Distributed Zero-Knowledge Proofs Over Networks

Aviv Bick[*] Gillat Kol[†] Rotem Oshman[‡§]

Abstract

Zero knowledge proofs are one of the most influential concepts in theoretical computer science. In the seminal definition due to Goldwasser, Micali and Rackoff dating back to the 1980s, a computationally-bounded verifier interacts with a powerful but untrusted prover, with the goal of becoming convinced that the input is in some language. In addition to the usual requirements of completeness and soundness, in a zero knowledge proof, we protect the prover's knowledge: assuming the prover is honest, anything that the verifier can deduce after interacting with the prover, it could have deduced by itself. Zero knowledge proofs have found many applications within theoretical computer science and beyond, e.g., in cryptography, client-cloud computing, blockchains and cryptocurrencies, electronic voting and auctions, and in the financial industry.

We define and study the notion of *distributed zero knowledge proofs*, reconciling the computational notion of zero-knowledge with the communication-based paradigm of distributed graph algorithms. In our setting, a *network* of verifiers interacts with an untrusted prover to decide some *distributed* language. As is usually the case in distributed graph algorithms, we assume that the verifiers have local views of the network and each only knows its neighbors. The prover, on the other hand, is assumed to know the entire network graph, as well as any input that the verifier may possess. As in the computational centralized setting, the protocol we design should protect this knowledge. In particular, due to the dual role of the underlying graph in distributed graph algorithms, serving as both the communication topology and the input to the problem, our protocol must protect the graph itself.

We construct communication-efficient distributed zero knowledge proofs for two central problems: the 3-coloring problem, one of the poster children of computational zero-knowledge, and for the spanning-tree verification problem, a fundamental building block for designing graph algorithms. We also give a general scheme for converting proof labeling-schemes to distributed zero-knowledge protocols with related parameters. Our protocols combine ideas from computational complexity, distributed computing, and cryptography.

1 Introduction

In the classical setting of interactive proofs, an efficient (polynomial-time) verifier interacts with a computationally unbounded but untrusted prover, in order to verify a claim of the form "$x \in L$" [20] (see also [2]). The interaction of the verifier and the prover is said to be *zero knowledge* if, whenever the prover follows the protocol, the verifier does not gain any knowledge from the communication, aside from the validity of the claim $x \in L$. Zero-knowledge protocols were shown to be extremely strong: virtually any language L that has an interactive proof system (that is, any language $L \in \mathsf{IP} = \mathsf{PSPACE}$ [37, 31]) has a zero-knowledge proof system that is (computationally) zero-knowledge ($\mathsf{CZK} = \mathsf{IP}$) [5].

Zero-knowledge proofs were the original motivation for the introduction of interactive proofs [20], and have since found many applications within theoretical computer science and beyond, including applications to cryptography (e.g., enforcing truthful behavior in cryptographic protocols, constructing identification schemes), cloud computing and delegation, blockchains and cryptocurrencies (e.g., private smart contracts), electronic voting and auctions (e.g., allowing participants to verify that their vote was counted, without revealing anything else to them), and in the financial industry (e.g., to help banks protect their confidential data while meeting regulatory requirements).

[*]Blavatnik School of Computer Science, Tel Aviv University.

[†]Computer Science Department, Princeton University. Supported by an Alfred P. Sloan Fellowship, the National Science Foundation CAREER award CCF-1750443, and by the E. Lawrence Keyes Jr. / Emerson Electric Co. Award.

[‡]Blavatnik School of Computer Science, Tel Aviv University. Research funded by the Israel Science Foundation, Grant No. 2801/20, and also supported by Len Blavatnik and the Blavatnik Family foundation.

[§]This work was funded by BSF grant no. 0603808692.

Defining (lack of) knowledge. Zero-knowledge protocols aim to protect the prover's "*knowledge*". Although the prover and the verifier have the same information (namely, they both know the language L and the element x), the prover may have more "knowledge", stemming from its greater computational power. Consider, for example, the scenario where the prover wishes to convince the verifier that the graph G is 3-colorable. Since the prover is computationally unbounded, if G is indeed 3-colorable, the prover can simply compute a proper coloring (the "NP-witness") and give it to the verifier; the verifier is then able to verify that the coloring is proper in polynomial time. This naïve solution is not considered zero-knowledge, because the verifier learns a proper coloring of the graph. But what if instead of fully learning a proper coloring, the verifier only learns *something* about the proper colorings of the graph, for example, that some large set $\mathcal{S}$ of potential colorings does not contain a proper coloring? What constitutes "knowledge" or "learning"?

The brilliant classical definition of zero-knowledge, due to [20], asserts that the verifier learns nothing from the prover if the verifier is able to *simulate* its conversation with the prover by itself (i.e., without actually interacting with the prover). More formally, an interactive proof is zero-knowledge if there exists an efficient (polynomial-time) algorithm, called a *simulator*, such that for any input $x \in L$, running the simulator on x produces an output that has the same distribution as the verifier's conversation with the prover on the input x. According to this definition, ruling out a set $\mathcal{S}$ of polynomially-many colorings does not count as new knowledge, because the verifier could have done it by itself (recall that the verifier is polynomial-time); but if $\mathcal{S}$ is of size, say, $2^{\sqrt{n}}$, then ruling out $\mathcal{S}$ may count as knowledge.

1.1 Defining distributed zero-knowledge.

Our goal is to adapt the classical centralized definition of zero-knowledge to the distributed network setting. The starting point for our work is the recently introduced framework of *interactive distributed proofs* [26] (see also [33]), which adapts the notion of interactive proofs to the distributed setting, and follows a long line of work on distributed non-interactive proofs [30, 29, 27, 28, 36, 4, 21, 35, 10, 15, 38, 14]. In this framework, a network of verifiers, each of which holds part of the input and only knows its neighbors, interacts with an untrusted prover that knows the entire network and all inputs, in order to compute a function of the network and inputs. The scarce resource here is communication; no computational limitations on the communicating parties are assumed (although computationally-efficient protocols are preferred).

In this work, we seek to protect the prover's additional "knowledge" in these interactive distributed proofs from the verifiers. Such zero-knowledge distributed proofs can allow central entities holding massive amounts of data — such as Facebook, knowing the topology of a social network, or 23andMe and other genetic companies having large graphs describing gene propagation — to convince their clients of some truth about the network, without revealing their "knowledge" about the network.

We emphasize that in distributed graph algorithms, the network graph often serves a dual role: it is both the communication topology and part of the *input* to the problem we want to solve (e.g., finding a minimum-weight spanning tree over the network graph, or finding a coloring of the graph). Initially, each node only knows the network size and its own neighbors and input (in some settings, e.g. [30], nodes may know even less), and the prover is the only entity that knows the entire graph topology and input. This global view of the network (and inputs) is the prover's advantage over the nodes; this is the "knowledge" we wish to protect.

For example, in many distributed proofs in the literature, the prover is asked to compute a spanning tree of the network, and give each node its parent in the tree. The nodes verify that the spanning tree is valid, and they can then verify sums of their inputs and other functions by checking them "up the tree", with the prover giving each node the partial value corresponding to its subtree [30]. However, this style of proof can reveal "global" information about the graph: e.g., in the spanning-tree based proofs in [30], each node is given its distance from the root of the tree (used to verify cycle-freeness of the "tree" given by the prover), providing the node with a lower bound on the diameter of the graph.

Towards a distributed definition. As in the case of centralized zero-knowledge proofs, for an interactive distributed proof to be considered zero-knowledge, the network should be able to *simulate* its interaction with the prover by itself, on any input in the language. Only information that cannot be efficiently computed by the network is considered "knowledge".

What does it mean that the network "can efficiently compute something by itself"? In the computational setting, efficient computations are typically modeled as the complexity class P. In the distributed world, efficiency can be modeled in many different ways, depending on the context and the physical layout of the network. Therefore, our definitions are parameterized by a class $\mathcal{A}$ of distributed algorithms that we consider "efficient" (e.g., 3-round $\mathcal{CONGEST}$ algorithms).

Inspired by the centralized definition of interactive proofs, in a distributed interactive proof Π, we allow the verifiers to:

(1) Communicate back-and-forth with the prover, and then

(2) Run a verification algorithm A from the class $\mathcal{A}$, to decide whether to accept or reject the proof.

In order for a protocol Π to be considered *distributed zero-knowledge*, the network nodes should be able to compute their *view* of the protocol Π by themselves: the view of each node consists of the node's randomness and input, and all the messages it sent or received from the prover and from other nodes when executing Π. Thus, there should exist an efficient distributed algorithm S that outputs at each node v a view that, as a random variable, has exactly the same distribution as the transcript that v would observe when interacting with the real prover. This means that whatever node v observed when interacting with the prover, it could have computed by itself using an efficient distributed algorithm. As in the case of centralized zero-knowledge, the algorithm S is called the *simulator*.

We emphasize that running the simulator — a distributed algorithm from the class $\mathcal{A}$ — may allow nodes to acquire information that they did not initially have: for example, a node can learn its neighbors' inputs. However, we do not consider this to be *knowledge*, because it can be computed by an efficient distributed algorithm; just as in the classical definition, an efficient verifier is able to learn something about the input (for example, ruling out a polynomially-sized family of 3-colorings), and this is not considered to be knowledge.

We also consider *coalitions* of nodes: what if after interacting with the prover, $k > 1$ nodes cooperate to try to gain knowledge by sharing their views with one another? Intuitively, to ensure that a proof system Π is zero-knowledge against coalitions of k nodes, the simulator S should be such that for every subset $V' \subseteq V$ of k or fewer nodes, the *joint distribution* of the outputs of the nodes in V' under S is identical to the joint distribution of the views of the nodes in V' under Π.

Our definition of distributed zero-knowledge. In light of the discussion above, we define distributed zero-knowledge as follows (see Section 2 for a formal definition): fix $n \in \mathbb{N}$ and a domain $\mathcal{X}$ from which the nodes' inputs are drawn. An *annotated graph* is a pair $(G = (V, E), I)$, where $I : V \to \mathcal{X}$ assigns an input to each node. A *distributed language* is a family of annotated graphs $L \subseteq \mathcal{G} \times \mathcal{X}^n$, where $\mathcal{G}$ is the set of all graphs on n vertices.

Let $\mathcal{A}$ be a non-empty set of distributed algorithms (our class of "efficient" algorithms). We first define the notion of $\mathcal{A}$*-interactive distributed proofs*, which generalize the notion of interactive distributed proofs from [26] by allowing the verifiers to run any algorithm from $\mathcal{A}$; then we define what it means for an $\mathcal{A}$-interactive distributed proof to be *zero-knowledge*.

An *interactive distributed proof system* [26] for a distributed language L consists of a set of algorithms, one for the prover and one for each node. To verify a claim of the form "$(G, I) \in L$", the prover interacts with each of the nodes over a series of synchronous rounds, and finally, each node decides whether to accept or reject. We require that if $(G, I) \in L$, then with high probability (say $2/3$) all the nodes accept; and if $(G, I) \notin L$, with high probability at least one node rejects. We define an $\mathcal{A}$*-interactive distributed proof* for a distributed language L to be as above, except that after the communication with the prover ends (and before they decide whether to accept to reject), we allow the nodes to run any protocol from the class $\mathcal{A}$ to jointly verify the prover's claims.

Next, let us define distributed zero-knowledge. Let $r, b, k \in \mathbb{N}$, $s < c \in [0, 1]$, $\mathcal{A} \neq \emptyset$. The class $\mathsf{dZK}[r, \ell, \mathcal{A}, k]$ is defined as the set of distributed languages L that have an $\mathcal{A}$-interactive distributed proof system Π, such that

1. In Π, the prover interacts with each node over r communication rounds, and in each round, the prover exchanges ℓ bits with the node.

2. In addition, Π satisfies the following *distributed zero-knowledge property*: there exists a simulator protocol $S \in \mathcal{A}$ such that for every $(G, I) \in L$, and for every coalition of nodes $V' \subseteq V$ of size $|V'| \leq k$, it holds that

$$(\mathsf{out}_S(G, I))_{V'} = (\mathsf{VIEW}(\Pi, G, I))_{V'} .$$

Here, the equality sign denotes equality between distributions, $(\mathsf{out}_S(G, I))_{V'}$ is the *joint distribution* over the outputs of all nodes $v \in V'$ on an execution of the simulator S on (G, I), and $(\mathsf{VIEW}(\Pi, G, I))_{V'}$ is the *joint distribution* over the views of all nodes $v \in V'$ at the end of the execution of the protocol Π on (G, I).

Strong distributed zero-knowledge. As we explained above, in a distributed zero-knowledge proof we are concerned with protecting the prover's global view of the network and inputs; whatever an efficient distributed algorithm (from the class $\mathcal{A}$) can compute by itself is not considered "knowledge". However, in some cases we may wish to impose an even stronger requirement, and ask that nodes learn *nothing* they did not initially know. This corresponds to weakening the simulator: while the verifiers will still use a protocol from the class $\mathcal{A}$ to verify the proof, the simulator S should now be a *zero-round* protocol, i.e., a protocol that does not communicate at all, where each node simulates its view completely by itself. We call this stronger notion *strong distributed zero-knowledge*, and denote it by $\mathsf{dSZK}[r, \ell, \mathcal{A}, k]$.

While our notion of distributed zero knowledge is meant to protect the prover's knowledge from the network nodes, à la centralized zero knowledge, *strong* zero knowledge protects the knowledge of the prover and of all the nodes in the network from node v, so that v learns nothing it did not already know,[1] à la secure multi-party computation (see Section 1.4 for an additional discussion).

Distributed knowledge as a continuum. We mention that we derive the definition of both dZK and dSZK as special cases of the more general definition of *distributed knowledge*, denoted $\mathsf{dK}[r, \ell, \mathcal{A}_V, \mathcal{A}_S, k]$. The definition of dK replaces the single class $\mathcal{A}$ of "efficient" protocols with two classes, $\mathcal{A}_V$ and $\mathcal{A}_S$: we allow the nodes to run any protocol from $\mathcal{A}_V$ to verify the prover's claims, and to run any protocol from $\mathcal{A}_S$ as a simulator (see Section 2 for a formal definition). We view $\mathcal{A}_V$ as the "real" or the "physical" capability of the network, the set of protocols that the nodes can actually carry out, and we view $\mathcal{A}_S$ as a set of protocols that reveal little knowledge.

The definition of dZK is derived from that of dK by taking $\mathcal{A}_V = \mathcal{A}_S$. In this case, anything that the network nodes can deduce from the interaction with the prover is within their physical power to calculate. This means that the "extra" knowledge of the prover is protected. (In fact, we view any setting in which $\mathcal{A}_V$ is at least as powerful as $\mathcal{A}_S$ (i.e., $\mathcal{A}_S \subseteq \mathcal{A}_V$) as protecting the prover's knowledge.) The definition of dSZK is obtained by taking this to the extreme, and setting $\mathcal{A}_S$ to be the class of zero-round, no-communication protocols. Note, however, that the definition of dK allows us to view the protection of the knowledge of other nodes as a continuum, and we can choose to protect the nodes' knowledge only partially by taking $\mathcal{A}_S$ to be a larger subset of $\mathcal{A}_V$.

By taking $\mathcal{A}_S$ to be more powerful than $\mathcal{A}_V$ in the definition of dK, we get other interesting classes that are not zero knowledge per se, but do admit some bound on the knowledge they reveal: by interacting with the prover, the nodes cannot learn more than what they could have deduced had they been able to physically run protocols from $\mathcal{A}_V$.[2]

1.2 Motivating example: 2-colorability vs. 3-colorability. To get a feel for our new definitions, we next explain why the "natural" distributed proof for verifying 2-colorability is, in fact, distributed zero-knowledge, while the "natural" proof for 3-colorability is not.

Before diving into the distributed setting, recall that as a computational problem, 3-colorability is an NP-complete problem, and was one of the early examples studied in the context of centralized zero-knowledge: based on cryptographic assumptions, [19] gave a zero-knowledge protocol for 3-colorability, and thus for all of NP. In contrast, the 2-colorability (bipartiteness) problem is in P, and thus has a trivial zero knowledge proof, where the verifier computes the answer without even consulting the prover. In the distributed setting, however, 2-colorability is a very hard problem: nodes must be able to distinguish between odd and even cycles of length $\Theta(n)$, which requires $\Omega(n)$ communication rounds.

2-colorability. Some distributed proofs are naturally zero-knowledge, or can easily be made so. For example, consider the natural interactive distributed proof for 2-colorability: if the graph is indeed 2-colorable, the honest prover computes a legal 2-coloring, gives each node v its color $c_v \in \{0, 1\}$, and every node verifies that its neighbors $u \in N(v)$ all have $c_u \neq c_v$. This proof can be made strong distributed zero-knowledge by having the prover apply a random permutation to the colors $\{0, 1\}$ before giving them to the nodes: with probability $1/2$ the color names remain unchanged, and with probability $1/2$ we swap color 0 with color 1 everywhere in the graph. With this minor modification, even a zero-communication simulator can easily compute the view of a given node v in a

[1]This is the case for coalitions of size $k = 1$. In general, distributed strong zero knowledge protects the knowledge of the prover and all the nodes not in a coalition from a coalition.

[2]An analogue in the centralize setting is, for example, allowing the zero-knowledge simulator to be $\mathsf{quasi} - \mathsf{P}$ instead of P, while the verifier still works in P. In this setting, we can say that the verifier, whose "real" power is P, cannot gain more from the interaction than if it were able to run $\mathsf{quasi} - \mathsf{P}$ algorithms.

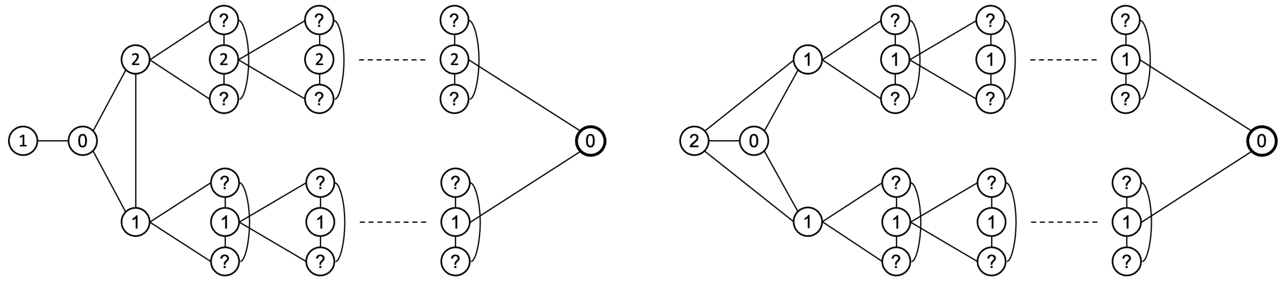

Figure 1: Example of non-trivial information that a node learns in the natural 3-colorability proof: the rightmost node (shown in bold) can distinguish between these two graphs by comparing its neighbors' colors. The figure depicts the only possible 3-coloring for each graph, up to permuting the color names.

2-colorable graph: node v receives from the prover a uniformly random color $c_v \in \{0, 1\}$, and from each neighbor $u \notin N(v)$ it receives the color $1 - c_v$.

3-colorability. In contrast to 2-colorability, the natural interactive distributed proof for 3-colorability is *not* zero-knowledge, nor can it be made zero-knowledge by simply shuffling the names of the colors at random. Consider the distributed proof for 3-colorability where the prover gives each node $v \in V$ its color $c_v \in \{0, 1, 2\}$, and each node v sends c_v to its neighbors and checks that they received a different color than c_v. Now that we have three colors rather than two, a node can learn non-trivial information about the graph by comparing the colors of its neighbors, even if the prover randomly permutes the color names; for example, a node can learn that a specific pair of its neighbors have the same color.

Fig. 1 shows two graphs where the rightmost node (shown in bold) has the same initial view: in both graphs it has two neighbors and is given the same ID (note that the numbers in the figure indicate colors, not IDs). In the graph on the left, both neighbors of the rightmost node will always have different colors, under any legal 3-coloring; in the graph on the right, the two neighbors will always have the *same* color. Therefore, if nodes tell one another their colors, the rightmost node is able to distinguish the two graphs, even though they are identical up to distance $\Theta(n)$ from it. It follows that for any reasonably "local" or "efficient" class $\mathcal{A}$, this natural proof for 3-colorability is not distributed zero-knowledge, and it is certainly not *strong* distributed zero-knowledge. We can overcome this difficulty by constructing a dSZK proof where the prover does give the nodes their colors, but they verify that the coloring is proper without directly sending their colors to one another (see Section 3).

We note, however, that the approach above, where the prover gives nodes their colors in some legal 3-coloring, cannot be used to obtain coalition-resilient dZK protocols. In fact, the example from Fig. 1 demonstrates that in any distributed proof that is resilient to coalitions of size at least 2, if the prover gives the nodes their colors, then the proof can only have a highly inefficient simulator, which runs in $\Omega(n)$ rounds. The reason is that if the two neighbors of the rightmost node in Fig. 1 collude, they can distinguish between the two graphs shown in the figure by comparing their colors. We see that to handle coalitions, the prover should not even tell a node its own color; instead, we will *split* the color of a node between nearby nodes using secret-sharing. The challenge now comes from deciding *which* nodes will receive shares for which nodes' colors, and how these nodes should verify that the coloring is proper, without requiring too much communication or disclosing non-local information about the graph.

1.3 Our results.

In this paper we adopt the popular $\mathcal{CONGEST}$ model of distributed network algorithms, and model the class of "efficient" protocols by $\mathcal{A} = \mathcal{C}(r', b)$, the class of $\mathcal{CONGEST}$ algorithms that run for r' synchronized rounds and send b bits on every edge in each round. Our goal is to minimize both r', the "locality" of the algorithm, and b, its communication, as these two parameters govern the information that algorithms from the class $\mathcal{C}(r', b)$ can collect about the annotated network. In addition, we would like to minimize the number of communication rounds with the prover and the number of bits that are exchanged with the prover, because the prover is an external entity, and interacting with it may be expensive.

1.3.1 Our constructions.

We give the following constructions for distributed zero-knowledge proofs.

dSZK for 3-colorability. While Section 1.2 shows that the naïve distributed proof for 3-colorability is not zero-knowledge, in Section 3 we show that there is a relatively simple distributed strong zero-knowledge proof for this problem. Our protocol is inspired by the seminal graph non-isomorphism protocol of [19]: the prover gives each node its color in a randomly-permuted 3-coloring, but instead of sending their colors directly to one another, every pair of neighboring nodes challenge the prover with a question that it can only reliably answer correctly if it gave them different colors.

THEOREM 1.1. *3-colorability* $\in \mathsf{dSZK}[3, O(\Delta), \mathcal{C}(1, O(1)), 1]$.

Here, Δ is the maximum degree in the graph (which may not be known to the nodes in advance). In our proof, each node u actually sends $O(\deg(u))$ bits to the prover. It is an intriguing open question whether there is a dZK proof for 3-colorability where the total number of bits exchanged with the prover is $o(|E|)$.

dZK for spanning-tree verification. One of the most central problems in distributed computing is to construct or to verify a spanning tree of the network. In the verification version, every node v is given a parent $p(v)$, and we wish to verify that the edges $\{\{v, p(v)\} : v \in V\}$ indeed form a spanning tree. We show that the spanning-tree verification problem, *STVer*, admits a distributed zero-knowledge proof with essentially *no overhead* compared to the non-interactive, non-zero-knowledge version of the proof: it is known that a PLS for *STVer* requires $\Omega(\log n)$ bits [30, 21], and our dZK proof achieves the same.

THEOREM 1.2. *STVer* $\in \mathsf{dZK}[1, O(\log n), \mathcal{C}(1, O(\log n)), 1]$.

We believe that the key to the efficiency of our proof is that the *input* to spanning-tree verification already induces some sparse structure over the graph topology, which, since it is part of the input, is not considered "knowledge we need to protect". Our protocol works by computing linear shares for the size of each node's subtree, carefully partitioning the shares between the node and its children, and homomorphically computing sums over the shares to verify that the tree contains no cycles and spans the network. See Section 4 for the details.

A general compiler. While the construction in Theorem 1.1, 1.2 are simple and and achieve good parameters, they only work against a single adversary (i.e., $k = 1$). However, we show that for every distributed language, it is possible to protect the prover's knowledge against larger coalitions: we give a compiler that takes a *proof labeling-scheme (PLS)* [30] and converts it into a distributed zero-knowledge proof, whose parameters are related to the length of the PLS and the size of the circuit describing it. A PLS is a particularly simple form of distributed proof, where the prover assigns each node a label, and the nodes then send one another their labels, and decide whether to accept or reject. (We mention that while for simplicity we state our results only for PLS, similar claims can be proved for all interactive distributed proofs.)

Our compiler is more involved, and it is based on *fully-linear PCPs* [8]. It was already shown in [8] that fully-linear PCPs can form the basis for a form of distributed zero-knowledge, but in [8] the communication topology was the complete graph, and there was no need to protect it. In our compiler, the prover distributes linear shares for each node's label in the PLS, as well as for a fully-linear PCP showing that the node would accept this label and its neighbors' labels, among $k+1$ nodes in the node's vicinity. The nodes then cooperate to collect the shares required to verify, for each $v \in V$, that v would accept.

The challenge on the distributed side is to choose an assignment of the shares of each node to other nodes in its vicinity, in a way that allows for efficient verification, and also ensures that the very assignment itself does not reveal too much information about the graph. We remark that typical building blocks that might be used for this purpose, such as coloring and dominating sets, cannot be used here, because computing them is known to require learning information from a radius of $\Omega(\log^* n)$ around each node (while we would like to avoid any dependence on n in the locality). In Section 6 we show how to assign to each node a set of k "helper nodes" at distance at most k from itself, such that nodes can communicate with their helper nodes without causing too much congestion, and the assignment itself can be computed in $\tilde{O}(k)$ rounds, using messages of size $O(k^2 \log n)$. This yields the following generic compiler:

THEOREM 1.3. *Let L be a distributed language that admits a PLS where every node receives a label comprising ℓ bits from the prover, collects its neighbors' labels, and uses a circuit of size at most s to decide whether to accept.*[3]

[3]For instance, 3-colorability has a PLS where every node receives a label of $\ell = 2$ bits, and can decide whether to accept or reject

Then, for every $k = o(n)$,

$$L \in \mathsf{dZK}[1, O(k^2(\Delta\ell + s)\log n), \mathcal{C}(\tilde{O}(k), O(k^2(\Delta\ell + s)\log n)), k].$$

1.4 Related work. We briefly discuss prior work on distributed proofs, zero-knowledge proofs, and fully-linear PCPs.

Interactive distributed proofs and distributed NP. As noted above, *interactive distributed proofs* were recently introduced by [26] (see also [3]). They show that interaction can exponentially decrease the communication cost compared to a non-interactive prover; however, some problems still retain non-trivial costs even with interaction. The work of [33] constructs a general scheme for transforming any interactive proof with a central verifier into a distributed interactive protocol, giving an excellent example as to how general ideas can be transferred between different models.

There is a large body of work studying the relative expressive power on various notions of "distributed NP", where a proof is distributed between a set of nodes, but there is no interaction with the prover; e.g., [30, 29, 27, 28, 36, 4, 21, 35, 10, 15]; we refer to the excellent surveys [38, 14] for a comprehensive overview.

Distributed zero-knowledge for a complete network. In [8], the notion of a *fully-linear PCP* is defined, and is used to construct distributed zero-knowledge proofs for a setting where all parties communicate with one another over private channels. The definition of zero-knowledge adopted in [8] is similar to secure MPC, or to what we call strong zero-knowledge with coalitions of size up to $n-1$ in the current paper. We use fully-linear PCPs in our generic compiler, in a way that is similar to their use in [8]: to verify that a node would accept its label and its neighbors' labels, while having the labels themselves split among $k+1$ nodes using linear secret-sharing. We emphasize that in [8], all nodes can talk to one another directly, and the input and proof are shared among all nodes. On the other hand, our compiler is *local*: the information relevant to a node v is shared only to within distance $O(k)$ of v, and the topology of the network beyond distance $O(k)$ is not revealed.

Secure multi-party computation over networks. Secure multi-party computation (SMPC) protocols allow a group of parties to jointly compute a value that depends on their private inputs, without revealing anything about their inputs, except for what is given by the computed value. There is a tremendous body of work studying SMPC for a wide range of security notions, starting from general SMPC compilers by [39, 18, 6, 13], which provide protocols for computing any function $f(x_1, \ldots, x_n)$ securely and can even tolerate large coalitions. These general results, and many others, assume that the parties communicate through private channels, and are designed for single-hop (clique) networks. The problem of SMPC over different multi-hop networks in various settings were also considered, see, e.g., [17, 12, 23, 22, 9, 11, 34].

Our definition of distributed zero knowledge is incomparable (or at least not directly comparable) to the models of SMPC over general networks studied in the literature. We assume the existence of a central "all knowing" prover, while in the SMPC setting, no single node has all the information. However, in our setting, the graph topology is unknown to the nodes, and each only has a local view of it. In fact, the graph is a part of the prover's "additional knowledge" (together with all inputs) that we wish to protect. In contrast, SMPC typically assumes that the network is known to all nodes, and only the nodes' inputs should be protected. We mention that the task of hiding the underlying network was recently considered as well: [24] give some negative results in the information-theoretic setting (which is the setting we adopt in this paper), and [32] gives a positive result under computational assumptions.

While the problems of distributed zero knowledge and SMPC over general networks are inherently different, they are also related: recall that our primary interest is to protect the knowledge of our *prover*. However, the information given to the prover is, collectively, known to the nodes. Thus, by protecting the prover, we *partially* also protect the *nodes* from each other. Nevertheless, the view we take in our definition of distributed zero knowledge considers the nodes to be an alliance that cooperates in order to check the prover's claims (by running the algorithm $A \in \mathcal{A}$), potentially sacrificing some of their own privacy in order to do so. Our definition of *strong distributed zero knowledge* more closely echoes SMPC, as it fully protects each node's privacy.

1.5 Discussion and open problems. In this paper we initiate the study of distributed zero-knowledge proofs and make some first steps towards understanding their power. Our work leaves many open problems, and we highlight a few interesting future directions.

using a linear-sized circuit.

Lower bounds. In this paper we design distributed zero-knowledge protocols; it will be very interesting to prove lower bounds for such protocols. For instance, can one show that any distributed zero-knowledge protocol for 3-coloring requires $\Omega(|E|)$ communication with the prover? Although showing lower bounds on the communication required by classical zero-knowledge protocols (and related protocols that reveal no information) is a notoriously hard open problem, it may be much more feasible in the distributed setting, because no single node knows the entire input.

It is also an interesting problem to separate distributed zero-knowledge from *strong* distributed zero-knowledge in terms of their communication cost, for reasonable classes $\mathcal{A}$ of distributed algorithms, such as the one we consider here.

Cheating verifiers. Our definition of distributed zero knowledge corresponds to the classical definition of *perfect* zero-knowledge with an *honest-but-curious* verifier. Many other variants of zero-knowledge proofs were studied in the literature, most notably the model of cheating verifiers: while here we assume that the verifiers always follow the protocol, a *cheating* (or *malicious*) verifier may deviate from the protocol in order to extract knowledge from the prover. Our protocols are not zero-knowledge against cheating verifiers; for example, in Section 3 we show that in our 3-coloring protocol, a single cheating verifier can obtain non-trivial and highly non-local information about the graph. Whether our protocols can be generalized to handle cheating verifiers is an intriguing question.

2 Defining Distributed Zero-Knowledge

Fix a network size $n \in \mathbb{N}$, and let $V = \{v_1, \ldots, v_n\}$ be the nodes of the network. We assume that n is known in advance to all the network nodes.[4] Let $\mathcal{G}(V)$ be the set of all graphs $G = (V, E)$. We use $N(v)$ to denote the neighborhood of a node $v \in V$.

Annotated graphs and distributed languages. Let $\mathcal{X}$ be a (possibly empty) input domain, and let $\mathcal{I}_{\mathcal{X}}(V)$ be the set of all assignments $I : V \to \mathcal{X}$ of inputs to network nodes. Let $\hat{\mathcal{G}} = \hat{\mathcal{G}}(V) = \mathcal{G}(V) \times \mathcal{I}_{\mathcal{X}}(V)$ be the set of annotated graphs where each node $v \in V$ receives the input $I(v)$. We abuse the notation by implicitly assuming that the input assignment I also specifies for each node a unique identifier from some domain $\{1, \ldots, N\}$, and a port numbering. We typically conflate a node with its ID in our notation.

A *distributed language* is a family of annotated graphs, $L \subseteq \hat{\mathcal{G}}$.

Distributed interactive proofs. We model a distributed interactive proof as a protocol that runs in a port-labeled network, where each node $v \in V$ of degree d has $d+1$ ports: port 0 is used to communicate with the prover, and ports $1, \ldots, d$ to communicate with v's neighbors. The nodes of the network have unique identifiers drawn from a domain $\{1, \ldots, N\}$, where N is polynomial in the network size n. Initially, each node knows only its own identifier, its degree, its input, and the network size n, which is fixed throughout. Each network node, as well as the prover, has *private randomness*, which is hidden from all other participants. Shared randomness between the nodes, or between the nodes and the verifier, is not assumed.

A distributed interactive proof is parameterized by the number of interaction rounds with the prover (r), the number of bits each node can send or receive from the prover in a given round (ℓ), and the class of possible verification protocols for the nodes ($\mathcal{A}$).[5] An $(r, \ell, \mathcal{A}_V)$*-protocol with a prover* proceeds as follows:

1. The network nodes communicate back-and-forth with each other and with the prover for r rounds, with the prover going first. In each round, either
   - The prover sends each node a message of length ℓ, or
   - Each node sends the prover a message of length ℓ, or
   - Each node sends each of its neighbors ℓ fresh random bits (different bits on each edge), which are not revealed to the prover.

[4]It is interesting to consider networks whose size is not known in advance, but in that case a zero-knowledge protocol should also protect the size of the network — if the nodes do not know it a-priori, they must not learn it. In fact, all of our protocols except the spanning-tree protocol also work under the assumption that the nodes only know some loose upper bound $n' = \text{poly}(n)$, but for simplicity we fix the network size throughout.

[5]We allow r and ℓ to be functions of the number of nodes ($r = r(n), \ell = \ell(n)$: when the distributed interactive proof is run with a graph G over n nodes, the number of interaction rounds with the prover will be $r(n)$, the number of bits each node can send or receive from the prover will be $\ell(n)$. We also assume that each protocol $A \in \mathcal{A}_V$ can be run over any annotated graph $\hat{G}$.

2. Following the interaction, the nodes run a protocol $A \in \mathcal{A}_V$ (the "verification" protocol), and each node outputs either "accept" or "reject".

(A more general definition would allow the nodes to run an algorithm from $\mathcal{A}$ in between every round of communication with the prover, but for the sake of simplicity, and to separate the cost of the communication with the prover from the cost of the network verification, we impose a more restricted structure.)

An annotated graph $\hat{G}$ is *accepted* if all nodes output "accept", and otherwise $\hat{G}$ is *rejected*.

Let $s < c \in [0,1]$. An $(r, \ell, \mathcal{A}_V)_{c,s}$*-distributed interactive proof system for a language* $L \subseteq \hat{\mathcal{G}}$ is an $(r, \ell, \mathcal{A}_V)$-protocol with a prover, $\Pi = (\mathsf{Prov}, \mathsf{Ver})$, satisfying the following two conditions:

- **Completeness:** For every $\hat{G} \in L$, Π accepts $\hat{G}$ with probability at least c.
- **Soundness:** For every $\hat{G} \notin L$, for any $\mathsf{Prov}^\star$, $\Pi^\star = (\mathsf{Prov}^\star, \mathsf{Ver})$ rejects $\hat{G}$ with probability at least $1 - s$.

Throughout the paper, we fix $s = 1/3, c = 1$ and omit c, s from our notation, unless stated otherwise.

Views. When executing a protocol Π with a prover, the *view* of a node $v \in V$ consists of v's input, randomness, and the list of all messages it received from the prover and from its neighbors over its ports. We think of v's view as a binary string encoded in some canonical representation. Let $\mathsf{VIEW}(\Pi, \hat{G})$ be a vector random variable, indexed by V, distributed according to the joint distribution of the views of all the nodes $v \in V$ in the execution of Π on $\hat{G}$.

For a distributed algorithm A, we denote by $\mathsf{out}_A(\hat{G})$ the vector random variable indexed by V, distributed according to the joint distribution of the views of all the nodes $v \in V$ when they execute A in $\hat{G}$.

Distributed knowledge in a network. Our definition of knowledge is parameterized by a class $\mathcal{A}_S$ of distributed algorithms, which captures the amount and type of information that the prover may leak, and the size $k \in \mathbb{N}$ of the semi-honest coalition. Intuitively, we want an adversary that observes the views of up to k nodes to learn no more than it could have learned by running an algorithm from the class $\mathcal{A}_S$.

DEFINITION 2.1. (THE CLASS dK) *Let $r, \ell, k \in \mathbb{N}$ and let $\mathcal{A}_V, \mathcal{A}_S$ be non-empty sets of distributed algorithms. The class $\mathsf{dK}[r, \ell, \mathcal{A}_V, \mathcal{A}_S, k]$ is the set of all distributed languages L, for which there exist an $(r, \ell, \mathcal{A}_V)$-distributed interactive proof system* $\Pi = (\mathsf{Prov}, \mathsf{Ver})$ *and a simulator $S \in \mathcal{A}_S$ such that for every subset $U \subseteq V$ of size $|U| = k$, we have*

$$\left(\mathsf{out}_S(\hat{G})\right)_U \equiv \left(\mathsf{VIEW}(\Pi, \hat{G})\right)_U .$$

Here, "$\equiv$" denotes identity as distributions.

Distributed zero-knowledge and strong zero-knowledge. Our notions of zero-knowledge and strong zero-knowledge are obtained from Definition 2.1 as follows:

- $\mathsf{dZK}[r, \ell, \mathcal{A}, k] = \mathsf{dK}[r, \ell, \mathcal{A}, \mathcal{A}, k]$, that is, the set of distributed languages for which there exists an $(r, \ell, \mathcal{A})$-distributed interactive proof that has a simulator from the same class as the verifier, $\mathcal{A}$.
- $\mathsf{dSZK}[r, \ell, \mathcal{A}, k] = \mathsf{dK}[r, \ell, \mathcal{A}, \mathcal{C}[0,0], k]$, that is, the set of distributed languages for which there exists an $(r, \ell, \mathcal{A})$-distributed interactive proof that can be simulated by a protocol that runs in 0 rounds and sends 0 bits.

We remark that for some problems, to obtain strong distributed zero-knowledge, it may be desirable to allow the simulator the use of *shared randomness* among the nodes in the coalition (as in secure MPC).

The class of distributed verifiers and simulators that we consider in this work is the class $\mathcal{CONGEST}$ of synchronous distributed algorithms with bounded message size. We denote by $\mathcal{C}[t, b]$ the family of $\mathcal{CONGEST}$ algorithms that run in at most t rounds and send messages of at most b bits each. $\mathcal{CONGEST}$ algorithms may use private randomness, but shared randomness is not available to the nodes.

3 Single-Adversary Strong Zero-Knowledge for 3-Colorability

In this section we give a simple protocol for 3-colorability that is strong zero-knowledge against an adversary that can take over a single node. We ask the prover to give each node its color (as in the naïve 3-colorability protocol we discussed in the introduction), but instead of directly comparing colors with its neighbors, the nodes interact with the prover again to convince themselves that the coloring is proper, *without* directly comparing colors with their neighbors. The protocol is inspired by the seminal zero knowledge proof for graph-non-isomorphism from [20]. The protocol we present here requires 3 rounds of interaction with the prover; we remark that using *correlated randomness* [25] provided by the prover, and verified by the nodes using the *cut-and-choose* technique (which we will use in later sections), we can also obtain a 1-round strong zero-knowledge protocol with the same message size.

Jointly sampling a random object. Our protocol uses a symmetric mechanism that allows two nodes u, v to agree on a uniformly random object X from some domain $\mathcal{D}$: let $\mathcal{D} = \{D_0, \ldots, D_{N-1}\}$. Nodes u, v independently choose uniformly random numbers $r_u, r_v \in \{0, \ldots, N-1\}$ and send them to each other. Then they both select the object D_r, where $r = (r_u + r_v) \bmod N$. In the sequel, when we say that two nodes "agree on a uniformly random $X \in \mathcal{D}$", we mean that they execute this protocol to choose X.

The protocol. In our protocol, the honest prover begins by choosing a proper 3-coloring of the graph, and then it applies a random permutation to the color names.1 Let $c : V \to \{0, 1, 2\}$ be the resulting 3-coloring. The prover gives each node v its color $c(v)$, which is uniformly random in $\{0, 1, 2\}$. Next, every pair u, v of neighboring nodes interact with the prover to try to convince themselves that $c(u) \neq c(v)$, without revealing their colors to one another:

- Nodes u, v agree on a uniformly random bit $b_{\{u,v\}} \in \{0, 1\}$. (The value of $b_{\{u,v\}}$ is kept secret from the prover.)
- If $b_{\{u,v\}} = 0$, nodes u, v agree on a uniformly random permutation $\pi_{\{u,v\}}$ of the colors $\{0, 1, 2\}$. Node u sends $\pi_{\{u,v\}}(c(u))$ to the prover, and node v sends $\pi_{\{u,v\}}(c(v))$ to the prover. (The nodes do not disclose these colors to one another.)
- If $b_{\{u,v\}} = 1$, nodes u, v agree on a uniformly random color $a_{\{u,v\}} \in \{0, 1, 2\}$, and both nodes send $a_{\{u,v\}}$ to the prover.
- The prover must respond by guessing the value of $b_{\{u,v\}}$ and sending it to nodes u and v. If the prover sends the wrong value, the nodes reject.

Every node executes this in parallel for all its ports, yielding an overall proof length of $O(\Delta)$: a node u of degree d sends the prover a list of d colors, one for each of its edges, and the prover responds with a list of d bits. Node u accepts iff all the bits are correct (that is, they match the values u agreed on with its respective neighbors).

Completeness. Our protocol has completeness 1: suppose the prover gives the nodes a proper coloring. Then for every edge $\{u, v\} \in E$, if nodes u, v chose $b_{\{u,v\}} = 0$, the prover receives different colors, $\pi_{\{u,v\}}(c(u)) \neq \pi_{\{u,v\}}(c(v))$, from u and from v; whereas if $b_{\{u,v\}} = 1$ is chosen, the prover gets the same color, $a_{\{u,v\}}$, from both nodes. The prover is always able to distinguish these two cases and return the correct value of $b_{\{u,v\}}$. Since this holds for every edge, the prover can cause all nodes to accept.

Soundness. The soundness of our protocol is 1/2: suppose the graph is not 3-colorable. Then there is some edge $\{u, v\} \in E$ such that $c(u) = c(v)$. In this case, regardless of whether u, v choose $b_{\{u,v\}} = 0$ or $b_{\{u,v\}} = 1$, the prover receives the same uniformly random color from both nodes. Thus, the prover can only guess $b_{\{u,v\}}$ with probability 1/2. Of course, repeating the protocol (sequentially) several times will decrease the soundness error.

Strong zero-knowledge. Our protocol is secure against a single honest adversary, as witnessed by the following local simulator:

- Through port 0, receive a uniformly random $c(u) \in \{0, 1, 2\}$;
- On every port $i \neq 0$,
  - Send a random number $b_i^{\rightarrow} \in \{0, 1\}$, and receive a random number $b_i^{\leftarrow} \in \{0, 1\}$. Set $b_i = b^{\rightarrow} \oplus b^{\leftarrow}$.
  - If $b_i = 0$, send a random number $\pi_i^{\rightarrow} \in \{0, \ldots, 5\}$ and receive a random number $\pi_i^{\leftarrow} \in \{0, \ldots, 5\}$. Set π to be the permutation on $\{0, 1, 2\}$ whose index is $(\pi_i^{\rightarrow} + \pi_i^{\leftarrow}) \bmod 6$ among the 6 such permutations. On port 0, send $\pi(c(u))$
  - If $b_i = 1$, send a random number $a_i^{\rightarrow} \in \{0, 1, 2\}$ and receive a random number $a_i^{\leftarrow} \in \{0, 1, 2\}$. Send on port 0 the number $(a_i^{\rightarrow} + a_i^{\leftarrow}) \bmod 3$.

- Receive on port 0 the list $b_1, \ldots, b_{\deg(u)}$.

Our protocol is strong zero-knowledge against an *honest-but-curious* adversary, which follows the protocol but attempts to extract information from its local view; the protocol is not secure against a *malicious* adversary, which can deviate from the protocol in order to extract information from the prover. To demonstrate, we show that a malicious adversary that takes over the rightmost node in Figure 1 can distinguish the two graphs depicted in Figure 1 with probability at least 25/48. This is better than a random coin-toss, which means that a malicious adversary is able to extract meaningful information about the network graph.

Let v denote the bottom node in the figure, and let u_0, u_1 be its two neighbors. Let c, c_0, c_1 denote the colors given by the prover to nodes v, u_0, u_1 (respectively).

- The adversary follows the first step of the protocol truthfully, to select random values $b_0, b_1 \in \{0, 1\}$ with nodes u_0, u_1 (respectively). If $b_0 \neq 0$ or $b_1 \neq 0$, the adversary quits, and guesses which graph it is (from the two depicted in Figure 1) at random.

- If $b_0 = b_1 = 0$, the adversary follows the next step to select random permutations π_0, π_1 with u_0, u_1 (resp.). If $\pi_0 \neq \pi_1$, the prover quits and guesses at random.

- Suppose the adversary has been "lucky": $b_0 = b_1 = 0$, and $\pi_0 = \pi_1$. According to the protocol, it is now supposed to send, for both edges, the same permuted color, $\pi_0(c) = \pi_1(c)$. Instead, the adversary chooses some other color $c' \neq \pi_0(c)$ and sends it to the adversary for both edges.

- If the prover returns the same guess for b_0 and b_1, the adversary guesses that it is the graph on the right (where its neighbors have the same color). Otherwise, the adversary guesses that it is the graph on the left (where its neighbors have different colors).

We claim that the adversary has probability 25/48 of guessing the correct graph: the probability that $b_0 = b_1 = 0$ and $\pi_0 = \pi_1$ is $(1/4) \cdot (1/6) = 1/24$. If this does not occur, the adversary guesses at random, and has probability 1/2 of guessing the right graph. Now assume $b_0 = b_1 = 0$ and $\pi_0 = \pi_1$, and for convenience, let us denote $\pi = \pi_0 = \pi_1$. Nodes u_0, u_v send $\pi(c_0), \pi(c_1)$ (resp.) to the prover, while node v sends a color $c' \neq \pi(c)$. We know that the honest prover will guess "$b_0 = 0$" iff $\pi(c_0) = c'$, and "$b_1 = 0$" iff $\pi(c_1) = c'$. Therefore, if $c_0 = c_1$, the prover will guess the same value, $b_0 = b_1$, for both edges, whereas if $c_0 \neq c_1$, it will guess different values, $b_0 \neq b_1$. The adversary guesses accordingly: if the prover guesses the same value for both edges, the adversary guesses that it is the left graph in Figure 1, and if the prover guesses different values, the adversary guesses that it is the right graph. This guess will always be correct.

The adversary's overall probability of guessing the correct graph is

$$\left(1 - \frac{1}{24}\right) \cdot \frac{1}{2} + \frac{1}{24} \cdot 1 = 25/48.$$

4 Single-Adversary Zero Knowledge Proof for Verifying a Spanning Tree

In this section we construct a zero-knowledge proof for verifying a given spanning tree. In this proof nodes will need to communicate with their parent in the tree, and learn how many siblings they have; thus, the proof is not *strong* zero-knowledge, and instead it is "plain" zero-knowledge with respect to the class $\mathcal{A}$ of constant-round $\mathcal{CONGEST}$ algorithms.

Suppose that each node $v \in V$ is given either the ID of a node $p_v \in V$ or $\perp$, and we wish to verify that the subgraph induced by the edges $H = \{\{v, p_v\} : v \in V, p_v \neq \perp\}$ is a spanning tree of the network.[6] If we were not concerned about the knowledge that verifiers acquire, we would simply ask the prover to give every node $v \in V$ the size $s_v \in \mathbb{N}$ of v's subtree; then, each node v would verify that

$$s_v = 1 + \sum_{u : p_u = v} s_u, \tag{4.1}$$

[6]In networks that do not have unique identifiers, we can encode the input by asking for the parent's port rather than its actual ID, and the protocol remains the same.

and we also verify that at the root r we have $s_r = n$, and at every leaf v we have $s_v = 1$.[7] We refer to (4.1) as the *consistency condition.* If the consistency condition is satisfied, it guarantees that H is cycle-free, since $s_v < s_{p_v}$ for each node v, unless $p_v = \perp$. In addition, by verifying that any node v with $p_v = \perp$ has $s_v = n$, and that any node v that is a leaf has $s_v = 1$, we guarantee that H is spanning. Together, H must be a spanning tree.[8]

In a zero-knowledge proof, we cannot afford to give s_v to every node v, because this discloses non-trivial information about the graph: e.g., if $s_v = 2$ but node v has only one child, it learns that its child has another neighbor other than itself; this is "distance 2 information" for v. It is not difficult to extend this example to show that disclosing s_v can reveal to v information about the network graph at distance $\Omega(n)$ from itself.

Instead of providing s_v, we ask the prover to divide s_v into two *shares*, using Shamir's secret-sharing scheme. Each share by itself is uniformly random and independent of s_v, but from the two shares together we can reconstruct s_v. We give one share of s_v to v itself, and we give the other share to all of v's children, and also to another node in v's vicinity (more on this below). Then we verify consistency by computing *over* the shares, as is done in secure MPC (e.g., [6]), without ever reconstructing the value of s_v. One can also view this protocol as using *correlated randomness*, provided by the prover, to allow the nodes to securely verify the consistency condition without revealing their shares.

Next, we give a high-level description of the spanning tree verification protocol, followed by a detailed description and the proof of correctness.

4.1 Overview of the protocol. We first describe a simplified version where for each $v \in V$, node v is given one share of s_v, and the other share is given to all of v's children and also to v's parent (if it has children and/or a parent, respectively). This results in potentially long proofs, since nodes with high degree in the tree will need to receive many shares; we then show how to distribute the shares more efficiently in the network, to reduce the proof length to $O(\log n)$.

Hiding the subtree size. Let $q \in [n, 2n]$ be a prime number, and let $\mathbb{F}_q^c[x]$ denote the univariate polynomials of degree at most c over the field $\mathbb{F}_q$, in the formal variable x.

For each node $v \in V$, the prover divides the size s_v of v's subtree into two shares, as follows: the prover samples a line $S_v = a_0 + a_1 x \in \mathbb{F}_q^1[x]$, such that $S_v(0) = a_0 = s_v$, and a_1 is uniformly random in $\mathbb{F}_q$. Then it produces the two shares $S_v(1), S_v(2)$. Note that $S_v(1)$ and $S_v(2)$ by themselves are uniformly random in $\mathbb{F}_q$ and independent of the secret $S_v(0) = s_v$, but together they lie on the unique line S_v, allowing us to reconstruct S_v and recover $S_v(0) = s_v$. The prover gives one share of S_v to node v, and the other share to its parent and children.

If v is the root or a leaf, the prover gives v *both* shares of S_v, allowing v to reconstruct $S_v(0) = s_v$ and verify that $s_v = n$ if v is the root, or that $s_v = 1$ if v is a leaf. This does not violate the zero-knowledge property: the honest prover *always* gives $s_r = n$ to the root r, and $s_v = 1$ to a leaf v. Nodes can learn whether they are the root or a leaf using one round of communication, by simply having each node v send p_v to all its neighbors. Therefore no "non-local information" is conveyed by allowing the root and leafs to learn s_v.

In the sequel, we describe how the shares for the subtree size s_v are distributed and how their consistency is checked. To simplify the discussion, we do not discuss the root here; it is handled in the detailed version of the protocol given in Section 4.2.

Choosing evaluation points. As we explained above, the shares of each s_v are two evaluations $S_v(1), S_v(2)$ of a random polynomial S_v such that $S_v(0) = s_v$. We would like node v to receive one share of s_v, and the children and parent of v to receive the other share. We have the freedom to choose whether v gets $S_v(1)$ and its children get $S_v(2)$, or vice-versa, but our choice must not reveal information about the tree.

To decide which nodes will get which shares, the prover computes a 2-coloring of the tree (that is, it partitions the tree into odd and even layers), using the colors $\{1, 2\}$. Then, as in the 2-colorability proof from Section 1.2, the prover randomly permutes the color names, and gives each node its permuted color. The nodes verify that their colors indeed represent a proper 2-coloring of H. The color assigned to each node determines which shares it will receive: if v is colored 1, it will receive the share $S_v(1)$, and its children and parent will receive the share $S_v(2)$; and vice-versa if v is colored 2.

[7]The root r knows that it is the root, because it has $p_r = \perp$, and leafs can learn in a single round that they are leafs, because none of their neighbors have them as parents.

[8]This proof is different from the one originally given in [30], where all nodes are given the ID of the root and their distance from it. The proof from [30] can also be adapted to obtain a zero-knowledge version, but this is more complicated, since in a zero-knowledge proof we do not want to tell all nodes the ID of the root of the tree.

Checking consistency of s_v. For convenience, let us denote $\overline{1} = 2, \overline{2} = 1$. The prover distributes the shares in the tree so that for each node v with children $u_1, \ldots, u_\ell$,

- v has one share $S_v(a)$ of s_v, and one share $S_{u_i}(a)$ of each s_{u_i} $(i = 1, \ldots, \ell)$, for some $a \in \{1, 2\}$.
- Each child u_i of v has the other share $S_v(b)$ of s_v, and the other share $S_{u_i}(b)$ of s_{u_i}, where $b = \overline{a}$.

Let $q' \in [2n^2, 3n^2]$ be a prime number.[9] To verify the consistency condition at node v, we would like to evaluate the polynomial $S_v(x) - \sum_{i=1}^{\ell} S_{u_i}(x)$ at $x = 0$, and check that the answer is 1; since $S_v(0) = s_v$ and $S_{u_i}(0) = s_{u_i}$ for each child u_i, this guarantees that $s_v = \sum_{i=1}^{\ell} s_{u_i} + 1$, as required. However, we must carry out the verification in a way that does not reveal $s_v, s_{u_1}, \ldots, s_{u_\ell}$ themselves, so instead of evaluating $S_v(0) - \sum_{i=1}^{\ell} S_{u_i}(0)$ directly, we will evaluate the polynomial

$$Z(x) := [S_v(x) + X_v(x)] - \sum_{i=1}^{\ell} [S_{u_i}(x) + Y_{u_i}(x)]$$

at $x = 0$, where $X_v(x), Y_{u_1}, \ldots, Y_{u_\ell} \in \mathbb{F}_{q'}^1$ are random lines such that $X_v(0) = \sum_{i=1}^{\ell} Y_{u_i}(0)$. We have

$$Z(0) = [S_v(0) + X_v(0)] - \sum_{i=1}^{\ell} [S_{u_i}(0) + Y_{u_i}(0)] = S_v(0) - \sum_{i=1}^{\ell} S_{u_i}(0),$$

so verifying that $Z(0) = 1$ ensures consistency; the addition of the random lines $X_v, Y_{u_1}, \ldots, Y_{u_\ell}$ protects the prover's secrets.

To this end, we ask the prover to sample uniformly random lines $X_v, Y_{u_1}, \ldots, Y_{u_\ell} \in \mathbb{F}_{q'}^1[x]$ subject to $X_v(0) = \sum_{i=1}^{\ell} Y_{u_i}(0)$. We cannot trust that the prover will actually give us lines satisfying the constraint, so instead, we use the *cut-and-choose* technique: we ask the prover for t independent copies, where t is a security parameter. We choose one random copy, and reveal all the rest, to verify that the constraint was satisfied for all of them (otherwise we reject). We then use the one copy that was not revealed for the remainder of the protocol. If the prover cheats by giving us at least one copy that does not satisfy the constraint, then we will catch it with probability at least $1 - 1/t$.

We evaluate $Z(0)$ as follows:

- The prover gives $X_v(a), Y_{u_1}(a), \ldots, Y_{u_\ell}(a)$ to node v, and $X_v(b), Y_{u_i}(b)$ to each child u_i.
- Each child u_i sends $S_{u_i}(b) + Y_{u_i}(b)$ to the parent v, and some child sends $S_v(b) + X_v(b)$ as well.
- The parent v computes

  $$Z(a) = [S_v(a) + X_v(a)] - \sum_{i=1}^{\ell} [S_{u_i}(a) + Y_{u_i}(a)]$$

  from the shares it was given by the prover, and

  $$Z(b) = [S_v(b) + X_v(b)] - \sum_{i=1}^{\ell} [S_{u_i}(b) + Y_{u_i}(b)]$$

  using the information it was sent by its children. It then interpolates to find Z, and verifies that $Z(0) = 1$.

We note that all the values the children send to v are independent of the shares $S_v(b), S_{u_1}(b), \ldots, S_{u_\ell}(b)$, even given all the other information v has. Thus, v learns nothing about $s_v, s_{u_1}, \ldots, s_{u_\ell}$, because it has only one share for each of them.

[9]Our verification procedure involves modular arithmetic, which is carried out in a field of size greater than any intermediate value we might operate on when interacting with an honest prover, allowing us to pretend that we are working over the natural numbers. We also show that it is not possible for a cheating prover to exploit the fact that we use modular arithmetic.

Verifying share equality across layers. For each node v, the prover is meant to give the share $S_v(\overline{c_v})$ to v's parent and children. The prover may try to cheat by claiming different values for $S_v(\overline{c_v})$ to different nodes. To be very explicit, let us denote by $[\![V]\!]_u$ the value that the prover gave node u for the variable V (in our case V is $S_v(\overline{c_v})$); then we must ensure that v's parent p and any child u of v received the same value, $[\![S_v(\overline{c_v})]\!]_p = [\![S_v(\overline{c_v})]\!]_u$. However, we cannot ask p and u to send the values $[\![S_v(\overline{c_v})]\!]_p, [\![S_v(\overline{c_v})]\!]_u$ to one another in order to compare them, because the only path connecting p and u may go through v itself, and we do not want to reveal $S_v(\overline{c_v})$ to v!

Instead, we ask the prover to produce for node v and its neighbors a "one-time pad", a uniformly random value $M_v \in_U \mathbb{F}_q$, and to give M_v to node v and its parent and children. We verify that the prover gives the same values using the cut-and-choose technique, as we did for $X_v, Y_{v,u}$ above: the prover provides the nodes with t copies, $\left\{M_v^j\right\}_{j=1}^t$. Node v chooses a uniformly random $j_v \in_U \{1, \ldots, t\}$ and sends it to its parent and children, who respond by sending back the values they received for $\left\{M_v^j\right\}_{j \neq j_v}$. Node v verifies that all nodes sent the same value, ensuring that if the prover tried to cheat by giving $\left[\!\!\left[\left\{M_v^j\right\}_{j=1}^t\right]\!\!\right]_{u_1} \neq \left[\!\!\left[\left\{M_v^j\right\}_{j=1}^t\right]\!\!\right]_{u_2}$ to two neighbors u_1, u_2 of v, then v catches it with probability at least $1 - 1/t$. Finally, we use the remaining one-time pad, $M_v^{j_v}$, to encrypt $S_v(\overline{c_v})$: each node u that is either a parent or a child of v sends $[\![S_v(\overline{c_v})]\!]_u + [\![M_v^{j_v}]\!]_u$ to v (the sum is over $\mathbb{F}_q$), and v verifies that it received the same value from all nodes that sent it.

Distributing the shares more efficiently. In the scheme we described above, the parent v was given shares $S_{u_i}(a), Y_{u_i}(a)$ and one-time pads $\left\{M_{u_i}^j\right\}_{j=1}^t$ for each child u_i. This requires $O(\Delta \log n)$ bits from the prover. We can distribute the shares more efficiently, and decrease the proof length to $O(\log n)$, by giving $S_{u_i}(a), Y_{u_i}(a), \left\{M_{u_i}^j\right\}_{j=1}^t$ to a *sibling* of u_i instead of to the parent v, assuming u_i has a sibling; if it does not, then v has only one child, so giving v the shares for its children requires $O(\log n)$ bits. The sibling then sends $S_{u_i}(a), Y_{u_i}(a), \left\{M_{u_i}^j\right\}_{j=1}^t$ up to the parent, and we proceed with the verification. Essentially, this shifts the communication burden from the prover to the edges of the tree.

4.2 Detailed protocol and correctness proof.

The details for our protocol are given below. All arithmetic is carried out in $\mathbb{F}_q$. We use $\mathit{Children}(v) = \{u \in N(v) : p_u = v\}$ to denote v's children in the input tree.

Algorithm 1 Spanning Tree — Prover's Protocol

```
1: Prover: Sample $c \in_U \{1, 2\}$. Next, for each $v \in V$,

     • Calculate the size $s_v$ of $v$'s subtree (including $v$ itself) and its depth $depth_v$ (i.e., its distance from the
       root of the tree). Let $c_v = (depth_v + c) \bmod 2 + 1$ be the point at which $v$ will receive polynomial
       evaluations.

     • Sample $S_v \in \mathbb{F}^1_{n+1}[x]$, a uniformly random polynomial of degree 1 over $\mathbb{F}_{n+1}$, subject to $S_v(0) = s_v$.

     • Generate $X_v = \left\{X^i_v\right\}^t_{i=1} \subseteq \mathbb{F}^1_q[x]$, and $Y_{v,u} = \left\{Y^i_{v,u}\right\}^t_{i=1} \subseteq \mathbb{F}^1_q[x]$ where each tuple $(X^i_v, Y^i_{v,u_1}, \ldots, Y^i_{v,u_d})$
       is a tuple of uniformly random degree-1 polynomials subject to $X^i_v(0) = \sum_{u \in Children(v)} Y^i_{v,u}(0)$.

     • Generate $\left\{M^j_v\right\}^t_{j=1} \subseteq \mathbb{F}_q$, iid uniformly random values.

2: Prover → $v$:
   Denote by $p$ the parent of $v$ in the tree (if $v$ is not the root). The prover sends to $v$:

     • $c_v$,
     • $S_v(c_v)$ and, if $v$ is not the root, also $S_p(c_v)$,
     • $X_v(c_v) = \left\{X^i_v(c_v)\right\}^t_{i=1}$, and if $v$ is not the root, also $X_p = \left\{X^i_p(c_v)\right\}^t_{i=1}$.

   If $v$ is not the root, and $v$ has at least one sibling: let $v_0 < \ldots < v_{d-1}$ be the children of $p$ (ordered by their IDs), and
   let $v = v_i$. The prover also gives $v$:

     • The name $w_v = v_{(i+1) \bmod d}$ of the next sibling after $v$,
     • $S_{w_v}(c_p)$, $\left\{Y^j_{p,w_v}(c_p)\right\}^t_{j=1}$, $\left\{M^j_{w_v}\right\}^t_{j=1}$. Here, $c_p = \overline{c_v}$ is the color of $v$'s parent in the tree (in the previously
       computing 2-coloring).

   Regardless of whether $v$ is the root, if $v$ has only one child $u$, the prover also gives $v$:

     • $S_u(c_v)$, $\left\{Y^j_{v,u}(c_v)\right\}^t_{j=1}$, $\left\{M^j_u\right\}^t_{j=1}$.

   If $v$ is either the root ($s_v = n$) or a leaf ($s_v = 1$), the prover also gives $v$:

     • $S_v(\overline{c_v})$.
```

Algorithm 2 Spanning Tree — Verifiers' Protocol: Checking the Tree Structure and Promises

```
1: Verify the 2-coloring: send the color $c_v \in \{1,2\}$ to $v$'s parent and children (if any), and verify that the 2-coloring is proper (i.e., the colors received from the parent and/or children are not $c_v$).
2: Check whether $v$ is a root, whether $v$ has children, and whether $v$ has any siblings, using two rounds of communication: each node $v$ sends $p_v$ to node $p_v$ (unless $p_v = \perp$, in which case $v$ is a root), and the parent $p_v$ sends back the number of neighbors $u \in N(p_v)$ that have $p_u = p_v$.
   Let $d$ be the degree of $v$, and let $u_0, \dots, u_{d-1}$ be the children of $v$, in order of their IDs.
3: Verify the ordering of siblings: if $v$ has children, it sends to each child $u_i$ the name of its next sibling, $u_{(i+1) \bmod d}$. Each child verifies that this matches the value $w_{u_i}$ that it was told by the prover (if it has siblings).
4: Check the promise for $X_v$, $Y_v$ and $M_v$:
   • Node $v$ chooses a uniformly random number $z_v \in_U [t]$, and sends it to its children and parent (if any).
   • Upon receiving $z_p$ from $v$'s parent $p$ (if any), node $v$ sends its parent the values: $\{X_p^j(c_v)\}_{j \neq z_p}$, $\{Y_{p,v}^j(c_v)\}_{j \neq z_p}$, $\{M_p^j\}_{j \neq z_p}$.
   • Upon receiving $z_u$ from a child $u$ (if any), node $v$ sends $u$ the values $\{M_u^j\}_{j \neq z_u}$: If $v$ has only one child, then it was given this value by the prover. If $v$ has more than one child, then $\{M_u^j\}_{j \neq z_u}$ was given to the next child $u'$ after $u$ in order of IDs; node $v$ sends $z_u$ to $u'$, receives back $\{M_u^j\}_{j \neq z_u}$, and forwards it to $u$.
   • If $v$ has children, it now has, for each child $u_i$ and $j \neq z_p$, the values:
      – $X_v^j(c_{u_i})$ (sent by $u_i$),
      – $X_v^j(c_v)$ (provided by the prover),
      – $Y_{v,u_i}^j(c_v)$ (provided by the prover if $u_i$ is an only child, or sent by $u_{(i-1) \bmod d}$ if $u_i$ has siblings),
      – $Y_{v,u_i}^j(c_{u_i})$ (sent by $u_i$).
     Note that $c_{u_i} = \overline{c_p}$ for each child $u_i$ (a 2-coloring). Node $v$ now interpolates:
      – The lines $\{X_v^j\}_{j \neq z_v}$,
      – The lines $\{Y_{v,u_i}\}_{j \neq z_v, i=0,\dots,d-1}$,
     and verifies that $X_v^j(0) = \sum_{i=0}^{d-1} Y_{p,u_i}(0)$ for each $j \neq z_v$.
   • If $v$ is an inner node (not a leaf or a root), it verifies that its parent and children sent the same values for $\{M_v^j\}_{j \neq z_v}$.
5: Verify share equality across layers:
   • For each $u \in N(v)$ that is the parent or a child of $v$, node $v$ sends $M_u^{z_u} + S_u(c_v)$ to $u$: if $u$ is the parent of $v$, or if $u$ is a single child of $v$, then $v$ was given $M_u^{z_u}, S_u(c_v)$ by the prover; otherwise, the next sibling $u'$ after $u$ sends $M_u^{z_u} + S_u(c_v)$ up to $v$, and $v$ forwards the value to $u$.
   • If $v$ is an inner node (not a leaf or the root), it verifies that its parent and children sent the same values for $M_v^{z_v} + S_v(\overline{c_v})$.
   • If $v$ is a leaf or a root, it sends $S_v(\overline{c_v})$, which it was given by the prover, to its parent or children (respectively), who were also supposed to be given this value. The receiving nodes verify that they received the same value from the prover.
```

Algorithm 3 Spanning Tree — Verifiers' Protocol: Checking the Consistency Condition

```
1: Verify the consistency condition over the subtree sizes:
    • If $v$ has a parent $p$, it sends it: $S_p(c_v) + X_p^{z_p}(c_v)$, and $S_v(c_v) + Y_{p,v}^{z_p}(c_v)$.
    • If $v$ has siblings, it sends to its parent $p$ the share $S_{w_v}(c_p)$, which it was given by the prover.
    • If $v$ has children $u_0 < \ldots < u_{d-1}$, it now has for each child $u_i$ the shares:
        – $S_v(c_{u_i}) + X_v^{z_v}(c_{u_i})$ (sent up by $u_i$),
        – $S_v(c_v) + X_v^{z_v}(c_v)$ (computed from the values $S_v(c_v), X_v^{z_v}(c_v)$ that $v$ was given by the prover).
      It interpolates to find the line $S_v + X_v^{z_v}$. In addition, node $v$ has the shares:
        – $S_{u_i}(c_{u_i}) + Y_{v,u_i}^{z_v}(c_{u_i})$ (sent up by $u_i$),
        – $S_{u_i}(c_v) + Y_{v,u_i}^{z_v}(c_v)$ (computed from the values $S_{u_i}(c_v), Y_{v,u_i}^{z_v}(c_v)$, either sent up by a sibling of $u_i$ or given
          to $v$ directly if it has only one child).
      It interpolates to find the line $S_{u_i} + Y_{v,u_i}^{z_v}$.
      Node $v$ evaluates and verifies that
          $$\left[S_v + X_v^{z_p}\right](0) = 1 + \sum_{i=0}^{d-1} \left[S_{u_i} + Y_{v,u_i}^{z_p}\right](0).$$
    • If $v$ is a root, the prover also gave it the share $S_v(\overline{c_v})$, in addition to $S_v(c_v)$. Node $v$ interpolates to find the line
      $S_v$, and verifies that $S_v(0) = n$.
    • Similarly, if $v$ is a leaf, it was also given the share $S_v(\overline{c_v})$. It now interpolates to find the line $S_v$ and verifies
      that $S_v(0) = 1$.
```

Completeness. In the case of an honest prover, all verification checks succeed. In particular, for each node v the consistency condition is satisfied, and the promise $X_v^{z_v}(0) = \sum_{u \in Children(v)} Y_{v,u}^{z_v}(0)$ is satisfied. Therefore, $S_v(0) + X_v^{z_v}(0) = 1 + \sum_{u \in Children(v)} S_u(0) + Y_{v,u}^{z_v}(0)$, as required. Finally, leafs v each have $S_v(0) = 1$ and the root r has $S_r(0) = n$, as required.

Soundness. Consider $(G, I) \notin \mathcal{L}$. If the prover provides an incorrect 2-coloring of the subgraph induced by the pointers in I, or if it lies about the local structure — whether a given node has children, a parent, or siblings — it will be caught in the first steps of the verification.

If there is some node v and index i such that $X_v^i(0) \neq \sum_{u \in Children(v)} Y_{v,u}^i(0)$, then the verifiers will catch this with probability $1 - 1/t$, where t is a soundness parameter (a sufficiently large constant). Similarly, if the prover provides $\left[\!\left[\left\{M_v^j\right\}_{j=1}^t\right]\!\right]_u \neq \left[\!\left[\left\{M_v^j\right\}_{j=1}^t\right]\!\right]_v$ to some nodes u, v at distance 2 in the tree, then it will be caught with probability $1 - 1/t$ when the node w that neighbors both u, v in the tree requests $\left\{M_v^j\right\}_{j \neq z_w}$. Let us condition on the event that the prover was not able to cheat in this manner without being caught. This implies that all objects satisfy their promise, and moreover, for each share $S_v(\overline{c_v})$ that is given to more than one node, the prover gave the same value to all nodes that receive the share — to v's parent and children if v is an inner node, and to v's parent or children if v is a leaf or a root, respectively. Otherwise the prover would be caught:

- If v is an inner node, the prover would be caught by v upon receiving different values for $M_v^{z_v} + S_v(\overline{c_v})$ from its children and parent.
- If v is a leaf or a root, the prover would be caught by v's parent or children (respectively) after v sends them $[\![S_v(\overline{c_v})]\!]_v$ and they compare it with the value they received from the prover.

In the sequel, we simply use "$S_v(a)$" to denote the single value $[\![S_v(a)]\!]_u$ for all nodes $u \in V$ (since, as we just said, it is the same value for all nodes).

For each node v, the prover provides shares $S_v(c_v)$ (to v itself) and $S_v(\overline{c_v})$ (to v's parent or sibling, and also to v itself if it is a leaf or the root). This uniquely determines a line $S_v \in \mathbb{F}_q^1[x]$.

To rule out a prover that exploits the modular arithmetic over $\mathbb{F}_q$ to cheat, define "true subtree sizes" $\{s'_v\}_{v \in V} \subseteq \mathbb{N}$ inductively over the height of v, with $s'_v = 1$ if v is a leaf and $s'_v = 1 + \sum_{u \in Children(v)} s'_u$ if v is not a leaf. We prove by induction on the height of v that $s'_v = S_v(0)$: if v is a leaf, then $S_v(0) = 1 = s'_v$, otherwise v rejects, as v is able to reconstruct S_v by itself and verify that $S_v(0) = 1$. Now let v be a node that is not a leaf, and suppose that for each child $u \in \mathit{Children}(v)$ we have $s'_u = S_u(0)$. In particular, $S_u(0) < n$, because s'_u is the true size of u's subtree, and the network contains n nodes in total.

Because node v did not reject,

$$S_v(0) + X_v^{z_v}(0) \neq 1 + \sum_{u \in \mathit{Children}(v)} \left(S_u(0) + Y_{v,u}^{z_v}(0)\right) \bmod q,$$

and because $X_v, \{Y_{v,u}\}_{u \in \mathit{Children}(v)}$ satisfy $X_v^{z_v}(0) = \sum_{u\,\mathit{Children}(v)} Y_{v,u}^{z_v}(0) \bmod q$, this implies that

$$S_v(0) = 1 + \sum_{u \in \mathit{Children}(v)} S_u(0) \bmod q.$$

Since $q > n^2$, $|\mathit{Children}(v)| < n$, and $S_u(0) = s'_u \leq n$ for each child u, over the natural numbers we have

$$S_v(0) = 1 + \sum_{u \in \mathit{Children}(v)} s'_u.$$

Thus, by definition of s'_v we have $s'_v = S_v(0)$.

We can now conclude that the input $\{p_v\}_{v \in V}$ indeed forms a spanning tree: first, since for every node v with $p_v = u \neq \perp$ we have $S_v(0) = s'_v < s'_u = S_u(0)$, the input induces a forest (there cannot be cycles). Thus, there is some node $r \in V$ that has $p_r = \perp$. Node r is given both $S_r(1), S_r(2)$ by the prover, and it verifies that $S_r(0) = n$. Since $S_r(0) = s'_r$, we see that the forest is spanning, i.e., it is a spanning tree.

$\mathcal{CONGEST}(O(1), O(\log n))$-dZK against a single adversary. Finally, we prove that our protocol is $(\mathcal{C}(O(1), O(\log n)), 1)$-dZK, by defining a simulator in the class $\mathcal{CONGEST}$ that runs in $O(1)$ rounds and sends $O(\log n)$-bit messages, and proving that for any single node, the simulator produces a view that is identical to the real view. Our goal is to "fill in" all the values received from either the prover of neighboring nodes, so that the distribution of the resulting view is identical to the real proof.

The simulator at each node v begins by sending v's parent to all neighbors. Following this step, v knows the names of its children (i.e., the neighbors that have v as their parent). It orders the children according to their IDs, $u_1 < \ldots, u_d$, and sends to each child u_i the name of the next sibling, $u_{(i+1) \bmod d}$, or informs u_i that it is an only child. This provides each node with the local structural information that the prover is supposed to give it. Note that this information completely determines the communication *pattern* of the verifier — which messages will be sent, and when. However, the *values* sent still need to be "filled in" by the simulator.

The remainder of the simulation is carried out locally, without communication; The simulator flips a fair coin in $\{1, 2\}$ and sets c_v to the result. It uses $\overline{c_v}$ for the colors that are supposed to be sent by all the neighbors to node v.

The simulator prepares the following random degree-1 polynomials, and "fills in" the following values the prover is supposed to send:

- If node v is not the root or a leaf: $S_v \in_U \mathbb{F}_{n+1}^1[x]$. Node v receives from the "prover" $S_v(c_v)$.
- If node v is the root: S_v is a random polynomial in $\mathbb{F}_{n+1}^1[x]$, subject to $S_v[0] = n$. Node v receives from the "prover" both $S_v(1)$ and $S_v(2)$.
- If node v is a leaf: S_v is a random polynomial in $\mathbb{F}_{n+1}^1[x]$, subject to $S_v[0] = 1$. Node v receives from the "prover" both $S_v(1)$ and $S_v(2)$.
- $X_v^j \in_U \mathbb{F}_q^1[x]$, for each $j = 1, \ldots, t$. Node v receives from the "prover" $X_v^j(c_v)$ for each j.
- If v has a parent p: $S_p \in_U \mathbb{F}_{n+1}^1[x]$. Node v receives from the "prover" $S_p(c_v)$.
- If v has a parent p: $X_p^j \in_U F_q^1[x]$ for each $j = 1, \ldots, n$. Node v receives from the "prover" $X_p^j(c_v)$ for each j.
- If v has a sibling, and v's next sibling is $w_v = u$: $S_u \in_U \mathbb{F}_{n+1}^1[x]$ and $Y_{p,u}^j \in_U \mathbb{F}_q^1[x]$ for $j = 1, \ldots, t$.

- If v has children, the simulator samples (but does not use yet):
  - $\left\{Y_{v,u}^j \in \mathbb{F}_q^1[x]\right\}_{u\in Children(v), j=1,\dots,t} \subseteq \mathbb{F}_q^1[x]$, iid uniform subject to the constraint that
$$\sum_{u\in Children(v)} Y_{v,u}^j(0) = X_v^j(0), \tag{4.2}$$
    for each j.
  - $\{S_u\}_{u\in Children(v)} \subseteq \mathbb{F}_q^1[x]$, iid uniform subject to the constraint that
$$1 + \sum_{u\in Children(v)} \left(S_u(0) + Y_{v,u}^j(0)\right) = S_v(0) + X_v^j(0), \tag{4.3}$$
    for each $j = 1, \dots, t$.
- If v has a single child u: node v receives from the "prover" $S_u(c_v)$ and $\left\{Y_{v,u}^j(c_v)\right\}_{j=1}^t$.
- Finally, the simulator samples "one-time pads" $\left\{M_u^j\right\}_{j=1}^t$ for each node u that is either v itself, or a parent or child of v. It gives $\left\{M_v^j\right\}_{j=1}^t$ to v, and if u is a single child of v, or if u is the next sibling after v, it also gives $\left\{M_u^j\right\}_{j=1}^t$ to v.

Next, we simulate the verification part of the protocol, without communication, by "filling in" values as sampled above. Observe that above, the distribution of every value the simulator plugs in to replace communication from the prover is the same as the distribution of the corresponding value sent by the prover in the real proof: these are all single shares, which are independent and uniform from their respective fields (except in the case of the root and of leafs, where we give both shares). The values *not* received from the prover (e.g., $S_v(\overline{c_v})$ when v is not the root or a leaf) may not be distributed correctly.

Crucially, the following distributions are identical to the prover's real distribution:

- The joint distribution of $\left\{X_v^j, \{Y_{v,u}^j\}_{u\in Children(v)}\right\}_{j=1}^t$ (iid uniform subject to (4.2)),
- For each $j = 1, \dots, t$, the joint distribution of $1 + \sum_{u\in Children(v)} \left(S_u + Y_{v,j}^j\right)$ and $S_v + X_v^j$ (uniform subject to (4.3)).

The *joint distribution of all these polynomials together* is incorrect, as it depends on the true subtree size s_v, but this is not exposed by the simulator's output: for each $j = 1, \dots, t$, the simulator exposes *either* the polynomials $X_v^j, \{Y_{v,u}^j\}_{u\in Children(v)}$, *or* the polynomials $1 + \sum_{u\in Children(v)} \left(S_u + Y_{v,j}^j\right)$ and $S_v + X_v^j$, but never both for the same value of j.

From the simulator's output, only the following polynomials can be reconstructed:

- $\left\{X_v^j, \{Y_{v,u}^j\}_{u\in Children(v)}\right\}_{j\neq z_v}$, and
- $X_v^{z_v}, \{Y_{v,u}^{z_v}\}_{u\in Children(v)}$.

The joint distribution of these polynomials is identical to the prover's real distribution, as they are independent of one another and each is uniform subject to (4.2) or (4.3), respectively — both in the simulator's distribution and in the prover's real distribution. All the rest of the values output by the simulator are single shares of some polynomial for which the simulator does not output the other share, or one-time pads that are revealed to verify the promise. These are each uniformly random, even conditioned on all the other values output by the simulator, matching their distribution in the real proof.

5 A Generic Compiler

In this section we describe a compiler that takes a proof labeling scheme (PLS) for some graph language L and a coalition size $k \geq 1$, and constructs an efficient dZK proof for L.

In a PLS for a language L, the prover assigns to each node v a label $a_v \in \{0,1\}^{\ell_{PLS}}$ (where ℓ_{PLS} is the *length* of the PLS), and the nodes send their labels to their neighbors. Then, each node v applies a circuit D_v to its input, its label, and its neighbors' labels, to decide whether to accept or reject. As usual, the input is considered *accepted* iff all nodes accept.

Our compiler takes a PLS of length ℓ_{PLS} and circuits of size μ, and constructs a distributed interactive proof for L that uses one round with the prover, has a locality radius of $O(k)$ for both the verifier and the simulator, and uses messages of $O\left(k^2\left(\Delta+\mu\right)+\log n\right)$ bits.

Intuitively, similarly to Section 4, we use a form of secret-sharing to divide the label of each node u into more than k shares. We give one share each to node u and to k nodes in u's vicinity, called *u's helper nodes.* Then, we run a verification protocol over the shares, to check that every node would accept under the original PLS, without ever reconstructing the labels themselves. The verification protocol relies on *full-linear PCPs* [8], a special type of probabilistically-checkable proof where the verifier accesses the proof *and the input* only through linear queries. As shown in [8], a fully-linear PCP can be verified even when the input is spread across many parties using a linear secret-sharing scheme, rather than stored in one location as in a typical PCP.

5.1 Assigning helper nodes. Suppose that the prover in the PCP we wish to compile produced the labels $\{a_v\}_{v\in V}$. For each node v, we choose k *helper nodes*, and split v's label a_v, as well as a proof that v's circuit D_v should accept, between v and its helper nodes ($k+1$ shares). The helper nodes of v and its neighbors must then communicate in order to verify that v would accept the labels $\{a_v\} \cup \{a_u\}_{u\in N(v)}$ in the original PLS (i.e., that the circuit D_v would output "accept" when given v's input, and the labels of v and its neighbors).

In order for the verifier produced by our compiler to be efficient, the helper nodes of v should not be too far from v, and communicating with them should not cause too much congestion in the network. These requirements are captured by the following definition:

Definition 5.1. ((k,d,c,h)-helper assignment) *Given $k,d,c,h \in \mathbb{N}$, a* (k,d,c,h)-helper assignment *is a collection $\{(H_v,P_v)\}_{v\in V}$, where for each $v\in V$,*

- *$H_v \subseteq V$ is a set of exactly k nodes, which we refer to as v's* helper nodes*.*
- *$P_v \subseteq (V^*)^k$ is a set of k paths, leading from v to each of its helper nodes.*

We require:

- *Bounded distance: for every $v\in V$, each path $\pi \in P_v$ is of length at most d.*
- *Bounded congestion: for each edge $e\in E$, the total number of times that e occurs in all the paths in $\{P_v\}_{v\in V}$ is at most c.*
- *Bounded help: no node $v \in V$ is assigned as a helper node to more than h nodes, that is, $|\{u\in V : v \in H_u\}| \leq h$ for every $v\in V$.*

In our compiler, the prover constructs a helper assignment $\{(H_v,P_v)\}_{v\in V}$, and provides each node v with (H_v,P_v). It is important that such an assignment can be computed efficiently: even though we are not concerned with the prover's computational resources, the helper assignment itself may disclose information about the network; to bound the quality and quantity of information, we must show that we can compute the helper assignment via an efficient distributed algorithm, so that the nodes can efficiently *simulate* their interaction with the prover. In Section 6 we show that a $(k,O(k),O(k^2),O(k))$-helper assignment can be computed efficiently in $\mathcal{CONGEST}$; for the remainder of the current section, we assume that we have an algorithm $\mathcal{A}_H$ that computes such an assignment.

For each node v, we order the helper nodes H_v by their IDs; if $H_v = \{u_1,\ldots,u_k\}$ such that $u_1 < \ldots < u_k$, then we denote v's helpers by $\{\hat{v}_i\}_{i=1}^k$, where $\hat{v}_1 = u_1,\ldots,\hat{v}_k = u_k$. For convenience, we denote $\hat{v}_0 = v$.

5.2 Fully-linear PCPs. In the protocol produced by our compiler, the prover must convince each node v that the circuit D_v from the original PLS would output "accept" when given v's input, the label of v, and the labels of v's neighbors. However, the prover must do this without revealing any information about v's inputs or the labels. To this end, we ask the prover to provide node v and its helpers with a *fully-linear PCP* [8], a type of proof that can be verified even when the input is split into many shares, without disclosing any information.

Fully-linear probabilistically checkable proofs [8]. Let $\mathbb{F}$ be a finite field, let $L \in \mathbb{F}^n$ be a language, and let $R \subseteq \mathbb{F}^n \times \mathbb{F}^h$ be a binary relation, such that $x \in \mathbb{F}^n$ iff there exists a *witness*, $w \in \mathbb{F}^h$, such that $(x, w) \in R$.

A *fully-linear probabilistically checkable proof system* (FLPCP) for R over $\mathbb{F}$ with proof length m, soundness error ϵ, and query complexity ℓ is a pair of algorithms (P_{LPCP}, V_{LPCP}) (the *prover* and the *verifier*), with the following properties:

- For every $(x, w) \in R$, the prover $P_{LPCP}(x, w)$ outputs a proof $\pi \in \mathbb{F}^m$.
- The verifier V_{LPCP} consists of a query algorithm Q_{LPCP} and a decision algorithm D_{LPCP}.
  - The query algorithm Q_{LPCP} takes no input and outputs ℓ queries $q_1, \ldots, q_\ell \in \mathbb{F}^m$, which are independent of x, as well as state information $st \in \{0,1\}^*$.
  - The decision algorithm D_{LPCP} takes as input the state st, and the ℓ answers $\langle (x \parallel \pi), q_1 \rangle, \ldots, \langle (x \parallel \pi), q_\ell \rangle \in \mathbb{F}$ to Q_{LPCP}'s queries. (Here, $\parallel$ denotes concatenation, and $\langle \cdot, \cdot \rangle$ denotes the inner product.) It outputs "accept" or "reject."

We require the following properties:

- Completeness: for all $(x, w) \in R$, the verifier always accepts a valid proof:

$$Pr\left[D_{LPCP}(st, \langle (x \parallel \pi), q_1 \rangle, \ldots, \langle (x \parallel \pi), q_\ell \rangle) = accept : \begin{array}{c} \pi \ \leftarrow P_{LPCP}(x, w) \\ (st, q_1, ..., q_\ell) \leftarrow Q_{LPCP}() \end{array}\right] = 1.$$

- Soundness: for all $x^* \notin L$, and for all "false proofs" $\pi^* \in \mathbb{F}^m$, the probability that the verifier accepts is at most ϵ:

$$Pr\left[D_{LPCP}(st, \langle (x \parallel \pi^*), q_1 \rangle, \ldots, \langle (x \parallel \pi^*), q_\ell \rangle) = accept : (st, q_1, ..., q_\ell) \leftarrow Q_{LPCP}()\right] \leq \epsilon.$$

- Strong honest-verifier zero knowledge (strong HVZK): there exists a simulator S_{LPCP} such that for all $(x, w) \in R$, the following distributions are identical:

$$S_{LPCP}() \equiv \left\{ \begin{array}{c} (q_1, \ldots, q_\ell) \\ \langle (x \parallel \pi), q_1 \rangle, \ldots, \langle (x \parallel \pi), q_\ell \rangle \end{array} : \begin{array}{c} \pi \leftarrow P_{LPCP}(x, w) \\ (q_1, \ldots, q_l) \leftarrow Q_{LPCP}() \end{array} \right\}.$$

  That is, the simulator is able to sample, with no input, from the true distribution of the queries and answers that would be observed by the verifier when interacting with the prover.

As shown in [8], FLPCPs can be used when the input (x) is shared among many parties using linear secret-sharing; since the FLPCP uses linear queries, we can implement the queries by having each party apply them to its own share of the input, and then adding up the results.

Here we will use a Hadamard linear PCP [1, 7], which was shown in [8] to be an FLPCP with the strong-HVZK property above. Given an arithmetic circuit C over the field $\mathbb{F}$ and an input $x \in \mathbb{F}$ to the circuit, there is a Hadamard linear PCP for the relation $\{(C, x) : C(x) = 0\}$ which has proof length $O(|C|^2)$, uses 3 queries, and has soundness error $\epsilon = O(1)/|\mathbb{F}|$.[10][11]

[10]The size $|C|$ of an arithmetic circuit C is the number of multiplication gates in the circuit.

[11]Strictly speaking, the Hadamard PCP we use is for the *satisfiability problem* of arithmetic circuits, while here we want to verify *satisfaction*, $C(x) = 0$. However, it is easy to take a given circuit C and an input x, and produce a new circuit C_x of roughly the same size, which is satisfiable iff $C(x) = 0$.

5.3 From PLS to fully-linear PCP. Next we explain how we instantiate the FLPCP construction from [8] in our compiler. Suppose that under the PLS we wish to compile, each node v receives a label $a_v \in \mathbb{F}^{\ell_{PLS}}$.[12] Let $A_v = a_v, a_{u_1}, \ldots, a_{u_d}$, where each u_i is the neighbor of v that v communicates with through the port numbered i. We view A_v as an element of $\mathbb{F}^{\ell_{PLS}\cdot(\deg(v)+1)}$.

Under the PLS, node v decides whether to accept or reject by applying an arithmetic circuit $D_v^{\deg(v)}(I(v), A_v)$. We assume that an output of 0 corresponds to "accept". Given $d \in [n]$ and $I \in \mathcal{X}$, we let $D_v^{d,I}$ denote the circuit obtained by hard-wiring the value $I(v) = I$ into D_v^d. In other words, when node v's input is I and its degree is $\deg(v) = d$, the decision output by node v under the PLS is $D_v^{d,I}(A_v) = D_v^d(I, A_v)$.

For each $v \in V$, degree d, and input I, let $\Pi_v^{d,I}$ be an FLPCP (satisfying the Strong-HVZK property, as explained above) for the language

$$L_v^{d,I} = \left\{ A_v \in \mathbb{F}^{\ell_{PLS}\cdot(d+1)} : D_v^{d,I}(A_v) = 0 \right\}.$$

Let $\ell_{FLPCP}(v, d, I)$ be the proof length of the FLPCP. We also use ℓ_{FLPCP} to denote the maximum length of an FLPCP used by any node in the annotated graph, that is, $\ell_{FLPCP} = \max_{v\in V} \ell_{FLPCP}(v, \deg(v), I(v))$.

5.4 Our protocol. In the protocol produced by our compiler, the honest prover begins by constructing a $(k, O(k), O(k^2), O(k))$-helper assignment, using the deterministic algorithm $\mathcal{A}_H$ given in Section 6. It then gives each node v its helper nodes H_v and the paths to them, P_v. The prover also informs each node v of all nodes that v needs to help, $u \in V$ such that $v \in H_u$, and provides some auxiliary information needed to route messages that may pass through v. Throughout the protocol, whenever node v and a helper node $u \in H_v$ need to communicate, they do so by routing a message over the corresponding path in P_v.

Next, the prover computes:

- The labels $\{a_v\}_{v\in V}$ of the original prover from the PLS,
- For each $v \in V$, a proof π_v produced using the fully-linear PCP $\Pi_v^{\deg(v),I(v)}$, proving that $D_v^{\deg(v),I(v)}(A_v) = 0$.

The prover divides a_v and π_v into $k+1$ shares each, $\{a_v^i\}_{i=0}^k$, $\{\pi_v^i\}_{i=0}^k$, using a bitwise secret-sharing scheme: $a_v^1, \ldots, a_v^{k+1} \in \mathbb{F}^{\ell_{PLS}}$ are uniformly random subject to $\sum_{i=0}^k a_v^i = a_v$, where the sum is taken over $\mathbb{F}^{\ell_{PLS}}$, and similarly for π_v (over $\mathbb{F}^{\ell_{FLPCP}(v,\deg(v),I(v))}$).

For each $v \in V$ and $i \in \{0, \ldots, k\}$, the prover gives a_v^i and π_v^i to node $\hat{v}_i$.

With the prover's assistance, the nodes cooperate to verify the FLPCP, in order to ensure that each node would accept under the original PLS. The FLPCP π_v corresponding to node v is verified by node v and its helpers; however, since node v's decision in the PLS can depend on its neighbors' labels, node v and its helpers must first collect shares of the labels of $N(v)$. The verification thus involves the following high-level steps, carried out in parallel for all nodes $v \in V$:

(1) For each $u \in N(v)$, node u and its helpers send shares of a_u to node v and its helpers, with each $\hat{u}_i$ sending one share of a_u to $\hat{v}_i$.

Here it is important not to re-use the same shares $a_u^0, \ldots, a_u^k$ of a_u originally provided by the prover: for example, if some node w serves as a helper node for $k+1$ neighbors $z_1, \ldots, z_{k+1}$ of u, and if w is the i-th helper for z_i ($w = \hat{(z_i)}_i$), then w will receive $k+1$ different shares of a_u, potentially exposing a_u. Thus, node u and its helpers prepare a scrambled version $\tilde{a}_u^{0,v}, \ldots, \tilde{a}_u^{k,v}$ especially for node v, and send it to node v and its helpers.

In addition, nodes never send shares "in the clear"; we use the prover to implement "secure channels" between each $\hat{u}_i$ and $\hat{v}_i$, so that intermediate nodes on the path between them learn nothing.

(2) Each helper $\hat{v}_i$ of v prepares a share A_v^i of A_v (the labels of v and its neighbors): if the neighbors of v are $u_1, \ldots, u_d$ (arranged according to the port numbering of v),

$$A_v^i = a_v^i, \tilde{a}_{u_1}^i, \ldots, \tilde{a}_{u_d}^i.$$

[12]Typically, in proof labeling schemes, we work with $\mathbb{F}$ (binary labels), but here we will work with some field $\mathbb{F}$ whose size depends on the soundness error we are aiming for.

(3) Node v samples three[13] random queries $q_{v,1}, q_{v,2}, q_{v,3} \in \mathbb{F}^{\deg(v)\cdot \ell_{PLS}+\ell_{FLPCP}(v,\deg(v),I(v))}$, and sends $q_{v,1}, q_{v,2}, q_{v,3}$ to its helper nodes $\hat{v}_1, \dots, \hat{v}_k$.

(4) Each helper node $\hat{v}_i$ computes three responses, $\left\{r^i_{v,j} = \langle q_{v,j}, A^i_v \parallel \pi^i_v \rangle\right\}_{j=1,2,3}$, and sends $r^i_{v,1}, r^i_{v,2}, r^i_{v,3}$ back to v.

(5) Finally, node v computes $r_{v,j} = \sum_{i=0}^{k} r^i_{v,j}$ for each $j = 1, 2, 3$ (where the sum is over $\mathbb{F}^{\deg(v)\cdot \ell_{PLS}+\ell_{FLPCP}(v,\deg(v),I(v))}$), and uses the decoder of the FLPCP $\Pi_v^{\deg(v),I(v)}$ to check whether the responses $r_{v,1}, r_{v,2}, r_{v,3}$ should be accepted; otherwise, it rejects.

We note that the queries and responses of the FLPCP can be sent "in the clear"; by themselves they convey no information, and as shown in [8], this is true even given a share of the input.

Next we detail how the nodes implement "secure channels" with the prover's help, and how nodes produce scrambled version of their shares.

Sending secret messages. Suppose that node u wishes to send a message $m \in_U \mathbb{F}^L$ to node v, using the path $u_0, \dots, u_m$, where $u_0 = u, u_m = v$. The intermediate nodes on the path are allowed to learn that a message is being sent, and they can learn the path from u to v, but not the message itself: from their perspective, m should remain uniformly distributed in $\mathbb{F}^L$.

To this end, the prover provides nodes u, v with a *message kit*, $\left\{M^i\right\}_{i=1}^{t} \in_U \mathbb{F}^L$, where t is a security parameter. Each M^i can essentially serve as a *one-time pad* to encrypt a message that will be sent from u to v. The prover is meant to give both nodes u, v the same values for $\left\{M^i\right\}_{i=1}^{t}$, but of course, we cannot trust that it does so. Therefore, we use the cut-and-choose technique: node u selects a random index $i^* \in [t]$, and sends to node v all the *other* values, $\left\{M^i\right\}_{i \neq i^*}$, over the path between the two nodes. Node v then makes sure that it received the same values as u from the prover, otherwise it rejects. If the prover cheated by sending a different value for at least one one-time pad M^i, there is probability at least $1 - 1/t$ that it will be caught.

Finally, node u encrypts its message using the one remaining one-time pad that was not revealed, M^{i^*}: node u sends $m + M^{i^*}$ (the sum, as usual, is over $\mathbb{F}^L$) to node v along the path between them. Node v, which knows M^{i^*}, is able to decrypt the message; the intervening nodes learn nothing — they know nothing about M^{i^*}, so to them, the message $m + M^{i^*}$ appears uniformly random.

Scrambling the shares. For each neighbor $u \in N(v)$, node v and its helpers prepare "a scrambled version" $\tilde{a}^{0,u}_v, \dots, \tilde{a}^{k,u}_v$ of the shares $a^0_v, \dots, a^k_v$ of a_v, as follows:

- Node v prepares $k+1$ shares, $o^0_{v,u}, \dots, o^k_{v,u}$, of the value $0^{\ell_{PLS}} \in \mathbb{F}^{\ell_{PLS}}$.
- For each $i \in \{1, \dots, k\}$, node v securely sends $o^i_{v,u}$ to $\hat{v}_i$, as described above.
- For each $i \in \{0, \dots, k\}$, node $\hat{v}_i$ sets $\tilde{a}^{i,u}_v = a^i_v + o^i_{v,u}$.

The scrambled version $\tilde{a}^{0,u}_v, \dots, \tilde{a}^{k,u}_v$ still satisfies

$$\sum_{i=0}^{k} \tilde{a}^{i,u}_v = a_v,$$

so it is a valid split of a_v into $k+1$ shares. However, now we do not need to worry about helper nodes collecting too many shares: for any collection of neighbors $u_1, \dots, u_d \in N(v)$ and for any indices $i_1, \dots, i_d \in \{0, \dots, k\}$, the joint distribution of $\left\{\tilde{a}^{i_j,u_j}_v\right\}_{j=1}^{d}$ is uniformly random and independent of a_v, even if $d \geq k+1$. Thus, even if a single helper node is assigned to help $d \geq k+1$ neighbors $u_1, \dots, u_d$ of v in different helper roles (indices), it will still not learn a_v.

[13]Recall that the FLPCP we work with, the Hadamard linear PCP, uses three queries.

5.5 Detailed description and analysis of the protocol. Below, we put all the pieces together and detail the protocol carried out by the prover and by the verifiers.

The protocol is detailed in Algorithms 4 and 5. All sums appearing in the protocol are over the respective fields of the summands. Whenever we say that a message is routed or sent from u to $\hat{u}_i$ or vice-versa, we mean that the message is sent on the corresponding path from $paths_u$ provided by the prover, with intermediate nodes on the path using the routing information provided by the prover to forward the message along the path.

Algorithm 4 General Compiler — Communicating with the Prover

1: **Prover:**

- Execute the algorithm $\mathcal{A}_H$ to compute a (k, d, c, h)-helper assignment, assigning to every node $v \in V$ a set $H_v \subseteq V$, together with paths $paths_v \subseteq (V^*)^k$, leading from v to each of its helper nodes.
- Compute the labels under the PLS. Denote v's label by $a_v \in \mathbb{F}^{\ell_{PLS}}$. Let $a_v^0, \ldots, a_v^k \in \mathbb{F}^{\ell_{PLS}}$ be uniformly random subject to $\sum_{i=0}^k a_v^i = a_v$.
- Compute the FLPCP π_v for the language $L_v^{\deg(v),I(v)}$, attesting that $D_v^{\deg(v),I(v)}(A_v) = 0$. Let $\pi_v^0, \ldots, \pi_v^k \in \mathbb{F}^{\ell_{FLPCP}(v,\deg(v),I(v))}$ be uniformly random such that $\sum_{i=0}^k \pi_v^i = \pi_v$.
- For each edge $\{u, v\} \in E$ and for each $i \in \{1, \ldots, k\}$, generate a secret-message kit of iid uniform values $\left\{M_{\{u,v\}}^{i,j}\right\}_{j=1}^t \subseteq \mathbb{F}^{\ell_{PLS}}$ that will be used to send $\tilde{a}_v^{i,u}$ from $\hat{v}_i$ to $\hat{u}_i$.
- For each $i \in \{1, \ldots, k\}$, generate a secret-message kit of iid uniform values $\left\{M_{helper}^{i,j}\right\}_{j=1}^t \subseteq \mathbb{F}^{\ell_{PLS}}$ that will be used to communicate with $\hat{v}_i$.

2: **Prover $\to$ v:**

- The set of v's helper nodes H_v along with paths $paths_v \subseteq (V^*)^k$ to them.
- Helping information: the set of nodes u such that $v \in H_u$, and for each such node u, the index i such that $v = \hat{u}_i$, and the IDs of u's neighbors, ordered by the port number through which u sends them messages.
- Message kits: for each u such that $v \in H_u$, and for each neighbor $w \in N(u)$, if $v = \hat{u}_i$, then v receives $\left\{M_{helper,u}^{i,j}\right\}_{j=1}^t$ and $\left\{M_{\{u,w\}}^{i,j}\right\}_{j=1}^t$.
- Routing information: for each node $u \in V$ and helper $u' \in H_u$ such that v appears on the path $p_{u,u'} \in paths_u$ from u to u', the IDs of the neighbors of v that appear before and after, respectively, it on the path $p_{u,u'}$.

Complexity. The protocol requires one round of interaction with the prover — in fact, it is a Merlin-Arthur protocol, where the prover sends a message, and then the verifiers carry out their verification with no further interaction with the prover.

To bound the size of the messages used in the verification, recall that in an (k, d, c, h)-helper assignment, each path is of length at most d, each node participates in at most c paths, and each node is assigned to help at most h other nodes. Describing the helper assignment therefore requires:

- $O(kd \log n)$ bits, to encode the helper nodes H_v and paths $paths_v$ of a given node v.
- $O(kh\Delta \log n)$ bits, to tell each node which nodes it must help, and the neighborhood of each node it must help.
- $O(c \log n)$ bits, to encode the routing information (for each path that a node participates in, the endpoints of the path, and the previous and next hops along the path).

Algorithm 5 General Compiler — Verification at Node v

- The nodes verify that the helper assignment and all the information associated with it are valid, by executing the algorithm $\mathcal{A}_H$ themselves and verifying that its output matches what the prover claimed. This includes checking the helping and routing information, which can be done by observing $\mathcal{A}_H$ as it runs (see Section 6).
- Verify the message kits: for each edge $\{u, w\}$ and $i \in \{1, \ldots, k\}$ such that $\hat{u}_i = v$, if $u < w$, choose a uniformly random $j^i_{\{u,w\}} \in \{1, \ldots, t\}$, and send $j^i_{\{u,w\}}$ and $\left\{M^{i,r}_{\{u,w\}}\right\}_{r \neq j^i_{u,w}}$ to $\hat{w}_i$. The message is first routed from v to u, then sent on the edge $\{u, w\}$, then routed from w to $\hat{w}_i$.

  Upon receiving a message $j^i_{\{u,w\}}, \left\{N^{i,r}_{\{u,w\}}\right\}_{r \neq j^i_{\{u,w\}}}$, node v verifies that it was given matching values by the prover, i.e., that $N^{i,r}_{\{u,w\}} = M^{i,r}_{\{u,w\}}$ for each $r \neq j^i_{\{u,w\}}$.

  Similarly, the message kit $\left\{M^{i,j}_{helper,u}\right\}_{j=1}^{t}$ is verified having v choose a random $j_{helper,u}$ and send $\left\{M^{i,j}_{helper,u}\right\}_{j \neq j_{helper,u}}$ to u, which then checks that it received the same value from the prover.
- Prepare scrambled shares: for each $u \in N(v)$, node v prepares $k + 1$ shares, $o^0_{v,u}, \ldots, o^k_{v,u}$, of the value $0^{\ell_{PLS}} \in \mathbb{F}^{\ell_{PLS}}$. It routes each $o^i_{v,u} + M^{i,j_{helper,v}}_{helper,v}$ to $\hat{v}_i$, and node $\hat{v}_i$ then sets $\tilde{a}^{i,u}_v = a^i_v + o^i_{v,u}$ (including $i = 0$, i.e., node v itself).
- Send the scrambled shares: for each edge $\{u, w\}$ and $i \in \{1, \ldots, k\}$ such that $\hat{u}_i = v$, node v routes $\tilde{a}^{i,w}_u$ to $\hat{w}_i$ using the same path as above.
- Node v samples random queries $q_{v,1}, q_{v,2}, q_{v,3} \in \mathbb{F}^{\deg(v)\cdot\ell_{PLS}+\ell_{FLPCP}(v,\deg(v),I(v))}$, and sends $q_{v,1}, q_{v,2}, q_{v,3}$ to nodes $\hat{v}_1, \ldots, \hat{v}_k$.
- Each helper node $\hat{v}_i$ computes three responses, $\left\{r^i_{v,j} = \langle q_{v,j}, A^i_v \parallel \pi^i_v\rangle\right\}_{j=1,2,3}$, and sends $r^i_{v,1}, r^i_{v,2}, r^i_{v,3}$ back to v.
- Finally, node v computes $r_{v,j} = \sum_{i=0}^{k} r^i_{v,j}$ for each $j = 1, 2, 3$ (where the sum is over $\mathbb{F}^{\deg(v)\cdot\ell_{PLS}+\ell_{FLPCP}(v,\deg(v),I(v))}$), and uses the decoder of the FLPCP $\Pi^{\deg(v),I(v)}_v$ to check whether the responses $r_{v,1}, r_{v,2}, r_{v,3}$ should be accepted; otherwise, it rejects.

Taking $d = O(k), c = O(k^2), h = O(k)$, we see that $O(k^2 \log n)$ bits suffice.

Next consider the shares sent and received by a node v, including those that v simply forwards along on the path between other nodes:

- "Encrypted" scrambled shares of the form $\tilde{a}^{i,w}_u + M^{i,j}_{\{u,w\}} \in \mathbb{F}^{\ell_{PLS}}$: there are at most $c = O(k^2)$ such shares that v needs to forward on behalf of other nodes, for a total of $O(k^2\ell_{PLS})$ bits (treating $|\mathbb{F}|$ as a constant). In addition, for each u that v helps, v needs to send one scrambled share for each neighbor of u, for a total of $O(k\Delta\ell_{PLS})$ bits.
- Queries and responses of the FLPCP: these are each of size $O(\Delta\ell_{PLS} + \ell_{FLPCP})$. Node v may need to send out or forward $c = O(k^2)$ of them, for a total of $O(k^2(\Delta\ell_{PLS} + \ell_{FLPCP}))$ bits.

Finally, "opening" all but one value in a secret-message kit requires messages of size $O(\ell_{PLS} \cdot t + \log n)$, and each node may need to send out or forward $c + \Delta h$ such messages. Since we take the security parameter t to be a constant, this is subsumed by the costs above.

Completeness. As usual, it is not difficult to go through the protocol and verify that the honest prover convinces all nodes to accept with probability 1; this follows from the completeness of the FLPCP.

Soundness. Following the successful local verification of the helper assignment, each node u is able to reliably route messages to and from the nodes in $H_u \cup \{v : u \in H_v\} \cup \{u' : u \in H_v, u' \in H_w, (v, w) \in E\}$. We take this as a given from now on.

The prover may try to cheat, and with some small probability it will not be caught. The following bad events each occur with probability at most $1/t$:

1. For at least one pair of nodes v, u that are meant to share a secret-message kit, the prover gives different values of the message kit to v and u: since v, u compare a random subset of $t-1$ values from the kit, if there is any value that differs, the prover will be caught with probability at least $1 - 1/t$.

2. Some node v and its helpers verify the FLPCP of node v using shares $\left\{A_v^i \parallel \pi_v^i\right\}_{i=0}^{k}$ such that $\sum_{i=0}^{k} \pi_v^i$ is not a valid proof for $\sum_{i=0}^{k} A_v^i \in L_v^{\deg(v),I(v)}$, but the verification succeeds (the verifier of the FLPCP returns "accept"). Since the soundness error of the FLPCP is $O(1)/|\mathbb{F}|$, by choosing $|\mathbb{F}| = \Omega(t)$ we can drive the probability of this event down to $1/t$.

If neither of these bad events occur, a graph that is not in the language will not be accepted: if $(G, I) \notin L$, then for any assignment of labels to the nodes, there is a node $v \in V$ such that $D_v^{\deg(v),I(v)}(A_v) \neq 0$, that is, node v rejects in the original PLS. The proof π_v provided by the prover is therefore invalid (the input is not in the language). Whenever the prover does not provide invalid message kits without being caught, the shares $\left\{A_v^i \parallel \pi_v^i\right\}_{i=0}^{k}$ on which node v and its helpers execute the FLPCP verification are indeed valid shares of $A_v \parallel \pi_v$ (all secret messages are conveyed successfully). Therefore, the verification at v will fail.

$\mathcal{CONGEST}(\tilde{O}(k), O(k^2(\Delta\ell_{PLS} + \ell_{FLPCP}) \log n))$-knowledge. The distributed simulator begins by computing the helper assignment, using $\mathcal{A}_H$. This requires $O(k \log k)$ rounds, and messages of size $O(k^2 \log n)$. Each node then sends its neighborhood to all its helper nodes. This requires $O(c\Delta \log n) = O(k^2 \Delta \log n)$ bits. From this point on, the communication *pattern* — which messages are sent in the protocol and at what time — is fixed, as it is determined solely by helper assignment, and specifically, by the paths on which the node appears and the neighborhood of the nodes that it must help; only the *values* of messages sent or received by the node remain to be filled in.

The simulator now samples all the values that are supposed to be provided to nodes by the prover, as follows: each node v produces $k+1$ shares $a_v^0, \ldots, a_v^k$ of the value $0 \in \mathbb{F}^{\ell_{PLS}}$, and $k+1$ shares $\pi_v^0, \ldots, \pi_v^k$ of the value $0 \in \mathbb{F}^{\ell_{FLPCP}}$. For each $i = 1, \ldots, k$, node v sends a_v^i, π_v^i to its i-th helper node $\hat{v}_i$. In addition, each message kit that is meant to be shared between two nodes u, v is sampled by the smaller of the two nodes and sent to the other.

Finally, the simulator executes the verification protocol, using the values sampled above, and outputs the resulting view. Note that not all values that a node gains access to during the simulator's run are exposed by the simulator's output: for example, the simulator may route all the shares $a_v^0, \ldots, a_v^k$ through the same node u, but u's simulator output includes at most one share a_v^i.

6 Computing an Assignment of Helper Nodes

In this section we show how to compute a $(k, O(k), O(k^2), O(k))$-helper assignment (see Definition 5.1) in $\tilde{O}(k)$ rounds, using messages of $O(k^2 \log n)$ bits. The construction has two parts:

(1) Using a variation on Controlled-GHS [16], we partition the graph into a collection of rooted trees, such that each rooted tree is of size at least $k+1$. We refer to these trees as *fragments*. We note that in contrast to the typical application of Controlled-GHS (which is usually used to compute minimum-weight spanning trees), here we impose no *upper bound* on the size or the diameter of the fragments we construct.

(2) In parallel, all nodes of the graph choose helper nodes and find paths to their helper nodes. Informally, the idea is as follows: suppose that node v belongs to some fragment F, and let s be the size of v's subtree in F (including v itself).

- If $s \geq k+1$, then v recruits k nodes from its subtree in F as its helper nodes, choosing nodes at distance at most k from itself.
- If $s < k+1$, then node v reaches upwards and asks for help from its lowest ancestor in F that has a subtree of size at least $k+1$: we know that such an ancestor exists, because F is of size at least $k+1$, and furthermore, the lowest ancestor is at distance at most k from v.

Nodes that received help requests handle them by first trying to "matchmake" among its descendants: if more than k descendants asked node u for help, node u partitions the requesting nodes into batches of size $\Theta(k)$, and assigns the nodes in each batch to help the nodes in their batch. If fewer than k descendants asked node u for help, node u simply uses its own helper nodes and assigns them to help all the requesting nodes as well. All in all, the load added to each helper node is at most $O(k)$.

We now describe in more detail how to construct the fragments, and how to assign helper nodes and paths once the fragments are computed.

6.1 Constructing a forest of fragments.

Throughout this section, fix a threshold L (which we will later set to $k + 1$). Our goal is to partition the graph into a collection of rooted trees, called *fragments*, each of size at least L.

The fragments may be very large, and they can have large diameter, because we only impose a *lower bound* on the fragment size. Thus, we cannot expect nodes to *know* which fragment they belong to; they may not be able to communicate across the entire fragment to agree on, e.g., a unique identifier for the fragment. Instead, all we require is that nodes should have a pointer to their parent in the fragment, and know whether or not their subtree is of size at least L. We call the resulting structure an *L-forest*:

Definition 6.1. (L-Forest) *An* L-forest *is an assignment $up : V \to V \cup \{\perp\}$ of* parent pointers*, such that the directed graph formed by the edges $\{(v, up(v)) : v \in V\}$ is a collection of rooted trees, each of size at least L.*

We now show how to compute an L-forest in $\mathcal{CONGEST}(O(L \log L), O(\log n))$, for any $L \leq n$. The forest is computed using a variation on Controlled-GHS [16], starting with each node in its own singleton fragment (with the *up*-pointer set to $\perp$), and merging fragments until no fragments of size less than L remain. A fragment of size $< L$ is called *small*, and a fragment of size $\geq L$ is called *large*.

During the process, we maintain the following invariants:

- The *up* pointers whose values are not $\perp$ induce a collection of rooted trees (fragments).
- Every node v stores a Boolean variable $small(v)$, which is 1 iff the size of v's current fragment is less than L.
- If v belongs to a small fragment ($small(v) = 1$), then v also stores the ID of the root of its fragment, which we denote by $leader(v)$. We also let $leader(F)$ denote the ID of the root of fragment F.

In the sequel, we often say that small fragments "send messages" or "receive messages", meaning that some fragment node sends or receives the message, and all fragment nodes know that the message was sent or received.

We construct the L-forest in $O(\log L)$ steps, where in each step,

(1) Each small fragment identifies a neighboring fragment that it would like to merge with, and sends it a *merge request*;

(2) Any small fragment F that sent a merge request to a fragment F' is merged into F'. The *up*-pointers are modified accordingly, and we update the local variables $small(v), leader(v)$ at all fragment nodes $v \in F$.

6.1.1 Merge requests.

At the beginning of each step, each small fragment F identifies a fragment $F' \neq F$ that it would like to merge with, as follows:

(a) **Primary merge requests:** each $v \in F$ searches for an outgoing edge $e_v = (v, u)$ where $u \notin F$, such that either

- u's fragment is large, or
- u's fragment is small, and $leader(v) < leader(u)$.

If more than one such edge exists, node v chooses e_v arbitrarily from among them. Next, the fragment nodes disseminate the set $\{e_v\}_{v \in F}$ to the entire fragment using pipelining, which requires $O(L)$ rounds.

Let $e_F = (v, u)$ be the smallest edge in $\{e_v\}_{v \in F}$, in lexicographic order. Node v sends a merge request to node u, asking to merge with u's fragment. This request is called a *primary merge request*.

(b) In each small fragment, the fragment nodes inform one another whether or not a primary merge request was received by any node in the fragment. This again requires $O(L)$ rounds (convergecast to the root, which then broadcasts the answer to all nodes). The fragment nodes then send the answer (i.e., whether or not a primary merge request was received) on all outgoing edges of the fragment.

(c) **Secondary merge requests:** now consider a small fragment F that did not send or receive a primary merge request. Then F has an outgoing edge (v, u), where $v \in F$ and $u \notin F$, such that u's fragment sent a primary merge request (see Observation 1 below). Fragment F chooses the lexicographically-smallest such edge (v, u), using pipelining. Node v now sends a merge request to node v along the edge (v, u), and this request is called a *secondary merge request.*

Before proceeding, let us make a few simple observations about the structure of the subgraph induced by the merge requests.

Observation 1. *If F is a small fragment that did not send or receive a primary merge request in some merge step, then there is some small fragment F' adjacent to F (i.e., there is an edge $\{v, u\}$ such that $v \in F$ and $u \in F'$) such that F' did send a primary merge request.*

Proof. Since F is a small fragment, its size is smaller than n, so there is some edge $\{v, u\}$ such that $v \in F$ and $u \notin F$. Let F' be the fragment to which u belongs. Since F was not able to send any primary merge requests, F' must be a small fragment, and we must have $\mathit{leader}(F') < \mathit{leader}(F)$ (otherwise F' would be a candidate to which F could send a primary merge request). But this means that F' itself *does* have at least one neighboring fragment it can send a primary merge request to — fragment F itself. Thus, F' sends some primary merge request. □

Observation 2. *Every small fragment is either the origin or the target of at least one (primary or secondary) merge request.*

Proof. From the previous observation, we see that every fragment that does not send or receive a primary merge request is able to send a secondary merge request. □

Define the *fragment graph* to be the directed graph whose nodes are the fragments, and containing all edges (F, F') such that F sent a merge request to F'.

Observation 3. *The fragment graph contains no directed or undirected cycles,*[14] *and each fragment has out-degree at most* 1.

Proof. Let E_1, E_2 be the set of undirected edges $\{F, F'\}$ such that F sent or received a primary or secondary merge request from F', respectively. Each fragment has out-degree at most 1 in $E_1 \cup E_2$, because no fragment sends more than one merge request.

There are no cycles (directed or undirected) in E_1, because large fragments do not send merge requests, and small fragments only send primary merge requests to fragments whose leader has a larger ID. Now consider the edges in E_2. A fragment F sends a secondary merge request to F' only if F neither sent nor received a primary merge request. Thus, F is not adjacent to any edges in E_1. Moreover, fragment F is adjacent to exactly one edge in E_2: fragment F sends out only one secondary merge request, and is not the target of any secondary merge requests, because it did not send a primary merge request. Thus, the degree in $E_1 \cup E_2$ of any fragment F adjacent to some edge in E_2 is exactly 1, and so adding the edges in E_2 to the edges in E_2 does not create any cycles, directed or undirected. □

6.1.2 Merging fragments.

The mergers are performed in parallel for all fragments. A small fragment H merges into a fragment F along an outgoing edge (v, u) (where $v \in H, u \in F$) as follows:

1. First, we re-orient the tree inside H so that node v becomes the root, by having node v carry out a breadth-first search (BFS) inside fragment H: the BFS originates at node v, and it travels along the edges of the

[14] We make the distinction between directed and undirected cycles because strictly speaking, a directed cycle of length 2 is not considered an undirected cycle but rather an undirected edge.

undirected tree that underlies v's fragment; that is, all edges $\{\{u, w\} : up(u) = w \text{ or } up(w) = u\}$. As the BFS travels, nodes that receive the BFS along an edge travelled "in the wrong direction" flip the edge: that is, if node u receives the BFS token from node w such that $up(u) \neq w$ (but $up(w) = u$), node u sets $up(u) = w$ after u forwards the token.

2. Node v sets $up(v) = u$, effectively merging the fragments.

3. Node v checks whether the fragment resulting from the merger is still small. To do this, node v initiates a BFS to depth at most $L - 1$, by sending out a BFS token, $BFS(v, 1)$, which is forwarded along tree edges in both directions (i.e., the BFS travels from node u to node w if $up(u) = w$ or $up(w) = u$), increasing the hop-count at each step (a node that receives $BFS(v, d)$ sends to its children $BFS(v, d + 1)$). The token travels until it either hits a dead end (a node with no tree edges other than the one through which the token was received), or has traveled to distance $L - 1$ from v, or L rounds have passed; the token remains at the last node it was able to reach. For convenience, we sometimes write $BFS(v)$ to denote a BFS token $BFS(v, d)$ originating at v when we do not care about the distance travelled (d).

   Since multiple BFS instances may be initiated in the new fragment, nodes *pipeline* BFS tokens that they receive: each node stores all BFS tokens it needs to forward, and in each round, it forwards the BFS token with the smallest-ID source to all neighbors in the tree, except the edge through which the token was received (and the token is then removed from the set of tokens that need to be forwarded). It is not guaranteed that all BFS instances will be able to complete, but the BFS initiated by the smallest-ID node will complete in L rounds.

   The BFS phase is capped at L rounds, at which point each BFS token $BFS(v, d)$ becomes a *convergecast token*, $CON(v, s)$ carrying the following value s:

   - If the BFS token reached a dead end, or traveled to distance $L - 1$ from its source ($d = L - 1$), then it converts into a convergecast token with value 1, $CON(v, 1)$.
   - Otherwise, the BFS token converts into a convergecast token with value L, $CON(v, L)$ (signifying that the fragment is large, which caused the BFS not to complete).

   Next we begin a *convergecast phase*, which is again capped at L rounds, during which convergecast tokens are sent back to their sources: a node u that previously sent a token $BFS(v)$ to nodes $w_1, \ldots, w_\ell$ waits to receive convergecast tokens $CON(v, s_1), \ldots, CON(v, s_\ell)$ from $w_1, \ldots, w_\ell$, respectively, and node u then sends a token $CON(v, \max(L, 1 + \sum_{i=1}^{\ell} s_i))$ along the edge from which u received $BFS(v)$.

   As in the BFS phase, nodes may receive multiple convergecast tokens that must be returned along the same edge; we again use pipelining, with each node forwarding the convergecast token with the smallest ID first. We say that the BFS and convergecast initiated by node v *completed* if node v receives a convergecast token from all of its neighbors.

   Finally, after the BFS and the convergecast phases complete, we begin a *broadcast* phase, where nodes inform one another whether their current fragment is small or large. Initially, each node u sets $small(u)$ as follows:

   - If u initiated a BFS and convergecast that completed, and received back convergecast tokens $CON(u, s_1), \ldots, CON(u, s_\ell)$ from all of its tree neighbors, and if furthermore we have $1 + \sum_{i=1}^{\ell} s_i < L$, then u sets $small(u) = 1$.
   - Otherwise, $small(u) = 0$.

   Next, for L rounds, all nodes send their current $small(u)$ value to all their tree neighbors; if a value of 1 is received from any neighbor, node u sets $small(u) = 1$, and will send 1 in future rounds as well.

4. We inform the nodes of small (merged) fragments of the identity of the leader, the root of their new fragment: any node v such that $small(v) = 1$ and $up(v) = \perp$ sends out a message informing its fragment that it is the new leader, and the message is forwarded down tree edges to the entire fragment, for L rounds.

The correctness of the merge step follows from the following observations.

OBSERVATION 4. *At any point in the execution, the subgraph induced by the up-pointers is a collection of rooted trees.*

Proof. By induction on the number of steps: initially all *up*-pointers are $\perp$. Now suppose that at the beginning of the current merge step, we indeed have a collection of rooted trees. We first change the orientation of some of the rooted trees, which does not create cycles. Then we add *up*-edges corresponding to merge requests. We know from Observation 3 that the merge requests induce an acyclic graph over the fragments, which each fragment having out-degree 1. Thus, the underlying graph obtained after adding the *up*-edges corresponding to the merge requests is a forest. Moreover, a fragment F that merges into another fragment F' along an edge (v, u) where $v \in F$, $u \in F'$ first re-orients itself so that v is the root of F; an easy induction on the number of fragments merged shows that the resulting merged fragment is oriented towards the root. ◻

OBSERVATION 5. *Following the merge step, each node v has $small(v) = 1$ iff v's fragment (following the merger) is small.*

Proof. An easy induction on rounds shows that if node u sends a convergecast token $CON(v, s)$ where $s < L$, then s is the size of the subtree of u in the directed tree corresponding to the fragment of u and v, with all tree edges oriented outwards away from v.

Let F be a small fragment (following the merger), and let v be the smallest node that initiates a BFS in F. Note that at least one node in F initiates a BFS: since F is small, it was formed by the merger of at least two small fragments, and the node that sent a merge request initiates a BFS.

During the BFS and convergecast phases, the tokens associated with v are never delayed: they are always forwarded immediately by any node that receives them, as v has the smallest ID in the fragment (and no tokens are received from outside the fragment — tokens are sent only along tree edges). Thus, after L rounds, the BFS token of v is able to complete a depth-L traversal, and another L rounds suffice for the convergecast to complete. Since the fragment is small, the count that node v receives is less than L, and so node v sets $small(v) = 1$ at the beginning of the broadcast phase. Finally, since the fragment is small, the L rounds of the broadcast phase suffice for all nodes in the fragment to learn this fact and set their *small* values to 1 as well.

Now suppose that F is a large fragment. We claim that all nodes $v \in F$ set $small(v) = 0$ at the beginning of the broadcast phase, and thus, after the broadcast phase, all nodes v will still have $small(v) = 0$: the only nodes v that set $small(v) = 1$ are those nodes that initiated a BFS and convergecast which were able to complete, returning a complete count of the nodes in F to v. But in this case, since F is a large fragment, the count cannot be less than L, so v will set $small(v) = 0$. ◻

This completes the description of the L-forest computation. The time required is $O(L \log L)$: wince each fragment of size less than L is able to join with another fragment in every round, as long as there remain small fragments, the size of the smallest fragment is at least doubled in each round. Thus, after $O(\log L)$ merge steps, every fragment is of size at least L. Each merge step requires $O(L)$ rounds, for a total of $O(L \log L)$ rounds.

6.2 Assigning helper nodes. Given a $(k+1)$-forest, we show that in $O(k)$ additional rounds, we can construct an $(k, O(k), O(k^2), O(k))$-helper assignment. Our goal is to assign to each node $v \in V$ a set H_v of k helper nodes, and construct a collection of paths P connecting each $v \in V$ to all its helper nodes, such that each path is of length $d = O(k)$, the total congestion of P is at most $c = O(k^2)$ on each edge, and no node is assigned to help more than $O(k)$ other nodes.

6.2.1 Computing the subtree size. The *subtree* of a node v is the set of v's descendants in its fragment, that is, the set of nodes u such that v is reachable from u by following *up*-pointers (including v itself). We say that node v *has a large subtree* if v's subtree is of size at least $k + 1$.

We begin by having each node v check whether its subtree is large or not. This is done by having every node initiate a BFS which is sent down the tree for $L = k + 1$ steps, followed by a convergecast that computes the size of the subtree.

Formally, each node v sends a BFS token $BFS(v)$ to its children in its fragment, and the token is forwarded to depth at most L down the tree. Note that the token $BFS(v)$ reaches a node u in v's subtree after exactly $\mathrm{dist}_F(u, v)$ rounds, where $\mathrm{dist}_F(u, v)$ denotes the distance from node u up to node v by following *up*-pointers. In particular, since the fragment is a rooted tree, no node receives more than one BFS token in any round, and so there is no congestion. If a BFS token reaches a leaf, it remains at that node.

After L rounds of BFS, we begin a convergecast up the tree, to compute the size of each node's subtree. Each BFS token converts into a convergecast token of size 1. To avoid congestion, we arrange the convergecasts in reverse order of the distance they must travel: if node u received a token $BFS(v, d)$, then node u will send back a convergecast token $CON(v, s)$ in round $L - d + 1$ of the convergecast. This ensures that convergecasts pipeline neatly up the tree, with each node needing to send at most one convergecast token $CON(v, s)$ per round, after receiving from its children their own tokens $CON(v, s_1), \ldots, CON(v, s_\ell)$ and summing their sizes, $s = 1 + \sum_{i=1}^{\ell} s_i$.

Eventually, each node v receives a convergecast counting the number of nodes at distance at most L down the tree from v. If there are at least $L = k + 1$ such nodes, then node v has a large subtree, and otherwise it does not.

6.2.2 Assigning helpers to nodes with large subtrees. Recall that node v *has a large subtree* if v's subtree is of size at least $k + 1$. All such nodes are able to recruit helper nodes from inside their own subtree, in parallel: each node v that has a large subtree sends a BFS token to its children, which is forwarded down the tree edges for at most k hops by the nodes that receive it. The BFS token stores the path along which it has traveled. As above, a node u never needs to forward more than one BFS token in any given round, because there is at most one node that initiates a BFS at any given distance up the tree from v.

After reaching depth k or a leaf, nodes start returning the BFS tokens up the tree edges, with each token sent up carrying paths to at most k nodes: when the children $w_1, \ldots, w_\ell$ of a node u send their tokens back up to u, node u sends up a token carrying the first $\leq k$ paths it finds in its children's tokens. We arrange the convergecasts in reverse order of the distance they must travel, as explained in Section 6.2.1. Eventually, each originating node v receives tokens from its children, and selects a set H_v of k helper nodes together with paths P_v of length $\leq k$ reaching each helper node. (This many helper nodes are guaranteed to exist for v, because its subtree is connected and has size $\geq k + 1$; thus, a BFS to depth k explores at least k nodes, excluding v.)

The size of each token sent during this phase is $O(k^2)$ bits. Also, since the BFS downward-propagation phase takes k rounds, and nodes send at most one token per round, each node forwards a total of at most k tokens; therefore, no node appears more than k times altogether on all paths chosen, and no node helps more than k other nodes.

6.2.3 Nodes with small subtrees. If v has a small subtree (of size $< k + 1$), then v has an ancestor u which has a large subtree, and is at distance at most k from u: we know that the tree to which v belongs has size $\geq k+1$, because all trees in the forest have size $\geq L = k + 1$. Thus, as we go up from v to the root of its tree, we will eventually reach some node whose subtree is of size $\geq k$. Let u be the first such node. The distance from v to u is at most k, because after going up k hops, the size of the subtree of the node we reach is clearly at least $k + 1$.

To find helper nodes, node v sends up a request $\mathsf{REQ}(v)$ to node u, which is forwarded by all nodes on the path from v to u. This requires message size $O(k)$, because u is the lowest ancestor of v that has a large subtree; thus, on every edge leading up to u, at most k requests are sent. Nodes that have a large subtree store all requests they receive, and do not forward them further up the tree. Note that a node u only receives requests from descendants at distance at most k from itself, as we explained above.

We wait for k rounds for nodes with large subtrees to collect all the requests from their descendants. Then, each node u with a large subtree assigns helper nodes as follows:

- If u received at most $k + 1$ requests from its descendants, it answers each request $\mathsf{REQ}(v)$ by sending down to v own helper nodes and paths, H_u and P_u, and the path from u to v. Node v then sets $H_v = H_u$, and it computes P_v by preprending the path from v to u to each path in P_u. This creates paths of length at most $O(k)$, congestion $O(k^2)$, and each of v's helper nodes is assigned to help at most $k + 1$ additional nodes, for a total of $O(k)$.
- If u receives more than $k + 1$ requests from its descendants, it partitions the requests into batches $B_1, \ldots, B_{t+1}$, such that
  - For each $i = 1, \ldots, t$ we have $k + 1 \leq |B_i| \leq 2k$,
  - $|B_{t+1}| \leq k$,

– For each child v of u, all requests made by v and its descendants belong to the same batch.

Such a partition exists, because only nodes with small subtrees send requests to u: we can create the partition greedily by starting with an empty batch and adding subtrees of u's children one after the other until the batch first reaches size at least $k + 1$, then moving on to a new batch and continuing in this manner. The size of each batch thus created is at most $2k$, because adding an entire small subtree to a batch containing fewer than $k + 1$ requests yields a total of at most $k + k = 2k$ requests.

For each full-sized batch $B_i = \{\mathsf{REQ}(v_j) \mid j = 1, \ldots, s\}$ where $i \leq t$, $k + 1 \leq s \leq 2k$, node u defines a helper set $H_i = \{v_1, \ldots, v_{k+1}\}$ comprising the first $k + 1$ nodes that made the requests in the batch. Node u answers each request $\mathsf{REQ}(v_j)$ by sending down the set H_i, together with paths from v_j to the nodes of H_i: each path goes up from v_j to B_i, then down to a descendant $v_{j'}$ where $j' \neq i$. Each node in the batch chooses k nodes in H_i other than itself as its helper nodes (since $|H_i| = k + 1$, this is possible).

The length of the paths is $O(k)$, and the congestion is at most $O(k^2)$ as well, since we created at most $(k + 1)^2$ paths going through each edge from u to a child.

For the "remainder" batch B_{t+1} (if it is not empty), u sends down the helper nodes H_1 from the first batch and paths to them. Once again, the length of the paths is $O(k)$, the additional congestion is at most $O(k^2)$, and each node is assigned to help at most k more nodes.

References

[1] 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.

[2] László Babai and Shlomo Moran. Arthur-merlin games: a randomized proof system, and a hierarchy of complexity classes. *Journal of Computer and System Sciences*, 36(2):254–276, 1988.

[3] Alkida Balliu, Gianlorenzo D'Angelo, Pierre Fraigniaud, and Dennis Olivetti. What can be verified locally? In *STACS*, volume 66, pages 8:1–8:13, 2017.

[4] Mor Baruch, Pierre Fraigniaud, and Boaz Patt-Shamir. Randomized proof-labeling schemes. In *Symposium on Principles of Distributed Computing (PODC)*, pages 315–324. ACM, 2015.

[5] Michael Ben-Or, Oded Goldreich, Shafi Goldwasser, Johan Håstad, Joe Kilian, Silvio Micali, and Phillip Rogaway. Everything provable is provable in zero-knowledge. In Shafi Goldwasser, editor, *Advances in Cryptology (CRYPTO)*, volume 403, pages 37–56, 1988.

[6] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In Janos Simon, editor, *Symposium on Theory of Computing (STOC)*, pages 1–10, 1988.

[7] Nir Bitansky, Alessandro Chiesa, Yuval Ishai, Omer Paneth, and Rafail Ostrovsky. Succinct non-interactive arguments via linear interactive proofs. In Amit Sahai, editor, *Theory of Cryptography*, pages 315–333, 2013.

[8] Dan Boneh, Elette Boyle, Henry Corrigan-Gibbs, Niv Gilboa, and Yuval Ishai. Zero-knowledge proofs on secret-shared data via fully linear pcps. In *CRYPTO 2019*, volume 11694 of *Lecture Notes in Computer Science*, pages 67–97.

[9] Elette Boyle, Shafi Goldwasser, and Stefano Tessaro. Communication locality in secure multi-party computation - how to run sublinear algorithms in a distributed setting. In Amit Sahai, editor, *Theory of Cryptography Conference (TCC)*, volume 7785, pages 356–376, 2013.

[10] Keren Censor-Hillel, Ami Paz, and Mor Perry. Approximate proof-labeling schemes. In *International Colloquium on Structural Information and Communication Complexity*, pages 71–89. Springer, 2017.

[11] Nishanth Chandran, Wutichai Chongchitmate, Juan A. Garay, Shafi Goldwasser, Rafail Ostrovsky, and Vassilis Zikas. The hidden graph model: Communication locality and optimal resiliency with adaptive faults. In *Innovations in Theoretical Computer (ITCS)*, pages 153–162, 2015.

[12] Nishanth Chandran, Juan A. Garay, and Rafail Ostrovsky. Improved fault tolerance and secure computation on sparse networks. In Samson Abramsky, Cyril Gavoille, Claude Kirchner, Friedhelm Meyer auf der Heide, and Paul G. Spirakis, editors, *International Colloquium on Automata, Languages, and Programming, (ICALP)*, volume 6199, pages 249–260, 2010.

[13] David Chaum, Claude Crépeau, and Ivan Damgård. Multiparty unconditionally secure protocols (extended abstract). In Janos Simon, editor, *Symposium on Theory of Computing (STOC)*, pages 11–19, 1988.

[14] Laurent Feuilloley and Pierre Fraigniaud. Survey of distributed decision. *Bulletin of the EATCS*, 119, 2016.

[15] Laurent Feuilloley and Pierre Fraigniaud. Error-sensitive proof-labeling schemes. In *Symposium on Distributed Computing (DISC)*, pages 16:1–16:15, 2017.

[16] Juan A. Garay, Shay Kutten, and David Peleg. A sublinear time distributed algorithm for minimum-weight spanning trees. *SIAM Journal on Computing*, 27(1):302–316, 1998.

[17] Juan A. Garay and Rafail Ostrovsky. Almost-everywhere secure computation. In *Advances in Cryptology (EUROCRYPT)*, volume 4965 of *Lecture Notes in Computer Science*, pages 307–323, 2008.

[18] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In Alfred V. Aho, editor, *Symposium on Theory of Computing (STOC)*, pages 218–229, 1987.

[19] Oded Goldreich, Silvio Micali, and Avi Wigderson. Proofs that yield nothing but their validity for all languages in NP have zero-knowledge proof systems. *Journal of the ACM (JACM)*, 38(3):691–729, 1991.

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

[21] Mika Göös and Jukka Suomela. Locally checkable proofs in distributed computing. *Theory of Computing*, 12(1):1–33, 2016.

[22] Shai Halevi, Yuval Ishai, Abhishek Jain, Ilan Komargodski, Amit Sahai, and Eylon Yogev. Non-interactive multiparty computation without correlated randomness. In Tsuyoshi Takagi and Thomas Peyrin, editors, *Advances in Cryptology (ASIACRYPT)*, volume 10626, pages 181–211, 2017.

[23] Shai Halevi, Yuval Ishai, Abhishek Jain, Eyal Kushilevitz, and Tal Rabin. Secure multiparty computation with general interaction patterns. In Madhu Sudan, editor, *Innovations in Theoretical Computer Science (ITCS)*, pages 157–168, 2016.

[24] Markus Hinkelmann and Andreas Jakoby. Communications in unknown networks: Preserving the secret of topology. *Theoretical Computer Science Journal*, 384(2-3):184–200, 2007.

[25] Yuval Ishai, Eyal Kushilevitz, Sigurd Meldgaard, Claudio Orlandi, and Anat Paskin-Cherniavsky. On the power of correlated randomness in secure computation. In *Theory of Cryptography (TCC)*, volume 7785 of *Lecture Notes in Computer Science*, pages 600–620, 2013.

[26] Gillat Kol, Rotem Oshman, and Raghuvansh R. Saxena. Interactive distributed proofs. In *Symposium on Principles of Distributed Computing (PODC)*, pages 255–264, 2018.

[27] Liah Kor, Amos Korman, and David Peleg. Tight bounds for distributed minimum-weight spanning tree verification. *Theory of Computing Systems*, 53(2):318–340, 2013.

[28] Janne H Korhonen and Jukka Suomela. Towards a complexity theory for the congested clique. In *Symposium on Distributed Computing (DISC)*, pages 55:1–55:3, 2017.

[29] Amos Korman and Shay Kutten. Distributed verification of minimum spanning trees. *Distributed Computing*, 20(4):253–266, 2007.

[30] Amos Korman, Shay Kutten, and David Peleg. Proof labeling schemes. *Distributed Computing*, 22:215–233, 2010.

[31] Carsten Lund, Lance Fortnow, Howard Karloff, and Noam Nisan. Algebraic methods for interactive proof systems. *Journal of the ACM (JACM)*, 39(4):859–868, October 1992.

[32] Tal Moran, Ilan Orlov, and Silas Richelson. Topology-hiding computation. In *Theory of Cryptography Conference (TCC)*, volume 9014 of *Lecture Notes in Computer Science*, pages 159–181, 2015.

[33] Moni Naor, Merav Parter, and Eylon Yogev. The power of distributed verifiers in interactive proofs. In Shuchi Chawla, editor, *Symposium on Discrete Algorithms (SODA)*, pages 1096–115, 2020.

[34] Merav Parter and Eylon Yogev. Secure distributed computing made (nearly) optimal. In *Symposium on Principles of Distributed Computing (PODC)*, pages 107–116, 2019.

[35] Boaz Patt-Shamir and Mor Perry. Proof-labeling schemes: broadcast, unicast and in between. In *International Symposium on Stabilization, Safety, and Security of Distributed Systems*, pages 1–17. Springer, 2017.

[36] Atish Das Sarma, Stephan Holzer, Liah Kor, Amos Korman, Danupon Nanongkai, Gopal Pandurangan, David Peleg, and Roger Wattenhofer. Distributed verification and hardness of distributed approximation. *SIAM Journal on Computing*, 41(5):1235–1265, 2012.

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

[38] Jukka Suomela. Survey of local algorithms. *ACM Computing Surveys (CSUR)*, 45(2):24, 2013.

[39] Andrew Chi-Chih Yao. Protocols for secure computations (extended abstract). In *Foundations of Computer Science (FOCS)*, pages 160–164, 1982.