Electronic Colloquium on Computational Complexity, Revision 1 of Report No. 108 (2017)

Delegating Computation: Interactive Proofs for Muggles*

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

Yael Tauman Kalai
Microsoft Research
yael@microsoft.com

Guy N. Rothblum
Weizmann Institute
rothblum@alum.mit.edu

Abstract

In this work we study interactive proofs for tractable languages. The (honest) prover should be efficient and run in polynomial time, or in other words a "muggle".[1] The verifier should be super-efficient and run in nearly-linear time. These proof systems can be used for delegating computation: a server can run a computation for a client and interactively prove the correctness of the result. The client can verify the result's correctness in nearly-linear time (instead of running the entire computation itself).

Previously, related questions were considered in the Holographic Proof setting by Babai, Fortnow, Levin and Szegedy, in the argument setting under computational assumptions by Kilian, and in the random oracle model by Micali. Our focus, however, is on the original interactive proof model where no assumptions are made on the computational power or adaptiveness of dishonest provers.

Our main technical theorem gives a public coin interactive proof for any language computable by a log-space uniform boolean circuit with depth d and input length n. The verifier runs in time $n \cdot \mathrm{poly}(d, \log(n))$ and space $O(\log(n))$, the communication complexity is $\mathrm{poly}(d, \log(n))$, and the prover runs in time $\mathrm{poly}(n)$. In particular, for languages computable by log-space uniform $\mathcal{NC}$ (circuits of $\mathrm{polylog}(n)$ depth), the prover is efficient, the verifier runs in time $n \cdot \mathrm{polylog}(n)$ and space $O(\log(n))$, and the communication complexity is $\mathrm{polylog}(n)$. Using this theorem we make progress on several questions:

- We show how to construct 1-round computationally sound arguments with polylog communication for any log-space uniform $\mathcal{NC}$ computation. The verifier runs in quasi-linear time. This result uses a recent transformation of Kalai and Raz from public-coin interactive *proofs* to one-round *arguments*. The soundness of the argument system is based on the existence of a PIR scheme with polylog communication.
- Interactive proofs with *public-coin*, log-space, poly-time verifiers for all of $\mathcal{P}$. This settles an open question regarding the expressive power of proof systems with such verifiers.
- Zero-knowledge interactive proofs with communication complexity that is quasi-linear in the *witness* length for any $\mathcal{NP}$ language verifiable in $\mathcal{NC}$, based on the existence of one-way functions.
- Probabilistically checkable arguments (a model due to Kalai and Raz) of size polynomial in the *witness* length (rather than the instance length) for any $\mathcal{NP}$ language verifiable in $\mathcal{NC}$, under computational assumptions.

*An extended abstract of this work appeared in STOC 2008, this is the full version. This work was supported by NSF grants CCF-0514167, CCF-0635297, NSF-0729011, CNS-0430336, by the RSA chair, by the Weizmann Chais Fellows, and by a Symantec Graduate Fellowship.

[1] *In the fiction of J.K. Rowling: a person who possesses no magical powers*; from the Oxford English Dictionary.

ISSN 1433-8092

Contents

1 Introduction

The power of efficiently verifiable proof systems is a central question in the study of computation. Classically, this was captured by the class $\mathcal{NP}$. There, a deterministic polynomial time verification procedure 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, 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 in the form of 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).

Prior to our work, interactive proofs were studied through the lenses of cryptography and of complexity theory. Both these settings focus on efficient proof verification without the requirement of *efficient proof generation*. In particular:

Complexity-theoretic setting. Past 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.

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 modulos in the quadratic residuosity example. This model is reasonable in settings where the input is generated by the prover himself. The prover can generate the input along with an auxiliary secret that enables him prove non-$\mathcal{BPP}$ properties. However, in settings (like ours) 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 tractable statements, where both the verifier *and the prover* are efficient. We thus require that the honest prover to be 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, which means that the prover has no auxiliary information to help him in the proving task, but rather should generate the proof efficiently.

We are motivated by applications of interactive proofs to proving the correctness of delegated (polynomial-time) computations. A *delegator* sends a computation to an untrusted *delegatee*, and seeks a proof that the computation was performed correctly. Here, the *statement to be proved* is that the delegated computation was executed correctly (i.e. that the given output is the correct one); the

[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.

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).

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. Specifically, let L be an efficiently computable language. We seek an interactive proof system that achieves:

- *Verifier* time complexity that is *linear* in the size of the input x and poly-logarithmic in the size of the computation of L. More generally, we ask that the verifier run in time and space that are considerably smaller than those required to compute the language.
- *Prover* time complexity that is polynomial in the size of the input.
- *Communication complexity* that is polylogarithmic in the size of the computation. More generally, we ask that the communication complexity be considerably smaller than the running time required to compute the language.

Delegating Computation. Beyond its complexity theoretic interest, the question of interactive proofs for efficient players is motivated by real-world applications. The main application we consider is delegating polynomial time computations to an untrusted party. 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.[3] 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. 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 delegator that he performed the computation correctly (and runs in polynomial time).

Roadmap for Section 1. We begin with an overview and discussion of our results. Our main result is described in Section 1.1. We further use our techniques to obtain several other results: constructing computationally-sound one-round argument systems for any ($\mathcal{L}$-uniform) $\mathcal{NC}$ computation, under computational assumptions (Section 1.2); Characterizing public-coin log-space interactive proofs (Section 1.3); Constructing low communication zero-knowledge proofs (Section 1.5); Constructing Interactive PCPs (IPCP) and Probabilistically Checkable Arguments (PCA), improving on [KR08, KR09] (Section 1.6). We then survey subsequent related works on delegating computation, which followed the original publication of our work (Section 1.7). A high-level overview of our techniques is given Section 1.8. We proceed in Section 2 with preliminaries. The full protocols, proofs and technical details are presented in the subsequent sections.

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 $\mathcal{L}$-uniform (or, more generally, can be generated using a log-space Turing

[3]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.

Machine).[4] We view this as a relaxed notion of uniformity. In particular, it captures logarithmic space uniform computations and uniform parallel computing classes; See the discussion following Corollary 1.2. 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.*

Theorem 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 $\mathrm{poly}(S(n))$. The verifier runs in time $n \cdot \mathrm{poly}(d(n), \log S(n))$ 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 $\mathrm{poly}(d(n), \log S(n))$.*

2. *The protocol has perfect completeness and soundness $1/2$.*[5]

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

An overview of the proof idea is given in Section 1.8; See Section 4 (and 3) for a full proof. The protocol in the proof of Theorem 1.1 provides 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 delegator 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 $\mathcal{L}$-uniform $\mathcal{NC}$, we have:

Corollary 1.2. *Let L be a language in $\mathcal{L}$-uniform $\mathcal{NC}$, i.e. computable by a family of $O(\log(n))$-space uniform circuits of size $\mathrm{poly}(n)$ and depth $\mathrm{polylog}(n)$. L has an interactive proof where:*

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

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

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

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 1.2 applies to any language in $\mathcal{NL}$, and even to any language computable by a $PRAM$ in poly-logarithmic parallel time

[4] 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 $\mathcal{L}$-uniform if it is log-space uniform.

[5] Throughout this work we focus on interactive proof systems with constant soundness. Soundness can be amplified via parallel or sequential repetition.

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 1.4 and 1.8 for more details.

Comparison to Prior Work on Interactive Proofs. We emphasize that Theorem 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 1.2 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 $\#\mathcal{P}$ and for $\mathcal{PSPACE}$, and thus naturally the honest prover needs to perform non-polynomial time computation (scale-down of the protocols to $\mathcal{P}$ or even to $\mathcal{L}$ retains the non-polynomial time provers). 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 $\mathcal{NC}$. They show how to improve the space complexity of the verifier, in particular achieving log-space and poly-time verifiers for $\mathcal{NC}$ computations. Their protocol, however, has a non-polynomial time prover as in [LFKN92, Sha92].

Our work puts severe restrictions on the runtime of the verifier (we also consider the space used by the verifier). This continues a sequence of works which investigated the power of interactive proofs with weak verifiers, often with unbounded (honest) 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-bounded verifiers. Kilian [Kil88] 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. A recent work by Goldwasser *et al.* [GGH$^+$07] considers the *parallel running time* of a verifier (who can still use a polynomial number of processors and communication complexity), and shows that all of PSPACE can still be recognized by constant-depth ($\mathcal{NC}^0$) verifiers.

Comparison to Prior Work in Other Models. The goal of the work of Babai, Fortnow, Lund and Szegedy [BFLS91] on Holographic Proofs for $\mathcal{NP}$ (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], [DR06], [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 $\mathcal{NP}$ 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.[6] 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.

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 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. We use the protocol of Theorem 1.1 together with a transformation of Kalai and Raz [KR09] (see below) to construct one-round arguments for any $\mathcal{L}$-uniform $\mathcal{NC}$ computation, assuming the existence of a computational private information retrieval (PIR) scheme with poly(κ)-communication, where κ 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 sub-exponential hardness of LWE [BV11], N-th Residuosity [Lip05], [IP07], and under the Φ-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

[6]With standard intractability assumptions, one could get arguments of small but polynomial communication complexity (using universal arguments [BG02]).

(note, however, that we make no claims for soundness if the verifier uses the same challenge more than once).

For security parameter κ, the size of the certificates is $\mathrm{poly}(\kappa, \log n)$, the (honest) prover runs in polynomial time, and the verifier runs in time $n \cdot \mathrm{poly}(\kappa, \log n)$ to verify a certificate (as in [Mic94]). Soundness holds only against computationally bounded cheating provers (see Section 2.6 for a definition and more details about the computational assumption).

Theorem 1.3. *Let L be a langauge 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 $\mathrm{poly}(\kappa)$, receiver work $\mathrm{poly}(\kappa)$, and sender work $\mathrm{poly}(n, \kappa)$ (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 $\mathrm{poly}(S(n))$, the verifier runs in time $n \cdot \mathrm{poly}(\kappa, d(n), \log(S(n)))$.*[7]

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 $\leq 2^{\kappa^3}$, 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 $\mathrm{poly}(\kappa, d(n))$. The verifier's short challenge depends only on the parameters n, $d(n)$, and κ, 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 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, taking κ to be a security parameter, [KR09] show how to convert any public-coin interactive proof system $(\mathcal{P}, \mathcal{V})$ (for a language L), with communication complexity ℓ, completeness c, and soundness s, into a one-round (two-message) argument system $(\mathcal{P}', \mathcal{V}')$ (for L), with communication complexity $\mathrm{poly}(\ell, \kappa)$, completeness c, and soundness $s + 2^{-\kappa^2}$ against malicious provers of size $\leq 2^{\kappa^3}$. The verifier $\mathcal{V}'$ runs in time $t_{\mathcal{V}} \cdot \mathrm{poly}(\kappa)$, where $t_{\mathcal{V}}$ is $\mathcal{V}$'s running time. The prover $\mathcal{P}'$ runs in time $t_{\mathcal{P}} \cdot \mathrm{poly}(\kappa, 2^{\lambda})$, where $t_{\mathcal{P}}$ is $\mathcal{P}$'s running time, and λ satisfies that each message sent by the prover $\mathcal{P}$ depends only on the λ previous bits sent by $\mathcal{V}$.

Note that if λ is super-logarithmic, then the resulting prover $\mathcal{P}'$ is inefficient. Fortunately, the protocol of Theorem 1.1 has the property that $\lambda = O(\log(S(n))$. Applying the transformation to the protocol yields a 1-round computationally sound argument system, where the resulting honest $\mathcal{P}'$ runs in time $\mathrm{poly}(S(n))$. For further details, see Section 6.

Applying [KR09] to Other Interactive Proofs. We note that the transformation of [KR09] 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 $\kappa = \mathrm{poly}(n)$,

[7]Moreover, if the verifier is given oracle access to the low-degree extension of its input, then its running time is only $\mathrm{poly}(\kappa, d(n), \log(S(n)))$.

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.

1.3 Public-Coin Log-Space Verifiers

Theorem 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 $\mathcal{P}$. 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 $\mathcal{NC}$. 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, and using the fact that languages in $\mathcal{P}$ have $\mathcal{L}$-uniform poly-size circuits, we show the following theorem (see Section 4.2 for details):

Corollary 1.4. *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* $1/2$.
3. *The protocol is public-coin, with communication complexity* $\text{poly}(n)$.

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 1.5. *Let L be a language computable by a (non-uniform) circuit family $\mathcal{C}$ of size $S(n)$ and depth $d(n)$. There exists an on-line/off-line interactive proof $(\mathcal{P}(C,x), \mathcal{V}(x, data), \mathcal{V}_{pre}(C))$ 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 $\mathcal{V}_{pre}(C)$ takes time* $\text{poly}(S(n))$. *The output data is of length* $|data| = \text{poly}(d(n), \log(S(n)))$.

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

3. *The on-line verifier $\mathcal{V}(x, data)$ 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* $\text{poly}(d(n), \log(S(n)))$.

See Section 4.3 for the details.

1.5 Succinct Zero Knowledge Proofs

Aside from the primary interest (to us) of delegating computation, Theorem 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 κ is a security parameter. 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 R 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 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 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 $k \cdot poly(\kappa, d(n))$.*

In particular, for relations R in $\mathcal{NC}$, the protocol of Theorem 1.6 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, such as "The following is the result of proper decryption" or "The following is a result of a pseudo-random 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, 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 5.

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

Theorem 1.7. *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 $k \cdot poly(\kappa, d(n))$.*

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.

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., $R \in AC^0$) 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 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 $\mathrm{poly}(\log n, d)$. 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[8] (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 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 $\mathrm{poly}(n)$.

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 $\mathrm{poly}(\text{witness size}, \log n, d)$ and query complexity $\mathrm{poly}(\log n, d)$ 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 8 for details and theorem statements.

1.7 Subsequent Work

Following our work, a literature spanning both theory and practice has considered the question of delegating computations reliably. We survey the most closely related of these advancements below. Note that there are many other results that we do not mention, which consider various different models, or are concerned with practical efficiency.

Significant attention has been devoted to the construction of 2-message computationally sound delegation schemes. As noted above, Kalai and Raz [KR09] give a transformation from (many-round) Interactive Proofs to 2-message delegation schemes. Combined with our work, this gives a 2-message delegation scheme for any $\mathcal{L}$-uniform $\mathcal{NC}$ computation, using sub-exponentially secure PIR or Fully Homomorphic Encryption [RAD78, Gen09] (more generally, for any $\mathcal{L}$-uniform circuit of depth d, the verifier's computational complexity grows with d).

More recently, there has been a proliferation of two-message delegation schemes. Many of these results construct a 2-message delegation scheme for $\mathcal{NP}$ languages under *non-falsifiable* assumptions (see Naor [Nao03]). These works include [Gro10, Lip12, BCCT12, DFH12, GLR11, GGPR13, BCCT13]. Indeed, Gentry and Wichs [GW11] show barriers to basing the security of delegation schemes for $\mathcal{NP}$ languages on *falsifiable* assumptions. A different series of results construct 2-message delegation schemes in a *preprocessing model*, where the verifier is efficient only in the amortized setting. These results include [GGP10, CKV10, AIK10, PRV12] (as well as several of the works based on non-falsifiable assumptions).

[8]In both this work and in [KR08], the prover always runs in polynomial time.

Very recently, Kalai, Raz and Rothblum [KRR13, KRR14] showed that no-signalling multi-prover interactive proofs (NS MIPs) can be used to construct 2-message delegation schemes, and leveraged this connection to construct a 2-message delegation scheme for any polynomial time computation (more generally, for a computation that requires time t, the verifier's computational complexity grown polynomially with $\log(t)$). This new scheme's security can be based on sub-exponentially secure PIR or Fully Homomorphic Encryption.

Another related work of [CKLR11] studies the problem of *memory delegation*, where even the input is too long for the verifier to store. They propose a model where the verifier is allowed to run a preprocessing computation on an input, and can then verify the results of subsequent computations in sub-linear time (in particular, the verifier can delegate even the input storage). They propose computationally sound protocols based on cryptographic assumptions.

Rothblum, Vadhan and Wigderson [RVW13] study sublinear time verification for Interactive Proofs. In this model, called *Interactive Proofs of Proximity*, the verifier is allowed (sublinear-time) query access to the input, and can verify that the input is "close" to the language (or, alternatively, that a result is approximately correct). Unlike the setting of memory delegation, there is no pre-processing stage, and the verifier only gets sublinear-time query access to the input (via an oracle). In particular, most bits of the input are never read by the verifier. Building on our work, they construct such an Interactive Proof of Proximity (with information theoretic soundness) for any language in $\mathcal{NC}$. They also propose more efficient protocols for specific languages. Gur and Rothblum [GR13] study non-interactive (Merlin-Arthur) Proofs of Proximity.

Beyond the above works in the theory literature, several recent works have proposed and constructed systems for delegating computations. Cormode Mitzenmacher and Thaler [CMT12] gave the first implementation of a delegation system, with a protocol based on our work. Other systems based on the protocols proposed in this work include Thaler, Roberts, Mitzenmacher and Pfister [TRMP12], Thaler [Tha13], and Vu, Setty, Blumberg and Walfish [VSBW13]. Indeed, by now there are several different works on this topic using different underlying theoretical results. See Walfish and Blumberg [WB13] for a survey on this line of work.

1.8 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. Let C

be a depth d arithmetic circuit, i.e. the circuit C is composed of addition and multiplication gates over the finite field $\mathbb{GF}[2]$ with fan-in 2. Assume, without loss of generality, that the circuit is in a layered form, where there are as many layers as the depth of the circuit.[9]

In previous work, spanning both the single and multi prover models [LFKN92, Sha92, BFL91, KR08],[10] 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 bottom (input) of the circuit.[11] 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 low-degree extension of the first step in the computation. The first step of the computation is simply the input layer, and 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 an i-th layer in the circuit, to verification of a single point in the low degree extension of the "earlier" $(i+1)$-th layer.[12]

The main ingredient we use 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 layer i is a linear combination, or a weighted sum, of the values of that layer's gates. Each gate in layer i is a function of the values of two gates in layer $i+1$ (because we assumed that C is a layered circuit with fan-in 2). Thus, we can express the value of each point in the LDE of layer i as a weighted sum, over all gates g in layer i, and over all possible gate-pairs (k, ℓ) in layer $(i+1)$, of a low degree function of: (i) the values of gates k and ℓ, and (ii) a predicate that indicates whether gates k and ℓ 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, ℓ) 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.

[9]Every circuit can be converted into this format, without increasing its depth. The size is at most squared.

[10]One exception is the work of Feige and Kilian on refereed games [FK97], which is in a different model.

[11]I.e., layer 0 is the output layer, and layer d is the input layer.

[12]We note that at first glance this may seem similar to a problem faced by [FK97] in their work on refereed games. They also examine the computation step by step or layer by layer, but they do this for a *sequential* (exponential-time) computation.

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 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 ($\mathcal{AC}^0$) 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 $\mathcal{L}$ 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.

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 and 4 for details.

Organization of the Exposition. The full exposition of the main result (Theorem 1.1) is organized in two phases. First, to highlight and clarify some of the new technical ideas, we present in Section 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 3.1 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 4 we show how to *implement* the above bare-bones protocol for uniform circuits. We begin in Section 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 $\mathcal{NL}$ languages, the result is stated in that section as Theorem 4.4. We then proceed in Section 4.2 with a general result for $\mathcal{L}$-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 $\mathcal{L}$ computations. Thus, we can use the interactive proofs for $\mathcal{NL}$ computations (of Theorem 4.4) as a sub-protocol, where the prover proves the correctness of his answers. This gives the result of Theorem 1.1.

The remainder of the paper is devoted to applications and consequences. In Section 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 6. Section 7 shows constructions of new and improved IPCPs. Finally, in Section 8 we present new construction of PCAs.

2 Preliminaries

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

2.1 Turing Machines, Circuits, and Complexity Classes

We refer to Turing Machines running in time $t(n)$ and/or space $s(n)$, where $t(\cdot)$ and $s(\cdot)$ are functions from natural numbers to natural number. Throughout, we implicitly assume that $t(\cdot)$ and $s(\cdot)$ are time-constructible and space-constructible (respectively).

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

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.

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 an 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 2.1. 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) 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 $x \notin L$ 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.

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., [Gol99]). We denote by $\mathcal{AM}$ (i.e., Arthur-Merlin games) the class of languages that have protocols with a constant number of interaction rounds.

2.3 Low Degree Extension

Fix $\mathbb{H}$ to be an extension field of $\mathbb{GF}[2]$, and fix $\mathbb{F}$ to be an extension field of $\mathbb{H}$ (and in particular, an extension field of $\mathbb{GF}[2]$), where $|\mathbb{F}| = \text{poly}(|\mathbb{H}|)$.[13] 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 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 α 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

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

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 2.2. *There exists a Turing machine that takes as input an extension field $\mathbb{H}$ of $\mathbb{GF}[2]$,[14] 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 $k \leq |\mathbb{H}|^m$, and for every $z \in \mathbb{F}^m$,*

$$\tilde{W}(z) = \sum_{p \in \mathbb{H}^m} \tilde{\beta}(z, p) \cdot W(p), \tag{2}$$

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 (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 $\tilde{\beta}(z, p)$.*

The above Proposition is well-known. For completeness, we present the proof below.

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

[14] 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.

Proof of Proposition 2.2. 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 β, of degree at most $|\mathbb{H}| - 1$ in each variable. It is easy to see that $\tilde{\beta}$ satisfies Equation (2), since it satisfies Equation (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 $O(\log(|\mathbb{H}|))$.

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 β on $\mathbb{H}^m \times \mathbb{H}^m$, and is a polynomial of degree $|\mathbb{H}| - 1$ in each variable. ■

Claim 2.3. There exists a Turing machine that takes as input an extension field $\mathbb{H}$ of $\mathbb{GF}[2]$, 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 2.3. The proof is a direct corollary of Proposition 2.2. Let $W : \mathbb{H}^m \to \mathbb{F}$ be the function corresponding to w, as defined in Equation (1). By Equation 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 2.2, 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 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

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

with respect to $\mathbb{F}, \mathbb{H}, m$, where again $k \leq |\mathbb{H}|^m$.

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 $W : \mathbb{H}^m \to \mathbb{F}^\ell$ (in particular, W is again 0 on inputs whose lexicographic order is $|\mathbb{H}^m|$ or more). As before, there is a unique extension of W into a function $\tilde{W} : \mathbb{F}^m \to \mathbb{F}^\ell$ which agrees with W on $\mathbb{H}^m$, and where each of the outputs is a function of degree at most $|\mathbb{H}| - 1$ in every input variable. As before, the function $\tilde{W}$ is called the low-degree extension of w with respect to $\mathbb{H}, \mathbb{F}, m$ and denoted (as usual) by $\mathrm{LDE}_{\mathbb{H},\mathbb{F},m}(w)$.

Finally, note that $\tilde{W}$ can be expressed as the low degree extensions of ℓ standard functions (from $\mathbb{H}^m$ to $\mathbb{F}$), each computing one of the ℓ items in W's output. By Claim 2.3, the function $\tilde{W}$ can be expressed as an arithmetic circuit over $\mathbb{F}$, that can be generated and evaluated in time $|\mathbb{H}|^m \cdot \mathrm{poly}(|\mathbb{H}|, m, \ell)$ and space $O(\log(\ell) + m \cdot \log(|\mathbb{H}|))$.

2.4 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 7, the section on interactive PCPs.

Fix a finite field $\mathbb{F}$. 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 π being close to an m-variate polynomial of degree $\leq d$. This proof should be short (say, of size $\leq \mathrm{poly}(|\mathbb{F}|, m)$). The verifier has only oracle access to π, and is allowed to query π at only a few points (say, only one point).

For this work (specifically, for the application to interactive PCPs), we only need a low-degree test with constant error. Thus, we could use a simple low-degree test, such as the one of Rubinfeld and Sudan [RS96]. For the sake of convenience, we use a more powerful low-degree test (with smaller error), due to Moshkovitz and Raz [MR08]. This low-degree test is described in Figure 1, and is denoted by $(P_{\mathrm{LDT}}(\pi), V_{\mathrm{LDT}}^{\pi})$.

Lemma 2.4. *For any $m \geq 3$ and $1 \leq d \leq |\mathbb{F}|$, the low degree test $(P_{\mathrm{LDT}}(\pi), V_{\mathrm{LDT}}^{\pi})$ described in Figure 1 has the following guarantees.*

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

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

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

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

then there exists an m-variate polynomial $f : \mathbb{F}^m \to \mathbb{F}$ of total degree $\leq d$, such that $\Pr_{z \in \mathbb{F}^m}[\pi(z) = f(z)] \geq \gamma - \varepsilon$, where

$$\varepsilon \overset{\mathrm{def}}{=} 2^{10} m \sqrt[8]{\frac{md}{|\mathbb{F}|}}.$$

Low Degree Test for $\pi : \mathbb{F}^m \to \mathbb{F}$

1. The verifier chooses uniformly and independently $z_1, z_2, z_3 \in_R \mathbb{F}^m$. 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 $\eta : \mathbb{F}^3 \to \mathbb{F}$, which is supposedly the function π 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 η 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 $\alpha_1, \alpha_2, \alpha_3 \in_R \mathbb{F}$ and setting $z = \alpha_1 z_1 + \alpha_2 z_2 + \alpha_3 z_3$. He queries the oracle π at the point z, and accepts if and only if
$$\eta(\alpha_1, \alpha_2, \alpha_3) = \pi(z).$$

Figure 1: Low degree test $(P_{\text{LDT}}(\pi), V^{\pi}_{\text{LDT}})$

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

We refer the reader to [MR08] for a proof of Lemma 2.4.

2.5 Interactive Sum-Check Protocol

Fix a finite field $\mathbb{F}$ and a subset $H \subseteq \mathbb{F}$. In a sum-check protocol, a (not necessarily efficient) prover takes as input an m-variate polynomial $f : \mathbb{F}^m \to \mathbb{F}$ of degree $\leq d$ in each variable (think of d as significantly smaller than $|\mathbb{F}|$). His goal is to convince a verifier that
$$\sum_{z \in H^m} f(z) = \beta,$$
for some constant $\beta \in \mathbb{F}$. The verifier only has oracle access to f, and is given the constant $\beta \in \mathbb{F}$. He is required to be efficient in both its running time and its number of oracle queries. In Figure 2, we review the standard sum-check protocol, as it appeared for example in [LFKN92, Sha92]. We denote this protocol by $\left(P_{\text{SC}}(f), V^f_{\text{SC}}(\beta)\right)$.

Lemma 2.5. *Let $f : \mathbb{F}^m \to \mathbb{F}$ be an m-variate polynomial of degree at most d in each variable, where $d < |\mathbb{F}|$. The sum-check protocol $\left(P_{\text{SC}}(f), V^f_{\text{SC}}(\beta)\right)$, described in Figure 2, satisfies the following properties.*

- **Completeness:** *If $\sum_{z \in H^m} f(z) = \beta$ then*
$$\Pr\left[\left(P_{\text{SC}}(f), V^f_{\text{SC}}(\beta)\right) = 1\right] = 1.$$
- **Soundness:** *If $\sum_{z \in H^m} f(z) \neq \beta$ then for every (unbounded) interactive Turing machine $\tilde{P}$,*
$$\Pr\left[\left(\tilde{P}(f), V^f_{\text{SC}}(\beta)\right) = 1\right] \leq \frac{md}{|\mathbb{F}|}.$$

Sum-Check Protocol for $\sum_{t_1,\ldots,t_m \in H} f(t_1,\ldots,t_m) = \beta$

- 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

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

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

- **Complexity:** *$P_{\text{SC}}(f)$ is an interactive Turing machine, and $V_{\text{SC}}^f(\beta)$ is a probabilistic interactive Turing machine with oracle access to $f : \mathbb{F}^m \to \mathbb{F}$. The prover $P_{\text{SC}}(f)$ runs in time $\leq \text{poly}(|\mathbb{F}|^m)$.*[15] *The verifier $V_{\text{SC}}^f(\beta)$ runs in time $\leq \text{poly}(|\mathbb{F}|, m)$ and space $O(\log(|\mathbb{F}|) \cdot m)$, 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*

[15] 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 $\leq \text{poly}(|\mathbb{F}^m|)$.

consist of the verifier's random coin tosses.

For completeness, we include a proof of this lemma below.

Proof of Lemma 2.5: 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 $\sum_{z\in H^m} f(z) \neq \beta$. Assume for the sake of contradiction that there exists a cheating prover $\tilde{P}$ for which

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

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 $\left(\tilde{P}(f), V^f_{\text{SC}}(\beta)\right) = 1$. Notice that $\Pr[S|A_1 \wedge \ldots \wedge A_m] = 0$. We will reach a contradiction by proving that

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

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

$$\Pr[S|A_j \wedge \ldots \wedge A_m] \geq s - \frac{(m-j+1)d}{|\mathbb{F}|}. \qquad (3)$$

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,

$$\Pr[S|A_m] \geq s - \frac{d}{|\mathbb{F}|}.$$

Assume that Equation (3) holds for j, and we will show that it holds for $j-1$.

$$\begin{aligned} s - \frac{(m-j+1)d}{|\mathbb{F}|} \leq & \Pr[S|A_j \wedge \ldots \wedge A_m] \leq \\ & \Pr[S|\neg(A_{j-1}) \wedge A_j \wedge \ldots \wedge A_m] + \Pr[S|A_{j-1} \wedge A_j \wedge \ldots \wedge A_m] \leq \\ & \frac{d}{|\mathbb{F}|} + \Pr[S|A_{j-1} \wedge \ldots \wedge A_m], \end{aligned}$$

which implies that

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

as desired. ■

2.6 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. Kushilevitz and Ostrovsky [KO97] were the first to construct a single-database PIR scheme (with computational security).

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 κ, 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 be (computationally) hiding, and reveal no information about the index i. It also outputs an additional secret s, which will help the user to retrieve the desired element from the database. The database algorithm D^{PIR} takes as input a security parameter κ, the database $(x_1, \ldots, x_N)$ 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 κ, 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.[16]

Definition 2.6. Let κ 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 $\leq \text{poly}(\kappa, \log N)$, and D^{PIR} is of size $\leq \text{poly}(\kappa, N)$. The output of Q^{PIR} and D^{PIR} is of size $\leq \text{poly}(\kappa, \log N)$.

2. (Correctness:) $\forall N$, $\forall \kappa$, $\forall$database $x = (x_1, \ldots, x_N) \in \{0,1\}^N$, and $\forall i \in [N]$,
$$\Pr[R^{PIR}(\kappa, N, i, (q,s), a) = x_i \mid (q,s) \leftarrow Q^{PIR}(\kappa, N, i), a \leftarrow D^{PIR}(\kappa, x, q)] \geq 1 - 2^{-\kappa^3}.$$

3. (User Privacy:) $\forall N$, $\forall \kappa$, $\forall i, j \in [N]$, and $\forall$ (possibly non-uniform) adversary $\mathcal{A}$ of size at most 2^{κ^3},
$$\begin{aligned}\big| &\Pr[\mathcal{A}(\kappa, N, q) = 1 \mid (q,s) \leftarrow Q^{PIR}(\kappa, N, i)] - \\ &\Pr[\mathcal{A}(\kappa, N, q) = 1 \mid (q,s) \leftarrow Q^{PIR}(\kappa, N, j)]\big| \leq 2^{-\kappa^3}.\end{aligned}$$

3 The Bare-Bones Protocol for Delegating Computation

Our goal is constructing a protocol in which a prover, who is given a circuit $C : \{0,1\}^n \to \{0,1\}$ of size S and of depth d, and a string $x \in \{0,1\}^n$, proves to a verifier that $C(x) = 0$. The verifier's

[16]Definition 2.6 is not worded exactly as the one in [CMS99], but was shown to be equivalent to it in [KR06].

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 1.8. 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, since we give the verifier access to this oracle. We defer fully specifying the oracle function to Subsection 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 1.8 and the full Details in Section 4.

Let $C : \{0,1\}^n \to \{0,1\}$ be a boolean circuit. We now proceed to present our first result, the *bare-bones* protocol, denoted by $(\mathcal{P}_1, \mathcal{V}_1)$, for efficiently proving that $C(x) = 0$. The prover and the verifier take as input a string $x \in \{0,1\}^n$, and are both given oracle access to the function $\mathcal{F}$ specifying C (as defined in Subsection 3.1), where $\mathcal{F}$ is of degree poly-logarithmic in S. In the protocol, the verifier $\mathcal{V}_1$'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 $\mathcal{P}_1$ will remain efficient.

In Section 4 we show how to convert this bare-bones protocol into a standard interactive proof without any oracles, 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.

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 3.1. *Let $C : \{0,1\}^n \to \{0,1\}$ 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 3.1, of* $\mathrm{polylog}(S)$*-degree. Protocol $(\mathcal{P}_1^{\mathcal{F}}(x), \mathcal{V}_1^{\mathcal{F}}(x))$ has the following properties:*

- **Completeness:** *If $C(x) = 0$ then*

$$\Pr\left[(\mathcal{P}_1^{\mathcal{F}}(x), \mathcal{V}_1^{\mathcal{F}}(x)) = 1\right] = 1$$

- **Soundness:** *If $C(x) \neq 0$ then for every (unbounded) interactive Turing machine $\mathcal{P}^*$,*

$$\Pr\left[(\mathcal{P}^{*\mathcal{F}}(x), \mathcal{V}_1^{\mathcal{F}}(x)) = 1\right] \leq \frac{1}{100}$$

- **Complexity:** *The running time of the prover $\mathcal{P}_1$ is* $\mathrm{poly}(S)$. *The running time of the verifier $\mathcal{V}_1$ is $n \cdot \mathrm{poly}(d, \log(S))$ and it uses $O(\log(S))$ space. The communication complexity is* $\mathrm{poly}(d, \log(S))$.

  Moreover, the following four additional properties are satisfied:

  1. *The protocol is public-coin.*

2. *The verifier makes $O(d)$ queries to $\mathcal{F}$. Moreover, the points where the verifier queries $\mathcal{F}$ are determined solely by its (public) coins, and are uniformly random.*
3. *Each message sent by the prover $\mathcal{P}_1$ depends only on the preceding $O(\log(S))$ bits sent by the verifier (and on the input x and oracle $\mathcal{F}$).*[17]
4. *If, instead of the input x, $\mathcal{V}_1$ is given* oracle *access to the low degree extension of x (with respect to $\mathbb{H}, \mathbb{F}, m'$ as defined in Subsection 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, $\mathcal{V}_1$ 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 $\mathcal{F}$ and then proving Theorem 3.1. We begin in Subsection 3.1 with preliminaries, conventions, and specifications of the protocol's parameters, including the oracle function $\mathcal{F}$. The bare-bones protocol is given in Subsection 3.2. Finally, we prove Theorem 3.1 in Subsection 3.3.

3.1 Preliminaries

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

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

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

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

Jumping ahead, we note that the reason we require $|\mathbb{H}| \geq \max\{d, \log(S)\}$ (and we don't take, say $|\mathbb{H}| = 2$) is that for the sake of efficiency we need $|\mathbb{H}|$ and $|\mathbb{F}|$ to be polynomially related, and we need $|\mathbb{F}| = \text{poly}(d, \log(S))$ for error correction.

Let m be an integer such that

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

Let $m' \leq m$ be an integer such that

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

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 $C : \{0,1\}^n \to \{0,1\}$ can be converted into an arithmetic circuit $C : \mathbb{F}^n \to \mathbb{F}$ 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, we assume (without loss of generality) that $C : \mathbb{F}^n \to \mathbb{F}$ is an arithmetic circuit over the field $\mathbb{F}$. We also

[17] This fact will be used in Section 8, which uses the bare-bones protocol to construct *efficient* "short" probabilistically checkable arguments.

assume for simplicity that the circuit $C : \mathbb{F}^n \to \mathbb{F}$ is a *layered* arithmetic circuit of *fan-in 2* (over the gates $\times$ and $+$ and over the field $\mathbb{F}$) as follows.

A depth-d layered circuit is one where the gates are divided into $(d+1)$ layers. We think of the 0 layer as the output layer (comprised of the output gate), and of the d layer as the input layer (comprised of the input gates). For a layered circuit, wires can only connect gates in adjacent layers, i.e. the output wire of a gate in layer i can only be the input wire for a gate in layer $(i-1)$. 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.[18] 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 particular, we add dummy gates to the output layer, so the transformed circuit $C' : \mathbb{F}^n \to \mathbb{F}^S$ satisfies that for every $(x_1, \ldots, x_n) \in \mathbb{F}^n$,

$$C'(x_1, \ldots, x_n) = (C(x_1, \ldots, x_n), 0, \ldots, 0).$$

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

Moreover, as noted above we assume throughout that circuit is layered. We note that 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 increasing the depth of the circuit by at most a factor of $O(\log(S))$.

For each $0 \le i \le d-1$, 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 n gates in the d'th layer of C (i.e., the input layer) by $(g_{d,0}, g_{d,1}, \ldots, g_{d,n-1})$. For each $i \in [d-1]$, we associate with C two functions

$$\mathrm{add}_i, \mathrm{mult}_i : \{0, 1, \ldots, S-1\}^3 \to \{0, 1\},$$

defined by

$$\mathrm{add}_i(j_1, j_2, j_3) = \begin{cases} 1 & \text{if } g_{i-1,j_1} = g_{i,j_2} + g_{i,j_3} \\ 0 & \text{o.w.} \end{cases}, \tag{4}$$

and

$$\mathrm{mult}_i(j_1, j_2, j_3) = \begin{cases} 1 & \text{if } g_{i-1,j_1} = g_{i,j_2} \times g_{i,j_3} \\ 0 & \text{o.w.} \end{cases}. \tag{5}$$

Similarly, we associate with C two additional functions

$$\mathrm{add}_d, \mathrm{mult}_d : \{0, 1, \ldots, S-1\} \times \{0, 1, \ldots, n-1\}^2 \to \{0, 1\},$$

defined as in Equations (4) and (5), respectively.

We say that the functions $\{add_i, mult_i\}_{i \in [d]}$ *specify* the circuit C.

For each $i \in [d-1]$, let

$$\tilde{\mathrm{add}}_i, \tilde{\mathrm{mult}}_i : \mathbb{F}^{3m} \to \mathbb{F}$$

[18]Note the discrepancy between the input layer and the other layers. For our results in Sections 3 and 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 7.2 in Section 7 makes use of the fact that the input layer is small (of size n), whereas the other layers may be large (of size S).

be multivariate polynomials of degree $\leq \delta$ in each variable, that extend the functions add_i and mult_i, respectively.[19] Namely, the functions $\tilde{\text{add}}_i$ and $\tilde{\text{mult}}_i$ 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),

$$\tilde{\text{add}}_i(z_1, z_2, z_3) = \text{add}_i(\alpha(z_1), \alpha(z_2), \alpha(z_3))$$

and

$$\tilde{\text{mult}}_i(z_1, z_2, z_3) = \text{mult}_i(\alpha(z_1), \alpha(z_2), \alpha(z_3)).$$

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 $\tilde{\text{add}}_i, \tilde{\text{mult}}_i$ return 0. The fact that such functions $\tilde{\text{add}}_i$ and $\tilde{\text{mult}}_i$ exist follows from the fact that $\delta \geq |\mathbb{H}| - 1$. In particular, $\tilde{\text{add}}_i$ (resp. $\tilde{\text{mult}}_i$) could be the low degree extension of add_i (resp. mult_i),[20] though we will sometimes take them to be different extensions (of slightly higher degree).

Similarly, let

$$\tilde{\text{add}}_d, \tilde{\text{mult}}_d : \mathbb{F}^m \times \mathbb{F}^{m'} \times \mathbb{F}^{m'} \to \mathbb{F}$$

be multivariate polynomials of degree $\leq \delta$ in each variable, that extend the functions add_d and mult_d, respectively. Namely, the functions $\tilde{\text{add}}_d$ and $\tilde{\text{mult}}_d$ 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 $\alpha(z_2), \alpha(z_3) \leq n - 1$,

$$\tilde{\text{add}}_d(z_1, z_2, z_3) = \text{add}_d(\alpha(z_1), \alpha(z_2), \alpha(z_3))$$

and

$$\tilde{\text{mult}}_d(z_1, z_2, z_3) = \text{mult}_d(\alpha(z_1), \alpha(z_2), \alpha(z_3)).$$

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 $\alpha(z_2) > n - 1$, or $\alpha(z_3) > n - 1$, then both $\tilde{\text{add}}_d, \tilde{\text{mult}}_d$ return 0.

We say that the functions $\{\tilde{\text{add}}_i, \tilde{\text{mult}}_i\}_{i \in [d]}$ are *extensions of the functions that specify* the circuit C. Note that unlike the functions $\{\text{add}_i, \text{mult}_i\}_{i \in [d]}$ that specify C, the extensions $\{\tilde{\text{add}}_i, \tilde{\text{mult}}_i\}_{i \in [d]}$ 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 $\{\tilde{\text{add}}_i, \tilde{\text{mult}}_i\}_{i \in [d]}$ are *some* such extensions. We specify $\{\tilde{\text{add}}_i, \tilde{\text{mult}}_i\}$ 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 $\{\tilde{\text{add}}_i, \tilde{\text{mult}}_i\}_{i \in [d]}$:

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

where the prover and verifier can access $\tilde{\text{add}}_i$ or $\tilde{\text{mult}}_i$ by querying $\mathcal{F}$ with the proper i, a bit specifying $\tilde{\text{add}}$ or $\tilde{\text{mult}}$, 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 $v_d = (v_{d,0}, \ldots, v_{d,n-1}) \in \mathbb{F}^n$ with the d'th

[19] See Subsection 2.3 for a discussion on low degree extensions.

[20] We note that in Section 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 $\hat{f} : \mathbb{F}^m \to \mathbb{F}$ of degree $\leq |\mathbb{H}| - 1$ in each variable, such that for every $z \in \mathbb{H}^m$, $\hat{f}(z) = f(z)$.

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 $x = (x_1, \ldots, x_n) \in \mathbb{F}^n$, and are defined as follows: For each $0 \le i \le d$ 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 $v_d = (x_1, \ldots, x_n) \in \mathbb{F}^n$.

For each $0 \le i \le d-1$, let

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

be the low degree extension of v_i with respect to $\mathbb{H}, \mathbb{F}, m$ (as defined in Subsection 2.3). Claim 2.3 implies that the function $\tilde{V}_i$ is of degree $\le |\mathbb{H}| - 1$ in each of its m variables, and can be computed in time $\le \text{poly}(|\mathbb{F}|^m) = \text{poly}(S)$.

Let

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

be the low degree extension of v_d with respect to $\mathbb{H}, \mathbb{F}, m'$. Claim 2.3 implies that the function $\tilde{V}_d$ is of of degree $\le |\mathbb{H}| - 1$ in each of its m variables, and can be computed in time $\le |\mathbb{H}|^{m'} \cdot \text{poly}(|\mathbb{H}|, m') = n \cdot \text{poly}(d, \log(S))$.

3.2 The Bare-Bones Protocol

In this subsection, we present the bare-bones protocol $(\mathcal{P}_1, \mathcal{V}_1)$ for efficiently proving that $C(x) = 0$. In this protocol we give both the verifier $\mathcal{V}_1$ and the prover $\mathcal{P}_1$ oracle access to the set of functions

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

as defined in Subsection 3.1.[21,22] The prover and verifier also take as input the sting $x \in \{0,1\}^n$.

Protocol Overview. The prover wants 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 ($1 \le i \le d$) 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 the input x with respect to $\mathbb{H}, \mathbb{F}, m'$. 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 3.1: circuit size S, circuit depth d, input size n, where $n, d \le S$. We also defined there the fields $\mathbb{H}, \mathbb{F}$, integers m, m' and a degree parameter δ.

[21]We note that the functions in $\mathcal{F}$ could have been given to the prover $\mathcal{P}_1$ as input (say, via their truth-tables). We decided to give $\mathcal{P}_1$ 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 $\mathcal{P}_1$ can reconstruct the circuit C in time $O(|C|)$.

[22]In Section 4 we show how these oracles can be realized in some specific cases (for example, if C is an L-uniform circuit).

The layered arithmetic circuit $C : \mathbb{F}^n \to \mathbb{F}$ 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 $x \in \mathbb{F}^n$, and are both given oracle access to a set of functions $\mathcal{F} = \{\tilde{\text{add}}_i, \tilde{\text{mult}}_i\}_{i\in[d]}$ corresponding to C (as defined in Subsection 3.1), where each function in $\mathcal{F}$ is of degree $\leq \delta$ in each variable.

The protocol $(\mathcal{P}_1^{\mathcal{F}}(x), \mathcal{V}_1^{\mathcal{F}}(x))$ The prover needs to prove that $C(x) = 0$, or equivalently, that $\tilde{V}_0(0,\dots,0) = 0$. 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 $z_0 = (0,\dots,0)$, $r_0 = 0$). Finally, after the d'th phase, the verifier checks on his own that $\tilde{V}_d(z_d) = r_d$.

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 2.2, 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 $\leq \text{poly}(|\mathbb{H}|, m)$.

Notice that for every $p \in \mathbb{H}^m$,

$$\tilde{V}_{i-1}(p) = \sum_{\omega_1,\omega_2\in\mathbb{H}^m} \tilde{\text{add}}_i(p,\omega_1,\omega_2) \cdot \Big(\tilde{V}_i(\omega_1) + \tilde{V}_i(\omega_2)\Big) + \tilde{\text{mult}}_i(p,\omega_1,\omega_2) \cdot \tilde{V}_i(\omega_1) \cdot \tilde{V}_i(\omega_2).$$

Thus, for every $z \in \mathbb{F}^m$,

$$\tilde{V}_{i-1}(z) = \sum_{p,\omega_1,\omega_2\in\mathbb{H}^m} \tilde{\beta}(z,p) \cdot \Big(\tilde{\text{add}}_i(p,\omega_1,\omega_2) \cdot \Big(\tilde{V}_i(\omega_1) + \tilde{V}_i(\omega_2)\Big) + \tilde{\text{mult}}_i(p,\omega_1,\omega_2) \cdot \tilde{V}_i(\omega_1) \cdot \tilde{V}_i(\omega_2)\Big).$$

For every $z \in \mathbb{F}^m$, let $f_z : (\mathbb{F}^m)^3 \to \mathbb{F}$ be the function defined by

$$f_z(p,\omega_1,\omega_2) \stackrel{\text{def}}{=} \tilde{\beta}(z,p) \cdot \Big(\tilde{\text{add}}_i(p,\omega_1,\omega_2) \cdot \Big(\tilde{V}_i(\omega_1) + \tilde{V}_i(\omega_2)\Big) + \tilde{\text{mult}}_i(p,\omega_1,\omega_2) \cdot \tilde{V}_i(\omega_1) \cdot \tilde{V}_i(\omega_2)\Big).$$

Proposition 2.2, together with the definitions of $\tilde{\text{add}}_i$, $\tilde{\text{mult}}_i$ 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 2.[23]

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 $p, \omega_1, \omega_2 \in_R \mathbb{F}^m$ (chosen by the verifier). Recall that the verifier has oracle access to the functions $\tilde{\text{add}}_i$ and $\tilde{\text{mult}}_i$. Moreover, according to Proposition 2.2, computing the function $\tilde{\beta}$ requires time $\leq \text{poly}(|\mathbb{H}|, m)$. So, the main 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 $\mathcal{V}_1$ (who knows ω_1 and ω_2) receives two values v_1, v_2 and 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 $\mathcal{V}_1$. 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 γ uniquely, and that γ can be computed (by both $\mathcal{P}_1$ and $\mathcal{V}_1$) 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 $\mathcal{V}_1$.

3. Upon receiving a function $f : \mathbb{F} \to \mathbb{F}$ from the prover (supposedly, $f = \tilde{V}_i \circ \gamma$), the verifier $\mathcal{V}_1$ 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 $\mathcal{V}_1$ 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 $z_i \overset{\text{def}}{=} \gamma(t)$ and $r_i \overset{\text{def}}{=} f(t)$.

[23]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 $\text{poly}(S)$.

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 $\tilde{V}_d(z_d) = r_i$, where now $\omega_1, \omega_2, z_d \in \mathbb{F}^{m'}$.

The final verification. After the final verification phase, the verifier $\mathcal{V}_1$ 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 $\mathbb{F}, \mathbb{H}, m'$). 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 Proof of Theorem 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 2.5).

Soundness. For the soundness condition, fix any layered arithmetic circuit $C : \mathbb{F}^n \to \mathbb{F}$, any $x \in \mathbb{F}^n$ such that $C(x) \neq 0$, and any set of functions $\mathcal{F}$ that are low-degree extensions of the functions that specify the circuit C (as defined in Subsection 3.1). Assume that there exists a cheating prover $\mathcal{P}^*$ such that

$$\Pr\left[(\mathcal{P}^{*\mathcal{F}}(x), \mathcal{V}_1^{\mathcal{F}}(x)) = 1\right] = s.$$

Recall that the protocol $(\mathcal{P}_1^{\mathcal{F}}(x), \mathcal{V}_1^{\mathcal{F}}(x))$ 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 $(\mathcal{P}^{*\mathcal{F}}(x), \mathcal{V}_1^{\mathcal{F}}(x)) = 1$. For every $0 \le i \le d$, 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 $\neg(T_0)$. Notice that

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

For every $i \in [d]$, let E_i denote the event that indeed $\tilde{V}_i(w_1) = v_1$ and $\tilde{V}_i(w_2) = v_2$.[24] 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

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

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

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

[24] 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.

Thus,

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

All in all, we get that

$$s \leq \frac{7md\delta}{|\mathbb{F}|}.$$

Taking $\mathbb{F}$ such that $|\mathbb{F}| \geq 700md\delta = \text{poly}(|\mathbb{H}|)$, 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 ($1 \leq i \leq d$) the prover reduces the task of proving that $\tilde{V}_{i-1}(z_{i-1}) = r_{i-1}$ to the task of proving that $\tilde{V}_i(z_i) = r_i$. This is done by running a sum-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 $\mathcal{P}_1$ is $\text{poly}(|\mathbb{F}^m|) = \text{poly}(S)$, both in the sum-check protocol (see Lemma 2.5) and in the proceeding interactive process.

2. The running time of the verifier $\mathcal{V}_1$ (with oracle access to $\mathcal{F}$) is $\text{poly}(|\mathbb{F}|, m) = \text{poly}(d, \log(S))$, both in the sum-check protocol (see Lemma 2.5) and in the proceeding interactive process.

   The space used by $\mathcal{V}_1$ is $O(\log(|\mathbb{F}|) \cdot m) = O(\log(S))$, both in the sum-check protocol (see Lemma 2.5) 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_i, 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 $O(\log(|\mathbb{F}|) \cdot m) = O(\log(S))$, not much larger than the space used in a single phase.

3. The sum-check protocol has communication complexity $\text{poly}(|\mathbb{F}|, m)$ (see Lemma 2.5), and the proceeding interactive process has communication complexity $\text{poly}(|\mathbb{F}|)$. Thus, in total, each phase has communication complexity $\text{poly}(|\mathbb{F}|, m) = \text{poly}(d, \log(S))$. Moreover, the verifier $\mathcal{V}_1$ is public-coin, and the number of random bits it sends to the prover $\mathcal{P}_1$ in each phase is $O(\log(|\mathbb{F}|) \cdot m)$. This, together with the fact that the only information that the prover needs to "remember" for the next phase is the values i, z_i, 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 $O(\log(|\mathbb{F}|) \cdot m) = O(\log(S))$ random bits sent by the verifier.

4. In each phase, the verifier queries each $\tilde{\text{add}}_i$ and $\tilde{\text{mult}}_i$ only at a single location. The verifier's queries to $\tilde{\text{add}}_i$ and $\tilde{\text{mult}}_i$ 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 $\mathcal{V}_1$ 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 $\mathbb{F}, \mathbb{H}, m'$). This can be done (by Claim 2.3) in time $n \cdot \text{poly}(|\mathbb{H}|, m') = n \cdot \text{poly}(d, \log(S))$ 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. ■

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 $C : \{0,1\}^n \to \{0,1\}$ of size S and of depth d, and a string $x \in \{0,1\}^n$, 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 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 4.1 we show how to implement the bare-bones protocol for 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 $\{\tilde{\text{add}}_i, \tilde{\text{mult}}_i\}_{i\in[d]}$ can be evaluated by the verifier in polylog-time and log-space (without using any non-standard oracles).

Second, in Subsection 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 $\{\tilde{\text{add}}_i, \tilde{\text{mult}}_i\}_{i\in[d]}$. In these sub-protocols 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 1.4).

Finally, in Subsection 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 to 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 1.5.

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

3.1). Let $\mathbb{H}$ be an extension field of $\mathbb{GF}[2]$, $\mathbb{F}$ an extension field of $\mathbb{H}$ (and thus also of $\mathbb{GF}[2]$). We always think of the circuit C (which is defined over the field $\mathbb{GF}[2]$), as a *layered* arithmetic circuit with fan-in 2, over the extension field $\mathbb{F}$. 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 $add_i, mult_i$ as in Section 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 3.1, add_d and $mult_d$ 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 $add_i, mult_i$, as defined above, and the *boolean circuits* we construct to compute them, which we also call $add_i, mult_i$.

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

As before, let $\tilde{add}_i, \tilde{mult}_i : (\mathbb{F}^m)^3 \rightarrow \mathbb{F}$ be *some* extensions of $add_i, mult_i$ (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 α, α'), and answer as $add_i, mult_i$. 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 $\mathbb{H}^m$, $\tilde{add}_i, \tilde{mult}_i$ output some value in $\mathbb{F}$. An important property we want from these functions is that they have low degree δ, in particular δ will be significantly smaller than $|\mathbb{F}|$. Throughout this section we abuse notation and do not distinguish between *functions* $\tilde{add}_i, \tilde{mult}_i$, as described above, and the *arithmetic circuits* we construct to compute them, which we also call $\tilde{add}_i, \tilde{mult}_i$.

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 4.4. We proceed by first showing in Subsection 4.1.1 that languages in $\mathcal{NL}$ are computable by circuits for which $\tilde{add}_i, \tilde{mult}_i$ can be evaluated in poly-logarithmic time and logarithmic space. In Section 4.1.2 we implement the bare-bones protocol by having the verifier replace its oracle $\mathcal{F}$ with these easy to evaluate $\tilde{add}_i$ and $\tilde{mult}_i$.

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

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 $\{\tilde{add}_i, \tilde{mult}_i\}$ are polylog-size arithmetic circuits that are $\log\log$-space uniform.[25] 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 δ, and is significantly smaller than $|\mathbb{F}|$. We do this in three steps:

[25]Throughout 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 $2^{O(s(n))}$.

First, we show that every language in $\mathcal{NL}$ has (for every input length) a poly-size and polylog-depth arithmetic circuit, for which $add_i, mult_i$ are polylog-size, $\log\log$-space uniform, constant-depth ($\mathcal{AC}^0$) *boolean* circuits. This result is stated in Lemma 4.1. **Second**, we show how to compute the low-degree extensions of α, α' using a polylog-size, $\log\log$-space uniform and low degree (degree $|\mathbb{H}|-1$) arithmetic circuit.[26] This result is stated in Claim 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 $\tilde{add}_i, \tilde{mult}_i$ are polylog-size, $\log\log$-space uniform, low degree (δ) arithmetic circuits. This is the result we need for implementing the Bare-Bones protocol for $\mathcal{NL}$ computations, and it is stated in Lemma 4.3.

Step 1: Small, Uniform, Constant Depth *Boolean* $add_i, mult_i$. We begin by showing that every language in $\mathcal{NL}$ has (for each input length) a poly-size, polylog-depth circuit, for which $add_i, mult_i$ 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 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 $\mathrm{poly}(2^{s(n)})$ 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_i and $mult_i$ are $\mathrm{poly}(s(n))$-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_i if $b = 0$, and it outputs the circuit $mult_i$ if $b = 1$.

Proof of Lemma 4.1.

Preliminaries. We begin with notation and preliminaries, reviewing how to translate T's computations into questions about the adjacency matrix of its 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 ($O(1)$ 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

[26]Recall that $\alpha : \mathbb{H}^m \to \{0, 1, \ldots, |\mathbb{H}|^m - 1\}$ and $\alpha' : \mathbb{H}^{m'} \to \{0, 1, \ldots, |\mathbb{H}|^{m'} - 1\}$ are the lexicographic order functions.

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 ℓ from u to w and a path of length at most ℓ from w to v. Using arithmetics over $\mathbb{GF}[2]$ we get:

$$B_{p-1}[u,v] = 1 + \prod_{w \in \{0,1\}^{g(n)}} \big(1 + B_p[u,w] \cdot B_p[w,v]\big). \tag{6}$$

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 $B_{\log(t(n))} = B_x, B_{\log(t(n)-1)}, \ldots, B_1, B_0$. 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 $\text{poly}(2^{s(n)})$. 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 (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 $2^{3 \cdot g(n)} + 2$ gates. These include at most $2^{3 \cdot g(n)}$ "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 $2^{3 \cdot g(n)}, 2^{3 \cdot g(n)} + 1$ (respectively). By convention, if an intermediate layer includes less than $2^{3 \cdot g(n)}$ "standard" gates, then we sometimes use shorter labels, implicitly fixing their least significant bits to be 0 and ignoring them in the exposition below.[27]

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 $z \in \{0,1,\ldots,\text{poly}(2^{s(n)})\}$ 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 $add_i, mult_i$ are both $\log(g(n))$-space uniform, $\text{poly}(g(n))$-size, constant depth ($\mathcal{AC}^0$) circuits. The machine G will only be described after the detailed description of the layers.

[27]in fact the circuits add_i and $mult_i$ constructed below must verify that these bits are all 0, this is easily accomplished.

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 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 $2^{3 \cdot g(n)} + b$ (computing the constant $b \in \{0, 1\}$) is a multiplication gate, multiplying the gates $2^{3 \cdot g(n)} + b$ (computing b) and $2^{3 \cdot g(n)} + 1$ (computing 1) in layer $i + 1$. For an addition layer, its constant gate $2^{3 \cdot g(n)} + b$ is an addition gate, adding the gates $2^{3 \cdot g(n)} + b$ (computing b) and $2^{3 \cdot g(n)}$ (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 $2^{3 \cdot g(n)} + b$ is an addition gate of the input layer gates: $n + b$ (computing b), and n (computing 0).

The output of $add_i, mult_i$ 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 $z_1 = 2^{3 \cdot g(n)} + 1$ (constant value 1), add_i accepts iff $z_2 = 2^{3 \cdot g(n)} + 1$ (constant value 1) and $z_2 = 2^{3 \cdot g(n)}$ (constant value 0). These computations can be done by 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 add_i and $mult_i$, we always implicitly mean that they first check (in $\mathcal{AC}^0$) whether the gate label in layer $(i - 1)$ 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 $2^{2 \cdot g(n)}$ 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_i: $B_x[u, v] = 1$. 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 $x_i = 1$: $B_x[u, v] = x_i$.
4. Configuration u can go to v iff $x_i = 0$: $B_x[u, v] = 1 + x_i$.

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_i and $mult_i$ 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 $z_1 \in \{0, 1\}^{2 \cdot g(n)}$ 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 $\mathcal{AC}^0$ boolean circuit of size $\mathrm{poly}(g(n))$.

   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_1} 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_1} could make the internal state q_1 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 $\mathcal{AC}^0$ circuit of $\mathrm{poly}(g(n))$-size.

3. After running this computation, $add_{(\log(t(n)),1)}$ "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, $add_{(\log(t(n)),1)}$ 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, $add_{(\log(t(n)),1)}$ 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 $x_{i_1} = 1$ causes the transition. In this case, $add_{(\log(t(n)),1)}$ 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 $x_{i_1} = 0$ causes the transition. In this case, $add_{(\log(t(n)),1)}$ 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 $add_{(\log(t(n)),1)}$ as above is a $O(\log g(n))$-space uniform $\mathcal{AC}^0$ boolean circuit of size $\mathrm{poly}(g(n))$.

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 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 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 $\prod_{w \in \{0,1\}^{g(n)}} \big(1 + B_{p+1}[u, w] \cdot B_{p+1}[w, v]\big)$. Finally, the top layer of the sub-circuit adds 1 to each such product. It computes for each pair (u, v) the value $1 + \prod_{w \in \{0,1\}^{g(n)}} \big(1 + B_{p+1}[u, w] \cdot B_{p+1}[w, v]\big)$, which is the (u, v)-th entry of B_p (by Equation 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 add_i is always 0, $mult_i$ 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 $mult_i$ is always 0, add_i accepts $z_1 = (u, v, w)$ iff $z_2 = (u, v, w)$ and $z_3 = 2^{3 \cdot g(n)} + 1$ (the constant 1 gate).

For $k \in \{g(n), \ldots, 1\}$, the (p, k)-th layer has $2^{2 \cdot g(n)+k-1}$ 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 $\prod_{w \in \{0,1\}^{g(n)}} \left(1 + B_{p+1}[u, w] \cdot B_{p+1}[w, v]\right)$, just as required. For these layers, $add_{(p,k)}$ always outputs 0, as there are only multiplication gates. $mult_{(p,k)}$ 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 $2^{2 \cdot g(n)}$ 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) $1 + \prod_{w \in \{0,1\}^{g(n)}} \left(1 + B_{p+1}[u, w] \cdot B_{p+1}[w, v]\right)$. Here $mult_i$ always outputs 0, add_i accepts $z_1 = (u, v)$ iff $z_2 = (u, v)$ and $z_3 = 2^{3 \cdot g(n)} + 1$ (the constant 1 gate).

We conclude that each layer in each of the intermediate sub-circuits has, as required, circuits $add_i, mult_i$ that are $O(\log g(n))$-space uniform $\mathcal{AC}^0$ boolean circuit of size 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 add_i if $b = 0$, and the circuit $mult_i$ if $b = 1$. The existence of such a machine follows easily from the above constructions of add_i and $mult_i$ for each layer of the circuit C. Note that since all the intermediate sub-circuits are identical, there exists a single $\log(g(n))$-space machine that on input (n, i, b) generates add_i or $mult_i$ for each layer of these sub-circuits. The input sub-circuit (which is different from the intermediate ones) has only two layers, for which add_i and $mult_i$ 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 add_i or $mult_i$ as required. ■

Step 2: Small Low-Degree Circuits for α, α'. Recall that our goal in this section is proving that every language in $\mathcal{NL}$ has circuits for which $\tilde{add}_i$ and $\tilde{mult}_i$ are themselves small low-degree uniform arithmetic circuits. Lemma 4.1 was the first step towards this goal. We now turn our attention to the mappings α, α' that map arithmetic vectors to boolean (or numerical) labels of circuit gates. If we want to create small arithmetic circuits computing $\tilde{add}_i$, $\tilde{mult}_i$, these circuits should themselves be able to compute the mappings α and α'. We show that α, α' can be computed by $O(\log(|\mathbb{F}|) + \log(m))$-space uniform, poly$(|\mathbb{F}|, m)$-size arithmetic circuits (over $\mathbb{F}$) of degree $|\mathbb{H}| - 1$.

Claim 4.2. Fix $\mathbb{H}$, an extension field of $\mathbb{GF}[2]$, $\mathbb{F}$ an extension field of $\mathbb{H}$, and m an integer value. Let $\alpha : \mathbb{H}^m \to \mathbb{GF}[2]^{\log(|\mathbb{H}^m|)}$ be the function that maps a vector in $\mathbb{H}^m$ to its lexicographic order, represented as a sequence of $\log(|\mathbb{H}^m|)$ 0's and 1's (we can think of these as elements of $\mathbb{GF}[2]$ or of its extension field $\mathbb{F}$).

There exists an arithmetic circuit $T_{\mathbb{H},\mathbb{F},m} : \mathbb{F}^m \to \mathbb{F}^{\log(|\mathbb{H}|^m)}$ that computes the low degree extension (with respect to $\mathbb{H}, \mathbb{F}, m$) of α.[28] The circuit $T_{\mathbb{H},\mathbb{F},m}$ is of size poly$(|\mathbb{H}|) \cdot m$ and degree $|\mathbb{H}| - 1$ in each of its inputs. It can be generated (from $(|\mathbb{H}|, |\mathbb{F}|, m)$) or evaluated (on an input in $\mathbb{F}^m$) by a $O(\log(|\mathbb{F}|) + \log(m))$-space uniform Turing machine.

[28]Recall that the notion of a low degree extension also applies to functions with multiple outputs, as described in Section 2.3.

Proof of Claim 4.2. Let $\alpha_1 : \mathbb{H} \to \mathbb{F}^{\log(|\mathbb{H}|)}$ be the function that takes a *single* element of $\mathbb{H}$ and maps it to its lexicographic order, represented as a sequence of $\log(|\mathbb{H}|)$ 0's and 1's (elements of $\mathbb{GF}[2]$ and thus also of $\mathbb{F}$). Let $\tilde{\alpha_1}$ be the unique low-degree extension of α_1.[29] The circuit $T_{\mathbb{H},\mathbb{F},m}$ applies $\tilde{\alpha_1}$ to each of its m inputs (elements of $\mathbb{F}$), and outputs the concatenation of the m outputs of $\tilde{\alpha_1}$. The reason that $T_{\mathbb{H},\mathbb{F},m}$ indeed computes the low degree extension $\tilde{\alpha}$ of α, 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 2.2, 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 a $O(\log(|\mathbb{F}|) + \log(m))$-space uniform Turing machine. ■

Step 3: Small, Uniform, Low Degree *Arithmetic* $\tilde{add}_i, \tilde{mult}_i$. We are now ready to prove the main lemma of this subsection, showing that every language in $\mathcal{NL}$ has (for each input length) a poly-size and polylog-depth circuit, for which $\tilde{add}_i$ and $\tilde{mult}_i$ 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 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 $i \in \{1, \ldots, d(n)\}$, $\tilde{add}_i$ and $\tilde{mult}_i$ 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 $\tilde{add}_i$. If $b = 1$, then $\tilde{G}$ outputs the circuit $\tilde{mult}_i$.

Proof of Lemma 4.3. The circuit C is exactly the same circuit constructed in the proof of Lemma 4.1. The (arithmetic) circuits $\tilde{add}_i$ and $\tilde{mult}_i$ take as input 3 "gate labels" in $\mathbb{F}^m$. They apply the circuit $T_{\mathbb{H},\mathbb{F},m}$ (from Claim 4.2), which computes the low degree extension of α, 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 $\tilde{add}_i$ and $\tilde{mult}_i$ take the result of this translation, and use it as input for an arithmetization of the boolean circuit add_i or $mult_i$ (respectively) from Lemma 4.1. If the original gate labels were all in $\mathbb{H}^m$, then the output should be equal to add_i's or $mult_i$'s output on their boolean translations. A more detailed description follows.

[29] Again, in the past we usually worked with low degree extensions of functions that map multiple $\mathbb{H}$-elements to a single $\mathbb{F}$ element, whereas α_1 maps a single $\mathbb{H}$ element to many $\mathbb{F}$ 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.

Let $\{add_i, mult_i\}$ be the *boolean* circuit families constructed in Lemma 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 $\{add_i, mult_i\}$ 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 $\tilde{add}_i$ and $\tilde{mult}_i$, on the other hand, take as input gate labels represented as arithmetic vectors in $\mathbb{F}^m$.[30] 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 4.2. This adds a (multiplicative) $|\mathbb{H}|$-factor to the degree, and a $\text{poly}(|\mathbb{F}|, m)$ 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 (the "arithmetized" add_i or $mult_i$) correctly outputs whether or not the first gate is an addition or multiplication of the other two.

We obtain, as required, $\tilde{add}_i$ and $\tilde{mult}_i$ 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. ■

4.1.2 Realizing the Bare-Bones Protocol

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

Recall that to implement the bare-bones protocol one must have a way for the verifier to implement $\tilde{add}_i$ and $\tilde{mult}_i$ oracles, whose degree is not too large. This is exactly what Lemma 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 $\mathcal{NL}$ computations. We state this result more generally, for given non-deterministic time and space bounds.

Theorem 4.4. *Let L be a language that can be computed by a non-deterministic Turing Machine using space $s(n)$ and time $t(n)$ (we assume $s(n) = \Omega(\log(n))$). L has an interactive proof (an implementation of the bare-bones protocol) where:*

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

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

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

[30]We ignore here the fact that $add_{d(n)}$ and $mult_{d(n)}$ work over m' inputs instead of m, this can be handled in a similar manner.

Proof of Theorem 4.4. Fix an input length n. By Lemma 4.3, the language L can be computed by a circuit C of size $\mathrm{poly}(2^{s(n)})$ and depth $d(n) = O(s^2(n))$. Fix $\mathbb{H}, \mathbb{F}, m, m'$ as in the bare-bones protocol (see Subsection 3.1). 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 $\mathrm{poly}(s(n))$, $m = O(s(n)/\log(s(n)))$, and $m' = O(\log(n)/\log(s(n)))$.

The circuits $\tilde{add}_i$ and $\tilde{mult}_i$, constructed in Lemma 4.3, are of degree $\delta = \mathrm{poly}(s(n))$, and can be generated and evaluated (over $\mathbb{F}$) in time $\mathrm{poly}(s(n))$ 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 $\{\tilde{add}_i, \tilde{mult}_i\}$ with explicit computations of $\tilde{add}_i$ and $\tilde{mult}_i$ constructed above. From Theorem 3.1 we get that the protocol has perfect completeness, and soundness $\frac{1}{100}$. The prover's work is $\mathrm{poly}(2^{s(n)})$. The verifier's work is only $n \cdot \mathrm{poly}(s(n))$, and his space usage is $O(s(n))$. The protocol is public-coin, and the communication complexity is $\mathrm{poly}(s(n))$. ■

Plugging in the parameters for languages in $\mathcal{NL}$, i.e. space $O(\log(n))$ and time $\mathrm{poly}(n)$, we get that the prover is efficient, the verifier runs in quasi-linear time, and the communication complexity is $\mathrm{polylog}(n)$. This is stated formally below (the proof is immediate from Theorem 4.4):

Corollary 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 $\mathrm{poly}(n)$. L has an interactive proof (an implementation of the bare-bones protocol) where:*

1. *The prover runs in time $\mathrm{poly}(n)$, the verifier runs in time $n \cdot \mathrm{polylog}(n)$ and space $O(\log(n))$.*
2. *The protocol has perfect completeness and soundness $1/100$.*
3. *The protocol is public-coin, with communication complexity $\mathrm{polylog}(n)$.*

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 4.4 from the previous subsection for delegating $\mathcal{NL}$ computations. We proceed in two steps. First we show that for log-space uniform circuits, the functions $\tilde{add}_i$ and $\tilde{mult}_i$, which are the unique low degree extensions of add_i and $mult_i$, can themselves be computed in log-space (with respect to appropriately chosen fields). Then, applying Theorem 4.4 (or rather Corollary 4.5), we immediately conclude that a verifier can delegate the computation of $\tilde{add}_i$ and $\tilde{mult}_i$ 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 $\tilde{add}_i$ and $\tilde{mult}_i$ 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 $\tilde{add}_i$ or $\tilde{mult}_i$, the prover supplies it with the value, and proves that this value is correct by running, as a sub-protocol, the protocol of Theorem 4.4.

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

Claim 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 $\tilde{add}_i, \tilde{mult}_i$ for the circuit C_n (with respect to $\mathbb{H}, \mathbb{F}, m, m'$). Here we take $\tilde{add}_i, \tilde{mult}_i$ to be the *unique* low degree extensions of $add_i, mult_i$, 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 $\tilde{add}_i$ or $\tilde{mult}_i$.

Proof of Claim 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 add_i and $mult_i$ can be evaluated using $O(s(n))$ space. By Claim 4.2, the function α (or α') that converts a vector in $\mathbb{H}^m$ (resp. $\mathbb{H}^{m'}$) into its (boolean) lexicographic order can be computed in space $O(\log(|\mathbb{F}|) + \log(m)) = O(\log(s(n)))$ (the same holds for α').

Consider the two functions that take inputs in $\mathbb{H}^m$, convert them into boolean gate labels using α, and then run add_i or $mult_i$ on the result. By the above, these functions can be evaluated on inputs in $\mathbb{H}^m$ in space $O(s(n))$. Now, note that $\tilde{add}_i$ and $\tilde{mult}_i$ are the unique low-degree extension of these functions. Thus, by Claim 2.3, 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 $\tilde{add}_i, \tilde{mult}_i$ is $O(s(n))$. Clearly, the above holds also for $\tilde{add}_d, \tilde{mult}_d$, 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 of Section 1.1.

Theorem 4.7 (Theorem 1.1 of Section 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 $n \cdot \text{poly}(\log(d(n), S(n)))$ 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 $\text{poly}(\log(d(n), S(n)))$.*
2. *The protocol has perfect completeness and soundness $1/2$.*
3. *The protocol is public-coin, with communication complexity $d(n) \cdot \text{polylog}(S(n))$.*

Proof of Theorem 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.

Fix $\mathbb{H}, \mathbb{F}, m, m'$ as in the bare-bones protocol (see Subsection 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, taking $\tilde{add}_i$ and $\tilde{mult}_i$ to be the *unique* low-degree extensions of add_i and $mult_i$ respectively (of degree $|\mathbb{H}| - 1$ in each variable). The verifier in the bare-bones protocol queries $\tilde{add}_i$ and $\tilde{mult}_i$ at most $2d(n)$ times: In each phase of the protocol (or layer of the circuit) he queries $\tilde{add}_i$ once and queries $\tilde{mult}_i$ once. Now, when implementing the bare-bones protocol, the prover will send the verifier the values of $\tilde{add}_i$ and $\tilde{mult}_i$ 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 $\tilde{add}_i$ and $\tilde{mult}_i$ computations.

By Claim 4.6, since C is a $\log(S(n))$-space uniform circuit, $\tilde{add}_i$ and $\tilde{mult}_i$ can be computed in $O(\log(S(n)))$ space. In turn, by Theorem 4.4, there exists an interactive proof for verifying the correctness of each bit of $\tilde{add}_i$'s and $\tilde{mult}_i$'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 4.4, the total prover running time is $\mathrm{poly}(S(n))$, the verifier running time is $d(n) \cdot \mathrm{polylog}(S(n))$ (recall that the input to $\tilde{add}_i$ and $\tilde{mult}_i$ 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 $\mathcal{F}$ for computing $\{\tilde{add}_i, \tilde{mult}_i\}$ with an interactive sub-protocol where the prover sends the verifier the value of $\tilde{add}_i$ or $\tilde{mult}_i$, and then proves its correctness. From Theorem 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 $\mathrm{poly}(S(n))$. The verifier's work is only $\mathrm{polylog}(S(n)) \cdot (n + d(n))$, and its space usage is $O(\log(S(n)))$. The protocol is public-coin, and its communication complexity is $d(n) \cdot \mathrm{polylog}(S(n))$. ■

As an immediate consequence of Theorem 1.1 we obtain two main corollaries. The first, stated as Corollary 1.2 in Section 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 1.4 in Section 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 also yields a new corollary for interactive proofs for languages computed by uniform Turing Machines (rather than uniform circuits). Using the fact that a langauge 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 $\mathrm{poly}(t(n) \cdot 2^{s(n)})$ and depth $s^2(n)$ we conclude that:

Corollary 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* $\mathrm{poly}(t(n) \cdot 2^{s(n)})$*. The verifier runs in time* $n \cdot \mathrm{poly}(s(n))$ *and space* $\mathrm{poly}(s(n))$*.*

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

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

It is instructive to compare Corollary 4.8 in terms of the honest prover's running time with the well-known $\mathcal{IP} = \mathcal{PSPACE}$ theorem of [LFKN92, Sha92].

Theorem 4.9 ($\mathcal{IP} = \mathcal{PSPACE}$ [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* $2^{\mathrm{poly}(s(n),\log(t(n)))}$. *The verifier runs in time* $n \cdot \mathrm{poly}(s(n))$ *and space* $\mathrm{poly}(s(n))$.
2. *The protocol has perfect completeness and soundness* $1/2$.
3. *The protocol is public-coin, with communication complexity* $\mathrm{poly}(s(n))$.

Thus, using Theorem 1.1 (via Corollary 4.8), we actually obtain a significant reduction in the running time of the honest prover: from $2^{\mathrm{poly}(s(n)\cdot\log(t(n)))}$ to $\mathrm{poly}(t(n)\cdot 2^{s(n)})$. 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 $\mathrm{poly}(t(n))$ and space $\mathrm{poly}(s(n))$ (while maintaining the verifier running time and communication complexity of known protocol).

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) pre-processing 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 $\mathcal{C}$ 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 ($\mathrm{poly}(|C|)$-time) randomized computation $data \leftarrow \mathcal{V}_{pre}(C)$, resulting in an output $data$, which will be retained in the proceeding on-line interactive phase. The output $data$ of the verifier's pre-processing computation should be significantly smaller than $|C|$. In our construction, $data$ will be of size $\mathrm{poly}(d, \log(S))$ (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 $\mathcal{V}$ takes as input x and $data$ (but *not* the circuit C). The prover $\mathcal{P}$ 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 $data$. Moreover, $data$ 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 $data$). We make the usual completeness and (information-theoretic) soundness requirements. We require that the verifier's running time *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 $data$ 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 $\mathcal{F} = \{\tilde{add}_i, \tilde{mult}_i\}$ 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 *data* string. A formal Theorem follows:

Theorem 4.10 (Theorem 1.5 of Section 1.4, restated)**.** *Let L be a language computable by a (non-uniform) circuit family $\mathcal{C}$ of size $S(n)$ and depth $d(n)$. There exists an on-line/off-line interactive proof $(\mathcal{P}(C,x), \mathcal{V}(x, data), \mathcal{V}_{pre}(C))$ 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 $\mathcal{V}_{pre}(C)$ takes time $\mathrm{poly}(S(n))$. The output data is of length $|data| = \mathrm{poly}(d(n), \log(S(n)))$.*
2. *The prover $\mathcal{P}(C,x)$ runs in time $\mathrm{poly}(S(n))$.*
3. *The on-line verifier $\mathcal{V}(x, data)$ runs in time $n \cdot \mathrm{poly}(d(n), \log(S(n)))$ and space $O(\log(S(n)))$.*
4. *The communication complexity of the (on-line) interactive protocol is $\mathrm{poly}(d(n), \log(S(n)))$.*

Proof of Theorem 1.5. As noted above, the protocol is another implementation of the bare-bones protocol of Theorem 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 $\{\tilde{add}_i, \tilde{mult}_i\}_{i=1}^{d}$. Here we would like to implement these functions.

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

Thus, the pre-processing algorithm $\mathcal{V}_{pre}(C)$, chooses the $\mathrm{poly}(d \cdot \log(S))$ public coins. These immediately specify the verifier's $O(d)$ queries to the functions $\{\tilde{add}_i, \tilde{mult}_i\}$. Then, $\mathcal{V}_{pre}(C)$ computes the answers to all these queries. To do this, it computes the truth table (of size $\mathrm{poly}(S)$) of the boolean functions $add_i, mult_i$, and then computes their low-degree extensions (note that this can be done without knowing the input x). By Claim 2.3, computing the low-degree extension takes time $\mathrm{poly}(S)$. Finally, $\mathcal{V}_{pre}(C)$ outputs the string *data* consisting of the $\mathrm{poly}(d \cdot \log(S))$ random coins it chose, as well as the $O(d)$ oracle function values (each of size $\mathrm{poly}(d, \log(S))$). The total length of the *data* string is thus $\mathrm{poly}(d, \log(S))$. In the on-line interactive phase, the verifier $\mathcal{V}(x, data)$ simulates the verifier of the bare-bones protocol, while using the random coins and the oracle answers, as specified in *data*.

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

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 an $\mathcal{NC}$ circuit. We consider both the ($\mathcal{L}$)-uniform setting, and the non-uniform setting.

In the non-uniform setting, we use the bare-bones protocol (described in Subsection 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 κ 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 $L = \{x : \exists w \text{ s.t. } \mathcal{R}_L(x, w) = 1\}$ be an $\mathcal{NP}$ language. We think of the relation $\mathcal{R}_L$ as a Boolean circuit (rather than a function). We denote by d the depth of $\mathcal{R}_L$, by $n = |x|$ the instance size, by $k = |w|$ the witness size, and by κ the security parameter. The reader should think of k, d, κ as functions of n.

We start by recalling formally our two theorems, starting with the theorem for the non-uniform setting.

Theorem 5.1 (Theorem 1.6 of Section 1.5, restated)**.** *Assume one-way functions exist, and let $\kappa = \kappa(n) \geq \log(n)$ be a security parameter. Let L be a language in $\mathcal{NP}/\text{poly}$, 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 $k \cdot poly(\kappa, d(n))$.*

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

Theorem 5.2 (Theorem 1.7 of Section 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 $k \cdot poly(\kappa, d(n))$.*

Proof idea of Theorems 1.6 and 1.7. The idea is to use the bare-bones protocol of Theorem 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 1.5) by having the verifier compute the oracle answers by himself (an efficient computation). In the uniform case (as in Theorem 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 1.6: Fix a language L in $\mathcal{NP}$/poly, 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(κ) [Nao89, HILL99] (see Goldreich's book [Gol01] for the definition of a commitment scheme and for the proof method).

Our zero-knowledge interactive proof $(P_{\text{ZK}}(x,w), V_{\text{ZK}}(x))$ for L, makes use of the bare-bones protocol (presented in Section 3) to prove that $\mathcal{R}_L(x,w) = 1$, 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 $\tilde{G} : \mathbb{F}^{m'} \to \mathbb{F}$. It was shown in Theorem 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

$$(\mathcal{P}_1^{\mathcal{F}}(x,w), \mathcal{V}_1^{\mathcal{F},\tilde{G}}).$$

According to the notation in Section 3,

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

where $\tilde{add}_i, \tilde{mult}_i$ are some (low degree) extensions of $add_i, mult_i$, respectively. We take $\tilde{add}_i$ and $\tilde{mult}_i$ to be the unique low-degree extensions of add_i and $mult_i$ respectively, so as to ensure that they are uniquely defined (and can be computed in polynomial time). See Section 3 for the details.

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

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

2. The prover $P_{\mathrm{ZK}}(x, w)$ and verifier $V_{\mathrm{ZK}}(x)$ run the bare-bones protocol $(\mathcal{P}_1, \mathcal{V}_1)$ (see Theorem 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 $(P_{\mathrm{ZK}}(x, w), V_{\mathrm{ZK}}(x))$, this transcript will be converted to a transcript of the form:

   $$(r_1, \mathit{com}(m_1), r_2, \mathit{com}(m_2), \ldots, r_\ell, \mathit{com}(m_\ell)).$$

   The prover $P_{\mathrm{ZK}}(x, w)$ 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 $\mathrm{poly}(n)$. The verifier $V_{\mathrm{ZK}}(x)$ emulates the verifier $\mathcal{V}_1$. Recall that the bare-bones protocol is public-coin, and so the verifier $V_{\mathrm{ZK}}(x)$ 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 3.1, the verifier $\mathcal{V}_1^{\mathcal{F}, \tilde{G}}$ 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 $\mathrm{poly}(n)$ given the verifier's randomness.

   Both the prover $P_{\mathrm{ZK}}(x, w)$ and the verifier $V_{\mathrm{ZK}}(x)$ 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 3.1, the verifier $\mathcal{V}_1^{\mathcal{F}, \tilde{G}}$ 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 $P_{\mathrm{ZK}}(x, w)$ and the verifier $V_{\mathrm{ZK}}(x)$ compute this oracle query, denoted by $z \in \mathbb{F}^m$. This can be done in time $\mathrm{poly}(n)$.

5. According to Proposition 2.2,

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

   Moreover, $\tilde{\beta}$ can be evaluated in time $\mathrm{poly}(d, \log n)$ (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 $P_{\mathrm{ZK}}(x, w)$ and the verifier $V_{\mathrm{ZK}}(x)$ 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 $\mathrm{poly}(n)$.

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:

   $$(\mathit{com}(w), r_1, \mathit{com}(m_1), \ldots, r_\ell, \mathit{com}(m_\ell), v_1, \ldots, v_{O(d)}, t) \in L', \tag{7}$$

where the language L' is defined as follows. Equation (7) holds if the verifier $\mathcal{V}_1^{\mathcal{F},\tilde{G}}$, 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 $\mathcal{F}$, 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 $\mathcal{V}_1^{\mathcal{F},\tilde{G}}$ 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 κ). 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 $V_{\text{ZK}}(x)$ 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 $(P_{\text{ZK}}(x, w), V_{\text{ZK}}(x))$ is $k \cdot \text{poly}(\kappa, d)$, it suffices to prove that L' is an $\mathcal{NP}$ 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 $m_1, \ldots, m_\ell$ can be computed in time $k \cdot \text{poly}(\kappa, d)$. Moreover, given:

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

checking whether $\mathcal{V}_1^{\mathcal{F},\tilde{G}}$ accepts the transcript $(r_1, m_1, \ldots, r_\ell, m_\ell)$, assuming that the oracle answers of $\mathcal{F}$ are $v_1 \ldots, v_{O(d)}$, and given the value $\tilde{G}(z)$, 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 $\tilde{G}$ 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 1.7: The proof of Theorem 1.7 is almost identical to the proof (and protocol) of Theorem 1.6. The only differences are that now the circuit computing $\mathcal{R}_L$ 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 1.6 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 4.6, if the circuit computing $\mathcal{R}_L$ is itself $\mathcal{L}$-uniform, then $\tilde{add}_i, \tilde{mult}_i$ 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 of Section 4 to show that the prover can simply send to the verifier the values of $\tilde{add}_i, \tilde{mult}_i$ at the points that the verifier needs. The prover then proves that it sent the correct values using the protocol of Theorem 4.4 (with soundness $O(1/d)$). The verifier's running time to verify the values of $\tilde{add}_i, \tilde{mult}_i$ on the desired points is only $\text{polylog}(n)$.

We modify the above protocol of Theorem 1.6 in a similar manner. Instead of computing the values $v_1, \ldots, v_{O(d)}$ of $\tilde{add}_i, \tilde{mult}_i$ 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 1.6 (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. ■

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 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 the result of a computation. 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 cases of verifying that $x \notin L$, and of verifying function computations rather than language membership, 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 M and 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 m_V in advance (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 ($\mathcal{L}$-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), 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 2.6 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 6.1. *[KR09] Let $\kappa \geq \log n$ be a security parameter. Assume the existence of a secure PIR scheme (as defined in Definition 2.6), with communication $\mathrm{poly}(\kappa)$, receiver work $\mathrm{poly}(\kappa)$, and sender work $\mathrm{poly}(n, \kappa)$ (where n is the database size).*

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

1. *Completeness c, soundness s and communication complexity ℓ.*

2. *Verifier running time $t_{\mathcal{V}}$ and prover running time $t_{\mathcal{P}}$.*

3. *Each message sent by the prover depends only on the λ previous bits sent by $\mathcal{V}$.*

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 $(\mathcal{P}', \mathcal{V}')$ for L, with communication complexity $\ell' = \mathrm{poly}(\ell, \kappa)$, completeness $c' \geq c - 2^{-\kappa^2}$, and soundness $s' \leq s + 2^{-\kappa^2}$ against (possibly non-uniform) provers of size $\leq 2^{\kappa}$. The verifier $\mathcal{V}'$ runs in time $\leq t_{\mathcal{V}} \cdot \mathrm{poly}(\kappa)$. The prover $\mathcal{P}'$ runs in time $\leq \mathrm{poly}(t_{\mathcal{P}}, \kappa, 2^{\lambda})$.

Moreover, the resulting one-round argument system $(\mathcal{P}', \mathcal{V}')$ has the property that the first message, sent by $\mathcal{V}'$, depends only on the random coin tosses of $\mathcal{V}'$, 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, we directly obtain efficient one-round arguments for delegating computation:

Theorem 6.2 (Theorem 1.3 of Section 1.2, restated)**.** *Let L be a langauge 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 $\mathrm{poly}(\kappa)$, receiver work $\mathrm{poly}(\kappa)$, and sender work $\mathrm{poly}(n, \kappa)$ (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 $\mathrm{poly}(S(n))$, the verifier runs in time $n \cdot \mathrm{poly}(\kappa, d(n), \log(S(n)))$.*[31]

[31]Moreover, if the verifier is given oracle access to the low-degree extension of its input, then its running time is only $\mathrm{poly}(\kappa, d(n), \log(S(n)))$.

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 of size* $\leq 2^{\kappa^3}$*, 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* κ, *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 certificates are efficiently probabilistically checkable. The result of Micali [Mic94] on CS Proofs, requires the use of a random oracle, a primitive whose realization is by now notoriously questionable (though he obtains short certificates for any efficient computation, not only for $\mathcal{NC}$).

7 An Interactive PCP

In this section, we use the bare-bones protocol (described in Subsection 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 7.1 with a brief introduction to interactive PCPs, and then present our new construction in Subsection 7.2.

7.1 Preliminaries

More precisely, let $L = \{x : \exists w \;\; s.t. \;\; (x, w) \in \mathcal{R}_L\}$ be an $\mathcal{NP}$ language, described by a polynomial-time computable relation $\mathcal{R}_L$. Let p, q, ℓ, c, s be parameters as follows: p, q, ℓ are integers and c, s are reals, s.t. $0 \leq s < c \leq 1$ (informally, p is the *size* of the PCP string, q is the *number of queries* allowed to the PCP string, ℓ 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, ℓ, c, s as functions of the instance size n. An interactive PCP with parameters (p, q, ℓ, c, s) for membership in L is an interactive protocol between an (efficient) prover P and an (efficient) verifier V.[32] 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 π of p bits (think of π as an encoding of the witness w). The verifier is still not allowed to access π. The prover and the verifier then apply an interactive protocol, where the total number of bits communicated is ℓ. During the protocol, the verifier is allowed to access at most q bits of the string π. After the interaction, the verifier decides whether to accept or reject the statement $x \in L$.

Definition 7.1. [KR08] A pair (P, V) of probabilistic polynomial time interactive Turing machines is an *interactive PCP* for L with parameters (p, q, ℓ, c, s) if the requirements below hold. When such a pair exists, we say that $L \in \mathrm{IPCP}(p, q, \ell, c, s)$.

For every $(x, w) \in \mathcal{R}_L$, we require that the prover $P(x, w)$ generates a bit string π (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 π.

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 $\wedge, \vee, \neg, \oplus$) then L has an interactive PCP with the following parameters: the length of the PCP string is polynomial in the witness size (i.e., $p = \mathrm{poly}(|w|)$), it makes only a single query to the PCP oracle (i.e., $q = 1$), and it has poly-logarithmic communication complexity (i.e., $\ell = \mathrm{polylog}(|x|)$).

7.2 New Improved Interactive PCPs

We extend the results of [KR08]. We show that for every $\mathcal{NP}$ language $L = \{x : \exists w \;\; s.t. \;\; (x, w) \in \mathcal{R}_L\}$, if the relation $\mathcal{R}_L$ 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., $p = \mathrm{poly}(d, |w|)$), it makes only a single query to the PCP oracle (i.e., $q = 1$), and it has communication complexity $\ell = \mathrm{poly}(d, \log |x|)$. In particular, we match the parameters of [KR08] (up to polynomial factors) for any $\mathcal{R}_L$ that can be computed in 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 8 to construct "short" *efficient* probabilistically checkable arguments.

[32]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 also focus on efficient provers throughout this work.

Theorem 7.2. *Let $C : \{0,1\}^k \to \{0,1\}$ be a Boolean circuit of size S and depth d. Then, for any $\varepsilon \geq 1/S$,*[33] *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 π (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, ϵ, 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 7.2:

Corollary 7.3. *Let $L = \{x : \exists w \text{ s.t. } \mathcal{R}_L(x, w) = 1\}$ 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 depth of $\mathcal{R}_L$. Then, for any $\varepsilon > 1/n$,*[34]

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

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 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 ε in Theorem 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 $O(\log \frac{1}{s})$ and will increase the communication complexity ℓ by a factor of $O(\log \frac{1}{s})$.

Rather than proving Theorem 7.2 directly, as was done in [KR08], we prove a weaker version (stated in Theorem 7.4), that allows "many" queries to the PCP string. We note that this weaker version (Theorem 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 8.

We note that in [KR08] (Section 6) it was shown how to convert an interactive PCP with many queries into an interactive PCP with a single query, and in particular, how to get Theorem 7.2 from Theorem 7.4. Loosely speaking, the main idea for converting a many-query interactive PCP into a single-query one, is to have the new PCP string π' be the low-degree extension of the original PCP string π. Then, instead of sending all the queries to the new PCP string, the verifier will choose a random curve (or manifold) that goes through these queries, and ask the prover in the interactive proof for the value of π' on this curve (or manifold). Finally, he will query its PCP string on a single random point on this curve (or manifold) and check for consistency.

Theorem 7.4. *Let $C : \{0,1\}^k \to \{0,1\}$ be a Boolean circuit of size S and depth d. Then, for any soundness parameter $s > 2^{-S}$,*[35] *the satisfiability of C can be proven by an interactive*

[33]We require $\varepsilon > 1/S$ in order to ensure that the prover runs in time poly($|C|$). We could take $0 < \varepsilon < 1/S$ and then the prover's running time is polynomial in $1/\varepsilon$.

[34]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 $1/\varepsilon$.

[35]We require $s > 2^{-S}$ in order to ensure that the prover and verifier run in time poly($|C|$). We could take $0 < s < 2^{-S}$ and then the running time would be polynomial in $\log \frac{1}{s}$.

PCP *with the following parameters:* $p = \mathrm{poly}(k, d, \log S)$*,* $q = \mathrm{poly}(\log d, \log\log S, \log \frac{1}{s})$*,* $\ell = \mathrm{poly}(d, \log S, \log \frac{1}{s})$*, completeness* $c = 1$*, and soundness* s*. Moreover, the interactive PCP has the following two properties:*

1. *The* PCP *string* π (*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*, 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 7.4, starting with a high-level overview. Suppose we want an interactive PCP for proving that there exists a sting $w \in \{0,1\}^k$ such that $C(w) = 0$, where $C : \{0,1\}^k \to \{0,1\}$ is a circuit as in the theorem statement.

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

1. The PCP string π is simply the low-degree extension of the witness w.

2. Verify that π is close to a low-degree polynomial, by running a low degree test of [MR08] (described for completeness in Subsection 2.4).

3. Verify that the string w, encoded in the oracle π, satisfies $w \in \{0,1\}^k$.

4. Verify that the string w, encoded in the oracle π, 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 $\mathrm{poly}(k)$ and depth $\mathrm{polylog}(k)$. 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 $\mathrm{poly}(k)$, 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 $d = \mathrm{polylog}(k)$. 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 $\mathrm{polylog}(k)$ communication complexity) while [KR08] cannot go beyond AC^0.

Proof of Theorem 7.4. In what follows, we prove Theorem 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 7.4. Consider the following interactive PCP protocol (P, V) for proving the satisfiability of $C : \{0,1\}^k \to \{0,1\}$.

Parameters We set the parameters as follows:

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 3.2. In particular, $\mathbb{H}$ is an extension field of $\mathbb{GF}[2]$, $\mathbb{F}$ is an extension field of $\mathbb{H}$, and m, m' are integers, such that
$$d \leq |\mathbb{H}| \leq \text{poly}(d, \log S),$$
$$S \leq |\mathbb{H}|^m \leq \text{poly}(S),$$
$$k \leq |\mathbb{H}|^{m'} \leq \text{poly}(d, k),$$
and
$$|\mathbb{F}| \leq \text{poly}(|\mathbb{H}|).$$
Moreover, the parameters $\mathbb{H}, \mathbb{F}, m, m'$ should satisfy the following additional properties:
(a) $m' \geq 3$.
(b) $|\mathbb{F}| \geq m'(|\mathbb{H}| - 1)$.
(c) $2^{10}m' \sqrt[8]{\frac{(m')^2(|\mathbb{H}|-1)}{|\mathbb{F}}} \leq \frac{1}{12}$.
These properties guarantee that we can apply Lemma 2.4 with respect to $\mathbb{F}$, 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
$$C : \{0, 1\}^k \to \{0, 1\}$$
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 $(P(C, w), V^\pi(C))$**.**

1. **Computing the** PCP **string** π**.** The PCP string π is the low-degree-extension of w w.r.t. the parameters $\mathbb{H}, \mathbb{F}, m'$. Namely,
$$\pi \overset{\text{def}}{=} \text{LDE}_{\mathbb{H},\mathbb{F},m'}(w_0, w_1 \ldots, w_{k-1}).$$
The verifier is given oracle access to π. Note that $\pi : \mathbb{F}^{m'} \to \mathbb{F}$ is a multivariate polynomial of degree $|\mathbb{H}| - 1$ in each variable, and thus is of total degree $\leq m' \cdot (|\mathbb{H}| - 1)$.

2. **Running the low degree test on π.** The verifier V checks that π is close to an m'-variate polynomial $f : \mathbb{F}^{m'} \to \mathbb{F}$ that has total degree $\leq m' \cdot (|H| - 1)$. This is done by running the low degree test $(P_{\mathrm{LDT}}(\pi), V^{\pi}_{\mathrm{LDT}})$ described in Subsection 2.4. 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 $\mathbb{F}$). Let $\mathcal{F} = \{\tilde{\mathrm{add}}_i, \tilde{\mathrm{mult}}_i\}_{i \in [d]}$ be a set of functions corresponding to C (as defined in Subsection 3.1), such that for every $i \in [d]$, $\tilde{\mathrm{add}}_i$ is the *unique* low degree extension of add_i, and $\tilde{\mathrm{mult}}_i$ is the *unique* low degree extension of 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 $\mathrm{poly}(S)$).

   The prover and the verifier run the bare-bones protocol $(\mathcal{P}_1^{\mathcal{F}}(w), \mathcal{V}_1^{\mathcal{F}}(w))$ described in Section 3.2 for proving that $C(w) = 0$ (with respect to $\delta = |\mathbb{H}| - 1$). The prover $P(C, w)$ (of the interactive PCP system) emulates $\mathcal{P}_1^{\mathcal{F}}(w)$ by computing the functions in $\mathcal{F}$ on his own (and thus simulating the oracle $\mathcal{F}$). The verifier $V^{\pi}(C)$ (of the interactive PCP system) emulates $\mathcal{V}_1^{\mathcal{F}}(w)$ by computing the functions in $\mathcal{F}$ on his own, and using his oracle π instead of w.

   Recall that according to the third (additional) property of Theorem 3.1, the verifier $\mathcal{V}_1$ 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 $\mathbb{H}, \mathbb{F}, m'$). In this case, $\mathcal{V}_1$ 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 $\mathbb{F}$ can be represented by $O(\log d + \log\log S)$ bits). Since with high probability (assuming the low degree test passes), the oracle π of the interactive PCP is close to the low degree extension of w (with respect to $\mathbb{H}, \mathbb{F}, m'$), the verifier can use the oracle π of the interactive PCP as an oracle to the low degree extension of w.

   If the verifier $\mathcal{V}_1^{\mathcal{F}}(w)$ of the bare-bones protocol rejects then the verifier $V^{\pi}(C)$ 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 $\Psi : \mathbb{F}^k \to \mathbb{F}$, defined as follows:

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

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

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

Analysis of the protocol $(P(C, w), V^{\pi}(C))$. The fact that $\pi = \mathrm{LDE}_{\mathbb{H},\mathbb{F},m'}(w)$ implies that the PCP string is of size $p = |\mathbb{F}|^{m'} \cdot (\log|\mathbb{F}|) \leq \mathrm{poly}(k, d)$. Theorem 3.1 and Lemma 2.4 imply that the protocol $(P(C, w), V^{\pi}(C))$ has communication complexity $\ell = \mathrm{poly}(d, \log S)$ and completeness

$c = 1$. Note that our protocol queries π 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 $C : \{0,1\}^k \to \{0,1\}$, of size S and depth d, which is not satisfiable. Fix any unbounded (cheating) prover $\tilde{P}$, and any function $\tilde{\pi} : \mathbb{F}^{m'} \to \mathbb{F}$. Let E denote the event that $(\tilde{P}, V^{\tilde{\pi}})(C) = 1$, and let

$$s \stackrel{\text{def}}{=} \Pr[E].$$

Assume for the sake of contradiction that $s > \frac{11}{12}$. According to Lemma 2.4, there exists an m'-variate polynomial $f : \mathbb{F}^{m'} \to \mathbb{F}$ of degree $\leq m' \cdot (|\mathbb{H}| - 1)$ such that

$$\Pr_{z \in_R \mathbb{F}^{m'}} [\tilde{\pi}(z) = f(z)] \geq s - \varepsilon,$$

where ε is defined in Lemma 2.4. Let

$$\gamma \stackrel{\text{def}}{=} \Pr_{z \in_R \mathbb{F}^{m'}} [\tilde{\pi}(z) \neq f(z)] \leq 1 - (s - \varepsilon).$$

Our contradiction assumption (that $s > \frac{11}{12}$), together with our assumption that $\varepsilon \leq \frac{1}{12}$, implies that

$$\gamma \leq \frac{1}{6}. \tag{8}$$

Define $(\tilde{w}_0, \tilde{w}_1, \ldots, \tilde{w}_{k-1}) \in \mathbb{F}^k$ by

$$\tilde{w}_i \stackrel{\text{def}}{=} f(\alpha^{-1}(i)),$$

where $\alpha : \mathbb{H}^{m'} \to \{0, 1, \ldots, k' - 1\}$ ($k' \stackrel{\text{def}}{=} |\mathbb{H}|^{m'}$) is the lexicographic order of $\mathbb{H}^{m'}$.

Recall that both times when emulating the verifier $\mathcal{V}_1$ 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

$$\Pr[\neg B] \leq 2\gamma \leq \frac{1}{3}. \tag{9}$$

Let A denote the event that $\tilde{w}_0, \tilde{w}_1, \ldots, \tilde{w}_{k-1} \in \{0,1\}$. Theorem 3.1 implies that

$$\Pr[E | A \wedge B] \leq \frac{1}{100},$$

and

$$\Pr[E | \neg A \wedge B] \leq \frac{1}{100}.$$

using some basic facts from probability theory, we conclude that

$$\begin{aligned} s = \Pr[E] \leq & \Pr[E|B] + \Pr[\neg B] \leq \\ & \Pr[E|A \wedge B] + \Pr[E|\neg A \wedge B] + \Pr[\neg B] \leq \\ & \frac{1}{100} + \frac{1}{100} + \frac{1}{3} < \frac{5}{12}, \end{aligned}$$

contradicting our assumption that $s \geq \frac{11}{12}$.

It remains to show that the two additional properties, required by the statement in Theorem 7.2, are attained.

1. The fact that π 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 $\pi \overset{\text{def}}{=} \text{LDE}_{\mathbb{H},\mathbb{F},m'}(w_0, w_1, \ldots, w_{k-1})$, and from the fact that the parameters $\mathbb{H}, \mathbb{F}, m'$ 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 Ψ.

   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.

■

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 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* $\mathcal{G}$, a *proof generation algorithm* $\mathcal{P}$, and a *verification algorithm* $\mathcal{V}$. It is also associated with five parameters κ, p, q, c, s, where κ, p, q are integers and c, s are reals, s.t. $0 \leq s < c \leq 1$. (Informally, κ 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 κ, p, q, c, s as functions of the instance size n.

Let L be an $\mathcal{NP}$ language, defined by $L = \{x : \exists w \ \ s.t. \ \ (x, w) \in R_L\}$. Suppose that Alice wishes to prove to Bob that $x \in L$. Assume that Bob applied in the past the challenge generation algorithm $\mathcal{G}$, and thus is associated with a pair of secret key and public challenge $(SK, PK) \leftarrow \mathcal{G}(1^\kappa)$. 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 $\pi \leftarrow \mathcal{P}(x, w, PK)$ of p bits. Bob is allowed to access at most q bits of the string π, and based on these bits he decides whether to accept or reject the statement $x \in L$.

Definition 8.1. [KR09] A triplet $(\mathcal{G}, \mathcal{P}, \mathcal{V})$ of probabilistic Turing machines is a **PCA system** for L with parameters (κ, p, q, c, s), if the following holds:

- $\mathcal{G}$ is a probabilistic Turing machine that runs in time $\text{poly}(\kappa)$, and $\mathcal{V}$ is a probabilistic oracle machine that runs in time $\text{poly}(\kappa, n)$.
- For every $(x, w) \in \mathcal{R}_L$ (where $|x| = n$) and every $(SK, PK) \leftarrow \mathcal{G}(1^{\kappa(n)})$, the algorithm $\mathcal{P}(x, w, PK)$ generates a bit string π of size at most $p(n)$, and the oracle machine $\mathcal{V}^{\pi}(x, SK, PK)$ reads at most $q(n)$ bits of π.
- **Completeness:** For every $(x, w) \in \mathcal{R}_L$ (where $|x| = n$),
$$\Pr[\mathcal{V}^{\pi}(x, SK, PK) = 1] \geq c(n)$$
(where the probability is over $(SK, PK) \leftarrow \mathcal{G}(1^{\kappa(n)})$, over $\pi \leftarrow \mathcal{P}(x, w, PK)$, and over the randomness of $\mathcal{V}$).
- **Soundness:** For every $x \notin L$ (where $|x| = n$), and every (possibly non-uniform) cheating prover $\tilde{\mathcal{P}}$ of size $\leq 2^{\kappa(n)}$,
$$\Pr[\mathcal{V}^{\tilde{\pi}}(x, SK, PK) = 1] \leq s(n)$$
(where $\tilde{\pi} = \tilde{\mathcal{P}}(PK)$, and the probability is over $(SK, PK) \leftarrow \mathcal{G}(1^{\kappa(n)})$ and over the randomness of $\mathcal{V}$).

Remark. Note that in Definition 8.1 we did not specify the complexity of $\mathcal{P}$. We say that a PCA system $(\mathcal{G}, \mathcal{P}, \mathcal{V})$ is *efficient* if $\mathcal{P}$ runs in time $\text{poly}(\kappa, n)$.

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

Theorem 8.2. *[KR09] Assume the existence of a (uniform) poly-logarithmic PIR scheme.*[36] *Assume that there exists an interactive* PCP *system $(\mathcal{P}, \mathcal{V})$ with parameters (p, q, ℓ, c, s) for some $\mathcal{NP}$ language L, with the following properties:*

1. *for every input $x \in L$, every auxiliary input $w \in \{0,1\}^*$,*[37] *and for every $i \in [\ell]$, the message sent by the (honest) prover $\mathcal{P}(x, w)$ in the i'th round of the protocol $(\mathcal{P}(x, w), \mathcal{V}(x))$ depends only on the message sent by $\mathcal{V}$ in the i'th round of the protocol (and on x, w and the random coin tosses of $\mathcal{P}$),*[38] *and does not depend on the messages sent by $\mathcal{V}$ before the i'th round.*

[36]The definition of a (uniform) poly-logarithmic PIR scheme can be found in Section 2.6.

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

[38]We think of each round as consisting of a message sent by the verifier $\mathcal{V}$ followed by a message sent by the prover $\mathcal{P}$.

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 π, and the input x), and can be computed in time $\leq \mathrm{poly}(\ell)$.*

Then, for any security parameter $\kappa \geq \max\{\ell, \log n\}$ there exists a PCA *system $(\mathcal{G}', \mathcal{P}', \mathcal{V}')$ with parameters (κ, p', q', c', s') for the language L, where $p' = \mathrm{poly}(p, \kappa)$, $q' = \mathrm{poly}(q, \kappa)$, $c' \geq c - 2^{-\kappa^2}$, and $s' \leq s + 2^{-\kappa^2}$. The prover $\mathcal{P}'$ runs in time $\leq \mathrm{poly}(\kappa, n, 2^{\lambda})$, where λ is the length of the longest message sent from $\mathcal{V}$ to $\mathcal{P}$ in the interactive phase of the interactive* PCP *system $(\mathcal{P}, \mathcal{V})$.*

Applying Theorem 8.2 to our interactive PCP system, results with a PCA system that is both *efficient* and *short* for many $\mathcal{NP}$ languages, as stated in the following theorem.

Theorem 8.3. *Assume the existence of a (uniform) poly-logarithmic PIR scheme. Fix any $\mathcal{NP}$ language $L = \{x : \exists w \text{ s.t. } (x, w) \in \mathcal{R}_L\}$. Let $n = |x|$ denote the instance size, let $k = |w|$ denote the witness size, and let d denote the depth of $\mathcal{R}_L$. Then for any soundness parameter $s > 2^{-n}$ and for any security parameter $\kappa \geq \mathrm{poly}(d, \log n, \log \frac{1}{s})$ there exists a PCA system for L with parameters (κ, p', q', c', s'), where $p' = \mathrm{poly}(k, \kappa)$, $q' \leq \mathrm{poly}(\kappa)$, $c' \geq 1 - 2^{-\kappa^2}$, and $s' \leq s + 2^{-\kappa^2}$. Moreover, the prover of this PCA system runs in time $\leq \mathrm{poly}(n, \kappa)$.*

Remark. Applying Theorem 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 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 polylog(n).

9 Acknowledgements

We thank Ran Raz and Salil Vadhan for their encouragement, illuminating conversations and insightful comments. We also thank anonymous referees for many helpful comments.

References

[AIK10] Benny Applebaum, Yuval Ishai, and Eyal Kushilevitz. From secrecy to soundness: Efficient verification via secure computation. In *ICALP (1)*, pages 152–163, 2010.

[AKS04] Manindra Agrawal, Neeraj Kayal, and Nitin Saxena. PRIMES is in P. *Annals of Mathematics*, 160(2):781793, 2004.

[ALM$^+$98] Sanjeev Arora, Carsten Lund, Rajeev Motwani, Madhu Sudan, and Mario Szegedy. Proof verification and the hardness of approximation problems. *J. ACM*, 45(3):501–555, 1998.

[And03] David P. Anderson. Public computing: Reconnecting people to science. In *Conference on Shared Knowledge and the Web*, 2003.

[And04] David P. Anderson. BOINC: A system for public-resource computing and storage. In *GRID*, pages 4–10, 2004.

[AS98] Sanjeev Arora and Shmuel Safra. Probabilistic checking of proofs: A new characterization of np. *J. ACM*, 45(1):70–122, 1998.

[Bab85] László Babai. Trading group theory for randomness. In *STOC*, pages 421–429, 1985.

[Bar01] Boaz Barak. How to go beyond the black-box simulation barrier. In *FOCS*, pages 106–115, 2001.

[BBFG91] Richard Beigel, Mihir Bellare, Joan Feigenbaum, and Shafi Goldwasser. Languages that are easier than their proofs. In *Proceedings of the 32nd Annual IEEE Symposium on Foundations of Computer Science*, pages 19–28, 1991.

[BCCT12] Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again. In *ITCS*, pages 326–349, 2012.

[BCCT13] Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. Recursive composition and bootstrapping for snarks and proof-carrying data. In *STOC*, pages 111–120, 2013.

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

[BFLS91] László Babai, Lance Fortnow, Leonid A. Levin, and Mario Szegedy. Checking computations in polylogarithmic time. In *Proceedings of the 23rd Annual ACM Symposium on Theory of Computing*, pages 21–31. ACM, 1991.

[BG02] Boaz Barak and Oded Goldreich. Universal arguments and their applications. In *Proceedings of the 17th Annual IEEE Conference on Computational Complexity*, pages 194–203, 2002.

[BGG$^+$88] Michael Ben-Or, Oded Goldreich, Shafi Goldwasser, Johan Håstad, Joe Kilian, Silvio Micali, and Phillip Rogaway. Everything provable is provable in zero-knowledge. In *CRYPTO*, pages 37–56, 1988.

[BGH$^+$06] Eli Ben-Sasson, Oded Goldreich, Prahladh Harsha, Madhu Sudan, and Salil P. Vadhan. Robust pcps of proximity, shorter pcps, and applications to coding. *SIAM J. Comput.*, 36(4):889–974, 2006.

[BGKW88] Michael Ben-Or, Shafi Goldwasser, Joe Kilian, and Avi Wigderson. Multi-prover interactive proofs: How to remove intractability assumptions. In *Proceedings of the 20th Annual ACM Symposium on Theory of Computing*, pages 113–131, 1988.

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

[Blu87] Manuel Blum. How to prove a theorem so no-one else can claim it. In *Proceedings of the International Congress of Mathematicians*, pages 1444–1451, 1987.

[BV11] Zvika Brakerski and Vinod Vaikuntanathan. Efficient fully homomorphic encryption from (standard) lwe. In *FOCS*, pages 97–106, 2011.

[CD97] Ronald Cramer and Ivan Damgård. Linear zero-knowledge - a note on efficient zero-knowledge proofs and arguments. In *Proceedings of the 29th Annual ACM Symposium on Theory of Computing*, pages 436–445, 1997.

[CGH04] Ran Canetti, Oded Goldreich, and Shai Halevi. The random oracle methodology, revisited. *Journal of the ACM*, 51(4):557–594, 2004.

[CKGS98] Benny Chor, Eyal Kushilevitz, Oded Goldreich, and Madhu Sudan. Private information retrieval. *Journal of the ACM*, 45(6):965–981, 1998.

[CKLR11] Kai-Min Chung, Yael Tauman Kalai, Feng-Hao Liu, and Ran Raz. Memory delegation. In *CRYPTO*, pages 151–168, 2011.

[CKV10] Kai-Min Chung, Yael Tauman Kalai, and Salil P. Vadhan. Improved delegation of computation using fully homomorphic encryption. In *CRYPTO*, pages 483–501, 2010.

[CL88] Anne Condon and Richard E. Ladner. Probabilistic game automata. *Journal of Computer and System Sciences*, 36(3):452–489, 1988.

[CL89] Anne Condon and Richard J. Lipton. On the complexity of space bounded interactive proofs (extended abstract). In *Proceedings of the 30th Annual IEEE Symposium on Foundations of Computer Science*, pages 462–467, 1989.

[CMS99] Christian Cachin, Silvio Micali, and Markus Stadler. Computationally private information retrieval with polylogarithmic communication. In *EUROCRYPT*, pages 402–414, 1999.

[CMT12] Graham Cormode, Michael Mitzenmacher, and Justin Thaler. Practical verified computation with streaming interactive proofs. In *ITCS*, pages 90–112, 2012.

[Con91] Anne Condon. Space-bounded probabilistic game automata. *Journal of the ACM*, 38(2):472–494, 1991.

[DFH12] Ivan Damgård, Sebastian Faust, and Carmit Hazay. Secure two-party computation with low communication. In *TCC*, pages 54–74, 2012.

[Din07] Irit Dinur. The pcp theorem by gap amplification. *Journal of the ACM*, 54(3):12, 2007.

[DNRS03] Cynthia Dwork, Moni Naor, Omer Reingold, and Larry J. Stockmeyer. Magic functions. *Journal of the ACM*, 50(6):852–921, 2003.

[DR06] Irit Dinur and Omer Reingold. Assignment testers: Towards a combinatorial proof of the PCP theorem. *SIAM J. Comput.*, 36(4):975–1024, 2006.

[DS92a] Cynthia Dwork and Larry J. Stockmeyer. Finite state verifiers i: The power of interaction. *Journal of the ACM*, 39(4):800–828, 1992.

[DS92b] Cynthia Dwork and Larry J. Stockmeyer. Finite state verifiers ii: Zero knowledge. *Journal of the ACM*, 39(4):829–858, 1992.

[DS02] Cynthia Dwork and Larry J. Stockmeyer. 2-round zero knowledge and proof auditors. In *STOC*, pages 322–331, 2002.

[FGL$^+$96] Uriel Feige, Shafi Goldwasser, László Lovász, Shmuel Safra, and Mario Szegedy. Interactive proofs and the hardness of approximating cliques. *Journal of the ACM*, 43(2):268–292, 1996.

[FK97] Uriel Feige and Joe Kilian. Making games short (extended abstract). In *Proceedings of the 29th Annual ACM Symposium on Theory of Computing*, pages 506–516, 1997.

[FL93] Lance Fortnow and Carsten Lund. Interactive proof systems and alternating time-space complexity. *Theoretical Computer Science*, 113(1):55–73, 1993.

[For89] Lance Fortnow. Complexity-theoretic aspects of interactive proof systems. PhD thesis. Technical Report MIT/LCS/TR-447, Massachusetts Institute of Technology, 1989.

[FS86] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *CRYPTO*, pages 186–194, 1986.

[Gen09] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *STOC*, pages 169–178, 2009.

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

[GGP10] Rosario Gennaro, Craig Gentry, and Bryan Parno. Non-interactive verifiable computing: Outsourcing computation to untrusted workers. In *CRYPTO*, pages 465–482, 2010.

[GGPR13] Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. Quadratic span programs and succinct nizks without pcps. In *EUROCRYPT*, pages 626–645, 2013.

[GK03] Shafi Goldwasser and Yael Tauman Kalai. On the (in)security of the fiat-shamir paradigm. In *FOCS*, pages 102–, 2003.

[GLR11] Shafi Goldwasser, Huijia Lin, and Aviad Rubinstein. Delegation of computation without rejection problem from designated verifier cs-proofs. *IACR Cryptology ePrint Archive*, 2011:456, 2011.

[GMR89] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof-systems. *SIAM Journal on Computing*, 18(1):186–208, 1989.

[GMW91] Oded Goldreich, Silvio Micali, and Avi Wigderson. Proofs that yield nothing but their validity, or all languages in np have zero-knowledge proof systems. *Journal of the ACM*, 38(1):691–729, 1991.

[Gol99] Oded Goldreich. *Modern cryptography, probabilistic proofs and pseudorandomness*, volume 17 of *Algorithms and Combinatorics*. Springer-Verlag, Berlin, 1999.

[Gol01] Oded Goldreich. *The Foundations of Cryptography - Volume 1*. Cambridge University Press, 2001.

[GR13] Tom Gur and Ron Rothblum. Non-interactive proofs of proximity. *Electronic Colloquium on Computational Complexity (ECCC)*, 20:78, 2013.

[Gro10] Jens Groth. Short pairing-based non-interactive zero-knowledge arguments. In *ASIACRYPT*, pages 321–340, 2010.

[GW11] Craig Gentry and Daniel Wichs. Separating succinct non-interactive arguments from all falsifiable assumptions. In *STOC*, pages 99–108, 2011.

[HILL99] Johan Håstad, Russell Impagliazzo, Leonid A. Levin, and Michael Luby. A pseudorandom generator from any one-way function. *SIAM J. Comput.*, 28(4):1364–1396, 1999.

[IKOS07] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. Zero-knowledge from secure multiparty computation. In *Proceedings of the 39th Annual ACM Symposium on Theory of Computing*, pages 21–30, 2007.

[IP07] Yuval Ishai and Anat Paskin. Evaluating branching programs on encrypted data. In *TCC*, pages 575–594, 2007.

[Kil88] Joe Kilian. Zero-knowledge with log-space verifiers. In *FOCS*, pages 25–35, 1988.

[Kil92] Joe Kilian. A note on efficient zero-knowledge proofs and arguments (extended abstract). In *Proceedings of the 24th Annual ACM Symposium on Theory of Computing*, pages 723–732, 1992.

[Kil95] Joe Kilian. Improved efficient arguments (preliminary version). In *CRYPTO*, pages 311–324, 1995.

[KO97] Eyal Kushilevitz and Rafail Ostrovsky. Replication is not needed: Single database, computationally-private information retrieval. In *FOCS*, pages 364–373, 1997.

[KR06] Yael Tauman Kalai and Ran Raz. Succinct non-interactive zero-knowledge proofs with preprocessing for logsnp. In *Proceedings of the 47th Annual IEEE Symposium on Foundations of Computer Science*, pages 355–366. IEEE Computer Society, 2006.

[KR08] Yael Tauman Kalai and Ran Raz. Interactive pcp. In *ICALP (2)*, pages 536–547, 2008.

[KR09] Yael Tauman Kalai and Ran Raz. Probabilistically checkable arguments. In *CRYPTO*, 2009.

[KRR13] Yael Tauman Kalai, Ran Raz, and Ron D. Rothblum. Delegation for bounded space. In *STOC*, pages 565–574, 2013.

[KRR14] Yael Tauman Kalai, Ran Raz, and Ron D. Rothblum. How to delegate computations: The power of no-signaling proofs. In *STOC*, 2014.

[LFKN92] Carsten Lund, Lance Fortnow, Howard J. Karloff, and Noam Nisan. Algebraic methods for interactive proof systems. *J. ACM*, 39(4):859–868, 1992.

[Lip05] Helger Lipmaa. An oblivious transfer protocol with log-squared communication. In *ISC*, pages 314–328, 2005.

[Lip12] Helger Lipmaa. Progression-free sets and sublinear pairing-based non-interactive zero-knowledge arguments. In *TCC*, pages 169–189, 2012.

[LMN93] Nathan Linial, Yishay Mansour, and Noam Nisan. Constant depth circuits, fourier transform, and learnability. *Journal of the ACM*, 40(3):607–620, 1993.

[Mer07] The great internet mersenne prime search, project webpage. http://www.mersenne.org/, 2007.

[Mic94] Silvio Micali. Cs proofs (extended abstract). In *Proceedings of the 35th Annual IEEE Symposium on Foundations of Computer Science*, pages 436–453. IEEE, 1994.

[MR08] Dana Moshkovitz and Ran Raz. Sub-constant error low degree test of almost-linear size. *SIAM J. Comput.*, 38(1):140–180, 2008.

[Nao89] Moni Naor. Bit commitment using pseudo-randomness. In *CRYPTO*, pages 128–136, 1989.

[Nao03] Moni Naor. On cryptographic assumptions and challenges. In *CRYPTO*, pages 96–109, 2003.

[PRV12] Bryan Parno, Mariana Raykova, and Vinod Vaikuntanathan. How to delegate and verify in public: Verifiable computation from attribute-based encryption. In *TCC*, pages 422–439, 2012.

[PS94] Alexander Polishchuk and Daniel A. Spielman. Nearly-linear size holographic proofs. In *Proceedings of the 26th Annual ACM Symposium on Theory of Computing*, pages 194–203, 1994.

[RAD78] Ron Rivest, Leonard Adleman, and Michael Dertouzos. On data banks and privacy homomorphisms. In *Foundations on Secure Computation*, pages 169–179, 1978.

[RS96] Ronitt Rubinfeld and Madhu Sudan. Robust characterizations of polynomials with applications to program testing. *SIAM J. Comput.*, 25(2):252–271, 1996.

[RV09] Guy Rothblum and Salil Vadhan. Are pcps inherent in efficient arguments? Unpublished manuscript, to appear in CCC, 2009.

[RVW13] Guy N. Rothblum, Salil P. Vadhan, and Avi Wigderson. Interactive proofs of proximity: delegating computation in sublinear time. In *STOC*, pages 793–802, 2013.

[SET99] ET, phone SETI@home!. Science@NASA headlines, 1999.

[SET07] SETI@home project website. http://setiathome.berkeley.edu/, 2007.

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

[Tha13] Justin Thaler. Time-optimal interactive proofs for circuit evaluation. In *CRYPTO (2)*, pages 71–89, 2013.

[TRMP12] Justin Thaler, Mike Roberts, Michael Mitzenmacher, and Hanspeter Pfister. Verifiable computation with massively parallel interactive proofs. *CoRR*, abs/1202.1350, 2012.

[VSBW13] Victor Vu, Srinath T. V. Setty, Andrew J. Blumberg, and Michael Walfish. A hybrid architecture for interactive verifiable computation. In *IEEE Symposium on Security and Privacy*, pages 223–237, 2013.

[WB13] Michael Walfish and Andrew J. Blumberg. Verifying computations without reexecuting them: from theoretical possibility to near-practicality. *Electronic Colloquium on Computational Complexity (ECCC)*, 20:165, 2013.