Fully Local Succinct Distributed Arguments

Eden Aldema Tshuva
Tel Aviv University, Israel

Rotem Oshman
Tel Aviv University, Israel

Abstract

Distributed certification is a proof system for detecting illegal network states or improper execution of distributed algorithms. A certification scheme consists of a proving algorithm, which assigns a certificate to each node, and a verification algorithm where nodes use these certificates to decide whether to accept or reject. The system must ensure that all nodes accept if and only if the network is in a legal state, adhering to the principles of completeness and soundness. The main goal is to design a scheme where the verification process is local and the certificates are succinct, while using as efficient as possible proving algorithm.

In cryptographic proof systems, the soundness requirement is often relaxed to computational soundness, where soundness is guaranteed only against computationally bounded adversaries. Computationally sound proof systems are called arguments. Recently, Aldema Tshuva, Boyle, Cohen, Moran, and Oshman (TCC 2023) showed that succinct distributed arguments can be used to enable any polynomially bounded distributed algorithm to certify its execution with polylogarithmic-length certificates. However, their approach required a global communication phase, adding $O(D)$ communication rounds in networks of diameter D, which limits its applicability to local algorithms.

In this work, we give the first construction of a fully local succinct distributed argument system, where the prover and the verifier are both local. We show that a distributed algorithm that runs in R rounds, has polynomial local computation, and messages of B bits each can be compiled into a self-certifying algorithm that runs in $R + \text{polylog}(\text{n})$ rounds and sends messages of size $B + \text{polylog}(\text{n})$, with certificates of length $\text{polylog}(n)$. This construction has several applications, including self-certification for local algorithms, ongoing certification of long-lived algorithms, and efficient local mending of the certificates when the network changes.

2012 ACM Subject Classification Theory of computation → Cryptographic protocols

Keywords and phrases distributed certification, proof labeling schemes, SNARG

Digital Object Identifier 10.4230/LIPIcs.DISC.2024.1

Funding *Eden Aldema Tshuva*: Supported in part by AFOSR Award FA9550-23-1-0312, and an Algorand Foundation grant.
Rotem Oshman: Research funded by the Israel Science Foundation, Grant No. 2801/20, and also supported by Len Blavatnik and the Blavatnik Family Foundation.

1 Introduction

In this work we study *distributed certification*, a mechanism that is useful for ensuring correctness and fault-tolerance in distributed algorithms: the goal is to efficiently check, on demand, whether the system is in a legal state or not. To that end, the network computes in advance auxiliary information in the form of *certificates* stored at the nodes of the network, and we design an efficient *verification procedure* that allows the nodes to interact with one another and use their certificates to verify that the system is in a legal state. Since we do not trust that the system is in a legal state at verification time, we think of the certificates as being provided by an *untrusted prover*, whose goal is to convince us that the system is in a legal state even when it is not. One can therefore view distributed certification as a distributed analog of NP.

Distributed certification was implicit in early work on fault detection and self-stabilization (e.g., [4]), as a mechanism for detecting that the network has changed (for instance, due to the failure of a communication link) and action must be taken to address the change. It was formalized as an object of independent interest in [34], and has since received significant attention in distributed computing literature (e.g., [32, 33, 12, 20, 25, 22, 17, 42, 40, 23, 22, 30, 41, 9]). Almost all work in the area is solely concerned with optimizing the length of the certificates, which is viewed as a proxy for the efficiency of the verification algorithm: in [34] and most of the follow-up work, the verification algorithm consists of a single round of communication, where nodes send their certificates to their neighbors, and then output a local decision whether to accept or reject. Our work departs from most of the literature on distributed certification in two important ways: first, in addition to the certificate length, we are also concerned with the efficiency of the *prover algorithm*, that is, the algorithm that computes the certificates; and second, following [2], we relax the correctness requirement from *perfect soundness* to *computational soundness.* Next we discuss these two aspects of our work and lay out our motivation for departing from the approach taken by most prior work.

Proving as fast as computing. In the field of delegation of computation (the sequential notion analogous to distributed certification), a great amount of effort has been devoted to constructing provers that add minimal overhead on top of the algorithm whose correctness they aim to certify [44, 3, 26, 24, 10, 49, 50, 11, 35]. This is referred to as "proving as fast as computing". Efficient provers are needed for any practical deployment of a delegation scheme, and therefore designing proof systems where the prover is efficient is a key element in applications such as proofs on the blockchain; for instance, [8, 7] have made great progress in the efficiency of the prover and are used in practice. See [47] for a survey of the subject.

In the distributed setting the need for efficient provers is much the same: in order for distributed certification to serve as a practical mechanism for fault tolerance, we must be able to compute the certificates efficiently. Thus, the goal of our work is *proving as fast as distributed computing*:

> *Given a distributed algorithm $\mathcal{D}$ that runs in polynomial communication rounds and local computation steps, construct a prover that runs alongside $\mathcal{D}$, adding at most a polylogarithmic overhead to the rounds and local computation steps.*

In other words, our goal is to obtain distributed certification schemes where both the verifier and the prover are local (in terms of the overhead they add to the system), in contrast to traditional distributed certification, where only the verifier is a local distributed algorithm, and the prover is all-powerful.

Computational soundness. Most of the work on distributed certification is set in the *information theoretic* world, where the prover and the network nodes are computationally unbounded. The two requirements from a certification scheme for a network property $\mathcal{L}$ are:

- Completeness: if the property $\mathcal{L}$ holds, then there exists a certificate assignment that convinces all nodes to accept; and
- Soundness: if the property $\mathcal{L}$ does not hold, then no certificate assignment convinces all nodes to accept.

Unfortunately, the information-theoretic setting inherits some powerful lower bounds from nondeterministic two-party communication complexity: for example, it is known that some network properties require $\Omega(n^2)$-bit certificates [25], and some simple and natural properties such as proving that the network has a given diameter require $\Omega(n)$-bit certificates, even when the verifier is randomized [22].

This motivates us to consider the following relaxation of the soundness requirement, known as *computational soundness* ([39]):

- Computational soundness: if the property $\mathcal{L}$ does not hold, then no *poly-size prover*[1] can construct a certificate assignment that convinces all nodes to accept.

A proof system that has computational soundness is called an *argument*, and in the distributed setting we call it a *distributed argument*.

One might ask whether computational soundness indeed captures the type of faults from which the network wishes to protect itself. We argue that the answer is *yes*, in most if not all practical scenarios, if one is willing to assume standard cryptographic assumptions hold. The key here is that any fault that could be simulated by an efficient algorithm, cannot break computational soundness, since if it could, that would mean that an efficient algorithm can solve believed-to-be hard problems, such as the discrete logarithm. For example, if we wish to protect against hardware or software faults, then we should demand soundness against all certificates generated generated for an illegal state due to a buggy execution of a distributed algorithm in the network, or against those generated by a buggy version of the honest prover algorithm. But even a buggy prover is still an *efficient algorithm*. Similarly, faults caused by topology changes can also be simulated by an efficient algorithm, which again means that such faults cannot break a computationally sound certification scheme.

We remark that although in this work we weaken the soundness requirement, and construct a local distributed proving algorithm, we still require soundness against *global provers*: the argument that we construct is sound against any polynomial-size "cheating prover" that sees the entire network and tries to produce certificates that fool the network into accepting even though the network is not in a legal state.

Distributed SNARGs. In delegation of computation (the sequential notion analogous to distributed certification), the gold standard is to construct a *succinct non-interactive argument* (SNARG) whose security relies on standard cryptographic hardness assumptions, such as learning with errors [16] or, bilinear maps [48], and decisional Diffie-Hellman [14].[2] A SNARG is a computationally sound proof system in which a polynomial-size prover $\mathcal{P}$ certifies a statement of the form "$x \in \mathcal{L}$," where x is an input of size n and $\mathcal{L}$ is a language, by providing a computationally weak verifier $\mathcal{V}$ with a proof π, of length $|\pi| = \text{polylog}(n)$. The verifier then examines the input x and the proof π, and decides in *linear time* in n whether to accept or reject.[3] It is guaranteed that the honest prover $\mathcal{P}$ can convince the verifier $\mathcal{V}$ to accept any true statement with probability 1 (*perfect completeness*), and at the same time, no poly-size cheating prover can convince the verifier to accept with non-negligible probability (*computational soundness*). The requirement that the proof π be of polylogarithmic length is called *succinctness*.

In recent years, the fruitful line of work on delegation of computation has culminated in the construction of SNARGs for all properties in P [16, 48, 27, 14, 28]. In [2], this was extended to distributed network algorithms. A *distributed* SNARG [2] for a property $\mathcal{L}$ is a computationally sound proof system $(\mathcal{P}, \mathcal{V})$, consisting of

[1] Computational soundness, like other computational hardness notions, models the adversary as a non-uniform machine of polynomial size, as it is at least as strong as randomized.

[2] Throughout this work, we refer to SNARGs for *deterministic computations*, which prove that some polynomial-time computation was executed correctly, and not SNARGs for NP, which are a much stronger cryptographic primitive that is not known to exist under standard cryptographic assumptions.

[3] Technically, the prover and the verifier take as input a security parameter λ, and their running time is polynomial in λ. We defer the discussion of the security parameter to Section 2.

- A prover $\mathcal{P}$, which may or may not be a distributed algorithm (both options were considered in [2]). Given a network graph $G = (V, E)$ and an input assignment $x : V \to \mathcal{X}$ specifying the input $x(v)$ to each node $v \in V$, the prover constructs a *proof* in the form of a certificate assignment $\pi : V \to \{0,1\}^*$, with each node v receiving a certificate $\pi(v)$ of length polylog(n) (where $n = |V|$).
- A verification procedure $\mathcal{V}$, which is a one-round distributed algorithm where every node $v \in V$ initially knows its UID, its input $x(v)$, its neighbors in G, and its certificate $\pi(v)$. Each node sends a (possibly different) message on each one of its edges, receives the messages sent by its neighbors, carries out some local computation, and then outputs *accept* or *reject*. The proof is considered accepted if and only if all nodes accept.

It was recently shown in [2] that any network property in P admits a distributed SNARG. Moreover, [2] constructed a *distributed prover*, which allows a polynomial-time distributed algorithm to certify the correctness of its output using certificates of size polylog(n). However, the prover constructed in [2] is *global*: although it is a distributed algorithm, it collects information from all the nodes of the network, which requires $\Omega(D)$ rounds in networks of diameter D (using messages of polylogarithmic size). This means that in some cases the prover's overhead may eclipse the running time of the distributed algorithm whose correctness it certifies, e.g., if the original algorithm is a local algorithm.

1.1 Our Contribution

In this work, we construct a *fully local distributed argument* that certifies the correctness of any polynomial distributed algorithm. That is, for a polynomial distributed algorithm $\mathcal{D}$, it certifies the following property:

$$\mathcal{L}_{\mathcal{D}} = \left\{ (G, x, y) : \begin{array}{l} \mathcal{D} \text{ produces output } y : V \to \mathcal{Y} \text{ when executed in} \\ \text{the network } G = (V, E) \text{ with input assignment } x : V \to \mathcal{X} \end{array} \right\}.$$

Our construction uses two cryptographic primitives: *collision-resistant hash functions* and *batch arguments for* NP. These are known to exist under several standard cryptographic assumptions: subexponential hardness of Diffie-Hellman [18, 14]; polynomial hardness of learning with errors [1, 15]; and polynomial hardness of bilinear maps [48].

▶ **Theorem 1.** *Assume collision-resistant hash functions and batch arguments for* NP *exist. Then for any distributed algorithm $\mathcal{D}$ that runs in* poly(n) *rounds local computation time, there is a distributed argument $(\mathcal{P}, \mathcal{V})$ certifying the property $\mathcal{L}_{\mathcal{D}}$, where the prover $\mathcal{P}$ is a distributed algorithm that adds an overhead of* polylog(n) *rounds to the execution of $\mathcal{D}$, sends* polylog(n)*-bit messages, and produces certificates of length* polylog(n)*, and the verifier $\mathcal{V}$ runs in one round and sends* polylog(n)*-bit messages.*

Our construction relies on *low-diameter network decompositions*, and represents a novel connection between this highly useful primitive and distributed certification.

Applications of our construction. Fully local distributed arguments have several applications. First, they enable efficient certification of local algorithms, where previous constructions either had an overhead of $\Theta(D)$ rounds or produced very long certificates (or both). That is, a distributed algorithm that runs in a small number of rounds but still has high communication complexity (i.e., it uses long messages), could now be certified in a few more rounds, using low communication complexity, and be verified in one round, with one message on each edge. Second, a local prover can be used to efficiently *mend* a proof of correctness, instead

Table 1 Generic distributed certification schemes, and the costs they incur when certifying an algorithm that runs for R rounds and sends B-bit messages in networks with n nodes, maximum degree Δ and diameter D.

	Soundness	Certificate	Verifier Message	Prover Overhead
PLS [34]	Perfect	$R \cdot B \cdot \Delta$	$R \cdot B \cdot \Delta$	No overhead
LCP [25][4]	Perfect	$\Theta(n^2)$	$\Theta(n^2)$	Not distributed
RPLS [23]	Statistical	$R \cdot B \cdot \Delta^2$	$O(\log n)$	1 round
LVD-SNARGs [2][5]	Computational	$\text{poly}(\lambda, \log n)$	$\text{poly}(\lambda, \log n)$	Not distributed
LVD-SNARGs [2]	Computational	$\text{poly}(\lambda, \log n)$	$\text{poly}(\lambda, \log n)$	$O(D)$
This Work	Computational	$\text{poly}(\lambda, \log n)$	$\text{poly}(\lambda, \log n)$	$\text{polylog}\, n$

of re-computing it from scratch when a change occurs in the network. Many distributed algorithms support *local correction* (also called *fixing*, *mending* or *healing*) of their output, that is, if a change in the network causes the output of the algorithm to become incorrect, there is a local procedure that executes only in the area of the network where the change occurred and "fixes" the output of the algorithm (see, e.g., [5, 19, 36, 6, 31] and the references therein). Following the execution of the local correction procedure, our fully local prover can also mend the correctness certificate, by executing the prover to re-certify correctness in the area of the network that was modified by the correction procedure. This application of our work creates a new tie between local correction and distributed certification, areas that both arose originally from fault tolerance and self-stabilization but have drifted apart over time.

Finally, fully local distributed arguments are an important step towards *incrementally verifiable distributed computation*. In sequential computing, incrementally verifiable computation (IVC, [46, 43]) allows for the incremental construction of a certificate of correctness, which is updated after each step taken by the sequential algorithm, and does not require storing the entire trace of the computation in memory. Incrementally verifiable computation is especially relevant in distributed systems, which are often long-lived and reactive. As a first step towards incrementally verifiable distributed computation, it is necessary to have a low-overhead prover that can be called many times during the computation without blocking for a long time, and our construction takes the first step in this direction.

1.2 Related Work

There are several known approaches to obtain generic schemes for certifying the correctness of any given distributed algorithm, although as we mentioned above, most of the focus in prior work has been on the efficiency of the verifier, not the prover. In Table 1 we summarize the tradeoffs that each approach achieves between the length of the certificates, the communication of the verifier, and the complexity of the prover. The table covers only schemes where the verifier runs for one round; it is sometimes possible to trade off certificate size against verifier rounds (see, e.g. [21, 42]), but the total communication over all verification rounds in the information-theoretic setting remains, in general, high. Next, we give a brief overview of each approach.

[4] In [25], the certificate and message size also depend on the size of the input to each node. That is, $\Theta(n^2)$ refers to the case where there is no input to the nodes or the input is of constant size.
[5] In [2], the property is assumed to be in P, and R and B are assumed to be at most polynomial in n.

In the first work to introduce proof labeling schemes [34] it is pointed out that any distributed algorithm can be certified by storing the entire transcript of the algorithm at each node. This can result in long certificates. In [23] it is shown that randomization can be used to exponentially decrease the communication of the verifier, but this comes at the cost of even longer certificates (as well as weakening soundness from *perfect soundness* to *statistical soundness*, where the verifier has some small probability of accepting an invalid proof). Another generic approach is to store a description of the entire network as the certificate at each node [25]. In addition to long certificates, this approach requires the prover to *know* the entire network, which rules out an efficient distributed implementation. However, it has the advantage of not being dependent on the communication complexity of the distributed algorithm to be certified, which could be useful for the certification of highly expensive algorithms.

The first work to introduce computationally sound distributed certification is [2], which showed that any network property in P can be certified using certificates of polylogarithmic length in this setting, assuming the prover knows the entire network. In addition, [2] constructs a generic scheme with a *distributed* prover that can certify the correctness of any given distributed algorithm that runs in polynomial rounds and local computation time. However, the prover in this case requires $O(D)$ rounds in networks of diameter D.

2 Preliminaries

In this section, we describe our network model (which is fairly standard) and the common reference string model, and then go over the two cryptographic primitives used in our construction; hash families with local openings and batch arguments for NP. The description of a batch argument is brief and the full syntax and definition can be found in Appendix A.2. Distributed Merkle trees, which are another existing construct we use, are discussed in Section 3.1, and defined formally in Appendix A.1. Moreover, for lack of space, we defer the full definition of our *fully local distributed* SNARG (fl-DSNARG) to Appendix A.4.

Network model. A synchronous distributed network is modeled as an undirected, connected graph $G = (V, E)$, where the nodes V are the processors participating in the computation, and the edges E represent bidirectional communication links between them. Each network node has a unique identifier v from some UID domain $[\widetilde{n}]$, and we assume that the size of the UID domain is polynomial in the network size n, so that a UID can be encoded in $O(\log n)$ bits. We often conflate the UID of a node with the vertex representing it in the network graph. In each communication round, each node sends a message to each of its neighbors; the nodes then receive the messages sent to them, carry out some internal computation, and then the next round begins. The input to the computation is represented by an assignment $x : V \to \mathcal{X}$, and the output by an assignment $y : V \to \mathcal{Y}$, where $\mathcal{X}, \mathcal{Y}$ are some input and output domains (respectively). Initially, each network node $v \in V$ knows its UID, its input $x(v)$, its neighborhood $N(v)$ in G, and the size n of the network (or a polynomial upper bound on the size, such as $\widetilde{n}$). Each node v eventually produces the output $y(v)$. We restrict attention to algorithms where the round complexity, the message length and the internal computation are polynomial in the size of the network.

The common reference string model and computational hardness. Our work is set in the *common reference string* (CRS) model, which is also the model in which the SNARG constructions of [16, 48, 27, 14] are set. In this model, all parties – in our case, the prover and all the network nodes – have access to a string that is sampled randomly by a trusted setup process, denoted by Gen, which takes a *security parameter* λ in unary representation. (This can be viewed as public randomness.) The security parameter governs the computational resources that must be invested to break the security or soundness of the protocol: we say that a task that involves the CRS is *hard* (or *computationally hard*) if given a CRS sampled using $\mathsf{Gen}(1^\lambda)$, no poly-size (in λ) adversary can succeed in the task, except with *negligible probability* – that is, probability smaller than $1/\lambda^c$ for any constant c. Batch arguments, described below, are defined in the CRS model and their soundness properties hold with respect to such a security parameter.

Collision resistance and hash families with local openings. A *hash family with local openings*, also sometimes known as a *hash tree*, allows a party that holds a vector $(x_1, \ldots, x_n)$ to compute a short hash of the vector, and later to locally *open* specific locations $i \in [n]$, producing a certificate that convinces another party that the value hashed in location i is x_i. The interface is as follows:

- $\mathsf{Gen}(1^\lambda) \to \mathsf{hk}$: a trusted, randomized setup procedure that takes a security parameter λ and outputs a hash key hk. The hash key can be thought of as a descriptor for a hash function chosen at random from a collision-resistant hash family,[6] which will be used in the computation of the hash value and its local openings.
- $\mathsf{Hash}(\mathsf{hk}, x) \to \mathsf{val}$: takes a hash key hk and a bit vector $x \in \{0,1\}^*$, and returns a hash value val.
- $\mathsf{Open}(\mathsf{hk}, x, i) \to (b, \rho)$: takes a hash key hk, a bit vector $x \in \{0,1\}^*$ and an index $i \in [|x|]$, and produces a bit b and an *opening* ρ, which is meant to serve as a certificate that $x_i = b$.
- $\mathsf{Verify}(\mathsf{hk}, \mathsf{val}, i, b, \rho) \in \{0,1\}$: takes a hash key hk, a hash value val, an index i, a bit b, and an opening ρ, and outputs an acceptance bit b. This procedure is meant to be executed by the other party, which does not know the value hashed, and wishes to verify that it has b in location i.

Our requirements of a hash family with local openings are as follows:

- Efficiency and succinctness: the procedures above run in time polynomial in their input, and output values of length at most $\text{poly}(\lambda, \log|x|)$.
- Completeness: for every hk generated by Gen, every input x and every index $i \in [|x|]$, if $\mathsf{val} = \mathsf{Hash}(\mathsf{hk}, x)$ and $(b, \rho) = \mathsf{Open}(\mathsf{hk}, x, i)$, then $\mathsf{Verify}(\mathsf{hk}, \mathsf{val}, i, b, \rho) = 1$.
- Collision-resistance with respect to openings: it is computationally hard, given a hash key hk generated by Gen, to find a hash value val, an index i, and two openings ρ_0, ρ_1, such that both $\mathsf{Verify}(\mathsf{hk}, \mathsf{val}, i, 0, \rho_0) = 1$ and $\mathsf{Verify}(\mathsf{hk}, \mathsf{val}, i, 1, \rho_1) = 1$.

We often describe the existence of an opening ρ such that $\mathsf{Verify}(\mathsf{hk}, \mathsf{val}, i, b, \rho_0) = 1$ by saying that *the hash value* val *opens to* b *in location/index* i.

Merkle tree [38] is a tree-based hash family with local openings that can be constructed from any collision-resistant hash family. Since collision-resistant hash families are known to exist under the assumption of either the hardness of the discrete logarithm problem [18] or the learning with errors problem [1], Merkle trees – and hash families with local openings in general – are also guaranteed to exist under the same assumptions. A Merkle tree over values $(x_1, \ldots, x_n)$ is a binary tree, where the leaves are $x_1, \ldots, x_n$, and each inner node is

[6] A *collision-resistant hash family* is a family of functions $\mathcal{H}$, such that it is computationally hard, given a random function from the family $h \in \mathcal{H}$, to find colliding inputs: x, y such that $h(x) = h(y)$.

the hash of the concatenation of its two children.[7] Merkle trees form the foundation for the distributed Merkle tree construction of [2], which is utilized in our construction of an $\mathsf{fl\text{-}DSNARG}$ (see Section 3.1).

Batch arguments for NP (BARGs) and their use in SNARG constructions. In the SNARG constructions of [16, 48, 27, 14], to prove that $x \in \mathcal{L}$ for a language $\mathcal{L}$ that is decided by a Turing machine M, the prover essentially proves the following statement:"there exist configurations $\mathsf{cf}_0, \ldots, \mathsf{cf}_T$ such that cf_0 is the initial configuration of M on input x, cf_T is an accepting configuration, and for each $i = 0, \ldots, T-1$, the machine M transitions from cf_i to cf_{i+1}". This highly-structured statement is a special case of T instances of an *index language*: an NP-language of the form $\mathcal{L} = \{(C, i) : \exists w.\, C(i, w) = 1\}$, where C is a circuit (in this case, verifying the transitions of the Turing machine), and i is an index. To prove such statements, [16, 48, 27, 14] use *batch arguments for* NP (BARGs), which we describe next, as they also serve as the basis for our construction in the current paper.

A batch argument for an index language $\mathcal{L}$ allows a prover to convince a verifier of a conjunction of the form $\varphi(C) = \bigwedge_{i=1}^{k} \exists w_i.\, C(i, w_i) = 1$, where the circuit C is known to both the prover and the verifier, but only the prover knows the witnesses $w_1, \ldots, w_k$. To prove this statement, the prover produces a short proof π, which the verifier is able to check. Crucially, the length of the proof π is linear in the length of a single witness $|w_i|$, but only polylogarithmic in the number of statements k.[8]

The BARGs we use in this work, like the BARGs used to construct SNARGs for P, are of a special type, called a *somewhere-extractable* BARG (seBARG). We give here a brief description of a seBARG. See Appendix A.2 for the full syntax and definition. A seBARG allows for the extraction of *one witness* from a convincing proof π, as follows:

- The procedure Gen that generates the CRS for the BARG can be called either in regular mode or in *trapdoor mode*. In trapdoor mode, Gen takes in addition to the security parameter λ an index $i \in [k]$, called the *binding index*. It outputs a pair $(\mathsf{crs}, \mathsf{td})$, where td is a *trapdoor* that can later be used to recover the i-th witness.

  In trapdoor mode, the Gen procedure has a property called *index hiding*: it is computationally hard to *find* the binding index i, given crs. This means that the prover, which is given only crs and not the trapdoor td, "cannot tell" which index we are interested in. In fact, it is hard to even tell whether Gen was called in regular mode or in trapdoor mode, as the distributions of the resulting string crs are computationally indistinguishable.
- The seBARG has an auxiliary *extraction* procedure, $\mathcal{E}(\mathsf{td}, C, \pi)$, which takes a trapdoor td, a circuit C and a proof π, and extracts one witness w.
- The seBARG has the *somewhere argument of knowledge* property: suppose we call Gen in trapdoor mode with a binding index i, and obtain $(\mathsf{crs}, \mathsf{td})$. Given only crs, it is computationally hard to find a proof π that is accepted by the verifier, such that when we extract a witness w_i using $\mathcal{E}(\mathsf{td}, C, \pi)$, we have $C(i, w) \neq 1$. In other words, it is hard for a poly-size adversary to fool the verifier into accepting a proof π if when we extract the i-th witness we find an inconsistency: the witness is not an NP-witness matching index i.

[7] More accurately, the leafs of a Merkle tree over $(x_1, \ldots, x_n)$ are hash values of $x_1, \ldots, x_n$, taken by a hash function collision-resistant hash family.

[8] Batch arguments for general NP languages allow proving a conjunction of NP statements. A batch argument for an index language allows for a highly efficient verification, as the verifier does not have to read k instances.

3 Technical Overview

In this section, we give a high-level overview of our construction of a fully local distributed SNARG (fl-DSNARG). This overview is somewhat informal, and some technical details are glossed over or omitted. The full construction and analysis are deferred to the full version of this paper.

Given a distributed algorithm $\mathcal{D}$, an input assignment x and an output assignment y, we wish to construct an argument that certifies the execution of each node, to prove that each node v indeed outputs $y(v)$ when $\mathcal{D}$ is executed with input x. We must take into consideration both the local computation of the node and the messages it sends and receives. A naïve approach would be to have each individual node construct a SNARG proof attesting to the internal computation steps that it takes while executing $\mathcal{D}$, but this is not enough: the challenge is that from the perspective of each node, the messages it receives from other nodes are essentially *inputs* to its computation, and the messages it sends are *outputs*. We must verify the consistency of these messages across each edge: messages that node u "attests to sending" to v should indeed be received at node v (i.e., they should be reflected correctly in the proof attesting to v's internal computation).

Unfortunately, while the real input x and output y of the distributed algorithm we are trying to certify are available at verification time, the messages sent by the algorithm are not: we cannot afford to store all messages sent and received as part of the certificate, as this would require far too much space. The solution is to carefully construct a hash of the messages, and use it to have nodes verify that the messages are consistent with the rest of their internal computation.

Recall from Section 2 that current centralized SNARG constructions consist of a batch argument for NP (a BARG) asserting the conjunction of T statements $S_1, \ldots, S_T$, each describing a single transition of a Turing machine. The configurations of the Turing machine are not available explicitly at verification – they are not part of the SNARG proof; instead, only a hash of the configurations is included in the proof. The proof consists (informally) of a batch argument proving that for each step i, the configuration hash opens in the appropriate locations to two configurations $\mathsf{cf}_i, \mathsf{cf}_{i+1}$, such that the Turing machine indeed transitions from cf_i to cf_{i+1}.[9] (The openings are part of the witness encoded inside the batch argument.)

We use a similar idea to handle the messages of the distributed algorithm: we construct multiple *local distributed Merkle trees*, which together are analogous to a hash tree of the messages, in such a way that each node v can compute openings to all the messages it sent or received. Intuitively, the message-hash has a "slot" (an index) for each directed edge $u \to v$ and round r, which is meant to record the message $m_r^{u \to v}$ that is sent from node u to node v in round r. We use the message-hash and the openings to construct two batch arguments: one attesting to the correctness of the internal computation steps at node i, and the second attesting to the consistency between the messages recorded "inside" the internal computation of node v, and the message-hash.

Consistency is verified at both endpoints of every directed edge $u \to v$: node u verifies that the message that it *sent* to node v in round r is indeed recorded in the message-hash in the slot for message $m_r^{u \to v}$, and node v verifies that the message that it *received* from node v in round r is recorded in the message-hash in the slot for message $m_r^{u \to v}$. This ties together the messages sent and received, and ensures that our proof captures the true execution of the distributed algorithm.

[9] Technically, we work with a hash of *a hash* of the configurations (two levels of hashing), so this description is not quite accurate. We give a more detailed one in Section 3.2.

Next we describe distributed Merkle trees, as introduced in [2], and our way of constructing multiply such trees where each one is *local*, using a new notion of *low-diameter edge cover*.

3.1 Local Distributed Merkle Trees

Distributed Merkle trees. As mentioned above, the idea of hashing together all the messages and using this hash to construct succinct arguments for distributed algorithms was introduced in [2], and implemented in the form of a *distributed Merkle tree* (DMT). We introduce DMTs here in a concise manner, see Appendix A.1 for the full syntax and definition.

A DMT is a hash with local openings for a collection of values $\{x_{v\to u}\}_{\{v,u\}\in E}$, one value for every directed edge $v \to u$ such that $\{v,u\} \in E$. The values are initially unordered, but an order will be imposed on them when the DMT is constructed. Initially, each node v knows all values $x_{u\to w}$ such that $u = v$ or $w = v$, that is, all values corresponding to edges that touch v. The DMT is essentially a hash of hashes:

- First, each node v computes a hash $\mathsf{rt}(v)$ (specifically, a Merkle tree) of its own "outgoing" values, $\{x_{v\to w}\}_{w\in N(v)}$; we call $\mathsf{rt}(v)$ the *local root of node v.*
- Then, the nodes compute together a global hash, rt, of the individual hashes $\{\mathsf{rt}(v) : v \in V\}$. We call rt the *global root* of the DMT.

Recall that the values $\{x_{v\to u}\}_{\{u,v\}\in E}$ are initially unordered. As the network constructs the DMT, it imposes an order over the nodes, and each node learns the index $I(v)$ where its own local root is hashed inside the DMT. Since node v constructed its own local root $\mathsf{rt}(v)$, it already knows the index $I_{v\to w}$ where it hashed each value $x_{v\to w}$. We think of the concatenation of these indices, $I(v) \parallel I_{v\to w}$, as *the index of* $x_{v\to w}$ in the DMT.

A DMT acts much like a regular Merkle tree over the values $\{x_{v\to u}\}_{\{v,u\}\in E}$. With the information that node v obtains during the construction of the DMT, it can produce an opening from the global root rt to any value $x_{v\to u}$ or $x_{u\to v}$ where $u \in \mathbb{N}(v)$. The DMT serves in [2] as a hash of all the messages sent in the network: each value $x_{v\to u}$ is itself a hash of all the messages that node v sent to node u during the execution of the algorithm.

In [2] it is shown that a DMT can be constructed in $O(D)$ rounds in networks of diameter D, using messages of polylogarithmic length. In other words, the DMT construction algorithm of [2] is *global* in nature: it first constructs a spanning tree of the entire network, and then computes the DMT by aggregating hash values up the tree and propagating openings down the tree.[10] This seems unavoidable, as the DMT is a hash of a collection of values that are initially spread across the entire network. However, one of our main technical contributions is to show that succinct distributed SNARGs do not require a global DMT; rather, we can get away with using a collection of *local* DMT*s*, each applied to a low-diameter subgraph of the original network graph, and thereby reduce the overhead of the prover from $O(D)$ rounds to $\mathrm{polylog}(n)$.

Using local DMTs. The key observation that enables us to construct a local prover is that both during the proving stage and at verification, each node requires access only to *its own messages* (sent or received).[11] Thus, there is no need to have a single DMT covering the entire network graph and providing all nodes with a single hash of all the messages; instead,

[10] However, the algorithm in [2] is still more communication-efficient than simply gathering the entire network's transcript in one location to compute the hash tree, as it uses polylogarithmic messages.

[11] The nodes do not actually require access to the messages themselves, but need to be able to verify consistency of them against some hash value.

we can compute many "small" DMTs, each covering a small neighborhood and providing the nodes of that neighborhood with one hash that they can use to access the messages they sent or received within that neighborhood. Moreover, we do not even need all edges of a given node to be covered by the *same* DMT: the crucial property we require is that *every edge must be covered by at least one* DMT, so that the messages that flow across the edge can be incorporated into the certificates of the two nodes at the endpoints of the edge.

With this observation in mind, our goal is to cover all edges of the network by a collection of subgraphs $H_1, \dots, H_k$, with each subgraph H_i maintaining its own DMT. The trade-off that governs our construction is a familiar one for distributed graph algorithms:

- On the one hand, we would like each subgraph H_i to have a *small diameter*, so that we can compute the DMT for the subgraph in a small number of rounds.
- On the other hand, each node should belong to only a small number of subgraphs, as each subgraph corresponds to a separate DMT and increases both the length of the certificate that the node eventually computes and the number (or alternatively, the size) of messages that the node must route during the proving stage, when the DMTs are constructed.

We call the cover $H_1, \dots, H_k$ a *low-diameter edge cover* (defined formally in Appendix A.3), and show below that it can easily be constructed from a *low-diameter decomposition* of G^2, the power-2 graph induced by our network graph G. (In G^2, two nodes $u, v \in V$ are neighbors if and only if their distance in G is at most 2.) We discuss how existing low-diameter decomposition constructions [45, 13] can be extended to handle G^2 while remaining in the CONGEST model in Appendix B.

In each cluster H_i, we compute a DMT over all messages sent over edges of H_i. Each such "local" DMT has a similar structure to the global DMT from [2]. The local DMT for cluster H_i requires $O(\text{diam}(H_i))$ rounds to construct, and this is why we require a small diameter for each cluster.

Constructing a low-diameter edge cover. Suppose we are given an (ℓ, m)*-low diameter decomposition* of $G^2 = (V, E')$: a partition of the nodes V into clusters $U_1, \dots, U_\ell \subseteq V$, and a coloring $c : \{1, \dots, \ell\} \to \{1, \dots, m\}$ of the clusters, such that:

1. The subgraph $G^2[U_i]$ induced by each U_i has diameter at most d, and
2. The coloring c is a proper coloring of the cluster graph: for any $i \neq j$ such that for nodes $u \in U_i$ and $v \in U_j$ there is an edge $\{u, v\} \in E'$ we have $c(i) \neq c(j)$.

Then we can obtain a low-diameter edge cover by defining subgraphs $H_1 = G[S_1], \dots, H_\ell = G[S_\ell]$ that each includes one cluster and all the nodes that are adjacent to it in G:

$$S_i = U_i \cup \{v \in V \;:\; \exists u \in U_i.\, \{v, u\} \in E\}\,.$$

For each node v, denote by $C(v) \subseteq \{S_1, \dots, S_\ell\}$ the set of clusters to which v belongs. We have the following properties:

- The diameter of each subgraph H_i is at most $2d + 2$: the original cluster $G^2[U_i]$ has diameter at most d with respect to G^2, which translates to diameter at most $2d$ with respect to G. Adding nodes adjacent to U_i in G increases the diameter to at most $2d + 2$.
- Every edge $\{u, v\} \in E$ is covered by some cluster $H_i = G[S_i]$: since $U_1, \dots, U_\ell$ is a partition of V, there is some $i \in [\ell]$ such that $u \in U_i \subseteq S_i$, and consequently $v \in S_i$. Thus, $\{u, v\}$ is covered by S_i.
- Each node belongs to at most m clusters of the edge cover: if v belongs two clusters S_i and S_j where $i \neq j$, then there exist nodes $u_i \in S_i, u_j \in S_j$ that are both at distance at most 1 from v in G. But this means that u_i and u_j are neighbors in G^2, and hence clusters U_i, U_j are adjacent in G^2, and must have a different color ($c(i) \neq c(j)$). This implies that $|C(v)| \leq m$.

3.2 Constructing the Distributed Argument

To construct our fully local distributed SNARG, we first need to fix a concrete model for the internal computation carried out by the network nodes, as the argument will need to refer to these computation steps. We begin by presenting such a model, and then outline the construction of the fl-DSNARG.

Modeling polynomial-time distributed algorithms. Consider a distributed algorithm $\mathcal{D}$ that runs in R rounds, with each node taking T local computation steps in each round (including steps required to read or produce messages). For the sake of concreteness, we model $\mathcal{D}$ as a Turing machine[12] $M_{\mathcal{D}}$, which has three tapes:

- The first tape of $M_{\mathcal{D}}$ at node v contains the information available to node v throughout the computation: its UID, its neighbors, and its input $x(v)$.
- On the second tape, $M_{\mathcal{D}}$ writes and receives messages. At the beginning of each round r, the messages that were sent to node v in round $r-1$ appear on this tape; during the round, $M_{\mathcal{D}}$ erases these messages and instead writes the messages that node v sends in round r. For simplicity, we assume in this overview that each message consists of a single bit. (In the full version of this paper, we allow messages to be of polynomial size.)
- The third tape is a work tape, and stores the current internal state of node v.

We denote by $\mathsf{cf}_{r,t}(v)$ the configuration of $M_{\mathcal{D}}$ at node v in the t-th computation step of round r. For each $t < T$, the configuration $\mathsf{cf}_{r,t+1}(v)$ is obtained from $\mathsf{cf}_{r,t}(v)$ by a computation step of $M_{\mathcal{D}}$, representing an internal computation step of node v. However, configuration $\mathsf{cf}_{r+1,1}(v)$ is obtained from $\mathsf{cf}_{r,T}(v)$ by writing on the first tape the messages that v's neighbors sent to node v in round r, as recorded in the third tape of their final round-r configurations, $\{\mathsf{cf}_{r,T}(u) : u \in N(v)\}$. This represents the receipt of these messages by node v at the end of round r.

We refer to the sequence $\mathsf{cf}_{0,0}(v), \ldots, \mathsf{cf}_{R,T}(v)$ as the *trace* of the computation at node v, and denote it by $\mathsf{Trace}(v)$.

Constructing the distributed argument. Fix a distributed algorithm $\mathcal{D}$ where each node executes a Turing machine $M_{\mathcal{D}}$, a network graph $G = (V, E)$, an input assignment $x : V \to \mathcal{X}$ and an output assignment $y : V \to \mathcal{Y}$. Let R, T be the number of rounds and the local computation time of $\mathcal{D}$, respectively. As it runs alongside the original algorithm $\mathcal{D}$, the prover records the execution of $\mathcal{D}$ at each node v: it stores the trace $\mathsf{Trace}(v) = \mathsf{cf}_{0,0}(v), \ldots, \mathsf{cf}_{R,T}(v)$ of the Turing machine $M_{\mathcal{D}}$ executed at node v, and the messages $\{m_r^{v\to u}, m_r^{u\to v}\}_{u\in N(v), r\in[R]}$ sent and received by v (respectively) on each edge $\{v, u\} \in E$ in each round r.[13]

After $\mathcal{D}$ terminates, the prover begins constructing the certificates. The first step is to compute a low-diameter edge cover of the network graph G, as described in Section 3.1. Let $S_1, \ldots, S_\ell \subseteq V$ be the resulting clusters, and for each node v, let $S(v) \subseteq \{1, \ldots, \ell\}$ be the indices of clusters to which node v belongs. In each cluster S_i, we compute a DMT of all messages sent over edges belonging to $G[S_i]$, as described above. In the sequel, we use the notation $(\cdot)_i(v)$ for the DMT associated with cluster i at node v; for example, $\mathsf{rt}_i(v)$ is the local root of node v in the DMT for cluster i.

[12] For simplicity, we assume that all nodes execute the same Turing machine, which takes the UID of the node as input. However, this is not essential; we could have each node v execute a different machine M_v.
[13] We believe that the space requirement of our prover can be reduced to have polylogarithmic overhead on top of the original algorithm $\mathcal{D}$, but this is technically non-trivial, and we defer it to future work.

The remainder of the prover's computation is local: each node uses the information it stored while $\mathcal{D}$ was running, and the DMTs that we constructed, to compute a certificate $\pi(v)$, consists of the following (see Figure 1 for an illustration).

- A hash with local openings $\mathsf{hTrace}(v)$ of the vector $(\mathsf{hCf}_{0,0}(v), \ldots, \mathsf{hCf}_{R,T}(v))$, where each $\mathsf{hCf}_{r,t}(v)$ is itself a hash with local openings of the configuration $\mathsf{cf}_{r,t}(v)$.
- The set $S(v)$ of clusters to which node v belongs.
- For each cluster $i \in S(v)$, the root rt_i of the DMT for cluster S_i, as well as the index and the opening from the root rt_i down to the local root $\mathsf{rt}_i(v)$, which hashes all messages sent by node v over edges belonging to cluster i.
- A BARG proof $\beta^{\mathsf{int}}(v)$ asserting that the *internal computation* of node v is correct, namely, that each configuration $\mathsf{cf}_{r,t+1}(v)$ in the trace of v is obtained from the preceding configuration $\mathsf{cf}_{r,t}(v)$ by a transition of $M_{\mathcal{D}}$.[14] This is a conjunction of $R \cdot (T-1)$ statements, with the (r,i)-th statement asserting (roughly) that there exist two hashed configurations $\mathsf{hCf}, \mathsf{hCf}'$ such that:
  - $\mathsf{hTrace}(v)$ opens to hCf in the index corresponding to step (r,t) of the computation, and to hCf' in the index corresponding to step $(r,t+1)$.
  - The configuration hashes hCf and hCf' are of successive configurations $\mathsf{cf}, \mathsf{cf}'$ (respectively), such that cf' is obtained from cf by one step of $M_{\mathcal{D}}$. This statement is delicate to prove, since it concerns the configurations "under the hash" and not the hashes $\mathsf{hCf}, \mathsf{hCf}'$ themselves (at least not directly), but it can be done using a technique from [29]. In short, it involves proving that the hashes $\mathsf{hCf}, \mathsf{hCf}'$ are of configurations that are only different in one location, and this could be done for a locally-openable hash.
- A BARG proof $\beta^{\mathsf{cons}}(v)$ asserting the consistency of the messages written in v's trace with the messages recorded in the DMTs to which v belongs. This is a conjunction of $R \cdot \widetilde{n}^2$ statements, where $\widetilde{n}$ is the size of the UID space: statement $(r,u,w) \in [R] \times [\widetilde{n}] \times [\widetilde{n}]$ asserts that *if* the edge (u,w) exists in the network, then for each of its ends $v \in \{u,w\}$, *the same* message is recorded in the appropriate index (corresponding to round r and edge (u,w)) of the DMT and trace of node v (which again is u or w).
  In more detail, we require that if the edge (u,w) exists and $v \in \{u,w\}$ is one of its ends, then there exist a message $m \in \{0,1\}$ and a configuration hash hCf such that:
  - The DMT for the cluster covering edge $\{u,w\}$ opens to m in the location corresponding to round r and directed edge (u,w),
  - hTrace opens to the configuration hash hCf in location (r,T) if v is the sender (i.e., $u=v$), or in location $(r+1,1)$ if v is the receiver (i.e., $w=v$), and
  - hCf opens to m in the location where the message sent/received on edge (u,v) is recorded.

  If the edge (u,w) does not exist, or is not adjacent to node v, then the statement (r,u,w) is simply *true* (i.e., it imposes no requirements). The mechanism for checking inside the BARG whether or not the edge (u,w) exists and touches node v is somewhat subtle, and we defer the details to the full version of this paper.

[14] Recall that the transition from step (r,T) to step $(r+1,1)$ involves receiving messages; it is not a local computation step. It must still be attested to, for example to ensure that the internal state of the machine does not change between these two steps, but we omit the details here.

We note that despite the fact that our construction uses multiple local DMTs, the argument presented above is simpler than the argument constructed using the global DMT in [2]: separating the requirements into *internal correctness* and *message consistency*, and creating a separate BARG for each, simplifies both the structure of the argument and the proof of its soundness.

For technical reasons related to the proof of soundness, we actually need two copies of each BARG: $\beta^{\mathsf{int}}(v)^j$ and $\beta^{\mathsf{cons}}(v)^j$, for $j \in \{1,2\}$. Each of the four BARGs uses its own crs, and we will see that this helps us "catch a cheating prover in a lie". This is discussed in Section 3.3 below.

Verifying the certificates. At verification time, each node v informs its neighbors of the clusters $S(v)$ to which it belongs, and also sends a collection $\{(\rho_c(v), I_c(v)) : c \in S(v)\}$ consisting of v's local root and index inside the local DMT for each cluster to which v belongs. This allows each neighbor $u \in N(v)$ to compute the location in the DMT of each message sent on the edge (v,u).

Next, each node v verifies the four BARGs, $\beta^{\mathsf{int}}(v)^j$ and $\beta^{\mathsf{cons}}(v)^j$ for $j = 1,2$, stored in its certificate $\pi(v)$. At this point it has all the information needed to do so. If the BARG verification succeeds, node v outputs *accept*, and otherwise it outputs *reject*.

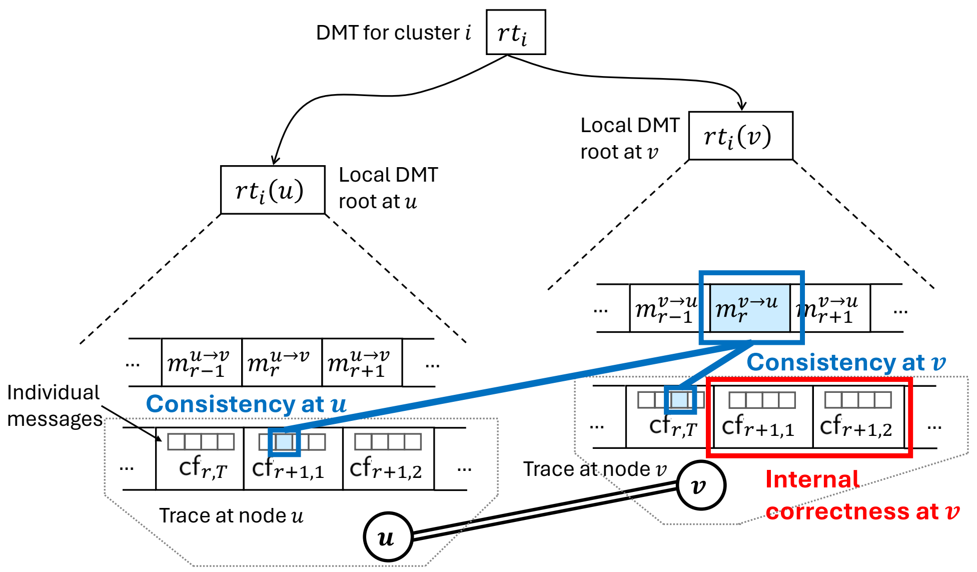

Figure 1 The figure shows the DMT for the cluster i that covers edge $\{u,v\}$, and "under the hash", the messages sent from node u to node v and vice-versa, under the respective local roots $\mathsf{rt}_i(u), \mathsf{rt}_i(v)$. The figure also shows the trace at each node, "under the hash". Inside each configuration, small boxes indicate messages written on the second tape. In configuration $\mathsf{cf}_{r,T}(v)$, these are the messages sent by node v in round r; in configuration $\mathsf{cf}_{r+1,1}(v)$, these are the messages received by node v in round r (and similarly for node u). The internal correctness BARG at node v (in red) asserts that each configuration $\mathsf{cf}_{r,t+1}(v)$ is the successor to $\mathsf{cf}_{r,t}(v)$ according to $M_{\mathcal{D}}$. The consistency BARGs at node v and at node u (in blue) together assert that each message hashed inside the DMT matches the corresponding messages in the traces of u and of v.

3.3 The Soundness of Our Construction

In this section, we give the main ideas for our proof of computational soundness.

Fix a distributed algorithm $\mathcal{D}$, and let $\mathcal{L}_{\mathcal{D}}$ be the language of all annotated graphs (G, x, y) such that when $\mathcal{D}$ executes in the network G with input assignment x, the output it produces is y. Let $(\mathsf{Gen}, \mathcal{P}, \mathcal{V})$ be our fl-DSNARG for the language $\mathcal{L}_{\mathcal{D}}$, as described above.

Recall that computational soundness requires that no poly-size adversary can fool the verifier into accepting the proof of an incorrect statement, except with negligible probability (in the security parameter and in the size of the graph). We capture this requirement in the form of the following experiment, which we call ExpSound, where a poly-size adversary $\mathcal{A}$ tries to break the soundness of the argument:

- A crs is sampled by calling the trusted setup procedure Gen of the fl-DSNARG. In our construction, several of the primitives that we use require a common reference string: the DMT uses a CRS to select a hash function, and the BARGs use their own internal hash functions as well. The Gen procedure of our fl-DSNARG instantiates these common reference strings by calling the Gen procedures of the respective primitives, and returns one value, crs, consisting of all of them together.
- The adversary $\mathcal{A}$ is given crs, and outputs an annotated graph (G, x, y), and a certificate assignment π to the nodes of G.

We say that $\mathcal{A}$ *wins* the experiment if it can produce a network G, an input x and output y such that the algorithm $\mathcal{D}$ does *not* output y on (G, x), and a certificate assignment π that convinces all nodes to accept, nonetheless. If there is a *poly-size* adversary that can win the experiment with non-negligible probability, then soundness is broken.

To prove the soundness of our argument we assume towards contradiction that *there is* a poly-size adversary $\mathcal{A}$ that can win experiment ExpSound. We use $\mathcal{A}$ to construct a poly-size adversary $\mathcal{A}'$ that breaks the soundness of one of our building blocks: the *collision-resistance with respect to openings* property of the hash family, the *index-hiding* property of the BARG, or the *somewhere argument of knowledge* property of the BARG. Since we assume that these properties hold for the primitives we use, this is a contradiction.

We consider each computation step $(r, t) \in [R] \times [T]$ of the distributed algorithm $\mathcal{D}$, and define an experiment $\mathsf{ExpSound}_{r,t}$, which is the same as ExpSound, except that the crs for the two BARGs $\beta^{\mathsf{int}}(v)^1$ and $\beta^{\mathsf{cons}}(v)^1$ is generated in trapdoor mode, binding the crs to index (r, t), while the other two copies, $\beta^{\mathsf{int}}(v)^2$ and $\beta^{\mathsf{cons}}(v)^2$, are set up in regular mode (without a trapdoor). By the *index-hiding* property of the BARG, no poly-size adversary can tell whether the Gen procedure is called in regular mode or in trapdoor mode; therefore, our cheating adversary $\mathcal{A}$ wins the new experiment $\mathsf{ExpSound}_{r,t}$ with almost the same probability that it wins the original experiment, ExpSound, where all four BARGs were set up in regular mode. (If the probability was noticeably different, then we could break the index-hiding property by running $\mathcal{A}$ and checking whether it wins. The noticeable difference between the winning probability for ExpSound and for $\mathsf{ExpSound}_{r,t}$ translates to a noticeable advantage in guessing whether crs was generated in trapdoor mode or not.)

Next we use the *somewhere argument of knowledge* property of the BARG to claim that whenever $\mathcal{A}$ wins the experiment $\mathsf{ExpSound}_{r,t}$, we can use the trapdoor associated with the binding index (r, t) to extract NP-witnesses $w^{\mathsf{int}}_{r,t}(v), w^{\mathsf{cons}}_{r,t}(v)$ to the (r, t)-th statement of the BARGs $\beta^{\mathsf{int}}(v)^1$ and $\beta^{\mathsf{cons}}(v)^1$, again with a very close probability to the original winning probability of $\mathcal{A}$. These witnesses are accepted by the circuit of the respective BARGs.

We would now like to argue that these witnesses reflect the *true* state of the distributed algorithm after the t-th computation step of round r: that is, they match the witnesses that would be generated by an honest prover $\mathcal{P}$, and contain, e.g., the true hash values of internal configurations and messages that the algorithm $\mathcal{D}$ generates at this point in its computation. We will then use the *collision resistance to openings* property of the hash family to reach a contradiction. If we could claim this for *every* $r \in [R]$ and $t \in [T]$, then in particular it would be true for the final state of the network, in step (R, T), where the output y is produced. Since the output is encoded in the internal configuration of the network nodes, whenever the adversary $\mathcal{A}$ wins $\mathsf{ExpSound}_{R,T}$, we can use it to find a collision in the hash of the internal configurations: if $\mathcal{A}$ wins, then for some node $v \in V$, the output $y(v)$ produced by $\mathcal{A}$ does not match the true output $y'(v)$ of the algorithm $\mathcal{D}$. The witness $w^{\mathsf{int}}_{R,T}(v)$ contains a hash $\mathsf{hCf}_{R,T}(v)$ of the false final configuration $\mathsf{cf}_{R,T}(v)'$, which includes the false output $y(v)$. But we know that this witness matches what the honest prover would produce, that is, the hash of the true final configuration $\mathsf{cf}_{R,T}(v)$, including the true output $y'(v)$. Thus, the true configuration $\mathsf{cf}_{R,T}(v)$ and the false configuration $\mathsf{cf}_{R,T}(v)'$ hash to the same value, $\mathsf{hCf}_{R,T}(v)$, and we found a collision.

To prove that the witnesses extracted from the certificates in each experiment $\mathsf{ExpSound}_{r,t}$ are the true witnesses that would be generated by the honest prover, we define *hybrid experiments* $\left\{\mathsf{ExpSound}'_{r,t}\right\}_{(r,t)\in[R]\times[T]}$, where we use two trapdoors: the first two copies of the BARGs are set up with a binding index of (r, t), while the second two copies are set up with a binding index of $(r, t+1)$. The winning condition for experiment $\mathsf{ExpSound}'_{r,t}$ requires the adversary to output certificates $\pi(v)$ at each node v such that

- All certificates are accepted.
- For each node, upon extracting the witnesses for indices (r, t) and $(r, t + 1)$ from the respective BARGs, all four witnesses are accepted by the respective BARG circuits.[15]
- For each node, the witnesses for index (r, t) are the true witnesses that would be generated by the honest prover. And finally,
- There exists a node where the witnesses for index $(r, t + 1)$ are *not* the true witnesses that would be generated by the honest prover.

Winning this experiment with non-negligible probability again breaks the index-hiding property of the BARG, because it essentially means that the adversary can tell whether the binding index is (r, t), in which case it produces true witnesses matching the honest prover at all nodes, or $(r, t + 1)$, in which case it produces a false witness at some node. Proving this step also relies on the fact that the witnesses are accepted by the BARG circuit, which asserts that the transition from step (r, t) to step $(r, t + 1)$ is legal. This means that if the witness for step (r, t) is the true witness, then either the witness for step $(r, t + 1)$ is also the true witness, or we have broken the *somewhere proof of knowledge* property of the BARG (it accepts, despite the extraction of an inappropriate witness).

After proving that the adversary cannot win experiment $\mathsf{ExpSound}'_{r,t}$ except with negligible probability, we chain together the entire sequence $\mathsf{ExpSound}'_{1,1}, \ldots, \mathsf{ExpSound}'_{R,T}$ and argue that since the adversary does not win *any* of these experiments with non-negligible probability, either it produces false witnesses *for the initial state of the network*, or it produces true witnesses for all computation steps (in which case we are done, as we explained above). However, the prover cannot lie about the initial state of the network without breaking collision resistance, for reasons similar to those we outlined for the final configuration.

[15] Recall that the BARG circuit is simply the circuit that verifies (i, w_i), not to be confused with the BARG verifier, which verifies the BARG proof.

References

1 Miklós Ajtai. Generating hard instances of lattice problems. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, pages 99–108, 1996.

2 Eden Aldema Tshuva, Elette Boyle, Ran Cohen, Tal Moran, and Rotem Oshman. Locally verifiable distributed snargs. In *Theory of Cryptography Conference*, pages 65–90. Springer, 2023.

3 Scott Ames, Carmit Hazay, Yuval Ishai, and Muthuramakrishnan Venkitasubramaniam. Ligero: Lightweight sublinear arguments without a trusted setup. In *Proceedings of the 2017 acm sigsac conference on computer and communications security*, pages 2087–2104, 2017. doi:10.1145/3133956.3134104.

4 B. Awerbuch, B. Patt-Shamir, and G. Varghese. Self-stabilization by local checking and correction. In *Proceedings 32nd Annual Symposium of Foundations of Computer Science*, pages 268–277, 1991.

5 B. Awerbuch and M. Sipser. Dynamic networks are as fast as static networks. In *[Proceedings 1988] 29th Annual Symposium on Foundations of Computer Science*, pages 206–219, 1988.

6 Alkida Balliu, Juho Hirvonen, Darya Melnyk, Dennis Olivetti, Joel Rybicki, and Jukka Suomela. Local mending. In Merav Parter, editor, *Structural Information and Communication Complexity*, pages 1–20, 2022. doi:10.1007/978-3-031-09993-9_1.

7 Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Fast reed-solomon interactive oracle proofs of proximity. In *45th international colloquium on automata, languages, and programming (icalp 2018)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2018.

8 Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable, transparent, and post-quantum secure computational integrity. *Cryptology ePrint Archive*, 2018.

9 Aviv Bick, Gillat Kol, and Rotem Oshman. Distributed zero-knowledge proofs over networks. In *SODA*, pages 2426–2458. SIAM, 2022. doi:10.1137/1.9781611977073.97.

10 Jonathan Bootle, Andrea Cerulli, Essam Ghadafi, Jens Groth, Mohammad Hajiabadi, and Sune K Jakobsen. Linear-time zero-knowledge proofs for arithmetic circuit satisfiability. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 336–365. Springer, 2017. doi:10.1007/978-3-319-70700-6_12.

11 Jonathan Bootle, Alessandro Chiesa, and Jens Groth. Linear-time arguments with sublinear verification from tensor codes. In *Theory of Cryptography: 18th International Conference, TCC 2020, Durham, NC, USA, November 16–19, 2020, Proceedings, Part II 18*, pages 19–46. Springer, 2020. doi:10.1007/978-3-030-64378-2_2.

12 Keren Censor-Hillel, Ami Paz, and Mor Perry. Approximate proof-labeling schemes. *Theoretical Computer Science*, 811:112–124, 2020. doi:10.1016/J.TCS.2018.08.020.

13 Yi-Jun Chang and Mohsen Ghaffari. Strong-diameter network decomposition. In *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*, pages 273–281, 2021. doi:10.1145/3465084.3467933.

14 Arka Rai Choudhuri, Sanjam Garg, Abhishek Jain, Zhengzhong Jin, and Jiaheng Zhang. Correlation intractability and SNARGs from sub-exponential DDH. In *Proceedings of the 43rd Annual International Cryptology Conference, CRYPTO 2023, Part IV*, volume 14084 of *LNCS*, pages 635–668. Springer, 2023. doi:10.1007/978-3-031-38551-3_20.

15 Arka Rai Choudhuri, Abhishek Jain, and Zhengzhong Jin. Non-interactive batch arguments for NP from standard assumptions. In *Proceedings of the 41st Annual International Cryptology Conference, CRYPTO 2021, Part IV*, volume 12828 of *LNCS*, pages 394–423. Springer, 2021. doi:10.1007/978-3-030-84259-8_14.

16 Arka Rai Choudhuri, Abhishek Jain, and Zhengzhong Jin. SNARGs for P from LWE. In *62nd IEEE Annual Symposium on Foundations of Computer Science (FOCS)*, pages 68–79, 2021.

17 Pierluigi Crescenzi, Pierre Fraigniaud, and Ami Paz. Trade-offs in distributed interactive proofs. In *DISC*, volume 146 of *LIPIcs*, pages 13:1–13:17. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPICS.DISC.2019.13.

18 Ivan Bjerre Damgård. Collision free hash functions and public key signature schemes. In *Workshop on the Theory and Application of of Cryptographic Techniques*, pages 203–216. Springer, 1987.

19 Shlomi Dolev. *Self-Stabilization*. MIT Press, 2000.

20 Yuval Emek, Yuval Gil, and Shay Kutten. Locally Restricted Proof Labeling Schemes. In *36th International Symposium on Distributed Computing (DISC 2022)*, volume 246, pages 20:1–20:22, 2022. doi:10.4230/LIPICS.DISC.2022.20.

21 Laurent Feuilloley, Pierre Fraigniaud, Juho Hirvonen, Ami Paz, and Mor Perry. Redundancy in distributed proofs. *Distributed Comput.*, 34(2):113–132, 2021. doi:10.1007/S00446-020-00386-Z.

22 Pierre Fraigniaud, Pedro Montealegre, Rotem Oshman, Ivan Rapaport, and Ioan Todinca. On distributed Merlin-Arthur decision protocols. In *SIROCCO*, volume 11639 of *LNCS*, pages 230–245. Springer, 2019. doi:10.1007/978-3-030-24922-9_16.

23 Pierre Fraigniaud, Boaz Patt-Shamir, and Mor Perry. Randomized proof-labeling schemes. *Distributed Computing*, 32:217–234, 2019. doi:10.1007/S00446-018-0340-8.

24 Alexander Golovnev, Jonathan Lee, Srinath Setty, Justin Thaler, and Riad S Wahby. Brakedown: Linear-time and field-agnostic snarks for r1cs. In *Annual International Cryptology Conference*, pages 193–226. Springer, 2023. doi:10.1007/978-3-031-38545-2_7.

25 Mika Göös and Jukka Suomela. Locally checkable proofs in distributed computing. *Theory Comput.*, 12(1):1–33, 2016. doi:10.4086/TOC.2016.V012A019.

26 Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. Cryptography with constant computational overhead. In *Proceedings of the fortieth annual ACM symposium on Theory of computing*, pages 433–442, 2008. doi:10.1145/1374376.1374438.

27 Yael Kalai, Alex Lombardi, Vinod Vaikuntanathan, and Daniel Wichs. Boosting batch arguments and RAM delegation. In *Proceedings of the 55th Annual ACM Symposium on Theory of Computing (STOC)*, pages 1545–1552, 2023. doi:10.1145/3564246.3585200.

28 Yael Tauman Kalai, Alex Lombardi, and Vinod Vaikuntanathan. Snargs and ppad hardness from the decisional diffie-hellman assumption. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 470–498. Springer, 2023. doi:10.1007/978-3-031-30617-4_16.

29 Yael Tauman Kalai, Omer Paneth, and Lisa Yang. How to delegate computations publicly. In *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing, STOC 2019*, pages 1115–1124. ACM, 2019. doi:10.1145/3313276.3316411.

30 Gillat Kol, Rotem Oshman, and Raghuvansh R. Saxena. Interactive distributed proofs. In *Symposium on Principles of Distributed Computing (PODC)*, pages 255–264, 2018. URL: https://dl.acm.org/citation.cfm?id=3212771.

31 Michael König and Roger Wattenhofer. On local fixing. In *Principles of Distributed Systems*, pages 191–205. Springer International Publishing, 2013. doi:10.1007/978-3-319-03850-6_14.

32 Amos Korman and Shay Kutten. Distributed verification of minimum spanning trees. In *Proceedings of the twenty-fifth annual ACM symposium on Principles of distributed computing*, pages 26–34, 2006. doi:10.1145/1146381.1146389.

33 Amos Korman and Shay Kutten. Distributed verification of minimum spanning trees. In *Proceedings of the twenty-fifth annual ACM symposium on Principles of distributed computing*, pages 26–34, 2006. doi:10.1145/1146381.1146389.

34 Amos Korman, Shay Kutten, and David Peleg. Proof labeling schemes. In *Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing*, pages 9–18, 2005. doi:10.1145/1073814.1073817.

35 Jonathan Lee, Srinath Setty, Justin Thaler, and Riad Wahby. Linear-time and post-quantum zero-knowledge snarks for r1cs. *Cryptology ePrint Archive*, 2021.

36 Christoph Lenzen, Jukka Suomela, and Roger Wattenhofer. Local algorithms: Self-stabilization on speed. In *Stabilization, Safety, and Security of Distributed Systems*, pages 17–34, 2009. `doi:10.1007/978-3-642-05118-0_2`.

37 Nathan Linial and Michael Saks. Low diameter graph decompositions. *Combinatorica*, 13(4):441–454, 1993. `doi:10.1007/BF01303516`.

38 Ralph C. Merkle. A certified digital signature. In *Proceedings of the 9th Annual International Cryptology Conference, CRYPTO '89*, volume 435 of *LNCS*, pages 218–238. Springer, 1989. `doi:10.1007/0-387-34805-0_21`.

39 Silvio Micali. Computationally sound proofs. *SIAM Journal on Computing*, 30(4):1253–1298, 2000. `doi:10.1137/S0097539795284959`.

40 Pedro Montealegre, Diego Ramírez-Romero, and Ivan Rapaport. Shared vs private randomness in distributed interactive proofs. *arXiv preprint arXiv:2006.16191*, 2020. `arXiv:2006.16191`.

41 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. `doi:10.1137/1.9781611975994.67`.

42 Rafail Ostrovsky, Mor Perry, and Will Rosenbaum. Space-time tradeoffs for distributed verification. In *International Colloquium on Structural Information and Communication Complexity*, pages 53–70. Springer, 2017. `doi:10.1007/978-3-319-72050-0_4`.

43 Omer Paneth and Rafael Pass. Incrementally verifiable computation via rate-1 batch arguments. In *2022 IEEE 63rd Annual Symposium on Foundations of Computer Science (FOCS)*, pages 1045–1056. IEEE, 2022. `doi:10.1109/FOCS54457.2022.00102`.

44 Noga Ron-Zewi and Ron D Rothblum. Proving as fast as computing: succinct arguments with constant prover overhead. In *Proceedings of the 54th Annual ACM SIGACT Symposium on Theory of Computing*, pages 1353–1363, 2022. `doi:10.1145/3519935.3519956`.

45 Václav Rozhoň and Mohsen Ghaffari. Polylogarithmic-time deterministic network decomposition and distributed derandomization. In *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing*, pages 350–363, 2020.

46 Paul Valiant. Incrementally verifiable computation or proofs of knowledge imply time/space efficiency. In *Theory of Cryptography Conference*, pages 1–18. Springer, 2008. `doi:10.1007/978-3-540-78524-8_1`.

47 Michael Walfish and Andrew J Blumberg. Verifying computations without reexecuting them. *Communications of the ACM*, 58(2):74–84, 2015. `doi:10.1145/2641562`.

48 Brent Waters and David J. Wu. Batch arguments for NP and more from standard bilinear group assumptions. In *Proceedings of the 42nd Annual International Cryptology Conference, CRYPTO 2022, Part II*, volume 13508 of *LNCS*, pages 433–463. Springer, 2022. `doi:10.1007/978-3-031-15979-4_15`.

49 Tiacheng Xie, Jiaheng Zhang, Yupeng Zhang, Charalampos Papamanthou, and Dawn Song. Libra: Succinct zero-knowledge proofs with optimal prover computation. In *Advances in Cryptology–CRYPTO 2019: 39th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18–22, 2019, Proceedings, Part III 39*, pages 733–764. Springer, 2019. `doi:10.1007/978-3-030-26954-8_24`.

50 Jiaheng Zhang, Tianyi Liu, Weijie Wang, Yinuo Zhang, Dawn Song, Xiang Xie, and Yupeng Zhang. Doubly efficient interactive proofs for general arithmetic circuits with linear prover time. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 159–177, 2021. `doi:10.1145/3460120.3484767`.

A Full Syntax, Formal Definitions and Statements

A.1 Distributed Merkle Trees

We give here the full definition of a distributed Merkle tree, adopted from [2], with a minor change we discuss bellow.

Syntax. An efficient distributed Merkle tree DMT is associated with a recursive hash family with local openings

$$\mathsf{MT} = (\mathsf{MT.Gen}, \mathsf{MT.Hash}, \mathsf{MT.Open}, \mathsf{MT.Verify})$$

and consists of the following algorithms:

$\mathsf{Gen}(1^\lambda) \to \mathsf{hk}$. A randomized algorithm that takes as input the security parameter λ and outputs a hash key $\mathsf{hk} = \mathsf{MT.Gen}(1^\lambda)$.

$\mathsf{DistMake}(\mathsf{hk}; G; x) \to \{(\mathsf{val}_v, \mathsf{rt}_v, I_v, \rho_v, \beta_v)\}_{v \in V(G)}$. A distributed algorithm that executes in a distributed network G, with all nodes receiving the same hash key hk, and each node $v \in V(G)$ initially holding a collection of inputs $x(v) = \{x_{v\to u}\}_{u \in N(v)}$ (one input $x_{u\to v}$ for each neighbor $u \in N(v)$). The output at each node v consists of:
- A hash value val_v, which is the same at all nodes,
- A local MT-root rt_v,[16]
- An index $I_v \in \{0,1\}^*$,
- An opening path ρ_v, and
- A set β_v of openings $(\rho_{v\to u})$ of index and opening path for every neighbor $u \in N(v)$.

▶ **Definition 2** (DMT). *A* DMT *is required to satisfy the following properties:*

Well-formedness.
- *All nodes $v \in V(G)$ output the same value val_v,*
- *All indices I_v are of length $c \cdot \lceil \log n \rceil$, for some constant c,*

MT-***functionality.*** *Fix a hash key hk, a network G of size n and input assignment to it $x : V(G) \to \{0,1\}^*$, where for every $v \in V(G)$, $x(v) = \{x_{v\to u}\}_{u\in N(v)}$, such that for every edge $\{v,u\} \in E(G)$, $x_{v\to u} \in \{0,1\}^\ell$. Let*

$$\left\{(\mathsf{val}_v, \mathsf{rt}_v, I_v, \rho_v, p_v, \hat{F}_v, \beta_v)\right\}_{v\in V(G)} = \mathsf{DistMake}(\mathsf{hk}, G, x),$$

where $\beta_v = \{\rho_{v\to u}\}_{u\in N(v)}$. For each directed edge (v,u), let $\mathsf{Index}(v,u) = I_v \parallel \mathsf{id}(u)$, and $\mathsf{Opening}(v,u) = \rho_v \parallel \rho_{v\to u}$. We say that the DMT *satisfies* MT-functionality *if for every such output, there exists a constant c and a vector $\vec{x}$ of length at most $\leq 2^{c\cdot\lceil\log n\rceil + \lceil \log \widetilde{n}\rceil + \lceil \log \ell \rceil}$ (where $\widetilde{n}$ denotes the size of the UID domain) such that:*
- *For every $v \in V(G)$ and $u \in N(v)$ we have $\vec{x}_{\mathsf{Index}(v,u)} = x_{v\to u}$,*
- *For every $v \in V(G)$, $\mathsf{val}_v = \mathsf{MT.Hash}(\mathsf{hk}, \vec{x})$,*
- *For every $v \in V(G)$ and $u \in N(v)$ we have:* $(\vec{x}_{v\to u}, \mathsf{Opening}(v,u)) = \mathsf{MT.Open}(\mathsf{hk}, \vec{x}, \mathsf{Index}(v,u))$.

Efficiency. *At each node, the local computation executed by* $\mathsf{DistMake}$ *runs in time* $\mathrm{poly}(\lambda, n, m)$.

[16] Throughout this section and the sequel, we use both val and rt to denote MT-values, which are also themselves MT-roots (the construction is recursive). We use val to denote a "final" value, the root of the entire network, which is later exposed to the algorithm using the DMT; we typically use rt for intermediate values handled inside the distributed Merkle.

Low round complexity and low communication complexity. DistMake *runs in* $O(D)$ *synchronized communication rounds on networks of diameter* D*, and uses messages of length* $\mathrm{poly}(\lambda, \log n)$*.*

▶ Remark 3. In [2], the set β_v returned from the algorithm DistMake also contain indices $\{I_v \parallel I_{v \to u}\}_{u \in N(v)}$, and the MT-functionality property is defined with respect to $\mathsf{Index}(v, u) = I_v \parallel I_{v \to u}$, where $I_v \parallel I_{v \to u}$ is the port number of u as represented in the node v. In this work, we simplify this by considering the UIDs instead of port numbers of the nodes. This means that the hash value val now depends on the size of the UID domain $\widetilde{n} = |\mathcal{U}|$, where it used to depend on the maximal degree, but this does not come with a meaningful cost as (1) we assume $\widetilde{n} = \mathrm{poly}(n)$ and (2) the dependency (of previously Δ and now $\widetilde{n}$) is *logarithmic*.

▶ **Theorem 4** ([2]). *For every recursive hash family with local openings, there exists a respective distributed Merkle tree.*

A.2 Somewhere Extractable Batch Arguments (seBARGs)

Syntax. A seBARG for index language consists of the following algorithms:

$\mathsf{Gen}(1^\lambda, k, 1^s, i) \to (\mathsf{crs}, \mathsf{td})$**.** A randomized setup procedure that takes a security parameter λ, the number of statements k, the size of the circuit 1^s, and an optional index i, and generates a common reference string crs and if provided an index i, a trapdoor td.

$\mathcal{P}(\mathsf{crs}, C, w_1, \ldots, w_k) \to (b, \pi)$**.** A polynomial-time prover algorithm that takes the crs, a circuit C and a list of witnesses $w_1, \ldots, w_k$, and outputs a bit b and a proof π.

$\mathcal{V}(\mathsf{crs}, C, \pi) \to b$**.** A polynomial-time verification algorithm that takes the crs, a circuit C, and a proof π and outputs an acceptance bit.

$\mathcal{E}(\mathsf{td}, C, \pi) \to w_i$**.** A polynomial-time extraction algorithm that takes a trapdoor td, a circuit C, and a proof π, and outputs a witness w_i.

▶ **Definition 5** (seBARG). *A* seBARG *satisfies the following requirements.*

Succinctness. *The length of the* crs *and of the proof* π *is at most* $\mathrm{poly}(s, \lambda, \log k)$*.*

Verifier Efficiency. *The verifier runs in time* $\mathrm{poly}(s, \lambda, \log k)$*.*

Completeness. *For any* $\lambda \in \mathbb{N}$ *and* $s = s(\lambda)$ *of size at most* 2^λ*, for any circuit* $C : [k] \times \{0,1\}^m \to \{0,1\}$ *of size at most* s*, any witnesses* $w_1, \ldots, w_k \in \{0,1\}^m$ *and any index* $i^* \in [k]$

$$\Pr\left[\ \mathcal{V}(\mathsf{crs}, C, \pi) = 1 \ \middle|\ \begin{array}{l} (\mathsf{crs}, \mathsf{td}) \leftarrow \mathsf{Gen}(1^\lambda, k, 1^s, i^*) \\ \pi \leftarrow \mathcal{P}(\mathsf{crs}, C, w_1, \ldots, w_k) \end{array} \right] = 1.$$

Index hiding. *For any poly-size adversary* $\mathcal{A}$ *and polynomials* $k = k(\lambda)$ *and* $s = s(\lambda)$*, there exists a negligible function* $\mathrm{negl}(\cdot)$ *such that for every* $\lambda \in \mathbb{N}$

$$\Pr\left[\begin{array}{l} i_0, i_1 \in [k] \\ \mathcal{A}(\mathsf{crs}) = b \end{array} \ \middle|\ \begin{array}{l} (i_0, i_1) \leftarrow \mathcal{A}(1^\lambda) \\ b \leftarrow \{0,1\} \\ (\mathsf{crs}, \mathsf{td}) \leftarrow \mathsf{Gen}(1^\lambda, k, 1^s, i_b) \end{array} \right] \le \frac{1}{2} + \mathrm{negl}(\lambda).$$

Somewhere argument of knowledge. *For any poly-size adversary $\mathcal{A}$, polynomials $k = k(\lambda)$ and $s = s(\lambda)$, and index $i^* = i^*(\lambda) \in [k(\lambda)]$, there exists a negligible function* $\mathrm{negl}(\cdot)$ *such that for every $\lambda \in \mathbb{N}$*

$$\Pr\left[\begin{array}{c|l} \mathcal{V}(\mathsf{crs}, C, \pi) = 1 & (\mathsf{crs}, \mathsf{td}) \leftarrow \mathsf{Gen}(1^\lambda, k, 1^s, i^*) \\ \wedge\ C(i^*, w) = 0 & (C, \pi) \leftarrow \mathcal{A}(\mathsf{crs}) \\ & w \leftarrow \mathcal{E}(\mathsf{td}, C, \pi) \end{array}\right] \leq \mathrm{negl}(\lambda).$$

▶ **Theorem 6** ([15, 48, 27, 14]). seBARG*s for* NP*, and in particular, for the index languages, exist assuming either: (1)* LWE*, (2)* DLIN*, or (3) subexponential* DDH*.*

A.3 Low-Diameter Edge Cover

For a graph $G = (V, E)$ and a mapping S from V to subsets of $\mathcal{U}$, denote by T_S the image of S (that is, the set $\{t \in \mathcal{U} \mid \exists v \in V : \ t \in S(v)\}$), and for every $t \in T_S$, denote by V_t the set of nodes which have t in their image: $V_t^S = \{v \in V \mid t \in S(v)\}$.

▶ **Definition 7** ((D, s)-edge-cover). *For a graph G, we say a mapping $S : V \to \mathcal{U}$ is an edge cover of G if for every edge $\{v, u\} \in E$, we have $S(v) \cap S(u) \neq \emptyset$.*

We say S is diameter-D if for every $t \in T$, we have that the graph induced by V_t^S, $G[V_t]$ is of strong-diameter at most D.

We say S is s-succinct if for every node $v \in V$, we have $|S(v)| \leq s$.

▶ Remark 8. We remark that unlike the classical definition of graph decomposition, here we think of the clusters from the point of view of the nodes; and for that reason define the edge-cover to be a mapping from nodes to all of the sets it belongs to, rather than simply a set of subsets of the graph nodes.

▶ **Theorem 9.** *There exists a* $(\mathrm{polylog}(n), \mathrm{polylog}(n))$*-edge-cover algorithm in the* CONGEST *model.*

A.4 Fully Local Distributed SNARG

We give here the full definition of a *fully local distributed* SNARG (fl-DSNARG), which is mostly adopted from [2], with the only difference being the improved efficiency requirement from the prover.

Syntax. A locally verifiable distributed SNARG with a round-efficient distributed prover for a distributed algorithm $\mathcal{D}$ and corresponding graph language $\mathcal{L}_{\mathcal{D}}$ consists of the following algorithms.

$\mathsf{Gen}(1^\lambda, n) \to \mathsf{crs}$. A randomized algorithm that takes as input a security parameter 1^λ and a graph size n, and outputs a common reference string crs.

$\mathcal{P}(\mathsf{crs}; G; x) \to (y, \pi)$. A *distributed algorithm* that runs in the network G, where all of the nodes have access to the common reference string crs obtained from Gen, and each node $v \in V(G)$ inputs $x(v)$, and outputs (1) an assignment of outputs $y : V(G) \to \{0,1\}^*$ of $\mathcal{D}$ when executed in G, and (2) an assignment of proofs $\pi : V(G) \to \{0,1\}^*$.

$\mathcal{V}(\mathsf{crs}; G; x, \pi) \to b$. A *distributed decision algorithm* that takes as a common input to the entire network a common reference string crs, executes in the network G, where each node $v \in V(G)$ is assigned with an input $x(v)$ and a proof $\pi(v)$, and outputs acceptance bits $b : V \to \{0,1\}^*$.

▶ **Definition 10** (fl-DSNARG). *Let $\mathcal{D}$ be a distributed algorithm, and let $\mathcal{L}_\mathcal{D}$ be its corresponding graph language. An* fl-DSNARG $(\mathsf{Gen}, \mathcal{P}, \mathcal{V})$ *for $\mathcal{D}$ must satisfy the following properties:*

Completeness. *For any $(G, x) \in \mathcal{L}_\mathcal{D}$,*

$$\Pr\left[\ \mathcal{V}(\mathsf{crs}; G; x, \pi) = 1 \ \middle|\ \begin{array}{l} \mathsf{crs} \leftarrow \mathsf{Gen}(1^\lambda, n) \\ \pi \leftarrow \mathcal{P}(\mathsf{crs}; G; x) \end{array}\right] = 1.$$

Soundness. *For any poly-size algorithm $\mathcal{P}^*$ and polynomial $n = n(\lambda)$, there exists a negligible function* $\mathrm{negl}(\cdot)$ *such that*

$$\Pr\left[\ \begin{array}{l} (G, x) \notin \mathcal{L}_\mathcal{D} \\ \wedge\ \mathcal{V}(\mathsf{crs}; G; x, \pi) = 1 \end{array} \ \middle|\ \begin{array}{l} \mathsf{crs} \leftarrow \mathsf{Gen}(1^\lambda, n) \\ (G, x, \pi) \leftarrow \mathcal{P}^*(\mathsf{crs}) \end{array}\right] \le \mathrm{negl}(\lambda).$$

Succinctness. *The* crs *and the proof $\pi(v)$ at each node v are of length at most* $\mathrm{poly}(\lambda, \log n)$.

Verifier efficiency. *$\mathcal{V}$ runs in a single synchronized communication round, during which each node sends a (possibly different) message of length* $\mathrm{poly}(\lambda, \log n)$ *to each neighbor. At each node v, the local computation executed by $\mathcal{V}$ runs in time* $\mathrm{poly}(\lambda, |\pi(v)|, |x(v)|, \deg(v)) = \mathrm{poly}(\lambda, n)$.

Prover efficiency. *$\mathcal{P}$ adds an overhead of* $\mathrm{polylog}(n)$ *communication rounds to the rounds of $\mathcal{D}$, where in each of these rounds, each node sends a message of length* $\mathrm{poly}(\lambda, \log n)$ *to each neighbor. At each node, the local computation executed by $\mathcal{P}$ runs in time* $\mathrm{poly}(\lambda, n)$.

The following theorem states the existence of fl-DSNARG, assuming the existence of the ingredients we used, to complement Theorem 1.

▶ **Theorem 11.** *Assume the existence of a (D, c)-edge-cover algorithm in the* CONGEST *model, a distributed Merkle tree, and a somewhere extractable argument of knowledge for* NP.

Then, for every distributed algorithm $\mathcal{D}$ that runs in polynomial rounds and local computation time, there exists an fl-DSNARG.

B G^2 Strong-Diameter Decomposition in the CONGEST Model

We require a $(\mathrm{polylog}(n), \mathrm{polylog}(n))$-decomposition algorithm that satisfies the following properties:

- It is a *strong-diameter* decomposition algorithm,
- it is in the CONGEST model, and
- it can be extended to graph powers while remaining in the CONGEST model.

While the first two requirements are rather obvious, the last one may seem trivial given the second requirement, but it is in fact more delicate. It is true that given an algorithm in the LOCAL model, to simulate its execution on G^2, is rather simple; each node could start by collecting its distance-2 neighborhood and then simulate each step of the original algorithm as if it was operating on G^2, while suffering a factor of 2 in the number of rounds. However, this does not generally work in the CONGEST model, as the distance-2 neighborhood of each node might be much larger than the number of connections it can use to collect the information.

In [45], a CONGEST algorithm for *weak-diameter* is constructed using the building block of *weak-diameter ball-carving* algorithm. Their weak-diameter ball-carving is then extended to be simulatable on G^k in the CONGEST model for any constant k, while preserving the round complexity. It then uses a classical CONGEST reduction from *ball-carving* to graph decomposition [37], where the ball-carving algorithm is executed $\log n$ times.

In [13], a strong-diameter decomposition is constructed using a transformation from weak-diameter ball carving to strong-diameter ball carving in the CONGEST model, following by the same classical reduction from ball-carving to decomposition. Their transformation satisfies the property that if the original algorithm runs in $\operatorname{polylog} n$ rounds and produces $\operatorname{polylog} n$-diameter clusters, then the new algorithm also runs in $\operatorname{polylog} n$ rounds and produces $\operatorname{polylog} n$-diameter clusters (with different polynomial dependencies in $\log n$). Then, combining this transformation with the weak-diameter CONGEST ball-carving of [45], they obtain a $\operatorname{polylog} n$ rounds $\operatorname{polylog} n$ strong-diameter ball-carving in the CONGEST model, followed by a corresponding strong-diameter decomposition in the CONGEST model.

Since the weak-diameter ball carving of [45] could be simulated on G^2 in the CONGEST model, to see that the strong-diameter decomposition of [13] could be simulated on G^2 in the CONGEST model it remains to show that their weak-diameter to strong diameter ball-carving transformation could be also simulated in G^2 in the CONGEST model. We observe that the transformation of [13] uses communication between the nodes in the following two ways, which both could be simulated on G^2 in the CONGEST model:

- Counting the number of nodes in a cluster, by gathering information over Steiner trees. This could be simulated for G^2 since each node has to transfer only a *number* of nodes, where this number is still bounded by n in G^2, and so could be described in $O(\log n)$ bits.
- Computing a radius around a node v such that the ratio between the number of nodes in the cluster within that radius around v and the number of nodes beyond that radius exceeds some parameter. This is done by growing a BFS tree around v and gathering the number of nodes within each distance. Here as well, we have that nodes only transfer numbers, which are bounded by n, and thus their description is of size $O(\log n)$.