Recursive Composition and Bootstrapping for SNARKs and Proof-Carrying Data

Nir Bitansky*
nirbitan@tau.ac.il
Tel Aviv University

Ran Canetti*
canetti@tau.ac.il
Boston University and
Tel Aviv University

Alessandro Chiesa
alexch@csail.mit.edu
MIT

Eran Tromer†
tromer@tau.ac.il
Tel Aviv University

December 28, 2012

Abstract

Succinct non-interactive arguments (SNARGs) enable verifying NP statements with much lower complexity than required for classical NP verification (in fact, with complexity that is *independent* of the NP language at hand). In particular, SNARGs provide strong solutions to the problem of verifiably delegating computation.

Despite recent progress in the understanding and construction of SNARGs, there remain unattained goals. First, *publicly-verifiable SNARGs* are only known either in the random oracle model, or in a model that allows expensive offline preprocessing. Second, known SNARGs require from the prover significantly more time or space than required for classical NP verification.

We show that, assuming collision-resistant hashing, *any* SNARG having a natural *proof of knowledge* property (i.e., a SNARK) can be "bootstrapped" to obtain a *complexity-preserving* SNARK, i.e., one without expensive preprocessing and where the prover's time and space complexity is essentially the same as that required for classical NP verification. By applying our transformation to known publicly-verifiable SNARKs with expensive preprocessing, we obtain the first publicly-verifiable complexity-preserving SNARK in the plain model (and in particular, eliminate the expensive preprocessing), thereby attaining the aforementioned goals. We also show an analogous transformation for privately-verifiable SNARKs, assuming fully-homomorphic encryption. Curiously, our transformations do not rely on PCPs.

At the heart of our transformations is *recursive composition* of SNARKs and, more generally, new techniques for constructing and using *proof-carrying data* (PCD) systems, which extend the notion of a SNARK to the distributed setting. Concretely, to bootstrap a given SNARK, we recursively compose the SNARK to obtain a "weak" PCD system for shallow distributed computations, and then use the PCD framework to attain stronger, complexity-preserving SNARKs and PCD systems.

*Supported by the Check Point Institute for Information Security, Marie Curie grant PIRG03-GA-2008-230640, and ISF grant 0603805843. The first author was also supported by the Fulbright program.

†Supported by the Check Point Institute for Information Security and by the Israeli Centers of Research Excellence (I-CORE) program (center No. 4/11).

Contents

1 Introduction

Succinct arguments. We study proof systems [GMR89] for the purpose of verifying NP statements faster than by deterministically checking an NP witness in the traditional way. When requiring statistical soundness, significant savings in communication (let alone verification time) are unlikely [BHZ87, GH98, GVW02, Wee05]. If we settle for proof systems with *computational* soundness, known as *argument systems* [BCC88], then significant savings can be made. Using collision-resistant hashing (CRHs) and probabilistically-checkable proofs (PCPs) [BFLS91], Kilian [Kil92] showed a four-message interactive argument for NP where, to prove membership of an instance x in a given NP language L with NP machine M, communication and the verifier's time are bounded by $\mathrm{poly}(k + |M| + |x| + \log t)$, while the prover's running time by $\mathrm{poly}(k + |M| + |x| + t)$. Here, t is the classical NP verification time of M for the instance x, k is a security parameter, and poly is a *universal* polynomial (i.e., independent of k, M, x, and t). We call such argument systems *succinct*.

Proof of knowledge. A strengthening of computational soundness is (computational) *proof of knowledge*: it guarantees that, whenever the verifier is convinced by an efficient prover, not only a valid witness for the theorem *exists*, but also such a witness can be *extracted efficiently* from the prover. This captures the intuition that convincing the verifier of a given statement can only be achieved by (essentially) going through specific intermediate stages and thereby explicitly obtaining a valid witness along the way, which can be efficiently recovered by a *knowledge extractor*. Proof of knowledge is a natural property (satisfied by most proof system constructions, including the aforementioned one of Kilian [BG08]) that is useful in many applications of succinct arguments. It is also *essential* to the results of this paper.

Non-interactive succinct arguments. Kilian's protocol requires four messages. A challenge, which is of both theoretical and practical interest, is removing interaction from succinct arguments. As a first step in this direction, Micali [Mic00] constructed *one-message* succinct non-interactive arguments for NP, in the random oracle model, by applying the Fiat-Shamir paradigm [FS87] to Kilian's protocol.

In the plain model, it is known that one-message solutions are impossible for hard-enough languages (against non-uniform provers), so one usually considers the weaker goal of two-message succinct arguments where the verifier message is generated *independently* of the statement later chosen by the prover. Such arguments are called SNARGs. More precisely, a SNARG for a language L is a triple of algorithms $(\mathcal{G}, \mathcal{P}, \mathcal{V})$ where: (i) the generator $\mathcal{G}$, given the security parameter k, samples a *reference string* σ and a corresponding *verification state* τ ($\mathcal{G}$ can be thought to be run during an offline phase, by the verifier, or by someone the verifier trusts); (ii) the prover $\mathcal{P}(\sigma, x, w)$ produces a proof π for the statement "$x \in L$" given a witness w; (iii) $\mathcal{V}(\tau, x, \pi)$ deterministically verifies the validity of π for that statement.

Extending earlier work [ABOR00, DLN+04, Mie08, DCL08], several recent works showed how to remove interaction in Kilian's PCP-based protocol and obtain SNARGs of knowledge (*SNARKs*) using *extractable collision-resistant hashes* [BCCT12, DFH12, GLR11], or construct MIP-based SNARKs using fully-homomorphic encryption *with an extractable homomorphism property* [BC12].

The use of non-standard assumptions in the aforementioned works may be partially justified in light of the work of Gentry and Wichs [GW11], which shows that no SNARG can be proven sound via a black-box reduction to a falsifiable assumption [Nao03]. (We remark that [GW11] rule out SNARGs only for hard-enough NP languages. For the weaker goal of verifying deterministic polynomial-time computations, there are constructions relying on standard assumptions in various models.)

The preprocessing model. A notion that is weaker than a SNARK is that of a *preprocessing* SNARK: here, the verifier is allowed to conduct an expensive offline phase. More precisely, the generator $\mathcal{G}$ takes as an

additional input a time bound B, may run in time $\mathrm{poly}(k+B)$ (rather than $\mathrm{poly}(k+\log B)$), and generates a reference string σ and a verification state τ that can be used, respectively, to prove and verify correctness of computations of length at most B. A set of works [Gro10, Lip12, GGPR12, BCI$^+$13] achieves this weaker goal using techniques that can be cast as a combination of *linear* PCPs and linearly homomoprhic encryption/encoding with suitable knowledge properties [BCI$^+$13].

1.1 Motivating Questions

In this work, we study three open questions regarding SNARGs and SNARKs:

Public verifiability. A basic question regarding SNARKs is whether the verification state τ needs to be kept secret. In a *designated-verifier* SNARK, τ must be kept secret; in particular, τ must be protected from leakage, including the verifier's responses when checking proofs. (Of course, a new pair (σ, τ) can always be generated afresh to regain security.) In contrast, in a *publicly-verifiable* SNARK, the verification state τ associated with the reference string σ can be published. Thus leakage is not a concern, τ and σ can be used repeatedly, anyone who trusts the generation of τ can verify proofs, and proofs can be publicly archived for future use.

The SNARKs in [DCL08, Mie08, BCCT12, DFH12, GLR11, BC12] are of the designated-verifier kind (and there, indeed, an adversary learning the verifier's responses on, say, k proofs can break soundness). In contrast, Micali's protocol is publicly verifiable, but is in the random-oracle model. The protocols based on linear PCPs [Gro10, Lip12, GGPR12, BCI$^+$13] are also publicly verifiable, but only yield the weaker notion of preprocessing SNARKs. We thus ask:

Q-1: *Can we construct publicly-verifiable SNARKs without preprocessing in the plain model?*

Of course, we could always assume that Micali's protocol, when the random oracle is instantiated with a sufficiently-complicated hash function, is sound. However, this assumption does not seem to be satisfying, because it strongly depends on the specific construction, and does not shed light on the required properties from such a hash function. Instead, we would like to have a solution whose soundness is based on a *concise* and *general* assumption that is "construction-independent" and can be studied separately.

Complexity-preserving SNARKs. While typically the focus in SNARKs is on minimizing the resources required by the verifier, minimizing those required by the prover is another critical goal: e.g., the verifier may be *paying* to use the prover's resources by renting servers from the cloud, and the more resources are used the greater the cost to the verifier. These resources include, not only time complexity, but also space complexity, which tends to be a severe problem in practice (often more so than time complexity).

When instantiating the PCP-based SNARK constructions of [Mic00, DCL08, Mie08, BCCT12, DFH12, GLR11] with known time-efficient PCPs [BSCGT12], the SNARK prover runs in time $t \cdot \mathrm{poly}(k)$ and the SNARK verifier in time $|x| \cdot \mathrm{poly}(k)$. However, the quasilinear running time of the prover is achieved via the use of FFT-like methods, which unfortunately demand $\Omega(t)$ space even when the computation of the NP verification machine M requires space s with $s \ll t$.

The situation is even worse in the preprocessing SNARKs of [Gro10, Lip12, GGPR12, BCI$^+$13], where the generator runs in time $\Omega(t) \cdot \mathrm{poly}(k)$ to produce a reference string σ of length $\Omega(t) \cdot \mathrm{poly}(k)$. This string must then be stored somewhere and accessed by the prover every time it proves a new statement; thus, once again, $\Omega(t)$ space is needed (in contrast to a SNARK without preprocessing where the generator runs in time $\mathrm{poly}(k)$ and the reference string is short).

Ideally, we want SNARKs that simultaneously enable the verifier to run fast *and* enable the prover to use an amount of resources that is as close as possible to those required by the original computation. We

thus define a *complexity-preserving* SNARK to be a SNARK where the prover runs in time $t \cdot \mathrm{poly}(k)$ and space $s \cdot \mathrm{poly}(k)$, and the verifier runs in time $|x| \cdot \mathrm{poly}(k)$, when proving and verifying that a t-time s-space random-access machine M non-deterministically accepts an input x. We ask:

Q-2: *Can we construct complexity-preserving SNARKs?*

The SNARKs constructed by [BC12] are, in fact, complexity-preserving. However, that construction is for designated verifiers and also relies on a rather specific knowledge assumption. The case of public verifiability remains open, as well as whether there are more generic approaches to construct complexity-preserving SNARKs.

SNARK composition and proof-carrying data. It is tempting to use a SNARK to produce proofs that, in addition to attesting to the correctness of a given computation, also attest that a previous SNARK proof for another (related) computation has been verified (and so on recursively). An intriguing question is thus whether one can achieve stronger cryptographic primitives via such *recursive composition* of SNARKs, and under what conditions.

Several works have in fact studied this question. Valiant [Val08] studied the problem of *incrementally-verifiable computation* (IVC), where a deterministic computation is compiled into a new computation (with polynomially-related time and space) that after each step outputs, in addition to the current state, a short proof attesting to the correctness of the entire computation so far. Valiant showed (when phrased in our terminology) that IVC can be obtained by recursively composing publicly-verifiable SNARKs that have very efficient knowledge extractors, and conjectured that such SNARKs exist. In another work along these lines, Boneh, Segev, and Waters [BSW12] studied *targeted malleability* (TM), and showed how to obtain certain forms of TM by recursively composing publicly-verifiable preprocessing SNARKs that may have an expensive online verification.

Chiesa and Tromer [CT10] formulated and studied the security goal of *enforcing local properties* in dynamic distributed computations; this goal, in particular, captures many scenarios which seem to require SNARK recursive composition, such as the goals in [Val08] and [BSW12] (by choosing appropriate local properties to enforce). To achieve this security goal, [CT10] introduced a cryptographic primitive called *proof-carrying data* (PCD), which allows to dynamically compile a distributed computation into one where messages are augmented by short proofs attesting to the fact that the local property holds; this, without incurring significant overhead in communication or computation. They showed how to use recursive proof composition to obtain PCD, but only in a model where parties can access a signature oracle. We ask:

Q-3: *Under what conditions can SNARKs be recursively composed?*
More generally, what forms of PCD can be achieved in the plain model?

We further discuss and motivate the notions of verifying local properties of distributed computations and the framework of proof-carrying data in Section 1.3.

1.2 Our Results

In this work, we positively answer all three questions. To do so, we develop techniques demonstrating that the three questions are, in fact, tightly related to one another.

A bootstrapping theorem for SNARKs and PCD. Our main technical result consists of two generic transformations. The first transformation takes *any* SNARK (possibly having poor efficiency, e.g., having expensive preprocessing, a prover running in quadratic time, a prover requiring large space, and so on) and outputs a PCD system, with analogous efficiency properties, for a large class of distributed computations.

The second transformation takes *any* PCD system (such as the one output by the firsttransformation) and outputs a *complexity-preserving* SNARK or PCD system. These transformations work in both publicly-verifiable or designated-verifier cases (where SNARKs can be proved secure based on potentially weaker knowledge assumptions).

Theorem (informal)**.** *Assume existence of collision-resistant hash functions. Then:*
(i) Any publicly-verifiable SNARK can be efficiently transformed into a publicly-verifiable PCD system for distributed computations of constant depth or over paths of polynomial depth.
(ii) Any publicly-verifiable PCD system (for distributed computations of constant depth or over paths of polynomial depth) can be efficiently transformed into a complexity-preserving publicly-verifiable SNARK or PCD system.
(Where the depth *of a distributed computation is, roughly, the length of the longest path in the graph representing the distributed computation over time.)*
Assuming existence of fully-homomorphic encryption, an analogous statement holds for the designated-verifier case.

While this theorem implies a significant efficiency improvement for preprocessing SNARKs (as it removes the expensive preprocessing), it is also useful for improving the efficiency of SNARKs that do not have expensive preprocessing, yet are still not complexity preserving, such as PCP-based constructions in the plain model.

Applying our theorem to any of the preprocessing SNARKs of [Gro10, Lip12, GGPR12, BCI$^+$13], we obtain positive answers to the three aforementioned open questions:

Corollary (informal)**.** *There exist publicly-verifiable SNARKs and PCD systems (for a large class of distributed computations), in the plain model, under "knowledge-of-exponent" assumptions. Moreover, there exist such SNARKs and PCD systems that are complexity-preserving.*

To prove our main theorem, we develop three generic tools:
1. **SNARK Recursive Composition:** "A (publicly- or privately-verifiable) SNARK can be composed a constant number of times to obtain a PCD system for constant-depth distributed computations (without making special restrictions on the efficiency of the knowledge extractor)."
2. **PCD Depth Reduction:** "Distributed computations of constant depth can express distributed computations over paths of polynomial depth."
3. **Locally-Efficient RAM Compliance:** "The problem of checking whether a random-access machine non-deterministically accepts an input within t steps can be reduced to checking that a certain local property holds throughout a distributed computation along a path of $t \cdot \mathrm{poly}(k)$ nodes and every node's local computation is only $\mathrm{poly}(k)$, independently of t, where k is the security parameter."

Succinct arguments without the PCP Theorem. When combined with the protocols of [Gro10, Lip12, GGPR12, BCI$^+$13], our transformations yield SNARK and PCD constructions that, unlike all previous constructions (even interactive ones), *do not invoke the PCP Theorem* but only elementary probabilistic-checking techniques [BCI$^+$13]. (Note that the "PCP-necessity" result of [RV09] does not apply here.) This provides an essentially different path to the construction of succinct arguments, which deviates from all previous approaches (such as applying the Fiat-Shamir paradigm [FS87] to Micali's "CS proofs" [Mic00]). We find this interesting both on a theoretical level (as it gives us the *only* known complexity-preserving publicly-verifiable SNARKs) and on a heuristic level (as the construction seems quite simple and efficient).

technique	main assumption	generator time	prover time	prover space	verifier time	verification	complexity preserving?
PCP	extractable CRH	$\mathrm{poly}(k)$	$t \cdot \mathrm{poly}(k)$	$t \cdot \mathrm{poly}(k)$	$\mathrm{poly}(k)$	designated	no
linear PCP	linear-only hom.	$t \cdot \mathrm{poly}(k)$	$t \cdot \mathrm{poly}(k)$	$t \cdot \mathrm{poly}(k)$	$\mathrm{poly}(k)$	public	no
MIP	hom. extraction	$\mathrm{poly}(k)$	$t \cdot \mathrm{poly}(k)$	$s \cdot \mathrm{poly}(k)$	$\mathrm{poly}(k)$	designated	yes
this work	any SNARK	$\mathrm{poly}(k)$	$t \cdot \mathrm{poly}(k)$	$s \cdot \mathrm{poly}(k)$	$\mathrm{poly}(k)$	public	yes

Table 1: Features of known SNARK constructions vs. those obtained using our transformations.

1.3 More on Proof-Carrying Data and Compliance Engineering

Succinct arguments focus on the case of a single prover and a single verifier. This suffices for capturing, say, a client interacting with a single worker performing some self-contained computation for the client. However, reality is often more complex: computations may be performed by multiple parties, each party with its own role, capabilities, and trust relations with others.

In general, there are multiple (even apriori unboundedly many) parties, where each party i, given inputs from some other parties, and having its own local input linp_i, executes a program, and sends his output message z_i to other parties, each of which will in turn act likewise (i.e., perform a local computation and send the output message to other parties), and so on in a dynamical fashion. In other words, reality often involves possibly complex *distributed computations*.

There are many *security goals*, both about integrity and privacy, that one may wish to achieve in distributed computations. The study of *multi-party computation* (MPC) in the past three decades has focused on formulating definitions and providing solutions that are as comprehensive as possible for secure distributed computations. This ambitious goal was met by powerful generic constructions, working for any polynomial-time functionality and in the presence of arbitrary malicious behavior, e.g., [GMW87, BOGW88]. A major caveat of generic MPC protocols, however, is that they often require the parties participating in the computation to interact heavily with all other parties and perform much more expensive computations than their "fair share". In fact, such overheads are inherent for some security goals, e.g., *broadcast* [FLP85, KY86].

Chiesa and Tromer [CT10, CT12] introduced and studied a specific security goal, *enforcing compliance with a given local property*; as we shall see, for this goal, it is possible to avoid the aforementioned caveats. More concretely, the goal is to ensure that any given message z output by any party during the distributed computation is the result of *some* previous distributed computation in which every party's local computation (including the party's input messages, local inputs, and output message) satisfies a prescribed local property $\mathbb{C}$; i.e., the goal is to ensure that there is some "explanation" for the generation of the message z as the aggregate of many local computations, each satisfying $\mathbb{C}$. For example, the local property $\mathbb{C}$ might be "*the local input* linp_i *is a program* prog_i *bearing a valid signature of the system administrator and, moreover, the output message* z_i *is the correct output of running* prog_i *on the input messages*". Such a local property would ensure that a message z resulting from a distributed computation satisfying $\mathbb{C}$ is in fact the result of correctly executing only programs vetted by the system administrator.

Here, the focus is not on the behavior of specific parties with respect to specific inputs but, rather, whether the generation of a *given message* can be properly explained by some "compliant" behavior. As we shall see shortly, the advantage of studying this security goal is that it will ultimately allow for solutions that do not introduce additional interaction between parties and do not need to rely on a fixed set of parties who are all familiar with each other and jointly come together to compute some functionality. We also note that, in its most basic form, the security goal only talks about integrity and not about privacy.

Proof-carrying data. To fulfill the above goal, Chiesa and Tromer [CT10] proposed the *Proof-Carrying Data* (PCD) solution approach: each party i behaves exactly as in the original distributed computation (where there is no integrity guarantee), except that i also appends to his message z_i a *succinct proof* π_i asserting that z_i is consistent with some distributed computation in which every local computation satisfies $\mathbb{C}$. Party i generates the proof π_i based on z_i, inp_i, previous messages, and their proofs. Crucially, generating π_i does not require party i to perform much more work than generating z_i in the first place. Furthermore, the "natural" evolution of the distributed computation, including its communication pattern, is unaffected. This solution approach extends Valiant's notion of incrementally-verifiable computation [Val08], which can be cast as verifying a "path" distributed computation where the local property to be enforced is the transition function of a fixed deterministic machine, and the set of parties is fixed according to the number of steps made by the machine. See Figure 1 on page 8 for a diagram of this idea.

An abstraction for SNARK recursive composition, and compliance engineering. As already mentioned, in this work we use recursive composition of SNARKs to obtain PCD systems for a large class of distributed computations. While describing the proof to parts of our main theorem could be done without using PCD systems, PCD systems enable us to state, once and for all, exactly what we can "squeeze" out of recursive composition of (even the most basic and inefficient) SNARKs. From thereon, we can forget about the many technical details required to make recursive composition work, and only focus on the corresponding *guarantees*, rather than implementation details. Specifically, armed with PCD systems, we can concentrate on the simpler and cleaner task of *compliance engineering*: how to express a given security goal as a local property. We can then achieve the security goal by enforcing the local property by using a PCD system.

For example, already in this work, after constructing a "weak" PCD system, we solve all other technical challenges (including obtaining stronger PCD systems) via compliance engineering. As another example, *targeted malleability* [BSW12] can be obtained via a suitable choice of local property and then enforcing the local property by using a PCD system over any homomorphic encryption scheme.[1] The class of distributed computations supported by the resulting construction is the same as that of the PCD system used.

More generally, we believe that investigating the power (and limits) of compliance engineering is a very interesting question: what security goals can be efficiently achieved by enforcing local properties?

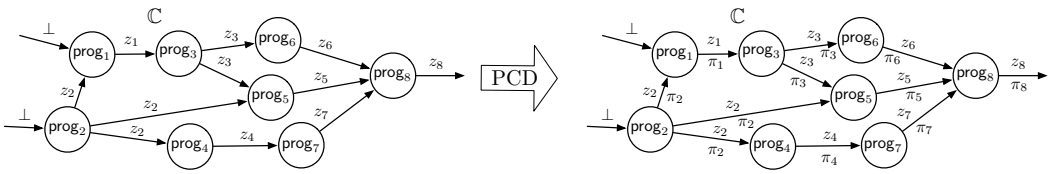

Figure 1: Proof-carrying data enables each party in a distributed computation to augment his message z_i with a short easy-to-verify proof π_i that is computed "on-the-fly", based on previous messages and proofs. At any point during the computation, anyone may inspect a message to decide if it is *compliant* with the given local property $\mathbb{C}$. Distributed computations are represented as directed acyclic graphs "unfolding over time".

[1] More precisely, the PCD system must have a *zero-knowledge* property. Zero-knowledge PCD systems are easily defined and, as expected, follow from recursive composition of zero-knowledge SNARKs.

1.4 The Ideas In A Nutshell

We now give the basic intuition, stripping away all abstraction layers, behind one part of our main theorem. Specifically, we explain how to transform any (possibly very inefficient) SNARK $(\mathcal{G}, \mathcal{P}, \mathcal{V})$ into a complexity-preserving SNARK $(\mathcal{G}^\star, \mathcal{P}^\star, \mathcal{V}^\star)$ (that, in particular, has no expensive preprocessing), assuming collision-resistant hashing. Consider first the case where $(\mathcal{G}, \mathcal{P}, \mathcal{V})$ is publicly verifiable.

Suppose that $(\mathcal{G}, \mathcal{P}, \mathcal{V})$ is a preprocessing SNARK (this only makes the transformation harder because we must get rid of preprocessing). Recall that, in such a SNARK, the online verification phase is succinct, but the offline phase is allowed to be expensive, in the following sense. The generator $\mathcal{G}$ takes as an additional input a time bound B, may run in time $\mathrm{poly}(k+B)$, and generates a (potentially long) reference string σ and a (short) verification state τ that can be used, respectively, to prove and verify correctness of computations of length at most B. The (online) verifier $\mathcal{V}$ still runs in time $\mathrm{poly}(k)$, independently of B. (Additionally, no guarantees are made about the time and space complexity of the honest prover, except that they both are $\mathrm{poly}(k+B)$.) We would like to construct $(\mathcal{G}^\star, \mathcal{P}^\star, \mathcal{V}^\star)$ so that $\mathcal{G}^\star$ runs in time $\mathrm{poly}(k)$ (which in particular bounds the size of the reference string σ) and $\mathcal{P}^\star$ runs in time $t \cdot \mathrm{poly}(k)$ and space $s \cdot \mathrm{poly}(k)$.

At high-level, the idea is to first represent the long t-step computation of the random-access machine M to be verified as a collection of $O(t)$ small $\mathrm{poly}(k)$-step computations, and then use recursive composition to aggregate many SNARK proofs for the correctness of each of these small computations into a *single* SNARK proof. Indeed, by repeatedly invoking the prover $\mathcal{P}$ of the "inefficient" preprocessing SNARK $(\mathcal{G}, \mathcal{P}, \mathcal{V})$ only to prove the correctness of *small computations*, we would "localize" the effect of the SNARK's inefficiency. Specifically, when running the generator $\mathcal{G}$ in the offline phase in order to produce (σ, τ), we would only have to budget for a time bound $B = \mathrm{poly}(k)$, thereby making running $\mathcal{G}$ cheap. Furthermore, if the collection of small computations can be computed in time t and space s (up to $\mathrm{poly}(k)$ factors), then running $\mathcal{P}$ on each of these small computations in order to produce the final proof would only take time $t \cdot \mathrm{poly}(k)$ and space $s \cdot \mathrm{poly}(k)$. Overall, we would achieve complexity preservation. Let us make this intuition somewhat more concrete.

Starting point: incrementally-verifiable computation. A natural starting point towards fulfilling the above plan is trying to use of the idea of incrementally-verifiable computation (IVC) [Val08]. Recall that the goal in IVC is to transform a given computation into a new computation that after every step outputs its entire state and a proof of its correctness so far, while preserving the time and space efficiency of the original computation (up to $\mathrm{poly}(k)$ factors).

Specifically, to be convinced that there exists a witness w for which the random-access machine M accepts w within t steps, it suffices to be convinced that there is a sequence of t' states $S_0, S_1, \ldots, S_{t'}$ of M (with $t' \leq t$) that (a) starts from an initial state, (b) ends in an accepting state, and (c) every state correctly follows the previous one according to the transition function of M. Equivalently, from the perspective of proof-carrying data, we can think of a distributed computation of at most t parties, where the i-th party P_i receives the state S_{i-1} of M at step $i-1$, evaluates one step of M, and sends the state S_i of M at step i to the next party; the last party checks that the received state is an accepting one.

This suggests the following solution, which we can think of as happening "in the mind" of the new SNARK prover $\mathcal{P}^\star$. Using the SNARK prover $\mathcal{P}$, the first party P_1 proves to the second party P_2 that the state S_1 was generated by running the first step of M correctly. Then, again using $\mathcal{P}$, P_2 proves to the third party P_3 that, not only did he evaluate the second step of M correctly and obtained the state S_2, but he *also* received state S_1 carrying a valid proof (i.e., accepted by the SNARK verifier $\mathcal{V}$) claiming that S_1 was generated correctly. Then, P_3 uses $\mathcal{P}$ to prove to the fourth party P_4 that, not only did he evaluate the third step of M correctly and obtained the state S_3, but he *also* received state S_2 carrying a valid proof claiming

that S_2 was generated correctly, and so on until the last party who, upon receiving a state carrying a proof of validity, proves that the last state is accepting. A verifier at the end of the chain gets a single, easy-to-verify, proof aggregating the correctness of all the steps in M's computation. Hopefully, by relying on the proof of knowledge property of the SNARK, it is possible to recursively extract from any convincing prover a full transcript of the computation attesting to its correctness.

From the above IVC-based approach to our eventual goal of complexity-preserving SNARKs without preprocessing there is still a significant gap; we now describe the difficulties and how we overcome them.

Challenge 1: IVC with preprocessing SNARKs, and the size of local computations. In his construction of IVC, Valiant relies on the existence of publicly-verifiable SNARKs that *do not have expensive preprocessing*. In our setting, we have only a preprocessing SNARK at hand, so we have to ensure that each of the computations whose correctness we are proving is shorter than the time bound B associated with the preprocessing. Specifically, B must be larger than the running time of the SNARK verifier $\mathcal{V}$ plus a single computation step of M. This is reminiscent of the *bootstrapping paradigm* in fully-homomorphic encryption [Gen09], where, in order to bootstrap a somewhat homomorphic scheme, homomorphic evaluation should support the decryption operation plus a single computation step. Whereas in bootstrapping of homomorphic encryption the challenge is to get the decryption circuit to be small enough, in our setting the running time of $\mathcal{V}$ (even for a preprocessing SNARK) is already $\mathrm{poly}(k)$-small, and the challenge is to get the computation required to perform one step of M to be small enough. Indeed, running step i in the "middle" of the computation requires computation proportional to the corresponding state S_i. Such a computations may thus be as large as the space s used by M, which in turn could be as large as $\Omega(t)$. If, instead, we could ensure that each local computation being proven is of size $\mathrm{poly}(k)$, then we could set $B = \mathrm{poly}(k)$ and thereby avoid expensive preprocessing.

To achieve this goal, we invoke a "computational" reduction [BEG$^+$91, BSCGT13] that transforms M into machine M' that requires only $\mathrm{poly}(k)$ space and preserves the proof of knowledge property (i.e., any computationally-bounded adversary producing a witness that M' accepts can be efficiently used to find a witness that M accepts). The idea is that M' emulates M but does not bother to explicitly store its random-acess memory; instead, reads from memory are satisfied by "guessing" the resulting value, and verifying its correctness via dynamic Merkle hashing. These guesses, and corresponding Merkle verification paths, are appended to the witness, whose length, crucially, does not affect the time to run a step of the machine. (To ensure that the new computation steps are small enough, we ensure that each step only looks at a small chunk of the witness, which is now at least as large as the original space s of M.)

This strategy also ensures that the resulting SNARK is complexity-preserving. Indeed, reducing M to the "small-space" M' and its representation as a $\tilde{O}(t)$-step distributed computation can be done "on the fly", using the same time t and space s as the original computation, up to $\mathrm{poly}(k)$ factors.

Challenge 2: extractor efficiency and the depth of the computation. As mentioned, to prove that the above approach is secure, we need to rely on proof of knowledge, in order to perform recursive extraction. This means that a proof of security based on recursive knowledge extraction will work for only a *constant* number of recursive compositions (due to the polynomial blowup in extractor size for each such composition). However, the distributed computation we described has *polynomial* depth. Valiant showed that, if the knowledge extractor is extremely efficient (linear in the prover's size), then the problem can be avoided by aggregating proofs along a tree rather than along a path. We avoid Valiant's assumption by extending his idea into aggregating proofs along "wide proof trees" of constant depth (similarly to the construction of SNARKs from extractable collision-resistant hash functions [BCCT12, GLR11].)

Another challenge: the case of designated-verifier SNARKs. So far we have assumed that the SNARK

$(\mathcal{G}, \mathcal{P}, \mathcal{V})$ is publicly verifiable. What happens in the designated-verifier case? In this case, it is not clear how a party can prove that he verified a received proof without actually knowing the corresponding private verification state (which we cannot allow because doing so would void the security guarantee of the SNARK). We solve this problem by showing how to carefully use fully-homomorphic encryption to recursively compose proofs *without* relying on intermediate parties knowing the verification state.

From intuition to proof through PCD. We have now presented all the high-level ideas that go into proving one part of our main theorem: how to transform *any* SNARK into a complexity-preserving one. Let us briefly outline how these ideas are formalized via results about the constructibility of PCD systems. Our first step is to transform any SNARK into a PCD system for constant-depth distributed computations; this step generalizes the notion of IVC to a richer class of distributed computations (not only paths) and to arbitrary local security properties (not only the transition function of a fixed machine). We then forget about the details of recursively composing SNARKs, and express the security goals we are interested in via the compliance of distributed computations with carefully-chosen local properties. In this spirit, we show how PCD systems for constant-depth distributed computations give rise to PCD systems for a class of polynomial-depth distributed computations (including polynomial-length paths). Finally, we show how these can in turn be used to obtain complexity-preserving SNARKs (that, in particular, have no preprocessing), by suitably representing a computation to be verified as a sequence of "small" computations in a distributed path computation.

Proving the above claims about PCD systems will enable us to construct complexity-preserving PCD systems as well. Next, we provide a more detailed discussion of these claims.

1.5 Roadmap

In Section 2, we discuss our results in somewhat more detail, describing each of the three tools we develop, and then how these come together for our main result. We then proceed to the technical sections of the paper, beginning with definitions of the universal relation and RAMs in Section 3, of SNARKs in Section 4, and of PCD in Section 5. After that, we give technical details for our three tools, in Section 6, Section 7, and Section 8 respectively. In Section 9, we finally give the technical details for how our tools come together to yield the transformations claimed by our main theorem.

2 Overview of Results

We discuss our results in more detail.

2.1 SNARKs and Proof-Carrying Data

To describe our results, we first recall in more detail what are SNARKs and Proof-Carrying Data. The formal definitions can be found in Section 4 and Section 5 respectively.

When discussing verification-of-computation problems, it is convenient to consider a canonical representation given by the *universal language* $\mathcal{L}_{\mathcal{U}}$ [BG08]. This language consists of all $y = (M, x, t)$, where M is a random-access machine, x is an input for M, and t is a time bound, such that there is a witness w for which $M(x, w)$ accepts within t time steps (see Section 3). When considering an NP language $L \subseteq \mathcal{L}_{\mathcal{U}}$, the machine M is the NP verification machine and $t = t(|x|)$ is the polynomial bound on its running time.

SNARKs. A *SNARK* $(\mathcal{G}, \mathcal{P}, \mathcal{V})$ for an NP language $L \subset \mathcal{L}_{\mathcal{U}}$ works as follows. The generator $\mathcal{G}(1^k)$, where k is the security parameter, samples a reference string σ and verification state τ in time $\mathrm{poly}(k)$. The prover $\mathcal{P}(\sigma, y, w)$, where $y = (M, x, t) \in L$ and w is a witness for y, produces a proof π in time $\mathrm{poly}(k + |y| + t)$.

The verifier $\mathcal{V}(\tau, y, \pi)$ deterministically decides whether to accept π as a proof for y, in time $\mathrm{poly}(k + |y|)$. The polynomial poly is *universal* (and thus independent of the NP language L, and its associated running time t). In terms of security, the SNARK proof of knowledge property states that: when a malicious efficient prover $\mathcal{P}^*(\sigma)$ produces a statement y (possibly depending on σ) and proof π that is accepted by $\mathcal{V}$, then, with all but negligible probability, a corresponding efficient extractor $\mathcal{E}_{\mathcal{P}^*}(\sigma)$ outputs a witness w for y.

PCD. We define PCD systems as in [CT10], except for minor modifications to suit our setting and the plain model. A *PCD system* is associated with a compliance predicate $\mathbb{C}$ representing a local security property to be enforced throughout a distributed computation. It is a triple $(\mathbb{G}, \mathbb{P}_{\mathbb{C}}, \mathbb{V}_{\mathbb{C}})$, where $\mathbb{G}$ is the generator, $\mathbb{P}_{\mathbb{C}}$ the prover, and $\mathbb{V}_{\mathbb{C}}$ the verifier; it induces a dynamic compiler to be used in a distributed computation as follows. The generator $\mathbb{G}$, on input the security parameter k, samples a reference string σ and a corresponding verification state τ. Then, any party in the distributed computation, having received proof-carrying input messages $\vec{\mathsf{z}}_{\mathsf{i}}$ and produced an output message z_{o} to be sent to a next party, invokes the PCD prover $\mathbb{P}_{\mathbb{C}}(\sigma, \mathsf{z}_{\mathsf{o}}, \mathsf{linp}, \vec{\mathsf{z}}_{\mathsf{i}}, \vec{\pi}_{\mathsf{i}})$, where $\vec{\mathsf{z}}_{\mathsf{i}}$ are the input messages, $\vec{\pi}_{\mathsf{i}}$ their proofs, and linp is any additional local input used (e.g., code or randomness), to produce a proof π_{o} for the claim that z_{o} is consistent with some $\mathbb{C}$-compliant distributed computation leading up to z_{o}. The verifier $\mathbb{V}_{\mathbb{C}}(\tau, \mathsf{z}_{\mathsf{o}}, \pi_{\mathsf{o}})$ can be invoked by any party knowing the verification state τ in order to verify the compliance of a message z_{o}. (If the PCD system is publicly verifiable, anyone can be assumed to know τ; in the designated-verifier case, typically, only some parties, or even just one, will know τ.)

From a technical perspective, we can think of a PCD system as a *distributed SNARK*: the proving algorithm is "distributed" among the parties taking part in the computation, each using a local prover algorithm (with local inputs) to prove compliance of the distributed computation carried out so far, based on previous proofs of compliance.

Succinctness. Analogously to a SNARK, the generator $\mathbb{G}(1^k)$ is required to run in time $\mathrm{poly}(k)$, the (honest) prover $\mathbb{P}_{\mathbb{C}}(\sigma, \mathsf{z}_{\mathsf{o}}, \mathsf{linp}, \vec{\mathsf{z}}_{\mathsf{i}}, \vec{\pi}_{\mathsf{i}})$ in time $\mathrm{poly}(k + |\mathbb{C}| + |\mathsf{z}_{\mathsf{o}}| + t_{\mathbb{C}}(|\mathsf{z}_{\mathsf{o}}|))$, and the verifier $\mathbb{V}_{\mathbb{C}}(\tau, \mathsf{z}_{\mathsf{o}}, \pi_{\mathsf{o}})$ in time $\mathrm{poly}(k + |\mathbb{C}| + |\mathsf{z}_{\mathsf{o}}|)$, where $t_{\mathbb{C}}(|\mathsf{z}_{\mathsf{o}}|)$ is the time to evaluate $\mathbb{C}(\mathsf{z}_{\mathsf{o}}; \vec{\mathsf{z}}_{\mathsf{i}}, \mathsf{linp})$ and poly is a universal polynomial. In other words, proof-generation by the prover $\mathbb{P}_{\mathbb{C}}$ is (relatively) efficient in the *local* computation (and independent of the computation performed by past or future nodes), and proof verification by the verifier $\mathbb{V}_{\mathbb{C}}$ is independent of the computation that produced the message (no matter how long and expensive is the history that led to the message being verified).

Security. Again analogously to a SNARK, a PCD system also has a proof of knowledge property: when a malicious prover $\mathbb{P}^*(\sigma)$ produces a message z_{o} and proof π_{o} such that $\mathbb{V}_{\mathbb{C}}(\tau, \mathsf{z}_{\mathsf{o}}, \pi_{\mathsf{o}}) = 1$ then, with all but negligible probability, the extractor $\mathbb{E}_{\mathbb{P}^*}(\sigma)$ outputs *a full transcript of a distributed computation* that is $\mathbb{C}$-compliant and leads to the message z_{o}. In other words, $\mathbb{V}_{\mathbb{C}}$ can only be convinced to accept a given message whenever the prover $\mathbb{P}^*$ actually "knows" a $\mathbb{C}$-compliant computation leading up to that message.

A useful notion: the distributed computation graph. It will be convenient to think of a distributed computation "unfolding over time" as a (labeled) directed acyclic graph (generated dynamically as the computation evolves) where computations occur at nodes, and directed edges denote messages exchanged between parties. (When the same party computes twice, it will be a separate node "further down" the graph; hence the graph is acyclic.) See Figure 1 for a graphical depiction.

Preprocessing SNARKs and PCD systems. We also consider the weaker notion of an *(expensive) preprocessing* SNARK, in which the generator takes as additional input a time bound B, may run in time $\mathrm{poly}(k + B)$, and the reference string it outputs only works for computations of length at most B. Similarly, we also consider *preprocessing* PCD systems, where the reference string works for distributed computations in which every node's computation is at most B (and not the entire distributed computation).

We now move to describe in more detail, each of the three main tools developed to obtain our transformation.

2.2 The SNARK Recursive Composition Theorem

Our first step is to show that the existence of a SNARK implies the existence of a PCD system, with analogous verifiability and efficiency properties, for the class of *constant-depth compliance predicates*. Here, the *depth* $\mathsf{d}(\mathbb{C})$ of a compliance predicate $\mathbb{C}$ is the length of the longest path in (the graph corresponding to) any distributed computation compliant with $\mathbb{C}$. (Note that a distributed computation of depth $\mathsf{d}(\mathbb{C})$, even a constant, may have many more "nodes" than $\mathsf{d}(\mathbb{C})$; e.g., it could be a wide tree of depth $\mathsf{d}(\mathbb{C})$.)

Theorem 1 (SNARK Recursive Composition— informal)**.**

(i) Any publicly-verifiable SNARK can be efficiently transformed into a corresponding publicly-verifiable PCD system for constant-depth compliance predicates.

(ii) Assuming the existence of FHE, any designated-verifier SNARK can be efficiently transformed into a corresponding designated-verifier PCD system for constant-depth compliance predicates.

Moreover, if the SNARK is of the preprocessing kind, then so is the corresponding PCD system; in such a case, our transformation further relies on collision-resistant hashing.

The purpose of the theorem is to cleanly encapsulate the idea of "recursive proof composition" of SNARKs within a PCD construction. After proving this theorem, every time we need to leverage the benefits of recursive proof composition, we can conveniently work "in the abstract" by engineering a (constant-depth) compliance predicate encoding the desired local property, and then invoke a PCD system to enforce this property across a distributed computation. We now outline the ideas behind the proof of the theorem; see Section 6 for details.

Part (i): the case of public verifiability. At high level, the PCD system $(\mathbb{G}, \mathbb{P}_{\mathbb{C}}, \mathbb{V}_{\mathbb{C}})$ is constructed by using the SNARK $(\mathcal{G}, \mathcal{P}, \mathcal{V})$ as follows. The PCD generator $\mathbb{G}$ invokes the SNARK generator $\mathcal{G}$. The PCD prover $\mathbb{P}_{\mathbb{C}}$ uses the SNARK prover $\mathcal{P}$ to perform recursive proof composition relative to the given compliance predicate $\mathbb{C}$. Roughly, when a party A wishes to begin a computation with message z_A, A uses $\mathcal{P}$ to generate a SNARK proof π_A for the claim "$\mathbb{C}(\mathsf{z}_A; \bot, \bot) = 1$"; π_A attests to the fact that z_A is a compliant "input" to the distributed computation. When a party B receives z_A, after performing some computation by using some local input linp_B (which may include a program) and then producing a message z_B, B uses $\mathcal{P}$ to generate a SNARK proof π_B for the claim "$\exists\, (\mathsf{linp}'_B, \mathsf{z}'_A, \pi'_A)$ s.t. $\mathbb{C}(\mathsf{z}_B; \mathsf{linp}'_B, \mathsf{z}'_A) = 1$ *and* π'_A is a valid SNARK proof for the $\mathbb{C}$-compliance of z'_A". And so on: in general, a party receiving input messages $\vec{\mathsf{z}}_{\mathsf{i}}$ with corresponding proofs $\vec{\pi}_{\mathsf{i}}$, having local input linp, and producing message z_{o}, runs the PCD prover $\mathbb{P}_{\mathbb{C}}(\sigma, \mathsf{z}_{\mathsf{o}}, \mathsf{linp}, \vec{\mathsf{z}}_{\mathsf{i}}, \vec{\pi}_{\mathsf{i}})$, which uses $\mathcal{P}$ to generate a SNARK proof π_{o} for the claim

"$\exists\, (\mathsf{linp}', \vec{\mathsf{z}}'_{\mathsf{i}}, \vec{\pi}'_{\mathsf{i}})$ s.t. $\mathbb{C}(\mathsf{z}_{\mathsf{o}}; \mathsf{linp}', \vec{\mathsf{z}}'_{\mathsf{i}}) = 1$ *and* each π'_i is a valid SNARK proof for the $\mathbb{C}$-compliance of z'_i";

the proof π_{o} attests to the fact that z_{o} can be "explained" with *some* $\mathbb{C}$-compliant distributed computation. The PCD verifier $\mathbb{V}_{\mathbb{C}}$ uses the SNARK verifier $\mathcal{V}$ to verify the proofs.

The proof of knowledge property of the SNARK is crucial for the above to work. Indeed, there likely *exists* a proof, say, π_1 for the $\mathbb{C}$-compliance of z_1, even if compliance does not hold, because the SNARK is only computationally sound. While such "bad" proofs may indeed exist, they are hard to find. Proving the statement above with a proof of knowledge, however, ensures that whoever is able to prove that statement also knows a proof π_1, and this proof can be found efficiently (and thus is not "bad").

A key technical point is how to formalize the statement that "π is a valid proof for the $\mathbb{C}$-compliance of z". Naively, such a statement would directly ask about the existence of a $\mathbb{C}$-compliant distributed computation transcript T leading to z. However, this would mean that each prover along the way would have to know

the entire distributed computation so far. Instead, by carefully using recursion, we can ensure that the statement made by each prover only involves its own proof-carrying input messages, local inputs, and outputs. Following [CT10], this is formally captured by proving SNARK statements regarding the computation of a special *recursive* "PCD machine" $\mathsf{M}^{\mathbb{C}}$. The machine $\mathsf{M}^{\mathbb{C}}$, given an alleged output message together with a witness consisting of proof-carrying inputs, verifies: (a) that the inputs and outputs are $\mathbb{C}$-compliant *as well as* (b) verifying that each input carries a valid proof that $\mathsf{M}^{\mathbb{C}}$ itself accepts z after a given number of steps. (Of course, to formalize this recursion, one has to use an efficient version of the Recursion Theorem.) See Section 6.1 for details.

While the core idea behind our construction is similar to the ideas used in [Val08] and in [CT10], the details and the proof are quite different: [Val08] focuses on a special case of PCD, while [CT10] work in a model where parties can access a signature oracle rather than in the plain model.

Part (ii): the case of designated verifiers. The more surprising part of the theorem, in our mind, is the fact that designated-verifier SNARKs can *also* be composed. Here, the difficulty is that the verification state τ (and hence the verification code) is not public. Hence, we cannot apply the same strategy as above and prove statements like "the SNARK verifier accepts". Intuitively, fully-homomorphic encryption (FHE) may help in resolving this problem, but it is not so clear how to use it. Indeed, if we homomorphically evaluate the verifier, we only obtain its answer *encrypted*, whereas we intuitively would like to know right away whether the proof we received is good or not, because we need to generate a new proof depending on it.

We solve this issue by directly proving that we homomorphically evaluated the verifier, and that a certain encrypted bit is indeed the result of this (deterministic) evaluation procedure. Then, with every proof we carry an encrypted bit denoting whether the data so far is $\mathbb{C}$-compliantor not;when we need to "compose" we ensure that the encrypted answer of the current verification is correctly multiplied with the previous bit, thereby aggregating the compliance up to this point. For further details see Section 6.2.

The case of preprocessing SNARKs. Our theorem also works with preprocessing SNARKs. Specifically, when plugging a preprocessing SNARK into the SNARK Recursive Composition Theorem, we obtain a corresponding preprocessing PCD system, where (as in a preprocessing SNARK) the PCD generator $\mathbb{G}$ also takes as input a time bound B, and produces a reference string and verification state that work as long as the amount of local computation performed by a node (or, more precisely, the time to compute $\mathbb{C}$ at a node) in the distributed computation is bounded by B. More concretely, if $\mathbb{G}$ invokes the SNARK generator $\mathcal{G}$ with time bound B', the computation allowed at each node i is allowed to be, roughly, as large as $B' - \deg(i) \cdot t_{\mathcal{V}}$, where $t_{\mathcal{V}}$ is the running time of SNARK verifier $\mathcal{V}$ and $\deg(i)$ is the number of incoming inputs (which is also the number of proofs to be verified); thus we can simply set $B' = B + \max_i \deg(i) \cdot t_{\mathcal{V}}$. (The degree will always be bounded by a fixed polynomial in the security parameter in our applications.) Unlike completeness, the security properties are not affected by preprocessing; the proof of the SNARK Recursive Composition Theorem in the case with no preprocessing carries over to the preprocessing case. Yet, while we do not need a different security proof for the preprocessing case, setting up a PCD construction that works properly in this setting should be done with care. For example, in their construction of encryption with *targeted malleability*, Boneh, Segev, and Waters [BSW12] recursively composed preprocessing SNARKs without leveraging the fast running time of the SNARK verifier, and hence they needed a preprocessing step that budgets for an *entire* distributed computation and not just a *single* node's computation (as in our case). This difference is crucial; for instance, it is essential to our result that allows to remove preprocessing from a SNARK or PCD system.

Why only constant depth? The restriction to constant-depth compliance predicates arises because of technical reasons during the proof of security. Specifically, we must recursively invoke the SNARK knowledge

property in order to "dig into the past", starting from a given message and proof. The recursion works for at most a constant number of times, because each extraction potentially blows up the size of the extractor by a polynomial, and that is why we need $\mathsf{d}(\mathbb{C}) = O(1)$. (See Remark 6.3 for more details.) Still, we next show that constant-depth compliance predicates can already be quite expressive.

2.3 The PCD Depth-Reduction Theorem

PCD systems for constant-depth compliance predicates are significantly more powerful than SNARKs; yet, they may seem at first sight to not be as expressive as we would like. In general, we may be interested in compliance predicates of polynomial depth, i.e., that allow for compliant distributed computations that are polynomially deep. To alleviate this restriction, we prove that PCD systems for constant-depth compliance predicates can "bootstrap themselves" to yield PCD systems for polynomial-depth compliance predicates, at least for distributed computations that evolve over a path. Specifically, in a **path** PCD system, *completeness* does not necessarily hold for any compliant distributed computation, but only for those where the associated graph is a path, i.e., each node has only a single input message. We show:

Theorem 2 (PCD Depth Reduction— informal)**.** *Assume there exist collision-resistant hash functions. Any PCD system for constant-depth compliance predicates can be efficiently transformed into a corresponding path PCD system for polynomial-depth compliance predicates. The verifiability properties carry over, as do efficiency properties. (The result also holds for additional classes of graphs; see Remark 8.8.)*

At high-level, the proof consists of two main steps:

- *Step 1.* Say that $\mathbb{C}$ has polynomial depth $\mathsf{d}(\mathbb{C}) = k^c$. We design a new compliance predicate $\textsc{Tree}_{\mathbb{C}}$ of *constant* depth c that is a "tree version" of $\mathbb{C}$. Essentially, $\textsc{Tree}_{\mathbb{C}}$ forces any distributed computation that is compliant with it to be structured in the form of a k-ary tree whose leaves are $\mathbb{C}$-compliant nodes of a computation along a path, and whose internal nodes aggregate information about the computation. A message at the root of the tree is $\textsc{Tree}_{\mathbb{C}}$-compliant only if the leaves of the tree have been "filled in" with a $\mathbb{C}$-compliant distributed computation along a path.
- *Step 2.* We then design a new PCD system $(\mathbb{G}', \mathbb{P}'_{\mathbb{C}}, \mathbb{V}'_{\mathbb{C}})$ based on $(\mathbb{G}, \mathbb{P}_{\textsc{Tree}_{\mathbb{C}}}, \mathbb{V}_{\textsc{Tree}_{\mathbb{C}}})$ that, intuitively, dynamically builds a (shallow k-ary) Merkle tree of proofs "on top" of an original distributed computation. Thus, the new prover $\mathbb{P}'_{\mathbb{C}}$ at a given "real" node along the path will run $\mathbb{P}_{\textsc{Tree}_{\mathbb{C}}}$ for each "virtual" node in a slice of the tree constructed so far; roughly, $\mathbb{P}_{\textsc{Tree}_{\mathbb{C}}}$ will be responsible for computing the proof of the current virtual leaf, as well as merging any internal virtual node proofs that can be bundled together into new proofs, and forwarding all these proofs to the next real node. The number of proofs sent between real nodes is small: at most ck. The new verifier $\mathbb{V}'_{\mathbb{C}}$ will run $\mathbb{V}_{\textsc{Tree}_{\mathbb{C}}}$ for each subproof in a given proof of the new PCD system.

Essentially, the above technique combines the wide Merkle tree idea used in the construction of SNARKs in [BCCT12, GLR11] and (once properly abstracted to the language of PCD) the idea of Valiant [Val08] for building proofs "on top" of a computation in the special case of IVC. For the above high-level intuition to go through, there are still several technical challenges to deal with; we account for these in the full construction and the proof of the theorem in Section 8.

Effect of preprocessing. When the starting PCD system is a preprocessing one, there is a bound B on the computation allowed at any node. Using a preprocessing PCD system in the PCD Depth-Reduction Theorem yields a preprocessing *path* PCD system where the bound on the computation allowed at each node along the path is equal to the one of the starting PCD system, up to polynomial factors in k.

2.4 The Locally-Efficient RAM Compliance Theorem

So far we have shown how, given any SNARK, we can obtain a PCD system for constant-depth compliance predicates, and then obtain a path PCD system for polynomial-depth compliance predicates; both PCD systems inherit the efficiency and verifiability features of the given SNARK.

We now discuss the last ingredient required for our main technical result. Looking ahead, our proof strategy to achieve complexity preservation, say, in a SNARK will be to reduce the task of verifying an NP statement "$\exists\ w$ s.t. $M(x,w) = 1$ in time t" to the task of verifying that a path distributed computation is compliant with a corresponding (polynomial-depth) compliance predicate $\mathbb{C}_{(M,x,t)}$. We can then verify compliance with $\mathbb{C}_{(M,x,t)}$ of such a distributed computation by using the path PCD system we constructed from the SNARK. Moreover, if we can ensure that each node along the path of the distributed computation only performs a *small* amount of computation, then we can "localize" the impact of any inefficiency of the path PCD system. Concretely, preprocessing becomes inexpensive (because it only needs to budget for small local computations), and computing a proof of compliance for the entire distributed computation can be done in roughly the same time and space as those required to compute $M(x,w)$.

At high-level, to achieve the above, we engineer the compliance predicate $\mathbb{C}_{(M,x,t)}$ so to force any distributed computation compliant with $\mathbb{C}_{(M,x,t)}$ to verify the computation of the random-access machine M on x (and some witness w), *one step at a time* for at most t steps. While verifying a single step of M seems like a "small and local" computation, such verification takes time at least linear in the size of M's state, which can be as large as M's space complexity s. Because s could be on the order of t, naively breaking the computation of M into many single-step computations does *not* yield small-enough local computations.

To overcome this problem, we proceed in two steps. First, we invoke a reduction by Ben-Sasson et al. [BSCGT13]: given collision-resistant hashing, the problem of verifying an NP statement "$\exists\ w$ s.t. $M(x,w) = 1$ in time t" can be reduced to the simpler task of verifying a new NP statement "$\exists\ w$ s.t. $M'(x,w) = 1$ in time t'", where M' is a $\mathrm{poly}(k)$-space machine and $t' = t \cdot \mathrm{poly}(k)$. The reduction follows from techniques for *online memory checking* of Blum et al. [BEG+91], which use Merkle hashing to outsource the machine's memory and dynamically verify its consistency using only a small $\mathrm{poly}(k)$-size "trusted" memory. Second, we engineer a compliance predicate for ensuring correct computation of M', one state transition at a time. Crucially, the overall reduction allows to compute a compliant distributed computation using the same time and space as those originally required by M (up to $\mathrm{poly}(k)$ factors).

We now state informally the result; for details, see Section 7.

Theorem 3 (Locally-Efficient RAM Compliance — informal)**.** *Let $\mathcal{H}$ be a family of collision-resistant hash functions. There is a linear-time transformation from any instance (M,x,t) and function $h \in \mathcal{H}$ to a compliance predicate $\mathbb{C}_{(M,x,t),h}$ with depth $t \cdot \mathrm{poly}(k)$ satisfying the following properties.*

1. **Completeness:** *Given w such that $M(x,w)$ accepts in time t and space s, one can generate, in time $(|M| + |x| + t) \cdot \mathrm{poly}(k)$ and space $(|M| + |x| + s) \cdot \mathrm{poly}(k)$, a distributed computation on a path that is compliant with $\mathbb{C}_{(M,x,t),h}$. Each node in the distributed computation performs $\mathrm{poly}(k + |M| + |x|)$ work.*
2. **Proof of knowledge:** *From any efficient adversary that, given a random h, outputs a distributed computation compliant with $\mathbb{C}_{(M,x,t),h}$, we can efficiently extract w such that $M(x,w)$ accepts in time t.*

2.5 Putting Things Together: A General Technique for Preserving Complexity

Equipped with the SNARK Recursive Composition, PCD Depth-Reduction, and Locally-Efficient RAM Compliance Theorems, we restate our main theorem and sketch its proof.

Theorem 4 (Main Theorem, restated)**.** *Let $\mathcal{H}$ be a collision-resistant hash-function family.*

1. ***Complexity-Preserving SNARK from any SNARK.*** *There is an efficient transformation $\mathbb{T}_{\mathcal{H}}$ such that for any publicly-verifiable SNARK $(\mathcal{G}, \mathcal{P}, \mathcal{V})$ there is a polynomial p for which $(\mathcal{G}^\star, \mathcal{P}^\star, \mathcal{V}^\star) := \mathbb{T}_{\mathcal{H}}(\mathcal{G}, \mathcal{P}, \mathcal{V})$ is a publicly-verifiable SNARK that is complexity-preserving with a polynomial p, i.e.,*
   - *the generator $\mathcal{G}^\star$ runs in time $p(k)$ (in particular, there is no expensive preprocessing);*
   - *the prover $\mathcal{P}^\star$ runs in time $t \cdot p(k)$ and space $s \cdot p(k)$ when proving that a t-time s-space NP random-access machine M non-deterministically accepts an input x;*
   - *the verifier $\mathcal{V}^\star$ runs in time $|x| \cdot p(k)$.*
2. ***Complexity-Preserving PCD from any SNARK.*** *There is an efficient transformation $\mathbb{T}'_{\mathcal{H}}$ such that for any publicly-verifiable SNARK $(\mathcal{G}, \mathcal{P}, \mathcal{V})$ there is a polynomial p for which $(\mathbb{G}^\star, \mathbb{P}^\star, \mathbb{V}^\star) := \mathbb{T}'_{\mathcal{H}}(\mathcal{G}, \mathcal{P}, \mathcal{V})$ is a publicly-verifiable PCD for constant-depth compliance predicates that is complexity-preserving with polynomial p, i.e.,*
   - *the generator $\mathbb{G}^\star$ runs in time $p(k)$;*
   - *the prover $\mathbb{P}^\star$ runs in time $t \cdot p(k)$ and space $s \cdot p(k)$ when proving that a message $\mathsf{z_o}$ is $\mathbb{C}$-compliant, using local input linp and received inputs $\vec{\mathsf{z}}_\mathsf{i}$, and evaluating $\mathbb{C}(\mathsf{z_o}; \mathsf{linp}, \vec{\mathsf{z}}_\mathsf{i})$ takes time t and space s;*
   - *the verifier $\mathbb{V}^\star$ runs in time $|\mathsf{z_o}| \cdot p(k)$.*

Assuming fully-homomorphic encryption, similar statements hold for the designated-verifier cases.

Proof sketch. We first sketch the proof to the first item; we follow the plan outlined in Section 1.4. Let $(\mathcal{G}, \mathcal{P}, \mathcal{V})$ be *any* SNARK, and assume (for the worst) that it has expensive preprocessing. We invoke the SNARK Recursive Composition Theorem to obtain a corresponding PCD system $(\mathbb{G}, \mathbb{P}, \mathbb{V})$ for constant-depth compliance predicates, and then the PCD Depth-Reduction Theorem to obtain a corresponding path PCD system $(\mathbb{G}', \mathbb{P}', \mathbb{V}')$ for polynomial-depth compliance predicates. Both transformations preserve the verifiability and efficiency of the SNARK (including preprocessing).

We now use $(\mathbb{G}', \mathbb{P}', \mathbb{V}')$ to construct a complexity-preserving SNARK $(\mathcal{G}^\star, \mathcal{P}^\star, \mathcal{V}^\star)$ as follows. The new generator $\mathcal{G}^\star$, given input 1^k, outputs $(\sigma', \tau') := ((h, \sigma), (h, \tau))$, where $h \leftarrow \mathcal{H}_k$, $(\sigma, \tau) \leftarrow \mathbb{G}'(1^k, k^c)$, and c is a constant that only depends on $(\mathcal{G}, \mathcal{P}, \mathcal{V})$. The new prover $\mathcal{P}^\star$, given a reference string σ', instance (M, x, t), and a witness w, invokes the Locally-Efficient RAM Compliance Theorem in order to compute the polynomial-depth compliance predicate $\mathbb{C}_{(M,x,t),h}$ and, using the prover $\mathbb{P}'$, computes a proof for each message in the path distributed computation obtained from (M, x, t) and w (each time using the previous proof); it outputs the final such proof as the SNARK proof. (We assume, without loss of generality, that $|M|$ and $|x|$ are bounded by a fixed $\mathrm{poly}(k)$; if that is not the case (e.g., M encodes a large non-uniform circuit), $\mathcal{P}^\star$ can work with a new instance $(U_h, \tilde{x}, \mathrm{poly}(k) + t)$, where U_h is a universal random-access machine that, on input $(\tilde{x}, \tilde{w})$, parses $\tilde{w}$ as (M, x, t, w), verifies that $\tilde{x} = h(M, x, t)$, and then runs $M(x, w)$ for at most t steps.) The new verifier $\mathcal{V}^\star$ similarly deduces $\mathbb{C}_{(M,x,t),h}$ and uses $\mathbb{V}'$ to verify a proof.

Overall, we "localized" the use of the (inefficient) PCD system $(\mathbb{G}', \mathbb{P}', \mathbb{V}')$ (obtained from the inefficient SNARK $(\mathcal{G}, \mathcal{P}, \mathcal{V})$), so the SNARK $(\mathcal{G}^\star, \mathcal{P}^\star, \mathcal{V}^\star)$ is complexity preserving.

To obtain the second item of the theorem, we invoke again the SNARK Recursive Composition Theorem and the PCD Depth-Reduction Theorem, but this time with the complexity-preserving SNARK $(\mathcal{G}^\star, \mathcal{P}^\star, \mathcal{V}^\star)$; the resulting PCD systems are complexity preserving. □

See Figure 2 for a summary of how our theorems come together and Section 9 for more details.

Instantiations. Our theorem provides a technique to improve the algorithmic properties of *any* SNARK. For concreteness, let us discuss what we obtain via our theorem from known SNARK constructions.

From preprocessing SNARKs. When plugging into our theorem any of the publicly-verifiable preprocessing SNARKs in [Gro10, Lip12, GGPR12, BCI+13] (each of which can, roughly, be based on "knowledge-of-exponent" [Dam92, BP04] and variants of computational Diffie-Hellman assumptions in bilinear groups),

we obtain the *first* constructions, in the plain model, of publicly-verifiable SNARKs and PCD systems that are *complexity-preserving* (and, in particular, have *no* expensive preprocessing).

The aforementioned preprocessing SNARKs do not invoke the PCP Theorem but instead rely on simpler probabilistic-checking techniques (which can be cast as *linear* PCPs [BCI$^+$13]). While at first sight, these techniques seem to inherently require an expensive preprocessing, our transformation shows that, in fact, they can be used to obtain stronger solutions with *no* preprocessing (in fact, that are complexity-preserving), still without invoking the PCP Theorem.

From PCP-based SNARKs. When plugging into our theorem any of the PCP-based SNARKs in [Mic00, BCCT12, DFH12, GLR11], we obtain complexity-preserving SNARKs based on the PCP Theorem; this, regardless of how poor is the time or space complexity of the PCP in the SNARK we start with. In particular, our theorem circumvents the seemingly-inherent suboptimal time-space tradeoffs of PCP-based SNARKs.

Technical comparison. Our main theorem says that PCD systems for a large class of distributed computations can be obtained from collision-resistant hashing and any SNARK (that may have expensive preprocessing). Our theorem does *not* rely on the SNARK knowledge extractor being very fast; we only assume that the extractor is of polynomial size.

For convenience, we conclude by spelling out what our PCD constructions imply, via compliance engineering (see Section 1.3), for the special cases of incrementally-verifiable computation (IVC) and targeted malleability (TM) and how it compares to the relevant previous work. Valiant [Val08] obtained IVC for every $\mathrm{poly}(k)$-space machine from publicly-verifiable SNARKs having very efficient knowledge extractors; we obtain IVC for *any* machine, under the same assumptions as our theorem. Boneh, Segev, and Waters [BSW12] obtained TM for constant-depth distributed computations and a reference string as long as the entire computation, from publicly-verifiable preprocessing SNARKs; we obtain TM, with $\mathrm{poly}(k)$-size reference string, for distributed computations that are of constant depth or polynomially-long paths, under the same assumptions as our theorem.

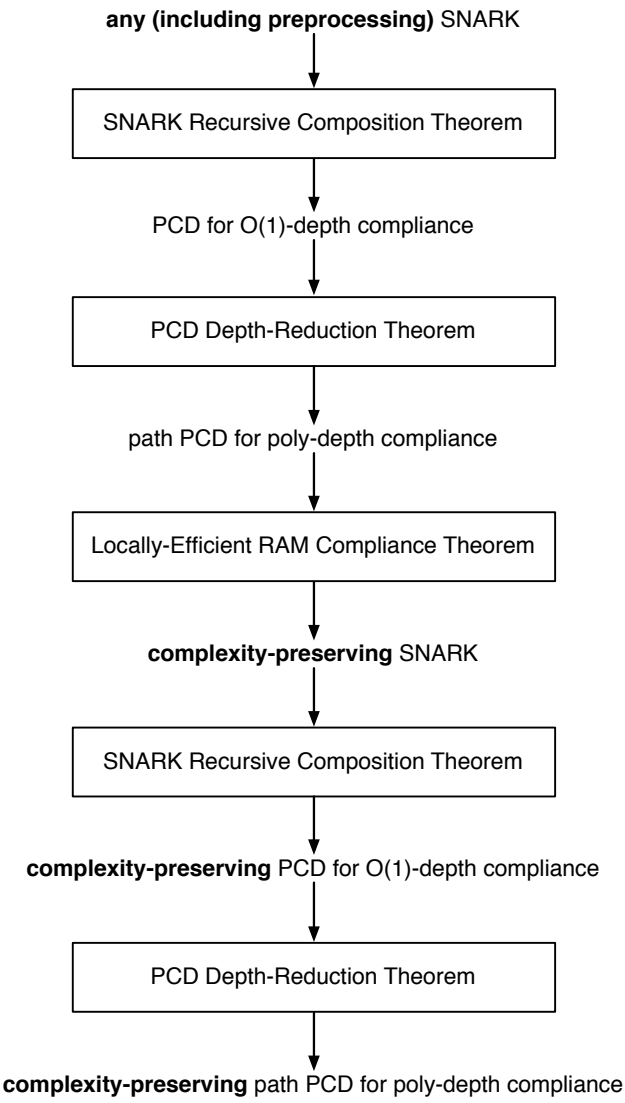

Figure 2: Summary of how our three main results come together; see Section 2.5 for a high-level discussion. Starting from *any* SNARK, our main result produces a corresponding *complexity-preserving* SNARK and PCD system (for a large class of distributed computations and compliance predicates).

3 The Universal Language on Random-Access Machines

We define the *universal relation* [BG08] (along with related notions), which provides us with a canonical form to represent verification-of-computation problems. Because, the notion of *preserving complexity* (of SNARKs and PCD schemes) is defined relative to *random-access machines* [CR72, AV77], we make them our choice of abstract machine for the universal relation.[2] Doing so is also convenient because verification-of-computation problems typically arise in the form of *algorithms* (e.g., "is there w that makes algorithm $\boldsymbol{A}$ accept (x, w)?").

Definition 3.1. *The* **universal relation** *is the set $\mathcal{R}_{\mathcal{U}}$ of instance-witness pairs $(y, w) = \big((M, x, t), w\big)$, where $|y|, |w| \leq t$ and M is a random-access machine, such that M accepts (x, w) after at most t steps.*[3] *We denote by $\mathcal{L}_{\mathcal{U}}$ the* **universal language** *corresponding to $\mathcal{R}_{\mathcal{U}}$.*

For any $c \in \mathbb{N}$, we denote by $\mathcal{R}_c$ the subset of $\mathcal{R}_{\mathcal{U}}$ consisting of those pairs $(y, w) = \big((M, x, t), t\big)$ for which $t \leq |x|^c$; in other words, $\mathcal{R}_c$ is a "generalized" NP relation, where we do not insist on the same machine accepting different instances, but only insist on a fixed polynomial bounding the running time in terms of the instance size. We denote by $\mathcal{L}_c$ the language corresponding to $\mathcal{R}_c$.

4 SNARKs

A *succinct non-interactive argument* (SNARG) is a triple of algorithms $(\mathcal{G}, \mathcal{P}, \mathcal{V})$ that works as follows. The generator $\mathcal{G}$, on input the security parameter k and a time bound B, samples a *reference string* σ and a corresponding *verification state* τ. The honest prover $\mathcal{P}(\sigma, y, w)$ produces a proof π for the statement $y = (M, x, t)$ given a valid w, provided that $t \leq B$; then $\mathcal{V}(\tau, y, \pi)$ deterministically verifies π.

The SNARG is *adaptive* if the prover may choose the statement *after* seeing σ, otherwise, it is *non-adaptive*. The SNARG is *fully-succinct* if $\mathcal{G}$ runs "fast", otherwise, it is of the *preprocessing* kind.

Definition 4.1. *A triple of algorithms $(\mathcal{G}, \mathcal{P}, \mathcal{V})$, where $\mathcal{G}$ is probabilistic and $\mathcal{V}$ is deterministic, is a* **SNARG** *for the relation $\mathcal{R}_{\mathcal{U}}$ if the following conditions are satisfied:*

1. *Completeness*

   For every large enough security parameter $k \in \mathbb{N}$, every time bound $B \in \mathbb{N}$, and every instance-witness pair $(y, w) = \big((M, x, t), w\big) \in \mathcal{R}_{\mathcal{U}}$ with $t \leq B$,

$$\Pr\left[\mathcal{V}(\tau, y, \pi) = 1 \;\middle|\; \begin{array}{c}(\sigma, \tau) \leftarrow \mathcal{G}(1^k, B) \\ \pi \leftarrow \mathcal{P}(\sigma, y, w)\end{array}\right] = 1 \ .$$

2. *Soundness (depending on which notion is considered)*

[2] While random-access machines can be (nondeterministically) simulated by multitape Turing machines with only polylogarithmic overhead in running time [Sch78, GS89], the space complexity of the random-access machine is *not* preserved by this simulation. It is not known how to avoid the large space usage of this simulation. Thus, it is indeed important that we define the universal relation with respect to random-access machines and not Turing machines.

[3] While the witness w for an instance $y = (M, x, t)$ has size at most t, there is *no a-priori* polynomial bounding t in terms of $|x|$. Also, the restriction that $|y|, |w| \leq t$ simplifies notation but comes with essentially no loss of generality: see [BSCGT13] for a discussion of how to deal with "large inputs" (i.e., x or w much larger than t, in the model where M has random access to them).

- ***non-adaptive:*** *For every polynomial-size prover $\mathcal{P}^*$, every large enough security parameter $k \in \mathbb{N}$, every time bound $B \in \mathbb{N}$, and every instance $y = (M, x, t) \notin \mathcal{L}_{\mathcal{U}}$,*

$$\Pr\left[\mathcal{V}(\tau, y, \pi) = 1 \;\middle|\; \begin{array}{c}(\sigma, \tau) \leftarrow \mathcal{G}(1^k, B) \\ \pi \leftarrow \mathcal{P}^*(\sigma, y)\end{array}\right] \leq \mathrm{negl}(k) \ .$$

- ***adaptive:*** *For every polynomial-size prover $\mathcal{P}^*$, every large enough security parameter $k \in \mathbb{N}$, and every time bound $B \in \mathbb{N}$,*

$$\Pr\left[\begin{array}{c}\mathcal{V}(\tau, y, \pi) = 1 \\ y \notin \mathcal{L}_{\mathcal{U}}\end{array} \;\middle|\; \begin{array}{c}(\sigma, \tau) \leftarrow \mathcal{G}(1^k, B) \\ (y, \pi) \leftarrow \mathcal{P}^*(\sigma)\end{array}\right] \leq \mathrm{negl}(k) \ .$$

3. *Efficiency*

   There exists a universal polynomial p such that, for every large enough security parameter $k \in \mathbb{N}$, every time bound $B \in \mathbb{N}$, and every instance $y = (M, x, t)$ with $t \leq B$,

   - *the generator $\mathcal{G}(1^k, B)$ runs in time* $\begin{cases} p(k + B) & \text{for a fully-succinct SNARG} \\ p(k + \log B) & \text{for a preprocessing SNARG} \end{cases}$;
   - *the prover $\mathcal{P}(\sigma, y, w)$ runs in time* $\begin{cases} p(k + |M| + |x| + t + \log B) & \text{for a fully-succinct SNARG} \\ p(k + |M| + |x| + B) & \text{for a preprocessing SNARG} \end{cases}$;
   - *the verifier $\mathcal{V}(\tau, y, \pi)$ runs in time $p(k + |M| + |x| + \log B)$;*
   - *an honestly generated proof has size $p(k + \log B)$.*

A *complexity-preserving* SNARG is a SNARG where the generator, prover, and verifier complexities are essentially optimal:

Definition 4.2. *A triple of algorithms $(\mathcal{G}, \mathcal{P}, \mathcal{V})$ is a* **complexity-preserving SNARG** *if it is a SNARG where efficiency is replaced by the following stronger requirement:*

Complexity-preserving efficiency

There exists a universal polynomial p such that, for every large enough security parameter $k \in \mathbb{N}$, every time bound $B \in \mathbb{N}$, and every instance $y = (M, x, t)$ with $t \leq B$,

- *the generator $\mathcal{G}(1^k, B)$ runs in time $p(k + \log B)$;*
- *the prover $\mathcal{P}(\sigma, y, w)$ runs in time $(|M| + |x| + t) \cdot p(k + \log B)$;*
- *the prover $\mathcal{P}(\sigma, y, w)$ runs in space $(|M| + |x| + s) \cdot p(k + \log B)$;*
- *the verifier $\mathcal{V}(\tau, y, \pi)$ runs in time $(|M| + |x| + \log t) \cdot p(k + \log B)$;*
- *an honestly generated proof has size $p(k + \log B)$.*

A *SNARG of knowledge*, or SNARK for short, is a SNARG where soundness is strengthened as follows:

Definition 4.3. *A triple of algorithms $(\mathcal{G}, \mathcal{P}, \mathcal{V})$ is a* **SNARK** *if it is a SNARG where adaptive soundness is replaced by the following stronger requirement:*

Adaptive proof of knowledge[4]

For every polynomial-size prover $\mathcal{P}^*$ there exists a polynomial-size extractor $\mathcal{E}_{\mathcal{P}^*}$ such that for every large enough security parameter $k \in \mathbb{N}$, every auxiliary input $z \in \{0,1\}^{\mathrm{poly}(k)}$, and every time bound $B \in \mathbb{N}$

$$\Pr\left[\begin{array}{c|l} \mathcal{V}(\tau,y,\pi)=1 & (\sigma,\tau)\leftarrow\mathcal{G}(1^k,B) \\ (y,w)\notin\mathcal{R}_{\mathcal{U}} & (y,\pi)\leftarrow\mathcal{P}^*(z,\sigma) \\ & \quad w\leftarrow\mathcal{E}_{\mathcal{P}^*}(z,\sigma)\end{array}\right]\leq \mathrm{negl}(k)\ .$$

One may want to distinguish between the case where the verifier state σ is allowed to be public or needs to remain private. Specifically, a *publicly-verifiable SNARK* (pvSNARK) is one where security holds even if σ is public; in contrast, a *designated-verifier SNARK* (dvSNARK) is one where σ needs to remain secret.

The SNARKs given in Definition 4.3 are for the universal relation $\mathcal{R}_{\mathcal{U}}$ and are called *universal SNARKs*.[5] In this work, we neither rely on nor achieve universal SNARKs. Instead, we rely on and achieve SNARKs for NP: these are SNARKs in which, when the verifier $\mathcal{V}$ is given as additional input a constant $c > 0$, proof of knowledge only holds with respect to the NP relation $\mathcal{R}_c \subset \mathcal{R}_{\mathcal{U}}$ (see Section 3). (Even in a SNARK for NP, though, the polynomial p governing the efficiency of the SNARK is still required to be *universal*, that is, independent of c.) Thus, *everywhere in this paper, when we say "SNARK", we mean "SNARK for* NP*"*. (And this is indeed the definition of SNARK studied, and achieved, by previous work.)

The technical difference between a universal SNARK and a SNARK for NP will not matter much for most of the paper, except for when proving the SNARK Recursive Composition Theorem in Section 6 (and this is why we first give the more natural definition of a universal SNARK). For completeness, we now also define a SNARK for NP.

Definition 4.4. *A* **SNARK for** NP *is defined as in Definition 4.3,* except *that proof of knowledge is replaced by the following weaker requirement:*

Adaptive proof of knowledge for NP

For every polynomial-size prover $\mathcal{P}^*$ there exists a polynomial-size extractor $\mathcal{E}_{\mathcal{P}^*}$ such that for every large enough security parameter $k \in \mathbb{N}$, every auxiliary input $z \in \{0,1\}^{\mathrm{poly}(k)}$, every time bound $B \in \mathbb{N}$, and every constant $c > 0$,

$$\Pr\left[\begin{array}{c|l} \mathcal{V}_c(\tau,y,\pi)=1 & (\sigma,\tau)\leftarrow\mathcal{G}(1^k,B) \\ (y,w)\notin\mathcal{R}_c & (y,\pi)\leftarrow\mathcal{P}^*(z,\sigma) \\ & \quad w\leftarrow\mathcal{E}_{\mathcal{P}^*}(z,\sigma)\end{array}\right]\leq \mathrm{negl}(k)\ .$$

Remark 4.5 (fully-succinct SNARKs for NP)**.** In a fully-succinct SNARK for NP, there is no need to provide a time bound B to $\mathcal{G}$, because we can set $B := k^{\log k}$. We can then write $\mathcal{G}(1^k)$ to mean $\mathcal{G}(1^k, k^{\log k})$; then, because $\log B = \mathrm{poly}(k)$, $\mathcal{G}$ will run in time $\mathrm{poly}(k)$, $\mathcal{P}$ in time $\mathrm{poly}(k + |M| + |x| + t)$, and so on.

Remark 4.6 (multi-instance extraction)**.** In this work we perform recursive extraction along tree structures. In particular, we will be interested in provers producing a vector of instances $\vec{y}$ together with a vector of corresponding proofs $\vec{\pi}$. In such cases, it is convenient to use an extractor that can extract a vector of witnesses $\vec{w}$ containing a valid witness for each proof accepted by the SNARK verifier. This notion of extraction can be shown to follow from the "single-instance" extraction notion of Definition 4.3.

[4] One can also formulate weaker proof of knowledge notions; in this work we focus on the above strong notion.

[5] Barak and Goldreich [BG08] define universal arguments for $\mathcal{R}_{\mathcal{U}}$ with a black-box "weak proof-of-knowledge" property. In contrast, our proof of knowledge property is not restricted to black-box extractors, and does not allow the extractor to be an implicit representation of a witness.

Lemma 4.7 (adaptive proof of knowledge for instance vectors)**.** *Let $(\mathcal{G}, \mathcal{P}, \mathcal{V})$ be a SNARK (as in Definition 4.3). Then for any polynomial-size prover $\mathcal{P}^*$ there exists a polynomial-size extractor $\mathcal{E}_{\mathcal{P}^*}$ such that for every large enough security parameter $k \in \mathbb{N}$, every auxiliary input $z \in \{0,1\}^{\mathrm{poly}(k)}$, and every time bound $B \in \mathbb{N}$,*

$$\Pr\left[\exists\, i \text{ s.t. } \begin{array}{c}\mathcal{V}(\tau, y_i, \pi_i) = 1 \\ (y_i, w_i) \notin \mathcal{R}_{\mathcal{U}}\end{array} \;\middle|\; \begin{array}{r}(\sigma,\tau) \leftarrow \mathcal{G}(1^k, B) \\ (\vec{y}, \vec{\pi}) \leftarrow \mathcal{P}^*(z, \sigma) \\ \vec{w} \leftarrow \mathcal{E}_{\mathcal{P}^*}(z, \sigma)\end{array}\right] \leq \mathrm{negl}(k) \ .$$

Remark 4.8 (security in the presence of a verification oracle)**.** A desirable property (especially so when preprocessing is expensive) is the ability to generate σ once and for all and then reuse it in polynomially-many proofs (potentially by different provers). Doing so requires security also against provers that *have access to a proof-verification oracle*, namely, have oracle access to $\mathcal{V}(\tau, \cdot, \cdot)$. For pvSNARKs, this *multi-theorem proof of knowledge*[6] is automatically guaranteed. For dvSNARKs, however, multi-theorem proof of knowledge needs to be required explicitly as an additional property. Usually, this is achieved by ensuring that the verifier's response "leaks" only a negligible amount of information about τ.[7] The transformations presented in this paper will preserve multi-theorem proof of knowledge; see [BCI+13] for a formal definition of the property.

Remark 4.9 (generation assumptions)**.** Depending on the model and required properties, there may be different trust assumptions about who runs $\mathcal{G}(1^k)$ to obtain (σ, τ), publish σ, and make sure the verifier has access to τ. For example, in a dvSNARK, the verifier may run $\mathcal{G}$ himself and then publish σ (or send it to the appropriate prover when needed) and keep τ secret for later; in such a case, we may think of σ as a *verifier-generated* reference string. As another example, in a pvSNARK, the verifier may run $\mathcal{G}$ and then publish σ; other verifiers, if they do not trust him, may have to run their own $\mathcal{G}$ to obtain outputs that they trust; alternatively, we could assume that σ is a *global reference string* that everyone trusts. For both dvSNARKs and pvSNARKs, when requiring a zero-knowledge property, we must assume that σ is a *common* reference string (i.e., a trusted party ran $\mathcal{G}$). The transformations presented in this paper will preserve zero-knowledge, whenever available; in this paper, though, *we do not study zero knowledge*.

Remark 4.10 (the SNARK extractor $\mathcal{E}$)**.** In Definition 4.3, we require that any polynomial-size family of circuits $\mathcal{P}^*$ has a specific polynomial-size family of extractors $\mathcal{E}_{\mathcal{P}^*}$. In particular, we allow the extractor to be of arbitrary polynomial-size and to be "more non-uniform" than $\mathcal{P}^*$. In addition, we require that, for any prover auxiliary input $z \in \{0,1\}^{\mathrm{poly}(k)}$, the polynomial-size extractor manages to perform its witness-extraction task given the same auxiliary input z. The definition can be naturally relaxed to consider only specific distributions of auxiliary inputs according to the required application.

One could consider stronger notions in which the extractor is a uniform machine that gets $\mathcal{P}^*$ as input, or even only gets black-box access to $\mathcal{P}^*$. (For the case of adaptive SNARGs, this notion cannot be achieved based on black-box reductions to falsifiable assumptions [GW11].) In common security reductions, however, where the primitives (to be broken) are secure against arbitrary polynomial-size non-uniform adversaries, the non-uniform notion seems to suffice (and is indeed the one we adopt in Definition 4.3). The transformations presented in this paper preserve the notion of extraction; e.g., if you start with a SNARK with uniform extraction, then you will obtain a complexity-preserving SNARK with uniform extraction too.

[6] Security against such provers can be formulated for soundness or proof of knowledge, both in the non-adaptive and adaptive regimes. Because in this paper we are most interested in adaptive proof of knowledge, we shall refer to this setting.

[7] Note that $O(\log k)$-theorem soundness always holds; the "non-trivial" case is whenever $\omega(\log k)$. Weaker solutions to support more theorems include simply assuming that the verifier's responses remain secret (so that there is no leakage on τ), or re-generating σ every logarithmically-many rejections, e.g., as in [KR06, GKR08, KR09, GGP10, CKV10].

5 Proof-Carrying Data

In Section 5.1, we begin by specifying the (syntactic) notion of a distributed computation that is considered in proof-carrying data, the notion of *compliance*, and other auxiliary notions. Then, in Section 5.2, we define *proof-carrying data (PCD) systems*, which are the cryptographic primitive that formally captures the framework for proof-carrying data.

5.1 Distributed Computations And Their Compliance With Local Properties

We view a distributed computation as a directed acyclic[8] graph $G = (V, E)$ with node labels $\mathsf{linp}\colon V \to \{0,1\}^*$ and edge labels $\mathsf{data}\colon E \to \{0,1\}^*$. The node label $\mathsf{linp}(v)$ of a node v represents the *local input* (which may include a local program) used by v in his local computation. (Whenever v is a source or a sink, we require that $\mathsf{linp}(v) = \bot$.) The edge label $\mathsf{data}(u, v)$ of a directed edged (u, v) represents the *message* sent from node u to node v. Typically, a party at node v uses the local input $\mathsf{linp}(v)$ and input messages $(\mathsf{data}(u_1, v), \ldots, \mathsf{data}(u_c, v))$, where $u_1, \ldots, u_c$ are the parents of v in lexicographic order, to compute an output message $\mathsf{data}(v, w)$ for a child node w; the party also similarly computes a message for every other child node. We can think of the messages on edges going out from sources as the "inputs" to the distributed computation, and the messages on edges going into sinks as the "outputs" of the distributed computation; for convenience we will want to identify a single distinguished output.

Definition 5.1. *A* **(distributed computation) transcript** *is a triple* $\mathsf{T} = (G, \mathsf{linp}, \mathsf{data})$*, where* $G = (V, E)$ *is a directed acyclic graph* G*,* $\mathsf{linp}\colon V \to \{0,1\}^*$ *are node labels, and* $\mathsf{data}\colon E \to \{0,1\}^*$ *are edge labels; we require that* $\mathsf{linp}(v) = \bot$ *whenever* v *is a source or a sink. The output of* T*, denoted* $\mathsf{out}(\mathsf{T})$*, is equal to* $\mathsf{data}(\tilde{u}, \tilde{v})$ *where* $(\tilde{u}, \tilde{v})$ *is the lexicographically first edge such that* $\tilde{v}$ *is a sink.*

A proof-carrying transcript is a transcript where messages are augmented by proof strings, i.e., a function $\mathsf{proof}\colon E \to \{0,1\}^*$ provides for each edge (u, v) an additional label $\mathsf{proof}(u, v)$, to be interpreted as a proof string for the message $\mathsf{data}(u, v)$. (This is a syntactic definition; the semantics are discussed in Section 5.2.)

Definition 5.2. *A* **proof-carrying (distributed computation) transcript** PCT *is a pair* $(\mathsf{T}, \mathsf{proof})$ *where* T *is a transcript and* $\mathsf{proof}\colon E \to \{0,1\}^*$ *is an edge label.*

Next, we define what it means for a distributed computation to be *compliant*, which is the notion of "correctness with respect to a given local property". Compliance is captured via an efficiently-computable *compliance predicate* $\mathbb{C}$, which must be locally satisfied at each vertex; here, "locally" means with respect to a node's local input, incoming data, and outgoing data. For convenience, for any vertex v, we let $\mathsf{children}(v)$ and $\mathsf{parents}(v)$ be the vector of v's children and parents respectively, listed in lexicographic order.

Definition 5.3. *Given a polynomial-time predicate* $\mathbb{C}$*, we say that a distributed computation transcript* $\mathsf{T} = (G, \mathsf{linp}, \mathsf{data})$ *is* **$\mathbb{C}$-compliant** *(denoted by* $\mathbb{C}(\mathsf{T}) = 1$*) if, for every* $v \in V$ *and* $w \in \mathsf{children}(v)$*, it holds that*

$$\mathbb{C}(\mathsf{data}(v, w); \mathsf{linp}(v), \mathsf{inputs}(v)) = 1 \ ,$$

where $\mathsf{inputs}(v) := \big(\mathsf{data}(u_1, v), \ldots, \mathsf{data}(u_c, v)\big)$ *and* $(u_1, \ldots, u_c) := \mathsf{parents}(v)$*. Furthermore, we say that a message* z *is* $\mathbb{C}$*-compliant if there is* T *such that* $\mathbb{C}(\mathsf{T}) = 1$ *and* $\mathsf{out}(\mathsf{T}) = 1$*.*

[8] If the same party takes part in the computation at different times, we represent the party as multiple nodes.

Remark 5.4. We emphasize that in Definition 5.3 we consider *one* output message $\mathsf{data}(v, w)$ of v at a time. The reason is that if we were to simultaneously give as input to $\mathbb{C}$ all the output messages of v, then $\mathbb{C}$ may verify non-local properties (e.g., the messages sent to two different parties are the same). Such non-local properties are beyond the scope of the PCD framework; in particular, to enforce such non-local properties, additional communication among the parties may be required.

Remark 5.5 (polynomially-balanced compliance predicates)**.** We restrict our attention to polynomial-time compliance predicates that are also **polynomially balanced** with respect to the outgoing message. Namely, the running time of $\mathbb{C}(\mathsf{z_o}; \vec{\mathsf{z}}_\mathsf{i}, \mathsf{linp})$ is bounded by $t_\mathbb{C}(|\mathsf{z_o}|) = |\mathsf{z_o}|^{e_\mathbb{C}}$, for a constant exponent $e_\mathbb{C}$ that depends only on $\mathbb{C}$. This, in particular, implies that inputs for which $|\mathsf{linp}| + |\vec{\mathsf{z}}_\mathsf{i}|$ is greater than $t_\mathbb{C}(|\mathsf{z_o}|)$ are rejected. This restriction will simplify presentation, and the relevant class of compliance predicates is expressive enough for most applications that come to mind. We also assume that the constant $e_\mathbb{C}$ is hardcoded in the description of $\mathbb{C}$.

A notion that will be very useful to us is that of distributed computation transcripts that are B-bounded:

Definition 5.6. *Given a distributed computation transcript* $\mathsf{T} = (G, \mathsf{linp}, \mathsf{data})$ *and any edge* $(v, w) \in E$*, we denote by* $t_{\mathsf{T},\mathbb{C}}(v, w)$ *the time required to evaluate* $\mathbb{C}(\mathsf{data}(v, w); \mathsf{linp}(v), \mathsf{inputs}(v))$*. We say that* T *is* B**-bounded** *if* $t_{\mathsf{T},\mathbb{C}}(v, w) \leq B$ *for every edge* (v, w)*.*

Remark 5.7. Note that B is a bound on the time to evaluate $\mathbb{C}$ at any node, and not a bound on the sum of all such times. Furthermore, because we only consider polynomial-time computation, we always have a concrete super-polynomial bound, e.g. $t_{\mathsf{T},\mathbb{C}}(v, w) \leq k^{\log k}$, where k is the security parameter.

A property of a compliance predicate that plays an important role in many of our results is that of depth:

Definition 5.8. *The* **depth of a transcript** T*, denoted* $\mathsf{d}(\mathsf{T})$*, is the largest number of nodes on a source-to-sink path in* T *minus* 2 *(to exclude the source and the sink). The* **depth of a a compliance predicate** $\mathbb{C}$*, denoted* $\mathsf{d}(\mathbb{C})$*, is defined to be the maximum depth of any transcript* T *compliant with* $\mathbb{C}$*. If* $\mathsf{d}(\mathbb{C}) := \infty$ *(i.e., paths in* $\mathbb{C}$*-compliant transcripts can be arbitrarily long) we say that* $\mathbb{C}$ *has unbounded depth.*

5.2 Proof-Carrying Data Systems

A *proof-carrying data (PCD) system* for a class of compliance predicates $\mathbf{C}$ is a triple of algorithms $(\mathbb{G}, \mathbb{P}, \mathbb{V})$ that works as follows:

- The (probabilistic) generator $\mathbb{G}$, on input the security parameter k, outputs a *reference string* σ and a corresponding *verification state* τ.
- For any $\mathbb{C} \in \mathbf{C}$, the (honest) prover $\mathbb{P}_\mathbb{C} := \mathbb{P}(\mathbb{C}, \cdots)$ is given a reference string σ, inputs $\vec{\mathsf{z}}_\mathsf{i}$ with corresponding proofs $\vec{\pi}_\mathsf{i}$, a local input linp, and an output $\mathsf{z_o}$, and then produces a proof π_o attesting to the fact that $\mathsf{z_o}$ is consistent with some $\mathbb{C}$-compliant transcript.
- For any $\mathbb{C} \in \mathbf{C}$, the verifier $\mathbb{V}_\mathbb{C} := \mathbb{V}(\mathbb{C}, \cdots)$ is given the verification state τ, an output $\mathsf{z_o}$, and a proof string π_o, and accept if it is convinced that $\mathsf{z_o}$ is consistent with some $\mathbb{C}$-compliant transcript.

After the generator $\mathbb{G}$ has been run to obtain σ and τ, the prover $\mathbb{P}_\mathbb{C}$ is used (along with σ) at each node of a distributed computation transcript to *dynamically* compile it into a proof-carrying transcript by generating and adding a proof to each edge. Each of these proofs can be checked using the verifier $\mathbb{V}_\mathbb{C}$ (along with τ).

As in SNARKs, we say that a PCD system is *fully-succinct* if the generator $\mathbb{G}$ runs "fast", otherwise, it is of the *preprocessing* kind.

The formal definition. We now formally define the notion of PCD systems.[9] We begin by introducing the dynamic proof-generation process, which we call $\mathsf{ProofGen}$. We define $\mathsf{ProofGen}$ as an interactive protocol between a (not necessarily efficient) *distributed-computation generator* S and the PCD prover $\mathbb{P}$, in which both are given a compliance predicate $\mathbb{C} \in \mathbf{C}$ and a reference string σ. Essentially, at every time step, S chooses to do one of the following actions: add a new unlabeled vertex to the computation transcript so far (this corresponds to adding a new computing node to the computation), label an unlabeled vertex (this corresponds to a choice of local input by a computing node), or add a new labeled edge (this corresponds to a new message from one node to another). In case S chooses the third action, the PCD prover $\mathbb{P}_{\mathbb{C}}$ produces a proof for the $\mathbb{C}$-compliance of the new message, and adds this new proof as an additional label to the new edge. When S halts, the interactive protocol outputs the distributed computation transcript T, as well as T's output and corresponding proof. Intuitively, the completeness property requires that if T is compliant with $\mathbb{C}$, then the proof attached to the output (which is the result of dynamically invoking $\mathbb{P}_{\mathbb{C}}$ for each message in T, as T was being constructed by S) is accepted by the verifier. Formally the interactive protocol $\mathsf{ProofGen}(\mathbb{C}, \sigma, S, \mathbb{P})$ is defined as follows:

$\mathsf{ProofGen}(\mathbb{C}, \sigma, S, \mathbb{P}) \equiv$

1. Set T and PCT to be "empty transcripts".
   (That is, $\mathsf{T} = (G, \mathsf{linp}, \mathsf{data})$ and $\mathsf{PCT} = (\mathsf{T}, \mathsf{proof})$ with $G = (V, E) = (\emptyset, \emptyset)$.)
2. Until S halts and outputs a message-proof pair $(\mathsf{z_o}, \pi_{\mathsf{o}})$, do the following:
   (a) Give $(\mathbb{C}, \sigma, \mathsf{PCT})$ as input to S and obtain as output (b, x, y).
   (b) If $b =$ "`add unlabeled vertex`" and $x \notin V$, then set $V := V \cup \{x\}$ and $\mathsf{linp}(x) := \perp$.
   (c) If $b =$ "`label vertex`", $x \in V$, x is nor a source or sink, and $\mathsf{linp}(x) = \perp$, then $\mathsf{linp}(x) := y$.
   (d) If $b =$ "`add labeled edge`" and $x \notin E$:
      i. Parse x as (v, w) with $v, w \in V$.
      ii. Set $E := E \cup \{(v, w)\}$.
      iii. Set $\mathsf{data}(v, w) := y$.
      iv. If v is a source, set $\pi := \perp$.
      v. If v is not a source, set $\pi := \mathbb{P}_{\mathbb{C}}(\sigma, \mathsf{data}(v, w), \mathsf{linp}(v), \mathsf{inputs}(v), \mathsf{inproofs}(v))$, where $\mathsf{inputs}(v) := \big(\mathsf{data}(u_1, v), \ldots, \mathsf{data}(u_c, v)\big)$, $\mathsf{inproofs}(v) := \big(\mathsf{proof}(u_1, v), \ldots, \mathsf{proof}(u_c, v)\big)$, and $(u_1, \ldots, u_c) := \mathsf{parents}(v)$.
      vi. Set $\mathsf{proof}(v, w) := \pi$.
3. Output $(\mathsf{z_o}, \pi_{\mathsf{o}}, \mathsf{T})$.

Having defined $\mathsf{ProofGen}$, we are now ready for the definition:

Definition 5.9. *A* **proof-carrying data system** *for a class of compliance predicates* $\mathbf{C}$ *is a triple of algorithms* $(\mathbb{G}, \mathbb{P}, \mathbb{V})$*, where* $\mathbb{G}$ *is probabilistic and* $\mathbb{V}$ *is deterministic, such that:*

[9] At a technical level, our definition differs slightly than that given in [CT10]. First, we work in the plain model, while [CT10] worked in a model where parties had access to a signature oracle. Second, we limit ourselves to the case where a compliance predicate has a known polynomial running time, while [CT10] uniformly handled all polynomial-time compliance predicates; this difference is analogous to the difference between a universal SNARK and a SNARK for NP (see discussion in Section 4), and is not an important restriction for our purposes.

1. *Completeness*

   For every compliance predicate $\mathbb{C} \in \mathbf{C}$ and (possibly unbounded) distributed computation generator S,

$$\Pr\left[\begin{array}{c|r} \mathsf{T} \text{ is } B\text{-bounded} & (\sigma,\tau) \leftarrow \mathbb{G}(1^k, B) \\ \mathbb{C}(\mathsf{T}) = 1 & (\mathsf{z_o}, \pi_\mathsf{o}, \mathsf{T}) \leftarrow \mathsf{ProofGen}(\mathbb{C}, \sigma, S, \mathbb{P}) \\ \mathbb{V}_\mathbb{C}(\tau, \mathsf{z_o}, \pi_\mathsf{o}) \neq 1 & \end{array}\right] \leq \mathrm{negl}(k) \ .$$

2. *Proof of Knowledge*

   For every polynomial-size prover $\mathbb{P}^*$ there exists a polynomial-size extractor $\mathbb{E}_{\mathbb{P}^*}$ such that for every compliance predicate $\mathbb{C} \in \mathbf{C}$, every large enough security parameter $k \in \mathbb{N}$, every auxiliary input $z \in \{0,1\}^{\mathrm{poly}(k)}$, and every time bound $B \in \mathbb{N}$

$$\Pr\left[\begin{array}{c|r} \mathbb{V}_\mathbb{C}(\tau, \mathsf{z}, \pi) = 1 & (\sigma,\tau) \leftarrow \mathbb{G}(1^k, B) \\ \big(\mathsf{out}(\mathsf{T}) \neq \mathsf{z} \ \vee \ \mathbb{C}(\mathsf{T}) \neq 1\big) & (\mathsf{z}, \pi) \leftarrow \mathbb{P}^*(\sigma, z) \\ & \mathsf{T} \leftarrow \mathbb{E}_{\mathbb{P}^*}(\sigma, z) \end{array}\right] \leq \mathrm{negl}(k) \ .$$

3. *Efficiency*

   There exists a universal polynomial p such that, for every compliance predicate $\mathbb{C} \in \mathbf{C}$, every large enough security parameter $k \in \mathbb{N}$, every time bound $B \in \mathbb{N}$, and every B-bounded distributed computation transcript T,

   - *the generator $\mathbb{G}(1^k, B)$ runs in time* $\begin{cases} p(k+B) & \text{for a fully-succinct PCD} \\ p(k+\log B) & \text{for a preprocessing PCD} \end{cases}$;
   - *the prover $\mathbb{P}_\mathbb{C}(\sigma, \mathsf{data}(v,w), \mathsf{linp}(v), \mathsf{inputs}(v), \vec{\pi}_\mathsf{i})$ runs in time*

$$\begin{cases} p(k + |\mathbb{C}| + t_{\mathsf{T},\mathbb{C}}(v,w) + \log B) & \text{for a fully-succinct PCD} \\ p(k + |\mathbb{C}| + B) & \text{for a preprocessing PCD} \end{cases},$$

     where $t_{\mathsf{T},\mathbb{C}}(v,w)$ denotes the time to evaluate $\mathbb{C}(\mathsf{data}(v,w); \mathsf{linp}(v), \mathsf{inputs}(v))$ at an edge (v,w);
   - *the verifier $\mathbb{V}_\mathbb{C}(\tau, \mathsf{z}, \pi)$ runs in time $p(k + |\mathbb{C}| + |\mathsf{z}| + \log B)$;*
   - *an honestly generated proof has size $p(k + \log B)$.*

We shall also consider a restricted notion of PCD system: a **path PCD system** *is a PCD system where completeness is guaranteed to hold only for distributed computations transcripts T whose graph is a line.*

As with SNARKs (see Section 4), we distinguish between the case where the verifier state τ may be public or needs to remain private. Specifically, a *publicly-verifiable PCD system* is one where security holds even if τ is public. In contrast, a *designated-verifier PCD system* is one where τ needs to remain secret. Similarly to SNARKs, this affects whether security holds in the presence of a proof-verification oracle (see Remark 4.8): in the publicly-verifiable case this property is automatically guaranteed, while in the designated-verifier case this it does not follow directly (besides as usual the trivial guarantees for $O(\log k)$ verifications).

Remark 5.10. In a fully-succinct PCD system, there is no need to provide a time bound B to $\mathbb{G}$, because we can set $B := k^{\log k}$. In such cases, we write $\mathbb{G}(1^k)$ to mean $\mathbb{G}(1^k, k^{\log k})$; then, because $\log B = \mathrm{poly}(k)$, $\mathbb{G}$ will run in time $\mathrm{poly}(k)$, $\mathbb{P}$ in time $p(k + |\mathbb{C}| + t_{\mathsf{T},\mathbb{C}}(v,w))$, and so on.

Remark 5.11 (adversarial compliance predicates)**.** We could strengthen Definition 5.9 by allowing the adversary to choose the (polynomially-balanced) compliance predicate $\mathbb{C}$ for which he produces a message and proof. All of the theorems we discuss in this paper hold with respect to this stronger definition (though one has to be careful about how to formally state the results). For convenience of presentation and also because almost always $\mathbb{C}$ is "under our control", we choose to not explicitly consider this strengthening.

A *complexity-preserving* PCD system is a PCD system where the generator, prover, and verifier complexities are essentially optimal:

Definition 5.12. *A triple of algorithms $(\mathbb{G}, \mathbb{P}, \mathbb{V})$ is a* **complexity-preserving PCD system** *if it is a PCD system where efficiency is replaced by the following stronger requirement:*

Complexity-preserving efficiency

There exists a universal polynomial p such that, for every compliance predicate $\mathbb{C} \in \mathbf{C}$, every large enough security parameter $k \in \mathbb{N}$, every time bound $B \in \mathbb{N}$, and every B-bounded distributed computation transcript T,

- *the generator $\mathbb{G}(1^k, B)$ runs in time $p(k + \log B)$;*
- *the prover $\mathbb{P}_{\mathbb{C}}(\sigma, \mathsf{data}(v, w), \mathsf{linp}(v), \mathsf{inputs}(v), \vec{\pi}_{\mathsf{i}})$ runs in time $(|\mathbb{C}| + t_{\mathsf{T},\mathbb{C}}(v, w)) \cdot p(k + \log B)$, where $t_{\mathsf{T},\mathbb{C}}(v, w)$ denotes the time to evaluate $\mathbb{C}(\mathsf{data}(v, w); \mathsf{linp}(v), \mathsf{inputs}(v))$ at an edge (v, w);*
- *the prover $\mathbb{P}_{\mathbb{C}}(\sigma, \mathsf{data}(v, w), \mathsf{linp}(v), \mathsf{inputs}(v), \vec{\pi}_{\mathsf{i}})$ runs in space $(|\mathbb{C}| + s_{\mathsf{T},\mathbb{C}}(v, w)) \cdot p(k + \log B)$, where $s_{\mathsf{T},\mathbb{C}}(v, w)$ denotes the space to evaluate $\mathbb{C}(\mathsf{data}(v, w); \mathsf{linp}(v), \mathsf{inputs}(v))$ at an edge (v, w);*
- *the verifier $\mathbb{V}_{\mathbb{C}}(\tau, \mathsf{z}, \pi)$ runs in time $(|\mathbb{C}| + |\mathsf{z}|) \cdot p(k + \log B)$;*
- *an honestly generated proof has size $p(k + \log B)$.*

6 Proof Of The SNARK Recursive Composition Theorem

We provide here the technical details for the high-level discussion in Section 2.2. Concretely, we prove the SNARK Recursive Composition Theorem, which is one of the three tools we use in the proof of our main result (discussed in Section 9). Throughout this section, it will be useful to keep in mind the definitions from Section 3 (where the universal language $\mathcal{L}_{\mathcal{U}}$ is introduced), Section 4 (where SNARKs are introduced), Section 5.1 (where the notions of distributed computation transcripts, compliance predicates, and depth are introduced), and Section 5.2 (where PCD systems are introduced).

We prove a composition theorem for "all kinds" of SNARKs: we show how to use a SNARK to obtain a PCD system for constant-depth compliance predicates. More precisely, we present *two* constructions for this task, depending on whether the given SNARK is of the designated-verifier kind or the publicly-verifiable kind. (In particular, we learn that even designated-verifier SNARKs can be recursively composed, which may come as a surprise.) In sum, we learn that the existence of a SNARK implies the existence of a corresponding PCD system, with analogous verifiability and efficiency properties, for every compliance predicate whose depth is constant. (In particular, if the SNARK is of the preprocessing kind, so will the PCD system constructed from it.)

Formally:

Theorem 6.1 (SNARK Recursive Composition Theorem)**.**

1. *There exists an efficient transformation* RECCOMP *such that, for every publicly-verifiable SNARK* $(\mathcal{G}, \mathcal{P}, \mathcal{V})$, $(\mathbb{G}, \mathbb{P}, \mathbb{V}) = \text{RECCOMP}(\mathcal{G}, \mathcal{P}, \mathcal{V})$ *is a publicly-verifiable PCD system for every constant-depth compliance predicate.*

   (If $(\mathcal{G}, \mathcal{P}, \mathcal{V})$ *is a preprocessing SNARK, we further assume the existence of a collision-resistant hash-function family* $\mathcal{H}$*, and* RECCOMP *also depends on* $\mathcal{H}$*.)*

2. *Suppose that* $\mathcal{E}$ *is a fully-homomorphic encryption scheme. There exists an efficient transformation* $\text{RECCOMP}_{\mathcal{E}}$ *such that, for every designated-verifier SNARK* $(\mathcal{G}, \mathcal{P}, \mathcal{V})$, $(\mathbb{G}, \mathbb{P}, \mathbb{V}) = \text{RECCOMP}_{\mathcal{E}}(\mathcal{G}, \mathcal{P}, \mathcal{V})$ *is a designated-verifier PCD system for every constant-depth compliance predicate.*[10]

In Section 6.1 we prove the first part of the theorem (which deals with the publicly-verifiable case), and then in Section 6.2 we prove the second part of the theorem (which deals with the designated-verifier case).

Remark 6.2 (depth-reduction for PCD systems)**.** Constant-depth compliance predicates are not all that weak. Indeed, as discussed informally in Section 2.3 (and in detail in Section 8), the depth of a compliance predicate can always be improved *exponentially*, via the PCD Depth-Reduction Theorem, at least for all distributed computations evolving over paths.

Remark 6.3 (beyond constant depth)**.** In the SNARK Recursive Composition Theorem we have to restrict the depth of compliance predicates to constant because our security reduction is based on a recursive composition of SNARK extractors, where the extractor at a given level of the recursion may incur an arbitrary polynomial blowup in the size of the previous extractor; in particular, if we want the "final" extractors at the leaves of the tree to each have polynomial size, we must make the aforementioned restriction in the depth.

If one is willing to make stronger assumptions regarding the size of the extractor $\mathcal{E}_{\mathcal{P}^*}$ of a prover $\mathcal{P}^*$ then the conclusion of the SNARK Recursive Composition Theorem will be stronger. (Whether such stronger extractability assumptions are plausible or not should be judged on a case-by-case basis. Here we do not condemn or condone their use, but we simply state what are their implications to our theorem.)

Specifically, let us define the *size* of a compliance predicate $\mathbb{C}$, denoted $\mathsf{s}(\mathbb{C})$, to be the largest number of nodes in any transcript compliant with $\mathbb{C}$. Then, for example:

- By assuming that $|\mathcal{E}_{\mathcal{P}^*}| \leq C|\mathcal{P}^*|$ for some constant C (possibly depending on $\mathcal{P}^*$), that is assuming only a *linear blowup*, our result can be strengthened to yield PCD systems for *logarithmic-depth polynomial-size* compliance predicates.

  For instance, if a compliance predicate has $O(\log(k))$ depth and only allows $O(1)$ inputs per node, then it has polynomial size; more generally, if a compliance predicate has depth $\log_w(\text{poly}(k))$ and only allows w inputs per node, then it has polynomial size.

  An extractability assumption of this kind is implicitly used in Valiant's construction of incrementally-verifiable computation [Val08].

- By assuming that $|\mathcal{E}_{\mathcal{P}^*}| \leq |\mathcal{P}^*| + p(k)$ for some polynomial p (possibly depending on $\mathcal{P}^*$), that is assuming only an *additive blowup*, our result can be strengthened to yield PCD systems for *polynomial-size compliance predicates* (which, in particular, have polynomial depth).

  For instance, if a compliance predicate has polynomial depth and only allows one input per node, then it has polynomial size.

[10] We do not require the verification state to be "reusable"; that is, we do not require the SNARK to be secure against provers having access to a proof-verification oracle (see Remark 4.8). If this happens to be the case, then this "multi-theorem" proof-of-knowledge property is preserved by the transformation.

(An alternative way to obtain PCD systems for polynomial-size compliance predicates is to strengthen the extractability assumption to an "interactive online extraction"; see, e.g., [BP04, DFH12] for examples of such assumptions. For example, the kind of extraction that Damgård et al. [DFH12] assume for a collision-resistant hash is enough to construct a SNARK with the interactive online extraction that will in turn be sufficient for obtaining PCD systems for polynomial-size compliance predicates.)

More generally, it is always important that during extraction: (a) we only encounter distributed computation transcripts that are not too deep relative to the blowup of the extractor size, and (b) we only encounter distributed computation transcripts of polynomial size.

When we must limit the depth of a compliance predicate to constant (which we must when the blowup of the extractor may be an arbitrary polynomial), there is no need to limit its size, because any compliance predicate of constant depth must have polynomial size. However, when we limit the depth to a super constant value (which we can afford when making stronger assumptions on the blowup of the extractor), we must also limit the size of the compliance predicate to polynomial.[11]

6.1 Recursive Composition For Publicly-Verifiable SNARKs

We begin by giving the construction and proof, respectively in Section 6.1.1 and in Section 6.1.2, for the publicly-verifiable case with no preprocessing of the SNARK Recursive Composition Theorem (i.e., Item 1 of Theorem 6.1 with no preprocessing); it will be useful to keep in mind Remark 4.5. After that, in Section 6.1.3, we extend the discussion to the case with preprocessing. (And after that, in Section 6.2, we proceed to the designated-verifier case of the theorem.)

6.1.1 The Construction

We are given a publicly-verifiable (fully-succinct) SNARK $(\mathcal{G}, \mathcal{P}, \mathcal{V})$ for NP (see Definition 4.4). To construct a publicly-verifiable (fully-succinct) PCD system $(\mathbb{G}, \mathbb{P}, \mathbb{V})$ for constant-depth compliance predicates, we follow the PCD construction of Chiesa and Tromer [CT10]. At high-level, given a (constant-depth) compliance predicate $\mathbb{C}$, at each node in the distributed computation, the PCD prover $\mathbb{P}$ invokes the SNARK prover $\mathcal{P}$ to generate a SNARK proof attesting to the fact that the node knows (i) input messages (and a local input) that are $\mathbb{C}$-compliant with the claimed output message, *and also* (ii) corresponding proofs attesting that these input messages themselves come from a $\mathbb{C}$-compliant distributed computation. The PCD verifier $\mathbb{V}$ then invokes the SNARK verifier $\mathcal{V}$ on an appropriate statement. More precisely, the construction of the PCD system $(\mathbb{G},\ wPPCD, \mathbb{V})$ is as follows:

- **The PCD generator.** On input the security parameter 1^k, the PCD generator $\mathbb{G}$ runs the SNARK generator $\mathcal{G}(1^k)$ in order to obtain a reference string σ and verification state τ, and then outputs (σ, τ). Without loss of generality, we assume that both σ and τ include the security parameter 1^k in the clear; furthermore, because we are focusing on the publicly-verifiable case, we may also assume that σ includes τ in the clear.

[11] Interestingly, it seems that even if the size of a compliance predicate is not polynomial, a polynomial-size prover should not give rise to distributed computations of super-polynomial-size during extraction, but we do not see how to prove that this is the case. This technical issue is somewhat similar to the difficulty that Bitansky et al. found in constructing *universal* SNARKs in [BCCT12] and not just SNARKs for specific languages. Chiesa and Tromer [CT10] were able to construct PCD systems for emphany compliance predicate, with no restrictions on depth or size, but this was not in the plain model. We believe it to be an interesting open question to make progress on the technical difficulties we find in the plain model with the ultimate goal of understanding what it takes to construct PCD systems for any compliance predicate in the plain model.

Recall that we are temporarily focusing on the case where the publicly-verifiable SNARK $(\mathcal{G}, \mathcal{P}, \mathcal{V})$ is fully-succinct, so that σ and τ have size that is a fixed polynomial in the security parameter k; in particular, σ and τ could be merged into one public parameter, but we choose to keep them separate for the sake of clarity and exposition of the other cases.

- **The PCD prover.** For any compliance predicate $\mathbb{C}$, on input $(\sigma, \mathsf{z_o}, \mathsf{linp}, \vec{\mathsf{z}}_\mathsf{i}, \vec{\pi}_\mathsf{i})$, the PCD prover $\mathbb{P}_\mathbb{C} := \mathbb{P}(\mathbb{C}, \cdots)$ constructs a "SNARK theorem and witness" $(y, w) = ((M, x, t), w)$ and then runs the SNARK prover $\mathcal{P}(\sigma, y, w)$ to produce the outgoing proof π_o to attach to the outgoing message $\mathsf{z_o}$. More precisely (and recalling that τ is part of σ, i.e., it is public), $\mathbb{P}_\mathbb{C}$ chooses y and w as follows:

$$y := \big(\mathsf{M}_{\mathcal{V},\mathbb{C}}, (\mathsf{z_o}, \tau), t_{\mathcal{V},\mathbb{C}}(|\mathsf{z_o}| + |\tau|)\big) \text{ and } w := (\mathsf{linp}, \vec{\mathsf{z}}_\mathsf{i}, \vec{\pi}_\mathsf{i}) \ ,$$

  where $\mathsf{M}_{\mathcal{V},\mathbb{C}}$ is a machine called the *PCD machine* and $t_{\mathcal{V},\mathbb{C}}(n) = n^{e_{\mathcal{V},\mathbb{C}}}$ is a polynomial time bound; both $\mathsf{M}_{\mathcal{V},\mathbb{C}}$ and the exponent $e_{\mathcal{V},\mathbb{C}}$ only depend on (and are efficiently computable from) the SNARK verifier $\mathcal{V}$ and the compliance predicate $\mathbb{C}$. We define $\mathsf{M}_{\mathcal{V},\mathbb{C}}$ and $e_{\mathcal{V},\mathbb{C}}$ below.

- **The PCD verifier.** For any compliance predicate $\mathbb{C}$, on input $(\tau, \mathsf{z_o}, \pi_\mathsf{o})$, the PCD verifier $\mathbb{V}_\mathbb{C} := \mathbb{V}(\mathbb{C}, \cdots)$ checks that

$$\mathcal{V}_{e_{\mathcal{V},\mathbb{C}}}\Big(\tau, \big(\mathsf{M}_{\mathcal{V},\mathbb{C}}, (\mathsf{z_o}, \tau), t_{\mathcal{V},\mathbb{C}}(|\mathsf{z_o}| + |\tau|)\big), \pi_\mathsf{o}\Big) = 1 \ .$$

  (Recall that, in a SNARK for NP, $\mathcal{V}_c$ denotes the fact that the verifier is given as additional input a constant $c > 0$ and is only required to work for the relation $\mathcal{R}_c$; see Definition 4.4.)

Both the PCD prover $\mathbb{P}_\mathbb{C}$ and PCD verifier $\mathbb{V}_\mathbb{C}$ need to be able to efficiently generate the SNARK statement $(\mathsf{M}_{\mathcal{V},\mathbb{C}}, (\mathsf{z_o}, \tau), t_{\mathcal{V},\mathbb{C}}(|\mathsf{z_o}|+|\tau|))$ starting from $(\mathsf{z_o}, \tau)$; in particular, both need to efficiently generate $\mathsf{M}_{\mathcal{V},\mathbb{C}}$ and $t_{\mathcal{V},\mathbb{C}}(|\mathsf{z_o}| + |\tau|)$. We now define both $\mathsf{M}_{\mathcal{V},\mathbb{C}}$ and $t_{\mathcal{V},\mathbb{C}}$, and explain how these can be efficiently constructed.

The PCD machine $\mathsf{M}_{\mathcal{V},\mathbb{C}}$. The PCD machine $\mathsf{M}_{\mathcal{V},\mathbb{C}}$ takes input x and witness w where $x = (\mathsf{z_o}, \tau)$ and $w = (\mathsf{linp}, \vec{\mathsf{z}}_\mathsf{i}, \vec{\pi}_\mathsf{i})$. Then, $\mathsf{M}_{\mathcal{V},\mathbb{C}}$ verifies that: (a) the message $\mathsf{z_o}$ is $\mathbb{C}$-compliant with the local input linp and input messages $\vec{\mathsf{z}}_\mathsf{i}$, and (b) each π in the vector $\vec{\pi}_\mathsf{i}$ is a valid SNARK proof attesting to the $\mathbb{C}$-compliance of the corresponding message z in $\vec{\mathsf{z}}_\mathsf{i}$. The formal description of the machine $\mathsf{M}_{\mathcal{V},\mathbb{C}}$ is given in Figure 3; it is clear from its description that one can efficiently deduce $\mathsf{M}_{\mathcal{V},\mathbb{C}}$ from $\mathcal{V}$, $\mathbb{C}$, and $e_{\mathcal{V},\mathbb{C}}$.

The time bound $t_{\mathcal{V},\mathbb{C}}$. We want $t_{\mathcal{V},\mathbb{C}}(|\mathsf{z_o}|+|\tau|)$ to bound the computation time of $\mathsf{M}_{\mathcal{V},\mathbb{C}}\big((\mathsf{z_o}, \tau), (\mathsf{linp}, \vec{\mathsf{z}}_\mathsf{i}, \vec{\pi}_\mathsf{i})\big)$, for any witness $(\mathsf{linp}, \vec{\mathsf{z}}_\mathsf{i}, \vec{\pi}_\mathsf{i})$. We now explain how to choose the exponent $e_{\mathcal{V},\mathbb{C}}$ of the time bound function $t_{\mathcal{V},\mathbb{C}}(n) = n^{e_{\mathcal{V},\mathbb{C}}}$. Note that:

- The first part of the computation of the PCD machine $\mathsf{M}_{\mathcal{V},\mathbb{C}}$ is verifying $\mathbb{C}$-compliance at the local node, namely, verifying that $\mathbb{C}(\mathsf{z_o}; \mathsf{linp}, \vec{\mathsf{z}}_\mathsf{i}) = 1$; since $\mathbb{C}$ is polynomially balanced (see Remark 5.5), the time to perform this check is $t_\mathbb{C}(|\mathsf{z_o}|)$, where $t_\mathbb{C}$ is a polynomial depending on $\mathbb{C}$.

- The second part of $\mathsf{M}_{\mathcal{V},\mathbb{C}}$'s computation is verifying that the inputs are $\mathbb{C}$-compliant, by relying on the proofs that they carry; the time required to do so depends on the running time of the SNARK verifier $\mathcal{V}$ and how many such inputs there are.

Thus, letting $t_\mathcal{V}$ be the polynomial bounding the running time of the SNARK verifier $\mathcal{V}$, the total computation time of $\mathsf{M}_{\mathcal{V},\mathbb{C}}\big((\mathsf{z_o}, \tau), (\mathsf{linp}, \vec{\mathsf{z}}_\mathsf{i}, \vec{\pi}_\mathsf{i})\big)$ is:

$$t_\mathbb{C}(|\mathsf{z_o}|) + \sum_{\mathsf{z} \in \vec{\mathsf{z}}_\mathsf{i}} t_\mathcal{V}\big(k + |y_\mathsf{z}|\big) \tag{1}$$

$\mathsf{M}_{\mathcal{V},\mathbb{C}}(x, w) \equiv$

1. **Parsing input and witness.** Parse x as $(\mathsf{z_o}, \tau)$ and w as $(\mathsf{linp}, \vec{\mathsf{z}}_\mathsf{i}, \vec{\pi}_\mathsf{i})$. Intuitively, $\vec{\mathsf{z}}_\mathsf{i}$ are the input messages, $\vec{\pi}_\mathsf{i}$ corresponding proofs of $\mathbb{C}$-compliance, linp a local input, $\mathsf{z_o}$ an output message, and τ the SNARK verification state.
2. **Base case.** If $(\mathsf{linp}, \vec{\mathsf{z}}_\mathsf{i}, \vec{\pi}_\mathsf{i}) = \perp$, verify that $\mathbb{C}(\mathsf{z_o}; \perp, \perp) = 1$. (This corresponds to checking that $\mathsf{z_o}$ is a $\mathbb{C}$-compliant source for the distributed computation.)
3. **General case.** Verify:
   - *Compliance of the current node*: $\mathbb{C}(\mathsf{z_o}; \mathsf{linp}, \vec{\mathsf{z}}_\mathsf{i}) = 1$.
   - *Compliance of the past:* For each input z in $\vec{\mathsf{z}}_\mathsf{i}$ and corresponding proof π in $\vec{\pi}_\mathsf{i}$ verify that
   $$\mathcal{V}_{e_{\mathcal{V},\mathbb{C}}}\Big(\tau, \big(\mathsf{M}_{\mathcal{V},\mathbb{C}}, (\mathsf{z}, \tau), t_{\mathcal{V},\mathbb{C}}(|\mathsf{z}| + |\tau|)\big), \pi\Big) = 1 \ .$$
   (We now think of each z as an output of a previous distributed computation that carries a proof π attesting to the $\mathbb{C}$-compliance of z.)

Furthermore, if $\mathsf{M}_{\mathcal{V},\mathbb{C}}$ reaches the time bound $t_{\mathcal{V},\mathbb{C}}(|\mathsf{z_o}| + |\tau|)$, it halts and rejects. The function $t_{\mathcal{V},\mathbb{C}}(\cdot)$ is such that $t_{\mathcal{V},\mathbb{C}}(|\mathsf{z_o}| + |\tau|) = (k + |\mathsf{z_o}|)^{e_{\mathcal{V},\mathbb{C}}}$ where $e_{\mathcal{V},\mathbb{C}}$ is an exponent depending on (and efficiently computable from) $\mathcal{V}$ and $\mathbb{C}$. We explain how to choose $e_{\mathcal{V},\mathbb{C}}$ in the paragraph below.

(Above, the description of $\mathsf{M}_{\mathcal{V},\mathbb{C}}$ appears in its own code. This is only syntactic sugar, and, to give a completely formal definition of $\mathsf{M}_{\mathcal{V},\mathbb{C}}$, one needs to invoke an efficient version of the Recursion Theorem.)

Figure 3: The PCD machine $\mathsf{M}_{\mathcal{V},\mathbb{C}}$ for the publicly-verifiable case.

$$= t_\mathbb{C}(|\mathsf{z_o}|) + \sum_{\mathsf{z} \in \vec{\mathsf{z}}_\mathsf{i}} t_\mathcal{V}\Big(k + |\mathsf{M}_{\mathcal{V},\mathbb{C}}| + |(\mathsf{z}, \tau)| + \log\big(t_{\mathcal{V},\mathbb{C}}(|\mathsf{z}| + |\tau|)\big)\Big) \tag{2}$$

$$= t_\mathbb{C}(|\mathsf{z_o}|) + \sum_{\mathsf{z} \in \vec{\mathsf{z}}_\mathsf{i}} t_\mathcal{V}\Big(k + |\mathbb{C}| + |\mathcal{V}| + |\mathsf{z}| + |\tau| + \log\big(t_{\mathcal{V},\mathbb{C}}(|\mathsf{z}| + |\tau|)\big)\Big) \tag{3}$$

$$= t_\mathbb{C}(|\mathsf{z_o}|) + \sum_{\mathsf{z} \in \vec{\mathsf{z}}_\mathsf{i}} t_\mathcal{V}\Big(k + |\mathbb{C}| + |\mathsf{z}| + |\tau| + \log\big(t_{\mathcal{V},\mathbb{C}}(|\mathsf{z}| + |\tau|)\big)\Big) \tag{4}$$

$$\leq t_\mathbb{C}(|\mathsf{z_o}|) + \sum_{\mathsf{z} \in \vec{\mathsf{z}}_\mathsf{i}} t_\mathcal{V}\big(k + |\mathbb{C}| + |\mathsf{z}| + |\tau| + (\log k)^2\big) \tag{5}$$

$$\leq t_\mathbb{C}(|\mathsf{z_o}|) + t_\mathbb{C}(|\mathsf{z_o}|) \cdot t_\mathcal{V}\big(k + |\mathbb{C}| + t_\mathbb{C}(|\mathsf{z_o}|) + |\tau| + (\log k)^2\big) \ , \tag{6}$$

where (2) follows from (1) by expanding $|y_\mathsf{z}|$; (3) follows from (2) by expanding $|\mathsf{M}_{\mathcal{V},\mathbb{C}}|$ and $|(\mathsf{z}, \tau)|$; (4) follows from (3) by assuming without loss of generality that $|\mathcal{V}| \leq t_\mathcal{V}(k + |y|)$ for all k and y; (5) follows from (4) because all computations are bounded by some super-polynomial function in the security parameter, say $k^{\log k}$, and hence can bound $t_{\mathcal{V},\mathbb{C}}(|\mathsf{z}| + |\tau|)$ by $k^{\log k}$ and thus $\log t_{\mathcal{V},\mathbb{C}}(|\mathsf{z}| + |\tau|) \leq (\log k)^2$ (see Remark 5.7); (6) follows from (5) because $\mathbb{C}$ is polynomially-balanced and thus $|\vec{\mathsf{z}}_\mathsf{i}| \leq t_\mathbb{C}(|\mathsf{z_o}|)$.

Overall, from (6), we conclude that the total computation time of $\mathsf{M}_{\mathcal{V},\mathbb{C}}\big((\mathsf{z_o}, \tau), (\mathsf{linp}, \vec{\mathsf{z}}_\mathsf{i}, \vec{\pi}_\mathsf{i})\big)$ can be bounded by $t_{\mathcal{V},\mathbb{C}}(|\mathsf{z_o}| + |\tau|) = (|\mathsf{z_o}| + |\tau|)^{e_{\mathcal{V},\mathbb{C}}}$ where $e_{\mathcal{V},\mathbb{C}}$ is an exponent that can be efficiently computed from $\mathcal{V}$ (and $t_\mathcal{V}$) and $\mathbb{C}$ (and $t_\mathbb{C}$). (Note that the running time of $\mathcal{V}_{e_{\mathcal{V},\mathbb{C}}}$ is $t_\mathcal{V}$, which is is *independent* of $e_{\mathcal{V},\mathbb{C}}$; thus, there is no issue of circularity here; see Definition 4.4.)

6.1.2 Proof Of Security

We now show that $(\mathbb{G}, \mathbb{P}, \mathbb{V})$ is a (publicly-verifiable) PCD system for constant-depth compliance predicates. The completeness and efficiency properties of the PCD system immediately follow from those of the SNARK $(\mathcal{G}, \mathcal{P}, \mathcal{V})$. We thus concentrate on proving the adaptive proof of knowledge property. Let us fix a compliance predicate $\mathbb{C}$ with constant depth $\mathsf{d}(\mathbb{C})$.

Our goal is the following: for any (possibly malicious) polynomial-size prover $\mathbb{P}^*$, we need to construct a corresponding polynomial-size extractor $\mathbb{E}_{\mathbb{P}^*}$ such that, when $\mathbb{P}^*$ convinces $\mathbb{V}_{\mathbb{C}}$ that a message $\mathsf{z_o}$ is $\mathbb{C}$-compliant, the extractor can find a $\mathbb{C}$-compliant transcript T with output $\mathsf{z_o}$ (which "explains" why $\mathbb{V}_{\mathbb{C}}$ accepted). To achieve this goal, we employ a natural recursive extraction strategy, which we now describe.

Given the prover $\mathbb{P}^*$, we construct $\mathsf{d}(\mathbb{C})$ (families of) polynomial-size extractors $\mathcal{E}_1, \ldots, \mathcal{E}_{\mathsf{d}(\mathbb{C})}$, one for each potential depth of the distributed computation. To make notation lighter, we do not explicitly write the auxiliary input z that may be given to $\mathbb{P}^*$ and its extractor $\mathbb{E}_{\mathbb{P}^*}$ (e.g., any random coins used by $\mathbb{P}^*$); similarly for the SNARK provers and their extractors discussed below. This is not a problem because what we are going to prove holds also with respect to any auxiliary input distribution $\mathcal{Z}$, provided the underlying SNARK is secure with respect to the same auxiliary input distribution $\mathcal{Z}$.

Specifically, the extractors are constructed in the following way:

- Use the PCD prover $\mathbb{P}^*$ to construct the SNARK prover $\mathcal{P}_1^*$ that works as follows: on input σ, $\mathcal{P}_1^*$ computes $(\mathsf{z}_1, \pi_1) \leftarrow \mathbb{P}^*(\sigma)$, constructs the instance $y_1 := (\mathsf{M}_{\mathcal{V},\mathbb{C}}, (\mathsf{z}_1, \tau), t_{\mathcal{V},\mathbb{C}}(|\mathsf{z}_1| + |\tau|))$, and outputs (y_1, π_1). Then define $\mathcal{E}_1 := \mathcal{E}_{\mathcal{P}_1^*}$ to be the SNARK extractor for the SNARK prover $\mathcal{P}_1^*$. Like $\mathcal{P}_1^*$, $\mathcal{E}_1$ also expects input σ; $\mathcal{E}_1$ returns a string $(\vec{\mathsf{z}}_2, \vec{\pi}_2, \mathsf{linp}_1)$ that is (with all but negligible probability) a valid witness for the SNARK statement y_1, assuming that $\mathbb{V}_{\mathbb{C}}$ (and thus also $\mathcal{V}_{e_{\mathcal{V},\mathbb{C}}}$) accepts π_1.

- Use $\mathcal{E}_1$ to construct the new SNARK prover $\mathcal{P}_2^*$ that works as follows: on input σ, $\mathcal{P}_2^*$ computes $(\vec{\mathsf{z}}_2, \vec{\pi}_2, \mathsf{linp}_1) \leftarrow \mathcal{E}_1(\sigma)$ and then outputs $(\vec{y}_2, \vec{\pi}_2)$, where the vector of SNARK statements $\vec{y}_2$ contains an entry $y_{\mathsf{z}} := (\mathsf{M}_{\mathcal{V},\mathbb{C}}, (\mathsf{z}, \tau), t_{\mathcal{V},\mathbb{C}}(|\mathsf{z}| + |\tau|))$ for each entry z in $\vec{\mathsf{z}}_2$. Then define $\mathcal{E}_2 := \mathcal{E}_{\mathcal{P}_2^*}$ to be the SNARK extractor for the SNARK prover $\mathcal{P}_2^*$. Given σ, with all but negligible probability, $\mathcal{E}_2$ outputs a witness for each statement and convincing proof (y, π) in $(\vec{y}_2, \vec{\pi}_2)$. (See Remark 4.6.)

- In general, for each $1 < j \leq \mathsf{d}(\mathbb{C})$, we similarly define $\mathcal{P}_j^*$ and $\mathcal{E}_j := \mathcal{E}_{\mathcal{P}_j^*}$.

We can now define the extractor $\mathbb{E}_{\mathbb{P}^*}$. On input σ, $\mathbb{E}_{\mathbb{P}^*}$ constructs a distributed computation transcript T whose graph is a directed tree, by running $\mathcal{E}_1, \ldots, \mathcal{E}_{\mathsf{d}(\mathbb{C})}$ in order; each such extractor produces a corresponding level in the distributed computation tree. Specifically, each witness $(\vec{\mathsf{z}}, \vec{\pi}, \mathsf{linp})$ extracted by $\mathcal{E}_j$ corresponds to a node v on the j-th level of the tree, with local input $\mathsf{linp}(v) := \mathsf{linp}$ and incoming messages $\mathsf{inputs}(v) := \vec{\mathsf{z}}$. The tree has a single sink s with only one edge (s', s) going into it; the message on that edge is $\mathsf{data}(s, s') := \mathsf{z}_1$. (Recall that z_1 is the message output by $\mathbb{P}^*$.) The leaves of the tree are the vertices for which the extracted witnesses are $(\mathsf{linp}, \vec{\mathsf{z}}, \vec{\pi}) = \perp$.[12]

Note that each $\mathcal{E}_j$ is of polynomial size, because each $\mathcal{E}_j$ is constructed via a constant number of recursive invocations of the SNARK proof of knowledge, and each such invocation incurs an arbitrary polynomial blowup in the size of the extractor relative to its prover. Thus, we deduce that $\mathbb{E}_{\mathbb{P}^*}$ is also of polynomial size.

We are left to argue that the transcript T extracted by $\mathbb{E}_{\mathbb{P}^*}$ is $\mathbb{C}$-compliant and has output z_1:

[12] During extraction we may find the same message twice; if so, we could avoid extracting from this same message twice by simply putting a "pointer" from where we encounter it the second time to the first time we encountered it. We do not perform this "representation optimization" as it is inconsequential in this proof. (Though this optimization is important when carrying out the proof for super-constant $\mathsf{d}(\mathbb{C})$ starting from stronger extractability assumptions; see Remark 6.3.)

Proposition 6.4. *Let $\mathbb{P}^*$ be a polynomial-size PCD prover, and let $\mathbb{E}_{\mathbb{P}^*}$ be its corresponding polynomial-size extractor as defined above. Then:*

$$\Pr\left[\begin{array}{c|c} \mathbb{V}_{\mathbb{C}}(\tau,\mathsf{z}_1,\pi_1)=1 & (\sigma,\tau)\leftarrow\mathbb{G}(1^k) \\ \big(\mathsf{out}(\mathsf{T})\neq\mathsf{z}_1\ \vee\ \mathbb{C}(\mathsf{T})\neq 1\big) & (\mathsf{z}_1,\pi_1)\leftarrow\mathbb{P}^*(\sigma) \\ & \mathsf{T}\leftarrow\mathbb{E}_{\mathbb{P}^*}(\sigma)\end{array}\right]\leq \mathrm{negl}(k)\ .$$

Proof. By construction, $\mathsf{out}(\mathsf{T})=\mathsf{z}_1$ always. We are left to prove that (with all but negligible probability whenever $\mathbb{V}_{\mathbb{C}}$ accepts) it holds that $\mathbb{C}(\mathsf{T})=1$. The proof is by induction on the level of the extracted tree (going from root to leaves). Recall that there are at most $\mathsf{d}(\mathbb{C})=O(1)$ levels all together.

For the base case, we show that for all large enough $k\in\mathbb{N}$, except with negligible probability, whenever the prover $\mathbb{P}^*$ convinces the verifier $\mathbb{V}_{\mathbb{C}}$ to accept (z_1,π_1), the extractor $\mathcal{E}_1$ outputs $(\vec{\mathsf{z}}_2,\vec{\pi}_2,\mathsf{linp}_1)$ such that:

1. $\mathbb{C}(\mathsf{z}_1;\vec{\mathsf{z}}_2,\mathsf{linp}_1)=1$, and
2. for each (z,π) in $(\vec{\mathsf{z}}_2,\vec{\pi}_2)$, letting $y_{\mathsf{z}}:=(\mathsf{M}_{\mathcal{V},\mathbb{C}},(\mathsf{z},\tau),t_{\mathcal{V},\mathbb{C}}(|\mathsf{z}|+|\tau|))$, it holds that $\mathcal{V}_{e_{\mathcal{V},\mathbb{C}}}(\tau,y_{\mathsf{z}},\pi)=1$.

Indeed, $\mathbb{V}_{\mathbb{C}}(\tau,\mathsf{z}_1,\pi_1)=1$ implies $\mathcal{V}_{e_{\mathcal{V},\mathbb{C}}}(\tau,y_{\mathsf{z}_1},\pi_1)=1$, where $y_{\mathsf{z}_1}=(\mathsf{M}_{\mathcal{V},\mathbb{C}},(\mathsf{z}_1,\tau),t_{\mathcal{V},\mathbb{C}}(|\mathsf{z}_1|+|\tau|))$. By the SNARK proof of knowledge property, whenever $\mathbb{V}_{\mathbb{C}}$ accepts, with all but negligible probability, the extractor $\mathcal{E}_1$ outputs a valid witness $(\vec{\mathsf{z}}_2,\vec{\pi}_2,\mathsf{linp}_2)$ for the statement y_{z_1}. By construction of the PCD machine $\mathsf{M}_{\mathcal{V},\mathbb{C}}$, the extracted witness $(\vec{\mathsf{z}}_2,\vec{\pi}_2,\mathsf{linp}_2)$ satisfies both of the claimed properties.

For the inductive step, we can prove in a similar manner the compliance of a level in the extracted distributed computation tree, assuming compliance of the previous level. Specifically, assume that for each node v in level $1\leq j<\mathsf{d}(\mathbb{C})$ the following holds: for each (z,π) in $(\vec{\mathsf{z}}_v,\vec{\pi}_v)$, it holds that $\mathcal{V}_{e_{\mathcal{V},\mathbb{C}}}(\tau,y_{\mathsf{z}},\pi)=1$, where $(\vec{\mathsf{z}}_v,\vec{\pi}_v)$ are v's incoming messages and proofs (extracted by $\mathcal{E}_j$) and $y_{\mathsf{z}}:=(\mathsf{M}_{\mathcal{V},\mathbb{C}},(\mathsf{z},\tau),t_{\mathcal{V},\mathbb{C}}(|\mathsf{z}|+|\tau|))$. Then, with all but negligible probability, for any node u with $(u,v)\in E$, the extractor $\mathcal{E}_{j+1}$ outputs a valid witness $(\vec{\mathsf{z}}_u,\vec{\pi}_u,\mathsf{linp}_u)$ for the statement y_{z_u}, where z_u is the message in $\vec{\mathsf{z}}_v$ corresponding to the edge (u,v). We conclude that:

1. $\mathbb{C}(\mathsf{z}_u;\vec{\mathsf{z}}_u,\mathsf{linp}_u)=1$, and
2. for each (z,π) in $(\vec{\mathsf{z}}_u,\vec{\pi}_u)$, letting $y_{\mathsf{z}}:=(\mathsf{M}_{\mathcal{V},\mathbb{C}},(\mathsf{z},\tau),t_{\mathcal{V},\mathbb{C}}(|\mathsf{z}|+|\tau|))$, it holds that $\mathcal{V}_{e_{\mathcal{V},\mathbb{C}}}(\tau,y_{\mathsf{z}},\pi)=1$.

This completes the inductive step, and we can indeed conclude that T is compliant with $\mathbb{C}$. □

6.1.3 The Preprocessing Case

We now describe how to modify the aforementioned discussion for the case where $(\mathcal{G},\mathcal{P},\mathcal{V})$ is a preprocessing SNARK. In such a case, the generator $\mathcal{G}$ takes as additional input a time bound $B=B(k)$, and generates a reference string σ and corresponding verification state τ that only allow proving and verifying statements y of the form (M,x,t) where $|M|+|x|+t<B$; statements that do not meet this criteria are automatically rejected by the verifier. (Note that this differs from a SNARK for a relation $\mathcal{R}_c$, which allows t to grow as fast as, but not faster than, $|x|^c$.) Of course, the running time of the SNARK verifier $\mathcal{V}$ is still required to be $\mathrm{poly}(k+|y|)$ and is, in particular, independent of B. (The fact that the running time of $\mathcal{V}$, and not just the length of an honestly-generated proof, is short is crucial in our context.)

When using a preprocessing SNARK in the construction from Section 6.1.1, we obtain a *preprocessing PCD system*. (See Item 3 of Definition 5.9.) That is, the construction yields PCD systems where the generator $\mathbb{G}$ takes as additional input a time bound B, and the PCD system works only for B-bounded distributed computations (see Definition 5.6): namely, distributed computations where computing $\mathbb{C}$ at any

node takes time at most B. (We stress once more that the bound B is for a *single* node's computation time, and *not* for the sum of all such times!)

More precisely, the construction of the generator $\mathbb{G}$ in Section 6.1.1 has to be slightly modified: the generator $\mathbb{G}$, on input $(1^k, B)$, invokes $\mathcal{G}(1^k, B')$ where $B' := \mathrm{poly}_{\mathcal{V},\mathcal{H}}(k + B)$ for some $\mathrm{poly}_{\mathcal{V},\mathcal{H}}$ that only depends on $\mathcal{V}$ and the collision-resistant hash-function family $\mathcal{H}$. Essentially, we need to ensure that, whenever checking compliance of a message z takes time at most B, it holds that $t_{\mathcal{V},\mathbb{C}}(|\mathsf{z}| + |\tau|) \leq B'$. We now explain why the above choice of B' suffices.

Whenever computing $\mathbb{C}$ on a message z takes time at most B (as is the case in a B-bounded distributed computation), one can verify that $t_{\mathcal{V},\mathbb{C}}(|\mathsf{z}|+|\tau|) \leq \mathrm{poly}_{\mathcal{V}}(k+B+|\mathbb{C}|)$. Furthermore, we can assume without loss of generality that $|\mathbb{C}| = \mathrm{poly}_{\mathcal{H}}(k)$. Indeed, if that is not the case, we can consider $\mathbb{C}'$ that has hardcoded a hash ρ of $\mathbb{C}$, always expects a local input $\mathsf{linp}' = (\mathbb{C}, \mathsf{linp})$, and first checks that $\rho = h(\mathbb{C})$ and then checks that the output is $\mathbb{C}$-compliant relative to linp and the input messages. Thus, overall, $\mathrm{poly}_{\mathbb{C}}(|\mathsf{z}|) \leq B$ implies that $t_{\mathcal{V},\mathbb{C}}(|\mathsf{z}|) \leq \mathrm{poly}_{\mathcal{V},\mathcal{H}}(k + B)$, and thus our choice of B' suffices.

The aforementioned modification to the construction of $\mathbb{G}$ is the only one needed to make the construction from Section 6.1.1 work in the preprocessing setting.

We conclude by remarking that, when choosing $B = \mathrm{poly}(k)$, running $\mathbb{G}$ requires only $\mathrm{poly}(k)$ time; in other words, preprocessing becomes "inexpensive". One of the results of this paper is that, ultimately, we can always get rid of expensive preprocessing, and being able to choose $B = \mathrm{poly}(k)$ (and make do with enforcing $\mathbb{C}$-compliance in only $\mathrm{poly}(k)$-bounded distributed computations) is an important step when proving this fact. See Section 2.5 and Section 9 for more details.

6.2 Recursive Composition For Designated-Verifier SNARKs

In Section 6.1 we have proved the publicly-verifiable case of the SNARK Recursive Composition Theorem (i.e., Item 1 of Theorem 6.1). We now prove the designated-verifier case of the SNARK Recursive Composition Theorem (i.e., Item 2 of Theorem 6.1). In other words, we show that we can *also* recursively compose designated-verifier SNARKs to obtain designated-verifier PCD systems for constant-depth compliance predicates.[13]

As before, we first give the construction and proof, respectively in Section 6.2.1 and in Section 6.2.2, for the (designated-verifier) case with no preprocessing. Extending the discussion to the preprocessing case is completely analogous to the extension in the publicly-verifiable case, explained in Section 6.1.3.[14]

6.2.1 The Construction

We are given a designated-verifier (fully-succinct) SNARK $(\mathcal{G}, \mathcal{P}, \mathcal{V})$ for NP (see Definition 4.4). We need to construct a designated-verifier (fully-succinct) PCD system $(\mathbb{G}, \mathbb{P}, \mathbb{V})$ for constant-depth compliance predicates.

When we try to adapt the PCD construction for the publicly-verifiable case (see Section 6.1.1) to the designated-verifier case, we encounter the following difficulty: how does an arbitrary node in the computation prove that it obtained a convincing proof of compliance for its own input, when it cannot even verify the proof on its own? More concretely: the node does not know the verification state τ (because it is secret) and, therefore, cannot provide a witness for such a theorem.

[13] We recall that "designated-verifier" means (just like in the SNARK case) that verifying a proof requires a secret verification state, and *not* that the messages in the distributed computation are encrypted; see Section 5.

[14] In particular, this extension also relies on collision-resistant hashing; however, this assumption does not have to be explicitly required in the theorem statement, because it is implied by homomorphic encryption [IKO05].

We show how to circumvent this difficulty, using fully-homomorphic encryption (FHE). The idea goes as follows. We encrypt the private verification state τ and attach its encryption c^τ to the public reference string σ. Then, when a node is required to verify the proof it obtained, it homomorphically evaluates the SNARK verifier $\mathcal{V}$ on the encrypted verification state c^τ and the statement and proof at hand. In order to achieve compliance of the past, each node provides, as part of his proof, the result of the homomorphic evaluation $\hat{\mathsf{c}}$, and a proof that it "knows" a previous proof, such that $\hat{\mathsf{c}}$ is indeed the result of evaluating $\mathcal{V}$ on c^τ on this proof (and some corresponding statement). At each point the PCD verifier $\mathbb{V}_{\mathbb{C}}$, can apply the SNARK verifier $\mathcal{V}$ to check that: (a) the SNARK proof is valid, (b) the decryption of $\hat{\mathsf{c}}$ is indeed "1". (More precisely, we need to do an extra step in order to avoid the size of the proofs from blowing up due to appending $\hat{\mathsf{c}}$ at each node.)

We now convert the above intuitive explanation into a precise discussion. The construction of the PCD system $(\mathbb{G}, \mathbb{P}, \mathbb{V})$ is as follows:

- **The PCD generator.** On input the security parameter 1^k, the PCD generator $\mathbb{G}$ runs the SNARK generator $\mathcal{G}(1^k)$ in order to obtain a reference string σ and verification state τ, samples a secret key sk and an evaluation key ek for the FHE scheme $\mathcal{E}$, computes an encryption c^τ of the secret verification state, and then outputs $(\tilde{\sigma}, \tilde{\tau}) := \big((\sigma, \mathsf{c}^\tau), (\tau, \mathsf{sk})\big)$. Without loss of generality, we assume that both σ and τ include the security parameter 1^k in the clear; furthermore, we may also assume that c^τ includes the evaluation key ek. Recall that we are temporarily focusing on the case where the designated-verifier SNARK $(\mathcal{G}, \mathcal{P}, \mathcal{V})$ is fully-succinct, so that σ and τ have size that is a fixed polynomial in the security parameter.

- **The PCD prover and the PCD machine.** For any compliance predicate $\mathbb{C}$, given input $(\tilde{\sigma}, \mathsf{z}_\mathsf{o}, \mathsf{linp}, \vec{\mathsf{z}}_\mathsf{i}, \vec{\pi}'_\mathsf{i})$, the PCD prover $\mathbb{P}_{\mathbb{C}} = \mathbb{P}(\mathbb{C}, \cdots)$ works as follows:

  1. Parse each π'_i in $\vec{\pi}'_\mathsf{i}$ as a pair $(\pi_\mathsf{i}, \hat{\mathsf{c}}_\mathsf{i})$; construct corresponding vectors $\vec{\pi}_\mathsf{i}$ and $\vec{\hat{\mathsf{c}}}_\mathsf{i}$.
  2. First, $\mathbb{P}_{\mathbb{C}}$ computes a new evaluated verification bit $\hat{\mathsf{c}}_\mathsf{o}$ that "aggregates" the evaluations $\vec{\hat{\mathsf{c}}}_\mathcal{V}$ of the SNARK verifier together with the previous verification bits $\vec{\hat{\mathsf{c}}}_\mathsf{i}$. Concretely, $\mathbb{P}_{\mathbb{C}}$ computes:

  $$\hat{\mathsf{c}}_\mathsf{o} := \mathsf{Eval}_{\mathsf{ek}}\left(\prod, (\vec{\hat{\mathsf{c}}}_\mathsf{i}, \vec{\hat{\mathsf{c}}}_\mathcal{V})\right) \ ,$$

  where each ciphertext $\hat{\mathsf{c}}_\mathcal{V}$ in $\vec{\hat{\mathsf{c}}}_\mathcal{V}$ corresponds to a triple $(\mathsf{z}, \pi, \hat{\mathsf{c}})$ in $(\vec{\mathsf{z}}_\mathsf{i}, \vec{\pi}_\mathsf{i}, \vec{\hat{\mathsf{c}}}_\mathsf{i})$ and is the result of homomorphically evaluating the SNARK verifier as follows:

  $$\hat{\mathsf{c}}_\mathcal{V} = \mathsf{Eval}_{\mathsf{ek}}\left(\mathcal{V}_{e_{\mathcal{V},\mathbb{C}}}\left(\cdot, (\mathsf{M}_{\mathcal{V},\mathbb{C}}, (\mathsf{z}, \hat{\mathsf{c}}, \mathsf{c}^\tau), t_{\mathcal{V},\mathbb{C}}(|\mathsf{z}| + |\hat{\mathsf{c}}| + |\mathsf{c}^\tau|)), \pi\right), \mathsf{c}^\tau\right) \ ,$$

  where $\mathsf{M}_{\mathcal{V},\mathbb{C}}$ is a machine called the *PCD machine* and $t_{\mathcal{V},\mathbb{C}}(n) = n^{e_{\mathcal{V},\mathbb{C}}}$ is a polynomial time bound; both $\mathsf{M}_{\mathcal{V},\mathbb{C}}$ and the exponent $e_{\mathcal{V},\mathbb{C}}$ only depend on (and are efficiently computable from) the SNARK verifier $\mathcal{V}$ and the compliance predicate $\mathbb{C}$. We define $\mathsf{M}_{\mathcal{V},\mathbb{C}}$ and $e_{\mathcal{V},\mathbb{C}}$ below.
  3. Having computed $\hat{\mathsf{c}}_\mathsf{o}$, $\mathbb{P}_{\mathbb{C}}$ constructs a "SNARK theorem and witness" $(y, w) = ((M, x, t), w)$ and then runs the SNARK prover $\mathcal{P}(\sigma, y, w)$ to produce the proof π_o, in order to then attach the proof $\pi'_\mathsf{o} := (\pi_\mathsf{o}, \hat{\mathsf{c}}_\mathsf{o})$ to the outgoing message z_o. More precisely, $\mathbb{P}_{\mathbb{C}}$ chooses y and w as follows:

  $$y := \big(\mathsf{M}_{\mathcal{V},\mathbb{C}}, (\mathsf{z}_\mathsf{o}, \hat{\mathsf{c}}_\mathsf{o}, \mathsf{c}^\tau), t_{\mathcal{V},\mathbb{C}}(|\mathsf{z}_\mathsf{o}| + |\hat{\mathsf{c}}_\mathsf{o}| + |\mathsf{c}^\tau|)\big) \ \text{ and } \ w := (\mathsf{linp}, \vec{\mathsf{z}}_\mathsf{i}, \vec{\pi}_\mathsf{i}, \vec{\hat{\mathsf{c}}}_\mathsf{i}) \ .$$

- **The PCD verifier.** For any compliance predicate $\mathbb{C}$, on input $(\tilde{\tau}, \mathsf{z}_\mathsf{o}, \pi'_\mathsf{o})$, the PCD verifier $\mathbb{V}_{\mathbb{C}} := \mathbb{V}(\mathbb{C}, \cdots)$ checks that $\mathsf{Dec}_{\mathsf{sk}}(\hat{\mathsf{c}}_\mathsf{o}) = 1$ and

$$\mathcal{V}_{e_{\mathcal{V},\mathbb{C}}}\Big(\tau, (\mathsf{M}_{\mathcal{V},\mathbb{C}}, (\mathsf{z}_\mathsf{o}, \hat{\mathsf{c}}_\mathsf{o}, \mathsf{c}^\tau), t_{\mathcal{V},\mathbb{C}}(|\mathsf{z}_\mathsf{o}| + |\hat{\mathsf{c}}_\mathsf{o}| + |\mathsf{c}^\tau|)), \pi_\mathsf{o}\Big) = 1 \ .$$

(Recall that, in a SNARK for NP, $\mathcal{V}_c$ denotes the fact that the verifier is given as additional input a constant $c > 0$ and is only required to work for the relation $\mathcal{R}_c$; see Definition 4.4.)

Similarly to the publicly-verifiable case, both the PCD prover $\mathbb{P}_{\mathbb{C}}$ and PCD verifier $\mathbb{V}_{\mathbb{C}}$ need to be able to efficiently generate the SNARK statement $(\mathsf{M}_{\mathcal{V},\mathbb{C}}, (\mathsf{z_o}, \hat{\mathsf{c}}_\mathsf{o}, \mathsf{c}^\tau), t_{\mathcal{V},\mathbb{C}}(|\mathsf{z_o}| + |\hat{\mathsf{c}}_\mathsf{o}| + |\mathsf{c}^\tau|))$ starting from $(\mathsf{z_o}, \hat{\mathsf{c}}_\mathsf{o}, \mathsf{c}^\tau)$; in particular, both need to efficiently generate $\mathsf{M}_{\mathcal{V},\mathbb{C}}$ and $t_{\mathcal{V},\mathbb{C}}(|\mathsf{z_o}| + |\hat{\mathsf{c}}_\mathsf{o}| + |\mathsf{c}^\tau|)$. We now define both $\mathsf{M}_{\mathcal{V},\mathbb{C}}$ and $t_{\mathcal{V},\mathbb{C}}$, and explain how these can be efficiently constructed.

The PCD machine $\mathsf{M}_{\mathcal{V},\mathbb{C}}$. Similarly to the publicly-verifiable case, the heart of the construction is the design of the PCD machine $\mathsf{M}_{\mathcal{V},\mathbb{C}}$. The formal description of the machine $\mathsf{M}_{\mathcal{V},\mathbb{C}}$ is given in Figure 4; it is clear from its description that one can efficiently deduce $\mathsf{M}_{\mathcal{V},\mathbb{C}}$ from $\mathcal{V}$, $\mathbb{C}$, and $e_{\mathcal{V},\mathbb{C}}$.

$\mathsf{M}_{\mathcal{V},\mathbb{C}}(x, w) \equiv$

1. **Parsing input and witness.** Parse x as $(\mathsf{z_o}, \hat{\mathsf{c}}_\mathsf{o}, \mathsf{c}^\tau)$ and w as $(\mathsf{linp}, \vec{\mathsf{z}}_\mathsf{i}, \vec{\pi}_\mathsf{i}, \vec{\hat{\mathsf{c}}}_\mathsf{i})$. Intuitively, $\vec{\mathsf{z}}_\mathsf{i}$ are the input messages, $\vec{\pi}_\mathsf{i}$ corresponding (encrypted) proofs of $\mathbb{C}$-compliance and $\vec{\hat{\mathsf{c}}}_\mathsf{i}$ corresponding evaluated verification bits, linp a local input, $\mathsf{z_o}$ an output message, $\hat{\mathsf{c}}_\mathsf{o}$ an output evaluated verification bit, and c^τ the encrypted SNARK verification state.
2. **Base case.** If $(\mathsf{linp}, \vec{\mathsf{z}}_\mathsf{i}, \vec{\pi}_\mathsf{i}, \vec{\hat{\mathsf{c}}}_\mathsf{i}) = \bot$, verify that $\mathbb{C}(\mathsf{z_o}; \bot, \bot) = 1$. (This corresponds to checking that $\mathsf{z_o}$ is a $\mathbb{C}$-compliant source for the distributed computation.)
3. **General case:** Verify:
   - *Compliance of the current node:* $\mathbb{C}(\mathsf{z_o}; \mathsf{linp}, \vec{\mathsf{z}}_\mathsf{i}) = 1$.
   - *Compliance of the past:* Verify that the ciphertext $\hat{\mathsf{c}}_\mathsf{o}$ correctly aggregates the $\mathbb{C}$-compliance of the input messages: namely, verify that
   $$\hat{\mathsf{c}}_\mathsf{o} = \mathsf{Eval}_{\mathsf{ek}}\left(\prod, (\vec{\hat{\mathsf{c}}}_\mathsf{i}, \vec{\hat{\mathsf{c}}}_\mathcal{V})\right) \ ,$$
   where each ciphertext $\hat{\mathsf{c}}_\mathcal{V}$ in $\vec{\hat{\mathsf{c}}}_\mathcal{V}$ corresponds to a triple $(\mathsf{z}, \pi, \hat{\mathsf{c}})$ in $(\vec{\mathsf{z}}_\mathsf{i}, \vec{\pi}_\mathsf{i}, \vec{\hat{\mathsf{c}}}_\mathsf{i})$ and is the result of homomorphically evaluating the SNARK verifier as follows:
   $$\hat{\mathsf{c}}_\mathcal{V} = \mathsf{Eval}_{\mathsf{ek}}\left(\mathcal{V}_{e_{\mathcal{V},\mathbb{C}}}\left(\cdot, (\mathsf{M}_{\mathcal{V},\mathbb{C}}, (\mathsf{z}, \hat{\mathsf{c}}, \mathsf{c}^\tau), t_{\mathcal{V},\mathbb{C}}(|\mathsf{z}| + |\hat{\mathsf{c}}| + |\mathsf{c}^\tau|)), \pi\right), \mathsf{c}^\tau\right) \ .$$

Furthermore, if $\mathsf{M}_{\mathcal{V},\mathbb{C}}$ reaches the time bound $t_{\mathcal{V},\mathbb{C}}(|\mathsf{z_o}| + |\hat{\mathsf{c}}_\mathsf{o}| + |\mathsf{c}^\tau|)$, it halts and rejects. The function $t_{\mathcal{V},\mathbb{C}}(\cdot)$ is such that $t_{\mathcal{V},\mathbb{C}}(|\mathsf{z_o}| + |\hat{\mathsf{c}}_\mathsf{o}| + |\mathsf{c}^\tau|) = (|\mathsf{z_o}| + |\hat{\mathsf{c}}_\mathsf{o}| + |\mathsf{c}^\tau|)^{e_{\mathcal{V},\mathbb{C}}}$ where $e_{\mathcal{V},\mathbb{C}}$ is an exponent depending on (and efficiently computable from) $\mathcal{V}$ and $\mathbb{C}$. We explain how to choose $e_{\mathcal{V},\mathbb{C}}$ in the paragraph below.

(Above, the description of $\mathsf{M}_{\mathcal{V},\mathbb{C}}$ appears in its own code. This is only syntactic sugar, and, to give a completely formal definition of $\mathsf{M}_{\mathcal{V},\mathbb{C}}$, one needs to invoke an efficient version of the Recursion Theorem.)

Figure 4: The PCD machine $\mathsf{M}_{\mathcal{V},\mathbb{C}}$ for the designated-verifier case.

The time bound $t_{\mathcal{V},\mathbb{C}}$. Similarly to the publicly-verifiable case, we want $t_{\mathcal{V},\mathbb{C}}(|\mathsf{z_o}| + |\hat{\mathsf{c}}_\mathsf{o}| + |\mathsf{c}^\tau|)$ to bound the computation time of $\mathsf{M}_{\mathcal{V},\mathbb{C}}\big((\mathsf{z_o}, \hat{\mathsf{c}}_\mathsf{o}, \mathsf{c}^\tau), (\mathsf{linp}, \vec{\mathsf{z}}_\mathsf{i}, \vec{\pi}_\mathsf{i}, \vec{\hat{\mathsf{c}}}_\mathsf{i})\big)$, for any witness $(\mathsf{linp}, \vec{\mathsf{z}}_\mathsf{i}, \vec{\pi}_\mathsf{i}, \vec{\hat{\mathsf{c}}}_\mathsf{i})$. We now explain how to choose the exponent $e_{\mathcal{V},\mathbb{C}}$ of the time bound function $t_{\mathcal{V},\mathbb{C}}(n) = n^{e_{\mathcal{V},\mathbb{C}}}$. Note that:

- The first part of the computation of the PCD machine $\mathsf{M}_{\mathcal{V},\mathbb{C}}$ is (as before) verifying $\mathbb{C}$-compliance at the local node, namely, verifying that $\mathbb{C}(\mathsf{z_o}; \mathsf{linp}, \vec{\mathsf{z}}_\mathsf{i}) = 1$; since $\mathbb{C}$ is polynomially balanced (see Remark 5.5), the time to perform this check is $t_{\mathbb{C}}(|\mathsf{z_o}|)$, where $t_{\mathbb{C}}$ is a polynomial depending on $\mathbb{C}$.

- The second part of $\mathsf{M}_{\mathcal{V},\mathbb{C}}$'s computation is homomorphically evaluating the SNARK verifier for each input message and homomorphically aggregating the various encrypted bits; the time required to do so depends on the running time of the SNARK verifier $\mathcal{V}$ and how many such inputs there are.

Thus, letting $t_{\mathcal{V}}$ be the polynomial bounding the running time of the SNARK verifier $\mathcal{V}$, and letting $t_{\mathsf{Eval}_{\mathsf{ek}}}$ be a polynomial such that $t_{\mathsf{Eval}_{\mathsf{ek}}}(k+T)$ bounds the time needed by $\mathsf{Eval}_{\mathsf{ek}}$ to homomorphically evaluate a T-time algorithm, the total computation time of $\mathsf{M}_{\mathcal{V},\mathbb{C}}\big((\mathsf{z}_{\mathsf{o}},\hat{\mathsf{c}}_{\mathsf{o}},\mathsf{c}^{\tau}),(\mathsf{linp},\vec{\mathsf{z}}_{\mathsf{i}},\vec{\pi}_{\mathsf{i}},\vec{\hat{\mathsf{c}}}_{\mathsf{i}})\big)$ is:

$$t_{\mathbb{C}}(|\mathsf{z}_{\mathsf{o}}|)+\sum_{\mathsf{z}\in\vec{\mathsf{z}}_{\mathsf{i}}} t_{\mathsf{Eval}_{\mathsf{ek}}}\Big(k+t_{\mathcal{V}}\big(k+|y_{(\mathsf{z},\hat{\mathsf{c}})}|\big)\Big)+\sum_{\mathsf{z}\in\vec{\mathsf{z}}_{\mathsf{i}}} t_{\mathsf{Eval}_{\mathsf{ek}}}\Big(k+2|\vec{\mathsf{z}}_{\mathsf{i}}|\Big) \tag{7}$$

$$\leq t_{\mathbb{C}}(|\mathsf{z}_{\mathsf{o}}|)+2\cdot\sum_{\mathsf{z}\in\vec{\mathsf{z}}_{\mathsf{i}}} t_{\mathsf{Eval}_{\mathsf{ek}}}\Big(k+t_{\mathcal{V}}\Big(k+|\mathsf{M}_{\mathcal{V},\mathbb{C}}|+|(\mathsf{z},\hat{\mathsf{c}},\mathsf{c}^{\tau})|+\log\big(t_{\mathcal{V},\mathbb{C}}(|\mathsf{z}|+|\hat{\mathsf{c}}|+|\mathsf{c}^{\tau}|)\big)\Big)\Big) \tag{8}$$

$$= t_{\mathbb{C}}(|\mathsf{z}_{\mathsf{o}}|)+2\cdot\sum_{\mathsf{z}\in\vec{\mathsf{z}}_{\mathsf{i}}} t_{\mathsf{Eval}_{\mathsf{ek}}}\Big(t_{\mathcal{V}}\Big(k+|\mathbb{C}|+|\mathcal{V}|+|\mathsf{z}|+|\hat{\mathsf{c}}|+|\mathsf{c}^{\tau}|+\log\big(t_{\mathcal{V},\mathbb{C}}(|\mathsf{z}|+|\hat{\mathsf{c}}|+|\mathsf{c}^{\tau}|)\big)\Big)\Big) \tag{9}$$

$$= t_{\mathbb{C}}(|\mathsf{z}_{\mathsf{o}}|)+2\cdot\sum_{\mathsf{z}\in\vec{\mathsf{z}}_{\mathsf{i}}} t_{\mathsf{Eval}_{\mathsf{ek}}}\Big(t_{\mathcal{V}}\Big(k+|\mathbb{C}|+|\mathsf{z}|+|\hat{\mathsf{c}}|+|\mathsf{c}^{\tau}|+\log\big(t_{\mathcal{V},\mathbb{C}}(|\mathsf{z}|+|\hat{\mathsf{c}}|+|\mathsf{c}^{\tau}|)\big)\Big)\Big) \tag{10}$$

$$\leq t_{\mathbb{C}}(|\mathsf{z}_{\mathsf{o}}|)+2\cdot\sum_{\mathsf{z}\in\vec{\mathsf{z}}_{\mathsf{i}}} t_{\mathsf{Eval}_{\mathsf{ek}}}\Big(t_{\mathcal{V}}\big(k+|\mathbb{C}|+|\mathsf{z}|+|\hat{\mathsf{c}}|+|\mathsf{c}^{\tau}|+(\log k)^2\big)\Big) \tag{11}$$

$$\leq t_{\mathbb{C}}(|\mathsf{z}_{\mathsf{o}}|)+2\cdot t_{\mathbb{C}}(|\mathsf{z}_{\mathsf{o}}|)\cdot t_{\mathsf{Eval}_{\mathsf{ek}}}\Big(t_{\mathcal{V}}\big(k+|\mathbb{C}|+t_{\mathbb{C}}(|\mathsf{z}_{\mathsf{o}}|)+|\hat{\mathsf{c}}|+|\mathsf{c}^{\tau}|+(\log k)^2\big)\Big)\ , \tag{12}$$

(8) follows from (7) by expanding $|y_{(\mathsf{z},\hat{\mathsf{c}})}|$; (9) follows from (8) by expanding $|\mathsf{M}_{\mathcal{V},\mathbb{C}}|$ and $|(\mathsf{z},\hat{\mathsf{c}},\mathsf{c}^{\tau})|$; (10) follows from (9) by assuming without loss of generality that $|\mathcal{V}|\leq t_{\mathcal{V}}(k+|y|)$ for all k and y; (11) follows from (10) because all computations are bounded by some super-polynomial function in the security parameter, say $k^{\log k}$, and hence can bound $t_{\mathcal{V},\mathbb{C}}(|\mathsf{z}|+|\hat{\mathsf{c}}|+|\mathsf{c}^{\tau}|)$ by $k^{\log k}$ and thus $\log t_{\mathcal{V},\mathbb{C}}(|\mathsf{z}|+|\hat{\mathsf{c}}|+|\mathsf{c}^{\tau}|)\leq(\log k)^2$ (see Remark 5.7); (12) follows from (11) because $\mathbb{C}$ is polynomially-balanced and thus $|\vec{\mathsf{z}}_{\mathsf{i}}|\leq t_{\mathbb{C}}(|\mathsf{z}_{\mathsf{o}}|)$.

Overall, from (12), we conclude that the total computation time of $\mathsf{M}_{\mathcal{V},\mathbb{C}}\big((\mathsf{z}_{\mathsf{o}},\hat{\mathsf{c}}_{\mathsf{o}},\mathsf{c}^{\tau}),(\mathsf{linp},\vec{\mathsf{z}}_{\mathsf{i}},\vec{\pi}_{\mathsf{i}},\vec{\hat{\mathsf{c}}}_{\mathsf{i}})\big)$ can be bounded by $t_{\mathcal{V},\mathbb{C}}(|\mathsf{z}_{\mathsf{o}}|+|\hat{\mathsf{c}}_{\mathsf{o}}|+|\mathsf{c}^{\tau}|)=(|\mathsf{z}_{\mathsf{o}}|+|\hat{\mathsf{c}}_{\mathsf{o}}|+|\mathsf{c}^{\tau}|)^{e_{\mathcal{V},\mathbb{C}}}$ where $e_{\mathcal{V},\mathbb{C}}$ is an exponent that can be efficiently computed from $t_{\mathsf{Eval}_{\mathsf{ek}}}$, $\mathcal{V}$ (and $t_{\mathcal{V}}$), and $\mathbb{C}$ (and $t_{\mathbb{C}}$). (Note that the running time of $\mathcal{V}_{e_{\mathcal{V},\mathbb{C}}}$ is $t_{\mathcal{V}}$, which is is *independent* of $e_{\mathcal{V},\mathbb{C}}$; thus, there is no issue of circularity here; see Definition 4.4.)

6.2.2 Proof Of Security

We now show that $(\mathbb{G},\mathbb{P},\mathbb{V})$ is a (designated-verifier) PCD system for constant-depth compliance predicates. The completeness and efficiency properties of the PCD system immediately follow from those of the SNARK. We thus concentrate on proving the adaptive proof of knowledge property. Let us fix a compliance predicate $\mathbb{C}$ with constant depth $\mathsf{d}(\mathbb{C})$.

Our goal is (again) the following: for any (possibly malicious) polynomial-size prover $\mathbb{P}^*$, we need to construct a corresponding polynomial-size extractor $\mathbb{E}_{\mathbb{P}^*}$ such that, when $\mathbb{P}^*$ convinces $\mathbb{V}_{\mathbb{C}}$ that a message z_{o} is $\mathbb{C}$-compliant, the extractor can find a $\mathbb{C}$-compliant transcript T with output z_{o} (which "explains" why $\mathbb{V}_{\mathbb{C}}$ accepted). To achieve this goal, we employ a recursive extraction strategy similar to the one we used in the publicly-verifiable case (see Section 6.1.2), which we now describe.

Given the prover $\mathbb{P}^*$, we construct $\mathsf{d}(\mathbb{C})$ (families of) polynomial-size extractors $\mathcal{E}_1,\ldots,\mathcal{E}_{\mathsf{d}(\mathbb{C})}$, one for each potential depth of the distributed computation. As before, to make notation lighter, we do not explicitly write the auxiliary input z that may be given to $\mathbb{P}^*$ and its extractor $\mathbb{E}_{\mathbb{P}^*}$ (e.g., any random coins used

by $\mathbb{P}^*$). Unlike before, however, when we run SNARK extractors, we will need to explicitly specifcy the auxiliary input they get (in this case, an encryption of the verification state; see Remark 6.5). All mentioned implications hold also with respect any auxiliary input distribution $\mathcal{Z}$, provided the underlying SNARK is secure with respect to the auxiliary input distribution $\mathcal{Z}$.

Overall, the PCD extractor $\mathbb{E}_{\mathbb{P}^*}$ is defined analogously to the case of publicly-verifiable SNARKs, except that now statements refer to the new PCD machine as well as to ciphertexts $\hat{\mathsf{c}}$ of the aggregated verification bits, and the encrypted verification state c^τ.

- Use the PCD prover $\mathbb{P}^*$ to construct the SNARK prover $\mathcal{P}_1^*$ that works as follows: on input $(\sigma, \mathsf{c}^\tau)$, $\mathcal{P}_1^*$ computes $(\mathsf{z}_1, \pi_1, \hat{\mathsf{c}}_1) \leftarrow \mathbb{P}^*(\sigma, \mathsf{c}^\tau)$, constructs the instance $y_1 := (\mathsf{M}_{\mathcal{V},\mathbb{C}}, (\mathsf{z}_1, \hat{\mathsf{c}}_1, \mathsf{c}^\tau), t_{\mathcal{V},\mathbb{C}}(|\mathsf{z}_1| + |\hat{\mathsf{c}}_1| + |\mathsf{c}^\tau|))$ and outputs (y_1, π_1). (We think of c^τ as an auxiliary input to $\mathcal{P}_1^*$.) Then define $\mathcal{E}_1 := \mathcal{E}_{\mathcal{P}_1^*}$ to be the SNARK extractor for the SNARK prover $\mathcal{P}_1^*$. Like $\mathcal{P}_1^*$, $\mathcal{E}_1$ also expects input $(\sigma, \mathsf{c}^\tau)$; $\mathcal{E}_1$ returns a string $(\mathsf{linp}_1, \vec{\mathsf{z}}_2, \vec{\pi}_2, \vec{\hat{\mathsf{c}}}_2)$ that hopefully is (with all but negligible probability) a valid witness for the SNARK statement y_1, assuming that $\mathbb{V}_{\mathbb{C}}$ (and hence also $\mathcal{V}_{e_{\mathcal{V},\mathbb{C}}}$) accepts π_1. (As we shall see later on, showing the validity of such a witness will require invoking semantic security, because the SNARK prover receives c^τ as auxiliary input, while the guarantee of extraction is for when (σ, τ) are drawn independently of the auxiliary input.)

- Use $\mathcal{E}_1$ to construct the new SNARK prover $\mathcal{P}_2^*$ that works as follows: on input $(\sigma, \mathsf{c}^\tau)$, $\mathcal{P}_2^*$ computes $(\mathsf{linp}_1, \vec{\mathsf{z}}_2, \vec{\pi}_2, \vec{\hat{\mathsf{c}}}_2) \leftarrow \mathcal{E}_1(\sigma, \mathsf{c}^\tau)$ and then outputs $(\vec{y}_2, \vec{\pi}_2)$, where the vector of SNARK statements $\vec{y}_2$ contains an entry $y_{(\mathsf{z},\hat{\mathsf{c}})} := (\mathsf{M}_{\mathcal{V},\mathbb{C}}, (\mathsf{z}, \hat{\mathsf{c}}, \mathsf{c}^\tau), t_{\mathcal{V},\mathbb{C}}(|\mathsf{z}| + |\hat{\mathsf{c}}| + |\mathsf{c}^\tau|))$ for each $(\mathsf{z}, \hat{\mathsf{c}})$ in $(\vec{\mathsf{z}}_2, \vec{\hat{\mathsf{c}}}_2)$. Then define $\mathcal{E}_2 := \mathcal{E}_{\mathcal{P}_2^*}$ to be the SNARK extractor for the SNARK prover $\mathcal{P}_2^*$. Given $(\sigma, \mathsf{c}^\tau)$, with all but negligible probability, $\mathcal{E}_2$ should output a witness for each statement and convincing proof (y, π) in $(\vec{y}_2, \vec{\pi}_2)$. (See Remark 4.6.)

- In general, for each $1 < j \leq \mathsf{d}(\mathbb{C})$, we similarly define $\mathcal{P}_j^*$ and $\mathcal{E}_j := \mathcal{E}_{\mathcal{P}_j^*}$.

We can now define the extractor $\mathbb{E}_{\mathbb{P}^*}$. On input $(\sigma, \mathsf{c}^\tau)$, $\mathbb{E}_{\mathbb{P}^*}$ constructs a distributed computation transcript T whose graph is a directed tree, by running $\mathcal{E}_1, \ldots, \mathcal{E}_{\mathsf{d}(\mathbb{C})}$ in order; each such extractor produces a corresponding level in the distributed computation tree. Specifically, each witness $(\vec{\mathsf{z}}, \vec{\pi}, \vec{\hat{\mathsf{c}}}, \mathsf{linp})$ extracted by $\mathcal{E}_j$ corresponds to a node v on the j-th level of the tree, with local input $\mathsf{linp}(v) := \mathsf{linp}$ and incoming messages $\mathsf{inputs}(v) := \vec{\mathsf{z}}$. The tree has a single sink s with only one edge (s', s) going into it; the message on that edge is $\mathsf{data}(s, s') := \mathsf{z}_1$. (Recall that z_1 is the message output by $\mathbb{E}_{\mathbb{P}^*}$.) The leaves of the tree are the vertices for which the extracted witnesses are $(\vec{\mathsf{z}}, \vec{\pi}, \vec{\hat{\mathsf{c}}}, \mathsf{linp}) = \perp$. (See Footnote 12.)

As before, because $\mathsf{d}(\mathbb{C})$ is constant, each $\mathcal{E}_j$ is of polynomial size, and thus $\mathbb{E}_{\mathbb{P}^*}$ is of polynomial size.

Remark 6.5 (SNARK security with auxiliary input)**.** We require that the underlying SNARK is secure with respect to auxiliary inputs that are encryptions of random strings (independently of the state (σ, τ) sampled by the SNARK generator). Using FHE schemes with pseudo-random ciphertexts (e.g., [BV11]), we can relax the auxiliary input requirement to only hold for truly random strings (which directly implies security with respect to pseudo-random strings).

We are left to argue that the transcript T extracted by $\mathbb{E}_{\mathbb{P}^*}$ is $\mathbb{C}$-compliant and has output z_1:

Proposition 6.6. *Let $\mathbb{P}^*$ be a polynomial-size PCD prover, and let $\mathbb{E}_{\mathbb{P}^*}$ be its corresponding polynomial-size extractor as defined above. Then:*

$$\Pr\left[\begin{array}{c} \mathbb{V}_{\mathbb{C}}(\tau, \mathsf{sk}, \mathsf{z}_1, \pi_1, \hat{\mathsf{c}}_1) = 1 \\ \big(\mathsf{out}(\mathsf{T}) \neq \mathsf{z}_1 \ \vee \ \mathbb{C}(\mathsf{T}) \neq 1\big) \end{array} \middle| \begin{array}{r} \big((\sigma, \mathsf{c}^\tau), (\tau, \mathsf{sk})\big) \leftarrow \mathbb{G}(1^k) \\ (\mathsf{z}_1, \pi_1, \hat{\mathsf{c}}_1) \leftarrow \mathbb{P}^*(\sigma, \mathsf{c}^\tau) \\ \mathsf{T} \leftarrow \mathbb{E}_{\mathbb{P}^*}(\sigma, \mathsf{c}^\tau) \end{array} \right] \leq \mathrm{negl}(k) \ .$$

Proof. By construction, $\mathsf{out}(\mathsf{T}) = \mathsf{z}_1$ always. We are left to prove that (with all negligible probability whenever $\mathbb{V}_{\mathbb{C}}$ accepts) it holds that $\mathbb{C}(\mathsf{T}) = 1$. The proof is by induction in the level of the extracted tree (going from root to leaves). Recall that there are at most $\mathsf{d}(\mathbb{C}) = O(1)$ levels all together.

For the base case, we show that for all large enough $k \in \mathbb{N}$, except with negligible probability, whenever the prover $\mathbb{P}^*$ convinces the verifier $\mathbb{V}_{\mathbb{C}}$ to accept $(\mathsf{z}_1, \pi_1, \hat{\mathsf{c}}_1)$, the extractor $\mathcal{E}_1$ outputs $(\mathsf{linp}_1, \vec{\mathsf{z}}_2, \vec{\pi}_2, \vec{\hat{\mathsf{c}}}_2)$ such that:

1. $\mathbb{C}(\mathsf{z}_1; \vec{\mathsf{z}}_2, \mathsf{linp}_1) = 1$,
2. $\hat{\mathsf{c}}_1 = \mathsf{Eval}_{\mathsf{ek}}\left(\prod, (\vec{\hat{\mathsf{c}}}_2, \vec{\hat{\mathsf{c}}}_{\mathcal{V}})\right)$, where each $\hat{\mathsf{c}}_{\mathcal{V}}$ in $\vec{\hat{\mathsf{c}}}_{\mathcal{V}}$ corresponds to one $(\mathsf{z}, \pi, \hat{\mathsf{c}})$ in $(\vec{\mathsf{z}}_2, \vec{\pi}_2, \vec{\hat{\mathsf{c}}}_2)$ and is the result of homomorphically evaluating $\mathcal{V}_{e_{\mathcal{V},\mathbb{C}}}$ as required (i.e., $\hat{\mathsf{c}}_{\mathcal{V}} = \mathsf{Eval}_{\mathsf{ek}}(\mathcal{V}_{e_{\mathcal{V},\mathbb{C}}}(\cdot, y_{(\mathsf{z},\hat{\mathsf{c}})}, \pi), \mathsf{c}^{\tau})$, where $y_{(\mathsf{z},\hat{\mathsf{c}})} := (\mathsf{M}_{\mathcal{V},\mathbb{C}}, (\mathsf{z}, \hat{\mathsf{c}}, \mathsf{c}^{\tau}), t_{\mathcal{V},\mathbb{C}}(|\mathsf{z}| + |\hat{\mathsf{c}}| + |\mathsf{c}^{\tau}|)))$,
3. for each $\hat{\mathsf{c}}$ in $\vec{\hat{\mathsf{c}}}_2$, it holds that $\mathsf{Dec}_{\mathsf{sk}}(\hat{\mathsf{c}}) = 1$, and
4. for each $(\mathsf{z}, \pi, \hat{\mathsf{c}})$ in $(\vec{\mathsf{z}}_2, \vec{\pi}_2, \vec{\hat{\mathsf{c}}}_2)$, letting $y_{(\mathsf{z},\hat{\mathsf{c}})} := (\mathsf{M}_{\mathcal{V},\mathbb{C}}, (\mathsf{z}, \hat{\mathsf{c}}, \mathsf{c}^{\tau}), t_{\mathcal{V},\mathbb{C}}(|\mathsf{z}| + |\hat{\mathsf{c}}| + |\mathsf{c}^{\tau}|))$, it holds that $\mathcal{V}_{e_{\mathcal{V},\mathbb{C}}}(\tau, y_{(\mathsf{z},\hat{\mathsf{c}})}, \pi) = 1$.

Consider the alternative experiment where the prover $\mathbb{P}^*$, instead of receiving the encrypted verification state c^{τ}, receives an encryption of an arbitrary string, say $0^{|\tau|}$, denoted by c^0. We first argue that, in the alternative experiment, whenever $\mathbb{V}_{\mathbb{C}}$ accepts (and except with negligible probability), the first two conditions above must hold, and then (via semantic security) we deduce the same for the original experiment. Indeed, in the the alternative experiment, the SNARK prover $\mathcal{P}_1^*$ is only given the auxiliary input c^0, which is independent of the verification state τ; hence, the SNARK proof of knowledge can be invoked. Specifically, except with negligible probability, whenever $\mathbb{V}_{\mathbb{C}}$ (and hence also $\mathcal{V}_{e_{\mathcal{V},\mathbb{C}}}$) accepts, it must be that the extractor $\mathcal{E}_1$ outputs a valid witness $(\vec{\mathsf{z}}_2, \vec{\pi}_2, \vec{\hat{\mathsf{c}}}_2, \mathsf{linp}_1)$ for the statement $y_{(\mathsf{z}_1,\hat{\mathsf{c}}_1)} = (\mathsf{M}_{\mathcal{V},\mathbb{C}}, (\mathsf{z}_1, \hat{\mathsf{c}}_1, \mathsf{c}^0), t_{\mathcal{V},\mathbb{C}}(|\mathsf{z}_1| + |\hat{\mathsf{c}}_1| + |\mathsf{c}^0|))$ output by $\mathcal{P}_1^*$ (when given c^0 rather than c^{τ}). In particular, by construction of $\mathsf{M}_{\mathcal{V},\mathbb{C}}$, we deduce that $(\vec{\mathsf{z}}_2, \vec{\pi}_2, \vec{\hat{\mathsf{c}}}_2, \mathsf{linp}_1)$ satisfies the first two conditions above. Next, note that the first two conditions above can be efficiently tested given only $(\mathsf{z}_1, \pi_1, \hat{\mathsf{c}}_1, \vec{\mathsf{z}}_2, \vec{\pi}_2, \vec{\hat{\mathsf{c}}}_2, \mathsf{linp}_1, \mathsf{c}^0)$, by running the (deterministic) algorithms $\mathbb{C}$ and $\mathsf{Eval}_{\mathsf{ek}}$. (In particular, neither sk nor τ are required for such a test.) We can thus deduce that in the original experiment, where $\mathcal{P}_1^*$ and $\mathcal{E}_1$ are given c^{τ}, the first conditions hold with all but negligible probability. (For, otherwise, we could break the semantic security of the encryption scheme, by distinguishing encryptions of a random τ from encryptions of $0^{|\tau|}$.)

We have thus established that whenever $\mathbb{V}_{\mathbb{C}}$ accepts (and except with negligible probability), the first two conditions above must hold. We now argue that whenever the second condition holds, we can deduce the last two conditions. Indeed, since the statement $y_{(\mathsf{z}_1,\hat{\mathsf{c}}_1)}$ is accepted by $\mathcal{V}$, we know that $\mathsf{Dec}_{\mathsf{sk}}(\hat{\mathsf{c}}_1) = 1$. This and the correctness of $\mathsf{Eval}_{\mathsf{ek}}$ implies that all ciphertexts in $(\vec{\hat{\mathsf{c}}}_2, \vec{\hat{\mathsf{c}}}_{\mathcal{V}})$ must *also* decrypt to "1".[15] Hence, we deduce the third property (all ciphertexts in $\vec{\hat{\mathsf{c}}}_2$ decrypt to "1"), and by invoking the correctness of $\mathsf{Eval}_{\mathsf{ek}}$ once more we can deduce the last property (namely, for each $(\mathsf{z}, \pi, \hat{\mathsf{c}})$ in $(\ vadata_2, \vec{\pi}_2, \vec{\hat{\mathsf{c}}}_2)$, it holds that $\mathcal{V}_{e_{\mathcal{V},\mathbb{C}}}(\tau, y_{(\mathsf{z},\hat{\mathsf{c}})}, \pi) = 1$).

To complete the proof, we can proof in a similar manner the inductive step. That is, assuming that conditions three and four are satisfied by the j-th level of the tree, we can deduce that conditions one and two hold for level $j + 1$. This is done by first establishing conditions one and two in an alternative experiment where c^{τ} is replaced by c^0, and then invoking semantic security to deduce the same for the

[15] We can assume without loss of generality that all ciphertexts are decrypted to "0" or "1", either by using an encryption scheme where any ciphertext can be interpreted as such, or by adding simple consistency checks to the evaluated circuit.

original experiment. We then deduce, from the second property, the last two properties as we did for the base case. Overall, we can conclude that T is $\mathbb{C}$-compliant. □

7 Proof Of The Locally-Efficient RAM Compliance Theorem

We provide here the technical details for the high-level discussion in Section 2.4. Concretely, we prove the Locally-Efficient RAM Compliance Theorem, which is one of the three tools we use in the proof of our main result (discussed in Section 9). Throughout this section, it will be useful to keep in mind the definitions from Section 3 (where random-access machines and the universal language $\mathcal{L}_{\mathcal{U}}$ are introduced) and Section 5.1 (where the notions of distributed computation transcripts, compliance predicates, and depth are introduced).

We prove that membership in $\mathcal{L}_{\mathcal{U}}$ of an instance $y = (M, x, t)$ with $t \leq k^{\log k}$ can be "computationally reduced" to the question of whether there is a distributed computation compliant with $\mathbb{C}_y^h$ whose output is a predetermined value (e.g., the string "ok"), where $\mathbb{C}_y^h$ is a compliance predicate of depth $t \cdot \mathrm{poly}(k)$ and h is drawn from a collision-resistant hash-function family. Furthermore, it suffices to consider $\mathrm{poly}(k + |y|)$-bounded distributed computations (i.e, that are "locally-efficient"), and such a distributed computation can be generated from the instance y and a witness w for y in time $(|M| + |x| + t) \cdot \mathrm{poly}(k)$ and space $(|M| + |x| + s) \cdot \mathrm{poly}(k)$.

Theorem 7.1 (Locally-Efficient RAM Compliance Theorem)**.** *Let $\mathcal{H} = \{\mathcal{H}_k\}_{k\in\mathbb{N}}$ be a collision-resistant hash-function family. There exist functions $\Phi, \Psi_0, \Psi_1 \colon \{0,1\}^* \to \{0,1\}^*$ such that:*

1. Completeness: *For every instance $y = (M, w, t)$, witness w with $(y, w) \in \mathcal{R}_{\mathcal{U}}$, $k \in \mathbb{N}$ with $t \leq k^{\log k}$, and $h \in \mathcal{H}_k$, it holds that $\mathbb{C}_y^h(\mathsf{T}) = 1$ and $\mathsf{out}(\mathsf{T}) = \mathsf{ok}$, where $\mathbb{C}_y^h := \Phi(h, y)$ is a compliance predicate and $\mathsf{T} := \Psi_0(h, y, w)$ is a distributed computation transcript.*

2. Proof of knowledge: *For every polynomial-size adversary $\mathcal{A}$ and large enough security parameter $k \in \mathbb{N}$:*
$$\Pr\left[\begin{array}{c|r} t \leq k^{\log k} & h \leftarrow \mathcal{H}_k \\ \mathbb{C}_y^h(\mathsf{T}) = 1 & (M, x, t, \mathsf{T}) \leftarrow \mathcal{A}(h) \\ \mathsf{out}(\mathsf{T}) = \mathsf{ok} & y \leftarrow (M, w, t) \\ (y, w) \notin \mathcal{R}_{\mathcal{U}} & \mathbb{C}_y^h \leftarrow \Phi(h, y) \\ & w \leftarrow \Psi_1(h, y, \mathsf{T}) \end{array}\right] \leq \mathrm{negl}(k) \ .$$

3. Efficiency:
   - $\mathsf{d}(\mathbb{C}_y^h) \leq t \cdot \mathrm{poly}(k)$*;*
   - $\Phi(h, y)$ *runs in linear time;*
   - $\Psi_0(h, y, w)$ *is a* $\mathrm{poly}(k + |y|)$*-bounded distributed computation transcript whose graph is a path; furthermore, $\Psi_0(h, y, w)$ outputs the transcript in topological order while running in time* $(|M| + |x| + t) \cdot \mathrm{poly}(k)$ *and space* $(|M| + |x| + s) \cdot \mathrm{poly}(k)$*, where s is the space complexity of $M(x, w)$;*
   - $\Psi_1(h, y, \mathsf{T})$ *runs in linear time.*

The Locally-Efficient RAM Compliance Theorem thus ensures a very efficient *computational Levin reduction*[16] from verifying membership in $\mathcal{L}_{\mathcal{U}}$ to verifying certain local properties of distributed computations.

[16]Recall that a Levin reduction is a Karp (instance) reduction that comes with witness reductions going "both ways"; in the theorem statement, the instance reduction is Φ, the "forward" witness reduction is Ψ_0, and the "backward" witness reduction is Ψ_1. The soundness guarantee provided by Φ is only computational.

When invoking the reduction for a given instance y and then using a PCD system to enforce the compliance predicate $\mathbb{C}_y^h$, Ψ_0 preserves the completeness property of the PCD prover, while Ψ_1 ensures that the proof-of-knowledge property of the PCD verifier is preserved. (Conversely, if the PCD system used does not have a proof-of-knowledge property, then the Locally-Efficient RAM Compliance Theorem cannot be used, as can be seen from the security guarantee of the theorem statement. See the proof of Theorem 4 for more details.)

As discussed in Section 2.4, the proof of the Locally-Efficient RAM Compliance Theorem consists of two steps, respectively discussed in the next two subsections (Section 7.1 and Section 7.2).

Remark 7.2 (recalling random-access machines)**.** Random-access machines can be defined in many ways, depending on the choice of *architecture* (e.g., stack, accumulator, load/store, register/memory, and so on). In this work, we do not need to present a formal definition, but having a very rough idea of how random-access machines work will be helpful towards a better understanding of the material discussed in this section. For concreteness, our discussions assume random-access machines following the familiar load/store architecture; also, we assume that the random-access machine has sequential access to two tapes, one for the input and one for the witness. For additional details see, e.g., [BSCGT13].

7.1 Machines With Untrusted Memory

Ben-Sasson et al. [BSCGT13] observed that, provided collision-resistant hash functions exist, membership of an instance $y = (M, x, t)$ in the universal language $\mathcal{L}_{\mathcal{U}}$ can be "simplified" to membership of a corresponding instance $y' = (M', x, t')$ where M' is a machine with $\mathrm{poly}(k)$ space complexity and $t' = t \cdot \mathrm{poly}(k)$, when $t \leq k^{\log k}$. We briefly recall here their result, which follows from techniques for *online memory checking* [BEG+91].[17]

Lemma 7.3 ([BSCGT13])**.** *Let $\mathcal{H} = \{\mathcal{H}_k\}_{k\in\mathbb{N}}$ be a collision-resistant hash-function family. There exist functions $\Phi, \Psi_0, \Psi_1 \colon \{0,1\}^* \to \{0,1\}^*$ and $b \colon \mathbb{N}^2 \to \mathbb{N}$ such that:*

1. Syntax: *For every random-access machine M, $k \in \mathbb{N}$, $h \in \mathcal{H}_k$, $\Phi(h, M)$ is a random-access machine.*
2. Witness Reductions:
   - *For every instance $y = (M, x, t)$, witness w with $(y, w) \in \mathcal{R}_{\mathcal{U}}$, $k \in \mathbb{N}$ with $t \leq k^{\log k}$, and $h \in \mathcal{H}_k$, it holds that $\big((\Phi(h, M), x, b(t, k)), \Psi_0(h, y, w)\big) \in \mathcal{R}_{\mathcal{U}}$.*
   - *For every polynomial-size adversary $\mathcal{A}$ and sufficiently large $k \in \mathbb{N}$,*

$$\Pr\left[\begin{array}{c} t \leq k^{\log k} \\ \big((\Phi(h, M), x, b(t, k)), w'\big) \in \mathcal{R}_{\mathcal{U}} \\ (y, w) \notin \mathcal{R}_{\mathcal{U}} \end{array} \middle| \begin{array}{r} h \leftarrow \mathcal{H}_k \\ (M, x, t, w') \leftarrow \mathcal{A}(h) \\ y \leftarrow (M, x, t) \\ w \leftarrow \Psi_1(h, y, w') \end{array}\right] \leq \mathrm{negl}(k) \ .$$

3. Efficiency:
   - *$\Phi(h, M)$ is a $\mathrm{poly}(k)$-space random-access machine and $b(t, k) = t \cdot \mathrm{poly}(k)$;*
   - *$\Phi(h, M)$ and $\Psi_1(h, y, w')$ run in linear time;*
   - *$\Psi_0(h, y, w)$ runs in time $(|M| + |x| + t) \cdot \mathrm{poly}(k)$ and space $(|M| + |x| + s) \cdot \mathrm{poly}(k)$, where s is the space complexity of $M(x, w)$.*

[17] Unlike in [BEG+91], in our work (as in [BSCGT13]) universal one-way hash functions [NY89, Rom90] do not suffice because the machine M receives, besides the input x, a (potentially-malicious) witness w.

Remark 7.4. For computations that do not use more than $\mathrm{poly}(k)$ space, the RAM Untrusted Memory Lemma is not needed and one can directly proceed to the next step (discussed in Section 7.2).

Proof sketch. The idea is to construct from M a new machine $M' := \Phi(h, M)$ that uses the hash function h to delegate memory to "untrusted storage" by dynamically maintaining a Merkle tree over such storage.

More precisely, the program of M' is equal to the program of M after replacing every load and store instruction with corresponding sequences of instructions (which include computations of h) that implement *secure loads* and *secure stores*.[18] This mapping from h and M to M' can be performed in linear time by a function Φ.

The new machine M' always keeps in a register the most up-to-date root of the Merkle tree. Secure loads and secure stores are not atomic instructions in the new machine M' but instead "expand" into macros consisting of basic instructions (which include many "insecure" load and store instructions). Concretely, a secure load for address i loads from memory a claimed value and claimed hash, and then also loads all the other relevant information for the authentication path of the i-th leaf, in order to check the value against the locally-stored Merkle root. A secure store for address i updates the relevant information for the authentication path of the i-th leaf and then updates the locally-stored Merkle root. Because each secure load and secure store takes $\mathrm{poly}(k)$ instructions to complete, the running time of M' increases only by a multiplicative factor of $\mathrm{poly}(k)$.

The security property of h ensures that it is hard for the (efficient) untrusted storage to return inconsistent values. By thinking of the sequence of accessed values during the computation of M' on (x, w) as part of the new witness for M' (and not as part of memory), then we see that M' is "computationally equivalent" to M, except that its space requirement is only $\mathrm{poly}(k)$.

Given a witness w for M, extending w to a witness w' for M' (which includes all the memory accesses of the computation) can be done in time $(|M| + |x| + t) \cdot \mathrm{poly}(k)$ and space $(|M| + |x| + s) \cdot \mathrm{poly}(k)$ by a function Ψ_0 by simply running the computation. Going from a witness w' for M' to a witness w for M only requires Ψ_1 to take a prefix of w', and thus can be done in linear time. □

7.2 A Compliance Predicate for Checking RAM Computations

We show how membership of an instance $y = (M, x, t)$ can be reduced to the question of whether there is a distributed computation compliant with $\mathbb{C}_y$ whose output is a predetermined value (e.g., the string "ok"), where $\mathbb{C}_y$ is a compliance predicate of depth $O(t)$ that we call the *RAM Checker* for y. Furthermore, it suffices to consider $O(s + |y|)$-bounded distributed computations whose graph is a path, where s is the space complexity of M, and such a distributed computation can be generated from the instance y and a witness w in time $O(|M| + |x| + t)$ and space $O(|M| + |x| + s)$.

Essentially, $\mathbb{C}_y$ forces any distributed computation compliant with it to check the computation of M on x one step at a time, for at most t steps, and the only way such a distributed computation can produce the message ok is to reach an accepting state.

In Remark 7.7 below we explain how Lemma 7.3 and Lemma 7.5 (which formalizes the aforementioned reduction) can be combined to obtain the Locally-Efficient RAM Compliance Theorem (Theorem 7.1).

Lemma 7.5. *There exist functions $\Phi, \Psi_0, \Psi_1 \colon \{0,1\}^* \to \{0,1\}^*$ such that for every instance $y = (M, x, t)$:*

1. Syntax: $\mathbb{C}_y := \Phi(y)$ *is a compliance predicate.*

[18] In fact, the computation of M' begins with an initialization stage during which M' computes a Merkle tree over a sufficiently-large all-zero memory, and then proceeds to execute the (modified) program of M. One also needs to take care of additional technical details, such as ensuring that M' has enough registers to compute h and the register width is large enough for images of h

2. Witness Reductions:
   - *For every witness w with $(y, w) \in \mathcal{R}_{\mathcal{U}}$, $\mathbb{C}_y(\Psi_0(y, w)) = 1$ and $\mathsf{out}(\Psi_0(y, w)) = \mathsf{ok}$.*
   - *For every transcript T with $\mathbb{C}_y(\mathsf{T}) = 1$ and $\mathsf{out}(\mathsf{T}) = \mathsf{ok}$, $(y, \Psi_1(y, \mathsf{T})) \in \mathcal{R}_{\mathcal{U}}$.*
3. Efficiency:
   - $\mathsf{d}(\mathbb{C}_y) \leq t + 1$*;*
   - $\Phi(y)$ *runs in linear time;*
   - $\Psi_0(y, w)$ *is a $O(s+|y|)$-bounded distributed computation transcript whose graph is a path; furthermore, $\Psi_0(y, w)$ outputs the transcript in topological order while running in time $O(|M| + |x| + t)$ and space $O(|M| + |x| + s)$, where s is the space complexity of $M(x, w)$;*
   - $\Psi_1(y, \mathsf{T})$ *runs in linear time.*

Proof. We begin by giving the construction of the compliance predicate $\mathbb{C}_y$ from the instance y:

Construction 7.6. The *RAM Checker* $\mathbb{C}_y$ for an instance $y = (M, x, t)$ is defined as follows:

$\mathbb{C}_y(\mathsf{z_o}; \vec{\mathsf{z}}_\mathsf{i}, \mathsf{linp}) \stackrel{\text{def}}{=}$

1. Verify that $\vec{\mathsf{z}}_\mathsf{i} = (\mathsf{z_i})$ for some $\mathsf{z_i}$.
2. If $\mathsf{z_i} = \perp$:
   (a) Verify that $\mathsf{linp} = \perp$.
   (b) Verify that $\mathsf{z_o} = (\tau', S')$ for some timestamp τ' and state S' of M.
   (c) Verify that S' is an initial state of M.
3. If $\mathsf{z_o} = \mathsf{ok}$:
   (a) Verify that $\mathsf{linp} = \perp$.
   (b) Verify that $\mathsf{z_i} = (\tau, S)$ for some timestamp τ and state S of M.
   (c) Verify that S is a final accepting state of M.
4. Otherwise:
   (a) Verify that $\mathsf{z_i} = (\tau, S)$ for some timestamp τ and state S of M.
   (b) Verify that $\mathsf{z_o} = (\tau', S')$ for some timestamp τ' and state S' of M.
   (c) Verify that $\tau, \tau' \in \{0, 1, \ldots, t\}$.
   (d) Verify that $\tau' = \tau + 1$.
   (e) Verify that executing a single step of M starting with state S results in state S', when x is on the first tape of M and by supplying linp as the answer to a read to the second tape (if such a read is made).

The *state* of a random-access machine contains the values of the registers and the program counter, the position of the head on the two tapes, and the contents of random-access memory. It is thus easy to see that $\mathbb{C}_y(\mathsf{z_o}; \vec{\mathsf{z}}_\mathsf{i}, \mathsf{linp})$ runs in time $O(s + |y|)$, and thus it suffices to consider $O(s + |y|)$-bounded distributed computations.

Note that Case 1 ensures that $\vec{\mathsf{z}}_\mathsf{i}$ is a vector consisting of a single component; in particular, any distributed computation that is compliant with $\mathbb{C}_y$ must be a collection of disjoint paths. Case 2 is triggered when checking the first node of any such path (due to the condition $\mathsf{z_i} = \perp$), and verifies that the output data consists of a timestamped initial state of M. Case 3 is triggered whenever the output data is equal to ok (i.e., $\mathsf{z_o} = \mathsf{ok}$), and verifies that the input data consists of a timestamped final and accepting state of M. Case 4 is triggered at all other times; it verifies that both input and output data consist of timestamped states (so that, in particular, if a path contains the message with data ok, that message is the single and last message), that the timestamp grows by 1, and that $M(x, \cdot)$ goes from one state to the next when using linp as nondeterminism.

The mapping from y to $\mathbb{C}_y$ from Construction 7.6 can be performed by a function Φ in linear time.

Also, the depth (see Definition 5.8) of $\mathbb{C}_y$ is bounded by the time bound: specifically, $\mathsf{d}(\mathbb{C}_y) \leq t+1$. Indeed, as mentioned in Construction 7.6, any transcript that is compliant with $\mathbb{C}_y$ consists of disjoint paths. Because $\mathbb{C}_y$ ensures, along any such path, that timestamps increase by 1 from one message to the next and are bounded by t, the depth of $\mathbb{C}_y$ is at most $t+1$. (The "+1" comes from the ok message.)

Next, we discuss the witness reductions, by defining Ψ_0 and Ψ_1.

Define $\tilde{t}$ to be the number of steps that it takes for $M(x,w)$ to halt (note that $\tilde{t} \leq t$), and $S_0, \ldots, S_{\tilde{t}}$ to be the sequence of corresponding states. Define $a = (a_i)_{i=1}^{\tilde{t}}$ so that a_i is equal to the value read from the second tape in the i-th step (or an arbitrary value if no value is read from there in the i-th time step).

Next, define $\mathsf{T} := (G, \mathsf{linp}, \mathsf{data})$ where G is the (directed) path graph of $\tilde{t}+3$ nodes labeled $0, \ldots, \tilde{t}+2$, $\mathsf{linp}(0) := \mathsf{linp}(\tilde{t}+1) := \mathsf{linp}(\tilde{t}+2) := \perp$ and $\mathsf{linp}(i) := a_i$ for $i = 1, \ldots, \tilde{t}$, $\mathsf{data}(i, i+1) := (i, S_i)$ for $i = 0, 1, \ldots, \tilde{t}$, and $\mathsf{data}(\tilde{t}+1, \tilde{t}+2) := \mathsf{ok}$. In other words, T is the path whose vertices are labeled with the sequence a (and the sink and the source are labeled with $\perp$) and whose edges are labeled with the timestamped sequence of states of M followed by ok. See Figure 5 for a diagram.

On input (y,w), a function Ψ_0 can output T in topological order, in time $O(|M| + |x| + t)$ and space $O(|M| + |x| + s)$, by simply simulating $M(x,w)$ for at most t time steps, outputting labeled vertices and edges as it proceeds from one state of M to the next, and then adding the message ok after M halts. If $(y,w) \in \mathcal{R}_{\mathcal{U}}$, it is easy to see that $\mathbb{C}_y$ holds everywhere in T (so that $\mathbb{C}_y(\mathsf{T}) = 1$) and, moreover, T has output data ok (i.e., $\mathsf{out}(\mathsf{T}) = \mathsf{ok}$).

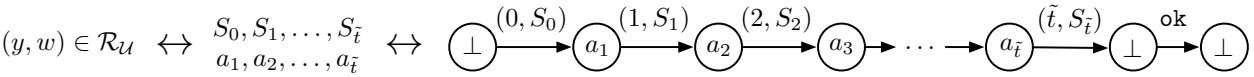

Figure 5: Constructing a $\mathbb{C}_y$-compliant transcript T starting from $(y,w) \in \mathcal{R}_{\mathcal{U}}$, and vice versa.

Now suppose that T is any transcript compliant with $\mathbb{C}_y$ and has output data ok (i.e., $\mathbb{C}_y(\mathsf{T}) = 1$ and $\mathsf{out}(\mathsf{T}) = \mathsf{ok}$). Because $\mathbb{C}_y$ disallows more than one message into a node, the graph of T is a set of disjoint paths. By assumption, there is a path p where the input data to the sink is equal to ok. Now construct w as follows. Let I be the subset of $[t]$ consisting of those indices i for which M, at the state transition of the $(i+1)$-th node in T, reads the next value from the second tape. Define $w := (\mathsf{linp}(i))_{i \in I}$, where the indexing is with respect to nodes in the path p. By compliance of T with $\mathbb{C}_y$ and because we know that the path p ends with the message ok, we deduce that $(y,w) \in \mathcal{R}_{\mathcal{U}}$. Once again see Figure 5.

Finally, on input (y, T), a function Ψ_1 can output w in linear time. □

Remark 7.7 (combining Lemma 7.3 and Lemma 7.5 to obtain Theorem 7.1)**.** In Section 7.1 we discussed how nondeterminism can be used to reduce the space complexity of a random-access machine to $\mathrm{poly}(k)$, by only incurring in a blowup in running time of $\mathrm{poly}(k)$. When combining the reduction of Lemma 7.3 from Section 7.1 with the reduction from Lemma 7.5 in this section, we obtain a proof to the Locally-Efficient RAM Compliance Theorem (Theorem 7.1). Concretely, first an instance y is reduced to a new instance y' by using a collision-resistant hash function (via Lemma 7.3), and then y' is reduced to $\mathbb{C}_{y'}$, the RAM Checker for y' (via Lemma 7.5).

8 Proof of The PCD Depth-Reduction Theorem

We provide here the technical details for the high-level discussion in Section 2.3. Concretely, we prove the PCD Depth-Reduction Theorem, which is one of the three tools we use in the proof of our main result (discussed in Section 9). Throughout this section, it will be useful to keep in mind the definitions from

Section 5.1 (where the notions of distributed computation transcripts, compliance predicates, and depth are introduced).

Recall that the SNARK Recursive Composition Theorem (discussed at high level in Section 2.2 and formally proved in Section 6) transforms any SNARK into a corresponding PCD system for (polynomially-balanced) constant-depth compliance predicates. The Locally-Efficient RAM Compliance Theorem (discussed at high level in Section 2.4 and formally proved in Section 7) tells us that membership in $\mathcal{L}_{\mathcal{U}}$ of an instance $y = (M, x, t)$ with $t \leq k^{\log k}$ can be "computationally reduced" to the question of whether there is a "locally-efficient" distributed computation compliant with $\mathbb{C}_y^h$ whose output is a predetermined value (e.g., the string "ok"), where $\mathbb{C}_y^h$ is a compliance predicate of depth $t \cdot \mathrm{poly}(k)$ and h is drawn from a collision-resistant hash-function family.

Unfortunately, the depth of $\mathbb{C}_y^h$ is *superconstant*. Thus, it seems that we cannot benefit from the SNARK Recursive Composition Theorem. (Unless we make stronger extractability assumptions; see Remark 6.3.)

To address the aforementioned problem and, more generally, to better understand the expressive power of constant-depth compliance predicates, we prove in this section a "Depth-Reduction Theorem" for PCD: a PCD system for constant-depth compliance predicates can be transformed into a corresponding *path* PCD system for polynomial-depth compliance predicates; furthermore, the transformation preserves the verifiability and efficiency properties of the PCD system. (This holds more generally; see Remark 8.8.)

Theorem 8.1 (PCD Depth-Reduction Theorem)**.** *Let $\mathcal{H} = \{\mathcal{H}_k\}_{k \in \mathbb{N}}$ be a collision-resistant hash-function family. There exists an efficient transformation $\textsc{DepthRed}_{\mathcal{H}}$ with the following properties:*

- Correctness: *If $(\mathbb{G}, \mathbb{P}, \mathbb{V})$ is a PCD system for constant-depth compliance predicates, then $(\mathbb{G}', \mathbb{P}', \mathbb{V}') = \textsc{DepthRed}_{\mathcal{H}}(\mathbb{G}, \mathbb{P}, \mathbb{V})$ is a path PCD for polynomial-depth compliance predicates.*[19]
- Verifiability Properties:
  - *If $(\mathbb{G}, \mathbb{P}, \mathbb{V})$ is publicly verifiable then so is $(\mathbb{G}', \mathbb{P}', \mathbb{V}')$.*
  - *If $(\mathbb{G}, \mathbb{P}, \mathbb{V})$ is designated verifier then so is $(\mathbb{G}', \mathbb{P}', \mathbb{V}')$.*
- Efficiency: *There exists a polynomial p such that the (time and space) efficiency of $(\mathbb{G}', \mathbb{P}', \mathbb{V}')$ is the same as that of $(\mathbb{G}, \mathbb{P}, \mathbb{V})$ up to the multiplicative factor $p(k)$.*

The main claim behind the theorem is that we can achieve an *exponential* improvement in the depth of a given compliance predicate $\mathbb{C}$, while at the same time maintaining completeness for transcripts that are paths, by constructing a new low-depth compliance predicate $\textsc{Tree}_{\mathbb{C}}$ that is a "tree version" of $\mathbb{C}$. One can then construct a new PCD system $(\mathbb{G}', \mathbb{P}', \mathbb{V}')$ that, given a compliance predicate $\mathbb{C}$, appropriately uses the old PCD system $(\mathbb{G}, \mathbb{P}, \mathbb{V})$ to enforce $\textsc{Tree}_{\mathbb{C}}$.

The basic idea is that the new compliance predicate $\textsc{Tree}_{\mathbb{C}}$ is to force any compliant distributed computation to build a Merkle tree of proofs with large in-degree r "on top" of the original distributed computation. (This technique combines the ideas of proof trees of Valiant [Val08] and of wide Merkle trees used in the security reduction of [BCCT12, GLR11].) As a result, the depth of the new compliance predicate will be $\lfloor \log_r \mathsf{d}(\mathbb{C}) \rfloor + 1$; in particular, when $\mathsf{d}(\mathbb{C})$ is bounded by a polynomial in the security parameter k (as is the case for the compliance predicate $\mathbb{C}_y^h$ produced by the Locally-Efficient RAM Compliance Theorem), by setting $r = k$, the depth of $\textsc{Tree}_{\mathbb{C}}$ becomes *constant* — and we can now benefit from the SNARK Recursive Composition Theorem.

[19] Recall that a **path** PCD system is one where *completeness* does not necessarily hold for any compliant distributed computation, but only for those where the associated graph is a path, i.e., each node has only a single input message. See Definition 5.9.

For expository purposes, in Section 8.1 give the intuition for the proof of the PCD Depth-Reduction Theorem for the specific compliance predicate $\mathbb{C}_y^h$ produced by the Locally-Efficient RAM Compliance Theorem. This concrete example, where we explain how to construct a Merkle tree of proofs on top of the step-by-step computation of a random-access machine with $\mathrm{poly}(k)$ space complexity, will build the necessary intuition for the more abstract setting of the general case (needed for our main theorem), which we present in Section 8.2.

8.1 Warm-Up Special Case: Reducing The Depth Of RAM Checkers

As discussed, we sketch the proof of the PCD Depth-Reduction Theorem for the special case where the desired compliance predicate is $\mathbb{C}_y^h$; recall that $\mathbb{C}_y^h$ is the compliance predicate generated by the Locally-Efficient RAM Compliance Theorem (Theorem 7.1) when invoked on the instance $y = (M, x, t)$. By relying on certain properties of $\mathbb{C}_y^h$, we are able to give a simpler proof sketch, and thereby build intuition for the general case (discussed in Section 8.2). Thus, the goal for now is to construct a path PCD system for $\mathbb{C}_y^h$, while only assuming the existence of PCD systems for constant-depth compliance predicates. Moreover, we must ensure that the verifiability and efficiency properties of the new PCD system are essentially the same as those of the PCD system we start with.

Step 1: Engineer a new compliance predicate. Recall from the proof of the Locally-Efficient RAM Compliance Theorem (discussed in Section 7) that $\mathbb{C}_y^h = \mathbb{C}_{y'}$ (see Remark 7.7), where $\mathbb{C}_{y'}$ is the RAM Checker for the instance $y' = (M', x, t')$, M' is a $\mathrm{poly}(k)$-space machine, and $t' = t \cdot \mathrm{poly}(k)$ when $t \leq k^{\log k}$ (see Section 7.1). Starting from $\mathbb{C}_{y'}$ and an in-degree parameter r, we show how to construct a *new* compliance predicate $\textsc{Tree}\mathbb{C}_{y'}^{\triangleq r}$ with $O(\log_r \mathsf{d}(\mathbb{C}_{y'}))$ depth.

The intuition of the construction is for $\textsc{Tree}\mathbb{C}_{y'}^{\triangleq r}$ to force any distributed computation that is compliant with it to have the structure of an r-ary tree whose leaves form a path that is compliant with $\mathbb{C}_{y'}$. In order to achieve this, $\textsc{Tree}\mathbb{C}_{y'}^{\triangleq r}$ enforces data flowing through the distributed computation to carry certain "metadata" information that helps $\textsc{Tree}\mathbb{C}_{y'}^{\triangleq r}$ figure out "where" in the distributed computation a given piece of data belongs. With this information available, $\textsc{Tree}\mathbb{C}_{y'}^{\triangleq r}$ can then reason as follows (see Construction 8.2 below for reference):

- *Leaf Node Stage:* the input data to the node consists of two messages $(0, \tau_1, S_1)$ and $(0, \tau_2, S_2)$. Then $\textsc{Tree}\mathbb{C}_{y'}^{\triangleq r}$ interprets (τ_1, S_1) and (τ_2, S_2) as two timestamped states of M' and uses $\mathbb{C}_{y'}$ to check that $\tau_2 = \tau_1 + 1$ and that the state S_2 follows from S_1 in one time step; then $\textsc{Tree}\mathbb{C}_{y'}^{\triangleq r}$ checks that the output data is $(1, \tau_1, S_1, \tau_2, S_2)$, which should be interpreted as claiming that the verification of the time interval $[\tau_1, \tau_2]$ for the machine M' took place.

- *Internal Node Stage:* the input data to the node consists of r messages $(d, \tau_i, S_i, \tau_i', S_i')$ each from a node at "level d of the tree". We interpret each message $(d, \tau_i, S_i, \tau_i', S_i')$ as claiming that verification of the time intervals $[\tau_i, \tau_i']$ of M' took place, and that the state of M' at time τ_i and τ_i' respectively was S_i and S_i'; $\textsc{Tree}\mathbb{C}_{y'}$ checks that these intervals are in fact contiguous and are accompanied by consistent states of the machine M'; then $\textsc{Tree}\mathbb{C}_{y'}$ checks that the output data is $(d+1, \tau_1, S_1, \tau_r', S_r')$, that is, that it correctly "collapses" the r input messages.

- *Output Stage:* for some r', the input data to the node consists of r' messages $(d_i, \tau_i, S_i, \tau_i', S_i')$ each from a node at "level d_i of the tree". The fact that the messages are coming from different levels of the tree signals that the node wants to claim that the computation of M' is done, and in this case $\textsc{Tree}\mathbb{C}_{y'}^{\triangleq r}$ verifies that the input messages carry consistent timestamps and states (as in the previous

case) and furthermore checks that $\tau_1 = 0$ and $S'_{r'} = \mathsf{ok}$. Then $\text{TREE}\mathbb{C}^{\triangleq r}_{y'}$ checks that the output data is $(\mathtt{data}, \mathsf{ok})$.

Given the above rough description, the only way to produce the message ok in a distributed computation compliant with $\text{TREE}\mathbb{C}^{\triangleq r}_{y'}$ is for a distributed computation to separately check each step of M' and then iteratively merge r intervals at a time, for a total of $\log_r t'$ times, until it produces a root attesting to the correct computation of M' for t' steps. When $t' \leq k^c$ for some c, $\log_r t'$ is constant by setting $r = k$. For reference, we give the following more precise construction (which, for instance, also shows how to deal with the nondeterminism of M'):

Construction 8.2. Given $r \in \mathbb{N}$ and an instance $y' = (M', x, t')$, define the following compliance predicate:

$\text{TREE}\mathbb{C}^{\triangleq r}_{y'}(\mathsf{z_o}; \vec{\mathsf{z}}_\mathsf{i}, \mathsf{linp}) \stackrel{\text{def}}{=}$

1. Input Stage
   If $\vec{\mathsf{z}}_\mathsf{i} = \bot$ and $\mathsf{linp} = \bot$:
   (a) Verify that $\mathsf{z_o}$ equals $(0, \tau, S)$ for some τ, S.
   (b) If $\tau = 0$, verify that $\mathbb{C}_{y'}((\tau, S); \bot, \bot)$
2. Leaf Node Stage
   If $\mathsf{z_o} = (1, \tau, S, \tau', S')$ and $\vec{\mathsf{z}}_\mathsf{i} = \big((0, \tau_1, S_1), (0, \tau_2, S_2)\big)$ for some $\tau, S, \tau', S', \tau_1, S_1, \tau_2, S_2$:
   (a) Verify that $\tau_2 = \tau_1 + 1$, $\tau = \tau_1$, and $\tau' = \tau_2$.
   (b) Verify that $S = S_1$ and $S' = S_2$.
   (c) If $S_2 = \mathsf{ok}$, verify that $\mathbb{C}_{y'}(\mathsf{ok}; ((\tau_1, S_1)), \bot)$ accepts.
   (d) Otherwise, verify that $\mathbb{C}_{y'}((\tau_2, S_2); ((\tau_1, S_1)), \mathsf{linp})$ accepts.
3. Internal Node Stage
   If $\mathsf{z_o} = (d{+}1, \tau, S, \tau', S')$ and $\vec{\mathsf{z}}_\mathsf{i} = \big((d, \tau_i, S_i, \tau'_i, S'_i)\big)_{i=1}^{r}$ for some $d, \tau, S, \tau', S', \tau_1, \ldots, \tau_r, S_1, \ldots, S_r$:
   (a) Verify that $\tau = \tau_1$, $\tau'_1 = \tau_2$, $\tau'_2 = \tau_3$, and so on until $\tau'_{r-1} = \tau_r$, $\tau'_r = \tau'$.
   (b) Verify that $S = S_1$, $S'_1 = S_2$, $S'_2 = S_3$, and so on until $S'_{r-1} = S_r$, $S'_r = S'$.
4. Output Stage
   If $\mathsf{z_o} = (\mathtt{data}, \mathsf{ok})$ and $\vec{\mathsf{z}}_\mathsf{i} = \big((d_i, \tau_i, S_i, \tau'_i, S'_i)\big)_{i=1}^{r'}$ for some $\mathsf{z}, d_1, \ldots, d_{r'}, \tau_1, \ldots, \tau_{r'}, S_1, \ldots, S_{r'}$:
   (a) Verify that $\tau'_1 = \tau_2$, $\tau'_2 = \tau_3$, and so on until $\tau'_{r'-1} = \tau_{r'}$.
   (b) Verify that $S'_1 = S_2$, $S'_2 = S_3$, and so on until $S'_{r'-1} = S_{r'}$.
   (c) Verify that $\tau_1 = 0$ and $S'_{r'} = \mathsf{ok}$.
5. If none of the above conditions hold, reject.

Recall from Lemma 7.5 that the depth of the old compliance predicate $\mathbb{C}_{y'}$ could be as bad as $t' + 1$. Instead, as promised, the depth of the new compliance predicate $\text{TREE}\mathbb{C}^{\triangleq r}_{y'}$ is much better:

Lemma 8.3. $\mathsf{d}\big(\text{TREE}\mathbb{C}^{\triangleq r}_{y'}\big) \leq \lfloor \log_r(t' + 1) \rfloor + 1$.

Proof. Any transcript compliant with $\text{TREE}\mathbb{C}^{\triangleq r}_{y'}$ consists of disjoint trees. In each such tree, nodes of different heights are forced to directly point to the root of the tree, and other nodes of the same height are grouped in sets of size r. Thus, the "worst possible height", given that any tree can have at most $t'+1$ leaves, is given by $\lfloor \log_r(t' + 1) \rfloor + 1$ (achieved by making maximal use of merging nodes of the same height). □

The depth reduction is meaningful because we can accompany it with guarantees that ensure that despite the fact that we switched to a new compliance predicate, the "semantics" of the compliance predicate have been preserved. Namely, given a transcript T compliant with $\mathbb{C}_{y'}$ and with output data ok, we can efficiently

produce a new transcript T' compliant with $\textsc{Tree}\mathbb{C}_{y'}^{\triangleq r}$ and with output data $(\mathtt{data}, \mathsf{ok})$. Conversely, given a transcript T' compliant with $\textsc{Tree}\mathbb{C}_{y'}^{\triangleq r}$ and with output data $(\mathtt{data}, \mathsf{ok})$, we can efficiently produce a new transcript T compliant with $\mathbb{C}_{y'}$ and with output data ok. Somewhat more precisely:

Lemma 8.4. *There exist efficient functions $\Psi_0, \Psi_1 \colon \{0,1\}^* \to \{0,1\}^*$ such that:*

- *For every transcript T with $\mathbb{C}_{y'}(\mathsf{T})$ and $\mathsf{out}(\mathsf{T}) = \mathsf{ok}$, it holds that $\textsc{Tree}\mathbb{C}_{y'}^{\triangleq r}(\mathsf{T}') = 1$ and $\mathsf{out}(\mathsf{T}') = (\mathtt{data}, \mathsf{ok})$, where $\mathsf{T}' := \Psi_0(y', r, \mathsf{T})$.*
- *For every transcript T' with $\textsc{Tree}\mathbb{C}_{y'}^{\triangleq r}(\mathsf{T}') = 1$ and $\mathsf{out}(\mathsf{T}') = (\mathtt{data}, \mathsf{ok})$, it holds that $\mathbb{C}_{y'}(\mathsf{T})$ and $\mathsf{out}(\mathsf{T}) = \mathsf{ok}$, where $\mathsf{T} := \Psi_1(y', r, \mathsf{T}')$.*

Proof. Let T be any path transcript that is compliant with $\mathbb{C}_{y'}$ having output data ok. By construction of $\mathbb{C}_{y'}$ (see Section 7.2), T is a path with $\tilde{t} + 3$ nodes for some $\tilde{t} \leq t'$. Let the $\tilde{t} + 2$ messages in T be $(0, S_0), \ldots, (\tilde{t}, S_{\tilde{t}}), \mathsf{ok}$ (i.e., all but the last message are timestamped states of the machine M'); let the local inputs in T be $a_1, \ldots, a_{\tilde{t}}, \perp$. Now construct a new transcript T' as follows. (See Figure 6 for a diagram of an example where $r = 2$ and $\tilde{t} = 4$.) First create $\tilde{t} + 2$ source nodes (necessarily labeled with $\perp$), and "above" them create $\tilde{t}$ *leaf* nodes; label the i-th leaf node with a_i for $i = 1, \ldots, \tilde{t} + 1$ and the $(\tilde{t} + 1)$-th leaf node with $\perp$. Then connect the first source node to the first leaf node, the last source node to the last leaf node, and every intermediate source node to the two adjacent leaf nodes. Label the edge going from the first source node to the first leaf node with $(0, 0, S_0)$, the edge going from the last source node to the last leaf node with $(0, \tilde{t} + 1, \mathsf{ok})$, and the two outgoing edges of the i-th source node with $(0, i - 1, S_{i-1})$ for $i = 2, \ldots, \tilde{t} + 1$. We have now constructed the "base" of the tree of T'; we now iteratively construct the rest of the tree by always trying to group sets of r consecutive nodes of the same height together under a parent; when this cannot be done anymore, all the topmost nodes point directly to a root, which itself points to a sink. More precisely, first group every consecutive set of r leaves (leaving any leftover leaves alone) and give a single parent (i.e., first level node) to each set of r leaves; label every edge from a leaf to its parent with $(1, \tau, S, \tau', S')$ where $(0, \tau, S)$ and $(0, \tau', S')$ are the first and second messages into the leaf. Then group every consecutive set r of first-level nodes (leaving any leftover first-level nodes alone) and give a single parent (i.e., second-level node) to each set of r leaves; label every edge from a first-level node to its parent with $(2, \tau_1, S_1, \tau'_r, S'_r)$ where $(1, \tau_1, S_1, \tau'_1, S'_1)$ and $(1, \tau_r, S_r, \tau'_r, S'_r)$ are the first and last messages into the first-level node; proceed in this manner, "merging" timestamp-state pairs of sets of r nodes at the same level, until no more grouping can be performed. Then take all the top-level nodes of the trees of different heights and make them all children of a new "root" node; these edges are again labeled with suitable level numbers and two timestamp-state pairs. Every internal node is labeled with $\perp$. Finally, put an edge with the message $(\mathtt{data}, \mathsf{ok})$ connecting the root to a sink node (necessarily labeled with $\perp$). It is easy to see that T' is compliant with $\textsc{Tree}\mathbb{C}_{y'}^{\triangleq r}$ and indeed has output data $(\mathtt{data}, \mathsf{ok})$. Clearly, this transformation can be performed efficiently by a function Ψ_0.

Conversely, let T' be any transcript that is compliant with respect to $\textsc{Tree}\mathbb{C}_{y'}^{\triangleq r}$ and has output data ok. We show how to "extract" a transcript T compliant with $\mathbb{C}_{y'}$ having output data ok. According to $\textsc{Tree}\mathbb{C}_{y'}^{\triangleq r}$, the only way to obtain the message $(\mathtt{data}, \mathsf{ok})$ is to receive messages $(d_i, \tau_i, S_i, \tau'_i, S'_i)$ with consistent timestamp-state pairs, $\tau_1 = 0$, and $S'_{r'} = \mathsf{ok}$. Again according to $\textsc{Tree}\mathbb{C}_{y'}^{\triangleq r}$, the only way to obtain $(d_i, \tau_i, S_i, \tau'_i, S'_i)$ with $d_i > 1$ is to receive r messages of level $d_i - 1$ that correctly "collapse" to the message; if instead $d_i = 1$, the only way to obtain the message is two receive two messages $(0, \tau, S)$ and $(0, \tau', S')$, consistent with the timestamps and states, such that $\mathbb{C}_{y'}((\tau', S'); ((\tau, S)), \mathsf{linp})$ and $\tau' = \tau + 1$ for some local input linp. Thus, the leaves of T' essentially form a $\mathbb{C}_{y'}$-compliant path transcript that ends with message ok, so we can construct T from T' by taking in order the messages we find at the leaves of the tree T'. Clearly, this transformation can be performed efficiently by a function Ψ_1. □

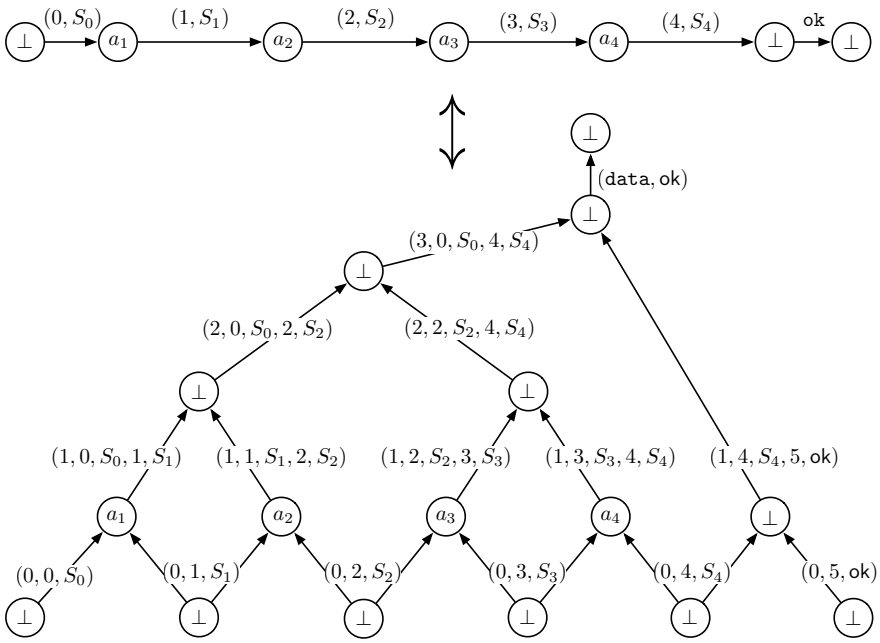

Figure 6: Going from T to T' and vice versa, with in-degree $r = 2$ and a computation with $\tilde{t} = 4$.

Step 2: Construct a new PCD system. Having shown how to construct $\text{TREE}\mathbb{C}_{y'}^{\triangleq r}$ from $\mathbb{C}_{y'}$, we sketch how to construct a PCD system that leverages $\text{TREE}\mathbb{C}_{y'}^{\triangleq r}$. Concretely, given a PCD system $(\mathbb{G}, \mathbb{P}, \mathbb{V})$ for constant-depth compliance predicates, we need to construct a *path* PCD system $(\mathbb{G}', \mathbb{P}', \mathbb{V}')$ for $\mathbb{C}_y^h = \mathbb{C}_{y'}$ (over a random choice of h). Very roughly, the construction is as:

- The new generator $\mathbb{G}'$, on input security parameter 1^k and time bound B, draws h from $\mathcal{H}$, runs the old generator $\mathbb{G}$ on input $(1^k, B')$ to obtain (σ, τ), and then outputs $(\sigma', \tau') := \big((h, \sigma), (h, \tau)\big)$. Intuitively, B' has to be larger than B to ensure that the computation in $\text{TREE}\mathbb{C}_{y'}^{\triangleq r}$ in addition to computation of $\mathbb{C}_{y'}$ (e.g., evaluations of h, consistency comparisons, and so on) can fit within the time bound B'. So suppose that evaluating $\mathbb{C}_{y'}$ at any node of a distributed computation transcript T takes time at most B; then, evaluating $\text{TREE}\mathbb{C}_{y'}^{\triangleq r}$ at any node of a corresponding distributed computation transcript T' (obtained following the proof of Lemma 8.4) takes time $\text{poly}(k + r + B)$. Thus picking $B' = \text{poly}(k + r + B)$ for some poly that only depends on $\mathcal{H}$ suffices.

- The new prover $\mathbb{P}'$, given reference string σ', output data $\mathsf{z_o}$, local input linp, input data $\mathsf{z_i}$ and proof π_{i}, proceeds as follows. First it parses σ' as (h, σ) and uses h to construct $\text{TREE}\mathbb{C}_{y'}^{\triangleq r}$. Then parses π_{i} as $(i, \vec{\mathsf{z}}_1, \ldots, \vec{\mathsf{z}}_D, \vec{\pi}_1, \ldots, \vec{\pi}_D)$, where i is a counter indicating how many nodes have computed on the path already, and the remaining vectors are data and proofs corresponding to a "vertical slice" of a virtual tree on top of the computation path so far. Given this information and using σ, $\mathbb{P}'$ invokes $\mathbb{P}_{\text{TREE}\mathbb{C}_{y'}^{\triangleq r}}$ to first create a proof for the current node (which should be interpreted as a new leaf added to the tree), and then, potentially, invoke $\mathbb{P}_{\text{TREE}\mathbb{C}_{y'}^{\triangleq r}}$ additional times to merge r nodes at the same level of the tree, until there are no such nodes left. Having produced all these proofs, $\mathbb{P}'$ updates the information in $\vec{\mathsf{z}}_1, \ldots, \vec{\mathsf{z}}_D$ and $\vec{\pi}_1, \ldots, \vec{\pi}_D$, and then outputs $(i + 1, \vec{\mathsf{z}}_1, \ldots, \vec{\mathsf{z}}_D, \vec{\pi}_1, \ldots, \vec{\pi}_D)$. In sum, the "real" prover $\mathbb{P}'$ is simulating in his mind many "virtual" provers $\mathbb{P}_{\text{TREE}\mathbb{C}_{y'}^{\triangleq r}}$ that maintain a distributed computation over a growing tree.

- The new verifier $\mathbb{V}'$, given verification state τ', data z, and proof π, proceeds as follows. First it parses τ' as (h, τ) and uses h to construct $\text{TREE}\mathbb{C}_{y'}^{\triangleq r}$. Then uses τ to invoke $\mathbb{V}_{\text{TREE}\mathbb{C}_{y'}^{\triangleq r}}$ on z and the appropriate subproof of π.

The above description is especially sketchy because for now we are avoiding the delicate issue of which subproof the verifier should actually verify. We deal with this issue, and tackle other issues that do not arise in the case of the compliance predicate $\mathbb{C}_{y'}$, in the general case, described in full details in the next subsection.

8.2 General Case

The compliance predicate $\mathbb{C}_{y'}$ is very specific: it is the RAM Checker of a $\text{poly}(k)$-space random access machine. In Section 8.1 we explained how to convert $\mathbb{C}_{y'}$ into a "semantically-equivalent" compliance predicate $\text{TREE}\mathbb{C}_{y'}^{\triangleq r}$ of much smaller depth, and then sketched how to construct a PCD system for $\mathbb{C}_{y'}$ by using a path PCD system for $\text{TREE}\mathbb{C}_{y'}^{\triangleq r}$. In this section we generalize the ideas of Section 8.1 to *any* (polynomial-depth) compliance predicate $\mathbb{C}$. We again proceed in two steps:

1. First, we show how to transform *any* compliance predicate $\mathbb{C}$ to a "tree" version $\text{TREE}_{\mathbb{C}}$ with much smaller depth. To make this work in the general case we need to be more careful because the data in the distributed computation may not be small. (In the case of $\mathbb{C}_{y'}$, the data was of length $\text{poly}(k + |y'|)$.) Thus, instead of comparing this data as we go up the tree, we compare hashes of data. Furthermore, we also need to properly handle every potential output of the distributed computation, while in Section 8.1 we only showed how to handle the output ok of $\mathbb{C}_{y'}$.

2. Second, we construct a *path* PCD system $(\mathbb{G}', \mathbb{P}', \mathbb{V}')$ for any polynomial-depth $\mathbb{C}$. As before, the idea is to map $\mathbb{C}$ to $\text{TREE}_{\mathbb{C}}$, which has constant depth, and use a PCD system for constant-depth compliance predicates to enforce $\text{TREE}_{\mathbb{C}}$. In Section 8.1 we only sketched the construction for the special case; here we shall give all the details for the general case.

Details follow.

Step 1: Engineer a new compliance predicate. We start again by giving the mapping from $\mathbb{C}$ to $\text{TREE}_{\mathbb{C}}$; this construction will be quite similar to the one we gave in Construction 8.2, except that, as already mentioned, we will be comparing *hashes* of data when going up the tree, rather than the original data itself.

Construction 8.5. Let $\mathcal{H}$ be a collision-resistant hash function family. For any compliance predicate $\mathbb{C}$, $h \in \mathcal{H}$, and $r \in \mathbb{N}$, define the following compliance predicate:

$\text{TREE}_{\mathbb{C}}^{h,\triangleq r}(\mathsf{z_o}; \vec{\mathsf{z}}_\mathsf{i}, \mathsf{linp}) \stackrel{\text{def}}{=}$

1. Input State
   If $\vec{\mathsf{z}}_\mathsf{i} = \bot$ and $\mathsf{linp} = \bot$:
   (a) Verify that $\mathsf{z_o}$ equals $(0, \tau, \mathsf{z})$ for some τ, z.
   (b) If $\tau = 0$, verify that $\mathbb{C}(\mathsf{z}; \bot, \bot)$ accepts.
2. Leaf Node Stage
   If $\mathsf{z_o} = (1, \tau, \rho, \tau', \rho')$ and $\vec{\mathsf{z}}_\mathsf{i} = \big((0, \tau_1, \mathsf{z}_1), (0, \tau_2, \mathsf{z}_2)\big)$ for some $\tau, \rho, \tau', \rho', \tau_1, \mathsf{z}_1, \tau_2, \mathsf{z}_2$:
   (a) Verify that $\tau_2 = \tau_1 + 1$, $\tau = \tau_1$, and $\tau' = \tau_2$.
   (b) Verify that $\rho = h(\mathsf{z}_1)$ and $\rho' = h(\mathsf{z}_2)$.

(c) Verify that $\mathbb{C}(\mathsf{z}_2; (\mathsf{z}_1), \mathsf{linp})$ accepts.

3. Internal Node Stage

   If $\mathsf{z}_\mathsf{o} = (d+1, \tau, \rho, \tau', \rho')$ and $\vec{\mathsf{z}}_\mathsf{i} = \left((d, \tau_i, \rho_i, \tau'_i, \rho'_i)\right)_{i=1}^{r}$ for some $\tau, \rho, \tau', \rho', \tau_1, \ldots, \tau_r, \mathsf{z}_1, \ldots, \mathsf{z}_r$:
   (a) Verify that $\tau = \tau_1$, $\tau'_1 = \tau_2$, $\tau'_2 = \tau_3$, and so on until $\tau'_{r-1} = \tau_r$, $\tau'_r = \tau'$.
   (b) Verify that $\rho = \rho_1$, $\rho'_1 = \rho_2$, $\rho'_2 = \rho_3$, and so on until $\rho'_{r-1} = \rho_r$, $\rho'_r = \rho'$.

4. Output Stage

   If $\mathsf{z}_\mathsf{o} = (\mathtt{data}, \mathsf{z})$ and $\vec{\mathsf{z}}_\mathsf{i} = \left((d_i, \tau_i, \rho_i, \tau'_i, \rho'_i)\right)_{i=1}^{r'}$ for some $\mathsf{z}, r', d_1, \ldots, d_{r'}, \tau_1, \ldots, \tau_{r'}, \mathsf{z}_1, \ldots, \mathsf{z}_{r'}$:
   (a) Verify that $\tau'_1 = \tau_2$, $\tau'_2 = \tau_3$, and so on until $\tau'_{r'-1} = \tau_{r'}$.
   (b) Verify that $\rho'_1 = \rho_2$, $\rho'_2 = \rho_3$, and so on until $\rho'_{r'-1} = \rho_{r'}$.
   (c) Verify that $\tau_1 = 0$ and $\rho'_{r'} = h(\mathsf{z})$.

5. If none of the above conditions hold, reject.

As promised, the depth of the new compliance predicate $\textsc{Tree}_{\mathbb{C}}^{h,\triangleq r}$ is much better than that of $\mathbb{C}$:

Lemma 8.6. *For any compliance predicate $\mathbb{C}$, $h \in \mathcal{H}$, and $r \in \mathbb{N}$,*

$$\mathsf{d}\left(\textsc{Tree}_{\mathbb{C}}^{h,\triangleq r}\right) \leq \lfloor \log_r \mathsf{d}(\mathbb{C}) \rfloor + 1 \ .$$

Proof. Any transcript compliant with $\textsc{Tree}_{\mathbb{C}}^{h,\triangleq r}$ consists of disjoint trees. In each such tree, nodes of different heights are forced to directly point to the root of the tree, and other nodes of the same height are always grouped in sets of size r. Thus, the "worst possible height", given that any tree can have at most $\mathsf{d}(\mathbb{C})$ leaves, is given by $\lfloor \log_r \mathsf{d}(\mathbb{C}) \rfloor + 1$ (achieved by making maximal use of merging nodes of the same height). □

As in Section 8.1, the depth reduction is meaningful because we can accompany it with guarantees that ensure that even if we switch to the new compliance predicate $\textsc{Tree}_{\mathbb{C}}^{h,\triangleq r}$, the "semantics" of the compliance predicate are preserved. Namely, given a transcript T compliant with $\mathbb{C}$ and with output data z_o, we can efficiently produce a new transcript T' compliant with $\textsc{Tree}_{\mathbb{C}}^{h,\triangleq r}$ and with output data $(\mathtt{data}, \mathsf{z}_\mathsf{o})$. Conversely, given a transcript T' compliant with $\textsc{Tree}_{\mathbb{C}}^{h,\triangleq r}$ and with output data $(\mathtt{data}, \mathsf{z}_\mathsf{o})$, we can efficiently produce a new transcript T compliant with $\mathbb{C}$ and with output data z_o. More precisely, the reverse direction holds provided that T' is produced by an *efficient* adversary $\mathcal{A}$ (when given as input $(\mathbb{C}, h, r)$ for a random h), because the guarantee relies on the adversary not being able to find collisions in h.

Lemma 8.7. *There exist efficient functions $\Psi_0, \Psi_1 \colon \{0,1\}^* \to \{0,1\}^*$ such that for every compliance predicate $\mathbb{C}$ and in-degree parameter $r \in \mathbb{N}$:*

- *For every $h \in \mathcal{H}$, output data z_o, and path transcript T with $\mathbb{C}(\mathsf{T})$ and $\mathsf{out}(\mathsf{T}) = \mathsf{z}_\mathsf{o}$, it holds that $\textsc{Tree}_{\mathbb{C}}^{h,\triangleq r}(\mathsf{T}') = 1$ and $\mathsf{out}(\mathsf{T}') = (\mathtt{data}, \mathsf{z}_\mathsf{o})$, where $\mathsf{T}' := \Psi_0(\mathbb{C}, h, r, \mathsf{T})$.*
- *For every polynomial-size adversary $\mathcal{A}$ and sufficiently large $k \in \mathbb{N}$,*

$$\Pr\left[\begin{array}{c} \textsc{Tree}_{\mathbb{C}}^{h,\triangleq r}(\mathsf{T}') = 1 \\ \mathbb{C}(\mathsf{T}) \neq 1 \ \vee \ \mathsf{out}(\mathsf{T}) \neq \mathsf{z}_\mathsf{o} \end{array} \middle| \begin{array}{r} h \leftarrow \mathcal{H}_k \\ \mathsf{T}' \leftarrow \mathcal{A}(\mathbb{C}, h, r) \\ \mathsf{T} \leftarrow \Psi_1(\mathbb{C}, h, r, \mathsf{T}') \\ (\mathtt{data}, \mathsf{z}_\mathsf{o}) \leftarrow \mathsf{out}(\mathsf{T}') \end{array}\right] \leq \mathrm{negl}(k) \ .$$

Proof. Let T be any path transcript that is compliant with $\mathbb{C}$; T is a path with $\tilde{\mathsf{d}}+2$ nodes for some $\tilde{\mathsf{d}} \leq \mathsf{d}(\mathbb{C})$. Let the messages in T be $\mathsf{z}_0, \ldots, \mathsf{z}_{\tilde{\mathsf{d}}}$; in particular, the output message $\mathsf{z}_\mathsf{o} := \mathsf{out}(\mathsf{T})$ of T is equal to $\mathsf{z}_{\tilde{\mathsf{d}}}$; let the local inputs in T be $\mathsf{linp}_1, \ldots, \mathsf{linp}_{\tilde{\mathsf{d}}}$. Now construct a new transcript T' as follows. (See Figure 7 for a diagram of an example where $r = 2$ and $\tilde{\mathsf{d}} = 5$.) First create $\tilde{\mathsf{d}} + 1$ source nodes (necessarily labeled with $\perp$), and "above" them create $\tilde{\mathsf{d}}$ *leaf* nodes; label the i-th leaf node with linp_i for $i = 1, \ldots, \tilde{\mathsf{d}}$. Then connect the first source node to the first leaf node, the last source node to the last leaf node, and every intermediate source node to the two adjacent leaf nodes. Label the edge going from the first source node to the first leaf node with $(0, 0, \mathsf{z}_0)$, the edge going from the last source node to the last leaf node with $(0, \tilde{\mathsf{d}}, \mathsf{z}_{\tilde{\mathsf{d}}})$, and the two outgoing edges of the i-th source node with $(0, i-1, \mathsf{z}_{i-1})$ for $i = 2, \ldots, \tilde{\mathsf{d}}$. We have now constructed the "base" of the tree of T'; we now iteratively construct the rest of the tree by always trying to group sets of r consecutive nodes of the same height together under a parent; when this cannot be done anymore, all the topmost nodes point directly to a root, which itself points to a sink. More precisely, first group every consecutive set of r leaves (leaving any leftover leaves alone) and give a single parent (i.e., first level node) to each set of r leaves; label every edge from a leaf to its parent with $(1, \tau, h(\mathsf{z}), \tau', h(\mathsf{z}'))$ where $(0, \tau, \mathsf{z})$ and $(0, \tau', \mathsf{z}')$ are the first and second messages into the leaf. Then group every consecutive set r of first-level nodes (leaving any leftover first-level nodes alone) and give a single parent (i.e., second-level node) to each set of r leaves; label every edge from a first-level node to its parent with $(2, \tau_1, \rho_1, \tau'_r, \rho'_r)$ where $(1, \tau_1, \rho_1, \tau'_1, \rho'_1)$ and $(1, \tau_r, \rho_r, \tau'_r, \rho'_r)$ are the first and last messages into the first-level node; proceed in this manner, "merging" timestamp-hash pairs of sets of r nodes at the same level, until no more grouping can be performed. Then take all the top-level nodes of the trees of different heights and make them all children of a new "root" node; these edges are again labeled with suitable level numbers and two timestamp-hash pairs. Every internal node is labeled with $\perp$. Finally, put an edge with the message $(\mathtt{data}, \mathsf{z}_{\tilde{\mathsf{d}}})$ connecting the root to a sink node (necessarily labeled with $\perp$). It is easy to see that T' is compliant with $\textsc{Tree}_{\mathbb{C}}^{h,\triangleq r}$ and indeed has output data $(\mathtt{data}, \mathsf{z}_{\tilde{\mathsf{d}}})$. Clearly, this transformation can be performed efficiently by a function Ψ_0.

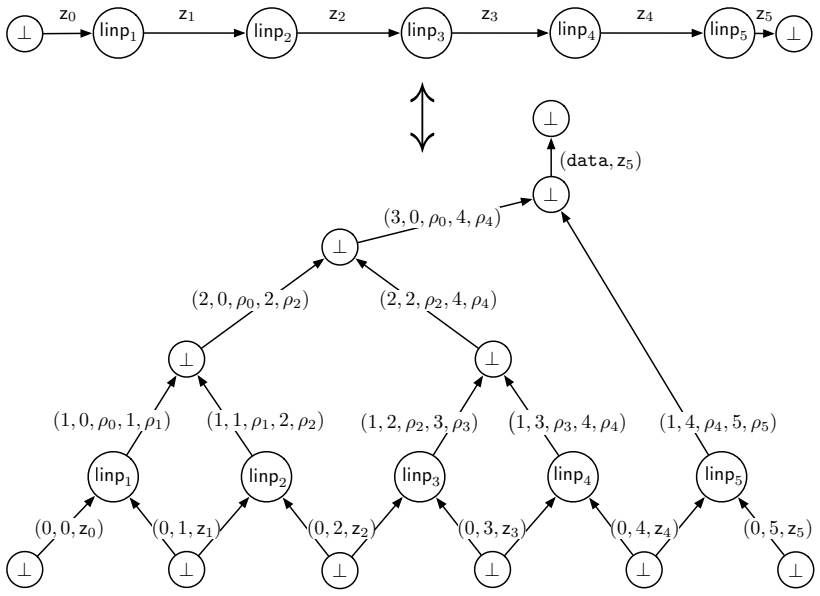

Figure 7: Going from T to T' and vice versa, with in-degree $r = 2$ and a computation with $\tilde{\mathsf{d}} = 5$. Here $\rho_i = h(\mathsf{z}_i)$ for $i = 1, \ldots, 5$.

With all but negligible probability in k over a random choice of h in $\mathcal{H}_k$, on input $(\mathbb{C}, h, r)$, the adversary

$\mathcal{A}$ does not find any collisions for h. Conditioned on $\mathcal{A}$ not having found any collisions and outputting a transcript T' compliant with $\text{TREE}_{\mathbb{C}}^{h,\triangleq r}$ having output data $(\mathtt{data}, \mathsf{z_o})$, we show how to "extract" a transcript T compliant with $\mathbb{C}$ having output data $\mathsf{z_o}$. According to $\text{TREE}_{\mathbb{C}}^{h,\triangleq r}$, the only way to obtain the message $(\mathtt{data}, \mathsf{z_o})$ is to receive messages $(d_i, \tau_i, \rho_i, \tau_i', \rho_i')$ with consistent timestamp-hash pairs, $\tau_1 = 0$, and $\rho_{r'}' = h(\mathsf{z_o})$. Again according to $\text{TREE}_{\mathbb{C}}^{h,\triangleq r}$, the only way to obtain $(d_i, \tau_i, \rho_i, \tau_i', \rho_i')$ with $d_i > 1$ is to receive r messages of level $d_i - 1$ that correctly "collapse" to the message; if instead $d_i = 1$, the only way to obtain the message is two receive two messages $(0, \tau, \mathsf{z})$ and $(0, \tau', \mathsf{z}')$, consistent with the timestamps and hashes, such that $\mathbb{C}(\mathsf{z}'; \mathsf{z}, \mathsf{linp})$ and $\tau' = \tau + 1$ for some local input linp. Thus, the leaves of T' essentially form a $\mathbb{C}$-compliant path transcript that ends with message $\mathsf{z_o}$, so we can construct T from T' by taking in order the messages we find at the leaves of the tree T'. Clearly, this transformation can be performed efficiently by a function Ψ_1. □

Step 2: Construct a new PCD system. Having shown how to construct $\text{TREE}_{\mathbb{C}}^{h,\triangleq r}$ from $\mathbb{C}$, we need to construct a PCD system that leverages $\text{TREE}_{\mathbb{C}}^{h,\triangleq r}$. Concretely, given a PCD system $(\mathbb{G}, \mathbb{P}, \mathbb{V})$ for constant-depth compliance predicates, we explain how to construct a *path* PCD system $(\mathbb{G}', \mathbb{P}', \mathbb{V}')$ for polynomial-depth compliance predicates. The high-level idea is as follows.

- The new generator $\mathbb{G}'$, on input security parameter 1^k and time bound B, draws h from $\mathcal{H}$, runs the old generator $\mathbb{G}$ on input $(1^k, B')$ to obtain (σ, τ), and then outputs $(\sigma', \tau') := \big((h, \sigma), (h, \tau)\big)$. As explained in Section 8.1, B' has to be larger than B, and picking $B' = \text{poly}(k + r + B)$ for some poly that only depends on $\mathcal{H}$ suffices.

- Given a compliance predicate $\mathbb{C}$, the new prover $\mathbb{P}_{\mathbb{C}}'$, given reference string σ', output data $\mathsf{z_o}$, local input linp, input data $\mathsf{z_i}$ and proof π_i, proceeds as follows. First it parses σ' as (h, σ) and uses h to construct $\text{TREE}_{\mathbb{C}}^{h,\triangleq r}$. Then it uses $\mathbb{P}_{\text{TREE}_{\mathbb{C}}^{h,\triangleq r}}$ to generate a new leaf message and proof. Then it parses π_i as a vector of proofs, each corresponding to a tree root, and again uses $\mathbb{P}_{\text{TREE}_{\mathbb{C}}^{h,\triangleq r}}$ to "merge" groups of r message-proof pairs corresponding to the same level of the tree, until there are no such groups to be found. Essentially, $\mathbb{P}_{\mathbb{C}}'$ is using $\mathbb{P}_{\text{TREE}_{\mathbb{C}}^{h,\triangleq r}}$ to dynamically maintain a "vertical slice" of a tree-like distributed computation compliance with $\text{TREE}_{\mathbb{C}}^{h,\triangleq r}$, arising from a path distributed computation compliant with $\mathbb{C}$.

- Given a compliance predicate $\mathbb{C}$, the new verifier $\mathbb{V}_{\mathbb{C}}'$, given verification state τ', data z, and proof π, proceeds as follows. First it parses τ' as (h, τ) and uses h to construct $\text{TREE}_{\mathbb{C}}^{h,\triangleq r}$. Then uses τ to invoke $\mathbb{V}_{\text{TREE}_{\mathbb{C}}^{h,\triangleq r}}$ on z and the appropriate subproof of π.

The above sketch leaves out many details; see Figure 8 for a detailed construction.

Ingredients. A PCD system $(\mathbb{G}, \mathbb{P}, \mathbb{V})$ for constant-depth compliance predicates and a collision-resistant hash-function family $\mathcal{H}$. In the construction, one should take the in-degree parameter r to equal k.

Output. A *path* PCD system $(\mathbb{G}', \mathbb{P}', \mathbb{V}')$ for polynomial-depth compliance predicates. (In particular, $\mathbb{P}'$ expects only a single proof-carrying message.)

The new generator $\mathbb{G}'$. Given security parameter 1^k and time bound B, $\mathbb{G}'$ proceeds as follows:

1. $h \leftarrow \mathcal{H}_k$;
2. $(\sigma, \tau) \leftarrow \mathbb{G}(1^k, \mathrm{poly}(k + r + B))$, where poly only depends on $\mathcal{H}$;
3. $\sigma' := (h, \sigma)$;
4. $\tau' := (h, \tau)$;
5. output (σ', τ').

The new prover $\mathbb{P}'$. Given a polynomial-depth compliance predicate $\mathbb{C}$, reference string σ', output data $\mathsf{z_o}$, local input linp, input data $\mathsf{z_i}$ and proof π_{i}, $\mathbb{P}'_{\mathbb{C}}$ proceeds as follows:

1. parse σ' as (h, σ) and construct $\textsc{Tree}_{\mathbb{C}}^{h,\triangleq r}$ (see Construction 8.5);
2. parse π_{i} as $(\pi_{\mathsf{all}}, i, \vec{\mathsf{z}}_1, \ldots, \vec{\mathsf{z}}_D, \vec{\pi}_1, \ldots, \vec{\pi}_D)$;
3. set $\mathsf{z}'_{\mathsf{i}} := (0, i, \mathsf{z_i})$ and compute $\pi_{0,i} \leftarrow \mathbb{P}_{\textsc{Tree}_{\mathbb{C}}^{h,\triangleq r}}(\sigma, \mathsf{z}'_{\mathsf{i}}, \bot, \bot, \bot)$;
4. set $\mathsf{z}'_{\mathsf{o}} := (0, i+1, \mathsf{z_o})$ and compute $\pi_{0,i+1} \leftarrow \mathbb{P}_{\textsc{Tree}_{\mathbb{C}}^{h,\triangleq r}}(\sigma, \mathsf{z}'_{\mathsf{o}}, \bot, \bot, \bot)$;
5. $\rho_i \leftarrow h(\mathsf{z_i})$;
6. $\rho_{i+1} \leftarrow h(\mathsf{z_o})$;
7. $\pi_{1,i+1} \leftarrow \mathbb{P}_{\textsc{Tree}_{\mathbb{C}}^{h,\triangleq r}}(\sigma, (1, i, \rho_i, i+1, \rho_{i+1}), \mathsf{linp}, (\mathsf{z}'_{\mathsf{i}}, \mathsf{z}'_{\mathsf{o}}))$;
8. add an extra coordinate to the end of $\vec{\mathsf{z}}_1$ and set it to $(1, i, \rho_i, i+1, \rho_{i+1})$;
9. add an extra coordinate to the end of $\vec{\pi}_1$ and set it to $\pi_{1,i+1}$;
10. for $d = 1, \ldots, D$ (in this order), if there are r coordinates in $\vec{\mathsf{z}}_d$ then:
    (a) parse $\vec{\mathsf{z}}_d$ as $\big((d, \tau_j, \rho_j, \tau'_j, \rho'_j)\big)_{j=1}^{r}$;
    (b) $\pi_{d+1,i+1} \leftarrow \mathbb{P}_{\textsc{Tree}_{\mathbb{C}}^{h,\triangleq r}}(\sigma, (d+1, \tau_1, \rho_1, \tau'_d, \rho'_d), \bot, \vec{\mathsf{z}}_d, \vec{\pi}_d)$;
    (c) set $\vec{\mathsf{z}}_d$ and $\vec{\pi}_d$ to be the vector with zero coordinates;
    (d) add an extra coordinate to the end of $\vec{\mathsf{z}}_{d+1}$ and set it to $(d+1, \tau_1, \rho_1, \tau'_d, \rho'_d)$;
    (e) add an extra coordinate to the end of $\vec{\pi}_{d+1}$ and set it to $\pi_{d+1,i+1}$;
11. $\pi_{\mathsf{all}} \leftarrow \mathbb{P}_{\textsc{Tree}_{\mathbb{C}}^{h,\triangleq r}}(\sigma, \mathsf{z_o}, \bot, \vec{\mathsf{z}}_1 \circ \cdots \circ \vec{\mathsf{z}}_D, \vec{\pi}_1 \circ \cdots \circ \vec{\pi}_D)$;
12. output $(\pi_{\mathsf{all}}, i+1, \vec{\mathsf{z}}_1, \ldots, \vec{\mathsf{z}}_D, \vec{\pi}_1, \ldots, \vec{\pi}_D)$.

The new verifier $\mathbb{V}'$. Given a polynomial-depth compliance predicate $\mathbb{C}$, verification state τ', data z, and proof π, $\mathbb{V}'_{\mathbb{C}}$ proceeds as follows:

1. parse τ' as (h, τ) and construct $\textsc{Tree}_{\mathbb{C}}^{h,\triangleq r}$ (see Construction 8.5);
2. parse π_{i} as $(\pi_{\mathsf{all}}, i, \vec{\mathsf{z}}_1, \ldots, \vec{\mathsf{z}}_D, \vec{\pi}_1, \ldots, \vec{\pi}_D)$;
3. $b \leftarrow \mathbb{V}_{\textsc{Tree}_{\mathbb{C}}^{h,\triangleq r}}(\tau, \mathsf{z}, \pi_{\mathsf{all}})$;
4. output b.

Figure 8: The transformation $\textsc{DepthRed}_{\mathcal{H}}$, which constructs $(\mathbb{G}', \mathbb{P}', \mathbb{V}')$ from $(\mathbb{G}, \mathbb{P}, \mathbb{V})$.

Remark 8.8 (depth reduction beyond paths)**.** Focusing on paths yields the simplest example of a PCD Depth-Reduction Theorem. We could modify the mapping from $\mathbb{C}$ to $\textsc{Tree}_{\mathbb{C}}$, as well as the corresponding construction of $(\mathbb{G}', \mathbb{P}', \mathbb{V}')$, to also support distributed computations that evolve over graphs that are not just paths. For example, we could have a PCD Depth-Reduction Theorem for graphs that have the shape of a "Y" instead of for paths, by building a wide Merkle tree independently on each of the three segments of the "Y". More generally, the PCD Depth-Reduction Theorem works at least for graphs satisfying a certain property that we now formulate. Let G be a directed acyclic graph with a single sink s; for a vertex v in G, define $\phi(v) := 0$ if v is a source and $\phi(v) := (\deg(v) - 1) + \sum_{p \text{ parent of } v} \phi(p)$ otherwise; then define

$\Phi(G) := \phi(s)$. Essentially, $\Phi(G)$ measures how "interactive" is the graph G when viewed as a distributed computation; see Figure 9 for examples. Having defined this measure of interactivity, one can verify that the PCD Depth-Reduction Theorem holds for all graphs G for which $\Phi(G)$ is a fixed polynomial in the security parameter k: namely, assuming that collision-resistant hash functions exist, any PCD system for constant-depth compliance predicates can be efficiently transformed into a corresponding "$\mathcal{C}$-graph PCD system" for polynomial-depth compliance predicates, where $\mathcal{C}$ is the class of graphs G for which $\Phi(G) = \mathrm{poly}(k)$. (And, as in the basic case, the verifiability properties carry over, as do efficiency properties.)

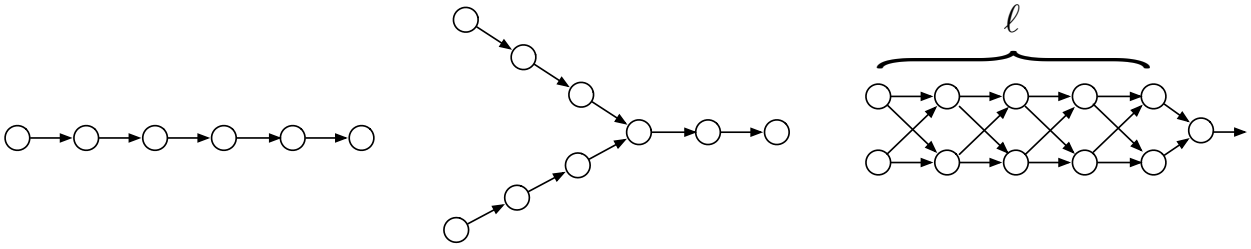

Figure 9: For the path graph, $\Phi = 0$; for the "Y" graph, $\Phi = 1$; for the "braid" graph, $\Phi = 2^{\ell+1} - 1$. The first two graphs are not very "interactive", whereas the last one is.

9 Putting Things Together

In Section 2.5 we explained at high level how our three main tools can be combined to obtain our main theorem. In Sections 6, 7, and 8, we have provided details for each of our three tools; we now provide additional details for how these tools come together to obtain our main theorem.

Theorem 9.1 (Main Theorem (Theorem 4 restated))**.** *Let $\mathcal{H}$ be a collision-resistant hash-function family.*

1. ***Complexity-Preserving SNARK from any SNARK.*** *There is an efficient transformation $\mathbb{T}_{\mathcal{H}}$ such that for any publicly-verifiable SNARK $(\mathcal{G}, \mathcal{P}, \mathcal{V})$ there is a polynomial p for which $(\mathcal{G}^\star, \mathcal{P}^\star, \mathcal{V}^\star) := \mathbb{T}_{\mathcal{H}}(\mathcal{G}, \mathcal{P}, \mathcal{V})$ is a publicly-verifiable SNARK that is complexity-preserving with a polynomial p (see Definition 4.2), i.e.,*
   - *the generator $\mathcal{G}^\star(1^k)$ runs in time $p(k)$ (in particular, there is no expensive preprocessing);*
   - *the prover $\mathcal{P}^\star(\sigma, (M, x, t), w)$ runs in time $(|M| + |x| + t) \cdot p(k)$ and space $(|M| + |x| + s) \cdot p(k)$ when proving that a t-time s-space NP random-access machine M accepts (x, w);*
   - *the verifier $\mathcal{V}^\star(\tau, (M, x, t), \pi)$ runs in time $(|M| + |x|) \cdot p(k)$.*
2. ***Complexity-Preserving PCD from any SNARK.*** *There is an efficient transformation $\mathbb{T}'_{\mathcal{H}}$ such that for any publicly-verifiable SNARK $(\mathcal{G}, \mathcal{P}, \mathcal{V})$ there is a polynomial p for which $(\mathbb{G}^\star, \mathbb{P}^\star, \mathbb{V}^\star) := \mathbb{T}'_{\mathcal{H}}(\mathcal{G}, \mathcal{P}, \mathcal{V})$ is a publicly-verifiable PCD for constant-depth compliance predicates that is complexity-preserving with polynomial p (see Definition 5.12), i.e., for every constant-depth compliance predicate $\mathbb{C}$,*
   - *the generator $\mathbb{G}^\star(1^k)$ runs in time $p(k)$;*
   - *the prover $\mathbb{P}^\star_{\mathbb{C}}(\sigma, \mathsf{z_o}, \mathsf{linp}, \vec{\mathsf{z}}_\mathsf{i}, \vec{\pi}_\mathsf{i})$ runs in time $(|\mathbb{C}| + t) \cdot p(k)$ and space $(|\mathbb{C}| + s) \cdot p(k)$ when proving that a message $\mathsf{z_o}$ is $\mathbb{C}$-compliant, using local input linp and received inputs $\vec{\mathsf{z}}_\mathsf{i}$, and evaluating $\mathbb{C}(\mathsf{z_o}; \mathsf{linp}, \vec{\mathsf{z}}_\mathsf{i})$ takes time t and space s;*
   - *the verifier $\mathbb{V}^\star_{\mathbb{C}}(\tau, \mathsf{z}, \pi)$ runs in time $(|\mathbb{C}| + |\mathsf{z}|) \cdot p(k)$.*

Assuming a fully-homomorphic encryption scheme $\mathcal{E}$, there exist analogous transformations $\mathbb{T}_{\mathcal{H},\mathcal{E}}$ and $\mathbb{T}'_{\mathcal{H},\mathcal{E}}$ for the designated-verifier case.

Proof. Let $(\mathcal{G},\mathcal{P},\mathcal{V})$ be *any* SNARK, and assume (for the worst) that it is a preprocessing SNARK. In particular, there are (potentially large) polynomials p and q such that the following holds. The generator $\mathcal{G}(1^k, B)$ runs in time $p(B+k)$, and produces a reference string and verification state that allow proving and verifying statements $y = (M, x, t)$ with $t \leq B$. The prover $\mathcal{P}(\sigma, (M, x, t), w)$ runs in time $p(|M| + |x| + B + k)$ and space $q(|M| + |x| + B + k)$. The verifier $\mathcal{V}(\tau, (M, x, t), \pi)$ runs in time $p(|y| + k)$.

We invoke the SNARK Recursive Composition Theorem on $(\mathcal{G},\mathcal{P},\mathcal{V})$ to obtain a corresponding PCD system $(\mathbb{G},\mathbb{P},\mathbb{V})$ for constant-depth compliance predicates, and then the PCD Depth-Reduction Theorem to obtain a corresponding path PCD system $(\mathbb{G}',\mathbb{P}',\mathbb{V}')$ for polynomial-depth compliance predicates.

The efficiency of the PCD system $(\mathbb{G}',\mathbb{P}',\mathbb{V}')$ is comparable to that of the SNARK $(\mathcal{G},\mathcal{P},\mathcal{V})$ we started with. In other words, there is an "overhead polynomial" p' such that the following holds. The PCD generator $\mathbb{G}'(1^k, B)$ runs in time $p(B+k)\cdot p'(k)$, and produces a reference string and verification state that only for for B-bounded (path) distributed computations (see Definition 5.6): namely, they allow proving and verifying compliance of path distributed computations where computing $\mathbb{C}$ at each node's output takes time $t \leq B$. The PCD prover $\mathbb{P}'_{\mathbb{C}}(\sigma, \mathsf{z_o}, \mathsf{linp}, \vec{\mathsf{z}}_\mathsf{i}, \vec{\pi}_\mathsf{i})$ runs in time $p(|\mathbb{C}|+B+k)\cdot p'(k)$ and space $q(|\mathbb{C}|+B+k)\cdot p'(k)$. The PCD verifier $\mathbb{V}'_{\mathbb{C}}(\tau, \mathsf{z}, \pi)$ runs in time $p(|\mathbb{C}| + |\mathsf{z}| + k) \cdot p'(k)$. In addition, if $(\mathcal{G},\mathcal{P},\mathcal{V})$ is publicly-verifiable then so is $(\mathbb{G}',\mathbb{P}',\mathbb{V}')$; if $(\mathcal{G},\mathcal{P},\mathcal{V})$ is designated-verifier then so is $(\mathbb{G}',\mathbb{P}',\mathbb{V}')$.

Given the PCD system $(\mathbb{G}',\mathbb{P}',\mathbb{V}')$, we construct a complexity-preserving SNARK $(\mathcal{G}^\star,\mathcal{P}^\star,\mathcal{V}^\star)$ as follows. The new generator $\mathcal{G}^\star$, given input 1^k, outputs $(\sigma',\tau') := ((h,\sigma),(h,\tau))$, where $h \leftarrow \mathcal{H}_k$, $(\sigma,\tau) \leftarrow \mathbb{G}'(1^k, k^c)$, and c is a constant that only depends on $\mathcal{H}$ (see below). The new prover $\mathcal{P}^\star$, given a reference string $\sigma' = (h,\sigma)$, instance $y = (M, x, t)$, and a witness w, computes the compliance predicate $\mathbb{C}^h_y$ given by the Locally-Efficient RAM Compliance Theorem and, using the prover $\mathbb{P}'$, computes a proof for each message in the path distributed computation obtained from (M, x, t) and w (each time using the previous proof); it outputs the final such proof as the SNARK proof. The time required to compute $\mathbb{C}^h_y$ at any node is only $\mathrm{poly}(k + |y|)$ where poly only depends on $\mathcal{H}$. We can assume, without loss of generality, that $|M|$ and $|x|$ are bounded by a fixed $\mathrm{poly}(k)$. (If that is not the case (e.g., M encodes a large non-uniform circuit), $\mathcal{P}^\star$ can work with a new instance $(U_h, \tilde{x}, \mathrm{poly}(k) + t)$, where U_h is a universal random-access machine that, on input $(\tilde{x}, \tilde{w})$, parses $\tilde{w}$ as (M, x, t, w), verifies that $\tilde{x} = h(M, x, t)$, and then runs $M(x, w)$ for at most t steps.) Thus, $\mathrm{poly}(k + |y|) = k^c$ for a constant c that only depends on $\mathcal{H}$; k^c determines the "preprocessing budget" chosen above in the construction of $\mathcal{G}^\star$. Finally, the new verifier $\mathcal{V}^\star$ similarly deduces $\mathbb{C}^h_y$ and uses $\mathbb{V}'$ to verify the proof.

Recall that, when applying the Locally-Efficient RAM Compliance Theorem, the messages and local inputs for the path distributed computation are computed from (M, x, t) and w on-the-fly, one node a time in topological order, using the same time and space as M does (up to a fixed $\mathrm{poly}(k)$ factor). Thus overall, we have "localized" the use of the (inefficient) PCD system $(\mathbb{G}',\mathbb{P}',\mathbb{V}')$ (obtained from the inefficient SNARK $(\mathcal{G},\mathcal{P},\mathcal{V})$). Thus, the new SNARK $(\mathcal{G}^\star,\mathcal{P}^\star,\mathcal{V}^\star)$ is complexity preserving: the generator $\mathcal{G}^\star$ runs in time $p(k^c+k)\cdot p'(k)$, the prover $\mathcal{P}^\star$ runs in time $t\cdot\mathrm{poly}(k)\cdot p(k^c+k)\cdot p'(k)$ and space $s\cdot\mathrm{poly}(k)\cdot q(k^c+k)\cdot p'(k)$ (so time and space are preserved up to fixed $\mathrm{poly}(k)$ factors), and the verifier $\mathcal{V}^\star$ runs in time $|y|\cdot\mathrm{poly}(k)$.

The proof of knowledge property of $(\mathcal{G}^\star,\mathcal{P}^\star,\mathcal{V}^\star)$ follows from the proof of knowledge property of $(\mathbb{G}',\mathbb{P}',\mathbb{V}')$ and the guarantee of the Locally-Efficient RAM Compliance Theorem. Concretely, except with negligible probability over a random choice of (σ',τ'), if a polynomial-size prover $\mathcal{P}^*$, on input σ', outputs (y,π) such that $\mathcal{V}^\star(\tau', y, \pi) = 1$ (and thus, such that $\mathbb{V}'_{\mathbb{C}^h_y}(\tau, \mathsf{ok}, \pi) = 1$), we can efficiently extract from $\mathcal{P}^*$ an entire $\mathbb{C}^h_y$-compliant distributed computation transcript T with $\mathsf{out}(\mathsf{T}) = \mathsf{ok}$, and then (by the Locally-Efficient RAM Compliance Theorem) we can efficiently extract from T a witness w such that $M(x, w) = 1$.

To prove the second item of the theorem (namely, obtaining a complexity-preserving PCD system), we invoke *again* the SNARK Recursive Composition Theorem and the PCD Depth-Reduction Theorem, but this time we start with the complexity-preserving SNARK $(\mathcal{G}^\star, \mathcal{P}^\star, \mathcal{V}^\star)$; the resulting PCD systems are complexity preserving. □

Acknowledgments

We are grateful to Daniel Wichs for valuable discussions, in the early stages of this work, about understanding the construction of [Gro10] as a preprocessing SNARK. We also thank Daniel for pointing out a mistake, in an earlier draft of this paper, about a designated-verifier variant of [Gro10]. We also thank Yuval Ishai for valuable comments and discussions.

References

[ABOR00] William Aiello, Sandeep N. Bhatt, Rafail Ostrovsky, and Sivaramakrishnan Rajagopalan. Fast verification of any remote procedure call: Short witness-indistinguishable one-round proofs for NP. In *Proceedings of the 27th International Colloquium on Automata, Languages and Programming*, ICALP '00, pages 463–474, 2000.

[AV77] Dana Angluin and Leslie G. Valiant. Fast probabilistic algorithms for hamiltonian circuits and matchings. In *Proceedings on 9th Annual ACM Symposium on Theory of Computing*, STOC '77, pages 30–41, 1977.

[BC12] Nir Bitansky and Alessandro Chiesa. Succinct arguments from multi-prover interactive proofs and their efficiency benefits. In *Proceedings of the 32nd Annual International Cryptology Conference*, CRYPTO '12, pages 255–272, 2012.

[BCC88] Gilles Brassard, David Chaum, and Claude Crépeau. Minimum disclosure proofs of knowledge. *Journal of Computer and System Sciences*, 37(2):156–189, 1988.

[BCCT12] Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, ITCS '12, pages 326–349, 2012.

[BCI+13] Nir Bitansky, Alessandro Chiesa, Yuval Ishai, Rafail Ostrovsky, and Omer Paneth. Succinct non-interactive arguments via linear interactive proofs. In *Proceedings of the 10th Theory of Cryptography Conference*, TCC '13, pages ???–???, 2013.

[BEG+91] Manuel Blum, Will Evans, Peter Gemmell, Sampath Kannan, and Moni Naor. Checking the correctness of memories. In *Proceedings of the 32nd Annual Symposium on Foundations of Computer Science*, FOCS '91, pages 90–99, 1991.

[BFLS91] László Babai, Lance Fortnow, Leonid A. Levin, and Mario Szegedy. Checking computations in polylogarithmic time. In *Proceedings of the 23rd Annual ACM Symposium on Theory of Computing*, STOC '91, pages 21–32, 1991.

[BG08] Boaz Barak and Oded Goldreich. Universal arguments and their applications. *SIAM Journal on Computing*, 38(5):1661–1694, 2008. Preliminary version appeared in CCC '02.

[BHZ87] Ravi B. Boppana, Johan Håstad, and Stathis Zachos. Does co-NP have short interactive proofs? *Information Processing Letters*, 25(2):127–132, 1987.

[BOGW88] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In *Proceedings of the 20th Annual ACM Symposium on Theory of Computing*, STOC '88, pages 1–10, 1988.

[BP04] Mihir Bellare and Adriana Palacio. The knowledge-of-exponent assumptions and 3-round zero-knowledge protocols. In *Proceedings of the 24th Annual International Cryptology Conference*, CRYPTO '04, pages 273–289, 2004.

[BSCGT12] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, and Eran Tromer. On the concrete-efficiency threshold of probabilistically-checkable proofs, 2012. Electronic Colloquium on Computational Complexity, TR12-045.

[BSCGT13] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, and Eran Tromer. Fast reductions from RAMs to delegatable succinct constraint satisfaction problems. In *Proceedings of the 4th Innovations in Theoretical Computer Science Conference*, ITCS '13, pages ???–???, 2013.

[BSW12] Dan Boneh, Gil Segev, and Brent Waters. Targeted malleability: Homomorphic encryption for restricted computations. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, ITCS '12, pages 350–366, 2012.

[BV11] Zvika Brakerski and Vinod Vaikuntanathan. Efficient fully homomorphic encryption from (standard) LWE. In *Proceedings of the 51st Annual IEEE Symposium on Foundations of Computer Science*, FOCS '11, 2011.

[CKV10] Kai-Min Chung, Yael Kalai, and Salil Vadhan. Improved delegation of computation using fully homomorphic encryption. In *Proceedings of the 30th Annual International Cryptology Conference*, CRYPTO '10, pages 483–501, 2010.

[CR72] Stephen A. Cook and Robert A. Reckhow. Time-bounded random access machines. In *Proceedings of the 4th Annual ACM Symposium on Theory of Computing*, STOC '72, pages 73–80, 1972.

[CT10] Alessandro Chiesa and Eran Tromer. Proof-carrying data and hearsay arguments from signature cards. In *Proceedings of the 1st Symposium on Innovations in Computer Science*, ICS '10, pages 310–331, 2010.

[CT12] Alessandro Chiesa and Eran Tromer. Proof-carrying data: Secure computation on untrusted platforms (high-level description). *The Next Wave: The National Security Agency's review of emerging technologies*, 19(2):40–46, 2012.

[Dam92] Ivan Damgård. Towards practical public key systems secure against chosen ciphertext attacks. In *Proceedings of the 11th Annual International Cryptology Conference*, CRYPTO '92, pages 445–456, 1992.

[DCL08] Giovanni Di Crescenzo and Helger Lipmaa. Succinct NP proofs from an extractability assumption. In *Proceedings of the 4th Conference on Computability in Europe*, CiE '08, pages 175–185, 2008.

[DFH12] Ivan Damgård, Sebastian Faust, and Carmit Hazay. Secure two-party computation with low communication. In *Proceedings of the 9th Theory of Cryptography Conference*, TCC '12, pages 54–74, 2012.

[DLN+04] Cynthia Dwork, Michael Langberg, Moni Naor, Kobbi Nissim, and Omer Reingold. Succinct NP proofs and spooky interactions, December 2004. Available at `www.openu.ac.il/home/mikel/papers/spooky.ps`.

[FLP85] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985.

[FS87] Amos Fiat and Adi Shamir. How to prove yourself: practical solutions to identification and signature problems. In *Proceedings of the 6th Annual International Cryptology Conference*, CRYPTO '87, pages 186–194, 1987.

[Gen09] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the 41st Annual ACM Symposium on Theory of Computing*, STOC '09, pages 169–178, 2009.

[GGP10] Rosario Gennaro, Craig Gentry, and Bryan Parno. Non-interactive verifiable computing: outsourcing computation to untrusted workers. In *Proceedings of the 30th Annual International Cryptology Conference*, CRYPTO '10, pages 465–482, 2010.

[GGPR12] Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. Quadratic span programs and succinct NIZKs without PCPs. Cryptology ePrint Archive, Report 2012/215, 2012.

[GH98] Oded Goldreich and Johan Håstad. On the complexity of interactive proofs with bounded communication. *Information Processing Letters*, 67(4):205–214, 1998.

[GKR08] Shafi Goldwasser, Yael Tauman Kalai, and Guy N. Rothblum. Delegating computation: Interactive proofs for Muggles. In *Proceedings of the 40th Annual ACM Symposium on Theory of Computing*, STOC '08, pages 113–122, 2008.

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

[GMR89] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof systems. *SIAM Journal on Computing*, 18(1):186–208, 1989. Preliminary version appeared in STOC '85.

[GMW87] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or a completeness theorem for protocols with honest majority. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing*, STOC '87, pages 218–229, 1987.

[Gro10] Jens Groth. Short pairing-based non-interactive zero-knowledge arguments. In *Proceedings of the 16th International Conference on the Theory and Application of Cryptology and Information Security*, ASIACRYPT '10, pages 321–340, 2010.

[GS89] Yuri Gurevich and Saharon Shelah. Nearly linear time. In *Logic at Botik '89, Symposium on Logical Foundations of Computer Science*, pages 108–118, 1989.

[GVW02] Oded Goldreich, Salil Vadhan, and Avi Wigderson. On interactive proofs with a laconic prover. *Computational Complexity*, 11(1/2):1–53, 2002.

[GW11] Craig Gentry and Daniel Wichs. Separating succinct non-interactive arguments from all falsifiable assumptions. In *Proceedings of the 43rd Annual ACM Symposium on Theory of Computing*, STOC '11, pages 99–108, 2011.

[IKO05] Yuval Ishai, Eyal Kushilevitz, and Rafail Ostrovsky. Sufficient conditions for collision-resistant hashing. In *Proceedings of the 2nd Theory of Cryptography Conference*, TCC '05, pages 445–456, 2005.

[Kil92] Joe Kilian. A note on efficient zero-knowledge proofs and arguments. In *Proceedings of the 24th Annual ACM Symposium on Theory of Computing*, STOC '92, pages 723–732, 1992.

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

[KR09] Yael Tauman Kalai and Ran Raz. Probabilistically checkable arguments. In *Proceedings of the 29th Annual International Cryptology Conference*, CCC '09, pages 143–159, 2009.

[KY86] Anna R. Karlin and Andrew C. Yao. Probabilistic lower bounds for byzantine agreement. Unpublished manuscript, 1986.

[Lip12] Helger Lipmaa. Progression-free sets and sublinear pairing-based non-interactive zero-knowledge arguments. In *Proceedings of the 9th Theory of Cryptography Conference*, TCC '12, pages 169–189, 2012.

[Mic00] Silvio Micali. Computationally sound proofs. *SIAM Journal on Computing*, 30(4):1253–1298, 2000. Preliminary version appeared in FOCS '94.

[Mie08] Thilo Mie. Polylogarithmic two-round argument systems. *Journal of Mathematical Cryptology*, 2(4):343–363, 2008.

[Nao03] Moni Naor. On cryptographic assumptions and challenges. In *Proceedings of the 23rd Annual International Cryptology Conference*, CRYPTO '03, pages 96–109, 2003.

[NY89] Moni Naor and Moti Yung. Universal one-way hash functions and their cryptographic applications. In *Proceedings of the 21st Annual ACM Symposium on Theory of Computing*, STOC '89, pages 33–43, 1989.

[Rom90] John Rompel. One-way functions are necessary and sufficient for secure signatures. In *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing*, STOC '90, pages 387–394, 1990.

[RV09] Guy N. Rothblum and Salil Vadhan. Are PCPs inherent in efficient arguments? In *Proceedings of the 24th IEEE Annual Conference on Computational Complexity*, CCC '09, pages 81–92, 2009.

[Sch78] Claus-Peter Schnorr. Satisfiability is quasilinear complete in NQL. *Journal of the ACM*, 25:136–145, January 1978.

[Val08] Paul Valiant. Incrementally verifiable computation or proofs of knowledge imply time/space efficiency. In *Proceedings of the 5th Theory of Cryptography Conference*, TCC '08, pages 1–18, 2008.

[Wee05] Hoeteck Wee. On round-efficient argument systems. In *Proceedings of the 32nd International Colloquium on Automata, Languages and Programming*, ICALP '05, pages 140–152, 2005.