C(m)$ correctly (i.e. we got rid of the need to store the whole string $m$). Furthermore, after we fix $D$’s randomness, by the fact that it is non-adaptive, we get that the positions in which $B$ queries the noisy $C(m)$ are now also fixed, and independent of $x$. So we also hardwire the values of $C(m)$ in these positions (and only these positions) into $B$. For any $x$, we now have all the information to run $B$ and conclude whether $x$ is in $\Pi_{Yes}$ or $\Pi_{No}$.

Next we want to deal with adaptive decoders. If we proceed with the ideas described above, we run into the following problem: suppose the circuit has two (or more) levels of adaptivity. The queries in the second level do not only depend on the randomness of the decoder, but also on the values read from the received word at the first level, and in particular they also depend on the noise. The noise in our implementation depends on the specific $\Pi$-
instance \( x \). This means that we cannot hardwire the values of \( C(m) \) that are queried at the second level because they depend on \( x \! 

To solve this problem, we analyze the behavior of the decoder when its error rate changes in the middle of its execution. Specifically, suppose that the decoder \( D \) queries the received word in \( d \) levels of adaptivity. For every \( 0 \leq k \leq d \), we consider the behavior of the decoder when up to level \( k \) we give it access to the encoded message corrupted with error-rate \( 1/2 - 2\varepsilon \), and above the \( k \)’th level we give it access to the encoded message corrupted with error-rate \( 1/2 \). By a hybrid argument, there exists some level \( k \), in which the decoder has a significant advantage in decoding correctly when up to the \( k \)’th level it sees error rate \( 1/2 - 2\varepsilon \) (and error-rate \( 1/2 \) above it), over the case that up to the \((k - 1)\)’th level the error-rate is \( 1/2 - 2\varepsilon \) (and \( 1/2 \) from \( k \) and up). We now fix and hardwire randomness for the decoder, as well as noise for the first \( k - 1 \) levels (chosen according to error-rate \( 1/2 - 2\varepsilon \)), such that this advantage is preserved. Once the randomness of \( D \) and the noise for the first \( k - 1 \) levels are fixed, the queries at the \( k \)-th level (but not their answers) are also fixed. For this \( k \)-th level we can proceed as in the non-adaptive case (i.e. choose noise according to \( x \) and hardwire the fixed positions in \( C(m) \)). We now have to deal with queries above the \( k \)’th level. At first glance it is not clear that we have gained anything, because we still have to provide answers for these queries, and as argued above, these may now depend on the input \( x \) and therefore the query locations as well as the restriction of \( C(m) \) to these locations cannot be hard-wired. The key point is that for these “top” layers the error rate has changed to \( 1/2 \). So while we have no control on the query locations (as they depend on \( x \)) we do know their answers: they are completely random bits that have nothing to do with \( m \) or \( C(m) \)!

Thus, \( B \) can continue to run the decoder, answering its queries (in the levels above the \( k \)’th) with random values. We thus obtain a circuit that decides membership in \( \Pi \) correctly with a small advantage. Since the number of adaptivity levels is only \( d \) (the circuit depth of the decoder), the distinguishing advantage of the \( k \)-th hybrid is at least \( O(1/d) \), and in particular this advantage can now be amplified by using only additional depth of \( O(\log(d)) \).

This gives a circuit that computes \( \Pi \) and concludes the proof. Due to space constraints we
defer the formal proof of Theorem 1.4.6 to the full version of this paper [GR08].

**Proof of Theorem 1.4.6.** Fix $M \in \mathbb{N}$, $C = \mathcal{C}_M : \{0,1\}^M \rightarrow \{0,1\}^{N(M)}$, $\varepsilon = \varepsilon(M)$, $\ell = \ell(M)$, $S = S(M)$ and $d = d(M)$ as in the statement of the theorem. We show how to use the decoder $D$ to construct a circuit for computing $\Pi$ on instances of size $1/\varepsilon$ (and thus also for computing majority, by Claim 6.2.7) as promised in the theorem statement.

Let us start with some notation. For an advice string $a \in [\ell]$, an index $i \in [M]$, and a received word $y \in \{0,1\}^N$, we denote by $D^a(y, i, r)$ an execution of the decoder $D$ with advice $a$, randomness $r$, and (oracle) access to $y \in \{0,1\}^N$ to retrieve the $i$-th message bit (recall that we are working under the simplifying assumption from Remark 6.2.2). For $m \in \{0,1\}^M$ and $0 \leq \alpha \leq 1$, we use $\Gamma_{\alpha}(a, y, m)$ to denote the fraction of indices $i$ in $m$ that $D^a(a, i, r)$ recovers with probability at least $\alpha$ (the probability is over $D$’s randomness $r$). Formally:

$$\Gamma_{\alpha}(a, y, m) \overset{\text{def}}{=} \frac{1}{M} \left| \left\{ i \in [M] : \Pr_r[D^a(a, i, r) = m[i]] \geq \alpha \right\} \right|$$

Let $E_0$ be the uniform distribution on $\{0,1\}^N$, and $E_1$ be the distribution over $\{0,1\}^N$ in which every bit is chosen (independently) to be 1 with probability $1/2 - 2\varepsilon$ and 0 otherwise.

First we show that there exists a message $m \in \{0,1\}^M$, such that if $C(m)$ is corrupted with completely random noise, then with probability 9/10 over the noise, for every advice string $a$, the decoder $D$ cannot recover more than $3/5$ of $m$’s indices with probability greater than $3/5$ (over its random coins).

**Claim 6.5.2.** There exists a message $m \in \{0,1\}^M$ such that,

$$\Pr_{e \sim E_0}[\exists a \in [\ell] \text{ s.t. } \Gamma_{3/5}(a, C(m) \oplus e, m) > 3/5] \leq 1/10$$

Where the $\oplus$ operation between bit strings means bit-wise XOR.

**Proof.** The intuition is that if $e$ is drawn from $E_0$ (error rate $1/2$), then $C(m) \oplus e$ is independent of $C(m)$, and thus for most $m$’s, all of the $\ell$ possible outputs of the decoder are far from $m$. Formally:
Pr_{m \in \{0,1\}^M, e \in E_0} \left[ \exists a \in [\ell] \text{ s.t. } \Gamma_{3/5}(a, C(m) \oplus e, m) > 3/5 \right] =

Pr_{m \in \{0,1\}^M, e \in E_0} \left[ \exists a \in [\ell] \text{ s.t. } \Gamma_{3/5}(a, e, m) > 3/5 \right] =

Pr_{m \in \{0,1\}^M, e \in E_0} \left[ \exists a \in [\ell] \text{ s.t. } \frac{1}{M} \left| \left\{ i \in [M] : \Pr_r[D^e(a, i, r) = m[i]] \geq 3/5 \right\} \right| \geq 3/5 \right] =

Examining this last quantity, for any fixed error vector $e$ and advice $a$, let $m_a^e$ be the (single) message obtained by taking $m_a^e[i]$ to be the more probable answer (over $r$) of $D^e(a, i, r)$. Now, fixing $e$, and taking a random $m$, the probability that

$$\exists a \in [\ell] \text{ s.t. } \frac{1}{M} \left| \left\{ i \in [M] : \Pr_r[D^e(a, i, r) = m[i]] \geq 3/5 \right\} \right| \geq 3/5$$

is at most the probability that for the random $m$, for some $a \in [\ell]$, the fractional distance between $m_a^e$ and $m$ is at most $2/5$. Denote by $Vol_{2/5}(M)$ the volume of the $M$-dimensional sphere of radius $2M/5$ (in the $M$-dimensional Hamming cube). Taking $H$ to be the binary entropy function, the probability that there exists $a \in [\ell]$ such that $m$ is $2/5$-close to $m_a^e$ is (by a union bound) at most:

$$\frac{\ell \cdot Vol_{2/5}(M)}{2^M} \leq \frac{\ell \cdot 2^{(H(2/5)+o(1)) \cdot M}}{2^M} \leq \frac{1}{2^{\Omega(M)}} \leq 1/10$$

Where in the last inequality we assume $\ell \leq 2^{\kappa M}$ for a universal constant $\kappa$. We conclude that indeed:

$$\Pr_{m \in \{0,1\}^M, e \in E_0} \left[ \exists a \in [\ell] \text{ s.t. } \Gamma_{3/5}(a, C(m) \oplus e, m) > 3/5 \right] \leq 1/10$$

and thus certainly there exists an $m \in \{0,1\}^M$ for which

$$\Pr_{e \in E_0} \left[ \exists a \in [\ell] \text{ s.t. } \Gamma_{3/5}(a, C(m) \oplus e, m) > 3/5 \right] \leq 1/10$$
In contrast to the above claim, the decoding algorithm has the guarantee that for every message \( m \in \{0, 1\}^M \), with high probability over noise \( e \) of rate \( 1/2 - 2\epsilon \) or less, there exists an advice string \( a \in [\ell] \) such that when \( D \) is given this advice string and oracle access to the codeword \( C(m) \) corrupted by \( e \), it recovers every bit of \( m \) with probability \( 9/10 \).

**Claim 6.5.3.** For every message \( m \in \{0, 1\}^M \):

\[
\Pr_{e \sim E_1} \left[ \exists a \in [\ell] \text{ s.t. } \Gamma_{9/10}(a, C(m) \oplus e, m) = 1 \right] > 9/10
\]

*Proof.* Recall that the decoder \( D \) has the guarantee that if a codeword is corrupted in less than a \( 1/2 - \epsilon \)-fraction of its coordinates, then for some \( a \in [\ell] \), when \( D \) uses advice \( a \) it can recover each of the original message’s coordinates with probability at least \( 9/10 \) (over its coins). It remain only to show that the probability that \( e \) drawn from \( E_1 \) corrupts more than a \( 1/2 - \epsilon \)-fraction of \( C(m) \)'s coordinates is at most \( 1/10 \). This follows by a Chernoff bound, since \( e \) that is drawn from \( E_1 \) corrupts independently every coordinate of \( C(m) \) with probability \( 1/2 - 1/2\epsilon \). Then the probability that the fraction of coordinates corrupted is more than \( 1/2 - 1/\epsilon \) is exponentially small in \( 1/\epsilon \) (here we use that fact that \( 1/\epsilon \) is significantly smaller than \( N \), because \( \epsilon \geq 1/N^{\delta_1} \) for some universal constant \( \delta_1 > 0 \)). In particular, for \( \epsilon \) smaller than some universal constant \( \delta_2 > 0 \), this probability is indeed smaller than \( 1/10 \) as required. \( \blacksquare \)

Fix \( m \) as in Claim 6.5.2. We define a probabilistic circuit \( A_1 \) that for \( b \in \{0, 1\} \) gets oracle access to a string \( y = C(m) \oplus e \) where \( e \) is sampled from the distribution \( E_b \). The goal of the circuit is to guess the value of \( b \). We begin by constructing such a circuit that also gets oracle access to the string \( m \). The algorithm is described in Figure 6-1.

The algorithm \( A_1 \) (as described in Figure 6-1) can be implemented by a probabilistic oracle circuit of size \( \text{poly}(S, \ell) \) and depth \( O(d) \), where the circuit has oracle access to the message \( m \) and noisy codeword \( C(m) \oplus e \). Denote by \( \tilde{r} \) the randomness used by \( A_1 \).
Oracle access to: $m$ and $y = C(m) \oplus e$ where $e \leftarrow E_b$.

Output: $b$.

The algorithm:
Let $q = \Theta(\log(\ell))$. For every $a \in [\ell]$ do the following in parallel:

1. Choose random indices $i_1^a, \ldots, i_q^a \in [M]$.
2. Choose random strings $r_1^a, \ldots, r_q^a$ for $D$.
3. For every $j \in [q]$ run $D^a(i_j, r_j^a)$ to obtain a prediction for the bit $m[i_j^a]$. If for at least $\frac{43}{50}$ of the $j$’s, the prediction is equal to $m[i_j^a]$, output 1 and halt.

Otherwise (no $a \in [\ell]$ resulted in output 1), output 0 and halt.

Figure 6-1: Algorithm $A_1$

Claim 6.5.4.

$$\Pr_{e \leftarrow E_1, \bar{r}}[A_1^{m, C(m)} \oplus e(\bar{r}) = 1] - \Pr_{e \leftarrow E_0, \bar{r}}[A_1^{m, C(m)} \oplus e(\bar{r}) = 1] \geq 1/2$$

Proof. By Claim 6.5.3, when $e$ is drawn from $E_1$, with probability $9/10$, there exists $a \in [\ell]$ for which $D$ (with advice $a$) successfully recovers each of $m$’s indices with probability $9/10$ (over its random coins). In this case, when $A_1$ tries this $a$, with probability at least $1 - 1/poly(\ell)$, in at least $\frac{43}{50}$ of its $q$ experiments it will successfully retrieve the proper bit of $m$ (by a Chernoff bound). Taking a Union bound, we conclude that, when $e$ is drawn from $E_1$, the probability that $A_1$ outputs 1 is at least $8/10$.

By Claim 6.5.2, when $e$ is drawn from $E_0$, with probability $9/10$, for every $a \in [\ell]$, there exist a $2/5$ fraction of $m$’s indices, such that $D$ (with advice $a$) fails to recover each one of them with probability at least $2/5$ (over its coins). In this case, for any $a$ in the execution of $A_1$, the probability of successfully recovering bits of $m$ in a $\frac{43}{50}$ fraction of the experiments is at most $1/poly(\ell)$ (because at best, the decoder can recover with high probability $3/5$ of the bits of $m$, and is expected, over its randomness to recover each of the remaining $2/5$ bits with probability less than $3/5$). Taking a Union bound, when $e$ is drawn from $E_0$, the probability that $A_1$ outputs 1 is at most $2/10$. 

270
In conclusion:

\[
\Pr_{\epsilon \sim E_1, \bar{\epsilon}} [A^1_{m, C(m) \oplus \epsilon} (\bar{r}) = 1] - \Pr_{\epsilon \sim E_0, \bar{\epsilon}} [A^1_{m, C(m) \oplus \epsilon} (\bar{r}) = 1] \geq 8/10 - 2/10 = 6/10 > 1/2
\]

We now remove the need for oracle access to the message \( m \). This can be done by fixing (for each \( a \in [\ell] \)) all of the \( \bar{r}^{a}_{1}, \ldots, \bar{r}^{a}_{q} \) in the description of \( A_1 \), such that the difference in the probabilities of \( A_1 \) outputting 1 in Claim 6.5.4 is preserved (by averaging such a fixing exists). The values \( m[\bar{r}^{a}_{1}], \ldots, m[\bar{r}^{a}_{q}] \) (for every \( a \in [\ell] \)) can then be hard-wired into the circuit \( A_1 \) (there are only \( \text{poly}(\ell) \) of them). Let us call the new circuit \( A_2 \) which now has only oracle access to \( C(m) \). We have,

\[
\Pr_{\epsilon \sim E_1, \bar{\epsilon}} [A^2_{C(m) \oplus \epsilon} (\bar{r}) = 1] - \Pr_{\epsilon \sim E_0, \bar{\epsilon}} [A^2_{C(m) \oplus \epsilon} (\bar{r}) = 1] > 1/2
\]  

(6.2)

The next step is to remove the oracle access to \( C(m) \oplus e_b \) (these oracle queries are made by \( D \)). This is not straightforward since (as noted in the proof intuition) the queries of an adaptive decoder to the noisy codeword may depend on the noise, and through it (in our construction) on the input \( x \) itself. Since we do not know the query locations, we cannot hardwire the proper values of \( C(m) \) into the circuit. We use a hybrid argument to overcome this difficulty. This involves further notation.

Assume that the decoder \( D \) asks its queries in \( d \) levels of adaptivity (\( d \) is a bound on its depth, so it is certainly a bound on the number of adaptive levels). For \( d \) distributions, \( G^1, \ldots, G^d \) on \( \{0, 1\}^N \), we denote by \( A^2_{C(m) \oplus G^1, \ldots, G^d} (\bar{r}) \) the output of \( A_2 \), with randomness \( \bar{r} \), where queries to the noisy codeword are answered as follows: for every adaptivity level \( k \in [d] \) of the decoder \( D \), sample \( e^k \leftarrow G^k \). If in its \( k \)-th level, \( D \) queries the codeword in position \( j \in [N] \), then the answer is \( C(m)[j] \oplus e^k[j] \).

Note that if we use an oracle as described above (that generates a different noise vector for each adaptivity level), then if the same query is asked in different levels, the answers may be inconsistent. We want all answers to be consistent between the adaptivity levels.
and across all of $A_2$’s executions of $D$ (note that consistency across executions is important because the list-decoding guarantee is against a single fixed noise vector). To guarantee consistency, we modify $A_2$ so that the answers to queries across different executions (and within each execution) are always consistent; if $k$ is the first execution in the minimal level in which query $j$ is made (across the parallel executions), then the answer to query $j$ is always $C(m)[j] \oplus e^k[j]$. We note that this consistency guarantee can be realized with an $AC^0$ circuit, by always answering a query with the answer given to that query in the lexicographically first level and execution number.

Now, for every $0 \leq k \leq d$, we define

$$O^k \overset{def}{=} C(m) \oplus E_1, \ldots, E_1, E_0, \ldots, E_0$$

Consider running $A_2$ with oracle $O^k$. That is, for the first $k$ levels we give $A_2$ access to $C(m)$ corrupted with error rate $1/2 - 2\varepsilon$ and for the last $d-k$ levels we give it access to $C(m)$ corrupted with error rate $1/2$. By (6.2):

$$\text{Pr}_{\tilde{r}, E_1, \ldots, E_1} [A_2^{O^d}(\tilde{r}) = 1] - \text{Pr}_{\tilde{r}, E_0, \ldots, E_0} [A_2^{O^0}(\tilde{r}) = 1] \geq \frac{1}{2d}$$

This inequality holds because for the oracles $O^0$ and $O^d$, all the error vectors (in the different levels) have the same error rate. In this case, $A_2$ with the above "consistency modification", behaves identically to $A_1$ (with the hard-wired bits of $m$) with that same error rate. It follows, by triangle inequality, that there exists $1 \leq k \leq d$, such that,

$$\text{Pr}_{\tilde{r}, E_0, \ldots, E_0, E_1, \ldots, E_1} [A_2^{O^k}(\tilde{r}) = 1] - \text{Pr}_{\tilde{r}, E_0, \ldots, E_0, E_1, \ldots, E_1} [A_2^{O^{k-1}}(\tilde{r}) = 1] \geq \frac{1}{2d} \quad (6.3)$$

Fix such a $k$. Consider the circuit $A$ obtained from $A_2$ as follows: Fix $\tilde{r}$, as well as the noise for the answers of the oracle on the first $k-1$ levels, such that the advantage in Inequality (6.3) is preserved. After doing this, all the queries as well as their answers for the first $k-1$ levels are fixed. Hardwire all of them into the circuit (these are poly($S, \ell$) bits). Also, the
queries (but not their answers) in the k’th level are fixed. Hardwire these queries into the circuit, as well as the values of $C(m)$ in these positions.

We now use $A$ to answer a new guessing game. It is given access to a sample $e \leftarrow E_b$ ($b \in \{0, 1\}$) and it has to guess the value of $b$. It does so by simulating $A_2$ with the fixed randomness $\tilde{r}$, answering oracle queries as follows: for the first $k - 1$ levels it uses the fixed queries and their answers. For level $k$, if $A_2$ queries the received word in position $j$ (which is now fixed), $A$ returns as an oracle answer the value $C(m)[j] \oplus e[j]$ (recall that $C(m)[j]$ is hardwired). For the levels above $k$, $A$ returns random bits (uniformly and independently distributed) as oracle answers. Note that throughout $A$, just like $A_2$, guarantees consistency of answers to $D$’s oracle queries across the parallel executions and adaptivity levels.

Since $A_2$ is an oracle circuit of size $\text{poly}(S, \ell)$ and depth $O(d)$, then so is $A$. Also, it is clear that $A$ simulates $A_2^{O_k}$ when $b = 1$ and $A_2^{O_k^{-1}}$ when $b = 0$ (with fixed values that maximize the gap in (6.3)). Let $\tilde{r}'$ be the randomness of $A$. We have,

$$\Pr_{e \leftarrow E_1, \tilde{r}'}[A^e(\tilde{r}') = 1] - \Pr_{e \leftarrow E_0, \tilde{r}'}[A^e(\tilde{r}') = 1] \geq \frac{1}{2d} \tag{6.4}$$

Let

$$\gamma \overset{\text{def}}{=} \Pr_{e \leftarrow E_1, \tilde{r}'}[A^e(\tilde{r}') = 1] = \Pr[A_2^{O_k} = 1]$$

We are finally ready to describe a circuit $B$ that computes $\Pi$ correctly on instances of length $1/2\varepsilon$ with a small advantage (that will later be amplified). We assume w.l.o.g. that $1/2\varepsilon$ is an even integer. On input $x \in \Pi_{\text{yes}} \cup \Pi_{\text{no}}$ of length $1/2\varepsilon$, $B$ runs $A$ while simulating the noise $e \leftarrow E_b$ as follows: whenever $A$ queries $e$ in position $j$, $B$ chooses uniformly $i \in [1/2\varepsilon]$ and returns the bit $x[i]$. At the end of the execution, $B$ returns the same answer as $A$ does.

$B$ is also a circuit of size $\text{poly}(S, \ell)$ and depth $O(d)$ (inherited from $A$). If $x \in \Pi_{\text{yes}}$, then $\Pr_{i}[x[i] = 1] = 1/2 - 2\varepsilon$, and the simulated oracle is distributed identically to a sample from $E_1$. On the other hand, if $x \in \Pi_{\text{no}}$, then $\Pr_{i}[x[i] = 1] = 1/2$, and the simulated oracle is distributed identically to a sample from $E_0$. We conclude from Inequality (6.4):

273
Claim 6.5.5. If \( x \in \Pi_{\text{yes}} \), \( \Pr[B(x) = 1] \geq \gamma \). And if \( x \in \Pi_{\text{no}} \), \( \Pr[B(x) = 1] \leq \gamma - \frac{1}{2d} \).

Finally, we amplify the success probability of \( B \). This can be done by hard-wiring \( \gamma \) in the circuit, and running \( B \) a \( \text{poly}(d) \) number of times (in parallel) with independent random coins. If at least \( \gamma - \frac{1}{10d} \) of the executions return 1, then return 1, and otherwise 0. By Claim 6.5.5 and a Chernoff bound, this amplified version of \( B \) computes \( \Pi \) correctly on instances of size \( 1/2^e \) with probability of error at most 1/10. Furthermore, it is a circuit of size \( \text{poly}(S, \ell) \) and depth \( O(d) \) (note that counting the number of 1-answers in the \( \text{poly}(d) \) executions that are run for the final amplification step only requires additional depth \( O(\log(d)) \)).

By using known lower bounds for computing the majority function by \( \text{AC}^0[q] \) circuits (for a prime \( q \)) [Raz87, Smo87], we obtain the following corollary.

Corollary 6.5.6. Let \( \{C_M : \{0,1\}^M \rightarrow \{0,1\}^{N(M)}\}_{M \in \mathbb{N}} \) be a \((1/2 - \varepsilon, \ell)\)-locally-list-decodable code (where \( \varepsilon \) is in the range specified in Theorem 1.4.6) with a decoder that can be implemented by a family of \( \text{AC}^0[q] \) circuits of size \( s = s(M) \) and depth \( d = d(M) \). Then \( s = 2^{(1/\varepsilon)^{O(1/\ell)}} / \text{poly}(\ell) \).

### 6.6 Hardness Amplification

**What is Hardness Amplification?** Functions that are hard to compute on the average (by a given class of algorithms or circuits) have many applications, for example in cryptography or for de-randomization via the construction of pseudo-random generators (the “hardness vs. randomness” paradigm [BM84, Yao82, NW94]). Typically, for these important applications, one needs a function that no algorithm (or circuit) in the class can compute it on random inputs much better than a random guess. Unfortunately, however, it is often the case that one does not have or cannot assume access to such a “hard on the average” function, but rather only to a function that is “somewhat hard”: every algorithm in the class fails to compute it and errs, but only on relatively few inputs (e.g. a small constant
fraction, or sometimes even just a single input from every input length). In order to bridge this “hardness gap”, an approach that has been used (very successfully) is to find a way to convert “somewhat hard” functions to functions that are “very hard” (on the average). Procedures that attain this goal are called hardness amplification procedures or reductions.

Let us be more precise. We say that a Boolean function $f : \{0, 1\}^* \rightarrow \{0, 1\}$ is $\delta$-hard on the average for a circuit class $\mathcal{C} = \{\mathcal{C}_n\}_{n \in \mathbb{N}}$ (where circuits in the set $\mathcal{C}_n$ have input length $n$), if for every large enough $n$, for every circuit $C_n \in \mathcal{C}$:

$$\Pr_{x \in \mathbb{U}_n} [C_n(x) = f(x)] < 1 - \delta$$

The task of obtaining from a function $f$ that is $\delta$-hard for a class $\mathcal{C}$, a function $f'$ that is $\delta'$-hard for the class $\mathcal{C}$, where $\delta' > \delta$ is called hardness amplification from $\delta$-hardness to $\delta'$-hardness (against the class $\mathcal{C}$). Typical values for $\delta$ are small constants (close to 0), and sometimes even $2^{-n}$, in which case the hardness amplification is from worst-case hardness. Typical values for $\delta'$ (e.g. for cryptographic applications) are $1/2 - n^{-\omega(1)}$.

The most commonly used approach to prove hardness amplification results is via reductions, showing that if there is a sequence of circuits in $\mathcal{C}$ that computes $f'$ on more than a $1 - \delta'$ fraction of the inputs, then there is a sequence of circuits in $\mathcal{C}$ that computes $f$ on more than $1 - \delta$ fraction of the inputs. An important family of such reductions are so-called fully-black-box reductions which we define next.

**Definition 6.6.1.** A $(\delta, \delta')$-fully-black-box hardness amplification from input length $k$ to input length $n = n(k, \delta, \delta')$, is defined by an oracle Turing machine $Amp$ that computes a Boolean function on $n$ bits, and an oracle Turing machine $Dec$ that takes non-uniform advice of length $a = a(k, \delta, \delta')$. It holds that For every $f : \{0, 1\}^k \rightarrow \{0, 1\}$, for every $A : \{0, 1\}^n \rightarrow \{0, 1\}$ for which

$$\Pr_{x \in \mathbb{U}_n} [A(x) = Amp^f(x)] > 1 - \delta'$$
there is an advice string \( \alpha \in \{0,1\}^a \) such that

\[
\Pr_{x \in \mathbb{U}_k} \left[ \text{Dec}^A(a, x) = f(x) \right] > 1 - \delta
\]

where \( \text{Dec}^A(a, x) \) denotes running \( \text{Dec} \) with oracle access to \( A \) on input \( x \) and advice \( \alpha \).

If \( \text{Dec} \) does not take non-uniform advice \( (a = |\alpha| = 0) \), then we say that the hardness amplification is \textit{uniform}.

**The Complexity of Hardness Amplification.** We now elaborate on the role that the complexity of \( \text{Dec} \) plays in hardness amplification. Recall that hardness amplification is used to amplify the average-case hardness of functions that are somewhat hard. In particular, suppose we want to obtain from a function \( f : \{0,1\}^k \to \{0,1\} \) that is \( \delta \)-hard against some class \( (\text{of algorithms or circuits}) \ C \), a function \( f' : \{0,1\}^n \to \{0,1\} \) that is \( \delta' \)-hard against \( C \), using a hardness amplification procedure as defined in Definition 6.6.1. For this application, we need a \((\delta, \delta')\)-fully-black-box hardness amplification from length \( k \) to length \( n \) (as above), such that \( \text{Dec} \) itself (as a machine with non-uniform advice) is in the class \( C \). To see this, set \( f' = \text{Amp} f \). Then by contradiction, if there is \( A \in C \) that computes \( f' \) on more than \( 1 - \delta' \) fraction of the instances of length \( n \), then \( \text{Dec}^A(\alpha, \cdot) \) computes \( f \) on more than \( 1 - \delta \) fraction of the instances of length \( k \). Furthermore, \( \text{Dec}^A(\alpha, \cdot) \in C \) (here we assume that \( C \) is informally “closed under oracle access”), which is a contradiction to the \( \delta \)-hardness of \( f \). To summarize, the complexity of \( \text{Dec} \) determines against which class of algorithms or circuits the hardness amplification can be used. In particular, if one wants to use such hardness amplification to amplify hardness against uniform classes of algorithms or circuits, then the hardness amplification must be uniform.

We note that the question of finding functions that are average-case-hard for low complexity classes, such as \( AC^0[q] \), is of central importance for de-randomizing these classes [NW94]. This motivates the study of hardness amplification against such classes, especially since these are the only classes for which (unconditional) mildly average hardness results are known [Raz87, Smo87], and thus there is clear hope of unconditional de-randomization.
We now elaborate: a function $f$ that is very hard on the average (at least $1/2 + 1/\text{polyn}$) for a class can be used in the Nisan-Wigderson construction [NW94], to obtain efficient pseudo-random generators that fool statistical tests in the class. This, in turn, can give a de-randomization of the class. Unfortunately, for classes such as $\text{AC}^0[q]$, no such hardness results are known: [Raz87, Smo87] only give constant hardness (smaller than 1/2) of the mod $p$ function for a prime $p \neq q$. Consequently, we do not know how to unconditionally de-randomize probabilistic $\text{AC}^0[q]$ circuits, even using sub-exponential size deterministic $\text{AC}^0[q]$ circuits.

It is well known [STV01, TV07, Tre03, Vio03] that there is a tight connection between $(2 - k, \delta')$-fully-black-box hardness amplification (or in other words worst-case to average-case reductions) and binary locally (list) decodable codes. We state this fact without proof.

**Proposition 6.6.1.** There is a $(1/2 - \epsilon, \ell)$-locally-list-decodable code $\text{Enc} : \{0,1\}^K \rightarrow \{0,1\}^N$ with a decoder $D$, if and only if there is a $(2^{-k}, 1/2 - \epsilon)$-fully-black-box hardness amplification from length $k = \log K$ to length $n = \log N$ defined by $\text{Amp}$ and $\text{Dec}$, that takes $a = \log \ell$ bits of advice, where $\text{Amp}$ is $\text{Enc}$ and $\text{Dec}$ is $D$.

Our results, both positive and negative, on constructing locally list-decodable binary codes, lead to progress on the complexity of hardness amplification.

**New worst-case to average-case reductions for low complexity classes.** As stated above, it is well known that explicit binary locally list-decodable codes with polynomial rate (and even quasi-polynomial rate) imply worst-case to average-case reductions for the class $\text{EXP' }[\text{BFNW93, IW97, STV01}].$ To see this, consider such a code $C : \{0,1\}^M \rightarrow \{0,1\}^N$ that can be locally and list decoded from agreement $1/2 + \epsilon$. Let $f : \{0,1\}^* \rightarrow \{0,1\}$ be an $\text{EXP'}$-complete function, and let $T_f^n$ be its $M = 2^m$-bit truth-table for input length $m$. Now consider the function $\hat{f}$ whose truth table for input length $cm = \log N$ (for some constant $c$ depending on the rate of the code) is $C(T_f^n)$. $\hat{f} \in \text{EXP'}$ because the code is explicit and $T_f^n$ can be computed in time $2^{md}$ (for some constant $d$). Now suppose that there is an efficient algorithm $A$ that computes $\hat{f}$ correctly on a $1/2 + \epsilon$ fraction of the inputs of length $cm$. Then
the truth table of $A$ at that length has $1/2 + \varepsilon$ agreement with $C(T_7^m)$. This together with the decoder for $C$ gives an algorithm $B$ that solves $f$ correctly on every input (of length $m$) given some non-uniform advice: indeed, by letting $A$ answer the queries of the decoder, we can generate a list of circuits, one of which solves $f$ correctly everywhere, and the advice bits give the index of this circuit in the list. Now, since both the decoder and $A$ are efficient, we obtain an efficient algorithm that solves $f$ correctly on the worst-case. Thus there is an equivalence between the worst-case and average-case hardness of $\mathsf{E XP}$-complete languages, against efficient algorithms.

In the argument above, $B$ runs both the algorithm $A$ (whose efficiency we determine by hypothesis) and the decoder (whose efficiency is a property of the code). Thus the efficiency of the decoder determines the efficiency of the worst-case to average-case reduction, and hence the class of efficient algorithms against which we can establish worst-case/average-case equivalence in $\mathsf{E XP}$. By using our construction from Theorem 1.4.4, we obtain a reduction in non-uniform $\mathsf{AC}^0$ for polylogarithmic $\varepsilon$. However, we can do better. By using ideas from Trevisan and Vadhan [TV07], we can obtain a reduction in \textit{uniform} $\mathsf{AC}^0$. They show how to remove the non-uniform advice bits (assuming that there are not too many of them) by using the fact that the $\mathsf{E XP}$-complete languages have \textit{instance checkers}. Roughly speaking, an instance checker for a language $L$, is an efficient probabilistic oracle algorithm, that on input $x$ computes $L(x)$ when given access to an oracle that computes $L$ correctly, and when given a faulty oracle, either detects that the oracle does not compute $L$ correctly or it still outputs $L(x)$. We refer the reader to [TV07] for the formal definition. Using the fact that there are very efficient instance checkers for $\mathsf{E XP}$-complete languages, we obtain the following theorem.

\textbf{Theorem 6.6.2.} Let $C$ be a class of algorithms (or Boolean circuits) that can compute probabilistic \textit{uniform} $\mathsf{AC}^0$ circuits. Then for every $\mathsf{E XP}$ function $f : \{0, 1\}^* \rightarrow \{0, 1\}$, there is an $\mathsf{E XP}$ function $\hat{f} : \{0, 1\}^* \rightarrow \{0, 1\}$ such that: for every large enough $m$, if there is no algorithm (or family of circuits) in the class $C$ that computes $f$ at length $m$ correctly in the worst-case, then there is no algorithm (or family of circuits) in the class $C$ that can compute
\[
\hat{f} \text{ at length } n = \text{poly}(m) \text{ correctly on at least a } 1/2 + 1/(\log m)^\alpha \text{ fraction of the inputs, where } \\
\alpha > 0\text{ is a universal constant.}
\]

\textit{Proof.} Consider the language \( L = \{(M, x, c) : M \text{ is a deterministic TM that accepts } x \text{ within } c \text{ steps, where } c \text{ is in binary representation}\}. \) It is \( \mathcal{E} \mathcal{X} \mathcal{P} \)-complete under \( \mathcal{A} \mathcal{C}^0 \) reductions. So it is enough to show a worst-case to average-case reduction with the specified parameters from this language. We need the following easy claim.

\textbf{Claim 6.6.3.} \( L \) has an instance checker (with exponentially small error) that can be implemented in \( \mathcal{A} \mathcal{C}^0 \).

\textit{Proof.} One can apply our general approach presented in this paper to obtain the claim. However there is a much simpler way. It is well known that a complete language for a given class (of languages) has an instance checker, if it has a Probabilistic Checkable Proof in which the problem of computing every given bit in the proof can be done within the class. In particular \( \mathcal{E} \mathcal{X} \mathcal{P} \)-complete languages have this property. The computation of the instance checker involves running the PCP verifier as well as the reduction to the complete language.

Now take the PCP proof for an instance of \( L \), and append to it for every possible randomness of the verifier, the tableau of the verifier’s computation (given the input, the randomness and the bits read from the original proof). This is still a PCP proof with the property that every bit of it can be computed in \( \mathcal{E} \mathcal{X} \mathcal{P} \). But now the verifier simply needs to check the correctness of the tableau and this can be done in \( \mathcal{A} \mathcal{C}^0 \). Finally since \( L \) is complete under \( \mathcal{A} \mathcal{C}^0 \) reductions, we obtain an instance checker for \( L \) that can be implemented in \( \mathcal{A} \mathcal{C}^0 \).

We now proceed as in the outline above. Let \( T_m^L \) be the \( M = 2^m \)-bit truth-table of \( L \) for input length \( m \). Consider the function \( \hat{f} \) whose truth-table for input length \( n \) is \( C(T_m^L) \), where \( C \) is the code from Theorem 1.4.4. We set \( \varepsilon \) to be \( 1/\log m)^\alpha = 1/(\log \log M)^\alpha \), where \( \alpha > 0 \) is a constant that ensures that the \( \mathcal{A} \mathcal{C}^0 \) decoder for \( C \) is of size \( \text{poly log } M = \text{poly}(m) \).

By the parameters of \( C \), \( n = \log |C(T_m^L)| = O(m) \).

We prove the contrapositive. Suppose there is an algorithm \( A \) in the class \( C \) that computes \( \hat{f} \) correctly on \( 1/2 + 1/(\log m)^\alpha \) fraction of the inputs of length \( n \). Then we can run the decoder
for $C$ with oracle access to $A$ and obtain $\text{poly log } m$ algorithms (that use $A$ as an oracle) one of which computes $L$ correctly (everywhere) on input length $m$. Since $A \in C$ and the decoder is in $\mathcal{AC}^0$, we conclude that every algorithm in the list is in $C$. Now to solve a given input of length $m$, we run the $\mathcal{AC}^0$ instance checker for $L$ with each one of the algorithms in the list as an oracle (we do that in parallel and with independent random coins). Finally, we compute the approximate majority of the answers we receive from the instance checker. The whole procedure is in uniform $\mathcal{AC}^0$ making oracle calls to $A$, and hence it is in $C$. ■

**Negative Results.** Using this equivalence of fully black-box hardness amplification and local list-decoding, together with Theorem 1.4.6, we can show (informally) that worst-case to average-case hardness amplification with small non-uniform advice requires computing majority. This is stated formally in the theorem below:

**Theorem 6.6.4.** If there is a $(2^{-k}, 1/2 - \varepsilon(k))$-fully-black-box hardness amplification from length $k$ to length $n(k)$ where $Dec$ takes $a(k)$ bits of advice and can be implemented by a circuit of size $s(k)$ and depth $d(k)$, then for every $k \in \mathbb{N}$ there exists a circuit of size $\text{poly}(s(k), 2^{a(k)})$ and depth $O(d(k))$, that computes majority on $O(1/\varepsilon(k))$ bits.

It is known [Raz87, Smo87] that low complexity classes cannot compute majority. Thus, Theorem 6.6.4 shows limits on the amount of hardness amplification that can be achieved by fully-black-box worst-case to average-case reductions (that do not use too many bits of advice), in which $Dec$ can be implemented in low-level complexity classes. I.e. classes that cannot compute majority (e.g. $\mathcal{AC}^0$ and $\mathcal{AC}^0[q]$). The reason is that if there exists hardness amplification for which $Dec$ is in such a class, then by Theorem 6.6.4 there must be a circuit family in the same class for majority, contradicting known circuit lower bounds [Raz87, Smo87]. In particular, the theorem implies that there are no uniform (or even $O(\log 1/\varepsilon)$-non-uniform) $(2^{-k}, 1/2 - \varepsilon)$-fully-black-box worst-case to average-case reductions for $\varepsilon$ smaller than $1/\text{poly log } k$, where $Dec$ is a $\mathcal{AC}^0[q]$ circuit (for a prime $q$) of size $\text{poly}(k, 1/\varepsilon)$. This should be contrasted with [GGH+07] who showed such a fully-black-box reduction (with $Dec$ in $\mathcal{AC}^0$) for $\varepsilon \geq 1/\log^\beta k$, where $\beta$ is a universal constant.
Finally, we note that the worst-case lower bounds (which are actually mildly average-case lower bounds) of [Raz87, Smo87] hold against non-uniform $AC^0[q]$. This means that it may be possible to get the average-case hardness required for pseudo-randomness by using a lot of non-uniformity in a fully-black-box reduction (i.e. a reduction in which $Dec$ takes $\text{poly}(k)$ bits of advice). Shaltiel and Viola [SV08] rule out such non-uniform fully-black-box reductions in the special case that $Dec$ has only non-adaptive access to $A$.

**Extensions.** Theorem 6.6.4 can be extended in two ways: first to rule out hardness amplification from mildly hard functions (and not necessarily worst-case hard) to very hard functions, and second to rule out not necessarily fully black-box hardness amplification.

Let us start with the first direction. Proposition 6.6.1 can be extended to show a similar equivalence between $\delta$-approximate locally $(1/2 - \epsilon, \ell)$-list-decodable codes to $(\delta, 1/2 - \epsilon)$-fully-black-box hardness amplification (with the same translations between the parameters). Let $0 < \alpha < 1/2$ be an arbitrary constant. Theorem 1.4.6 can be extended to show that a $(1/2 - \alpha$)-approximate locally $(1/2 - \epsilon, \ell)$-list-decodable code implies circuits for majority with the same parameters as in the statement of Theorem 6.6.4. Putting the two together we obtain the following.

**Theorem 6.6.5.** Let $0 < \alpha < 1/2$ be an arbitrary constant. If there is a $(1/2 - \alpha, 1/2 - \epsilon(k))$-fully-black-box hardness amplification from length $k$ to length $n(k)$, where $Dec$ takes $\alpha(k)$ bits of advice and can be implemented by a circuit of size $s(k)$ and depth $d(k)$, then for every $k \in \mathbb{N}$ there exist a circuit of size $\text{poly}(s(k), 2^{\alpha(k)})$ and depth $O(d(k))$, that computes majority on $1/\epsilon(k)$ bits.

We conclude with an informal discussion about not necessarily fully black-box hardness amplification. Note that in definition 6.6.1, the hardness amplification is required to work for every function $f$. A more relaxed notion is (not necessarily fully) black-box reductions:

---

9The proof follows the outline of the proof of Theorem 1.4.6 using the fact that from error rate $1/2$ we cannot recover more than $1/2 + \alpha/2$ of the bits of the message $m$, while from error rate $1/2 - \epsilon$ we can recover at least $1/2 + \alpha$ of the bits. So by sampling bits from $m$ we can distinguish between the two cases.
Definition 6.6.6. A $(\delta, \delta')$-black-box hardness amplification from $f : \{0, 1\}^k \to \{0, 1\}$ to $f' : \{0, 1\}^n \to \{0, 1\}$ is defined by an oracle Turing machine $Dec$ that takes non-uniform advice of length $a = a(k, \delta, \delta')$ and the following holds; for every $A : \{0, 1\}^n \to \{0, 1\}$ for which

$$\Pr_{x \in \{0, 1\}^n} [A(x) = f'(x)] > 1 - \delta'$$

there is an advice string $a \in \{0, 1\}^a$ such that

$$\Pr_{x \in \{0, 1\}^n} [Dec^A(a, x) = f(x)] > 1 - \delta$$

This is relaxation of fully-black-box hardness amplification. In this case, the hardness amplification is not required to work for any function, but only for a specific and known function. Suppose we have a function $f$ that we already know is worst-case hard, or even $\delta$-hard on the average, against a low level class such as $\mathcal{AC}_0^0[q]$. Perhaps we can use specific properties of the function $f$ (e.g. random self-reducability) to construct a function $f'$, such that there is a $(\delta, 1/2 - 1/poly(n))$-black-box hardness amplification from $f$ to $f'$ that can be implemented by $\mathcal{AC}_0^0[q]$ circuits. This would not be a fully-black-box hardness amplification result, but it certainly suffices for de-randomization applications (in fact, usually for de-randomization one uses a specific and explicit hard function).

We note that the results of Theorems 6.6.4 and 6.6.5 can be extended to show that if a function $f$ is $\delta$-hard on the average for a low complexity class, and furthermore, there is a uniform (or even somewhat non-uniform) $(\delta + 1/poly(k), 1/2 - \varepsilon(k))$-hardness amplification from $f$ to any other function $f'$, where $Dec$ is of size $s(k)$ and depth $d(k)$, then there exists a circuit of similar size and depth that computes majority on $O(\varepsilon(k))$-bit inputs.

The basic idea is similar to the proof of Theorem 6.6.4. The decoder cannot, given an oracle for $f'$ that is only correct with probably $1/2$ (over the inputs), recover $f$ with probability greater than $1 - \delta$. This is because doing so would contradict the hardness of $f$: computing any $f'$ with error rate $1/2$ is computationally easy, so the oracle can be simulated by an $\mathcal{AC}_0^0$ circuit, and we get a circuit for computing $f$. On the other hand, the
reduction *does* recover from error rate $1/2 - \varepsilon(k)$, computing $f$ correctly with probability $1 - \delta - \text{poly log}(k)$. This gives a distinguisher between error rates $1/2$ and $1/2 - \varepsilon(k)$, which in turn (as in the proof of Theorem 6.6.4) leads to an algorithm for computing majority on $O(\varepsilon(k))$ bits. The full details are omitted.
Bibliography


288


289


290


[SET99] ET, phone SETI@home!. Science@NASA headlines, 1999.


