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

March 19, 2012

Abstract

Succinct non-interactive arguments of knowledge (SNARKs), and their generalization to distributed computations by *proof-carrying data* (PCD), are powerful tools for enforcing the correctness of computations in dynamic networks with multiple mutually-untrusting parties, with essentially minimal computational overhead. Current constructions achieve only variants with expensive setup, restricted functionality, or oracles.

We present recursive composition and bootstrapping techniques that:

1. Transform any SNARK with an expensive preprocessing phase into a SNARK without such a phase.
2. Transform any SNARK into a PCD system for constant-depth distributed computations.
3. Transform any PCD system for constant-depth distributed computations into a PCD system for distributed computation over paths of fixed polynomial length.

Our transformations apply to both the public and private verification settings, and assume the existence of CRHs (and FHE, for the private-verification setting).

By plugging into our transformations the NIZKs of [Groth, ASIACRYPT '10], whose security is based on a Knowledge of Exponent assumption in bilinear groups, we obtain *the first publicly-verifiable SNARKs and PCDs without preprocessing in the plain model*. (Previous constructions were either in the random-oracle model [Micali, FOCS '94] or in a signature oracle model [Chiesa and Tromer, ICS '10].) Interestingly, the constructions we obtain do not rely on the PCP Theorem and are quite efficient.

*Supported by the Check Point Institute for Information Security, Marie Curie grant PIRG03-GA-2008-230640, and ISF grant 0603805843.

†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

The general goal. The modern computing environment is a highly-connected one, where numerous machines, databases and networks, controlled by myriad organizations and individuals, are all simultaneously involved in various computations — and may not trust each other's results. Questions of correctness arise from the largest scale of outsourcing computational infrastructure and functionality, to the smallest scale of building devices out of externally-supplied components. Answering these challenges requires efficient mechanisms for ensuring integrity and generating trust. Cryptography offers a host of potential solutions, but as we argue below, to date these are only partial.

Succinct interactive arguments. The notion of interactive proofs [GMR89] provides the basic foundations for protocols that allow a party to verify correctness of claims made by other parties. Kilian [Kil92], based the work on probabilistically-checkable proofs starting from [BFLS91], showed an interactive proof system where the time required to verify correctness of an NP computation is only polylogarithmic in the time required to perform the computation in the first place; following tradition, we call such proof systems *succinct*. Kilian's protocol is only computationally sound, i.e., it is an *argument* system [BCC88]. (Indeed, there is evidence that obtaining succinct statistically-sound proofs for NP is hard; see e.g. [BHZ87, GH98, GVW02, Wee05].)

Knowledge extraction. A natural strengthening of soundness (which is also satisfied by Kilian's protocol) is *proof of knowledge*. This property, which turns out to be very useful in the context of succinct arguments, guarantees that whenever the verifier is convinced by a prover, we can not only conclude that a valid witness for the theorem *exists*, but also that such a witness can be *efficiently extracted* from the prover. This captures the intuition that 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.

Removing interaction. Kilian's protocol requires four messages. A challenge, which is of both theoretical and practical essence, is to come up with a non-interactive protocol that obtains similar properties. As a first step in this direction, Micali [Mic00] shows how to construct publicly-verifiable *one-message* succinct non-interactive arguments for NP, in the random oracle model, by applying the Fiat-Shamir paradigm [FS87] to Kilian's protocol. Valiant [Val08] shows that Micali's protocol is a proof of knowledge.

Extending earlier attempts [ABOR00, DLN+04, DCL08], a set of recent works [BCCT11, DFH11, GLR11] show how to construct two-message succinct arguments of knowledge in the plain model, where the verifier message is generated independently of the statement to be proven. Following [GW11, BCCT11], we call such arguments SNARGs of knowledge, or SNARKs. They are constructed assuming the existence of *extractable collision-resistant hash functions*.

Public verifiability. The SNARKs in [BCCT11, DFH11, GLR11] are of the *designated-verifier* kind: the verifier keeps a secret state τ associated with its message. This information is needed in order to verify the prover's message. In contrast, Micali's protocol is *publicly verifiable:* any proof generated in that protocol can be verified by anyone, without forcing the prover to generate the proof specifically for that verifier. Furthermore, proofs can be archived for future use, without any secrets associated with them. We thus ask:

Question 1: Can we construct publicly-verifiable SNARKs?

Indeed, one can always assume that Micali's protocol, when the random oracle is instantiated with a sufficiently complicated hash function, is secure; but this seems more like a heuristic rather than a rigorous guarantee. In contrast, Gentry and Wichs [GW11] show that no non-interactive succinct argument can be proven adaptively-sound via a black-box reduction to a falsifiable assumption [Nao03], even in the

designated-verifier case, and even if we drop the knowledge requirement. This suggests that non-standard assumptions may be inherent here. Still, we would like to have a solution whose security is based on a concise and as general as possible assumption that can be studied separately.

1.1 Verifying Arbitrary Distributed Computations

The solutions discussed so far concentrate on the case of a single prover and a single verifier. This suffices for capturing, say, a client interacting with a single "worker" who performs some self-contained computation on behalf of the client. However, reality is much more complex: computations involve multiple parties, where each party has its own role, capabilities, and trust relations with others.

Consider for instance a server that wishes to prove to a customer that the customer's website, which is hosted on the server, interacted correctly with clients and provided the clients with information taken from legitimate databases; or verifying correctness of an ongoing computation that moves from party to party and dynamically unfolds.

In general, we have multiple (even unboundedly many) parties, where party i, given inputs from some other parties, locally executes a program prog_i for t_i steps, and sends its output z_i to other parties, each of which will in turn act likewise (i.e., perform local computation and send the output to other parties), and so on.

How can a participant in such computation verify that its inputs comply with the computation that "was supposed to happen so far"? A first suggestion may be to use secure multiparty computation. However, these solutions (such as [GMW87, BOGW88]) require all parties to co-exist and interact heavily. Furthermore, they are far from being succinct.

Proof-carrying data. Instead, Chiesa and Tromer [CT10] propose an alternative solution approach: the parties maintain the original communication pattern of the original protocol, and only add to each message a *succinct proof* that asserts the correctness of the computation leading to the generation of the data. In fact, the proof will assert not only the correctness of the computation of the party itself, but rather the correctness of *all the computation done by all the parties leading to the current message.* This solution approach is called *Proof-Carrying Data* (PCD). See Figure 1 for a diagram of this idea.

Proof Carrying Data is indeed a very powerful tool. However, to date we only know how to construct general PCD systems in a model where all parties have access to a signature oracle [CT10]. This stands in sharp contrast with the case of SNARKs, which can be seen as "one hop PCDs". We thus ask:

Question 2: Is there PCD in the plain model? For which computations? And under which assumptions?

A natural approach to obtain PCD (that was used in [CT10]) is to recursively compose SNARKs; that is, have each party use a SNARK to prove to the next party down the line that the local computation was correct, and that the proofs provided by the previous parties were verified. The knowledge property is key in making the composition work. Can this approach be made to work even for SNARKs in the plain model?

1.2 Our Results

We give positive answers to the questions about publicly-verifiable SNARKs and PCDs in the plain model: we show that, starting from a significantly weaker notion of *preprocessing SNARKs* (which allows the verifier to conduct an expensive "offline" phase) we can obtain both publicly-verifiable SNARKs and PCDs for a large class of distributed computations. Preprocessing SNARKs are known to exist in the plain model under simple knowledge assumptions using bilinear techniques [Gro10]. Specifically, we prove:

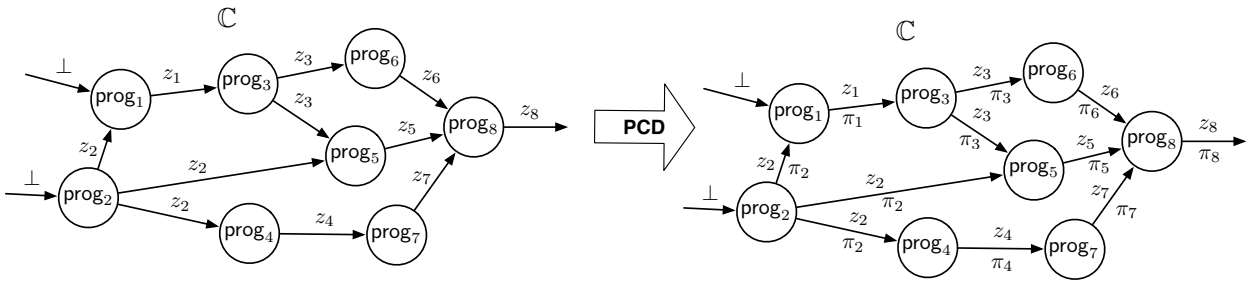

Figure 1: Proof-carrying data enables each party in a distributed computation to augment his message z_i with a short easy-to-verify proof π_i computed "on-the-fly". At any point during the computation, a party may inspect the computation's outputs to decide if they are "compliant" with a given property $\mathbb{C}$. Distributed computations are naturally represented as directed acyclic graphs, when "unfolded over time".

Theorem. *In both the publicly- and privately-verifiable cases, if there exist preprocessing SNARKs, collision-resistant hashes (and, in the privately-verifiable setting, fully homomorphic encryption), then there exist:*

(i) SNARKs with no preprocessing.

(ii) PCDs with no preprocessing for constant-depth distributed computations.

(iii) PCDs with no preprocessing for fixed-polynomial-depth distributed computations over paths.

(The *depth* of a distributed computation is, roughly, the length of the longest path in the graph representing the distributed computation when "unfolded" over time.)

As corollary to our main result we obtain:

> **Publicly-verifiable SNARKs in the plain model.** By plugging into our transformations the preprocessing succinct NIZKs of Groth [Gro10], we obtain (based on Groth's knowledge assumption) the first publicly-verifiable SNARK in the plain model (a.k.a. "CS proofs" [Mic00]).

To prove our results we develop three main tools:

1. **SNARK Recursive Composition Theorem:** "Any SNARK can be composed a constant number of times to obtain a PCD system for constant-depth distributed computations." (Surprisingly, this is also true for designated-verifier SNARKs.)

2. **PCD Bootstrapping Theorem:** "Distributed computations of constant depth can express distributed computations over paths of polynomial length."

3. **RAM Compliance Theorem**: "The problem of whether, for a given random-access machine M, input x, and time bound t, there is a witness w that makes $M(x, w)$ accept within t steps can be computationally reduced to the problem of checking the correctness of a distributed computation along a path, where every node's computation time is only $\text{poly}(|M|, |x|, k)$ for a security parameter k."

PCD & compliance engineering. The above theorems highlight proof-carrying data not only as a desirable primitive, but also as a powerful tool in its own right: it provides a flexible and expressive abstraction for

engineering and then ensuring the correctness (or *compliance*) of distributed computations, a task that lies at the heart of the techniques developed in this paper.

Succinct Arguments without PCPs. When combined with the bilinear techniques of [Gro10], our transformations yield SNARK and PCD constructions that, unlike all previous constructions (even interactive ones), do not invoke the PCP Theorem. The resulting construction, while not trivial, seems to be quite simple and efficient. We thus show an essentially different (and potentially more practical) path to the construction of succinct arguments, which diverges from previous PCP-based approaches (such as applying the Fiat-Shamir paradigm [FS87] to Micali's "CS proofs" [Mic00]). We find this interesting even on a heuristic level.

1.2.1 The ideas in a nutshell

Let us provide some intuition, stripping away all abstraction barriers, for perhaps the more surprising of our results: how can we remove the handicap of preprocessing from a (say, publicly-verifiable) SNARK?

In a preprocessing SNARK, in order to be convinced of the correctness of a long t-step computation, the verifier must first spend time proportional to t during an expensive "offline" phase to generate a public *reference string* and a short *verification state*. Our goal is to enable the verifier to avoid conducting such an expensive offline phase.

At high-level, our approach is to (a) represent a long t-computation as an collection of many smaller $\mathrm{poly}(k)$-computations (where k is the security parameter), and (b) develop techniques that allow for aggregating many SNARKs for the correctness of smaller computations into a *single succinct* proof. Intuitively these two steps should suffice because then we would only be using preprocessing SNARKs for proving the correctness of small computations, so that the preprocessing phase would be as cheap as $\mathrm{poly}(k)$ rather than as expensive as $\mathrm{poly}(t)$.

Specifically, to be convinced that , say, a machine M non-deterministically accepts within t steps, it suffices to be convinced that there is a sequence of $t' \leq t$ states $S_0, S_1, \ldots, S_{t'}$ of M 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, 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 way, the computation of each party is seemingly small: it only involves proving correctness of a *single* step of M.

The above discussion suggests the following solution approach: using the preprocessing SNARK, 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 the pre-processing SNARK, 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 with a valid proof claiming that S_1 was generated correctly. Then, P_3 proves 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 with 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 an accepting one.

A verifier at the end of the chain would receive a single easy-to-verify proof that aggregates the correctness of all steps in the computation of M. Furthermore, because each application of the preprocessing SNARK was on a small computation (namely, one step of the machine), the preprocessing phase is in fact *not expensive at all*.

So what is missing? The above intuition hides several important technical difficulties. A first difficulty is that an arbitrary state S_i in the "middle" of the computation of M may in fact be as large as t (i.e., the ma-

chine may use a lot of space). We solve this difficulty by invoking a computational reduction of Ben-Sasson et al. [BSCGT12] that transforms M to a new machine M' only requiring $\mathrm{poly}(k)$ space and preserves the proof of knowledge property (i.e., any efficient adversary producing a witness for M' can be used to find a witness for M); roughly, this is done by moving the memory into the witness via Merkle hashing.

A second difficulty is that the above approach requires us to be able to prove security for a recursive composition of polynomially many proofs. However, due to the potential blow-up in extraction running-time, we are only able to prove security for a constant-depth of recursive proof composition (without relying on SNARKs with strong extractability properties). To deal with this issue, instead of structuring the distributed computation and aggregating proofs along a line, we show how this can be done using a wide Merkle tree of constant depth, via a "tree-squashing trick" that was already used in the SNARK constructions of [BCCT11, GLR11] for similar technical reasons.

Another difficulty is how to make the construction described above go through even when the SNARK proofs require a private state in order to be verified: in this case, it is not clear how a party can prove that he verified a a received proof without actually knowing the corresponding verification state. We solve this problem by showing how to use fully-homomorphic encryption to recursively compose proofs without knowledge of the verification state. Formalizing the above solution approach, as well as the tools that enable us to overcome the aforementioned technical difficulties, will greatly benefit from the abstractions provided by PCD systems and compliance predicates; ultimately, these will provide clean and simple explanations for what is actually going on.

1.3 Roadmap

In Section 2, we discuss our results in slightly more detail, describing each of the three tools we develop, and then how they come together for our main result. We then proceed to the technical sections of the paper, beginning with definitions of the universal relation and random-access machines in Section 3, of SNARKs in Section 4, and of proof-carrying data in Section 5. After that, we give the technical details for our three tools, in Section 6, Section 7, and Section 8 respectively. In Section 9, we recall existing constructions of preprocessing SNARKs. In Section 10, we finally give the technical details for how our tools come together to yield the transformations claimed by our main theorem, and then what constructions can be plugged into these transformations. In Section 12, we discuss cryptographic applications of proof-carrying data.

2 Overview of Results

We discuss our results in more detail.

2.1 Recalling SNARKs and Proof-Carrying Data

Recall that *non-interactive succinct arguments of knowledge* (SNARKs) are two-message succinct arguments of knowledge where the verifier's message is independent of the statement to be proved.

Proof-carrying data (PCD), alluded to in Section 1.1 above, generalizes the notion of a SNARK to the setting of distributed computations, by offering a simple and general paradigm to reason about and ensure the security of distributed computations. Given a security property $\mathbb{C}$, called a *compliance predicate*, PCD enforces compliance with $\mathbb{C}$, in the presence of adversarial behavior, by attaching succinct proofs "on-the-fly" to every message exchanged between parties during the distributed computation.

In this paper, PCD will serve not only as an ultimate goal but also as a powerful *tool*; indeed, PCD allows to elegantly abstract away details about recursive proof composition of SNARKs and directly leverage the power of verifying certain carefully-chosen compliance predicates in distributed computations.

We now recall in slightly more detail the main components and guarantees of a *PCD system*, which formally captures the PCD framework as a cryptographic primitive; we do so by comparing these alongside the (more familiar) SNARK. We defer a formal discussion of SNARKs and PCD systems to Section 4 and Section 5 respectively.

Components. A SNARK for an NP language L is a triple of algorithms $(\mathcal{G}, \mathcal{P}, \mathcal{V})$ that works as follows. The (probabilistic) generator $\mathcal{G}$, on input the security parameter k, outputs a *reference string* σ and a corresponding *verification state* τ; the generator is run during an offline phase by the verifier or someone the verifier trusts The (honest) prover $\mathcal{P}(\sigma, y, w)$ produces a proof π for the statement $y = (M, x, t)$ given a witness w for $y \in L$; then $\mathcal{V}(\tau, y, \pi)$ verifies the validity of π.

Just like a SNARK, a PCD system for a given compliance predicate $\mathbb{C}$ (representing some desired security property) is also a triple of algorithms $(\mathbb{G}, \mathbb{P}_{\mathbb{C}}, \mathbb{V}_{\mathbb{C}})$; roughly, these work as follows. The (probabilistic) generator $\mathbb{G}$, on input the security parameter k, outputs a reference string σ and a corresponding verification state τ. The (honest) prover $\mathbb{P}_{\mathbb{C}}(\sigma, \mathsf{z_o}, \mathsf{prog}, \vec{\mathsf{z}}_{\mathsf{i}}, \vec{\pi}_{\mathsf{i}})$ is given σ, a (claimed) output $\mathsf{z_o}$, a local program prog (which may include code and local input and randomness), and input messages $\vec{\mathsf{z}}_{\mathsf{i}}$ with corresponding proofs $\vec{\pi}_{\mathsf{i}}$; $\mathbb{P}_{\mathbb{C}}$ then produces a proof π_{o} for the claim that $\mathsf{z_o}$ is consistent with a $\mathbb{C}$-compliant distributed computation leading up to it. The verifier $\mathbb{V}_{\mathbb{C}}$ is given the verification state τ, an output $\mathsf{z_o}$, and a proof string π_{o}, and should accept only when it is convinced that the output is consistent with some $\mathbb{C}$-compliant distributed computation leading up to $\mathsf{z_o}$.

A distributed computation graph. A distributed computation, when "unfolded over time", can simply be thought of as a (labeled) directed acyclic[1] graph (generated dynamically "on the fly" as the distributed computation evolves) where computations occur at nodes, and directed edges denote messages exchanged between parties. This graphical representation will be a very useful one to keep in mind when reasoning about distributed computations. See Figure 1.

Succinctness. A SNARK requires the generator $\mathcal{G}(1^k)$ to run in time $\mathrm{poly}(k)$, the (honest) prover $\mathcal{P}(\sigma, y, w)$ in time $\mathrm{poly}(k, |y|, t)$, and the verifier $\mathcal{V}(\tau, y, \pi)$ in time $\mathrm{poly}(k, |y|)$. Analogously, the generator $\mathbb{G}(1^k)$ is required to run in time $\mathrm{poly}_{\mathbb{C}}(k)$, the (honest) prover $\mathbb{P}_{\mathbb{C}}(\sigma, \mathsf{z_o}, \mathsf{prog}, \vec{\mathsf{z}}_{\mathsf{i}}, \vec{\pi}_{\mathsf{i}})$ in time $\mathrm{poly}_{\mathbb{C}}(k, |\mathsf{z_o}|, t_{\mathbb{C}}(|\mathsf{z_o}|))$, and the verifier $\mathbb{V}_{\mathbb{C}}(\tau, \mathsf{z_o}, \pi_{\mathsf{o}})$ in time $\mathrm{poly}_{\mathbb{C}}(k, |\mathsf{z_o}|)$, where $t_{\mathbb{C}}(|\mathsf{z_o}|)$ is the time to evaluate $\mathbb{C}(\mathsf{z_o}; \vec{\mathsf{z}}_{\mathsf{i}}, \mathsf{prog})$ and $\mathrm{poly}_{\mathbb{C}}$ is a polynomial only depending on $\mathbb{C}$.

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 mater how "long" and expensive is the history of computations that eventually led to the message being verified).

We can also consider a significantly-weaker *(expensive) preprocessing* definition in which the generator may run in time $\mathrm{poly}(k, B)$ and the reference string only works for computations of length at most B. (In the PCD case, the bound B refers to a single node's computation, and not the entire distributed computation.)

Security. A SNARK has a proof of knowledge property: when a malicious prover $\mathcal{P}^*(\sigma)$ produces a statement y and proof π such that $\mathcal{V}(\tau, y, \pi)$ accepts then, with all but negligible probability, the extractor $\mathcal{E}_{\mathcal{P}^*}(\sigma)$ outputs a valid witness w for y. In other words, $\mathcal{V}$ can only be convinced to accept a given statement whenever the prover $\mathcal{P}^*$ actually "knows" a valid witness for that statement.

Analogously, a PCD system also has a proof of knowledge property: when a malicious prover $\mathbb{P}^*(\sigma)$

[1] When the same party computes twice, it will be a separate node "further down" the graph.

produces a message $\mathsf{z_o}$ and proof π_o such that $\mathbb{V}_\mathbb{C}(\tau, \mathsf{z_o}, \pi_\mathsf{o}) = 1$ then, with all but negligible probability, the extractor $\mathbb{E}_{\mathbb{P}^*}(\sigma)$ outputs *an entire distributed computation* that is $\mathbb{C}$-compliant and leads up to the message $\mathsf{z_o}$. In other words, $\mathbb{V}_\mathbb{C}$ can only be confined to accept a given message whenever the prover $\mathbb{P}^*$ actually "knows" a $\mathbb{C}$-compliant computation leading up to that message.

Thus, a PCD system for $\mathbb{C}$ induces a *compiler* that, given a protocol for a distributed computation, yields an augmented protocol that enforces $\mathbb{C}$, because each party can simply use $\mathbb{P}_\mathbb{C}$ to compute proofs for outgoing messages based on proof-carrying incoming messages, and $\mathbb{V}_\mathbb{C}(\tau, \cdot, \cdot)$ can be used to succinctly verify any message. This compiler *respects* the original distributed computation because it preserves the computation's communication graph, dynamics, and efficiency (concretely, the compiler only asks parties to compute proofs and attach them to messages).

But what "security properties" should $\mathbb{C}$ express? Certainly we want to be able to express, if desired, the property that every node carried out its own computation without making any mistakes. But this is only one very specific choice for $\mathbb{C}$. Another example is having $\mathbb{C}$ require that, not only each party's computation was carried out without errors, but also that the program run by each party carried a signature valid under the system administrator's public key; in such a case, the *local program* supplied by each party would be the combination of the program and the signature. Generally, the choice of $\mathbb{C}$ varies with the application. This notion of compliance is essentially the most powerful integrity guarantee for distributed computation one can think of, short of interfering with the communication and dynamics of the original distributed computation. Expressing security properties as a compliance predicate (to be enforced by PCD) constitutes *compliance engineering*. When we use PCD as a technical tool in this paper, we do so by engineering a suitable $\mathbb{C}$.

Previous constructions. Chiesa and Tromer showed how to construct PCD systems from standard cryptographic assumptions in a model where every party has access to a signature oracle. As mentioned, PCD systems are not known to exist in the random-oracle model, despite the existence of CS proofs of knowledge in the random oracle model. In this paper we shall work in the *plain model*, and thus the definitions of PCD systems will be adapted to this model.

PCD systems essentially generalize the "computationally-sound proofs" of Micali [Mic00], which consider the "one-hop" case of a single prover and a single verifier, and also generalize the "incrementally verifiable computation" of Valiant [Val08], which considers the case of an a-priori fixed sequence of computations. Special cases of PCD systems include targeted malleability [BSW11] and computing on signatures [BF11] and authenticated data [ABC$^+$11]; see Section 2.6 for further discussion, and Section 12 for details.

2.2 Constant Depth PCDs and the SNARK Recursive Composition Theorem

Our first step is establishing PCDs for a specific class of *constant depth compliance predicates*. The depth of a predicate $\mathbb{C}$, denoted by $\mathsf{d}(\mathbb{C})$ 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 a given depth $\mathsf{d}(\mathbb{C})$ (even a constant) may have many more "nodes" than $\mathsf{d}(\mathbb{C})$; e.g., it could be a very wide tree of height $\mathsf{d}(\mathbb{C})$.

We show that any SNARK can be composed a constant number of times, yielding constant-depth PCDs:

Theorem 1 (SNARK Recursive Composition — informal)**.** *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. More precisely:*

(i) If there exist publicly-verifiable SNARKs, then there exist publicly-verifiable PCD systems for every constant-depth compliance predicate.

(ii) Assuming the existence of FHE, if there exist designated-verifier SNARKs, then there exist designated-verifier PCD systems for every constant-depth compliance predicate.

Moreover, the SNARK can be of the preprocessing kind, in which case so is the PCD system.

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 proof composition, we can simply and conveniently work "in the abstract" by engineering a (constant-depth) compliance predicate that will enforce the desired properties across a distributed computation, and then invoke the guarantees of a PCD system, without worrying about its implementation details (which happen to involve recursive composition of SNARKs). In the next subsection (Section 2.3), we discuss how to achieve more in certain special cases. We now outline the idea behind each part of the theorem above.

The case of public verifiability. In this case, the construction itself is similar to the one in Chiesa and Tromer [CT10] (where it was done with a signature oracle). The basic idea is to apply recursive verification of proofs relative to the compliance predicate $\mathbb{C}$.

Roughly, when a party A wishes to start a computation with a message z_A for party B, A will attach to z_A a SNARK proof π_A for the claim "$\mathbb{C}(\mathsf{z}_A; \perp, \perp) = 1$", which essentially says that z_A is a compliant "source input" for the distributed computation. When party B receives z_A and, after performing some computation according to some local input (or code) prog_B, B produces a message z_B for party C; B will also attach to z_B a SNARK proof π_B for the claim "$\exists\,(\mathsf{prog}_B, \mathsf{z}_A, \pi_A)$ s.t. $\mathbb{C}(\mathsf{z}_B; \mathsf{prog}_B, \mathsf{z}_A) = 1$ *and* π_A is a valid SNARK proof for the $\mathbb{C}$-compliance of z_A". And so on: in general, any party receiving input messages $\vec{\mathsf{z}}_\mathsf{i}$ with corresponding proofs $\vec{\pi}_\mathsf{i}$ will use the SNARK prover to generate π_o for the claim

"$\exists\,(\mathsf{prog}, \vec{\mathsf{z}}_\mathsf{i}, \vec{\pi}_\mathsf{i})$ s.t. $\mathbb{C}(\mathsf{z}_\mathsf{o}; \mathsf{prog}, \vec{\mathsf{z}}_\mathsf{i}) = 1$ *and* each π_i is a valid SNARK proof for the $\mathbb{C}$-compliance of z_i"

for the output message z_o, when giving a local input prog.

Crucially, the proof of knowledge property of the SNARK makes the above idea work. Indeed, there likely *exists* a proof, say, π_1 for the $\mathbb{C}$-compliance of z_1, even if this is not the case, 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, already 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 a "bad" one).

The above intuitive construction is formally captured by proving SNARK statements regarding the computation of a *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 proof that attests to its own $\mathbb{C}$-compliance. More precisely, for each incoming input z and proof π in the witness, it checks that the SNARK verifier accepts the statement saying that $\mathsf{M}^{\mathbb{C}}$ itself accepts z after a given number of steps.

The proof of security is technically quite different from the one in Chiesa and Tromer [CT10], as now we are working in the plain model, as opposed to a model where parties have black-box to a signature oracle.

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 verifier accepts". Intuitively, fully-homomorphic encryption (FHE) may help in resolving this problem; using FHE in the right way, however, is rather subtle.

Specifically, 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 evaluated

the verifier, and that a certain bit encryption 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 $\mathbb{C}$-compliant or 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.

Preprocessing. The SNARK Recursive Composition Theorem can also be instantiated with preprocessing SNARKs; we will make use of this fact in our end result (discussed in Section 2.5).

In a preprocessing SNARK, the verifier (or some generator trusted by him) is given, in an offline phase, a time bound B and then outputs a (possibly long) reference string σ for the prover along with a short verification state τ; the running time of this offline phase may be polynomial in B (and the security parameter). Later, in the "online" phase, a prover uses σ to generate a proof that can now be succinctly verified by the verifier by using τ; however, (σ, τ) work *only* for computations that are as as long as B.

When plugging a preprocessing SNARK into our SNARK Recursive Composition Theorem, we obtain a corresponding preprocessing PCD, where again the offline generator will get a time bound B, and the resulting PCD will only work as long as the amount of computation done at each node in the distributed computation (or, more precisely, the time to verify compliance at each node) is bounded by B. More concretely, using preprocessing SNARKs with time bound B', we construct PCD that allows the computation at each node i to be roughly as large as $B' - \deg(i) \cdot t_{\mathcal{V}}$, where $t_{\mathcal{V}}$ is the time required for SNARK verification and $\deg(i)$ is the number of incoming inputs (which is also the number of proofs to be verified); thus we can simply set $B' = B + \max \deg(i) \cdot t_{\mathcal{V}}$. (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.

Finally, note that, while we do not need to provide 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, [BSW11] studied targeted malleability, which is a special case of a PCD system, and presented a preprocessing construction that does not leverage the succinctness of the SNARK verifier, and hence their preprocessing is with respect to the *entire* distributed computation and not just a *single* node's computation (as in our case). This difference is crucial in our applications (which ultimately achieve removing preprocessing altogether.) See Section 12.1 for a technical comparison.

Why only constant depth? The restriction for constant-depth compliance predicates arises because of technical reasons during the proof of security. Specifically, we must apply SNARK extraction "into the past", which can be done iteratively at most a constant number of times as each extraction potentially blows up the size of the extractor by a polynomial factor. (See Remark 6.3 for more details.) Nevertheless, in the next section we show that constant depth PCDs can be rather expressive.

A formal discussion of SNARKs and PCDs can be respectively found in Section 4 and Section 5, and the proof of the SNARK Recursive Composition Theorem in Section 6.

2.3 The PCD Bootstrapping Theorem and Path PCD

The SNARK Recursive Composition Theorem presented in the previous section gives us PCDs for constant-depth compliance predicates from any SNARK. While already significantly more powerful than SNARKs, PCDs for only constant-depth compliance predicates are still not completely satisfying; in general, we may want to ensure the compliance of "polynomially-deep" distributed computations.

Nonetheless, we show that such PCDs can "bootstrap themselves" to yield PCDs for long paths. Specifically, a **path** PCD system is one where *completeness* does not necessarily hold for any compliant computation, but only for distributed computations where the associated graph is a path, i.e., each node has only a single input message. We show:

Theorem 2 (PCD Bootstrapping — informal)**.** *If there exist PCDs for constant-depth compliance predicates, then there exist path PCDs for compliance predicates of fixed polynomial depth.*[2] *(And the verifiability properties of the PCD carry over, as do the preprocessing properties.)*

The high-level idea of the proof is as follows. Say that $\mathbb{C}$ has fixed polynomial depth $\mathsf{d}(\mathbb{C}) = k^c$. We proceed in two steps:

1. 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 had been "filled in" with a $\mathbb{C}$-compliant distributed computation along a path.

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) 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. (Note that the number of proofs sent between real nodes is not very large: 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 [BCCT11, 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 incrementally-verifiable computation.

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 (where a diagram outlining the construction can also be found, see Figure 7).

Effect on the preprocessing case. Recall that if the constant depth PCD is a preprocessing one, there is a bound B on the allowed computation at each node. In particular, using such a preprocessing PCD in the Bootstrapping Theorem results in preprocessing path PCDs where the bound on the allowed computation at each node along the path is the same as allowed by the underlying constant depth PCD, up to polynomial factors in the security parameter.

2.4 The RAM Compliance Theorem

We will ultimately rely (as described in Section 2.5) on the SNARK Recursive Composition Theorem and PCD Bootstrapping Theorem for transforming a given preprocessing SNARK into one without preprocessing. One last ingredient that we rely on in this transformation is the following.

We seek to reduce the task of verifying SNARK statements into the task of verifying the compliance of a related distributed computation evolving over a path. Our goal is to make sure that each node along the

[2] This can be generalized; see Remark 8.9.

path only performs a *small* amount of work, which will, in turn, enable us to use SNARKs with "not-too-expensive" preprocessing to plug into the SNARK Recursive Composition Theorem, and then plugging the result into the PCD Bootstrapping Theorem for obtaining a PCD system to ensure the desired compliance.

In attempting to construct the reduction we seek, we encounter the following problem: an arbitrary machine M running in time t (on some input x) may in general use a large amount of memory (possibly as large as t), hence naïvely breaking its computation into smaller computations that go from one state to the next one, will not work — the resulting nodes may need to perform work as large as t (just to read the state).

To deal with this obstacle we invoke a result of Ben-Sasson et al. [BSCGT12] that essentially shows how to use Merkle hashing to transform any M to a new "computationally equivalent" machine M' that "outsources" its memory, and itself checks memory consistency as it runs. We can then engineer a compliance predicate for ensuring correct computation of M', one step at a time.

It will be convenient (and consistent with [BSCGT12]) to state our results in terms of random-access machines [CR72, AV77] (though, they naturally generalize to other abstract models of computation, such as Turing machines):

Theorem 3 (RAM Compliance Theorem — informal)**.** *Let $\mathcal{H}$ be a family of collision-resistant hashes. For any $y = (M, x, t)$ and any $h \in \mathcal{H}$, there exists a compliance predicate $\mathbb{C}_{y,h}$ with depth $O(t \log t)$, such that:*

1. **Completeness:** *Given $w \in \mathcal{R}_{\mathcal{U}}(y)$ it is possible to efficiently emulate a distributed computation along a path that is compliant with $\mathbb{C}_{y,h}$. Moreover, in such a distributed computation, each node performs work that is only $\mathrm{poly}(|y|, k)$.*

2. **Proof of knowledge:** *From any efficient adversary that, given a random h, outputs a $\mathbb{C}_{y,h}$-compliant distributed computation we can extract a witness $w \in \mathcal{R}_{\mathcal{U}}(y)$.*

The proof for the theorem consists of two main steps:

(i) **Delegate the Memory.** In general, to verify the correct execution of a random-access machine, one has to verify both *code consistency* (i.e., the current instruction executed correctly) and *memory consistency* (i.e., the value obtained from memory at a certain address was indeed the last value written there). We invoke as the first step a result of Ben-Sasson et al. [BSCGT12], who showed that a t-step random-access machine can be easily modified into a new $t \cdot \mathrm{poly}(k)$-step random-access machine that checks its own memory for consistency (by simply dynamically maintaining a Merkle tree over an "untrusted memory", thereby delegating memory to an untrusted storage). This is where the computational part of the theorem comes in, and we have reduced the problem to the simpler one of checking only code consistency of a random-access machine.

(ii) **Engineer Compliance for Code Consistency.** We show that ensuring code consistency of a random-access machine can be reduced to ensuring compliance of a distributed computation with respect to a compliance predicate $\mathsf{UMC}_{(M,x,t)}$, which we call the *Untrusted Memory Checker* for (M, x, t). Essentially, the idea is to simply go through the computation of the machine one step at a time, always ensuring code consistency. Crucially, ensuring code consistency (without memory consistency) only requires a very small state of the machine and verifying it is easy. So eventually the amount of computation done by each node is only $\mathrm{poly}(|y|, k)$. (This second step is yet another instance of useful compliance engineering.)

The the theorem is formally stated in Section 7; in Section 7.1 we recall the idea of the proof of the first step by Ben-Sasson et al. [BSCGT12], and then in Section 7.2 we give the details for the second step.

Bootstrapping for more efficiency. Interestingly, the RAM Compliance Theorem and the PCD Bootstrapping Theorem not only have theoretical consequences (which we will discuss shortly in Section 2.5), but also *efficiency* consequences. Namely, by combining the RAM Compliance Theorem and the PCD Bootstrapping Theorem, we can show that the mere existence of PCDs implies the existence of corresponding PCDs with *much better* efficiency:

Theorem 2.1 (PCD Linearization — informal)**.** *If there exist constant-depth PCDs, then there exist constant-depth PCDs where the PCD prover running time is* $t\mathrm{poly}(k)$ *where* t *is the time of the computation performed at the node (and polynomial in the security parameter).*

2.5 Putting Things Together: SNARKs and PCDs without Preprocessing

Finally, equipped with the SNARK Recursive Composition, PCD Bootstrapping, and RAM Compliance Theorems, we can now present our end result:

Theorem 4 (Main Theorem)**.** *Assume there exist preprocessing SNARKs. Then there exist:*

(i) SNARKs with no preprocessing.

(ii) PCDs for compliance predicates of constant depth with no preprocessing.

(iii) Path PCDs for compliance predicates of fixed polynomial depth with no preprocessing.

The above holds for both the publicly-verifiable and designated-verifier cases. In the private-verification case, we assume the existence of a fully-homomorphic encryption scheme; in both cases we assume the existence of collision-resistant hash functions.

We have already shown how the second part and third part of the theorem follow from the first part: we can simply invoke the SNARK Recursive Composition Theorem (discussed in Section 2.2) to obtain (ii) from (i), and then the PCD Bootstrapping Theorem (discussed in Section 2.3) to obtain (iii) from (ii). We are thus left to show the first part of the theorem: removing preprocessing from a SNARK.

To remove preprocessing, we follow the plan outlined at the beginning of Section 2.4. That is, using the RAM Compliance Theorem, we reduce the task of verifying SNARK statements to the task of verifying the compliance of multiple "small" computations along a path. Then we use the SNARK Recursive Composition and PCD Bootstrapping Theorems to enforce compliance of path computations. As a basis for the SNARK Recursive Composition Theorem, we use a pre-processing SNARK that only needs to account for "small computations" that are bounded by a fixed polynomial in the security parameter. A bit more concretely, it should account for the time required to compute compliance at each one of the nodes, plus the overhead of SNARK verification, which is again a fixed polynomial in the security parameter.

For a diagram summarizing how our main theorems come together, see Figure 2. More details can be found in Section 10.

Existing pre-processing SNARKs. The theorem above gives a compiler that takes as input preprocessing SNARKs and outputs a preprocessing-free SNARK (or even certain PCD systems if desired). Let us briefly discuss what constructions are known that we can plug into this compiler; for more details, see Section 9.

- Groth [Gro10] constructed publicly-verifiable SNARKs with preprocessing; his construction relies on a variant of the *Knowledge of Exponent* assumption [Dam92, BP04], which he calls q-PKEA, and a computational Diffie-Hellman assumption, which he calls q-CPDH, in bilinear groups.

Plugging Groth's construction into our compiler, we obtain, based on the same assumptions (and the existence of collision-resistance hash functions), the *first* construction, in the plain model, of publicly-verifiable SNARKs (a.k.a. "CS proofs" [Mic00]) and PCDs (for compliance predicates of constant depth, or polynomial depth for paths). Surprisingly, the resulting construction does not invoke the PCP Theorem and only involves quite simple techniques.

In summary, thanks to our machinery that shows how to generically remove preprocessing from any SNARK, we have thus shown how to leverage quite simple techniques, such as techniques based on pairings [Gro10], that at first sight seem to necessarily give rise to only preprocessing solutions, in order to obtain much more powerful solutions with *no* preprocessing. The resulting constructions, while certainly not trivial, seem to be quite simple, and do not invoke along the way additional probabilistic-checking machinery. We find this (pleasantly) surprising.

2.6 Applications

In order to further exemplify the power of compliance engineering, we show how proof-carrying data implies targeted malleability [BSW11] and a variant of computing on authenticated data [BF11, ABC$^+$11]. We are able to obtain for the first time, starting with a simple knowledge assumption, targeted malleability (with no preprocessing) and computing on authenticated data for any functionality in the plain model for any distributed computation of constant depth or paths of polynomial length.

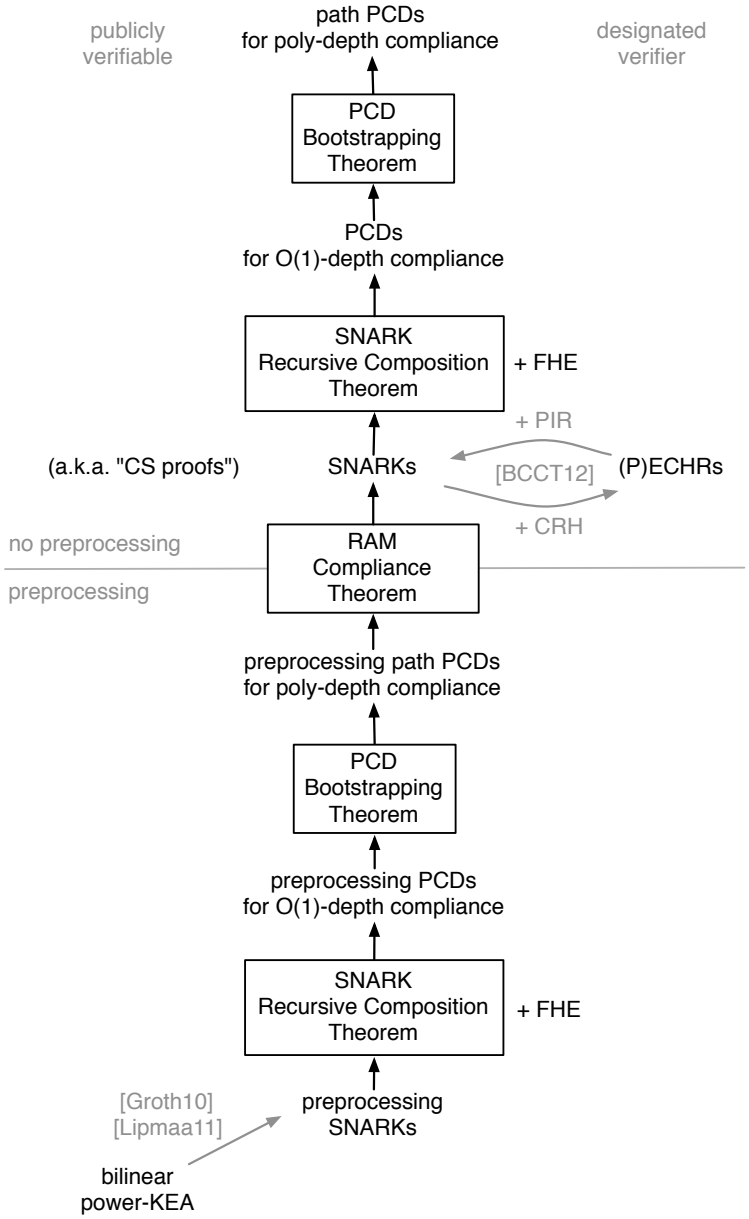

Figure 2: Summary of how our three main results come together; also see Section 2.5 for a high-level discussion. Solid gray arrows are results that were previously known or folklore. Dotted gray arrows are trivial implications.

3 Bounded-Halting Problems and Random-Access Machines

We define the universal relation and its language (along with related relations), which will us express, throughout the rest of this paper, the membership problems that we will be interested in. We also informally discuss random-access machines, the choice of abstract machine that we adopt, because they are a natural and convenient model for expressing one of our results.

3.1 Universal Relation and NP Relations

The *universal relation* [BG08] is defined to be the set $\mathcal{R}_{\mathcal{U}}$ of instance-witness pairs (y, w), where $y = (M, x, t)$, $|w| \leq t$, and M is an abstract machine (e.g., a Turing machine), such that M accepts (x, w) after at most t steps. While the witness w for each instance $y = (M, x, t)$ is of size at most t, there is *no a-priori* polynomial bounding t in terms of $|x|$. We denote by $\mathcal{L}_{\mathcal{U}}$ the *universal language* corresponding to $\mathcal{R}_{\mathcal{U}}$. At high level, we can say that deciding membership in the universal relation of a given instance (M, x, t) is a (non-deterministic) *bounded-halting problem* on the machine M.

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 $\mathcal{R}_c$.

3.2 Random-Access Machines

We choose random-access machines [CR72, AV77] to be the abstract machine used in the universal relation $\mathcal{R}_{\mathcal{U}}$; in the context of this paper, these will be more convenient to discuss than Turing machines.[3] We do not formally define random-access machines, but we provide an informal description of their operation.

A (two-tape) random-access machine M consists of a vector of *registers* and a vector of n *instructions*. Instructions may consist of operations to modify memory (such as load and store instructions), operations to read the next symbol on either of the two tapes, and a set of arithmetic / branch operations to operate on registers and possibly alter the program flow (by changing the *program counter*). We call δ_M the function that, on input values for the program counter, the register, and a value from memory (in case the program counter points to a load instruction), executes one step of computation of M and outputs the new values for the program counter and registers.

Note that in order to (directly) check whether $\big((M, x, t), w\big) \in \mathcal{R}_{\mathcal{U}}$, one has to run the machine M on input (x, w) for at most t steps. More precisely, running the machine entails

(i) *code consistency:* correctly executing its transition function at every step of computation (on input the current values of the program counter, registers, and appropriate memory value, in order to produce the new values for the program counter and registers), *and*

(ii) *memory consistency:* correctly maintaining its memory (i.e., for memory access, the value returned must be equal to the last written value to that memory address).

We also define a "weak" universal relation $\mathcal{R}'_{\mathcal{U}}$ (which in fact is a strict superset of $\mathcal{R}_{\mathcal{U}}$) corresponding to the case where the memory of the machine is "not trusted", that is, memory may return inconsistent values. (In particular, if a machine cares about meaningfully using memory, it must somehow verify on its own the

[3] Concretely, we build on a computational Levin reduction of Ben-Sasson et al. [BSCGT12] that shows how to delegate to untrusted storage the memory of a random-access machine.

returned values; see Section 7.1.) More precisely, witnesses in the universal relation $\mathcal{R}'_{\mathcal{U}}$ consist of pairs (w, m), where the i-th component of m is interpreted as the i-th value returned from untrusted memory; that is, $\big((M, x, t), (w, (m_i)_{i=1}^t)\big) \in \mathcal{R}'_{\mathcal{U}}$ if and only if M accepts (x, w) after at most t steps when responding to the memory read of step i (if there is one) with value m_i. In other words, $(M, x, t) \in \mathcal{L}'_{\mathcal{U}}$, where $\mathcal{L}'_{\mathcal{U}}$ is the language corresponding to $\mathcal{R}'_{\mathcal{U}}$, if and only if there is some run of $M(x, w)$ with untrusted memory that accepts (code consistency of an accepting run suffices for membership).

4 SNARKs

A *succinct non-interactive argument* (SNARG) is a triple of algorithms $(\mathcal{G}, \mathcal{P}, \mathcal{V})$ that works as follows. The (probabilistic) generator $\mathcal{G}$, on input the security parameter k, outputs 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 witness w; then $\mathcal{V}(\tau, y, \pi)$ verifies the validity of π. The SNARG is *adaptive* if the prover may choose the statement *after* seeing the σ, otherwise, it is *non-adaptive*.

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

1. *Completeness*

   For any $(y, w) \in \mathcal{R}_{\mathcal{U}}$,

   $$\Pr_{(\sigma,\tau)\leftarrow\mathcal{G}(1^k)}[\mathcal{V}(\tau, y, \pi) = 1 \mid \pi \leftarrow \mathcal{P}(\sigma, y, w)] = 1 \ .$$

   In addition, $\mathcal{P}(\sigma, y, w)$ runs in time $\mathrm{poly}(k, |y|, t)$.

2. *Soundness*

   Depending on the notion of adaptivity:

   - ***Non-adaptive soundness.*** *For all poly-size prover $\mathcal{P}^*$, large enough $k \in \mathbb{N}$, and $y \notin \mathcal{L}_{\mathcal{U}}$,*

     $$\Pr_{(\sigma,\tau)\leftarrow\mathcal{G}(1^k)}[\mathcal{V}(\tau, y, \pi) = 1 \mid \pi \leftarrow \mathcal{P}^*(\sigma, y)] \leq \mathrm{negl}(k) \ .$$

   - ***Adaptive soundness.*** *For all poly-size prover $\mathcal{P}^*$ and large enough $k \in \mathbb{N}$,*

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

3. *Succinctness*

   There exist a universal polynomial p such that, for any instance $y = (M, |x|, t)$ and large enough security parameter k,

   - *The running time of the prover $\mathcal{P}$ is at most $p(k, t)$.*
   - *The running time of the verifier $\mathcal{V}$ is at most $p(k + |y|) = p(k + |M| + |x| + \log t)$, when verifying an honestly generated proof.*
   - *The length of an honestly generated proof is $p(k)$.*

For the process of generating the reference string and the verification state, we consider two variants:

- ***"Full" succinctness.***
  The generator $\mathcal{G}(1^k)$ runs in time $p(k)$ (independently of the computation), and in particular the generated (σ, τ) are of size at most $p(k)$.
- ***Succinctness up to preprocessing.***
  The verifier reference string may depend on a pre-given time bound B for the computation (which may not be bounded by any fixed polynomial in the security parameter). In this case, the running time of $\mathcal{G}$ and the length of the reference string σ that it outputs may be $p(k, B)$.

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

Definition 4.2. *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 any poly-size prover $\mathcal{P}^*$ there exists a poly-size extractor $\mathcal{E}_{\mathcal{P}^*}$ such that for all large enough $k \in \mathbb{N}$ and all auxiliary inputs $z \in \{0,1\}^{\mathrm{poly}(k)}$,

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

Universal arguments vs. universal succinctness and non-universal soundness. The SNARG (and SNARKs) given in Definitions 4.1 and 4.2 are for the universal relation $\mathcal{R}_{\mathcal{U}}$ and are called *universal arguments*.[5] A weaker notion that we will also consider is a SNARG (or a SNARK) for a specific NP relation $\mathcal{R}_c \subset \mathcal{R}_{\mathcal{U}}$. In this case, soundness is only required to hold with respect to $\mathcal{R}_c$; in particular, the verifier algorithm may depend on c . Nevertheless, we require (and achieve) *universal succinctness*, where a universal polynomial p, independent of c, upper bounds the length of every proof and the verification time.

Remark 4.3 (Public vs. designated verification and security in the presence of a verification oracle)**.** We distinguish between the case where the verifier state τ may be public or needs to remain private. Specifically, a *publicly-verifiable SNARK* (pvSNARK for short) is one where security holds even if τ is public; in contrast, a *designated-verifier SNARK* (dvSNARK for short) is one where τ needs to remain secret.

One property that is greatly effected by the above distinction, is whether security (i.e., soundness or proof of knowledge) also holds when the malicious prover is given access to a verification oracle $\mathcal{V}(\tau, \cdot, \cdot)$. This property is essential in settings where the same verification state (and corresponding reference string) are used multiple times, and the verifier's responses to given proofs may leak and taken advantage of by a malicious prover. For pvSNARKs, this requirement is automatically guaranteed; for dvSNARKs, however, soundness in the presence of a verification oracle needs to be required explicitly as an additional property. It can be seen that given limited number of $O(\log k)$-verifications, soundness will inherently hold; the "non-trivial" case is when the number of verifications learned is $\omega(\log k)$ or even not bounded by any specific polynomial. (In the delegation regime, the issue is often circumvented by assuming that the verifier's responses remain secret, or re-generating σ, τ every logarithmically-many verifications, e.g., as in [KR06, Mie08, GKR08, KR09, GGP10, CKV10].)

[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 require the extractor to be an implicit representation of a witness.

Remark 4.4 (Assumptions on the reference string)**.** Depending on the model and required properties, there may be different trust assumptions about who runs $\mathcal{G}(1^k)$ to obtain (σ, τ) and publish σ and make sure the verifier has access to τ.

For example, in a dvSNARK, the verifier itself may run $\mathcal{G}$ 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. But if we further require the property of *zero-knowledge*, then we must assume that σ is a *common* reference string, and thus that a trusted party ran $\mathcal{G}$.

As another example, in a pvSNARK, the verifier may once again run $\mathcal{G}$ and then publish σ, though other verifiers if they do not trust him may have to run on their own $\mathcal{G}$ to obtain outputs that they trust; alternatively, we could assume that σ is a *global reference string* that everyone trusts. Once again, though, if we further require the property of zero-knowledge, then we must assume that σ is a *common* reference string.

The SNARK extractor $\mathcal{E}$. Above, we require that any poly-size family of circuits $\mathcal{P}^*$ has a specific poly-size family of extractors $\mathcal{E}_{\mathcal{P}^*}$; in particular, we allow the extractor to be of arbitrary poly-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 poly-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 SNARKs, 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 poly-size non-uniform adversaries, the non-uniform notion seems to suffice. In our case, going from a knowledge assumption to SNARKs, the notion of extraction will be preserved: if you start with uniform extraction you will get SNARK with uniform extraction.

Extraction for multiple theorems. In this work we will be interested in performing recursive extraction along tree structures; in particular, we will be interested in provers that may produce a vector of theorems $\vec{y}$ together with a vector of corresponding proofs $\vec{\pi}$. In such cases, it will be more convenient to apply corresponding extractors that can extract a vector of witnesses $\vec{w}$ for each of the proofs that is accepted by the SNARK verifier. This notion of extraction can be shown to follow from the single-witness extraction as defined above.

Lemma 4.5 (adaptive proof of knowledge for multiple inputs)**.** *Let $(\mathcal{P}, \mathcal{G}, \mathcal{V})$ be a SNARK (as in Definition 4.2), then for any multi-theorem poly-size prover $\mathcal{P}^*$ there exists a poly-size extractor $\mathcal{E}_{\mathcal{P}^*}$ such that for all large enough security parameter $k \in \mathbb{N}$ and any auxiliary input $z \in \{0,1\}^{\mathrm{poly}(k)}$,*

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

5 Proof-Carrying Data Systems

We formally *introduce proof-carrying data (PCD) systems*, which are the cryptographic primitive that formally captures the framework for proof-carrying data.

5.1 Compliance of Computation

We begin by specifying the (syntactic) notion of a distributed computation that is considered in proof-carrying data. Essentially, we view a distributed computation simply as a directed acyclic graph with edge labels (representing messages) and node labels (representing the program allegedly run at the node).

Definition 5.1. *A* **distributed computation transcript** *is a triple* $\mathsf{T} = (G, \mathsf{lprog}, \mathsf{data})$ *representing a directed acyclic graph* G *where each vertex* v *corresponds to a local program* $\mathsf{lprog}(v)$*, and each edge* (u, v) *corresponds to a message* $\mathsf{data}(u, v)$ *sent between* u *and* v*. Typically, the local program of* v *takes as input the message* $\mathsf{data}(u, v)$ *and some additional local input, and outputs a message* $\mathsf{data}(v, w)$ *for the following vertex* w*. Each source vertex has a single outgoing edge, carrying an input to the distributed computation (there are no programs at sources). The final outputs of the distributed computation is the data carried along edges going into sinks.*

A proof-carrying transcript is a transcript where the messages are augmented by proof strings. (This is a syntactic definition; the semantics are discussed in Section 5.2.)

Definition 5.2. *A* **proof-carrying transcript** $\mathsf{PCT} = (\mathsf{T}, \mathsf{proof})$ *augments a distributed computation transcript* T *with a corresponding proof string* $\mathsf{proof}(e)$ *for any edge* e*.*

Next, we define what it means for a distributed computation to be *compliant*, which is the notion of "correctness with respect to a given specification". Compliance is captured via an efficiently computable predicate $\mathbb{C}$ that should be locally satisfied in each vertex with respect to the the local program and the incoming and outgoing data.

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

$$\mathbb{C}(\mathsf{outdata}(v); \mathsf{lprog}(v), \mathsf{indata}(v)) = 1 \ ,$$

where $\mathsf{indata}(v)$ *and* $\mathsf{outdata}(v)$ *denote all incoming and outgoing messages corresponding* v *and* $\mathsf{lprog}(v)$ *is the local program at* v*.*

Remark 5.4 (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 $\mathbb{C}(\mathsf{z_o}; \vec{\mathsf{z}}_\mathsf{i}, \mathsf{prog})$ is bounded by $\mathrm{poly}_\mathbb{C}(|\mathsf{z_o}|)$, for a fixed $\mathrm{poly}_\mathbb{C}$ that depends only on $\mathbb{C}$. This, in particular, implies that inputs where $|\mathsf{prog}| + |\vec{\mathsf{z}}_\mathsf{i}| > \mathrm{poly}_\mathbb{C}(|\mathsf{z_o}|)$ are rejected. This restriction shall often simplify presentation, and the relevant class of compliance predicates is expressive enough for most applications that come to mind.

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

Definition 5.5. *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 PCD Systems

A *proof-carrying data (PCD) system* for a compliance predicate $\mathbb{C}$ is a triple of algorithms $(\mathbb{G}, \mathbb{P}_{\mathbb{C}}, \mathbb{V}_{\mathbb{C}})$ 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. The (honest) prover $\mathbb{P}_{\mathbb{C}}$ is given σ, inputs $\vec{\mathsf{z}}_{\mathsf{i}}$ with corresponding proofs $\vec{\pi}_{\mathsf{i}}$, a program prog, and an output z_{o} (corresponding to $(\mathsf{indata}(v), \mathsf{inproof}(v), \mathsf{lprog}(v))$ and an output in $\mathsf{outdata}(v)$ for some vertex v); $\mathbb{P}_{\mathbb{C}}$ then produces a proof π_{o} for the output z_{o} being consistent with some $\mathbb{C}$-compliant transcript. The verifier $\mathbb{V}_{\mathbb{C}}$ is given the verification state τ, an output z_{o}, and a proof string π_{o}, and is meant to accept only when it is convinces that the output is consistent with some $\mathbb{C}$-compliant transcript.

Using these algorithms, a distributed computation transcript T is compiled into a proof-carrying transcript PCT by generating and adding a proof to each edge. The process of generating proofs is defined inductively, by having each (internal) vertex v in the transcript T use the PCD prover $\mathbb{P}_{\mathbb{C}}$ to produce a proof π_{o} for each output z_{o} being consistent with a $\mathbb{C}$-compliant transcript.

More precisely, given a transcript $\mathsf{T} = (G, \mathsf{lprog}, \mathsf{data})$, we define a process $\mathsf{PrfGen}(\sigma, \mathsf{T})$ that assigns a proof for all edges in $E(G)$ as follows. All edges outgoing from sources are assigned empty proofs $\perp$. Next, moving up the graph from sources to sinks, each non source edge (u, v) is assigned a proof as follows. Let $\vec{\mathsf{z}}_{\mathsf{i}} = \mathsf{indata}(v)$ be v's incoming messages, let $\vec{\pi}_{\mathsf{i}} = \mathsf{inproof}(v)$ be its incoming proofs, let $\mathsf{prog} = \mathsf{lprog}(v)$ be its local program, and let $\mathsf{z}_{\mathsf{o}} = \mathsf{data}(u, v)$ be the message going from u to v. The proof π_{o} assigned to (u, v) is $\pi_{\mathsf{o}} = \mathbb{P}_{\mathbb{C}}(\sigma, \mathsf{z}_{\mathsf{o}}, \mathsf{prog}, \vec{\mathsf{z}}_{\mathsf{i}}, \vec{\pi}_{\mathsf{i}})$. We denote by $\mathsf{PrfGen}(\sigma, \mathsf{T}, (u, v))$ the proof assigned to (u, v) by this process. Above, we simplify exposition by assuming that the computation transcript T is generated independently of the proofs. This can be naturally extended to also consider computation transcripts that are dynamically generated, also depending on previous proofs (see Remark 5.7).

Running time and succinctness. Given a distributed computation graph G, we associate with each vertex v a time bound $t(v)$, which is intuitively is the time required for the local program lprog to perform its computation (and more generally a bound on the running time of the compliance predicate). We denote by k the security parameter. We require that the running time of prover $\mathbb{P}_{\mathbb{C}}$, when producing a proof at a given node v, is a fixed polynomial in $t(v)$ and k. The length of a any proof generated by $\mathbb{P}_{\mathbb{C}}$ is a fixed polynomial in k (and, essentially, independent of $t(v)$). The running time of the verifier $\mathbb{V}_{\mathbb{C}}$ is a fixed polynomial (depending on $\mathbb{C}$) in z_{o} and k. The running time of the generator $\mathbb{G}$ is ideally a fixed polynomial in the security parameter; in particular, both the reference string σ and the verification state τ are polynomial in k. However, we shall also consider PCDs with *preprocessing*. Here, $\mathbb{G}$ is also given a time bound B and may produce a reference string σ that depends on B; we still require though that the verification state τ is succinct (namely, it is a fixed polynomial in the security parameter.)

We now formally define the notion of PCDs.

Definition 5.6. *A $\mathbb{C}$-compliance* **proof-carrying data system** *is a triple of algorithms $(\mathbb{G}, \mathbb{P}_{\mathbb{C}}, \mathbb{V}_{\mathbb{C}})$, where $\mathbb{G}$ is probabilistic and $\mathbb{P}_{\mathbb{C}}$, $\mathbb{V}_{\mathbb{C}}$ are deterministic, that satisfies the following conditions:*

1. *Completeness*

   For every $\mathbb{C}$-compliant transcript $\mathsf{T} = (G, \mathsf{lprog}, \mathsf{data})$, edge $(u, v) \in E(G)$, and message $\mathsf{z}_{\mathsf{o}} = \mathsf{data}(u, v)$,

$$\Pr_{(\sigma,\tau)\leftarrow\mathbb{G}(1^k)} [\mathbb{V}_{\mathbb{C}}(\tau, \mathsf{z}_{\mathsf{o}}, \pi_{\mathsf{o}}) = 1 \mid \pi_{\mathsf{o}} \leftarrow \mathsf{PrfGen}(\sigma, \mathsf{T}, (u, v))] \geq 1 - \mathrm{negl}(k) \ .$$

2. *Proof of Knowledge*

   For every polynomial-size prover $\mathbb{P}^*$, there exists a polynomial-size knowledge-extractor $\mathbb{E}_{\mathbb{P}^*}$, such that for any large enough security parameter k and all auxiliary inputs $z \in \{0,1\}^{\mathrm{poly}(k)}$:

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

   where T_{z} is a transcript with a single sink edge (u,s) and corresponding message $\mathsf{data}(u,s)=\mathsf{z}$.

3. *Succinctness*

   Given a distributed computation transcript $\mathsf{T}=(G,\mathsf{lprog},\mathsf{data})$, we denote by $t_{\mathbb{C}}(v)$ the time required to run the compliance predicate on input $(\mathsf{indata}(v),\mathsf{lprog}(v),\mathsf{outdata}(v))$ for any vertex $v\in V(G)$ (and assume that, for any v, $t_{\mathbb{C}}(v)\leq k^{\log k}$). We require that there exist a fixed polynomial p (independently of any specific transcript T), such that for large enough security parameter k:

   - *The running time of the prover $\mathbb{P}_{\mathbb{C}}$ at any node v is at most $p(k,t_{\mathbb{C}}(v))$.*
   - *The length of any proof π produced by $\mathbb{P}_{\mathbb{C}}$ is at most $p(k)$.*
   - *The running time of the verifier $\mathbb{V}_{\mathbb{C}}$, when verifying $(\mathsf{z},\pi)=(\mathsf{proof}(e),\mathsf{data}(e))$ for any edge $e\in E(G)$ is at most $p(k,|\mathsf{z}|)$ (independently of $t_{\mathbb{C}}(v)$).*

   For the process of generating the reference string and the verification state, we consider two variants:

   - ***"Full" succinctness.***
     The generator $\mathbb{G}(1^k)$ runs in time $p(k)$ (independently of the computation), and in particular the generated (σ,τ) are of size at most $p(k)$.
   - ***Succinctness up to preprocessing.***
     The verifier reference string may depend on a pre-given time bound B for the computation (which may not be bounded by any fixed polynomial in the security parameter). In this case, $\mathbb{G}$ may output a reference string σ of size at most $p(k,B)$). In this model, proof and verification for any node v is only done so long that $t_{\mathbb{C}}(v)\leq B$ (in which case, the running times of the prover and verifier are the same as above).

We shall also consider a restricted notion of PCD system: a **path PCD system** *is a PCD system where completeness is only guaranteed to hold for distributed computations that evolve over paths (i.e., line graphs).*

Remark 5.7 (Proof-dependent transcripts)**.** Formally, Definition 5.6 only captures the case where the computation transcript T is independent of the proofs to be generated by the PCD system, in the sense that the entire proof generation procedure is done for some pre-existing transcript T. We can of course consider a more general definition where the computation transcript T (including data and local programs) is dynamically generated along with the proofs (and may potentially depend on these). For the sake of simplicity, we restrict attention to the simpler definition, but our constructions satisfy the more general definition as well.

Remark 5.8 (Public vs. designated verification and security in the presence of a verification oracle)**.** As was the case with SNARKs (see Remark 4.3), we distinguish between the case where the verifier state τ may be public or needs to remain private. Specifically, a *publicly-verifiable PCD system* (pvPCD for short) is one where security holds even if τ is public. In contrast, a *designated-verifier PCD system* (dvPCD for short) is one where τ needs to remain secret. Similarly to SNARKs, this affects whether security holds in the presence of a verification oracle: in the pvPCD case this property is automatically guaranteed, while in the dvPCD case this it does not follow directly (besides as usual the trivial guarantees for $O(\log k)$ verifications).

Remark 5.9. In Definition 5.6 we could have allowed the adversary to choose the compliance predicate $\mathbb{C}$ for which he chooses the "final output" z. And indeed 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.

6 A Recursive Composition Theorem for All Kinds of SNARKs

We provide here the technical details for the high-level discussion in Section 2.2.

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 made to compose, which may come as a surprise.)

Formally:

Theorem 6.1 (SNARK Recursive Composition Theorem)**.** *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. More precisely:*

(i) If there exist publicly-verifiable SNARKs, then there exist publicly-verifiable PCD systems for every constant-depth compliance predicate.

(ii) Assuming the existence of FHE, if there exist designated-verifier SNARKs, then there exist designated-verifier PCD systems for every constant-depth compliance predicate.[6]

Moreover, in either of the above statements, the SNARK can be of the preprocessing kind, in which case so will the PCD system.

We begin in Section 6.1 by discussing the construction for the publicly-verifiable case, and then in Section 6.2 we discuss the construction for the designated-verifier case.

Remark 6.2 (bootstrapping)**.** Constant-depth compliance predicates are not all that weak! Indeed, as we will show in Section 8, the depth of a compliance predicate can always be improved exponentially, via the PCD Bootstrapping 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.

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:

[6] We do not require the verification state to be reusable. If it happens to be, then this property will be preserved by our construction.

- 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 PCDs 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 $\log_w(\mathrm{poly}(k))$ and only allows w inputs per node, then it has polynomial size.

- 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 PCDs 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.

  (An alternative way to obtain PCDs for polynomial-size compliance predicates is to strengthen the extractability assumption to an "interactive online extraction"; see, e.g., [BP04, DFH11] for examples of such assumptions. For example, the kind of extraction that Damgård et al. [DFH11] 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 PCDs 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.[7]

6.1 Composition of Publicly-Verifiable SNARKs

We begin by proving the SNARK Recursive Composition Theorem for the publicly-verifiable case with no preprocessing (i.e., proving Item (i) of Theorem 6.1 with no preprocessing). We shall later extend the discussion to the case with preprocessing (and then to the designated-verifier case in Section 6.2).

The construction. We follow the PCD construction of Chiesa and Tromer [CT10]. At high-level, at each node in the distributed computation, the PCD prover invokes the SNARK prover to generate a proof attesting to the fact that the node knows input messages that are compliant with the claimed output message, *and also* that he knows corresponding proofs attesting that these input messages themselves come from a compliant distributed computation. The PCD verifier simply invokes the SNARK verifier (on an appropriate statement).

We are given two ingredients:

[7] 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 we found in constructing *universal* SNARKs in [BCCT11] and not just SNARKs for specific languages. Chiesa and Tromer [CT10] were able to construct PCDs for *any* 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 PCDs for any compliance predicate in the plain model.

- A compliance predicate $\mathbb{C}$ of constant depth $\mathsf{d}(\mathbb{C})$. (See Definition 5.5.)
- A publicly-verifiable SNARK $(\mathcal{G}, \mathcal{P}, \mathcal{V})$ for the relation $\mathcal{R}_c$ (as defined in Section 3.1), where c is a constant that depends on the running time of the compliance predicate $\mathbb{C}$, as well as on a universal constant, given by the SNARK verification time. We explain later on how to choose this parameter; for now, we advise the the reader to think of $(\mathcal{G}, \mathcal{P}, \mathcal{V})$ as a SNARK for the universal relation $\mathcal{R}_{\mathcal{U}}$.

We construct the corresponding PCD system as follows.

The PCD generator. Given as input the security parameter 1^k, the PCD generator $\mathbb{G}$ simply runs the SNARK generator $\mathcal{G}(1^k)$ in order to create a reference string σ and verification state τ, both of size that is a fixed polynomial in the security parameter. We assume that both (σ, τ) include the security parameter 1^k in the clear. Also, because we are focusing on the publicly-verifiable case, we may assume that σ includes τ in the clear. (Moreover, in the publicly-verifiable case with no preprocessing that we are currently discussing, σ and τ can in fact be merged, but we shall keep them separate for the sake of clarity and exposition of the other cases.)

The PCD prover and the PCD machine. Given input $(\sigma, \mathsf{z_o}, \mathsf{prog}, \vec{\mathsf{z}}_\mathsf{i}, \vec{\pi}_\mathsf{i})$, the PCD prover $\mathbb{P}_\mathbb{C}$ 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 "outgoing" proof π_o.

More precisely, $y := \mathsf{y}(\mathsf{z})$ and $w := (\mathsf{prog}, \vec{\mathsf{z}}_\mathsf{i}, \vec{\pi}_\mathsf{i})$, where $\mathsf{y}(\mathsf{z}) = \mathsf{y}(\mathsf{z}, \tau, k)$ is the SNARK statement $(\mathsf{M}^\mathbb{C}, (\mathsf{z}, \tau), t^k_{\mathsf{M}^\mathbb{C}}(|\mathsf{z}|))$ and $\mathsf{M}^\mathbb{C}$ is a fixed machine called *the PCD machine* (it only depends on the compliance predicate $\mathbb{C}$ and the SNARK verifier $\mathcal{V}$).

The PCD machine $\mathsf{M}^\mathbb{C}$ takes an input x and a witness w where $x = (\mathsf{z_o}, \tau, k)$ and $w = (\mathsf{prog}, \vec{\mathsf{z}}_\mathsf{i}, \vec{\pi}_\mathsf{i})$. (Note that we use here the fact that the verification state τ is public and given to the prover.) Then, $\mathsf{M}^\mathbb{C}$ verifies that: (a) the message $\mathsf{z_o}$ is $\mathbb{C}$-compliant with the local program prog and input messages $\vec{\mathsf{z}}_\mathsf{i}$, and (b) each $\pi \in \vec{\pi}_\mathsf{i}$ is a valid SNARK proof attesting to the consistency of a corresponding $\mathsf{z} \in \vec{\mathsf{z}}_\mathsf{i}$ with a previous $\mathbb{C}$-compliant distributed computation.

The formal description of the machine $\mathsf{M}^\mathbb{C}$ is given in Figure 3.

Input and witness. $(x, w) = ((\mathsf{z_o}, \tau, k), (\mathsf{prog}, \vec{\mathsf{z}}_\mathsf{i}, \vec{\pi}_\mathsf{i}))$, where $\vec{\mathsf{z}}_\mathsf{i}$ represent input messages, $\vec{\pi}_\mathsf{i}$ corresponding compliance proofs, prog a local program, $\mathsf{z_o}$ an output message, τ the SNARK verification state, and k the security parameter.

Time bound. If the machine $\mathsf{M}^\mathbb{C}$ reaches a time bound $t^k_{\mathsf{M}^\mathbb{C}}(|\mathsf{z_o}|)$ it halts and rejects. The function $t^k_{\mathsf{M}^\mathbb{C}}(|\mathsf{z}|)$ is a fixed polynomial in $(|\mathsf{z}|, k, |\langle\mathbb{C}\rangle|)$. (We explain how this function is chosen in the parameters paragraph below.)

Base case. If $(\mathsf{prog}, \vec{\mathsf{z}}_\mathsf{i}, \vec{\pi}_\mathsf{i}) = \perp$, 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 computation.)

General case. Verify:

- *Compliance of the current node*: $\mathbb{C}(\mathsf{z_o}; \mathsf{prog}, \vec{\mathsf{z}}_\mathsf{i}) = 1$.
- *Compliance of the past:* For each input $\mathsf{z} \in \vec{\mathsf{z}}_\mathsf{i}$ and corresponding proof $\pi \in \vec{\pi}_\mathsf{i}$ verify that $\mathcal{V}(\tau, \mathsf{y}(\mathsf{z}), \pi) = 1$, where $\mathsf{y}(\mathsf{z}) = \mathsf{y}(\mathsf{z}, \tau, k)$ is the SNARK statement $(\mathsf{M}^\mathbb{C}, (\mathsf{z}, \tau), t^k_{\mathsf{M}^\mathbb{C}}(|\mathsf{z}|))$. (We now think of each z as an output of a previous distributed computation carrying a proof π attesting to its $\mathbb{C}$-compliance.)

If any of the above fails, output reject.

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

The PCD verifier. Given input $(\tau, \mathsf{z_o}, \pi_\mathsf{o})$, the PCD verifier $\mathbb{V}_\mathbb{C}$ simply runs the SNARK verifier $\mathcal{V}$, with verification state τ, input statement $(M, x, t) = (\mathsf{M}^\mathbb{C}, (\mathsf{z_o}, \tau), t^k_{\mathsf{M}^\mathbb{C}}(|\mathsf{z_o}|))$, and proof π_o.

Parameters and succinctness. We explain how the time bound function $t^k_{\mathsf{M}^\mathbb{C}}$ is set. The main part of the computation of the PCD machine $\mathsf{M}^\mathbb{C}$ is, typically, verifying $\mathbb{C}$-compliance at the local node, namely, verifying that $\mathbb{C}(\mathsf{z_o}; \mathsf{prog}, \vec{\mathsf{z}}_\mathsf{i}) = 1$; since $\mathbb{C}$ is polynomially balanced (see Remark 5.4), the time to perform this check is at most $\mathrm{poly}_\mathbb{C}(|\mathsf{z_o}|)$, where $\mathrm{poly}_\mathbb{C}$ is a fixed polynomial depending only on $\mathbb{C}$. But the PCD machine also needs to verify $\mathbb{C}$-compliance of the past, and the time required to do so depends on the running time of the SNARK verification algorithm $\mathcal{V}$. Specifically, running each of the SNARK verifiers requires:

$$\sum_{\mathsf{z} \in \vec{\mathsf{z}}_\mathsf{i}} t_\mathcal{V}\Big(k, |\mathsf{z}| + |\langle \mathbb{C} \rangle|, \log\left(t^k_{\mathsf{M}^\mathbb{C}}(|\mathsf{z}|)\right)\Big) \ ,$$

where $t_\mathcal{V}$ is the running time bound on the SNARK verifier, which is a fixed (universal) polynomial. We can assume that all computations are bounded by some super-polynomial function in the security parameter, say $k^{\log k}$, and hence can bound $t^k_{\mathsf{M}^\mathbb{C}}(|\mathsf{z}|)$ by $k^{\log k}$. Hence, because $\log t^k_{\mathsf{M}^\mathbb{C}}(|\mathsf{z}|) \leq \log^2 k$, the succinctness of the SNARK verifier implies that the overhead of verifying each input $\mathsf{z} \in \vec{\mathsf{z}}_\mathsf{i}$ is a fixed polynomial in k and $|\mathsf{z}| + |\langle \mathbb{C} \rangle|$, independently of the actual computation time $t^k_{\mathsf{M}^\mathbb{C}}(|\mathsf{z}|)$. By the fact that $\mathbb{C}$ is balanced we also deduce that the number of nodes z summed over is a fixed polynomial in $|\mathsf{z_o}|$, i.e., $|\vec{\mathsf{z}}_\mathsf{i}| \leq \mathrm{poly}_\mathbb{C}(|\mathsf{z_o}|)$. Hence, we deduce that the above sum is also bounded by $\mathrm{poly}_\mathbb{C}(k, |\mathsf{z_o}|)$, where $\mathrm{poly}_\mathbb{C}$ is a fixed polynomial depending only on $\mathbb{C}$.

Overall, there is a fixed $\mathrm{poly}_\mathbb{C}$ that depends only on $\mathbb{C}$ (and the SNARK in use), such that:

$$t^k_{\mathsf{M}^\mathbb{C}}(|\mathsf{z_o}|) \leq \mathrm{poly}_\mathbb{C}(k, |\mathsf{z_o}|) \ .$$

Having established this bound, we can choose accordingly the relation $\mathcal{R}_c$ for which the SNARK is sound. (Note that the succinctness of the SNARK is universal and independent of c, so there is no issue of circularity here.)

We now show that $(\mathbb{P}_\mathbb{C}, \mathbb{G}, \mathbb{V}_\mathbb{C})$ is a $\mathbb{C}$-compliance (publicly-verifiable) PCD system assuming that $\mathbb{C}$ is of constant depth $\mathsf{d}(\mathbb{C})$.

The completeness of the system follows directly from the underlying SNARK. We concentrate on the adaptive proof of knowledge property. Our goal is to construct for any possibly malicious prover $\mathbb{P}^*$ a corresponding extractor $\mathbb{E}_{\mathbb{P}^*}$ such that, when $\mathbb{P}^*$ convinces $\mathbb{V}_\mathbb{C}$ of the $\mathbb{C}$-compliance of a message $\mathsf{z_o}$, the extractor can (efficiently) find a $\mathbb{C}$-compliant transcript to "explain" why the verifier accepted. For this purpose we employ a natural recursive extraction strategy, which we now describe.

The extraction procedure. Given a poly-size prover $\mathbb{P}^*$, we derive $\mathsf{d}(\mathbb{C})$ circuit families of extractors, one for each potential level of the distributed computation. To make notation lighter, we do not explicitly write the auxiliary input z which might be given to $\mathbb{P}^*$ and its extractor (e.g., any random coins used by $\mathbb{P}^*$); this is also done for the SNARK provers and their extractors discussed below. Doing so is all right because what we are going to prove holds also with respect to any auxiliary input distribution $\mathcal{Z}$, assuming the underlying SNARK is secure with respect to the same auxiliary input distribution $\mathcal{Z}$.

We start by defining $\mathcal{E}_1 := \mathcal{E}_{\mathcal{P}^*_1}$ to be the SNARK extractor for the SNARK prover $\mathcal{P}^*_1$ that, given σ, computes $(\mathsf{z}_1, \pi_1) \leftarrow \mathbb{P}^*(\sigma)$, constructs the instance $y_1 := (\mathsf{M}^\mathbb{C}, (\mathsf{z}_1, \tau_1), t^k_{\mathsf{M}^\mathbb{C}}(|\mathsf{z}_1|))$, and outputs (y_1, π_1). Like $\mathcal{P}^*_1$, $\mathcal{E}_1$ also expects input σ; $\mathcal{E}_1$ returns a string $(\vec{\mathsf{z}}_2, \vec{\pi}_2, \mathsf{prog}_1)$ that should be a valid witness for the SNARK statement y_1, assuming that $\mathbb{V}_\mathbb{C}$ (and hence also $\mathcal{V}$) accepts π_1.

Next, we augment $\mathcal{E}_1$ into a new SNARK prover $\mathcal{P}_2^*$ that, given σ, outputs multiple statements $\vec{y}_2$ and corresponding proofs $\vec{\pi}_2$, where for each $\mathsf{z} \in \vec{\mathsf{z}}_2$ (now viewed as an output message) there is an entry $\mathsf{y}(\mathsf{z}) := (\mathsf{M}^{\mathbb{C}}, (\mathsf{z}, \tau), t^k_{\mathsf{M}^{\mathbb{C}}}(|\mathsf{z}|))$ in $\vec{y}_2$. We can thus consider a new extractor $\mathcal{E}_2 := \mathcal{E}_{\mathcal{P}_2^*}$ for $\mathcal{P}_2^*$ that, given σ, should output a witness for each convincing proof and statement $(y, \pi) \in (\vec{y}_2, \vec{\pi}_2)$.

In general, for each $1 \leq j \leq \mathsf{d}(\mathbb{C})$ we define $\mathcal{P}_j^*$ and $\mathcal{E}_j := \mathcal{E}_{\mathcal{P}_j^*}$ according to the above rule.

Overall, the witness extractor $\mathbb{E}_{\mathbb{P}^*}$ operates as follows. Given as input the reference string σ, it constructs a transcript T for a directed tree, by applying each one of the extractors $\mathcal{E}_j$ defined above. Each such extractor produces a corresponding level in the computation tree: each witness $(\vec{\mathsf{z}}, \vec{\pi}, \mathsf{prog})$ (among the multiple witnesses) extracted by $\mathcal{E}_j$ corresponds to a node v on the j-th level of the tree, with local program $\mathsf{lprog}(v) := \mathsf{prog}$, and incoming messages $\mathsf{indata}(v) := \vec{\mathsf{z}}$ from its children. The tree has a single sink edge (s, s') with a corresponding message $\mathsf{data}(s, s') := \mathsf{z}_1$, which is the output of $\mathbb{P}^*$. The leaves of the tree are set to be the vertices for which the extracted witnesses are $(\mathsf{prog}, \vec{\mathsf{z}}, \vec{\pi}) = \bot$.[8]

Note that the the final (composed) extractor is of polynomial size, as the number of recursive levels is constant, and hence even an arbitrary polynomial blowup in the size of the extractor (relative to the corresponding prover) can be tolerated.

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

Proposition 6.4. *Let $\mathbb{P}^*$ be a polynomial size malicious PCD prover, and let $\mathbb{E}_{\mathbb{P}^*}$ be its corresponding extractor as defined above. Then, for any large enough security parameter $k \in \mathbb{N}$:*

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

where $\mathsf{T}_{\mathsf{z}_1}$ is a transcript with a single sink edge (s, s') and corresponding message $\mathsf{data}(s, s') = \mathsf{z}_1$.

Proof. The proof is by induction on the level of the extracted tree; recall that there is a constant number $\mathsf{d}(\mathbb{C})$ of levels all together. As 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 $(\vec{\mathsf{z}}_2, \vec{\pi}_2, \mathsf{prog}_1)$ such that:

1. Compliance holds, namely, $\mathbb{C}(\mathsf{z}_1; \vec{\mathsf{z}}_2, \mathsf{prog}_1) = 1$.
2. For each $(\mathsf{z}, \pi) \in (\vec{\mathsf{z}}_2, \vec{\pi}_2)$, $\mathcal{V}(\tau, \mathsf{y}(\mathsf{z}), \pi) = 1$, where $\mathsf{y}(\mathsf{z})$ is the SNARK statement $(\mathsf{M}^{\mathbb{C}}, (\mathsf{z}, \tau), t^k_{\mathsf{M}^{\mathbb{C}}}(|\mathsf{z}|))$.

Indeed, the fact that $\mathbb{V}_{\mathbb{C}}$ accepts $(\tau, \mathsf{z}_1, \pi_1)$ implies that the SNARK verifier $\mathcal{V}$ accepts $\mathsf{y}(\mathsf{z}_1) = (\mathsf{M}^{\mathbb{C}}, (\mathsf{z}_1, \tau), t^k_{\mathsf{M}^{\mathbb{C}}}(|\mathsf{z}_1|))$, and hence, by the SNARK proof of knowledge property, the extractor $\mathcal{E}_1$ will output (except with negligible probability) a valid witness $(\vec{\mathsf{z}}_2, \vec{\pi}_2, \mathsf{prog}_2)$ for the statement $\mathsf{y}(\mathsf{z}_1)$. This, in particular, means that the extracted witness satisfies both the required properties (as verified by the PCD machine $\mathsf{M}^{\mathbb{C}}$).

To complete the proof, we can continue inductively and prove in the same manner the compliance of each level in the extracted distributed computation tree. Specifically, assume that for each extracted node v in level $1 \leq j < \mathsf{d}(\mathbb{C})$, the second property holds; namely, for each $(\mathsf{z}, \pi) \in (\vec{\mathsf{z}}_{c(v)}, \vec{\pi}_{c(v)})$, it holds that $\mathcal{V}(\tau, \pi, \mathsf{y}(\mathsf{z})) = 1$, where $(\vec{\mathsf{z}}_{c(v)}, \vec{\pi}_{c(v)})$ denotes the incoming messages and proofs from v's children, extracted by $\mathcal{E}_j$, and $\mathsf{y}(\mathsf{z})$ is the SNARK statement $(\mathsf{M}^{\mathbb{C}}, (\mathsf{z}, \tau), t^k_{\mathsf{M}^{\mathbb{C}}}(|\mathsf{z}|))$. This, in turn, implies that, with all but negligible probability, for any of child $u \in c(v)$, the extractor $\mathcal{E}_{j+1}$ outputs a valid witness $(\vec{\mathsf{z}}_{c(u)}, \vec{\pi}_{c(u)},$

[8] 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 conducting the proof for super-constant $\mathsf{d}(\mathbb{C})$ starting from stronger extractability assumptions; see Remark 6.3.)

prog_u) for the statement $\mathsf{y}(\mathsf{z}_u)$, where $\mathsf{z}_u \in \vec{\mathsf{z}}_{c(v)}$ is the message corresponding to u. This, in turn, means that u satisfies both required properties:

1. Compliance, namely, $\mathbb{C}(\mathsf{z}_u; \vec{\mathsf{z}}_{c(u)}, \mathsf{prog}_u) = 1$.

2. For each $(\mathsf{z}, \pi) \in (\vec{\mathsf{z}}_{c(u)}, \vec{\pi}_{c(u)})$, $\mathcal{V}(\tau, \mathsf{y}(\mathsf{z}), \pi) = 1$, where $\mathsf{y}(\mathsf{z})$ is the SNARK statement $(\mathsf{M}^{\mathbb{C}}, (\mathsf{z}, \tau), t^k_{\mathsf{M}^{\mathbb{C}}}(|\mathsf{z}|))$.

Thus we can complete the induction and deduce compliance of the tree transcript T. □

The-preprocessing case. We now describe how to adapt the above to the case where the input SNARK is of the preprocessing kind. In such SNARKs, one has to pre-specify a time bound $B = B(k)$ when generating the reference string σ and corresponding verification state τ. In this case, the SNARK will only allow proving and verifying statements of the form (M, x, t) where $|M| + |x| + t < B$; statements that do not meet this criteria will be automatically rejected by the verifier. (Note that this differs from SNARKs for a relation $\mathcal{R}_c$ that do allow t to grow proportionally to $|x|^c$, but not faster than $|x|^c$.) Still, in preprocessing SNARKs, the verifier's running time is required to be as succinct as in SNARKs with no preprocessing and, in particular, independent of B (this succinctness is crucial in our context).

When using such SNARKs to construct PCD systems, the preprocessing constraint translates to a preprocessing constraint on the PCD. (Recall Item 3 of Definition 5.6.) That is, we can only construct PCD systems where the generator $\mathbb{G}$ is also given a time bound B, which is passed on as a time bound for the underlying SNARK (up to polynomial factors in the security parameter). Here, PCD proving and verifying is only carried out as long as the running time of the PCD machine $t^k_{\mathsf{M}^{\mathbb{C}}}(|\mathsf{z}|)$ is at most B. This essentially means that the PCD will work only as long as the computation and proof verification done at each node are bounded by B.[9] We stress again that, the SNARK verification time (and hence also $t^k_{\mathsf{M}^{\mathbb{C}}}$) are independent of B. Other than the latter restriction on the running time $t^k_{\mathsf{M}^{\mathbb{C}}}$, using essentially the same construction as before, we immediately obtain preprocessing PCDs.

A particularly useful special case is when $t^k_{\mathsf{M}^{\mathbb{C}}}$ happens to be bounded by a fixed polynomial in the security parameter; namely, not only is the SNARK verification time bounded by some $\mathrm{poly}(k)$, but also all the inputs and computation time of $\mathbb{C}$ are also bounded by some fixed $\mathrm{poly}(k)$:

Corollary 6.5 (preprocessing PCDs for fixed-polynomial computations)**.** *Let $\mathbb{C}$ be any compliance predicate of constant depth $\mathsf{d}(\mathbb{C}) = O(1)$, and assume that its running time $t_{\mathbb{C}}$ is bounded by a fixed polynomial in the security parameter. Then for any preprocessing SNARK, instantiating the PCD machine $\mathsf{M}^{\mathbb{C}}$ with the SNARK verifier, there is a fixed bound $p(k)$ on the running time $t^k_{\mathsf{M}^{\mathbb{C}}}$ of $\mathsf{M}^{\mathbb{C}}$. In particular, we can set accordingly $B(k) := p(k)$, and obtain a corresponding PCD system with preprocessing time that is only a fixed polynomial in the security parameter.*

Ultimately, one of the results in this paper will show that we can always get rid of preprocessing in a SNARK, and it is in the proof of this result that we will invoke the above corollary. (See high-level discussion in Section 2.5.)

Having dealt with the publicly-verifiable case, in the next subsection, we proceed to discuss the designated-verifier case.

[9] We emphasize that the bound B applies to the computation of a *single* node, and not the entire distributed computation!

6.2 Composition of Designated-Verifier SNARKs

We now show that (perhaps surprisingly) we can also compose designated-verifier SNARKs (dvSNARKs) to obtain designated-verifier PCDs for constant-depth compliance predicates.[10] That is, we prove the SNARK Recursive Composition Theorem for the designated-verifier case (i.e., Item (ii) of Theorem 6.1).

As before, we describe the construction for SNARKs with no preprocessing first; the extension to the preprocessing case is analogous to the discussion for the publicly-verifiable case of preprocessing, which can be found at the end of the previous section.

The construction. When we try to adapt the PCD construction for the publicly-verifiable case 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 (because it is secret) and, therefore, cannot provide a witness for such a theorem.

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 verification circuit $\mathcal{V}$ 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 construction.

The two ingredients now are:

- A compliance predicate $\mathbb{C}$ of constant depth $\mathsf{d}(\mathbb{C})$. (See Definition 5.5.)
- A designated-verifier SNARK $(\mathcal{G}, \mathcal{P}, \mathcal{V})$ for the relation $\mathcal{R}_c$ (as defined in Section 3.1), where c is a constant that depends on the running time of the compliance predicate $\mathbb{C}$, as well as on a universal constant, given by the SNARK verification time *and* homomorphic evaluation time overhead. We explain later on how to choose this parameter; for now, we advise the the reader to think of $(\mathcal{G}, \mathcal{P}, \mathcal{V})$ as a SNARK for the universal relation $\mathcal{R}_{\mathcal{U}}$.

We construct the corresponding PCD system as follows.

The PCD generator. Given as input the security parameter 1^k, the PCD generator $\mathbb{G}$ runs the SNARK generator $\mathcal{G}(1^k)$ in order to create a reference string σ and verification state τ, both of size that is a fixed polynomial in the security parameter. Then, $\mathbb{G}$ samples a secret key sk and an evaluation key ek for the FHE scheme, and computes an encryption c^τ of the secret verification state. The (private) verification state is then set to be $\tilde{\tau} := (\tau, \mathsf{sk})$, and the reference string is set to be $\tilde{\sigma} := (\sigma, \mathsf{c}^\tau)$ (and for simplicity of notation we assume that c^τ also includes the evaluation key ek). We assume throughout that both (σ, τ) include the security parameter 1^k in the clear.

[10] 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 Remark 5.8.

The PCD prover and the PCD machine. As in the publicly-verifiable case, the heart of the PCD system construction is the design of the PCD machine $\mathsf{M}^{\mathbb{C}}$. We define the PCD machine required for the current construction in Figure 4 and then proceed to describe how the prover uses it to generate PCD proofs.

Input and witness. $(x, w) = ((\mathsf{z_o}, \hat{\mathsf{c}}_\mathsf{o}, \mathsf{c}^\tau, k), (\mathsf{prog}, \vec{\mathsf{z}}_\mathsf{i}, \vec{\pi}_\mathsf{i}, \vec{\hat{\mathsf{c}}}_\mathsf{i}))$, where $\vec{\mathsf{z}}_\mathsf{i}$ represents input messages, $\vec{\pi}_\mathsf{i}$ corresponding compliance proofs and $\vec{\hat{\mathsf{c}}}_\mathsf{i}$ corresponding evaluated verification bits, prog a local program, $\mathsf{z_o}$ an output message, $\hat{\mathsf{c}}_\mathsf{o}$ an output evaluated verification bit, c^τ an encrypted SNARK verification state, and k the security parameter.

Time bound. If the machine $\mathsf{M}^{\mathbb{C}}$ reaches a time bound $t^k_{\mathsf{M}^{\mathbb{C}}}(|\mathsf{z_o}|)$ it halts and rejects. The function $t^k_{\mathsf{M}^{\mathbb{C}}}(|\mathsf{z}|)$ is a fixed polynomial in $(|\mathsf{z}|, k, |\langle\mathbb{C}\rangle|)$. (We explain how this function is chosen in the parameters paragraph below.)

Base case. If $(\mathsf{prog}, \vec{\mathsf{z}}_\mathsf{i}, \vec{\pi}_\mathsf{i}, \vec{\hat{\mathsf{c}}}_\mathsf{i}) = \bot$, 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 computation.)

General case. Verify:

- *Compliance of the current node:* $\mathbb{C}(\mathsf{z_o}; \mathsf{prog}, \vec{\mathsf{z}}_\mathsf{i}) = 1$.
- *Complianc of the past:* Verify that the cipher $\hat{\mathsf{c}}_\mathsf{o}$ "aggregates the correctness of the previous computation". That is,

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

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

$$\hat{\mathsf{c}}_\mathcal{V} = \mathsf{Eval}_{\mathsf{ek}}(\mathcal{V}(\cdot, \mathsf{y}(\mathsf{z}, \hat{\mathsf{c}}), \pi), \mathsf{c}^\tau) \ ,$$

  where $\mathsf{y}(\mathsf{z}, \hat{\mathsf{c}}) = \mathsf{y}(\mathsf{z}, \hat{\mathsf{c}}, \mathsf{c}^\tau, k)$ is the SNARK statement $(\mathsf{M}^{\mathbb{C}}, (\mathsf{z}, \hat{\mathsf{c}}, \mathsf{c}^\tau), t^k_{\mathsf{M}^{\mathbb{C}}}(|\mathsf{z}|))$.

If any of the above fails, output reject.

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

Given input $(\sigma, \mathsf{c}^\tau, \mathsf{z_o}, \mathsf{prog}, \vec{\mathsf{z}}_\mathsf{i}, \vec{\pi}_\mathsf{i}, \vec{\hat{\mathsf{c}}}_\mathsf{i})$, the PCD prover $\mathbb{P}_\mathbb{C}$ performs two main computations: first, it 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 $\vec{\hat{\mathsf{c}}}_\mathsf{i}$. That is, it 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 $\hat{\mathsf{c}}_\mathcal{V} \in \vec{\hat{\mathsf{c}}}_\mathcal{V}$ corresponds to $(\mathsf{z}, \pi, \hat{\mathsf{c}}) \in (\vec{\mathsf{z}}_\mathsf{i}, \vec{\pi}_\mathsf{i}, \vec{\hat{\mathsf{c}}}_\mathsf{i})$ and is the result of homomorphically evaluating the SNARK verifier on the proof π and the statement $\mathsf{y}(\mathsf{z}, \hat{\mathsf{c}}) = \mathsf{y}(\mathsf{z}, \hat{\mathsf{c}}, \mathsf{c}^\tau, k) = (\mathsf{M}^{\mathbb{C}}, (\mathsf{z}, \hat{\mathsf{c}}, \mathsf{c}^\tau), t^k_{\mathsf{M}^{\mathbb{C}}}(|\mathsf{z}|))$ attesting to the compliance of z with a previous compliant computation and the consistency of $\hat{\mathsf{c}}$ with a previous homomorphic evaluation. That is, $\hat{\mathsf{c}}_\mathcal{V} = \mathsf{Eval}_{\mathsf{ek}}(\mathcal{V}(\cdot, \mathsf{y}(\mathsf{z}, \hat{\mathsf{c}}), \pi), \mathsf{c}^\tau)$.

After computing $\hat{\mathsf{c}}_\mathsf{o}$, the PCD prover runs the SNARK prover $\mathcal{P}(\sigma, y, w)$ with the instance $y := \mathsf{y}(\mathsf{z_o}, \hat{\mathsf{c}}_\mathsf{o})$ and witness $w := (\mathsf{prog}, \vec{\mathsf{z}}_\mathsf{i}, \vec{\pi}_\mathsf{i}, \vec{\hat{\mathsf{c}}}_\mathsf{i})$, to produce the "outgoing" proof π_o.

The PCD verifier. Given input $(\tau, \mathsf{z_o}, \pi_\mathsf{o}, \hat{\mathsf{c}}_\mathsf{o})$, the PCD verifier $\mathbb{V}_\mathbb{C}$: (a) verifies that the SNARK verifier $\mathcal{V}$, with verification state τ, accepts the statement $(\mathsf{M}^{\mathbb{C}}, \mathsf{y}(\mathsf{z_o}, \hat{\mathsf{c}}_\mathsf{o}), t^k_{\mathsf{M}^{\mathbb{C}}}(|\mathsf{z_o}|))$ with proof π_o, and (b) verifies that $\mathsf{Dec}_{\mathsf{sk}}(\hat{\mathsf{c}}_\mathsf{o}) = 1$.

Parameters and succinctness. We explain how the time bound function $t^k_{\mathsf{M}^{\mathbb{C}}}$ is set. As before, the main part of the computation of the PCD machine $\mathsf{M}^{\mathbb{C}}$ is, typically, verifying $\mathbb{C}$-compliance at the local node, namely,

verifying that $\mathbb{C}(\mathsf{z_o}; \mathsf{prog}, \vec{\mathsf{z}}_\mathsf{i}) = 1$; since $\mathbb{C}$ is polynomially balanced (see Remark 5.4), the time to perform this check is at most $\mathrm{poly}_\mathbb{C}(|\mathsf{z_o}|)$, where $\mathrm{poly}_\mathbb{C}$ is a fixed polynomial depending only on $\mathbb{C}$. But now this verification is conducted under the encryption.

Thus, (homomorphically) checking $\mathbb{C}$-compliance of the past depends on the time to homomorphically evaluate the SNARK verification algorithm and the time to homomorphically aggregate the various encrypted bits. Specifically, performing the homomorphic evaluation of the SNARK verifier requires:

$$\sum_{\mathsf{z} \in \vec{\mathsf{z}}_\mathsf{i}} t_{\mathsf{Eval}_\mathsf{ek}}\left(k, t_\mathcal{V}\left(k, |\mathsf{z}| + |\langle \mathbb{C} \rangle|, \log\left(t^k_{\mathsf{M}^\mathbb{C}}(|\mathsf{z}|)\right)\right)\right) \quad ,$$

where $t_{\mathsf{Eval}_\mathsf{ek}}(k, T)$ is the time required to homomorphically evaluate a T-time function and $t_\mathcal{V}$ is the running time bound on the SNARK verifier, which is a fixed (universal) polynomial. We can assume that all computations are bounded by some super-polynomial function in the security parameter, such as $k^{\log k}$, and hence can bound $t^k_{\mathsf{M}^\mathbb{C}}(|\mathsf{z}|)$ by $k^{\log k}$. Hence, because $\log t^k_{\mathsf{M}^\mathbb{C}}(|\mathsf{z}|) \leq \log^2 k$, the succinctness of the SNARK verifier implies that the overhead of verifying each input $\mathsf{z} \in \vec{\mathsf{z}}_\mathsf{i}$ is a fixed polynomial in k and $|\mathsf{z}| + |\langle \mathbb{C} \rangle|$, independently of the actual computation time $t^k_{\mathsf{M}^\mathbb{C}}(|\mathsf{z}|)$. By the fact that $\mathbb{C}$ is balanced we also deduce that the number of nodes z summed over is a fixed polynomial in $|\mathsf{z_o}|$, i.e., $|\vec{\mathsf{z}}_\mathsf{i}| \leq \mathrm{poly}_\mathbb{C}(|\mathsf{z_o}|)$. Hence, we deduce that the above sum is bounded by $\mathrm{poly}_\mathbb{C}(k, |\mathsf{z_o}|)$, where $\mathrm{poly}_\mathbb{C}$ is a fixed polynomial depending only on $\mathbb{C}$.

We are left to note that the homomorphic multiplication (for aggregating the encrypted bits) is done over a number of encryptions, each of fixed size in the security parameter k, that is at most twice the number of components in the vector $\vec{\hat{\mathsf{c}}}_\mathsf{i}$, which is at most twice the number of components in the vector $\vec{\mathsf{z}}_\mathsf{i}$, which is at most $\mathrm{poly}_\mathbb{C}(|\mathsf{z_o}|)$.

We conclude that there is a fixed $\mathrm{poly}_\mathbb{C}$ that depends only on $\mathbb{C}$ (and the FHE and SNARK in use), such that:

$$t^k_{\mathsf{M}^\mathbb{C}}(|\mathsf{z_o}|) \leq \mathrm{poly}_\mathbb{C}(k, |\mathsf{z_o}|) \quad .$$

Having established this bound, we can choose accordingly the relation $\mathcal{R}_c$ for which the SNARK is sound. (Note that the succinctness of the SNARK is universal and independent of c, so there is no issue of circularity here.)

We now show that $(\mathbb{P}_\mathbb{C}, \mathbb{G}, \mathbb{V}_\mathbb{C})$ is a $\mathbb{C}$-compliance (designated-verifier) PCD system assuming that $\mathbb{C}$ is of constant depth $\mathsf{d}(\mathbb{C})$.

As before, the completeness of the system follows directly from the underlying SNARK. We concentrate on the adaptive proof of knowledge property. And again our goal is to construct for any possibly malicious prover $\mathbb{P}^*$ a corresponding extractor $\mathbb{E}_{\mathbb{P}^*}$ such that, when $\mathbb{P}^*$ convinces $\mathbb{V}_\mathbb{C}$ of the $\mathbb{C}$-compliance of a message $\mathsf{z_o}$, the extractor can (efficiently) find a $\mathbb{C}$-compliant transcript to "explain" why the verifier accepted. For this purpose we employ a natural recursive extraction strategy, similar to the one we used in the publicly-verifiable case, which we now describe.

The extraction procedure. Given a poly-size prover $\mathbb{P}^*$, we derive $\mathsf{d}(\mathbb{C})$ circuit families of extractors, one for each potential level of the distributed computation. As before, to make notation lighter, we do not explicitly write the auxiliary input z which might be given to $\mathbb{P}^*$ and its extractor (e.g., any random coins used by $\mathbb{P}^*$). When we apply SNARK extraction, however, we will need to explicitly refer to the auxiliary input it gets (in this case, an encryption of the verification state; see Remark 6.6). All mentioned implications hold also with respect any auxiliary input distribution $\mathcal{Z}$, assuming 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 ciphers $\hat{\mathsf{c}}$ of the aggregated verification bits, and the encrypted verification state c^τ.

Concretely, we start by defining $\mathcal{E}_1 := \mathcal{E}_{\mathcal{P}_1^*}$ to be the SNARK extractor for a SNARK prover $\mathcal{P}_1^*$ that is defined as follows. Given a SNARK reference string σ and the encrypted verification state c^τ as auxiliary input, $\mathcal{P}_1^*$ computes $(\mathsf{z}_1, \pi_1, \hat{\mathsf{c}}_1) \leftarrow \mathbb{P}^*(\sigma, \mathsf{c}^\tau)$, constructs a corresponding instance $y_1 := (\mathsf{M}^{\mathbb{C}}, (\mathsf{z}_1, \hat{\mathsf{c}}_1, \mathsf{c}^\tau), t^k_{\mathsf{M}^{\mathbb{C}}}(|\mathsf{z}_1|))$, and outputs (y_1, π_1). Like $\mathcal{P}_1^*$, $\mathcal{E}_1$ also expects a reference string σ and auxiliary input c^τ; $\mathcal{E}_1$ returns a string $(\mathsf{prog}_1, \vec{\mathsf{z}}_2, \vec{\pi}_2, \vec{\hat{\mathsf{c}}}_2)$ that is hopefully a valid witness for the SNARK statement y_1, assuming that $\mathbb{V}_{\mathbb{C}}$ (and hence also $\mathcal{V}$) accepts the proof π_1. (As we shall see later on, showing the validity of such a witness will require invoking semantic security, since the SNARK prover also has c^τ as auxiliary input, while the formal guarantee of extraction is only given when (σ, τ) are independent of the auxiliary input.)

Next, we augment $\mathcal{E}_1$ to a new SNARK prover $\mathcal{P}_2^*$ that, given σ and c^τ, outputs multiple statements $\vec{y}_2$ and corresponding proofs $\vec{\pi}_2$, where for each $(\mathsf{z}, \hat{\mathsf{c}}) \in (\vec{\mathsf{z}}_2, \vec{\hat{\mathsf{c}}}_2)$ there is an entry $\mathsf{y}(\mathsf{z}, \hat{\mathsf{c}}) := (\mathsf{M}^{\mathbb{C}}, (\mathsf{z}, \hat{\mathsf{c}}, \mathsf{c}^\tau), t^k_{\mathsf{M}^{\mathbb{C}}}(|\mathsf{z}|))$ in $\vec{y}_2$. We can thus consider a new extractor $\mathcal{E}_2 := \mathcal{E}_{\mathcal{P}_2^*}$ for $\mathcal{P}_2^*$ that, given $(\sigma, \mathsf{c}^\tau)$, should output a witness for each convincing proof and statement $(y, \pi) \in (\vec{y}_2, \vec{\pi}_2)$.

In general, for each $1 \leq j \leq \mathsf{d}(\mathbb{C})$ we define $\mathcal{P}_j^*$ and $\mathcal{E}_j := \mathcal{E}_{\mathcal{P}_j^*}$ according to the above rule.

Overall, the witness extractor $\mathbb{E}_{\mathbb{P}^*}$ combines the extractors $\mathcal{E}_j$ to construct the transcript T, similarly to the publicly-verifiable case. That is, given as input the reference string σ and the encrypted verification state c^τ, $\mathbb{E}_{\mathbb{P}^*}$ constructs a transcript T for a directed tree, by applying each one of the extractors $\mathcal{E}_j$ defined above. Each such extractor produces a corresponding level in the computation tree: each witness $(\vec{\mathsf{z}}, \vec{\pi}, \vec{\hat{\mathsf{c}}}, \mathsf{prog})$ (among the multiple witnesses) extracted by $\mathcal{E}_j$ corresponds to a node v on the j-th level of the tree, with local program $\mathsf{lprog}(v) := \mathsf{prog}$, and incoming messages $\mathsf{indata}(v) := \vec{\mathsf{z}}$ from its children. The tree has a single sink edge (s, s') with a corresponding message $\mathsf{data}(s, s') := \mathsf{z}_1$, which is the output of $\mathbb{P}^*$. The leaves of the tree are set to be the vertices for which the extracted witnesses are $(\vec{\mathsf{z}}, \vec{\pi}, \vec{\hat{\mathsf{c}}}, \mathsf{prog}) = \perp$. (See Footnote 8.)

As before, note that the the final (composed) extractor is of polynomial size, as the number of recursive levels is constant.

Remark 6.6 (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 ciphers (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:

Proposition 6.7. *Let $\mathbb{P}^*$ be a polynomial size malicious PCD prover, and let $\mathbb{E}_{\mathbb{P}^*}$ be its corresponding extractor as defined above. Then, for any large enough security parameter $k \in \mathbb{N}$:*

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

where $\mathsf{T}_{\mathsf{z}_1}$ is a transcript with a single sink edge (s, s') and corresponding message $\mathsf{data}(s, s') = \mathsf{z}_1$.

Proof. The proof is by induction on the level of the extracted tree; recall that there is a constant number of levels all together. As 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 $(\mathsf{prog}_1, \vec{\mathsf{z}}_2, \vec{\pi}_2, \vec{\hat{\mathsf{c}}}_2)$ such that:

1. Compliance holds, namely, $\mathbb{C}(\mathsf{z}_1; \vec{\mathsf{z}}_2, \mathsf{prog}_1) = 1$.
2. The cipher $\hat{\mathsf{c}}_1$ "aggregates" the correctness of the previous computation.
   Formally, $\hat{\mathsf{c}}_1 = \mathsf{Eval}_{\mathsf{ek}}\left(\prod, (\vec{\hat{\mathsf{c}}}_2, \vec{\hat{\mathsf{c}}}_{\mathcal{V}})\right)$, where each $\hat{\mathsf{c}}_{\mathcal{V}} \in \vec{\hat{\mathsf{c}}}_{\mathcal{V}}$ corresponds to $(\mathsf{z}, \pi, \hat{\mathsf{c}}) \in (\vec{\mathsf{z}}_2, \vec{\pi}_2, \vec{\hat{\mathsf{c}}}_2)$ and is the result of homomorphically evaluating the SNARK verifier as required; namely, $\hat{\mathsf{c}}_{\mathcal{V}} = \mathsf{Eval}_{\mathsf{ek}}(\mathcal{V}(\cdot, \mathsf{y}(\mathsf{z}, \hat{\mathsf{c}}), \pi), \mathsf{c}^{\tau})$, where $\mathsf{y}(\mathsf{z}, \hat{\mathsf{c}})$ is the statement $(\mathsf{M}^{\mathbb{C}}, (\mathsf{z}, \hat{\mathsf{c}}, \mathsf{c}^{\tau}), t^k_{\mathsf{M}^{\mathbb{C}}}(|\mathsf{z}|))$.
3. For each $\hat{\mathsf{c}} \in \vec{\hat{\mathsf{c}}}_2$, it holds that $\mathsf{Dec}_{\mathsf{sk}}(\hat{\mathsf{c}}) = 1$.
4. For each $(\mathsf{z}, \pi, \hat{\mathsf{c}}) \in (\vec{\mathsf{z}}_2, \vec{\pi}_2, \vec{\hat{\mathsf{c}}}_2)$, it holds that $\mathcal{V}(\tau, \mathsf{y}(\mathsf{z}, \hat{\mathsf{c}}), \pi) = 1$.

To show that the above indeed holds, we consider an alternative experiment where, instead of receiving the encrypted verification state c^{τ}, the prover $\mathbb{P}^*$ receives an encryption of an arbitrary string, say $0^{|\tau|}$, denoted by c^0. We show that in this case, whenever the verifier is convinced the first two conditions above conditions must hold, and then we deduce the same for the real experiment.

Indeed, since in the the alternative experiment, the SNARK prover $\mathcal{P}^*_1$ is only given c^0, which is independent of the verification state τ, the SNARK proof of knowledge guarantee kicks in. Namely, except with negligible probability, whenever $\mathbb{V}_{\mathbb{C}}$ (hence also $\mathcal{V}$) accepts, it must be that the extractor $\mathcal{E}_1$ outputs a valid witness $(\vec{\mathsf{z}}_2, \vec{\pi}_2, \vec{\hat{\mathsf{c}}}_2, \mathsf{prog}_1)$ for the statement $\mathsf{y}(\mathsf{z}_1, \hat{\mathsf{c}}_1) = (\mathsf{M}^{\mathbb{C}}, (\mathsf{z}_1, \hat{\mathsf{c}}_1, \mathsf{c}^0), t^k_{\mathsf{M}^{\mathbb{C}}}(|\mathsf{z}_1|))$ output by $\mathcal{P}^*_1$ (only this time when given c^0 rather than c^{τ}). This, in particular, means that the extracted witness satisfies the first two properties (as verified by the PCD machine $\mathsf{M}^{\mathbb{C}}$).

Next, note that the first two properties can be efficiently tested given:

$$(\mathsf{z}_1, \pi_1, \hat{\mathsf{c}}_1, \vec{\mathsf{z}}_2, \vec{\pi}_2, \vec{\hat{\mathsf{c}}}_2, \mathsf{prog}_1, \mathsf{c}^0) \ ,$$

simply by running (the deterministic) algorithms $\mathbb{C}$ and $\mathsf{Eval}_{\mathsf{ek}}$ (and note that neither sk nor τ are required for such a test). We can thus deduce that in the real experiment where $\mathcal{P}^*_1$ and $\mathcal{E}_1$ are given c^{τ}, the two first properties hold with all but negligible probability; otherwise, we can break the semantic security of the encryption, by distinguishing encryptions of a random τ from encryptions of $0^{|\tau|}$.

Now having established the two first properties, we can deduce the last two properties from the second. Indeed, since the statement $\mathsf{y}(\mathsf{z}_1, \hat{\mathsf{c}}_1)$ is accepted by the verifier, we know that $\mathsf{Dec}_{\mathsf{sk}}(\hat{\mathsf{c}}_1) = 1$. This and the correctness of $\mathsf{Eval}_{\mathsf{ek}}$ implies that all ciphers in $(\vec{\hat{\mathsf{c}}}_2, \vec{\hat{\mathsf{c}}}_{\mathcal{V}})$ must *also* decrypt to "1".[11] Hence, we deduce the third property (all ciphers 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 $(\mathsf{z}, \pi, \hat{\mathsf{c}}) \in (\vec{\mathsf{z}}_2, \vec{\pi}_2, \vec{\hat{\mathsf{c}}}_2)$, it holds that $\mathcal{V}(\tau, \mathsf{y}(\mathsf{z}, \hat{\mathsf{c}}), \pi) = 1$).

To complete the proof, we can continue inductively and prove in the same manner the compliance of each level in the extracted computation tree. That is, assuming that properties three and four are satisfied by the j-th level of the tree, we can deduce properties one and two for level $j + 1$. This is done by first establishing them in an alternative experiment where c^{τ} is replaced by c^0, and then invoking semantic security to deduce the same for the real experiment. We then deduce, from the second property, the last two properties as we did for the base case. □

[11] We can assume without loss of generality that all ciphers are decrypted to either "0" or "1", either by choosing FHE where any cipher can be interpreted as such, or by adding simple consistency checks to the evaluated circuit.

7 RAM Compliance Theorem

We provide here the technical details for the high-level discussion in Section 2.4.

In this section we prove the RAM Compliance Theorem; it will be useful to keep in mind the definitions from Section 3 (where the universal language $\mathcal{L}_{\mathcal{U}}$ is introduced and random-access machines are discussed informally). Recall that the RAM Compliance Theorem gives a simple proof of the second statement of the PCD Linearization Theorem (Theorem 2.1), and, when combined with the PCD Bootstrapping Theorem (discussed in Section 8), gives a simple proof of the second statement of the PCD Linearization Theorem (Theorem 2.1) and is one of the tools we use in the proof of our main result, discussed in Section 10.

We show that membership in $\mathcal{L}_{\mathcal{U}}$ of an instance $y = (M, x, t)$ can be computationally reduced to the question of whether there is a distributed computation compliant with $\mathbb{C}^h_y$, which has depth at most $t \cdot \mathrm{poly}(k)$ and a small computation of size $\mathrm{poly}(|y|, k)$ at each node, for a collision-resistant hash function h.

Formally:

Theorem 7.1 (RAM Compliance Theorem)**.** *Let $\mathcal{H}$ be a CRH function family. There exist deterministic polynomial-time transformations $\Phi, \Psi_0, \Psi_1 \colon \{0,1\}^* \to \{0,1\}^*$ such that, for every compliance predicate $\mathbb{C}$, the following properties hold:*

1. Completeness: *For every $(y, w) \in \mathcal{R}_{\mathcal{U}}$ and $h \in \mathcal{H}$ it holds that $\mathbb{C}^h_y(\mathsf{T}_w) = 1$, where $\mathbb{C}^h_y := \Phi(\mathbb{C}, y, h)$ is a compliance predicate and $\mathsf{T}_w := \Psi_0(\mathbb{C}, y, w, h)$ is a distributed computation transcript.*

2. Soundness: *For every polynomial-size circuit C and large enough security parameter $k \in \mathbb{N}$:*

$$\Pr_{h \leftarrow \mathcal{H}} \left[\begin{array}{c} \mathbb{C}^h_y(\mathsf{T}) = 1 \\ \textit{and} \\ \big(y, \Psi_1(\mathbb{C}, y, \mathsf{T})\big) \notin \mathcal{R}_{\mathcal{U}} \end{array} \middle| \; \big(y, \mathsf{T}\big) \leftarrow C(h) \right] \leq \mathrm{negl}(k) \ .$$

3. Efficiency: *For every y and $h \in \mathcal{H}$ it holds that $\mathsf{d}(\mathbb{C}^h_y) \leq t \cdot \mathrm{poly}(k)$; moreover, if $(y, w) \in \mathcal{R}_{\mathcal{U}}$ then $\mathsf{T}_w := \Psi(\mathbb{C}, y, w, h)$ is a distributed computation whose graph is a path and the computation performed at each node is only $\mathrm{poly}(|y|, k)$ (and thus independent of t whenever, say, $t \leq k^k$).*

In other words, the RAM Compliance Theorem ensures a *computational Levin reduction* from verifying membership in $\mathcal{L}_{\mathcal{U}}$ to verifying the correctness of certain distributed computations. Recall that a Levin reduction is simply a Karp (instance) reduction that comes with witness reductions going "both ways"; in the theorem statement, the instance reduction is Ψ, the "forward" witness reduction is Φ_0, and the "backward" witness reduction is Φ_1; in our case the soundness guarantee provided by Ψ is only computational.

When invoking the reduction for a given instance y and then using a PCD system for $\mathbb{C}^h_y$, Φ_0 preserves the completeness property of the PCD prover, and Φ_1 makes sure that we do not break the proof of knowledge property of the PCD verifier. We stress that, in order to be able to use the RAM Compliance Theorem, the PCD system must have a proof of knowledge property, as is clear from the soundness guarantee as stated in the theorem above. (Also see the proof of Theorem 10.1 for more details.)

As discussed in Section 2.4, the proof of the RAM Compliance Theorem divides into two steps, respectively discussed in the next two subsections (Section 7.1 and Section 7.2).

Remark 7.2 (optimization)**.** Of course, for *some* machine computations, the delegation of memory performed by the RAM Compliance Theorem is not needed: for example, for computations that do not use

much memory (say much less than $O(k)$ where k is the security parameter), we can directly reduce the correctness of computation of the machine to the correctness of a distributed computation where each node does not perform much work (and the compliance predicate for this distributed computation will be a simple modification of the one given in the proof of the RAM Compliance Theorem). This optimization may turn out to be useful in practice for certain machine computations.

7.1 Machines with Untrusted Memory

We are ultimately interested to decide whether a given instance (M, x, t) is in the universal language $\mathcal{L}_{\mathcal{U}}$ or not. Ben-Sasson et al. [BSCGT12] showed that this question can be "computationally simplified" by exhibiting a *computational Levin reduction* to the weaker language $\mathcal{L}'_{\mathcal{U}}$. We briefly recall their result.

Because membership in the weaker language $\mathcal{L}'_{\mathcal{U}}$ does not require the memory to behave honestly, the computational Levin reduction will essentially modify the machine M into a new machine M' that checks its own memory, by dynamically maintaining a Merkle tree over untrusted storage. Of course, the running time t' of the new machine M' will be larger than the old one, but only by a logarithmic factor in the memory size (up to polynomial factors in the security parameter due to the computation time of the hash function).

Formally:

Lemma 7.3 (RAM Untrusted Memory Lemma [BSCGT12])**.** *Let $\mathcal{H} = \{\mathcal{H}_k\}_{k\in\mathbb{N}}$ be a CRH function family. There exists a deterministic polynomial-time function $f\colon \{0,1\}^* \to \{0,1\}^*$ such that:*

1. Syntax: *For every random-access machine M, $k \in \mathbb{N}$, $h \in \mathcal{H}_k$, $f(h, M)$ is a random-access machine.*
2. Completeness: *There exists a function $B\colon \mathbb{N}^2 \to \mathbb{N}$ such that for every $(M, x, t) \in \mathcal{L}_{\mathcal{U}}$, $k \in \mathbb{N}$, and $h \in \mathcal{H}_k$, $(f(h, M), x, B(t, k)) \in \mathcal{L}'_{\mathcal{U}}$; $B(t, k) := t \cdot k^d$ for sufficiently large d will suffice.*

   Moreover, given w such that $\big((M, x, t), w\big) \in \mathcal{R}_{\mathcal{U}}$, it is possible to compute in polynomial time $m = (m_1, \ldots, m_{B(t,k)})$ such that $\big((f(h, M), x, B(t, k)), (w, m)\big) \in \mathcal{R}'_{\mathcal{U}}$.
3. Soundness: *For every polynomial-size circuit family $\{C_k\}_{k\in\mathbb{N}}$ and sufficiently large $k \in \mathbb{N}$,*

$$\Pr_{h\leftarrow\mathcal{H}_k}\left[\begin{array}{c} \big((f(h,M),x,B(t,k)),(w,m)\big) \in \mathcal{R}'_{\mathcal{U}} \\ \textit{and} \\ \big((M,x,t),w\big) \notin \mathcal{L}_{\mathcal{U}} \end{array} \;\middle|\; \big((M,x,t),(w,m)\big) \leftarrow C_k(h) \right] \leq \mathrm{negl}(k) \ .$$

In the theorem statement we have made implicit only the instance reduction f of the Levin reduction. The "forward" witness reduction, as we will see shortly in the proof sketch, will be simple: the vector m is simply the vector of snapshots of memory when running $M(x, w)$. As for the "backward" witness reduction, it is also simple: given a valid (w, m), the reduction would simply output w. (That is, we can think of f as a "systematic reduction" in a sense similar to [BSS08, Definition 3.9], where the witness we want for the origin language is simply "embedded" into a larger witness for the destination language.)

For completeness we sketch the proof of the lemma:

Proof sketch. We first describe f and then prove its properties.

Constructing f. The main idea is to construct from M a new machine $M' := f(h, M)$ that uses the collision-resistant hash function h to delegate its memory to "untrusted storage" by dynamically maintaining a Merkle tree over it.

More precisely, the program of M' is equal to the program of M after replacing every load and store instruction with corresponding *secure load* and *secure store* operations, the register size of M' is equal to

the maximum between the register size of M and the bit-length of images of h plus 1 (to ensure that M' can actually store images of h and index into a memory that is large enough to also contain the Merkle tree), and the number of registers of M' is equal to the number of registers of M plus enough registers to compute h and then some for storing temporary variables. (In fact, the computation of M' actually begins with an initialization stage which computes a Merkle tree for a sufficiently-large all-zero memory, and then proceeds to the (modified) program of M.) Crucially, M' will always keep in a register the most up-to-date root of the Merkle tree. (Clearly, all of these modifications can be performed in polynomial time by f.)

A secure load for address i in memory will load from memory a claimed value and claimed hash, and will also load all the other relevant information for the authentication path of the i-th "leaf", in order to check the value against the (trusted) root stored in one of the machine's registers.

A secure store for address i in memory will update the relevant information for the authentication path of the i-th "leaf" and then update the (trusted) root stored in one of the machine's registers.

Secure load and secure store operations will not be atomic instructions in the new machine M' but instead will "expand" into macros consisting of instructions from the old machine M, also using (untrusted) load and store instructions in their implementation. (This ensures that in each time step we only have a single memory access, which will simplify notation later on.)

Properties of f. The syntax property of f holds by construction. The completeness property of f holds because we can choose m to be the sequence of "honest" values returned from a non-adversarial memory. The soundness property of f holds because of the collision-resistant property of h; namely, if $(M, x, t) \notin \mathcal{L}_{\mathcal{U}}$, in order for an adversary to find (w, m) showing that $(f(h, M), x, B(t)) \in \mathcal{L}'_{\mathcal{U}}$, since $f(h, M)$ checks its own memory, the adversary must find some collision of h. (Note that it is crucial that the adversary be required to *exhibit* (w, m), because surely such (w, m) very often exists but may be hard to find.) □

7.2 A Compliance Predicate for Code Consistency

The advantage of working with $\mathcal{L}'_{\mathcal{U}}$ instead of $\mathcal{L}_{\mathcal{U}}$ (which can be achieved by relying on the computational Levin reduction guaranteed by Lemma 7.3) is that we only have to worry about code consistency and not also memory consistency.

We now show how code consistency can be "broken down" into the compliance of a distributed computation (by leveraging the inherent "self-reducibility" of the notion of correct computation) containing a path with a certain property. In the resulting distributed computation, since memory need not be explicitly maintained, messages and computations at each node are quite small, and this was the purpose of working with $\mathcal{L}'_{\mathcal{U}}$ instead of $\mathcal{L}_{\mathcal{U}}$.

Intuitively, we show how for any instance $y = (M, x, t)$ it is possible to efficiently construct a compliance predicate UMC_y, which we call the *Untrusted Memory Checker* for y, that verifies computation of y one step at a time (when given values from the second tape and memory as untrusted advice) and thus serves as a way to verify membership of y in $\mathcal{L}'_{\mathcal{U}}$.

Details follow.

The mapping from y to UMC_y is formally given by the following construction:

Construction 7.4. Given an instance $y = (M, x, t)$, define the following compliance predicate:

$\mathsf{UMC}_y(\mathsf{z_o}; \vec{\mathsf{z}}_\mathsf{i}, \mathsf{prog}) \stackrel{\text{def}}{=}$

1. Verify that $\vec{\mathsf{z}}_\mathsf{i} = (\mathsf{z}_\mathsf{i})$ for some z_i.
   (That is, ensure that $\vec{\mathsf{z}}_\mathsf{i}$ is a vector consisting of a single component.)

2. If $\mathsf{z_i} = \bot$:
   (a) Verify that $\mathsf{prog} = \bot$.
   (b) Verify that $\mathsf{z_o} = (\tau', S')$ for some time stamp τ' and configuration S' of M.
   (c) Verify that S' is an initial configuration of M.
3. If $\mathsf{z_o} = \mathsf{ok}$:
   (a) Verify that $\mathsf{prog} = \bot$.
   (b) Verify that $\mathsf{z_i} = (\tau, S)$ for some time stamp τ and configuration S of M.
   (c) Verify that S is a final accepting configuration of M.
4. Otherwise:
   (a) Verify that $\mathsf{z_i} = (\tau, S)$ for some time stamp τ and configuration S of M.
   (b) Verify that $\mathsf{z_o} = (\tau', S')$ for some time stamp τ' and configuration 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 configuration S results in configuration S', with x on the first input tape of M and by supplying prog as the answer to a read to the second input tape (if a read to that tape is made) or supplying prog as the answer to a memory read (if a read to memory is made); note that there cannot be both a read to the second tape and a load in the same time step.

The first thing to note about the new compliance predicate is that its depth is bounded by the time bound:

Lemma 7.5. $\mathsf{d}(\mathsf{UMC}_y) \leq t + 1$.

Proof. Any transcript that is compliant with UMC_y consists of disjoint paths, because UMC_y forbids multiple messages to enter a node; and each of these paths, because the timestamp is forced to grow each time and is bounded by t, has depth at most $t + 1$. □

Next we prove the completeness and soundness properties of the "compliance reduction":

Claim 7.6. *$y \in \mathcal{L}'_{\mathcal{U}}$ if and only if there exists a (distributed computation) transcript T that is compliant with respect to UMC_y and contains a source-to-sink path where the data entering the sink is the string "ok".*

Moreover, we can efficiently find witnesses "both ways":

- *Given (w, m) such that $(y, (w, m)) \in \mathcal{R}'_{\mathcal{U}}$, it is possible to efficiently construct a transcript T that is compliant with respect to UMC_y containing a path ending in ok.*
- *Given a transcript T that is compliant with respect to UMC_y containing a path ending in ok, it is possible to efficiently construct (w, m) such that $(y, (w, m)) \in \mathcal{R}'_{\mathcal{U}}$.*

Proof. We need to prove two directions:

($\rightarrow$) Assume that $y \in \mathcal{L}'_{\mathcal{U}}$, and let (w, m) be such that $\big(y, (w, m)\big) \in \mathcal{R}'_{\mathcal{U}}$. Construct a transcript T as follows.

First run M on input (w, m) to deduce for each time step whether the second input tape is read or instead memory is read (or neither); let $t' \leq t$ be the number of steps that it takes for M to halt and let $S_0, \ldots, S_{t'}$ be the sequence of configurations obtained by running M. Define $a = (a_i)_{i=1}^{t'}$ so that a_i is equal to the value being read from the second input tape if in the i-th step $M(x, w)$ did read the second input tape, or equal to the value being read from memory if in the i-th step $M(x, w)$ did read memory, or an arbitrary value otherwise.

Next define $\mathsf{T} := (G, \mathsf{lprog}, \mathsf{data})$ where G is the (directed) path graph of $t' + 3$ nodes, $\mathsf{lprog}(0) := \mathsf{lprog}(t' + 1) := \mathsf{lprog}(t' + 2) := \bot$ and $\mathsf{lprog}(i) := a_i$ for $i = 1, \ldots, t'$, $\mathsf{data}(i, i + 1) := (i, S_i)$ for

$i = 0, 1, \ldots, t'$, and $\mathsf{data}(t+1, t+2) := \mathsf{ok}$. 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 time-stamped sequence of configurations of M followed by ok.

See Figure 5 for a diagram.

Because $\big(y, (w, m)\big) \in \mathcal{R}'_{\mathcal{U}}$, we know that UMC_y holds at every node of the computation and, moreover, the only path we constructed satisfies the property of having the appropriate label entering the sink.

Finally, the constructive nature of this argument also yields an efficient way to compute T from (w, m).

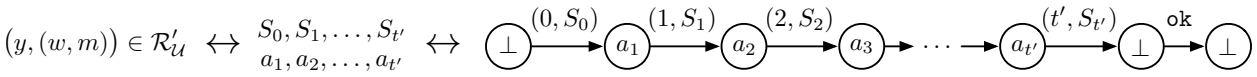

Figure 5: Constructing a UMC_y-compliant transcript starting from $\big(y, (w, m)\big) \in \mathcal{R}'_{\mathcal{U}}$, and vice versa.

($\leftarrow$) Consider any transcript T that is compliant with respect to UMC_y containing a source-to-sink path where the data entering the sink is ok.

We first note that because UMC_y never allows more than one message into a node, the graph of T is a set of disjoint paths. By assumption, there is a special path p where the label entering the sink is equal to ok.

Construct (w, m) as follows. Partition $[t']$ into three subsets I, J, K where I is the subset of those time steps in which M reads the second input tape at node i in path p, J is the subset of those time steps in which M reads memory at node i, and K is $[t'] - I - J$. Define $w := \mathsf{lprog}(I)$ and $m := \mathsf{lprog}(J)$, where the indexing is with respect to nodes in path p. Once again see Figure 5.

By compliance of T with UMC_y and because we know that the path p ends with the label ok, we know that $\big(y, (w, m)\big) \in \mathcal{R}'_{\mathcal{U}}$.

Finally, the constructive nature of this argument also yields an efficient way to compute (w, m) from T. $\square$

8 A Bootstrapping Theorem for PCD

We provide here the technical details for the high-level discussion in Section 2.3.

Recall that the SNARK Recursive Composition Theorem (Theorem 6.1) gives us PCDs only for (polynomially-balanced) compliance predicates with constant depth. The RAM Compliance Theorem (Theorem 7.1) tells us that we can (computationally) reduce membership in $\mathcal{L}_{\mathcal{U}}$ of an instance $y = (M, x, t)$ to the question of whether there is a distributed computation that has depth at most $O(t \log t)\mathrm{poly}(k)$ (and a small computation of size $\mathrm{poly}(|y|, k)$ at each node) but this depth is super-constant; it thus seems that we are not able to 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 "Bootstrapping Theorem" for PCD:

Theorem 8.1 (PCD Bootstrapping Theorem)**.** *If there exist PCDs for constant-depth compliance predicates, then there exist path PCDs for compliance predicates of fixed polynomial depth. (And the verifiability properties of the PCD carry over, as do the preprocessing properties.)*

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 compliance predicate $\textsc{Tree}_{\mathbb{C}}$ that is a "tree version" of $\mathbb{C}$.

The basic idea is that the new compliance predicate $\text{TREE}_{\mathbb{C}}$ will essentially force any compliant distributed computation to build a Merkle tree of proofs with large in-degree r (similar to the wide Merkle tree used in the security reduction of [BCCT11]) "on top" of the original distributed computation. Hence, 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 produced by the RAM Compliance Theorem), by setting $r = k$ the depth of $\text{TREE}_{\mathbb{C}}$ becomes *constant* — and we can now indeed benefit from the SNARK Recursive Composition Theorem.

For expository purposes, we begin by proving in Section 8.1 a special case of the PCD Bootstrapping Theorem for the specific compliance predicate $\mathbb{C}^h_y$ produced by the RAM Compliance Theorem (Theorem 7.1). This concrete example, where we will construct a Merkle tree of proofs on top of the step-by-step computation of a random-access machine with untrusted memory, will build the necessary intuition for the more abstract setting of the general case, which we present in Section 8.2, and is needed in one of our results.

8.1 Warm-Up Special Case

As discussed, we first prove the PCD Bootstrapping Theorem for the special case where the starting compliance predicate is $\mathbb{C}^h_y$ produced by the RAM Compliance Theorem (Theorem 7.1) for a given instance $y = (M, x, t)$.

In order to build intuition for the general case, here we will give a "non-blackbox" proof by relying on certain properties of the specific compliance predicate $\mathbb{C}^h_y$. So recall from the proof of the RAM Compliance Theorem (discussed in Section 7.1 and Section 7.6) that $\mathbb{C}^h_y = \mathsf{UMC}_{y'}$, where $\mathsf{UMC}_{y'}$ is the Untrusted Memory Checker for the instance $y' = (M', x, t')$, and $M' = f(h, M)$ and $t' = B(t, k)$).

We will construct a new compliance predicate $\text{TREE}\mathsf{UMC}_{y'}$ based on $\mathsf{UMC}_{y'}$ with exponentially better depth. The intuition of the construction is to add "metadata" to force verification of single nodes according to $\mathsf{UMC}_{y'}$ to lie at the leaves of a tree so that the only way to "exit" the tree and produce a valid final proof is to travel upwards through the tree nodes, which make sure to aggregate appropriately the metadata until the root of the tree, which produces the final proof.

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

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

1. Leaf Stage

   If $\mathsf{z_o} = (1, \mathsf{z}'_\mathsf{o})$ and $\vec{\mathsf{z}}_\mathsf{i} = \big((0, \mathsf{z}^1_\mathsf{i}), (0, \mathsf{z}^2_\mathsf{i})\big)$:
   (a) Parse z'_o as (τ, S, τ', S') and each z^i_i as (τ_i, S_i).
   (b) Parse z^1_i as (τ_1, S_1) and verify that $\tau = \tau_1$ and $S = S_1$.
   (c) If $\mathsf{z}^2_\mathsf{i} = \mathsf{ok}$, verify that $\tau' = \tau_1$ and $S' = \mathsf{ok}$;
   otherwise parse z^2_i as (τ_2, S_2) and verify that $\tau' = \tau_2$ and $S' = S_2$.
   (d) Verify that $\mathsf{UMC}_{y'}(\mathsf{z}^2_\mathsf{i}; (\mathsf{z}^1_\mathsf{i}), \mathsf{prog})$ accepts.

2. Internal Stage

   If $\mathsf{z_o} = \big((d+1, \mathsf{z}'_\mathsf{o})\big)$ and $\vec{\mathsf{z}}_\mathsf{i} = \big((d, \mathsf{z}^i_\mathsf{i})\big)^r_{i=1}$:
   (a) Parse z'_o as (τ, S, τ', S') and each z^i_i as $(\tau_i, S_i, \tau'_i, S'_i)$.
   (b) Verify that $\tau = \tau_1$, $\tau'_1 = \tau_2$, $\tau'_2 = \tau_3$, and so on until $\tau'_{r-1} = \tau_r$, $\tau'_r = \tau'$.
   (c) Verify that $S = S_1$, $S'_1 = S_2$, $S'_2 = S_3$, and so on until $S'_{r-1} = S_r$, $S'_r = S'$.

3. Exit Stage

   If $\mathsf{z_o} = \mathsf{ok}$ and $\vec{\mathsf{z}}_\mathsf{i} = \big((d_i, \mathsf{z}^i_\mathsf{i})\big)^{r'}_{i=1}$:

(a) Parse each z_i^i as $(\tau_i, S_i, \tau'_i, S'_i)$.
(b) Verify that $\tau'_1 = \tau_2$, $\tau'_2 = \tau_3$, and so on until $\tau'_{r'-1} = \tau_{r'}$.
(c) Verify that $S'_1 = S_2$, $S'_2 = S_3$, and so on until $S'_{r'-1} = S_{r'}$.
(d) Verify that $\tau_1 = 0$ and $S'_{r'} = \mathsf{ok}$.

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

We can immediately see that, as promised, the depth of the new compliance predicate $\textsc{TreeUMC}^{\triangleq r}_{y'}$ is improved; recall from Lemma 7.5 that the depth of the old compliance predicate $\mathsf{UMC}_{y'}$ could be as bad as $t'+1$, whereas the new bound is now given by the following lemma:

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

Proof. Any transcript that is compliant with $\textsc{TreeUMC}^{\triangleq r}_{y'}$ consists of disjoint trees. In each such tree, nodes of different heights are all 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 $t'+1$ 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 relevant because we can accompany it with completeness and soundness guarantees that ensure that despite the fact that we changed the compliance predicate to a different one, the "semantics" have been preserved:

Claim 8.4. *$y' \in \mathcal{L}'_{\mathcal{U}}$ if and only if there exists a (distributed computation) transcript T that is compliant with respect to $\textsc{TreeUMC}^{\triangleq r}_{y'}$ containing at least one sink with the data ok entering it.*

Moreover, we can efficiently find witnesses "both ways":

- *Given (w, m) such that $(y', (w, m)) \in \mathcal{R}'_{\mathcal{U}}$, it is possible to efficiently construct a transcript T that is compliant with respect to $\textsc{TreeUMC}^{\triangleq r}_{y'}$ containing a sink with the data "ok" entering it.*
- *Given a transcript T that is compliant with respect to $\textsc{TreeUMC}^{\triangleq r}_{y'}$ containing a sink with the data "ok" entering it, it is possible to efficiently construct (w, m) such that $(y', (w, m)) \in \mathcal{R}'_{\mathcal{U}}$.*

Proof. We need to prove two directions:

($\rightarrow$) Assume that $y' \in \mathcal{L}'_{\mathcal{U}}$, and let (w, m) be such that $\big(y', (w, m)\big) \in \mathcal{R}'_{\mathcal{U}}$. Construct a transcript T as follows.

First follow the forward direction in the proof of Claim 7.6 to construct a temporary transcript $\tilde{\mathsf{T}}$ that is compliant with respect to $\mathsf{UMC}_{y'}$ containing a path ending in ok; $\tilde{\mathsf{T}}$ will be a path graph with $\tilde{t}+3$ nodes for some $\tilde{t} \leq t'$ nodes.

Now consider $\tilde{t}+1$ nodes labeled the same way as in $\tilde{\mathsf{T}}$ (when excluding the source and sink from the path), and for each node i put two edges entering it, labeled with the messages entering and exiting node $i+1$ in $\tilde{\mathsf{T}}$. Construct a tree having the $\tilde{t}+1$ nodes as leaves as follows: 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; 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; and so on until no more grouping can be performed; finally, take all the top-level nodes and make them all children of the root. Next label the tree recursively by "merging" timestamp-configuration pairs as suggested by the second case in the description of $\textsc{TreeUMC}^{\triangleq r}_{y}$; finally put an edge exiting the root with the message ok.

See Figure 6 for a diagram of an example where $r = 2$ and $\tilde{t} = 4$.

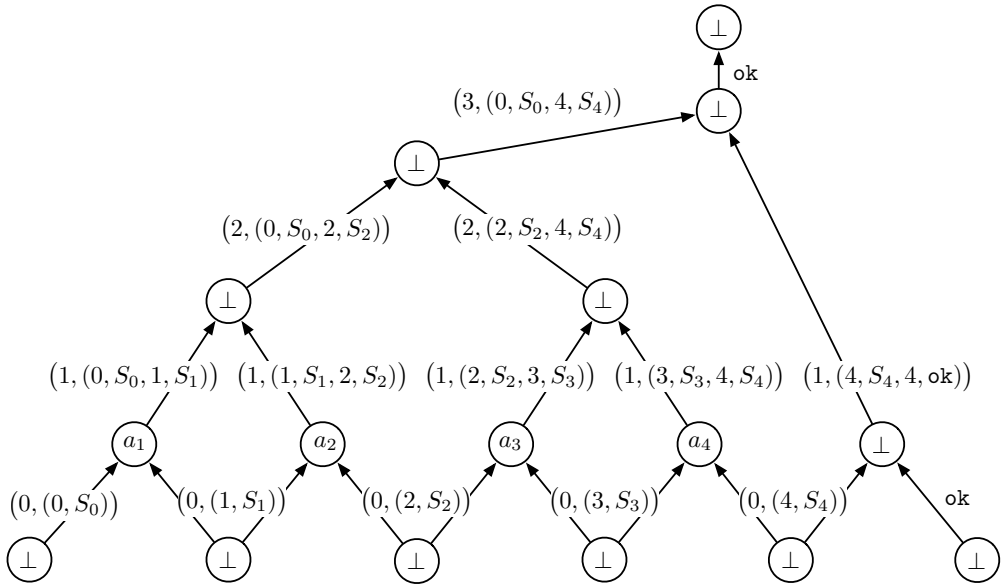

Figure 6: A compliant transcript for the tree construction with in-degree 2 and a computation of $\tilde{t} = 4$ time steps.

Because $\big(y', (w, m)\big) \in \mathcal{R}'_{\mathcal{U}}$, we know that T is compliant with $\text{TREEUMC}^{\triangleq r}_{y'}$ and indeed contains a sink with the data ok entering it.

($\leftarrow$) Consider any transcript T that is compliant with respect to $\text{TREEUMC}^{\triangleq r}_{y'}$ and contains a sink with the data ok entering it.

According to $\text{TREEUMC}^{\triangleq r}_{y'}$ the only way to obtain the message ok is to receive messages $(\tau_i, S_i, \tau'_i, S'_i)$ of possibly different depths spanning the entire (accepting) computation; also according to $\text{TREEUMC}^{\triangleq r}_{y'}$ the only way to obtain each of these messages is to receive r messages that correctly collapse to the message or is to receive two messages where the second one is correctly obtained from the first with one step of computation (and in this latter case we know that we are at a leaf).

Thus, we can simply trace back from the sink receiving the message ok to the leaves, and at the leaves we will find enough information to construct (w, m) such that $(y', (w, m)) \in \mathcal{R}'_{\mathcal{U}}$, in a similar manner as we did in the backward direction of the proof of Claim 7.6. □

8.2 General Case

In Section 8.1 we saw how for a very specific distributed computation that naturally evolves over a path, namely the step-by-step computation of a random access machine with untrusted memory (and we reiterate here that untrusted memory allows for the computation at each node to be small), we can enforce correctness by constructing a wide Merkle tree of proofs "on top" of it rather than directly composing proofs along a path.

More generally, given a compliance predicate $\mathbb{C}$, we can apply the same technique to obtain a "tree" version $\text{TREE}_{\mathbb{C}}$ of the compliance predicate with much smaller depth. To actually pull this through in the general case we need to be more careful because the data exchanged along the computation may not be so small (as was the case for a configuration of the random-access machine), so that, instead of comparing this data as we go up the tree, we compare their hashes.

Details follow.

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:

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

1. Leaf Stage

   If $\mathsf{z_o} = (1, \mathsf{z'_o})$ and $\vec{\mathsf{z}}_\mathsf{i} = \big((0, \mathsf{z}_\mathsf{i}^1), (0, \mathsf{z}_\mathsf{i}^2)\big)$:
   (a) Parse $\mathsf{z'_o}$ as $(\tau, \rho, \tau', \rho')$ and each z_i^i as (τ_i, z_i).
   (b) Verify that $\tau = \tau_1$ and $\tau' = \tau_2$.
   (c) Verify that $\rho = h(z_1)$ and $\rho' = h(z_2)$.
   (d) Verify that $\mathbb{C}(z_2; (z_1), \mathsf{prog})$ accepts.
2. Internal Stage

   If $\mathsf{z_o} = (d+1, \mathsf{z'_o})$ and $\vec{\mathsf{z}}_\mathsf{i} = \big((d, \mathsf{z}_\mathsf{i}^i)\big)_{i=1}^{r}$:
   (a) Parse $\mathsf{z'_o}$ as $(\tau, \rho, \tau', \rho')$ and each z_i^i as $(\tau_i, \rho_i, \tau'_i, \rho'_i)$.
   (b) Verify that $\tau = \tau_1$, $\tau'_1 = \tau_2$, $\tau'_2 = \tau_3$, and so on until $\tau'_{r-1} = \tau_r$, $\tau'_r = \tau'$.
   (c) Verify that $\rho = \rho_1$, $\rho'_1 = \rho_2$, $\rho'_2 = \rho_3$, and so on until $\rho'_{r-1} = \rho_r$, $\rho'_r = \rho'$.
3. Exit Stage

   If $\mathsf{z_o} = \mathsf{prog}$ and $\vec{\mathsf{z}}_\mathsf{i} = \big((d_i, \mathsf{z}_\mathsf{i}^i)\big)_{i=1}^{r'}$:
   (a) Parse each z_i^i as $(\tau_i, \rho_i, \tau'_i, \rho'_i)$.
   (b) Verify that $\tau'_1 = \tau_2$, $\tau'_2 = \tau_3$, and so on until $\tau'_{r'-1} = \tau_{r'}$.
   (c) Verify that $\rho'_1 = \rho_2$, $\rho'_2 = \rho_3$, and so on until $\rho'_{r'-1} = \rho_{r'}$.
   (d) Verify that $\tau_1 = 0$ and $\rho'_{r'} = h(\mathsf{prog})$.
4. If none of the above conditions hold, reject.

The main advantage of using the compliance predicate $\text{TREE}_{\mathbb{C}}^{h,\triangleq r}$ instead of $\mathbb{C}$ is that we have been able to squash the depth of the compliance predicate (assuming of course that the original depth was bounded to begin with):

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

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

Proof. Any transcript that is compliant with $\text{TREE}_{\mathbb{C}}^{h,\triangleq r}$ consists of disjoint trees. In each such tree, nodes of different heights are all 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). □

Next, we show that the transformation is "computationally sound":

Lemma 8.7. *For any compliance predicate $\mathbb{C}$ and $r \in \mathbb{N}$, with all but negligible probability over a random choice of h in $\mathcal{H}$, if an adversary $\mathcal{A}(h)$ outputs a transcript T' that is compliant with $\textsc{Tree}_{\mathbb{C}}^{h,\triangleq r}$ containing at least one sink with the data ok entering it, then we can extract from T' a transcript T that is compliant with $\mathbb{C}$.*

Proof. Due to the collision resistance property of the family $\mathcal{H}$, with all but negligible probability over a random choice of h in $\mathcal{H}$, the adversary $\mathcal{A}(h)$ does not find any collisions for h. Conditioned on $\mathcal{A}$ not having found any collisions, we proceed to extract T from T' as follows.

As already observed in the proof of Lemma 8.6, any transcript that is compliant with $\textsc{Tree}_{\mathbb{C}}^{h,\triangleq r}$ consists of disjoint trees. Consider a tree corresponding to the sink that has the data ok entering it. (By assumption, such a sink exists.) Then we see that, by construction of $\textsc{Tree}_{\mathbb{C}}^{h,\triangleq r}$, the only way we could have obtained the message ok is by receiving messages from nodes at different heights of a tree and these messages merged appropriately (namely, they had the appropriate time stamps and hashes); each of these messages must have been the product of merging corresponding messages at the same height in groups of r, and recursively so on until we reach messages whose depth is equal to 1. These messages must have been generated by a node which checked $\mathbb{C}$-compliance for inputs hashing down to the hashes contained in each depth-1 message. We thus have found a "path of $\mathbb{C}$-compliant computation" at the leaves of the tree, and thus can easily generate the transcript T by constructing a graph containing this path and labeling it appropriately. □

Next, we prove that that moving to the new compliance predicate $\textsc{Tree}_{\mathbb{C}}^{h,\triangleq r}$ instead of the old $\mathbb{C}$ does not hurt completeness when starting from a transcript that is a path; moreover, the transformation can be done "on-the-fly" by a sequence of parties on the same path without much overhead in computation and communication.

Lemma 8.8. *Any transcript T compliant with a compliance predicate $\mathbb{C}$ that is a path can be efficiently transformed into a transcript T' compliant with $\textsc{Tree}_{\mathbb{C}}^{h,\triangleq r}$, when given as input $\mathbb{C}$, $h \in \mathcal{H}$, and $r \in \mathbb{N}$.*

Moreover, the transformation to T' can be performed "on-the-fly" by parties along the path with only $r \cdot \lceil \log_r \mathsf{d}(\mathsf{T}) \rceil$ overhead in state sent between any two consecutive parties on the graph (and a corresponding additive polynomial in $r \cdot \lceil \log_r \mathsf{d}(\mathsf{T}) \rceil$ overhead in computation at each node).

Proof. Let T be a transcript that is a path graph and is compliant with $\mathbb{C}$. Consider the nodes in T that are not the source or a sink as the leaves of a tree we are going to construct (more precisely, each of these nodes will be pointed to by two edges, each coming from a different source); note that there are $\mathsf{d}(\mathsf{T}) \leq \mathsf{d}(\mathbb{C})$ such nodes. The tree above the leaves is constructed 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.

We can then label the resulting tree graph and produce a transcript T' as follows. We take the input label of each (non-source, non-sink) node in T and set the label of the corresponding leaf in the tree graph equal to it. All other nodes in the tree can be labeled arbitrarily, except that the root of the tree is labeled with the data coming out of the last node in T. As for the edge labels, we can start by labeling every left child of leaf nodes in the tree with the incoming data of the corresponding node in T, and then labeling every right child of leaf nodes in the tree with the outgoing data of the corresponding node in T. We then recursively label the tree going upwards following the labels suggested by the compliance predicate (i.e., at the first level we take hashes of the inputs, and then keep comparing hashes keeping only the first and last one, until the root where we ultimately compare the last hash with the claimed output that is available as a label there). By construction, T' is compliant with $\textsc{Tree}_{\mathbb{C}}^{h,\triangleq r}$.

Finally, note that the labeling of the above described tree can be done "on-the-fly" by parties places along the "original" path graph that gave rise to T. Specifically, the i-th party needs to communicate to the $(i+1)$-th party only the roots only the labels and proofs of the "current roots", of which there are at most $r \cdot \lceil \log_r i \rceil$; the proofs are of course short, and the labels are short because they only contain timestamps and hashes of inputs. The next party will thus receive, along with its "semantic" input, a vector of inputs and proofs from the inside of the tree; this party will produce the proof corresponding to its node, and will then try to build further the tree if it can, and then forwards its "semantic" output along with whatever state is needed to keep building the tree to the next party. Clearly this process has the claimed efficiency properties.

In Figure 7 we show how the computation of parties is modified when moving from computing proofs for $\mathbb{C}$ to computing proofs for $\text{TREE}_{\mathbb{C}}^{h,\triangleq r}$ with $r = 2$. □

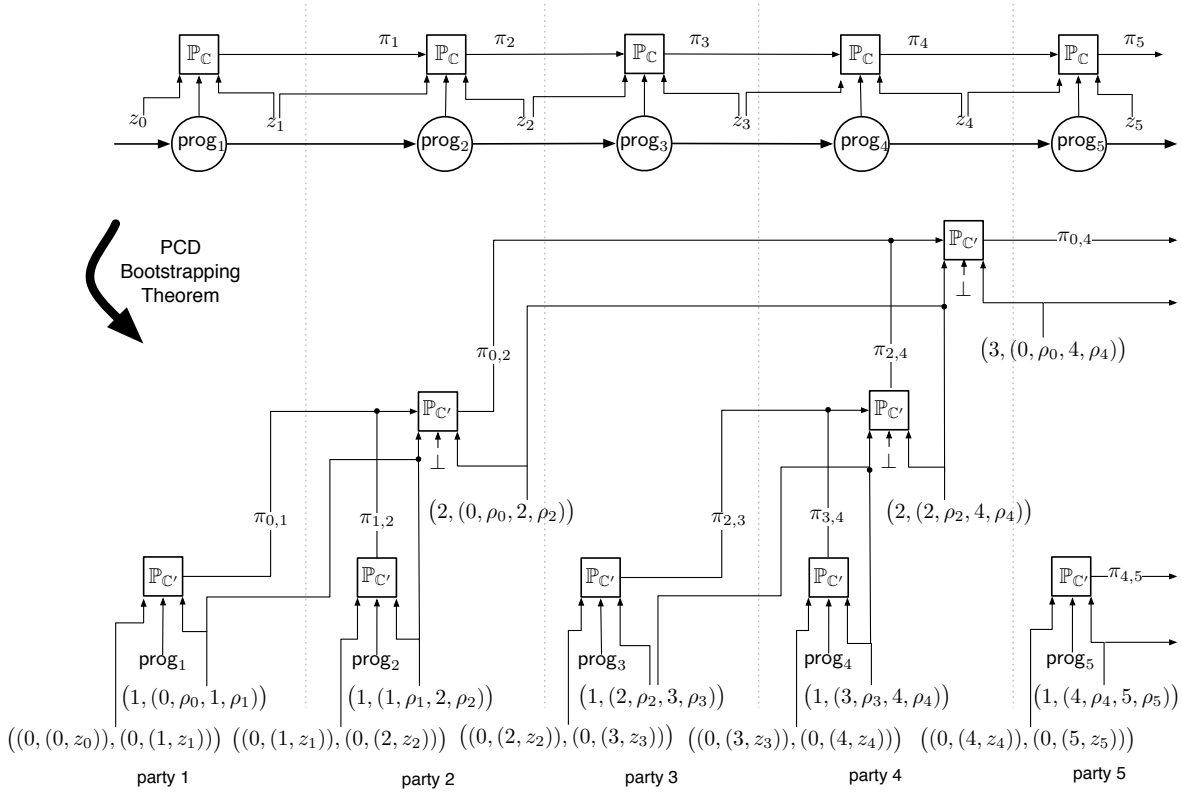

Figure 7: How the computation of parties is modified when moving from computing proofs for $\mathbb{C}$ to computing proofs as a Merkle tree of in-degree r with $r = 2$; essentially, each party will compute its own proof and attempt to build the tree as far "up" as possible, and then forward to the next party the relevant state.

We are now ready to prove the PCD Bootstrapping Theorem:

Proof of Theorem 8.1. We have already prove all the ingredients for the theorem, but let us explain how they come together. Let $(\mathbb{G}, \mathbb{P}_{\mathbb{C}'}, \mathbb{V}_{\mathbb{C}'})$ be a PCD system that is secure for constant-depth compliance predicates $\mathbb{C}'$. We explain how to construct $(\mathbb{G}', \mathbb{P}'_{\mathbb{C}}, \mathbb{V}'_{\mathbb{C}})$ that is a PCD system that is secure for compliance predicates of fixed polynomial depth though its completeness we can only show for transcripts that are paths.

As an intuitive guide for the construction, refer to Figure 7. The first step is to define the new compliance predicate $\mathbb{C}' := \text{TREE}_{\mathbb{C}}^{h,\triangleq k}$; note that we have chosen the in-degree of the Merkle tree to be equal to the security parameter k.

The new prover $\mathbb{P}'_{\mathbb{C}}$ will run the old prover $\mathbb{P}_{\mathbb{C}'}$ but with the new compliance predicate $\mathbb{C}'$ to compute the proof of the leaf of the node, and may have to run it more times in order to compute proofs internal to the Merkle tree.

The new verifier $\mathbb{V}'_{\mathbb{C}}$ will receive a claimed output and not just one but a vector of (possibly more) proofs, representing the current "state" of a partially built Merkle tree of proofs; in order to verify, $\mathbb{V}'_{\mathbb{C}}$ will verify each proof separately (by running the old verifier $\mathbb{V}_{\mathbb{C}'}$ with the new compliance predicate $\mathbb{C}'$) and then check consistency of the corresponding state messages (following the "Exit Stage" case in the definition of $\text{TREE}_{\mathbb{C}}$); indeed, in the proofs of completeness (Lemma 8.8) and soundness (Lemma 8.7) above we have used the message ok as a proxy for the verifier that we have described now.

As for the new generator $\mathbb{G}'$, if the original system PCD is not of the preprocessing kind, then there is nothing else to do (that is, we can just set $\mathbb{G}' = \mathbb{G}$. If instead the original PCD system was of the preprocessing kind, then the new generator $\mathbb{G}'$ may have to invoke the old generator $\mathbb{G}$ with a slightly higher time bound, if the computation at "internal nodes" of the tree exceeds the previous time bound. □

Remark 8.9. Of course concentrating on paths is merely the simplest example of a PCD Bootstrapping Theorem obtained given the techniques discussed here. We could generalize or modify the mapping from $\mathbb{C}$ to $\text{TREE}_{\mathbb{C}}$ to accommodate other structures. For example, we could have a PCD Bootstrapping Theorem for graphs that have the shape of a "Y" instead of for paths, by simply building a wide Merkle tree independently on each of the three segments of the "Y". It is interesting to understand to what extent one can find PCD Bootstrapping Theorems beyond the case for paths and other related basic graphs and apply them with success.

9 Constructions of Publicly-Verifiable Preprocessing SNARKs[12]

Groth [Gro10] shows how to construct *non-interactive succinct zero-knowledge arguments* with a common reference string (CRS) that is quadratic in the size of the NP-verification circuit C at hand. (Later, Lipmaa [Lip11] showed how to extend Groth's techniques to make the CRS quasilinear in the size of C, but with a prover that still runs in quadratic time.) The construction relies on a variant of the *Knowledge of Exponent* assumption [Dam92, BP04], which he calls q-PKEA, and a computational Diffie-Hellman assumption, which he calls q-CPDH, in bilinear groups.

While Groth mainly focuses on the succinctness of the proof and its zero-knowledge property, his construction can in fact be viewed as a publicly-verifiable (zero-knowledge) SNARK with preprocessing; indeed, in Groth's construction, not only are the proofs succinct in their length, but the verification state corresponding to the CRS as well as the verification time are succinct, i.e., polynomial in the security parameter and independent of the circuit size $|C|$.

We now recall at high-level the main components of Groth's construction, stressing the observations that are relevant for succinctness, which is required for the applications considered in this paper.

At high-level, the construction of Groth follows a (by now common) paradigm [GOS06] for constructing NIZKs over bilinear groups: the prover commits to the values of all the wires in the circuit C and, using homomorphic properties of the commitment, adds proofs attesting to their consistency. Then, to verify these

[12] We thank Daniel Wichs for discussions of the preprocessing constructions of Groth and Lipmaa, and for pointing out a mistake in a previous draft of this paper.

proofs, the verifier uses the bilinear map. However, without modification, this approach results in proofs and verification time that are proportional to the size of the circuit $|C|$.

To achieve succinctness, Groth uses *succinct commitments of knowledge*, where a commitment to all the values in the circuit C is of length that is a fixed polynomial in the security parameter. Then, using some additional algebraic properties of the commitments, Groth presents clever methods for "packing" all the consistency proofs into a few simultaneous proofs (while still maintaining succinctness). As the knowledge commitment includes the witness w satisfying the circuit C, Groth's protocol also enjoys a proof of knowledge property, as required to be a SNARK.

We address two aspects of the construction that are implicit in Groth:

- **Succinct verification.** As already observed by Lipmaa [Lip11], the verification procedure can be divided into two main stages. The first is an "offline preprocessing stage", where a (long) CRS σ and a corresponding short verification state τ are generated; concretely, τ consists of a constant number of group elements that are pre-computed based on the topology of the circuit C to be verified, so that $|\tau| = \mathrm{poly}(k)$ is independent of $|C|$. The second stage is an "online verification stage" where the proof received from the prover is verified against the short verification state τ; this verification involves a constant number of applications of the bilinear map, and thus overall requires $\mathrm{poly}(k)$ time, again independent of $|C|$.

- **Adaptive choice of inputs and universality.** In our applications, we will be interested in applying Groth's techniques to a fixed universal circuit deciding bounded halting problems, i.e., deciding membership in the universal language. (Recall the definitions in Section 3.) Specifically, for a given bound B, we will use a universal circuit U_B that takes any input $(y, w) = ((M, x, t), w)$ with $|w| \leq t$ and $|M| + |x| + t \leq B$, and checks whether M accepts (x, w) after at most t steps. (The circuit can be constructed in time $\mathrm{poly}(B)$, for some fixed polynomial poly.[13])

  An important property satisfied by Groth's construction is that the CRS σ and the corresponding verification state τ only depend on the (fixed) circuit U, and can in fact support different inputs $y = (M, x, t)$ that may be chosen by the prover, possibly depending on σ.

  Note that, as presented in Groth, the input y is hardwired into the verification circuit (and, as previously explained, the generation of (σ, τ) depends on this circuit). However, the construction can be slightly augmented to support an adaptive choice of inputs, where the same (σ, τ) can be used for many statements y, including those that are adaptively chosen by the prover. For example, this can be achieved by thinking about the input as part of the witness, and requiring the prover to provide an additional proof asserting that the relevant part of the commitment to the witness is indeed consistent with the claimed input y (chosen by verifier or prover). Such a consistency check is an integral part of the techniques developed by Groth (the CRS only suffers at most a linear blowup, and the verification state only grows additively in $\mathrm{poly}(k)$).[14]

Overall, we have the following:

Theorem 9.1 ([Gro10])**.** *Assuming q-PKEA and q-CPDH over bilinear groups, there exist preprocessing publicly-verifiable SNARKs where, given any size bound B with $|U_B| = O(\sqrt{q})$,*

[13] The bound can be refined: for any two given size bounds B_0 and B_1 there exists a circuit U_{B_0,B_1} of size $\mathrm{poly}(B_0)\tilde{O}(B_1)$ that can handle any input $(y, w) = ((M, x, t), w)$ with $|w| \leq t$, $|M| + |x| \leq B_0$, and $t \leq B_1$. See, e.g., [BSCGT12].

[14] In the terms of Groth's construction, this involves dividing the input from the commitment and performing a corresponding restriction argument.

- *it supports instances $y = (M, x, t)$ such that $|M| + |x| + t \leq B$,*
- *the reference string σ has size $p(k)|U_B|^2$,*
- *the verification state τ has size $p(k)$,*
- *the prover running time is $p(k)|U_B|^2$, and*
- *the verifier running time is $p(k, |y|)$,*

where p is a universal polynomial, independent of B.

9.1 Where are the PCPs?

The work mentioned above does not invoke the PCP Theorem. This is an interesting aspect, regardless of the ability to achieve public verifiability or not.

Concretely, Groth [Gro10] does not make explicit use of any probabilistic-checking techniques. This stands in contrast with all other known SNARK constructions, including Micali's construction in the random oracle model, who leverage the full machinery of PCPs.

While the original construction is not quite so efficient because the prover incurs in a quadratic blowup in running time (even in Lipmaa's construction with a shorter reference string), not having to invoke the PCP Theorem raises the hope that constructions leveraging their techniques (such as the transformations for removing preprocessing that we present in this paper) may be potentially more efficient than constructions that do invoke the PCP Theorem.

10 Putting Things Together: Main Theorem

In the previous sections we have shown three main results:

- in Section 6 the **SNARK Recursive Composition Theorem** (Theorem 6.1);
- in Section 7 the **RAM Compliance Theorem** (Theorem 7.1); and
- in Section 8 the **PCD Bootstrapping Theorem** (Theorem 8.1).

We proceed to explain how these results can be combined together to ascend from the humble properties of a preprocessing SNARK, to a SNARK without preprocessing, and then even beyond that to path PCDs for compliance predicates of fixed polynomial depth. Furthermore, along the way, the verifiability properties of the underlying SNARK will be perfectly preserved. Note that we *do not use PCPs* along the way; in particular, if the beginning preprocessing SNARK is practical enough, so will the SNARK with no preprocessing as well as the resulting PCD schemes. Another way to interpret our results is that building preprocessing SNARKs contains almost all the hardness of building the much stronger primitives of SNARKs with no preprocessing and PCDs. We find this to be quite surprising.

That is, we now prove the following theorem:

Theorem 10.1 (Main Theorem)**.** *Assume there exist preprocessing SNARKs. Then there exist:*

(i) SNARKs with no preprocessing.

(ii) PCDs for compliance predicates of constant depth with no preprocessing.

(iii) Path PCDs for compliance predicates of fixed polynomial depth with no preprocessing.

The above holds for both the publicly-verifiable and designated-verifier cases. In the private-verification case, we assume the existence of a fully-homomorphic encryption scheme; in both cases we assume the existence of collision-resistant hash functions.

In particular, in light of the discussion in Section 9, we obtain the following two corollaries.

Corollary 10.2. *Assuming q-PKEA and q-CPDH over bilinear groups:*

1. *there exist publicly-verifiable SNARKs with no preprocessing,*
2. *there exist publicly-verifiable PCDs for constant-depth compliance predicates, and*
3. *there exist publicly-verifiable path PCDs for compliance predicates of fixed polynomial depth.*

Proof. We can simply plug into our Theorem 10.1 the publicly-verifiable preprocessing SNARKs of Groth (see Theorem 9.1). □

Corollary 10.2 gives the first construction of a publicly-verifiable SNARK with no preprocessing (a.k.a. a "CS proof" [Mic00]) in the plain model based on a simple knowledge assumption in bilinear groups; at the same time we obtain the first PCD constructions in the plain model as well. Both constructions do not use PCPs, or "traditional" probabilistic checking techniques (such as proximity testing or sum checks).

In summary, thanks to our machinery that shows how to generically remove preprocessing from any SNARK, we have thus shown how to leverage quite simple techniques, such as techniques based on pairings [Gro10] that at first sight seem to necessarily give rise to only preprocessing solutions, in order to obtain much more powerful solutions with *no* preprocessing. The resulting constructions, while certainly not trivial, seem to be quite simple, and do not invoke along the way additional probabilistic-checking machinery. We find this (pleasantly) surprising.

We now proceed to sketch the proof of the theorem; see Figure 2 for a diagram of what we are about to explain.

Proof of Theorem 10.1. We discuss each implication separately.

(i) Removing preprocessing from a SNARK. Given a preprocessing SNARK, we wish to "improve" it via a black-box construction so the resulting SNARK does *not* suffer from the handicap of preprocessing.

The very high level idea is that we are going to break up the theorem that needs to be proved into a distributed computation of many small computations, and then use the fact that we can compose SNARKs (to obtain a PCD system) to prove correctness of this distributed computation, after properly structuring it to have low depth.

More precisely, the SNARK Recursive Composition Theorem tells us that SNARKs can be composed to obtain a PCD system (with preprocessing) for which we can show security for every constant-depth compliance predicate. Then, the PCD Bootstrapping Theorem tells us that we can lift any such PCD system to a PCD system for which we can show security for compliance predicates of fixed polynomial depth (though we can show completeness only for distributed computation transcripts that are paths). Now it is left to observe that RAM Compliance Theorem implies that membership in $\mathcal{L}_{\mathcal{U}}$ of an instance (M, x, t) can be computationally reduced to the question of whether there is a distributed computation compliant with UMC_y, which has depth at most $O(t \log t)\mathrm{poly}(k)$; moreover, this distributed computation is a path. Because each

of the computations in the resulting distributed computation are small, the fact that we are using a PCD with preprocessing *does not hurt* — and thus we have removed the preprocessing handicap. Here, we have invoked a special case of the SNARK Recursive Composition Theorem, described in Corollary 6.5, where the computations to be proven are all of fixed polynomial size in the security parameter.

Note that to meaningfully invoke the RAM Compliance Theorem, it was important for the SNARK (and thus PCD) to have a proof-of-knowledge property. Moreover, because the RAM Compliance Theorem guarantees a (computational) *Levin* reduction, this proof of knowledge is preserved through the reduction (as discussed in mode detail in Section 7).

Also note that the transformation preserves the verifiability properties of the SNARK. For example, if the beginning preprocessing SNARK was publicly verifiable, so will be the resulting SNARK coming out of our construction. We have thus shown a way to obtain publicly-verifiable SNARKs (also known as "CS proofs" [Mic00]) from merely a publicly-verifiable preprocessing SNARK.

(ii) Composing SNARKs. Once we have SNARKs without preprocessing, we can use our tools *once more* to go further, and obtain PCDs *without preprocessing*. Namely, we can use (again) the SNARK Recursive Composition Theorem to obtain PCD systems (this time without preprocessing) for which we can show security for every constant-depth compliance predicate.

(iii) Bootstrapping PCDs. Then, if we wish to, we can use (again) the PCD Bootstrapping Theorem to lift any PCD system for constant-depth compliance predicates to a PCD system (without preprocessing) for which we can show security for compliance predicates of fixed polynomial depth, though we can show completeness only for distributed computations evolving over paths.

□

Remark 10.3 (a word about efficiency)**.** We briefly discuss the efficiency of the SNARK prover $\mathcal{P}'$ and verifier $\mathcal{V}'$ generated by our Main Theorem, for a given underlying preprocessing SNARK $(\mathcal{G}, \mathcal{P}, \mathcal{V})$. (A similar discussion will hold for PCD provers and verifiers generated by our Main Theorem.)

The SNARK verifier $\mathcal{V}'$ is of course succinct, and will thus run in time $\mathrm{poly}(k, |y|)$ when given an instance y and proof π, where poly is a universal polynomial. More concretely (but still informally), $\mathcal{V}'$ will apply a reduction to the instance y to obtain a related compliance predicate $\mathbb{C}_y$ and then run the PCD verifier resulting from the recursive composition of proofs of $(\mathcal{G}, \mathcal{P}, \mathcal{V})$, which is essentially running the verifier $\mathcal{V}$ on an instance y' derived from $\mathbb{C}_y$. Overall, the cost of running $\mathcal{V}'$ is roughly the cost of applying the reduction from y to $\mathbb{C}_y$, and then running $\mathcal{V}$ on an instance y' derived from $\mathbb{C}_y$.

Note that, if the running time of $\mathcal{V}$ depends poorly on $|y|$ (e.g., it runs in time $\mathrm{poly}(k, |y|) = |y|^2 k^c$ for some c), then so will $\mathcal{V}'$.[15] We can fix this by invoking another reduction of [BSCGT12]:[16] on input $y = (M, x, t)$, $\mathcal{V}'$ can simply first hash y to obtain a digest $h(y)$, and then, instead of expecting a proof for y, $\mathcal{V}'$ will expect a proof for a related instance $\tilde{y} = (U, h(y), t')$, where (1) U is a fixed machine that interprets a given witness as a pair (y, w) and verifies that $h(y)$ is the hash of y and that $(y, w) \in \mathcal{R}_{\mathcal{U}}$, and (2) $t' = t \cdot \mathrm{poly}(k)$. In this way, we can ensure that the running time of $\mathcal{V}'$ is equal to $\mathrm{poly}(k)$ plus the time required to hash y (which is typically, $|y|\mathrm{poly}(k)$).

As for the prover $\mathcal{P}'$, we note that, for reasons that are similar to the proof of the PCD Linearization Theorem (Theorem 2.1), $\mathcal{P}'$ runs in time $t \cdot |y|^r \cdot \mathrm{poly}(k)$ where r is the exponent of $|y|$ in the running time of $\mathcal{P}$ (and this holds *even* if the underlying $\mathcal{P}'$ is super-linear, e.g., quadratic as in [Gro10]). If $r > 1$, we can apply the same instance-hashing trick described in the previous paragraph to ensure that $|y|$ only appears as a linear factor in the running time of $\mathcal{P}'$, and thus ensure that $\mathcal{P}'$ runs in time $t \cdot |y|^r \cdot \mathrm{poly}(k)$.

[15] This is, fortunately, not the case in [Gro10]: the verifier runs in time $|y|\mathrm{poly}(k)$.

[16] Concretely, see the Untrusted Input Lemma and Untrusted Code Lemma of [BSCGT12].

11 Zero Knowledge

In many applications of both SNARKs and PCDs a very desirable additional property is *zero knowledge*, which, for example, can be used to ensure *function privacy* or *worker input privacy*. We define this property for both primitives and explain how it can be obtained.

11.1 Zero-Knowledge SNARKs

In SNARKs, the property of zero knowledge ensures that the honest prover can generate valid proofs for true theorems without leaking any information about the theorem beyond the fact that the theorem is true (in particular, without leaking any information about the witness that he used to generate the proof for the theorem). Formally:

Definition 11.1. *A* **(perfect) zero-knowledge SNARK** *is a triple of algorithms $(\mathcal{G}, \mathcal{P}, \mathcal{V})$ that is a SNARK (following Definition 4.2) and, moreover, satisfies the following property:*

- *Zero Knowledge*

  There exists a stateful interactive polynomial-size simulator $\mathcal{S}$ such that for all stateful interactive polynomial-size distinguishers $\mathcal{D}$, sufficiently large security parameter $k \in \mathbb{N}$, and all auxiliary inputs $z \in \{0,1\}^{\mathrm{poly}(k)}$, the following holds:

$$\Pr_{(\sigma,\tau)\leftarrow\mathcal{G}(1^k)}\left[\begin{array}{c|c}(y,w)\in\mathcal{R}_{\mathcal{U}} & (y,w)\leftarrow\mathcal{D}(z,\sigma)\\ \mathcal{D}(\pi)=1 & \pi\leftarrow\mathcal{P}(\sigma,y,w)\end{array}\right] = \Pr_{(\sigma,\tau,\mathsf{trap})\leftarrow\mathcal{S}(1^k)}\left[\begin{array}{c|c}(y,w)\in\mathcal{R}_{\mathcal{U}} & (y,w)\leftarrow\mathcal{D}(\sigma)\\ \mathcal{D}(\pi)=1 & \pi\leftarrow\mathcal{S}(z,\sigma,y,\mathsf{trap})\end{array}\right].$$

  We can obtain zero-knowledge SNARKs in at least two ways:

- We proved in [BCCT11] that it is possible to combine designated-verifier SNARKs (dvSNARKs) with (not-necessarily-succinct) non-interactive zero-knowledge arguments of knowledge (e.g., [AF07]), using one of two different constructions (depending on whether the SNARK is "on top" or "on the bottom"), to obtain zero-knowledge dvSNARKs.

  The two constructions are even simpler in the case of publicly-verifiable SNARKs (pvSNARKs) as we do not have to worry about the privacy of the verifier verification state τ; we can thus also obtain via the same approach zero-knowledge pvSNARKs starting from pvSNARKs and non-interactive zero-knowledge arguments of knowledge.

- We could start with a zero-knowledge preprocessing SNARK at the "bottom of our result stack" and then construct corresponding SNARKs (with no preprocessing) as described in the proof of Theorem 10.1; the various transformations will preserve the zero-knowledge property. For example, we could plug into our transformations the construction of Groth [Gro10], which does have the zero-knowledge property. (See discussion in Section 9.)

11.2 Zero-Knowledge PCDs

In PCDs, the property of zero-knowledge ensures that the honest prover can generate valid proofs for his compliant output data without leaking any information about the data besides that fact that it is compliant (in

particular, without leaking any information about the input data and the local program he used to generate the proof for the output data). Formally:

Definition 11.2. *A $\mathbb{C}$-compliance* **(perfect) zero-knowledge PCD system** *is a triple of algorithms $(\mathbb{G}, \mathbb{P}_{\mathbb{C}}, \mathbb{V}_{\mathbb{C}})$ that is a PCD system (following Definition 5.6) and, moreover, further satisfies the following property:*

- *Zero Knowledge*

  There exists a stateful interactive polynomial-size simulator $\mathbb{S}_{\mathbb{C}}$ such that for all stateful interactive polynomial-size distinguishers $\mathcal{D}$, sufficiently large security parameter $k \in \mathbb{N}$, and all auxiliary inputs $z \in \{0,1\}^{\mathrm{poly}(k)}$, the following holds:

$$\Pr_{(\sigma,\tau)\leftarrow\mathbb{G}(1^k)}\left[\begin{array}{c|c} \mathbb{C}(\vec{\mathsf{z}}_{\mathsf{i}}, \mathsf{prog}, \vec{\mathsf{z}}_{\mathsf{o}}) = 1 & (\vec{\mathsf{z}}_{\mathsf{o}}, \mathsf{prog}, \vec{\mathsf{z}}_{\mathsf{i}}, \vec{\pi}_{\mathsf{i}}) \leftarrow \mathcal{D}(z,\sigma) \\ \mathbb{V}_{\mathbb{C}}(\tau, \vec{\mathsf{z}}_{\mathsf{i}}, \vec{\pi}_{\mathsf{i}}) = 1 & \pi_{\mathsf{o}} \leftarrow \mathbb{P}_{\mathbb{C}}(\sigma, \vec{\mathsf{z}}_{\mathsf{i}}, \vec{\pi}_{\mathsf{i}}, \mathsf{prog}, \mathsf{z}_{\mathsf{o}}) \\ \mathcal{D}(\pi_{\mathsf{o}}) = 1 & \end{array}\right]$$

$$= \Pr_{(\sigma,\tau,\mathsf{trap})\leftarrow\mathcal{S}(1^k)}\left[\begin{array}{c|c} \mathbb{C}(\vec{\mathsf{z}}_{\mathsf{i}}, \mathsf{prog}, \vec{\mathsf{z}}_{\mathsf{o}}) = 1 & (\vec{\mathsf{z}}_{\mathsf{o}}, \mathsf{prog}, \vec{\mathsf{z}}_{\mathsf{i}}, \vec{\pi}_{\mathsf{i}}) \leftarrow \mathcal{D}(z,\sigma) \\ \mathbb{V}_{\mathbb{C}}(\tau, \vec{\mathsf{z}}_{\mathsf{i}}, \vec{\pi}_{\mathsf{i}}) = 1 & \pi_{\mathsf{o}} \leftarrow \mathcal{S}(z, \sigma, \mathsf{z}_{\mathsf{o}}, \mathsf{trap}) \\ \mathcal{D}(\pi_{\mathsf{o}}) = 1 & \end{array}\right] .$$

As already mentioned in the previous subsection, both our SNARK Recursive Composition Theorem and our PCD Bootstrapping Theorem preserve the property of zero-knowledge. So we can obtain zero-knowledge PCDs simply by starting from zero-knowledge SNARKs.

We emphasize here that an important feature of the above definition is that an honest prover at some node in the computation can still "protect" himself with zero-knowledge even if previous provers were not honest, as long of course as he receives valid proofs to begin with.

Remark 11.3. The definition of a zero-knowledge SNARK (Definition 11.2) and zero-knowledge PCD (Definition 11.2) can of course be meaningfully relaxed in various ways.

For example, we may want to relax the indistinguishability requirement of the two distributions from exact equality to mere statistical indistinguishability, or even further to computational indistinguishability.

In another direction, we may want to let the SNARK simulator $\mathcal{S}$ or PCD simulator $\mathbb{S}_{\mathbb{C}}$ non-uniformly depend on the distinguisher $\mathcal{D}$, as opposed to insisting on the simulator being universal.

We choose to state the above strong version of the definition, because we can achieve it (as already explained).

12 Applications

SNARKs are a quite powerful cryptographic primitive: they allow for the construction of various extractable cryptographic primitives in the plain model, they have immediate applications to delegation of computation (including, thanks to their knowledge property, the ability to elegantly delegate large memories and data streams), and ultimately also enable constructing non-interactive succinct secure computation (A detailed discussion of all the above can be found in [BCCT11]).

In this paper, via Theorem 10.1 and its corollaries, we also obtain for the first time (certain kinds of) PCD systems in the plain model. Previously, PCD systems only had a proof of security in model where every party had access to signature oracle [CT10]. We thus want to briefly discuss a couple of the many applications specific to PCD (which cannot be obtained directly from SNARKs).

Compliance engineering. Recall that PCD provides a framework for reasoning about integrity of data in distributed computations. Specifically, integrity is abstracted away as a compliance predicate $\mathbb{C}$ that must locally hold at each node of computation, and each party computes on-the-fly short and easy-to-verify proofs to attach to each message exchanged with other parties in order to enforce $\mathbb{C}$-compliance.

Being able to enforce arbitrary compliance predicates on distributed computations is, indeed, very powerful. Yet, this is by far not the end of the journey — figuring out *what are useful compliance predicates* is a problem in its own right! This is especially true for real-world applications where a system designer may have to think hard to distill security properties to enforce in his distributed system. Thus, as SNARKs, and thus hopefully PCDs too, become more and more practical, we envision *compliance engineering* becoming an ever-more-relevant and exciting research direction.

In this paper, we have already exercised some compliance engineering skills. For example, the proof of the PCD Bootstrapping Theorem and the RAM Compliance Theorem were both problems about designing compliance predicates that would ensure the appropriate properties that we cared about in a distributed computation: in the former, the goal was to artificially slash down the depth of the input compliance predicate; in the second, the goal was to ensure the step-by-step execution of a random-access machine with untrusted memory.

We believe that, as we begin to explore and understand how to think about compliance in concrete problem domains, we are likely to uncover common problems and corresponding design patterns [GHJV95], thus improving our overall ability to correctly phrase desired security properties as compliance predicates and thus ultimately our ability to secure complex distributed systems.

In the following two applications we shall show how compliance engineering alone suffices for two recent goals whose integrity guarantees can be easily met by the guarantees of proof-carrying data.

12.1 Targeted Malleability

Boneh et al. [BSW11] recently defined *targeted malleability* as an intermediate notion between non-malleable encryption and fully-homomorphic encryption.

More precisely, given a set of "allowed functions" $\mathcal{F}$, the security goal is the following: for any efficient adversary that is given an encryption c of a message m and then outputs an encryption c' of message m', it should hold that either $c' = c$, m' is unrelated to m, or m' is obtained from m by applying a sequence of functions from the set $\mathcal{F}$ to m. (And here the security of the encryption could be considered, for example, with respect to a CPA game or, say, with respect to a CCA1 game.)

At the same time, of course, the functionality goal is to enable honest parties to successively apply any function from $\mathcal{F}$ on ciphertexts and, in order to make the problem interesting, the ciphertext size should not grow in the process.

Targeted malleability as a special case of PCD. Targeted malleability is, however, simply a *special case of (zero-knowledge) proof-carrying data*: we can easily engineer a compliance predicate $\mathbb{C}^{\mathsf{TM}}_{\mathsf{pk},\mathsf{ek},\mathcal{F}}$ that will ensure the desired security properties for a given set of functions $\mathcal{F}$. Here is the construction:

$\mathbb{C}^{\mathsf{TM}}_{\mathsf{pk},\mathsf{ek},\mathcal{F}}(\mathsf{z_o}; \vec{\mathsf{z}}_\mathsf{i}, \mathsf{prog}) \stackrel{\text{def}}{=}$

1. Source Case
   If $\vec{\mathsf{z}}_\mathsf{i} = \bot$ and $\mathsf{prog} = \bot$:
   (a) Verify that $\mathsf{z_o}$ is equal to $(\texttt{plain}, z)$ for some z.
2. Encrypt Case
   If $\vec{\mathsf{z}}_\mathsf{i} = ((\texttt{plain}, z))$ and $\mathsf{z_o} = (\texttt{cipher}, c)$:
   (a) Verify that c is the encryption of z under public key pk and randomness prog.

3. Compute Case
   If $\vec{\mathsf{z}}_{\mathsf{i}} = ((\mathtt{cipher}, c_i))_i$ and $\mathsf{z}_{\mathsf{o}} = (\mathtt{cipher}, c)$:
   (a) Interpret prog as a function f and verify that $f \in \mathcal{F}$.
   (b) Verify that c is the homomorphic evaluation of f on (encrypted) inputs $(c_i)_i$ with evaluation key ek.
4. Reject in all other cases.

In other words, the compliance predicate $\mathbb{C}^{\mathsf{TM}}_{\mathsf{pk},\mathsf{ek},\mathcal{F}}$ for targeted malleability simply makes sure that every edge exiting a source carries a plaintext that at the next node can only be encrypted and, following that, only allowed computations of ciphertexts can be done. Thus, the only way that a ciphertext can enter the system is if someone encrypted a message first and then sent it to someone and once a ciphertext has entered then it can only be modified via functions in $\mathcal{F}$. The above compliance predicate also captures the case with functions with multiple input.

Better conceptual understanding. Once again proof-carrying data provides a powerful framework to enforce integrity properties in distributed computations. In our view, though, proof-carrying data also helps us separate integrity concerns and other "application-specific" ones. For example, in the case of targeted malleability, Boneh et al. provided a definition that simultaneously enforces the integrity and confidentiality properties, whereas by abstracting away the integrity properties as a compliance predicate we can see how confidentiality interacts with the basic integrity guarantees. (In other words, we can see that "security of encryption scheme plus proof-carrying data for a carefully chosen compliance predicate implies targeted malleability".)

(Moreover, a technical difference between the security property of PCDs and targeted malleability is that we formalized the former as an extraction property whereas Boneh et al. formalized the latter as a simulatability property. It turns out that these two views are equivalent, also in other settings. See Section 13.)

Improved construction. Boneh et al. in their construction only assumed preprocessing publicly-verifiable SNARKs with a very weak succinctness requirement: the length of a proof is at most a multiplicative factor γ smaller than the size of the witness generating it; they did not pose any restrictions on the verifier running time (which potentially can be as long as the time to directly verify the original computation). With this assumption they showed how to construct targeted malleability for a pre-determined constant number of nodes, and using a different long reference string for each node.

Proving the existence of such SNARKs via black-box reductions to standard assumptions is not ruled out by the result of Gentry and Wichs [GW11]; on the other hand, the closest primitive that we know is the much more powerful preprocessing publicly-verifiable SNARKs of Groth [Gro10] (see Section 9) based on a Knowledge of Exponent assumption. Unfortunately, the weak succinctness requirements suggested by Boneh et al. are not enough for being used in our PCD Bootstrapping Theorem. We find it an interesting question to explore whether there are ways to generically "improve" these succinctness properties in a manner similar to what we did in this paper to go form preprocessing SNARKs to SNARKs with no preprocessing. (A first observation is that γ-compressing preprocessing publicly-verifiable SNARKs imply extractable collision-resistant hash functions [BCCT11] with compression factor close to γ; unfortunately, the compression factor is not enough to construct designated-verifier SNARKs form them as in [BCCT11], where a super-constant compression factor seems to be necessary.)

Nonetheless, since targeted malleability is anyways a special case of proof-carrying data, we can at least exhibit much better constructions of targeted malleability, by making assumptions as weak as the existence of preprocessing designated-verifier SNARKs (with succinctness as defined by us), by using the results in this paper to obtain polynomially-long chains of nodes for targeted malleability, a problem that was left open by [BSW11].[17] Furthermore, our solution does not suffer from preprocessing as in [BSW11].

[17] More precisely, in order to apply our PCD Bootstrapping Theorem, we first need to tweak $\mathbb{C}^{\mathsf{TM}}_{\mathsf{pk},\mathsf{ek},\mathcal{F}}$ to have polynomial depth.

12.2 Computing on Authenticated Data / Homomorphic Signatures

Boneh and Freeman [BF11] and Ahn et al. [ABC+11] recently introduced definitions for homomorphically computing on signatures as a way to compute on authenticated data, and thereby provide integrity guarantees for distributed computations for the supported functionalities.

The integrity guarantees of these two works follow a definition of unforgettability against chosen-message attack that still enables signatures for further messages to be derived, as long as these messages are "compliant". And, indeed, proof-carrying data can once again express the integrity guarantees desired in this setting via a simple compliance predicate: given a verification key vk and a "sub-compliance predicate" $\mathcal{C}$ determining which derivations are permitted,

$\mathbb{C}^{\mathsf{AD}}_{\mathsf{vk},\mathcal{C}}(\mathsf{z_o}; \vec{\mathsf{z}}_\mathsf{i}, \mathsf{prog}) \stackrel{\text{def}}{=}$

1. Source Case
   If $\vec{\mathsf{z}}_\mathsf{i} = \bot$ and $\mathsf{prog} = \bot$:
   (a) Verify that $\mathsf{z_o}$ is equal to $(\texttt{unsigned}, \mathsf{tag}, z)$ for some z.
2. Sign Case
   If $\vec{\mathsf{z}}_\mathsf{i} = ((\texttt{unsigned}, \mathsf{tag}, z))$ and $\mathsf{z_o} = (\texttt{signed}, \mathsf{tag}, z')$:
   (a) Verify that $z' = z$ and prog is a valid signature for (z', tag) under verification key vk.
3. Compute Case
   If $\vec{\mathsf{z}}_\mathsf{i} = ((\texttt{signed}, \mathsf{tag}, z_i))_i$ and $\mathsf{z_o} = (\texttt{signed}, \mathsf{tag}, z)$:
   (a) Interpret prog as a function f and verify that $f \in \mathcal{F}$.
   (b) Verify that $\mathcal{C}(z; (z_i)_i, \mathsf{prog})$ accepts.
4. Reject in all other cases.

We thus see that "security of signature scheme plus proof-carrying data for a carefully chosen compliance predicate implies computing on authenticated data".

Technical differences. We should mention that both [BF11] and [ABC+11] insisted on the ability to homomorphically compute on signatures without the help of the messages; syntactically, this is not exactly what we obtain by ensuring integrity with proof-carrying data, because previous messages are required for generating the proof of the previous message. However, we do not see how this can be a drawback in most natural applications.

Also, [ABC+11] give a definition of unforgettability that is stronger than [BF11]; we indeed achieve the stronger notion.

Improved construction. Both [BF11] and [ABC+11] only obtained security against chosen-message attack in the random-oracle model, and only selective security in the plain model; this, only for specific functionalities (constant-degree multivariate functions in the former, and quoting, subsets, weighted sums, averages, Fourier transforms in the latter). On the other hand, by leveraging proof-carrying data in the plain model, we obtain adaptive security in the plain model for any functionality.

(It is interesting to note here that even in the random-oracle model it is not clear how to do better than [BF11] or [ABC+11]. Namely, it is not known how to construct proof-carrying data in the random oracle model, but only in a model where every party has access to a simple signing functionality [CT10].)

Privacy. Both works consider the possibility that one may wish to hide the history of derivation of a signature. This notion is called *context hiding*, and it comes in two flavors: *weak context hiding* and *strong context hiding*. We only achieve the first.[18]

[18] This requires the use of *zero-knowledge* PCDs, which we shall formally define in a later version of this paper.

13 Integrity from Extraction or Simulation?

We have used "extraction-based" definitions to formulate the integrity guarantees of SNARKs (Definition 4.2) and PCDs (Definition 5.6). An alternative approach would have been to formulate them via a "simulation-based" definition, in a way that is more similar to, for example, the definition of [BSW11]. We wish to discuss how these two approaches are related.

13.1 First Warm Up: ECRHs

In order to illustrate what we mean between a definition based on *extraction* versus a definition based on *simulation*, we begin with a very simple example: extractable functions versus simulatable functions.

Recall that an *extractable collision-resistant function* (ECRH) [BCCT11] is simply a collision resistant function family $\mathcal{H}$ that is also an extractable function ensemble in the following sense:

Definition 13.1 (Function Extractability)**.** *We say that $\mathcal{H}$ is* **extractable** *if for any polynomial-size adversary $\mathcal{A}$ there exists a polynomial-size extractor $\mathcal{E}_{\mathcal{A}}$ such that for all large enough $k \in \mathbb{N}$ and any auxiliary input $z \in \{0,1\}^{\mathrm{poly}(k)}$:*

$$\Pr_{h \leftarrow \mathcal{H}_k}\left[\begin{array}{c|c} y \in \mathsf{Image}(h) & y \leftarrow \mathcal{A}(z,h) \\ h(x) \neq y & x \leftarrow \mathcal{E}_{\mathcal{A}}(z,h) \end{array}\right] \leq \mathrm{negl}(k) \ .$$

A natural question to ask is how does the above notion compare against the following seemingly weaker alternative notion:

Definition 13.2 (Function Simulatability)**.** *We say that $\mathcal{H}$ is* **simulatable** *if for any polynomial-size adversary $\mathcal{A}$ there exists a polynomial-size simulator $\mathcal{S}_{\mathcal{A}}$ such that for all large enough $k \in \mathbb{N}$ and any auxiliary input $z \in \{0,1\}^{\mathrm{poly}(k)}$ the following two distributions are computationally indistinguishable:*

$$\left\{(z,h,\tilde{y}) \;\middle|\; \begin{array}{c} y \leftarrow \mathcal{A}(z,h) \\ \tilde{y} \leftarrow \circ k ?_{\mathsf{real}}(h,y) \end{array}\right\}_{h \leftarrow \mathcal{H}} \text{ and } \left\{(z,h,\tilde{y}) \;\middle|\; \begin{array}{c} (y,x) \leftarrow \mathcal{S}(z,h) \\ \tilde{y} \leftarrow \circ k ?_{\mathsf{sim}}(h,y,x) \end{array}\right\}_{h \leftarrow \mathcal{H}} ,$$

where $\circ k ?_{\mathsf{real}}(h,y)$ outputs y if $y \in \mathsf{Image}(h)$ and $\perp$ otherwise, and $\circ k ?_{\mathsf{sim}}(h,y,x)$ outputs y if $h(x) = y$ and $\perp$ otherwise.

It turns out, however, that the above two definitions are equivalent:

Lemma 13.3. *A function ensemble $\mathcal{H}$ is extractable if and only if it is simulatable.*

Proof. Clearly, if $\mathcal{H}$ is extractable then it is also simulatable: the simulator may simply run the extractor.

Conversely, suppose that $\mathcal{H}$ is not extractable, that is, there exists a non-negligible fraction of $h \in \mathcal{H}$ for which the extractor $\mathcal{E}$ does not work. Then, for any candidate simulator $\mathcal{S}$, define the distinguisher $\mathcal{D}_{\mathcal{S}}$ as follows:

$\mathcal{D}_{\mathcal{S}}(z,h,\tilde{y}) \stackrel{\mathrm{def}}{=}$

1. If $\tilde{y} = \perp$, output a random bit. Otherwise continue.
2. Compute $x \leftarrow \mathcal{S}(z,h)$.
3. If $h(x) = \tilde{y}$, output "`simulation`" else output "`real world`".

In order to prove success of the distinguisher $\mathcal{D}_{\mathcal{S}}$ it suffices to not that it must be the case that any candidate simulator must induce the output $\tilde{y} = \perp$ with probability negligibly close to that of $\mathcal{A}$ (or else the "$\perp$ test" distinguisher would work). □

13.2 Second Warm Up: Soundness & Proof of Knowledge

Next, we compare extraction versus simulatability in a more standard setting: soundness and proof of knowledge for proof systems. For convenience, we shall use a non-interactive formulation of proof system that leverages the notation we have already introduced for SNARKs (see Section 4), though here we will not be concerned with succinctness properties as we are only discussing security notions.

We can directly capture various notions of soundness and proof-of-knowledge via an extraction-based definition:

Definition 13.4 (Extractability)**.** *We say that a proof system $(\mathcal{G}, \mathcal{P}, \mathcal{V})$ for the relation $\mathcal{R}$ is* **extractable** *if for any prover $\mathcal{P}^*$ there exists a an extractor $\mathcal{E}_{\mathcal{P}^*}$ such that for all large enough $k \in \mathbb{N}$ and all auxiliary inputs $z \in \{0,1\}^{\mathrm{poly}(k)}$:*

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

Note that:

(i) If both $\mathcal{P}^*$ and $\mathcal{E}_{\mathcal{P}^*}$ may be inefficient, the above requirement is **soundness***.*

(ii) If $\mathcal{P}^*$ may be inefficient but $\mathcal{E}_{\mathcal{P}^*}$ must be efficient, the above requirement is **proof of knowledge***.*

(iii) If $\mathcal{P}^*$ must be efficient but $\mathcal{E}_{\mathcal{P}^*}$ may be inefficient, he above requirement is **computational soundness***.*

(iv) If both $\mathcal{P}^*$ and $\mathcal{E}_{\mathcal{P}^*}$ must be efficient, the above requirement is **computational proof of knowledge***.*

Also: $(ii) \to (i) \to (iii)$, $(ii) \to (iv) \to (iii)$, and (i) and (iv) are incomparable.

Once again a natural question to ask is how does the above notion compare against the following seemingly weaker alternative notion:

Definition 13.5 (Simulatability)**.** *We say that a proof system $(\mathcal{G}, \mathcal{P}, \mathcal{V})$ for the relation $\mathcal{R}$ is* **simulatable** *if for any prover $\mathcal{P}^*$ there exists a a simulator $\mathcal{S}_{\mathcal{P}^*}$ such that for all large enough $k \in \mathbb{N}$ and all auxiliary inputs $z \in \{0,1\}^{\mathrm{poly}(k)}$ the following two distributions are computationally indistinguishable:*

$$\left\{(z,\sigma,\tilde{y}) \;\middle|\; \begin{array}{c}(y,\pi)\leftarrow\mathcal{A}(z,h)\\ \tilde{y}\leftarrow \mathtt{ok?}_{\mathsf{real}}(\tau,y,\pi)\end{array}\right\}_{(\sigma,\tau)\leftarrow\mathcal{G}(1^k)} \text{ and } \left\{(z,\sigma,\tilde{y}) \;\middle|\; \begin{array}{c}(y,w)\leftarrow\mathcal{S}(z,h)\\ \tilde{y}\leftarrow \mathtt{ok?}_{\mathsf{sim}}(\tau,y,w)\end{array}\right\}_{(\sigma,\tau)\leftarrow\mathcal{G}(1^k)} ,$$

where $\mathtt{ok?}_{\mathsf{real}}(\tau,y,\pi)$ outputs y if $\mathcal{V}(\tau,y,\pi)=1$ and $\perp$ otherwise, and $\mathtt{ok?}_{\mathsf{sim}}(\tau,y,w)$ outputs y if $(y,w)\in\mathcal{R}$ and $\perp$ otherwise.

Note that:

- *As in Definition 13.4, we have four flavors of simulatability, depending on whether the prover $\mathcal{P}^*$ or the simulator $\mathcal{S}_{\mathcal{P}^*}$ are required to be efficient (with analogous implications between the four flavors).*
- *We have the choice of considering a stronger requirement where the above two distributions are statistically indistinguishable (and not merely computationally so).*

It turns out here too that the above two definitions are (mostly) equivalent:

Lemma 13.6. *If a proof system $(\mathcal{G}, \mathcal{P}, \mathcal{V})$ is extractable then it is simulatable (when considering both with the same "flavor").*

If a proof system is simulatable with an efficient simulators then it is extractable (when considering both with the same "flavor").

Proof. Clearly, if $(\mathcal{G}, \mathcal{P}, \mathcal{V})$ is extractable (in one flavor), then it is also simulatable (in the same flavor).

For the converse direction, we can reason in the same way as we did in Lemma 13.3. □

Remark 13.7. Note that that in Section 13.1 when we discussed extraction versus simulation for functions, we did not raise the possibility that either the adversary or the extractor (or simulator) may be inefficient, because any function would trivially satisfy the definition under either of these relaxations. Because in this section we use proofs (as opposed to "ground truth") to establish the "goodness" of the output of an adversary in the real world (via the $\mathtt{ok?}_{\mathsf{real}}$ function), the (three) relaxations of the strongest definition (where both the adversary and extractor/simulator are efficient) are quite meaningful (i.e., non-trivial).

Also, in Section 13.1 there was not any need to consider the stronger requirement of statistical indistinguishability because we did not consider relaxations that required it.

Remark 13.8. One may want to consider definitions where the auxiliary input is restricted. The first thing to note is that requiring the definition to hold for all auxiliary input strings z is equivalent to requiring the definition to hold for all auxiliary input distributions $\mathcal{Z}$ (that may be inefficiently samplable). A weaker requirement would be to let the adversary $\mathcal{A}$ begin first by choosing an *efficient* auxiliary input distribution $\mathcal{Z}$ depending on σ, from his auxiliary input will be drawn. (Note that this distribution cannot be inefficient, else it would "invert" the σ.) An even weaker requirement would be to require the definition to hold only for all auxiliary input distributions $\mathcal{Z}$ that are efficiently samplable.

The "flavor" of auxiliary input distribution is preserved in the aforementioned implications between extraction and simulation.

13.3 Case of Interest: PCDs

We now come to comparing extraction versus simulatability in the setting of PCDs. As discussed, the purpose of this section is to better understand in a larger context our choice of security definition of PCDs.

Let us repeat

Definition 13.9 (Extractability)**.** *A $\mathbb{C}$-compliance* **proof-carrying data system** $(\mathbb{G}, \mathbb{P}_{\mathbb{C}}, \mathbb{V}_{\mathbb{C}})$ *is* **extractable** *if for every prover $\mathbb{P}^*$ there exists a knowledge-extractor $\mathbb{E}_{\mathbb{P}^*}$ such that for any large enough security parameter k and all auxiliary inputs $z \in \{0,1\}^{\mathrm{poly}(k)}$:*

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

Note that:

(i) If both $\mathbb{P}^*$ and $\mathbb{E}_{\mathbb{P}^*}$ may be inefficient, the above requirement is **soundness**.

(ii) If $\mathbb{P}^*$ may be inefficient but $\mathbb{E}_{\mathbb{P}^*}$ must be efficient, the above requirement is **proof of knowledge**.

(iii) If $\mathbb{P}^*$ must be efficient but $\mathbb{E}_{\mathbb{P}^*}$ may be inefficient, he above requirement is **computational soundness**.

(iv) If both $\mathbb{P}^*$ and $\mathbb{E}_{\mathbb{P}^*}$ must be efficient, the above requirement is **computational proof of knowledge**.

Also: $(ii) \to (i) \to (iii)$, $(ii) \to (iv) \to (iii)$, *and* (i) *and* (iv) *are incomparable.*

The analogous simulatability definition is the following:

Definition 13.10 (Simulatability)**.** *A* $\mathbb{C}$*-compliance* **proof-carrying data system** $(\mathbb{G}, \mathbb{P}_{\mathbb{C}}, \mathbb{V}_{\mathbb{C}})$ *is* **simulatable** *if for any prover* $\mathbb{P}^*$ *there exists a a simulator* $\mathbb{S}_{\mathbb{P}^*}$ *such that for all large enough* $k \in \mathbb{N}$ *and all auxiliary inputs* $z \in \{0,1\}^{\mathrm{poly}(k)}$ *the following two distributions are computationally indistinguishable:*

$$\left\{ (z, \sigma, \tilde{\mathsf{z}}) \;\middle|\; \begin{array}{c} (\mathsf{z}, \pi) \leftarrow \mathbb{P}^*(z, \sigma) \\ \tilde{\mathsf{z}} \leftarrow \mathit{ok}?_{\mathsf{real}}(\tau, \mathsf{z}, \pi) \end{array} \right\}_{(\sigma,\tau) \leftarrow \mathbb{G}(1^k)} \text{and} \left\{ (z, \sigma, \tilde{\mathsf{z}}) \;\middle|\; \begin{array}{c} \mathsf{T}_{\mathsf{z}} \leftarrow \mathbb{S}_{\mathbb{P}^*}(z, \sigma) \\ \tilde{\mathsf{z}} \leftarrow \mathit{ok}?_{\mathsf{sim}}(\tau, \mathsf{T}_{\mathsf{z}}) \end{array} \right\}_{(\sigma,\tau) \leftarrow \mathbb{G}(1^k)},$$

where $\mathit{ok}?_{\mathsf{real}}(\tau, \mathsf{z}, \pi)$ *outputs* z *if* $\mathbb{V}_{\mathbb{C}}(\tau, \mathsf{z}, \pi) = 1$ *and* $\perp$ *otherwise, and* $\mathit{ok}?_{\mathsf{sim}}(\tau, \mathsf{T}_{\mathsf{z}})$ *outputs* z *if* $\mathbb{C}(\mathsf{T}_{\mathsf{z}})$ *and* $\perp$ *otherwise (and where* z *is the data on the sink of* T_{z}*).*

Note that:

- *As in Definition 13.9, we have four flavors of simulatability, depending on whether the prover* $\mathbb{P}^*$ *or the simulator* $\mathbb{S}_{\mathbb{P}^*}$ *are required to be efficient (with analogous implications between the four flavors).*
- *We have the choice of considering a stronger requirement where the above two distributions are statistically indistinguishable (and not merely computationally so).*

It turns out here too that the above two definitions are (mostly) equivalent:

Lemma 13.11. *If a PCD system* $(\mathbb{G}, \mathbb{P}_{\mathbb{C}}, \mathbb{V}_{\mathbb{C}})$ *is extractable then it is simulatable (when considering both with the same "flavor").*

If a proof system is simulatable with efficient simulators then it is extractable (when considering both with the same "flavor").

Proof. Clearly, if $(\mathbb{G}, \mathbb{P}_{\mathbb{C}}, \mathbb{V}_{\mathbb{C}})$ is extractable (in one flavor), then it is also simulatable (in the same flavor).

For the converse direction, we can reason in the same way as we did in Lemma 13.3. □

Remark 13.12. One may wish to relax the above definitions by, for example, letting the compliance predicate be drawn from a family of compliance predicates, instead of requiring the definition to hold for every compliance predicate. This may be particularly convenient when, say, the compliance predicate is indexed with some sort of public key.

14 Other Related Work

Knowledge assumptions. A popular class of knowledge assumptions, which have been successfully used to solve a number of (at times notoriously open) cryptographic problems, is that of *Knowledge of Exponent* (KE) assumptions. These have the following flavor: if an efficient circuit, given the description of a finite group along with some other public information, computes a list of group elements that satisfies a certain algebraic relation, then there exists a knowledge extractor that outputs some related values that "explain" how the public information was put together to satisfy the relation. Most such assumptions have been proven secure against generic algorithms (see Nechaev [Nec94], Shoup [Sho97], and Dent [Den06]), thus offering some evidence for their truth. In the following we briefly survey prior works which relied on Knowledge of Exponent assumptions.

Damgård [Dam92] first introduced a Knowledge of Exponent assumption to construct a CCA-secure encryption scheme. Later, Hada and Tanaka [HT98] showed how to use two Knowledge of Exponent assumptions to construct the first three-round zero-knowledge argument. Bellare and Palacio [BP04] then showed that one of the assumptions of [HT98] was likely to be false, and proposed a modified assumption, using which they constructed a three-round zero-knowledge argument.

More recently, Abe and Fehr [AF07] extended the assumption of [BP04] to construct the first perfect NIZK for NP with "full" adaptive soundness. Prabhakaran and Xue [PX09] constructed statistically-hiding sets for trapdoor DDH groups [DG06] using a new Knowledge of Exponent assumption. Gennaro et al. [GKR10] used another Knowledge of Exponent assumption (with an interactive flavor) to prove that a modified version of the Okamoto-Tanaka key-agreement protocol [OT89] satisfies perfect forward secrecy against fully active attackers.

In a different direction, Canetti and Dakdouk [CD08, CD09, Dak09] study *extractable functions*. Roughly, a function f is extractable if finding a value x in the image of f implies knowledge of a preimage of x. The motivation of Canetti and Dakdouk for introducing extractable functions is to capture the abstract essence of prior knowledge assumptions, and to formalize the "knowledge of query" property that is sometimes used in proofs in the Random Oracle Model. They also study which security reductions are "knowledge-preserving" (e.g., whether it possible to obtain extractable commitment schemes from extractable one-way functions). In this direction, Bitansky et al. [BCCT11] showed how SNARKs can be used to construct a variety of extractable cryptographic primitives (and can themselves be built out of extractable collision-resistant hashes).

Delegation of computation. An important folklore application of succinct arguments is *delegation of computation* schemes: to delegate the computation of a function F on input x, the delegator sends F and x to the worker, the worker responds with a claimed output z for the computation, and then the delegator and worker engage in a succinct argument, respectively taking the roles of the verifier and the prover, so the delegator can be convinced that indeed $z = F(x)$.[19] In fact, because succinct arguments can "support" all of NP, the worker can also contribute his own input x' to the computation, and prove claims of the form "I know x' such that $z = F(x, x')$"; this is particularly valuable when the succinct argument has a proof-of-knowledge property, because the delegator can deduce that x' can be found efficiently (and not only that such an x' exists).[20]

When the succinct argument is a SNARK, the corresponding delegation scheme is particularly convenient. For example, if the SNARK is of the designated-verifier kind, then the worker can simply generate and send the reference string σ to the worker (and keep the corresponding verification state τ as a private state), along with any information about the computation to delegate (which the delegator can do since σ is independent of the statement to be proven), and the worker can reply with an adaptively-chosen statement and a proof. Recently, Bitansky et al. [BCCT11] showed how to construct such designated-verifier SNARKs from a simple and generic non-standard primitive, extractable collision-resistant hashes, and showed that this primitive is in fact necessary. (Concurrently to our work [DFH11, GLR11] also had similar ideas; see

[19] Of course, the first two messages of this interaction (namely, the delegator sending F and x to the worker, and the worker responding with the claimed output z) can be sent in parallel to messages of the following succinct argument if the succinct argument allows for it. For example, this is the case in Kilian's protocol [Kil92], where the first message from the verifier to the prover is independent of the statement being proved, and the prover can reply the verifier's first message with an adaptively-chosen statement; thus, overall, Kilian's protocol yields a four-message delegation of computation scheme.

[20] For example, the untrusted worker may store a long database x' whose short Merkle tree hash $h = \text{MT}(x')$ is known to the delegator; the delegator may then ask the worker to compute $F(x')$ for some function F. However, from the delegator's perspective, merely being convinced that "there exists $\tilde{x'}$ such that $h = \text{MT}(\tilde{x'})$ and $F(\tilde{x'}) = f$" is not enough. The delegator should also be convinced that the worker knows such a $\tilde{x'}$, which implies due to collision resistance of the Merkle tree hash that indeed $\tilde{x'} = z$.

[BCCT11] for more details on the comparison.)

When the SNARK is publicly-verifiable, the corresponding delegation scheme is even more convenient, because one can set up a global reference string σ for everyone to use, without the need for delegators to contact workers beforehand.

Finally, sometimes in delegation of computation one also cares about confidentiality, and not only soundness, guarantees. It is folklore the fact that succinct arguments (and thus SNARKs too) can be trivially combined with fully-homomorphic encryption [Gen09] (in order to ensure privacy) to obtain a delegation scheme with similar parameters.

Delegation of computation with preprocessing. Within the setting of delegation, where the same delegator may be asking an untrusted worker to evaluate an expensive function on many different inputs, even the weaker *preprocessing* approach may still be meaningful. In such a setting, the delegator performs a one-time function-specific expensive setup phase, followed by inexpensive input-specific delegations to amortize the initial expensive phase. Indeed, in the preprocessing setting a number of prior works have already achieved constructions where the online stage is only two messages [GGP10, CKV10, AIK10]. These constructions do not allow for an untrusted worker to contribute his own input to the computation, namely they are "P-delegation schemes" rather than "NP-delegation schemes". Note that all of these works do not rely on any knowledge assumption; indeed, the impossibility results of [GW11] only apply for NP and not for P.

However, even given that the preprocessing model is very strong, all of the mentioned works suffer from an important handicap: soundness over many delegations only as long as the verifier's answers remain secret. (A notable exception is the work of Benabbas et al. [BGV11], though their constructions are not generic, and are only for specific functionalities such as polynomial functions.)

Goldwasser et al. [GKR08] construct interactive proofs for log-space uniform NC where the verifier running time is quasi-linear. When combining [GKR08] with the PIR-based squashing technique of Kalai and Raz [KR06], one can obtain a succinct two-message delegation scheme. Canetti et al. [CRR11] introduce an alternative way of squashing [GKR08], in the preprocessing setting; their scheme is of the public coin type and hence the verifier's answers need not remain secret (another bonus is that the preprocessing state is publicly verifiable and can thus be used by anyone).

The techniques we develop for removing preprocessing do not seem to apply to any of the above works, because our techniques seem to inherently require the ability to delegate (with knowledge) "all" of NP, whereas the above works only support P computations.

Acknowledgments

We thank Daniel Wichs for discussions of the preprocessing constructions of Groth and Lipmaa, and for pointing out a mistake in a previous draft of this paper. We thank Yuval Ishai for valuable comments and discussions.

References

[ABC+11] Jae Hyun Ahn, Dan Boneh, Jan Camenisch, Susan Hohenberger, Abhi Shelat, and Brent Waters. Computing on authenticated data. Cryptology ePrint Archive, Report 2011/096, 2011.

[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.

[AF07] Masayuki Abe and Serge Fehr. Perfect NIZK with adaptive soundness. In *Proceedings of the 4th Theory of Cryptography Conference*, TCC '07, pages 118–136, 2007.

[AIK10] Benny Applebaum, Yuval Ishai, and Eyal Kushilevitz. From secrecy to soundness: Efficient verification via secure computation. In *Proceedings of the 37th International Colloquium on Automata, Languages and Programming*, ICALP '10, pages 152–163, 2010.

[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.

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

[BCCT11] Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again. Cryptology ePrint Archive, Report 2011/443, 2011.

[BF11] Dan Boneh and David Mandell Freeman. Homomorphic signatures for polynomial functions. Cryptology ePrint Archive, Report 2011/018, 2011.

[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.

[BGV11] Siavosh Benabbas, Rosario Gennaro, and Yevgeniy Vahlis. Verifiable delegation of computation over large datasets. In *Proceedings of the 31st Annual International Cryptology Conference*, CRYPTO '11, pages 111–131, 2011.

[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. Fast reductions from RAMs to delegatable succinct constraint satisfaction problems, 2012. Cryptology ePrint Archive.

[BSS08] Eli Ben-Sasson and Madhu Sudan. Short PCPs with polylog query complexity. *SIAM Journal on Computing*, 38(2):551–607, 2008.

[BSW11] Dan Boneh, Gil Segev, and Brent Waters. Targeted malleability: Homomorphic encryption for restricted computations. Cryptology ePrint Archive, Report 2011/311, 2011.

[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.

[CD08] Ran Canetti and Ronny Ramzi Dakdouk. Extractable perfectly one-way functions. In *Proceedings of the 35th International Colloquium on Automata, Languages and Programming*, ICALP '08, pages 449–460, 2008.

[CD09] Ran Canetti and Ronny Ramzi Dakdouk. Towards a theory of extractable functions. In *Proceedings of the 6th Theory of Cryptography Conference*, TCC '09, pages 595–613, 2009.

[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.

[CRR11] Ran Canetti, Ben Riva, and Guy N. Rothblum. Two 1-round protocols for delegation of computation. Cryptology ePrint Archive, Report 2011/518, 2011.

[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.

[Dak09] Ronny Ramzi Dakdouk. *Theory and Application of Extractable Functions*. PhD thesis, Yale University, Computer Science Department, December 2009.

[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.

[Den06] Alexander W. Dent. The hardness of the DHK problem in the generic group model. Cryptology ePrint Archive, Report 2006/156, 2006.

[DFH11] Ivan Damgård, Sebastian Faust, and Carmit Hazay. Secure two-party computation with low communication. Cryptology ePrint Archive, Report 2011/508, 2011.

[DG06] Alexander Dent and Steven Galbraith. Hidden pairings and trapdoor DDH groups. In Florian Hess, Sebastian Pauli, and Michael Pohst, editors, *Algorithmic Number Theory*, volume 4076 of *Lecture Notes in Computer Science*, pages 436–451. 2006.

[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`.

[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.

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

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[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.

[GKR10] Rosario Gennaro, Hugo Krawczyk, and Tal Rabin. Okamoto-Tanaka revisited: Fully authenticated Diffie-Hellman with minimal overhead. In *Proceedings of the 8th International Conference on Applied Cryptography and Network Security*, ACNS '10, pages 309–328, 2010.

[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.

[GOS06] Jens Groth, Rafail Ostrovsky, and Amit Sahai. Non-interactive zaps and new techniques for NIZK. In *Proceedings of the 11th Annual International Cryptology Conference*, CRYPTO '06, pages 97–111, 2006.

[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.

[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.

[HT98] Satoshi Hada and Toshiaki Tanaka. On the existence of 3-round zero-knowledge protocols. In *Proceedings of the 18th Annual International Cryptology Conference*, CRYPTO '98, pages 408–423, 1998.

[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.

[Lip11] Helger Lipmaa. Progression-free sets and sublinear pairing-based non-interactive zero-knowledge arguments. Cryptology ePrint Archive, Report 2011/009, 2011.

[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.

[Nec94] Vassiliy Ilyich Nechaev. Complexity of a determinate algorithm for the discrete logarithm. *Mathematical Notes*, 55:165–172, 1994.

[OT89] Eiji Okamoto and Kazue Tanaka. Key distribution system based on identification information. *Selected Areas in Communications, IEEE Journal on*, 7(4):481–485, May 1989.

[PX09] Manoj Prabhakaran and Rui Xue. Statistically hiding sets. In *Proceedings of the The Cryptographers' Track at the RSA Conference 2009*, CT-RSA 2009, pages 100–116, 2009.

[Sho97] Victor Shoup. Lower bounds for discrete logarithms and related problems. In *Proceedings of the International Conference on the Theory and Application of Cryptographic Techniques*, EUROCRYPT '97, pages 256–266, 1997.

[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.