Cluster Computing in Zero Knowledge

(extended version)

Alessandro Chiesa
alessandro.chiesa@inf.ethz.ch
ETH Zurich

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

Madars Virza
madars@mit.edu
MIT

April 28, 2015

Abstract

Large computations, when amenable to distributed parallel execution, are often executed on computer clusters, for scalability and cost reasons. Such computations are used in many applications, including, to name but a few, machine learning, webgraph mining, and statistical machine translation. Oftentimes, though, the input data is private and only the result of the computation can be published. Zero-knowledge proofs would allow, in such settings, to verify correctness of the output without leaking (additional) information about the input.

In this work, we investigate theoretical and practical aspects of *zero-knowledge proofs for cluster computations*. We design, build, and evaluate zero-knowledge proof systems for which: (i) a proof attests to the correct execution of a cluster computation; and (ii) generating the proof is itself a cluster computation that is similar in structure and complexity to the original one. Concretely, we focus on MapReduce, an elegant and popular form of cluster computing.

Previous zero-knowledge proof systems can in principle prove a MapReduce computation's correctness, via a monolithic NP statement that reasons about all mappers, all reducers, and shuffling. However, it is not clear how to generate the proof for such monolithic statements via parallel execution by a distributed system. Our work demonstrates, by theory and implementation, that proof generation can be similar in structure and complexity to the original cluster computation.

Our main technique is a bootstrapping theorem for succinct non-interactive arguments of knowledge (SNARKs) that shows how, via recursive proof composition and Proof-Carrying Data, it is possible to transform any SNARK into a *distributed SNARK for MapReduce* which proves, piecewise and in a distributed way, the correctness of every step in the original MapReduce computation as well as their global consistency.

Keywords: zero knowledge, cluster computing, MapReduce, proof-carrying data, succinct arguments

Contents

1 Introduction

We study theoretical and concrete aspects of *zero-knowledge proofs for cluster computations*, seeking proofs for which: (i) the output of the cluster computation carries a zero-knowledge proof of its correctness; and (ii) generating a proof is itself a cluster computation that is similar in structure and complexity to the original one.

1.1 Motivation

Consider the following motivating example. A server owns a private database $\mathtt{x}$, and a client wishes to learn $\mathtt{y} := F(\mathtt{x})$ for a public function F, selected either by himself or someone else. A (hiding) commitment cm to $\mathtt{x}$ is known publicly. For example, $\mathtt{x}$ may be a database containing genetic data, and F may be a machine-learning algorithm that uses the genetic data to compute a classifier $\mathtt{y}$. On the one hand, the client seeks *integrity of computation*: he wants to ensure that the server reports the correct output $\mathtt{y}$ (because the classifier $\mathtt{y}$ may be used for critical medical decisions). On the other hand, the server seeks *confidentiality* of his own input: he is willing to disclose $\mathtt{y}$ to the client, but no additional information about $\mathtt{x}$ beyond $\mathtt{y}$ (because the genetic data $\mathtt{x}$ may contain sensitive personal information).

Zero-knowledge proofs. Achieving the combination of the aforementioned security requirements seems paradoxical; after all, the client does not have the input $\mathtt{x}$, and the server is not willing to share it. Nevertheless, cryptography offers a powerful tool that is able to do just that: *zero-knowledge proofs* [GMR89]. More precisely, the server, acting as the prover, attempts to convince the client, acting as the verifier, that the following NP statement is true: "there exists $\tilde{\mathtt{x}}$ such that $\mathtt{y} = F(\tilde{\mathtt{x}})$ and $\tilde{\mathtt{x}}$ is a decommitment of cm". Indeed: (a) the proof system's *soundness* property addresses the client's integrity concern, because it guarantees that, if the NP statement is false, the prover cannot convince the verifier (with high probability);[1] and (b) the proof system's *zero-knowledge* property addresses the server's confidentiality concern, because it guarantees that, if the NP statement is true, the prover can convince the verifier without leaking any information about $\mathtt{x}$ (beyond was is leaked by the output $\mathtt{y}$).

Cluster computations. When F is amenable to parallel execution by a distributed system, it is often desirable, for scalability and cost reasons, to compute $\mathtt{y} := F(\mathtt{x})$ on a *computer cluster*. A computer cluster consists of nodes (e.g., commodity machines) connected via a network, and each node performs local computations as coordinated via messages with other nodes. Thus, to compute $F(\mathtt{x})$, a cluster may break $\mathtt{x}$ down into chunks and use these to assign sub-tasks to different nodes; the results of these sub-tasks may require further computation, so that nodes further coordinate, deduce more sub-tasks, and so on, until the final result $\mathtt{y}$ can be collected. Parallel execution by a distributed system is possible in many settings, including the aforementioned one of running machine-learning algorithms on private genetic data. Indeed, "cloud" service providers do offer users distributed programming interfaces (e.g., Amazon's "EMR" and Rackspace's "Big Data", both of which use the Hadoop framework).

The problem: how to do cluster computing in zero knowledge? In principle, any zero-knowledge proof system for NP can be used to express an NP statement that captures F's correct execution.

However, while F may have been efficient to execute on a computer cluster, the process of generating a proof attesting to its correctness may not be. Suppose, for example, that the NP statement to be proved must be expressed as an instance of circuit satisfiability. Then, one would have to construct a single circuit that expresses the correctness of the computation of every node in the cluster, as well as the correctness of communication among them. Proving the satisfiability of the resulting monolithic circuit via off-the-shelf zero-knowledge proof systems is a computation that looks nothing like the original one and, moreover, may not be suitable for efficient execution on a cluster.

Ideally, the proving process should be a distributed computation that is similar to the original one, in that the complexity of producing the proof is not much larger than that of the original computation and, likewise, has a cluster-friendly communication structure (compare Figure 1 and Figure 2).

We thus ask the following question:

To what extent can one efficiently perform cluster computing in zero knowledge?

[1]Sometimes a property stronger than soundness is required: *proof of knowledge* [GMR89, BG93], which guarantees that, whenever the client is convinced, not only can he deduce that a witness exists, but also that the prover *knows* one such witness.

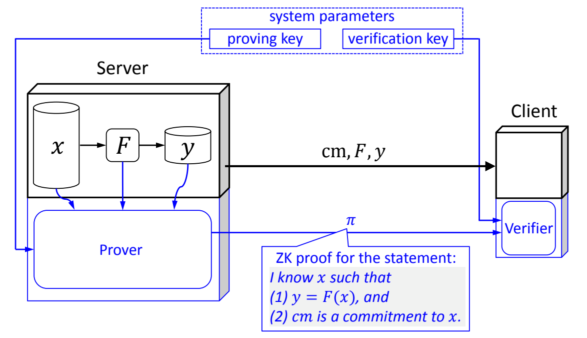

Figure 1: In the classical setting of zero-knowledge proofs, (i) the computation being proved is executed on a single machine, and (ii) the proving process is itself a computation that is executed on a single machine.

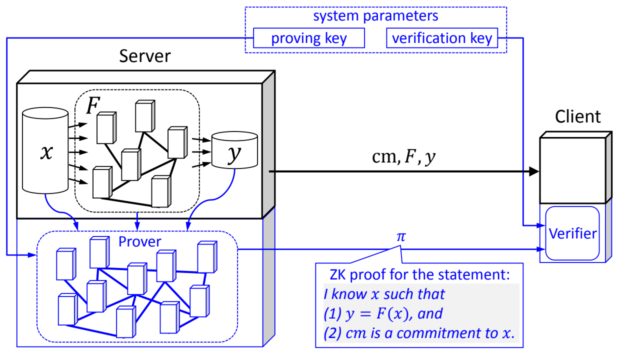

Figure 2: We study zero-knowledge proofs for cluster computations. In this setting, (i) the computation being proved is a cluster computation, and (ii) the proving process is itself a cluster computation (that is similar in structure and complexity to the original one).

1.2 Our focus: MapReduce

Cluster computing is a hypernym that encompasses numerous forms of distributed computing, as determined by the cluster's architecture (i.e., its programming model and its execution framework). Indeed, a cluster's architecture often depends on the class of envisioned applications (e.g., indexing the World Wide Web, performing astrophysical N-body simulations, executing machine-learning algorithms on genetic data, and so on).

In this work, we focus on a concrete, yet elegant and powerful, distributed architecture: *MapReduce* [DG04]. We review MapReduce later (in Section 2), and now only say that MapReduce can express many useful computations, including ones used for machine learning [CKLY$^+$06, WHK08, PHBB09], graph mining and processing [LS10, KCF12], statistical machine translation [BPXOD07, DCML08, LD10, PWB12], document similarity [Lin09], and bioinformatics [LSLPS09, Sch09]. For concreteness, we specialize to MapReduce the question raised in Section 1.1:

> Can one obtain zero-knowledge proofs attesting to the correctness of MapReduce computations, in which the proving process is itself distributed and can be efficiently expressed via MapReduce computations?

1.3 Our contributions

In this paper we present two main results, both contributing to the feasibility of cluster computing in zero knowledge.

1. **MapReduce in zero knowledge.** Under knowledge-of-exponent assumptions [Dam92, HT98, BP04], we construct a zero-knowledge proof system in which: (i) a proof attests to the correct execution of a MapReduce computation; and (ii) generating a proof consists of MapReduce computations with similar complexity as the original one. Moreover, the proof system is succinct and non-interactive, i.e., is a *zk-SNARK* [GW11, BCCT12, BCIOP13].

2. **A working prototype.** We design, build, and evaluate a working prototype for the aforementioned construction. The code implementing the prototype is integrated with `libsnark` [SCI], a `C++` library for zk-SNARKs.

At the heart of our construction (and implementation) lies a **new bootstrapping theorem** for zk-SNARKs. Informally:

> Assuming collision-resistant hashing, there is an efficient transformation that takes as input a zk-SNARK (even one with expensive preprocessing) and outputs a *distributed zk-SNARK for MapReduce*, i.e., a zk-SNARK for MapReduce where the prover can be efficiently implemented via MapReduce.

The transformation consists of the following two steps.

- **Step I:** use a given (non-distributed) zk-SNARK to obtain a *proof-carrying data* (PCD) system [CT10, CT12], a cryptographic primitive that enforces local invariants, the *compliance predicates*, in distributed computations.

- **Step II:** use the PCD system on a specially-crafted predicate to obtain a distributed zk-SNARK for MapReduce.

The theory for the first step is due to [BCCT13]; a special case was implemented in [BCTV14a], and our implementation generalizes it to support the MapReduce application. The second step is novel and is an example of using "compliance engineering" to conduct and prove correctness of non-trivial distributed computations. From an implementation standpoint, both steps require significant and careful engineering, as we explain later.

1.4 Prior work

zk-SNARKs. We study zero-knowledge proofs [GMR89] that are non-interactive [BFM88, NY90, BDSMP91]. Specifically, we study non-interactive zero-knowledge proofs that are *succinct*, i.e., short and easy to verify [Mic00]; these are known as *zk-SNARKs* [GW11, BCCT12, BCIOP13].

There are many zk-SNARK constructions in the literature, with different properties in efficiency and supported languages. In *preprocessing zk-SNARKs*, the complexity of the setup of public parameters grows with the size of the computation being proved [Gro10, Lip12, BCIOP13, GGPR13, PGHR13, BCGTV13, Lip13, FLZ13, BCTV14b, Lip14, KPPS$^+$14, ZPK14, DFGK14, WSRBW15, BBFR15, CFHK$^+$15]; in *fully-succinct zk-SNARKs*, that complexity is independent of computation size [Mic00, Val08, Mie08, DL08, BCCT12, DFH12, GLR11, BC12, BCCT13, BCTV14a, BCCG$^+$14]. Working prototypes have been achieved both for preprocessing zk-SNARKs [PGHR13, BCGTV13, BCTV14b, KPPS$^+$14, ZPK14, CFHK$^+$15] and for fully-succinct ones [BCTV14a]. Several works have also explored more in depth various applications of zk-SNARKs [CKLM13, BFRS$^+$13, DFKP13, BCGG$^+$14, FL14].

Prior work has not sought (or achieved) distributed zk-SNARKs for MapReduce. Of course, non-distributed zk-SNARKs for MapReduce (i.e., where the prover is not amenable to parallel distributed execution) can be achieved, trivially, via any zk-SNARK for NP: (a) express (the correctness of) the MapReduce computation via a suitable NP statement; then (b) prove satisfiability of that NP statement by using the zk-SNARK.

Finally, known zk-SNARK constructions (including the one in this work) rely on fairly strong assumptions. This may be partially justified in light of the work of Gentry and Wichs [GW11], which shows that no non-interactive succinct argument can be proven sound via a black-box reduction to a falsifiable assumption [Nao03].

Proof-carrying data. *Proof-Carrying Data* (PCD) [CT10, CT12] is a framework for enforcing local invariants in distributed computations; it is captured via a cryptographic primitive called *PCD system*. Proof-Carrying Data covers, as special examples, incrementally-verifiable computation [Val08] and targeted malleability [BSW12]. Its role in bootstrapping zk-SNARKs was shown in [BCCT13], and an implementation of it was achieved in [BCTV14a].

Outsourcing MapReduce computations. Braun et al. [BFRS$^+$13] construct (and implement) an interactive protocol for verifiably outsourcing MapReduce computations to untrusted servers. When interacting with the prover, the client has to perform himself the MapReduce shuffling phase; hence, their protocol is neither succinct nor zero knowledge.

Schuster et al. [SCFG$^+$15] construct (and implement) a protocol for verifiably outsourcing MapReduce computations to a cluster in which each node is equipped with a trusted processor. Their protocol additionally provides certain confidentiality guarantees (the client may hide from the cluster all information about the outsourced computation except for some "structural" details such as key-repetition patterns in the shuffle phase).

Both of the above works do not obtain a zk-SNARK and, a fortiori, neither a distributed zk-SNARK.

Other works on outsourcing computations. Numerous works [GKR08, KR09, AIK10, CKV10, GGP10, BGV11, CRR12, CTY11, SBW11, FG12, SMBW12, SVPB$^+$12, SBVB$^+$13, CMT12, TRMP12, VSBW13, Tha13, BFRS$^+$13, BTVW14, PR14] seek to verifiably outsource various classes of computation to untrusted powerful servers, e.g., in order to leverage cheaper cycles or storage. Some of these works have achieved working prototypes of their protocols.

Verifiable outsourcing of computations *is not our goal*. Rather, we study theoretical and practical aspects of zero-knowledge proofs for cluster computations. Zero-knowledge proofs are useful even when applied to relatively-small computations, and even with high overheads (e.g., see [MGGR13] for a recent example).[2]

1.5 Summary of challenges and techniques

Our construction (and implementation) rely on a new bootstrapping theorem for zk-SNARKs: any zk-SNARK can be transformed into a distributed zk-SNARK for MapReduce. The transformation is done in two steps, as follows.

[2] In this paper's setting, the client does not have the server's input, and so cannot conduct the computation on his own. It is thus *not meaningful* to compare "efficiency of outsourced computation at the server" and "efficiency of native execution at the client", since the latter was never an option.

1.5.1 From the zk-SNARK to a multi-predicate PCD system

The transformation's first step uses the given zk-SNARK to construct a *PCD system* [CT10, CT12], a cryptographic primitive that enforces a given local invariant, known as the *compliance predicate*, in distributed computations. Such a transformation was described by [BCCT13], following [Val08] and [CT10]. It was implemented by [BCTV14a], and used for obtaining scalable zero-knowledge proofs for random-access machine executions.

These prior works are constrained to enforcing a single compliance predicate at all nodes in the distributed computation. However, in MapReduce computations (as in many others), different nodes are subject to different requirements. In principle one can create a single compliance predicate expressing the disjunction of all these requirements; but the resulting predicate is large (its cost is the sum of each requirement's cost) and entails a large cost in proving time.

We thus extend [BCTV14a] to define, construct, and implement a *multi-predicate PCD system*, where different nodes may be subject to different compliance predicates, and yet the cost of producing the proof, at each node, depends merely on the compliance predicate to which this particular node is subject. The presence of multiple compliance predicates complicates the construction of the arithmetic circuits for performing recursive proof composition, as these must now verify a zk-SNARK proof relative to one out of a (potentially large) number of compliance predicates, each with its own verification key, at a cost that is essentially independent of the predicates that are not locally relevant.

Additional restrictions in the prior works, which we also relax, are that node arity (the number of input messages to a node) was fixed, and that a node's input lengths had to equal its output length. While not fundamental, these limitations cause sizable overheads in heterogenous distributed computations (of which MapReduce is an example).

1.5.2 From a multi-predicate PCD system to a distributed zk-SNARK for MapReduce

The transformation's second step uses the aforementioned multi-predicate PCD system to construct a distributed zk-SNARK for MapReduce.

For each individual mapper node or reducer node, correctness of the local computation is independent of other computations; so it is fairly straightforward to distill local "map" and "reduce" compliance predicates. However, the shuffle phase of the MapReduce computation is a global computation that involves all of the mappers' outputs. We wish to ensure *globally* correct shuffling, while only enforcing (via the PCD system) the preservation of a compliance predicate, *locally* at each node. (Of course, one could always consider a big shuffler node that takes all the shuffled messages as inputs, but doing so would prevent the proof generation from being distributed.)

We thus show how to decompose correct shuffling into a collection of simple local predicates, while preserving zero knowledge (which introduces subtleties). Roughly, we show that there is a parallel distributed algorithm to simultaneously compute, for each unique key k, a proof attesting that the list of values associated to k in the output of the shuffling process contains all the those values, and only those, that were paired with k by some mapper.

Subsequently, we use the map and reduce compliance predicates, along with those used to prove correct shuffling, and obtain a collection of compliance predicates with the property that any distributed computation that is complaint with these corresponds to a correct MapReduce computation.

Note how the extensions to basic PCD, mentioned in Section 1.5.1, come into play. First, we specify multiple compliance predicate, for the different stages of the computation, and only pay for the applicable one at every point. Second, because MapReduce computation has a communication pattern that is input-dependent and not very homogenous, we require PCD to support (directly and thus more efficiently) flexible communication patterns, with variable node arity and varying input and output message lengths.

1.6 Roadmap

The rest of this paper is organized as follows. After introducing basic definitions in Section 2, we define distributed zk-SNARKs for MapReduce in Section 3, and multi-predicate PCD systems in Section 4. We describe Step II of our transformation (compliance engineering using PCD) in Section 5, and Step I in (constructing PCD) in Section 6. In Section 7 we describe the implementation, and in Section 8 we provide an evaluation of it. In Section 9 we conclude with open problems. The appendices contain additional details, and are referenced from within the paper.

2 Preliminaries

We give notations and definitions needed for this paper's technical discussions.

2.1 Basic notations

We denote by λ the security parameter. We write $f = O_\lambda(g)$ to mean that there is $c > 0$ such that $f = O(\lambda^c g)$. We write $|a|$ to denote the number of bits needed to store a (whether a be a vector, a circuit, and so on); if a is a vector, $\mathsf{len}(a)$ denotes the number of components in a. We write $\mathsf{cost}(A)$ to denote the cost of computing A: if A is a machine, this is its running time; if A is a circuit, this is its number of gates.

To simplify notation, we do not make explicit adversaries' auxiliary inputs. We also do not make explicit the public parameters of some cryptographic primitives (e.g., commitment schemes, collision-resistant functions); such primitives require a parameter setup that involves sampling from a certain distribution and then publishing the result.

2.2 Commitments

A *commitment scheme* is a pair $\mathsf{COMM} = (\mathsf{COMM.Gen}, \mathsf{COMM.Ver})$ with the following syntax:

- $\mathsf{COMM.Gen}(z) \to (\mathsf{cm}, \mathsf{cr})$. On input data z, the *commitment generator* $\mathsf{COMM.Gen}$ probabilistically samples a commitment cm of z and corresponding commitment randomness cr.
- $\mathsf{COMM.Ver}(z, \mathsf{cm}, \mathsf{cr}) \to b$. On input data z, commitment cm, and commitment randomness cr, the *commitment verifier* $\mathsf{COMM.Ver}$ outputs $b = 1$ if cm is a valid commitment of z with respect to the randomness cr.

The scheme COMM satisfies the natural completeness, (computational) binding, and (statistical) hiding properties. We assume that cm does not even leak $|z|$, and thus $|\mathsf{cm}|$ is a fixed polynomial in the security parameter.

2.3 Merkle trees

We use Merkle trees [Mer89] (based on some collision-resistant function) as non-hiding succinct commitments to lists of values, in the familiar way. A *Merkle-tree scheme* is a tuple $\mathsf{MERKLE} = (\mathsf{MERKLE.GetRoot}, \mathsf{MERKLE.GetPath}, \mathsf{MERKLE.CheckPath})$ with the following syntax:

- $\mathsf{MERKLE.GetRoot}(\vec{z}) \to \mathsf{rt}$. Given input list $\vec{z} = (z_i)_{i=1}^n$, the *root generator* $\mathsf{MERKLE.GetRoot}$ deterministically computes a root rt of the Merkle tree with the list $\vec{z}$ at its leaves.
- $\mathsf{MERKLE.GetPath}(\vec{z}, i) \to \mathsf{ap}$. Given input list $\vec{z}$ and index i, the *authentication path generator* $\mathsf{MERKLE.GetPath}$ deterministically computes the authentication path ap for z_i.
- $\mathsf{MERKLE.CheckPath}(\mathsf{rt}, i, z_i, \mathsf{ap}) \to b$. Given root rt, input data z_i, index i, and authentication path ap, the *path checker* $\mathsf{MERKLE.CheckPath}$ outputs $b = 1$ if ap is a valid path for z_i as the i-th leaf in a Merkle tree with root rt.

The scheme MERKLE satisfies the natural completeness and (computational) binding properties.

2.4 MapReduce

We recall MapReduce, introduce useful notation for MapReduce, and then mention some extensions of MapReduce to which our results continue to apply.

2.4.1 Overview of MapReduce

MapReduce is a programming model for describing data-parallel computations to be run on computer clusters [DG04]. A *MapReduce job* consists of two functions, Map and Reduce, and an input, $\mathtt{x}$, which is a list of key-value pairs; executing the job results into an output, $\mathtt{y}$, which also is a list of key-value pairs. Computing $\mathtt{y}$ requires three phases: (i) *Map phase:* the function Map is separately invoked on each key-value pair in the list $\mathtt{x}$; each such invocation produces an intermediate sub-list of key-value pairs. (ii) *Shuffle phase:* all the intermediate sub-lists of key-value pairs are jointly shuffled so that pairs that share the same key are gathered together into groups. (iii) *Reduce phase:* the function Reduce is separately invoked on each group of key-value pairs; each such invocation produces an output key-value pair; all these pairs are concatenated (in some order) to form $\mathtt{y}$.

Naturally, efficiently computing the three phases on a computer cluster requires a suitable framework to assign computers to Map tasks, implement the distributed shuffle of intermediate key-value pairs, assign computers to Reduce tasks, and collect the various outputs; this is typically orchestrated by a master node. For now, we focus on the definition of the programming model and not the details of a framework that implements it.

2.4.2 Notation for MapReduce

We introduce notation that enables us to discuss MapReduce in more detail.

Keys, values, and records. First, we discuss the data associated to a MapReduce job. The main "unit of data" is a *record*, which is a pair (k, v) where k is its *key* and v is its *value*. We distinguish between different kinds of records, depending on which phase they belong to: input records are of *phase* 1 and lie in $\mathcal{K}^1 \times \mathcal{V}^1$; intermediate records are of *phase* 2 and lie in $\mathcal{K}^2 \times \mathcal{V}^2$; and output records are of *phase* 3 lie in $\mathcal{K}^3 \times \mathcal{V}^3$.

MapReduce pairs. Next, we discuss the functions associated to a MapReduce job. A *MapReduce pair* is a pair $(\mathsf{Map}, \mathsf{Reduce})$ where $\mathsf{Map}\colon \mathcal{K}^1 \times \mathcal{V}^1 \to (\mathcal{K}^2 \times \mathcal{V}^2)^*$ is its *Map function* and $\mathsf{Reduce}\colon \mathcal{K}^2 \times (\mathcal{V}^2)^* \to (\mathcal{K}^3 \times \mathcal{V}^3)$ is its *Reduce function*; both must run in polynomial time. In other words, on input a phase-1 record $(k^1, v^1) \in (\mathcal{K}^1 \times \mathcal{V}^1)$, Map outputs a list of phase-2 records $\big((k_i^2, v_i^2)\big)_i \in (\mathcal{K}^2 \times \mathcal{V}^2)^*$. Instead, on input a phase-2 key $k^2 \in \mathcal{K}^2$ and a list of phase-2 values $(v_i^2)_i \in (\mathcal{V}^2)^*$, Reduce outputs a phase-3 record $(k^3, v^3) \in (\mathcal{K}^3 \times \mathcal{V}^3)$.

MapReduce executions. Finally, we discuss how functions operate on data so to *execute* a MapReduce job. Given a MapReduce pair $(\mathsf{Map}, \mathsf{Reduce})$ and an input $\mathbb{x} \in (\mathcal{K}^1 \times \mathcal{V}^1)^*$, the output of the execution of $(\mathsf{Map}, \mathsf{Reduce})$ on $\mathbb{x}$, denoted $[\mathsf{Map}, \mathsf{Reduce}](\mathbb{x})$, is the result $\mathbb{y} \in (\mathcal{K}^3 \times \mathcal{V}^3)^*$ of the following (abstract) computation (see Figure 3).

1. **Map step.** For each $i \in \{1, \dots, \mathsf{len}(\mathbb{x})\}$, letting (k_i^1, v_i^1) be the i-th phase-1 record in $\mathbb{x}$, compute the list of phase-2 records $\big((k_{i,j}^2, v_{i,j}^2)\big)_j := \mathsf{Map}(k_i^1, v_i^1)$. This step produces a list of intermediate records $\mathbb{z} = \big((k_{i,j}^2, v_{i,j}^2)\big)_{i,j}$.
2. **Shuffle step.** Shuffle the list $\mathbb{z}$ so that records with the same key are grouped together. This step induces, for each unique key k^2 appearing in $\mathbb{z}$, a corresponding list $\vec{v^2}$ of values paired with k^2.
3. **Reduce step.** For each unique phase-2 key k^2 in $\mathbb{z}$ and its corresponding list of phase-2 values $\vec{v^2}$, compute the phase-3 record $(k^3, v^3) = \mathsf{Reduce}(k^2, \vec{v^2})$. The output $\mathbb{y}$ equals the concatenation of all of these phase-3 records.

We note that MapReduce jobs enjoy certain "symmetries" (which simplify the task of execution on clusters): the order of records in $\mathbb{x}$ or in $\mathbb{y}$ is irrelevant.[3] In terms of complexity measures, we say that the execution of $(\mathsf{Map}, \mathsf{Reduce})$ on $\mathbb{x}$ is (m, r, p)-*bounded* if each individual execution of Map takes at most m time, each individual execution of Reduce takes at most r time, and $\mathsf{len}(\mathbb{x}) \cdot m + \mathsf{len}(\mathbb{y}) \cdot r \leq p$ (where $\mathbb{y} := [\mathsf{Map}, \mathsf{Reduce}](\mathbb{x})$).[4]

The MapReduce language. We express, via a suitable language, the notion of "correct" MapReduce executions:

Definition 2.1. *For a MapReduce pair* $(\mathsf{Map}, \mathsf{Reduce})$*, the language* $\mathscr{L}_{(\mathsf{Map},\mathsf{Reduce})}$ *consists of the tuples* $(\mathbb{x}, \mathbb{y})$ *for which* $\mathbb{y} = [\mathsf{Map}, \mathsf{Reduce}](\mathbb{x})$.[5]

In this work, we consider the setting where an input $\mathbb{x}$ is not known to the user, but only its commitment cm is (as $\mathbb{x}$ is private). Thus, we work with a related relation, $\mathscr{R}^{\mathsf{COMM}}_{(\mathsf{Map},\mathsf{Reduce})}$, derived from $\mathscr{L}_{(\mathsf{Map},\mathsf{Reduce})}$ and a commitment scheme $\mathsf{COMM} = (\mathsf{COMM.Gen}, \mathsf{COMM.Ver})$ (using the syntax introduced in Section 2.2). In contrast to $\mathscr{L}_{(\mathsf{Map},\mathsf{Reduce})}$, instances in $\mathscr{R}^{\mathsf{COMM}}_{(\mathsf{Map},\mathsf{Reduce})}$ contain cm instead of $\mathbb{x}$, and witnesses are extended to contain decommitment information (i.e., the input and commitment randomness). More precisely, we define the relation $\mathscr{R}^{\mathsf{COMM}}_{(\mathsf{Map},\mathsf{Reduce})}$ as follows.

Definition 2.2. *For a MapReduce pair* $(\mathsf{Map}, \mathsf{Reduce})$ *and commitment scheme* COMM*, the relation* $\mathscr{R}^{\mathsf{COMM}}_{(\mathsf{Map},\mathsf{Reduce})}$ *consists of instance-witness pairs* $\big((\mathsf{cm}, \mathbb{y}), (\mathbb{x}, \mathsf{cr})\big)$ *such that* $\mathsf{COMM.Ver}(\mathbb{x}, \mathsf{cm}, \mathsf{cr}) = 1$ *and* $(\mathbb{x}, \mathbb{y}) \in \mathscr{L}_{(\mathsf{Map},\mathsf{Reduce})}$.[6]

[3] One only considers Map and Reduce functions that do not introduce asymmetries (by, e.g., leveraging the order of elements in a list).

[4] For simplicity, we ignore the cost of shuffling because it is typically on the order of the input and output sizes [GM12].

[5] Due to symmetry, $(\mathbb{x}, \mathbb{y}) \in \mathscr{L}_{(\mathsf{Map},\mathsf{Reduce})}$ if and only if $\big(\pi(\mathbb{x}), \pi'(\mathbb{y})\big) \in \mathscr{L}_{(\mathsf{Map},\mathsf{Reduce})}$ for any two permutations π and π' (of records).

[6] One may wonder why we define (and work with) the relation $\mathscr{R}^{\mathsf{COMM}}_{(\mathsf{Map},\mathsf{Reduce})}$ instead of using the corresponding language $\mathscr{L}^{\mathsf{COMM}}_{(\mathsf{Map},\mathsf{Reduce})}$. This is because the commitment scheme COMM is only computationally binding (as cm is much shorter than $\mathbb{x}$), and thus the language is not meaningful: it contains instances $(\mathsf{cm}, \mathbb{y})$ for which $\mathbb{y}$ is the output of the MapReduce computation on some *false* opening of the commitment cm (which cannot be found efficiently, but still exists). We thus always speak of proving *knowledge* of a witness fulfilling the above relation (for a given instance).

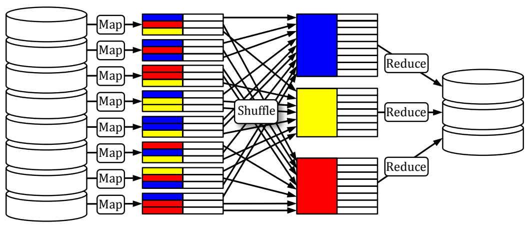

Figure 3: A single MapReduce execution.

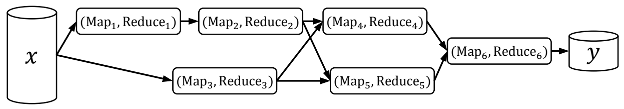
x
$(\mathrm{Map}_1, \mathrm{Reduce}_1)$
$(\mathrm{Map}_2, \mathrm{Reduce}_2)$
$(\mathrm{Map}_3, \mathrm{Reduce}_3)$
$(\mathrm{Map}_4, \mathrm{Reduce}_4)$
$(\mathrm{Map}_5, \mathrm{Reduce}_5)$
$(\mathrm{Map}_6, \mathrm{Reduce}_6)$
y

Figure 4: An execution of a MapReduce sequence of depth 6. Here, $I_1 = \{0\}$, $I_2 = \{1\}$, $I_3 = \{0\}$, $I_4 = \{2, 3\}$, $I_5 = \{2, 3\}$, $I_6 = \{4, 5\}$.

MapReduce sequences. A single MapReduce execution may be insufficient to run an algorithm. In such cases, instead of a single MapReduce pair, we consider a *MapReduce sequence* $\mathbf{S}$: a list $\big((I_i, \mathsf{Map}_i, \mathsf{Reduce}_i)\big)_{i=1}^{d}$ such that, for each i, $I_i \subseteq \{0, \dots, i-1\}$ and $(\mathsf{Map}_i, \mathsf{Reduce}_i)$ is a MapReduce pair. We call d the *depth of* $\mathbf{S}$. The output of the execution of $\mathbf{S}$ on an input $\mathtt{x}$, denoted $\mathbf{S}(\mathtt{x})$, is the result $\mathtt{y}$ obtained as follows: (1) set $\mathtt{y}^{(0)} := \mathtt{x}$; (2) for $i = 1, \dots, d$, compute $\mathtt{y}^{(i)} := [\mathsf{Map}_i, \mathsf{Reduce}_i](\mathtt{x}^{(i)})$ where $\mathtt{x}^{(i)}$ is the concatenation of all $\mathtt{y}^{(j)}$ with $j \in I_i$; (3) output $\mathtt{y} := \mathtt{y}^{(d)}$;[7] see Figure 4. In terms of complexity measures, similarly to above, we say that the execution of $\mathbf{S}$ on $\mathtt{x}$ is (m, r, p)*-bounded* if each individual execution of any Map_i takes at most m time, each individual execution of any Reduce_i takes at most r time, and $\sum_{i=1}^{d}(\mathsf{len}(\mathtt{x}^{(i-1)}) \cdot m + \mathsf{len}(\mathtt{x}^{(i)}) \cdot r) \le p$.

Family of MapReduce sequences. A *family of MapReduce sequences* is a family $(\mathbf{S}_N)_{N \in \mathbb{N}}$ where each $\mathbf{S}_N$ is a MapReduce sequence $\big((I_{N,i}, \mathsf{Map}_{N,i}, \mathsf{Reduce}_{N,i})\big)_{i=1}^{d_N}$.

2.4.3 Some extensions of MapReduce

Sometimes one considers a more general definition of MapReduce in which the Reduce function outputs a *list* of phase-3 records; instead, the above discussions focus on the case where the Reduce function outputs a *single* record.

In addition, the cryptographic setting that we consider motivates other extensions that are not typically considered. Specifically, it is natural to let a Map or Reduce function take two additional inputs: (i) *MapReduce parameters*, which allow a user to specify different settings even after a MapReduce pair has been fixed (e.g., zk-SNARK keys for a particular pair have already been generated); and (ii) an *auxiliary input*, which provides a further non-deterministic input that, unlike the "primary" input $\mathtt{x}$, is not fixed by the commitment cm.

This paper's results extend to cover the extensions above (and, in fact, our working prototype supports them).

[7]Of course, for such a computation to be well-defined, the domains and ranges of individual MapReduce executions must match up.

3 Definition of distributed zk-SNARKs for MapReduce

We (informally) define non-distributed zk-SNARKs for MapReduce, and then distributed zk-SNARKs for MapReduce. Throughout, we assume familiarity with the notations and definitions for MapReduce introduced in Section 2.4.

3.1 Non-distributed zk-SNARKs for MapReduce

A (non-distributed) *zk-SNARK for MapReduce* is a zk-SNARK for proving knowledge of witnesses in $\mathscr{R}^{\mathsf{COMM}}_{(\mathsf{Map},\mathsf{Reduce})}$, for a user-specified MapReduce pair $(\mathsf{Map},\mathsf{Reduce})$ and a fixed choice of commitment scheme COMM. That is, it is a cryptographic primitive that provides short and easy-to-verify non-interactive zero-knowledge proofs of knowledge for the relation $\mathscr{R}^{\mathsf{COMM}}_{(\mathsf{Map},\mathsf{Reduce})}$. Concretely, the primitive consists of a tuple $(\mathsf{COMM},\mathsf{MR.KeyGen},\mathsf{MR.Prove},\mathsf{MR.Verify})$ with the following syntax.

- $\mathsf{MR.KeyGen}(1^\lambda,\mathsf{Map},\mathsf{Reduce}) \to (\mathsf{pk},\mathsf{vk})$. On input a security parameter λ (presented in unary) and a MapReduce pair $(\mathsf{Map},\mathsf{Reduce})$, the *key generator* $\mathsf{MR.KeyGen}$ probabilistically samples a proving key pk and a verification key vk. We assume, without loss of generality, that pk contains (a description of) the MapReduce pair $(\mathsf{Map},\mathsf{Reduce})$.

The keys pk and vk are published as public parameters and can be used, any number of times, to prove/verify knowledge of witnesses in the relation $\mathscr{R}^{\mathsf{COMM}}_{(\mathsf{Map},\mathsf{Reduce})}$, as follows.

- $\mathsf{MR.Prove}(\mathsf{pk},\mathsf{cm},\mathbb{y},\mathbb{x},\mathsf{cr}) \to \pi_{\mathsf{MR}}$. On input a proving key pk, instance $(\mathsf{cm},\mathbb{y})$, and witness $(\mathbb{x},\mathsf{cr})$, the *prover* $\mathsf{MR.Prove}$ outputs a proof π_{MR} for the statement "there is $(\mathbb{x},\mathsf{cr})$ such that $\big((\mathsf{cm},\mathbb{y}),(\mathbb{x},\mathsf{cr})\big) \in \mathscr{R}^{\mathsf{COMM}}_{(\mathsf{Map},\mathsf{Reduce})}$".

- $\mathsf{MR.Verify}(\mathsf{vk},\mathsf{cm},\mathbb{y},\pi_{\mathsf{MR}}) \to b$. On input a verification key vk, commitment cm, output $\mathbb{y}$, and proof π_{MR}, the *verifier* $\mathsf{MR.Verify}$ outputs $b=1$ if he is convinced that there is $(\mathbb{x},\mathsf{cr})$ such that $\big((\mathsf{cm},\mathbb{y}),(\mathbb{x},\mathsf{cr})\big) \in \mathscr{R}^{\mathsf{COMM}}_{(\mathsf{Map},\mathsf{Reduce})}$.

As in other zk-SNARKs, the above tuple satisfies (variants of) the properties of completeness, succinctness, (computational) proof of knowledge, and (statistical) zero knowledge; we describe these in Appendix A. Here we recall succinctness: an honestly-generated proof π_{MR} has $O_\lambda(1)$ bits, and $\mathsf{MR.Verify}(\mathsf{vk},\mathsf{cm},\mathbb{y},\pi_{\mathsf{MR}})$ runs in time $O_\lambda(|\mathbb{y}|)$.

Costs of key generation. The above implies that $(\mathsf{pk},\mathsf{vk})$ is generated in time $O_\lambda(1)\cdot\mathsf{poly}(\mathsf{cost}(\mathsf{Map})+\mathsf{cost}(\mathsf{Reduce}))$, $|\mathsf{pk}| = O_\lambda(1)\cdot\mathsf{poly}(\mathsf{cost}(\mathsf{Map})+\mathsf{cost}(\mathsf{Reduce}))$, and $|\mathsf{vk}| = O_\lambda(1)$ (since $\mathsf{MR.Verify}$ runs in time $O_\lambda(|\mathbb{y}|)$ for any $\mathbb{y}$). These key-generation costs are between those of a preprocessing zk-SNARK (where key generation costs as much as the *entire* computation being proved) and a fully-succinct zk-SNARK (where key generation costs only a fixed polynomial in λ), because they do not depend on the number of mappers and reducers in the MapReduce computation.

One could strengthen the definition above to require "full succinctness", i.e., to further require that key generation depends polynomially on the security parameter only (and, in particular, that the MapReduce pair is not hard-coded into the keys). The results presented in this paper extend to achieve this stronger definition.

3.2 Distributed zk-SNARKs for MapReduce

A *distributed zk-SNARK for MapReduce* is a zk-SNARK for MapReduce where the prover consists of few MapReduce computations whose overall complexity is similar to the MapReduce computation being proved. More precisely, for every MapReduce pair $(\mathsf{Map},\mathsf{Reduce})$ and $(\mathsf{pk},\mathsf{vk})$ output by $\mathsf{MR.KeyGen}(1^\lambda,\mathsf{Map},\mathsf{Reduce})$, $\mathsf{MR.Prove}(\mathsf{pk},\cdot,\cdot,\cdot,\cdot)$ is a family of MapReduce sequences that is $(\mathsf{Map},\mathsf{Reduce})$-*complexity-preserving*, a property defined below.

Definition 3.1. *Given a MapReduce pair* $(\mathsf{Map},\mathsf{Reduce})$*, a family of MapReduce sequences* $(\mathbf{S}_N)_{N\in\mathbb{N}}$ *is* $(\mathsf{Map},\mathsf{Reduce})$-*complexity-preserving if, for all* $N\in\mathbb{N}$ *and* $\big((\mathsf{cm},\mathbb{y}),(\mathbb{x},\mathsf{cr})\big) \in \mathscr{R}^{\mathsf{COMM}}_{(\mathsf{Map},\mathsf{Reduce})}$ *with* $\mathsf{len}(\mathbb{x})+\mathsf{len}(\mathbb{y}) \le N$*:*

- *the depth of* $\mathbf{S}_N$ *is logarithmic in* N*, i.e.,* $d_N = O(\log N)$*; and*
- $\mathbf{S}_N$ *has a linear overhead compared to* $(\mathsf{Map},\mathsf{Reduce})$*, i.e., for all* $m,r,p\in\mathbb{N}$*, if* $\mathbb{x}$ *is* (m,r,p)*-bounded then the execution of* $\mathbf{S}_N$ *on* $(\mathsf{cm},\mathbb{y},\mathbb{x},\mathsf{cr})$ *is* $(O_\lambda(m),O_\lambda(r),O_\lambda(p))$*-bounded.*

4 Definition of multi-predicate PCD

Proof-carrying data (PCD) [CT10, CT12] is a cryptographic primitive that encapsulates the security guarantees achievable via recursive composition of proofs. Since recursive proof composition naturally involves multiple (physical or virtual) parties, PCD is phrased in the language of a *distributed computation* among computing nodes, who perform local computations, based on local data and input messages, and then produce output messages. Given a *compliance predicate* Π to express local checks, the goal of PCD is to ensure that any given message msg in the distributed computation is Π-*compliant*, i.e., is consistent with a history in which each node's local computation satisfies Π. This formulation covers, as special cases, incrementally-verifiable computation [Val08] and targeted malleability [BSW12].

Extending PCD to multiple predicates. The definition of PCD naturally generalizes to compliance with respect to a *vector* $\vec{\Pi}$ of compliance predicates (rather than a single predicate). Namely, a msg is $\vec{\Pi}$-*compliant* if it is consistent with a history in which each node's local computation satisfies some predicate Π in the vector $\vec{\Pi}$. Moreover, a message msg comprises two parts: the *type*, which records what kind of node output msg, and the *payload*, which is the rest.

The above *multi-predicate PCD* can be "simulated" via a *single-predicate PCD*, by folding all the predicates in the vector $\vec{\Pi}$ into a single predicate $\Pi^{\star}$ that (a) reasons about which predicate in $\vec{\Pi}$ to use at a give node, and (b) enforces a message's type and payload separation. However, this simulation incurs a significant overhead: the cost of $\Pi^{\star}$ is the sum of the costs of all the predicates in $\vec{\Pi}$, and this cost is incurred at every node regardless of which predicate is actually used to check compliance at a node. In contrast, in our construction of multi-predicate PCD (see Section 6), we incur, at each node, only the cost of the predicate that is actually used to check compliance.

Implications for MapReduce. As we discuss in Section 5, reducing the correctness of MapReduce computations to compliance of distributed computations involves multiple predicates that perform checks with different semantics: a predicate for mapper nodes, a predicate for reducer nodes, and various other predicates for other nodes that reason about shuffling. These predicates have different costs and, thus, it is crucial to leverage the flexibility offered by multi-predicate PCD (so to then obtain a distributed zk-SNARK for MapReduce).

Next, we define distributed-computation transcripts (our formal notion of distributed computations), compliance of a transcript T with respect to a given vector $\vec{\Pi}$ of compliance predicates, multi-predicate PCD, and transcript generators.

Transcripts. A *(distributed-computation) transcript* is a tuple $\mathsf{T} = (G, \mathrm{TYPE}, \mathrm{LOC}, \mathrm{PAYLOAD})$, where:

- $G = (V, E)$ is a directed acyclic graph with node set V and edge set $E \subseteq V \times V$;
- $\mathrm{TYPE}\colon V \to \mathbb{N}$ are node labels;
- $\mathrm{LOC}\colon V \to \{0,1\}^*$ are (another kind of) node labels; and
- $\mathrm{PAYLOAD}\colon E \to \{0,1\}^*$ are edge labels.

The *message* of an edge $(v, w) \in E$ is the pair $\mathrm{MSG}(v, w) := (\mathrm{TYPE}(v), \mathrm{PAYLOAD}(v, w))$. The *outputs* of the transcript T, denoted $\mathrm{OUTS}(\mathsf{T})$, is the set of messages $\mathrm{MSG}(\tilde{v}, \tilde{w})$ where $(\tilde{v}, \tilde{w}) \in E$ and $\tilde{w}$ is a sink. Typically, we denote a message by msg, and its type and payload by $\mathsf{msg.type}$ and $\mathsf{msg.payload}$.

Compliant transcripts and messages. A *compliance predicate* Π is a function with a *type*, denoted $\Pi.\mathsf{type}$. Given a vector $\vec{\Pi}$ of compliance predicates, we say that:

- a transcript $\mathsf{T} = (G, \mathrm{LOC}, \mathrm{TYPE}, \mathrm{PAYLOAD})$ is **$\vec{\Pi}$-compliant**, denoted $\vec{\Pi}(\mathsf{T}) = \mathsf{OK}$, if:
  - (i) for each $v \in V$, $\mathrm{TYPE}(v) = 0$ if and only if v is a source; and
  - (ii) for each non-source $v \in V$ and each $w \in \mathsf{children}(v)$, there is $\Pi \in \vec{\Pi}$ with $\mathrm{TYPE}(v) = \Pi.\mathsf{type}$ such that
$$\Pi\Big(\mathrm{MSG}(v, w), \mathrm{LOC}(v), \big(\mathrm{MSG}(u, v)\big)_{u \in \mathsf{parents}(v)}\Big) \text{ accepts.}$$
- a message msg is **$\vec{\Pi}$-compliant** if there is a transcript T such that $\vec{\Pi}(\mathsf{T}) = \mathsf{OK}$ and $\mathsf{msg} \in \mathrm{OUTS}(\mathsf{T})$.

A transcript T thus represents a distributed computation, in the following sense. For each node $v \in V$, the function LOC specifies the *local data* used at v; and, for each edge $(u, v) \in E$, the function MSG specifies the *message* sent from node u to node v. A node v with parent nodes $\mathsf{parents}(v)$ and children nodes $\mathsf{children}(v)$ uses the local data $\mathrm{LOC}(v)$ and the *input messages* $\big(\mathrm{MSG}(u, v)\big)_{u \in \mathsf{parents}(v)}$ to compute the *output message* $\mathrm{MSG}(v, w)$ for each child $w \in \mathsf{children}(v)$. As for the function TYPE, it assigns to each node $v \in V$ a quantity that determines the type of every message output by v; this quantity also determines which compliance predicates can be used to verify compliance of those messages (specifically, the type of the predicate and message must equal).

Multi-predicate PCD systems. A *multi-predicate PCD system* is a triple of polynomial-time algorithms $(\mathbb{G}, \mathbb{P}, \mathbb{V})$, called *key generator*, *prover*, and *verifier*. The key generator $\mathbb{G}$ is given as input a vector of predicates $\vec{\Pi}$, and outputs a proving key pk and verification key vk; these allow anyone to prove/verify that a message msg is $\vec{\Pi}$-compliant. This is achieved by attaching a short and easy-to-verify proof to each message: given pk, input messages $\vec{\mathsf{msg}}_{\mathsf{in}}$ with proofs $\vec{\pi}_{\mathsf{in}}$, local data loc, and an output message msg (allegedly, $\vec{\Pi}$-compliant), the prover $\mathbb{P}$ computes a new proof π to attach to msg; the verifier $\mathbb{V}(\mathsf{vk}, \mathsf{msg}, \pi)$ checks that msg is $\vec{\Pi}$-compliant. The triple $(\mathbb{G}, \mathbb{P}, \mathbb{V})$ must satisfy completeness, succinctness, (computational) proof of knowledge, and (statistical) zero knowledge; we describe these in Appendix B. Here we recall succinctness: an honestly-generated proof π has $O_\lambda(1)$ bits, and $\mathbb{V}(\mathsf{vk}, \mathsf{msg}, \pi)$ runs in time $O_\lambda(|\mathsf{msg}|)$.

Transcript generators. We sometimes think of a transcript as dynamically generated rather than as a static object, in the following sense. A *transcript generator* is an interactive algorithm TGen that works as follows: given the empty string, TGen answers with a first transcript extension te_1; then, given a second empty string, TGen answers with a second transcript extension te_2; and so on, until TGen halts and outputs a set of messages fin. A *transcript extension* is an instruction specifying how to extend the growing transcript T; TGen's output, obtained after all the extensions, equals $\mathrm{OUTS}(\mathsf{T})$. More precisely, the transcript generated by TGen is defined via the following iterative procedure.

- Initialize $\mathsf{T} = (G, \mathrm{TYPE}, \mathrm{LOC}, \mathrm{PAYLOAD})$ to be an empty transcript and $\mathsf{Inactives}$ to be an empty map.
- Start running TGen and, for $i = 1, 2, \ldots$, do the following until TGen halts and outputs fin.
  1. Keep running TGen until it outputs the next transcript extension te_i.
  2. If te_i looks like (v, w, msg), do as follows.
     (a) Check that $(v, w) \notin E$ and $\nexists\, u \in V \text{s.t.} (u, v) \in E$.
     (b) If any of v or w is not in V, add it to V; then add (v, w) to E.
     (c) Check that $\mathsf{msg.type} = 0$; then set $\mathrm{TYPE}(v) := \mathsf{msg.type}$ and $\mathrm{MSG}(v, w) := \mathsf{msg}$.
  3. If te_i looks like $\big((v, w, \mathsf{msg}), \mathsf{loc}, (\vec{u}, \vec{\mathsf{msg}}_{\mathsf{in}}, \vec{\mathsf{b}})\big)$, do as follows.
     (a) Check that: (i) $v \in V$; (ii) $(v, w) \notin E$; (iii) $\mathsf{Inactives}(v, \mathsf{msg}) = 0$; (iv) $\vec{u} = \mathsf{parents}(v)$; (v) $\vec{u}, \vec{\mathsf{msg}}_{\mathsf{in}}, \vec{\mathsf{b}}$ are vectors of a same length d; (vi) for $j = 1, \ldots, d$, $\vec{u}[j] \in V$, $(\vec{u}[j], v) \in E$, and $\mathrm{MSG}(\vec{u}[j], v) = \vec{\mathsf{msg}}_{\mathsf{in}}[j]$.
     (b) If $\exists\, w' \in V$ s.t. $(v, w') \in E$, check that $\mathrm{TYPE}(v) = \mathsf{msg.type}$ and $\mathrm{LOC}(v) = \mathsf{loc}$; otherwise, set $\mathrm{TYPE}(v) := \mathsf{msg.type}$ and $\mathrm{LOC}(v) := \mathsf{loc}$.
     (c) If $w \notin V$ then add w to V; then add (v, w) to E.
     (d) Set $\mathrm{MSG}(v, w) := \mathsf{msg}$.
     (e) For $j = 1, \ldots, d$, if $\vec{\mathsf{b}}[j] = 1$ then set $\mathsf{Inactives}(\vec{u}[j], \vec{\mathsf{msg}}_{\mathsf{in}}[j]) := 1$.
- Check that $\mathsf{fin} = \mathrm{OUTS}(\mathsf{T})$ and that no message in fin has a set bit in $\mathsf{Inactives}$.
- If any check above fails, output an empty transcript; else output T.

The vector $\vec{\mathsf{b}}$ in a transcript extension denotes which non-output messages are used for the last time as an input to a node; $\vec{\mathsf{b}}$ is exposed for efficiency reasons (as explained later down below). We use the language of transcript generators to formalize the completeness property for PCD systems (in Appendix B), as well as to state a theorem (in Section 5.1).

To each transcript generator TGen, we associate a matching *PCD meta-prover* TPrv that, given a proving key pk, generates proofs of compliance on the fly, ultimately producing proofs for the outputs of the distributed computation.

- Initialize $\mathsf{ActiveProofs}$ to be an empty map.
- Start running TGen and, for $i = 1, 2, \ldots$, do the following until TGen halts and outputs fin.
  1. Keep running TGen until it outputs the next transcript extension te_i.
  2. If te_i looks like (v, w, msg), set $\mathsf{ActiveProofs}(v, \mathsf{msg}) := \perp$.
  3. If te_i looks like $\big((v, w, \mathsf{msg}), \mathsf{loc}, (\vec{u}, \vec{\mathsf{msg}}_{\mathsf{in}}, \vec{\mathsf{b}})\big)$, do as follows.
     (a) For $j = 1, \ldots, d$, set $\vec{\pi}_{\mathsf{in}}[j] := \mathsf{ActiveProofs}(\vec{u}[j], \vec{\mathsf{msg}}_{\mathsf{in}}[j])$.
     (b) Compute the PCD proof $\pi := \mathbb{P}(\mathsf{pk}, \mathsf{msg}, \mathsf{loc}, \vec{\mathsf{msg}}_{\mathsf{in}}, \vec{\pi}_{\mathsf{in}})$, and set $\mathsf{ActiveProofs}(v, \mathsf{msg}) := \pi$.
     (c) For $j = 1, \ldots, d$, if $\vec{\mathsf{b}}[j] = 1$ then delete from $\mathsf{ActiveProofs}$ the entry for $(\vec{u}[j], \vec{\mathsf{msg}}_{\mathsf{in}}[j])$.
  4. Output the proofs in $\mathsf{ActiveProofs}$ for the messages in fin.

Note that TPrv does not employ the naive strategy that materializes the entire transcript generated by TGen and then recursively computes proofs over it; instead, using the "hints" provided in the vector of bits $\vec{\mathsf{b}}$, TPrv only stores information about messages and proofs that will be used later or are outputs of the distributed computation. This ensures that the complexity properties of TPrv are typically tightly related to those of TGen.

5 Step II: from multi-predicate PCD to distributed zk-SNARKs for MapReduce

We discuss Step II of our bootstrapping theorem: constructing a distributed zk-SNARK for MapReduce from a multi-predicate PCD system. This step itself consists of two main parts.

- **Compliance engineering** (Section 5.1): a reduction from the correctness of MapReduce computations to a question about the compliance of distributed computations with respect to a certain vector $\vec{\Pi}^{\mathsf{MR}}$ of predicates.
- **Construction of the proof system** (Section 5.2): suitably invoke the multi-predicate PCD system on the vector $\vec{\Pi}^{\mathsf{MR}}$ in order to construct a distributed zk-SNARK for MapReduce.

The combination of compliance engineering and PCD systems, exemplified via the above two steps, is a powerful tool for constructing zero-knowledge proofs for distributed computations: compliance engineering allows us to express the desired properties as the compliance of distributed computations, while PCD systems allow us to prove, in a distributed way (and in zero knowledge), the compliance of such distributed computations.

5.1 Compliance engineering for MapReduce

Given any MapReduce pair $(\mathsf{Map}, \mathsf{Reduce})$, we show how to construct a vector $\vec{\Pi}^{\mathsf{MR}}$ of compliance predicates for which suitable $\vec{\Pi}^{\mathsf{MR}}$-compliant transcripts correspond to instance-witness pairs in the relation $\mathscr{R}^{\mathsf{COMM}}_{(\mathsf{Map},\mathsf{Reduce})}$. Let us first clarify what "suitable" means, via the following definition.

Definition 5.1. *For an instance* $(\mathsf{cm}, \mathtt{y})$*, a transcript* T *is* $(\mathsf{cm}, \mathtt{y})$-compatible *if* $\mathrm{OUTS}(\mathsf{T})$ *contains a message with type* 1 *and payload* $(\mathsf{cm}, \mathsf{len}(\mathtt{y}))$ *and, for each* $i \in \{1, \ldots, \mathsf{len}(\mathtt{y})\}$*, a message with type* 2 *and payload* $(\mathsf{cm}, \mathtt{y}_i)$*.*[8]

Next, via the theorem below, we show how one can translate a question of the form

"Given an instance $(\mathsf{cm}, \mathtt{y})$, is there a witness $(\mathtt{x}, \mathsf{cr})$ such that $\big((\mathsf{cm}, \mathtt{y}), (\mathtt{x}, \mathsf{cr})\big)$ is in $\mathscr{R}^{\mathsf{COMM}}_{(\mathsf{Map},\mathsf{Reduce})}$?"

to a question of the form

"Given an instance $(\mathsf{cm}, \mathtt{y})$, is there a $\vec{\Pi}^{\mathsf{MR}}$-compliant $(\mathsf{cm}, \mathtt{y})$-compatible transcript T?"

Theorem 5.2. *There exists a commitment scheme* COMM *such that, for every MapReduce pair* $(\mathsf{Map}, \mathsf{Reduce})$*, there exist a vector* $\vec{\Pi}^{\mathsf{MR}}$ *of compliance predicates and two algorithms* $\mathsf{TGen}, \mathsf{TExt}$ *satisfying the following properties.*

- EFFICIENCY.
  - *The vector* $\vec{\Pi}^{\mathsf{MR}}$ *consists of* 7 *predicates, with the following sizes and (per-input) costs:*

$$\begin{array}{ll} |\vec{\Pi}^{\mathsf{MR}}[1]| = O_\lambda(|\mathsf{Map}|) & \mathsf{cost}(\vec{\Pi}^{\mathsf{MR}}[1]) = O_\lambda(\mathsf{cost}(\mathsf{Map})) \\ |\vec{\Pi}^{\mathsf{MR}}[2]| = O_\lambda(|\mathsf{Reduce}|) & \mathsf{cost}(\vec{\Pi}^{\mathsf{MR}}[2]) = O_\lambda(\mathsf{cost}(\mathsf{Reduce})) \\ |\vec{\Pi}^{\mathsf{MR}}[3]| = O_\lambda(1) & \mathsf{cost}(\vec{\Pi}^{\mathsf{MR}}[3]) = O_\lambda(1) \\ \vdots & \vdots \\ |\vec{\Pi}^{\mathsf{MR}}[7]| = O_\lambda(1) & \mathsf{cost}(\vec{\Pi}^{\mathsf{MR}}[7]) = O_\lambda(1) \end{array} .$$

  - *The algorithm* TGen *is* $(\mathsf{Map}, \mathsf{Reduce})$*-complexity-preserving.*
  - *The algorithm* TExt *is linear time.*
- COMPLETENESS. *For any instance* $(\mathsf{cm}, \mathtt{y})$*, if there is* $(\mathtt{x}, \mathsf{cr})$ *such that* $\big((\mathsf{cm}, \mathtt{y}), (\mathtt{x}, \mathsf{cr})\big)$ *is in* $\mathscr{R}^{\mathsf{COMM}}_{(\mathsf{Map},\mathsf{Reduce})}$*, then there is a* $\vec{\Pi}^{\mathsf{MR}}$*-compliant* $(\mathsf{cm}, \mathtt{y})$*-compatible transcript* T*; also,* T *is the transcript generated by* $\mathsf{TGen}(\mathsf{cm}, \mathtt{y}, \mathtt{x}, \mathsf{cr})$*.*
- PROOF OF KNOWLEDGE. *For any instance* $(\mathsf{cm}, \mathtt{y})$*, if there is a* $\vec{\Pi}^{\mathsf{MR}}$*-compliant* $(\mathsf{cm}, \mathtt{y})$*-compatible transcript* T*, then* $\mathsf{TExt}(\mathsf{T})$ *outputs* $(\mathtt{x}, \mathsf{cr})$ *such that* $\big((\mathsf{cm}, \mathtt{y}), (\mathtt{x}, \mathsf{cr})\big)$ *is in* $\mathscr{R}^{\mathsf{COMM}}_{(\mathsf{Map},\mathsf{Reduce})}$*.*

We now sketch a proof of the theorem. Recall proof of knowledge: we must construct a vector $\vec{\Pi}^{\mathsf{MR}}$ of predicates with the property that, given $(\mathsf{cm}, \mathtt{y})$, if there is a distributed-computation transcript T that is both $\vec{\Pi}^{\mathsf{MR}}$-compliant and $(\mathsf{cm}, \mathtt{y})$-compatible, then we can find $(\mathtt{x}, \mathsf{cr})$ for which $\mathsf{COMM.Ver}(\mathtt{x}, \mathsf{cm}, \mathsf{cr}) = 1$ and $\mathtt{y} = [\mathsf{Map}, \mathsf{Reduce}](\mathtt{x})$. Intuitively, we achieve proof of knowledge by engineering the predicates in $\vec{\Pi}^{\mathsf{MR}}$ so that the transcript T is forced

[8] Any two distinct types suffice for this definition; we fixed the types 1 and 2 for concreteness.

to encode within it a "history" of a correct MapReduce execution. Technically, the main challenge is that we are restricted to local checks: each predicate only sees input and output messages of a single node; in contrast, correct execution of a MapReduce computation (also) involves global properties, such as correct shuffling.

We introduce our approach in steps, by first describing two "failed attempts". For simplicity, we focus on the (artificial) case where each mapper outputs a *single* phase-2 record; later, we explain how this restriction can be lifted.

5.1.1 Failed attempt #1

It is natural to begin by designing two predicates that simply capture the correct execution of a mapper node and a reducer node, respectively; call these $\Pi^{\mathsf{Map}}_{\mathrm{exe}}$ and $\Pi^{\mathsf{Reduce}}_{\mathrm{exe}}$, and see Figure 6 for pseudocode. Set $\vec{\Pi}^{\mathsf{MR}} := (\Pi^{\mathsf{Map}}_{\mathrm{exe}}, \Pi^{\mathsf{Reduce}}_{\mathrm{exe}})$.

Now suppose that msg is a $\vec{\Pi}^{\mathsf{MR}}$-compliant message. What can we deduce about the history of computations that led to msg? If $\mathsf{msg.type} = \Pi^{\mathsf{Map}}_{\mathrm{exe}}.\mathsf{type}$, then msg was output by a node at which the predicate $\Pi^{\mathsf{Map}}_{\mathrm{exe}}$ was checked; similarly, if $\mathsf{msg.type} = \Pi^{\mathsf{Reduce}}_{\mathrm{exe}}.\mathsf{type}$, then msg was output by a node at which the predicate $\Pi^{\mathsf{Reduce}}_{\mathrm{exe}}$ was checked. Suppose, for example, that $\mathsf{msg.type} = \Pi^{\mathsf{Reduce}}_{\mathrm{exe}}.\mathsf{type}$. By construction of $\Pi^{\mathsf{Reduce}}_{\mathrm{exe}}$, we deduce that: (i) $\mathsf{msg.payload}$ is a phase-3 record (k^3, v^3), and (ii) there is a list of input messages $\vec{\mathsf{msg}}_{\mathsf{in}}$ whose payloads contain phase-2 records $\big((k^2_j, v^2_j)\big)_j$ that all share the same key and, moreover, result in (k^3, v^3) when given as input to Reduce.

However, as soon as we try to "dig further into the past", and deduce what properties each phase-2 record (k^2_j, v^2_j) satisfies, we run into issues not addressed by the present construction of $\Pi^{\mathsf{Map}}_{\mathrm{exe}}$ and $\Pi^{\mathsf{Reduce}}_{\mathrm{exe}}$. Namely,

- **Issue I:** How can we ascertain that each phase-2 record (k^2_i, v^2_i) was the correct output of some mapper node?
- **Issue II:** Even if so, where did that mapper node obtain its input phase-1 record?

5.1.2 Failed attempt #2

We augment $\Pi^{\mathsf{Map}}_{\mathrm{exe}}$ and $\Pi^{\mathsf{Reduce}}_{\mathrm{exe}}$ to address these issues. We address Issue I by inspecting message types: $\Pi^{\mathsf{Map}}_{\mathrm{exe}}$ ensures that its input messages have type 0 (i.e., are not output by previous nodes); while $\Pi^{\mathsf{Reduce}}_{\mathrm{exe}}$ ensures that they have type $\Pi^{\mathsf{Map}}_{\mathrm{exe}}.\mathsf{type}$ (i.e., are output by mapper nodes). We address Issue II by augmenting messages with a commitment cm to the (overall) input $\mathbb{x}$ and extending $\Pi^{\mathsf{Map}}_{\mathrm{exe}}$ to authenticate the received phase-1 record. We now explain these ideas.

First, we describe the commitment scheme COMM that we use to create cm. Essentially, COMM consists of (i) a Merkle-tree followed by a commitment to the resulting root, and also (ii) a commitment to the size of the committed data. See Figure 5 for more details; we denote the underlying commitment scheme by COMM^* and the Merkle-tree scheme by MERKLE (and use notation introduced in Section 2.2 and Section 2.3).

Next, see Figure 7 for pseudocode of the two (updated) predicates $\Pi^{\mathsf{Map}}_{\mathrm{exe}}$ and $\Pi^{\mathsf{Reduce}}_{\mathrm{exe}}$. Set $\vec{\Pi}^{\mathsf{MR}} := (\Pi^{\mathsf{Map}}_{\mathrm{exe}}, \Pi^{\mathsf{Reduce}}_{\mathrm{exe}})$.

Now suppose that msg is a $\vec{\Pi}^{\mathsf{MR}}$-compliant message with $\mathsf{msg.type} = \Pi^{\mathsf{Reduce}}_{\mathrm{exe}}.\mathsf{type}$. By (the new) construction of $\Pi^{\mathsf{Reduce}}_{\mathrm{exe}}$, we know that $\mathsf{msg.payload} = (\mathsf{cm}, k^3, v^3)$, where cm is a commitment and (k^3, v^3) is a phase-3 record; moreover, we also know that there is a list of messages $\vec{\mathsf{msg}}_{\mathsf{in}}$ such that: (i) for each j, $\vec{\mathsf{msg}}_{\mathsf{in}}[j].\mathsf{type} = \Pi^{\mathsf{Map}}_{\mathrm{exe}}.\mathsf{type}$ and $\vec{\mathsf{msg}}_{\mathsf{in}}[j].\mathsf{payload} = (\mathsf{cm}, k^2, v^2_j)$, where (k^2, v^2_j) is a phase-2 record; (ii) $(k^3, v^3) = \mathsf{Reduce}(k^2, (v^2_j)_j)$. In turn, each message $\vec{\mathsf{msg}}_{\mathsf{in}}[j]$ is $\vec{\Pi}^{\mathsf{MR}}$-compliant and, by (the new) construction of $\Pi^{\mathsf{Map}}_{\mathrm{exe}}$, we know that (k^2, v^2_j) is the result of running Map on some phase-1 record authenticated with respect to cm.

Overall, each $\vec{\Pi}^{\mathsf{MR}}$-compliant message msg with $\mathsf{msg.type} = \Pi^{\mathsf{Reduce}}_{\mathrm{exe}}.\mathsf{type}$ and $\mathsf{msg.payload} = (\mathsf{cm}, k^3, v^3)$ is the result of applying Reduce to some phase-2 records sharing the same key, each of which is in turn the result of applying

$\mathsf{COMM.Gen}(\vec{z})$	$\mathsf{COMM.Ver}(\vec{z}, \mathsf{cm}, \mathsf{cr})$
1. Compute $\mathsf{rt} := \mathsf{MERKLE.GetRoot}(\vec{z})$. 2. Compute n, the number of items in the list $\vec{z}$. 3. Compute $(\mathsf{cm_{rt}}, \mathsf{cr_{rt}}) \leftarrow \mathsf{COMM}^*.\mathsf{Gen}(\mathsf{rt})$. 4. Compute $(\mathsf{cm}_n, \mathsf{cr}_n) \leftarrow \mathsf{COMM}^*.\mathsf{Gen}(n)$. 5. Set $\mathsf{cm} := (\mathsf{cm_{rt}}, \mathsf{cm}_n)$. 6. Set $\mathsf{cr} := (\mathsf{cr_{rt}}, \mathsf{cr}_n)$. 7. Output $(\mathsf{cm}, \mathsf{cr})$.	1. Compute $\mathsf{rt} := \mathsf{MERKLE.GetRoot}(\vec{z})$. 2. Compute n, the number of items in the list $\vec{z}$. 3. Parse cm as a pair $(\mathsf{cm_{rt}}, \mathsf{cm}_n)$. 4. Parse cr as a pair $(\mathsf{cr_{rt}}, \mathsf{cr}_n)$. 5. Check that $\mathsf{COMM}^*.\mathsf{Ver}(\mathsf{rt}, \mathsf{cm_{rt}}, \mathsf{cr_{rt}}) = 1$. 6. Check that $\mathsf{COMM}^*.\mathsf{Ver}(n, \mathsf{cm}_n, \mathsf{cr}_n) = 1$. 7. Output 1 if the above checks succeeded (else, 0).

Figure 5: Choice of commitment scheme COMM (obtained from MERKLE and COMM^*).

Map to some phase-1 record authenticated relative to cm. However, these guarantees are not enough to imply a correct MapReduce computation, as we still need to tackle the following issue.

- **Issue III:** How do we ascertain the correctness of the shuffling phase? Namely, how do we ascertain that each list of phase-2 records (received by a particular reducer node) contains *all* the records having that same key?

Indeed, in principle, some phase-2 records may have been duplicated, dropped, or sent to the wrong reducer node (e.g., to different reducer nodes even if sharing the same key).

5.1.3 Our approach

Unlike previous ones, Issue III is conceptually more complex: tackling it requires ensuring correct shuffling, which is a global computation involving all phase-2 records (all the mappers' outputs); in contrast, we are restricted to only perform local checks encoded in compliance predicates. Nevertheless, we show how we can further extend $\Pi^{\mathsf{Map}}_{\mathrm{exe}}$ and $\Pi^{\mathsf{Reduce}}_{\mathrm{exe}}$, and also introduce additional compliance predicates, to ensure correct shuffling in a distributed manner.

Further extending $\Pi^{\mathsf{Map}}_{\mathrm{exe}}$ and $\Pi^{\mathsf{Reduce}}_{\mathrm{exe}}$. First, we extend $\Pi^{\mathsf{Map}}_{\mathrm{exe}}$ to store, in the output message, the index i relative to which the input message's phase-1 record was authenticated; subsequently, when receiving several input messages, $\Pi^{\mathsf{Reduce}}_{\mathrm{exe}}$ checks that all the indices contained in them are distinct.

The additional check in $\Pi^{\mathsf{Reduce}}_{\mathrm{exe}}$ prevents duplicate messages from being sent to the same reducer node. However, it does not prevent the same message from being sent to multiple reducer nodes, messages with the same key from being sent to multiple reducer nodes, or a message from being dropped. Additional "distributed bookkeeping" is required.

We thus further extend $\Pi^{\mathsf{Reduce}}_{\mathrm{exe}}$ to store in its output message two additional pieces of information: the phase-2 key k^2 shared among its input messages and the number d_{in} of these input messages. More precisely, only commitments to these are stored, and we denote them cm_{k^2} and $\mathsf{cm}_{d_{\mathsf{in}}}$. The commitments preserve zero knowledge, by not exposing internals of the MapReduce computation in output messages of reducer nodes.

See Figure 8 for pseudocode of the new construction of $\Pi^{\mathsf{Map}}_{\mathrm{exe}}$ and $\Pi^{\mathsf{Reduce}}_{\mathrm{exe}}$; the changes from the previous construction (see Figure 7) are highlighted in blue.

We now explain how we leverage, and verify, the messages' new information maintained by $\Pi^{\mathsf{Map}}_{\mathrm{exe}}$ and $\Pi^{\mathsf{Reduce}}_{\mathrm{exe}}$. At high level, we introduce new compliance predicates for checking two main distributed sub-computations: a tree-like distributed sub-computation that aggregates information stored by output messages of mapper nodes, and another tree-like distributed sub-computation that aggregates information stored by output messages of reducer nodes. By comparing the final outputs of these two tree-like sub-computations, we can check if correct shuffling occurred.

Aggregating mappers' outputs. We first describe the tree-like distributed sub-computation that aggregates the output messages of mapper nodes. Recall that each output message of a mapper node has a payload that looks like $(\mathsf{cm}, i, k^2, v^2)$, where cm is a commitment, i is an index, and (k^2, v^2) is a phase-2 record.

For each output message, we introduce a node to reformat the message into a new one with payload $(\mathsf{cm}, a^\perp, a^\top, b, c)$ where $a^\perp = a^\top = i$, $b = 1$, and $c = 1$; we design a compliance predicate, $\Pi^{\mathsf{Map}}_{\mathrm{fmt}}$, to check this reformatting and add it to $\vec{\Pi}^{\mathsf{MR}}$. Intuitively, $a^\perp$ and $a^\top$ denote the smallest and largest index seen so far; b counts the number of mappers; and c counts the number of phase-2 records output by mappers.[9]

Then, we introduce a tree of nodes to aggregate the reformatted messages into a final message, by pairwise transforming two input messages $(\mathsf{cm}, a_1^\perp, a_1^\top, b_1, c_2)$ and $(\mathsf{cm}, a_2^\perp, a_2^\top, b_2, c_2)$ to the message $(\mathsf{cm}, a_1^\perp, a_2^\top, b_1 + b_2, c_1 + c_2)$, provided that $a_1^\top < a_2^\perp$; we design a compliance predicate, $\Pi^{\mathsf{Map}}_{\mathrm{sum}}$, to check this aggregation and add it to $\vec{\Pi}^{\mathsf{MR}}$.

The final message, output by the "root node" has payload $(\mathsf{cm}, 1, M, M, S)$, where M is the total number of mapper nodes and S is the total number of phase-2 records output by all mappers. Crucially, one can see that if any output message of a mapper node is either duplicated or dropped, then it is not possible to obtain a $\vec{\Pi}^{\mathsf{MR}}$-compliant message with type $\Pi^{\mathsf{Map}}_{\mathrm{sum}}.\mathsf{type}$ and payload $(\mathsf{cm}, 1, M, M, S)$.

See Figure 11 and Figure 12 for pseudocode of $\Pi^{\mathsf{Map}}_{\mathrm{fmt}}$ and $\Pi^{\mathsf{Map}}_{\mathrm{sum}}$, respectively; in terms of efficiency, both predicates are "small" in that they have size and cost $O_\lambda(1)$.

Aggregating reducers' outputs. We now turn to the tree-like distributed sub-computation that aggregates output messages of reducer nodes. Recall that each output message of a reducer node has a payload that looks like $(\mathsf{cm}, k^3,$

[9]Recall that for now we are in the artificial special case where each mapper outputs a single phase-2 record, so that c is initially 1.

```
$\Pi^{\mathsf{Map}}_{\mathsf{exe}}(\mathsf{msg}, \mathsf{loc}, \vec{\mathsf{msg}}_{\mathsf{in}})$
1. Parse $\vec{\mathsf{msg}}_{\mathsf{in}}[1].\mathsf{payload}$ as a phase-1 record $(k^1, v^1)$.
2. Parse $\mathsf{msg}.\mathsf{payload}$ as a phase-2 record $(k^2, v^2)$.
3. Check that $\big((k^2, v^2)\big) = \mathsf{Map}(k^1, v^1)$.
```

```
$\Pi^{\mathsf{Reduce}}_{\mathsf{exe}}(\mathsf{msg}, \mathsf{loc}, \vec{\mathsf{msg}}_{\mathsf{in}})$
1. Parse each $\vec{\mathsf{msg}}_{\mathsf{in}}[j].\mathsf{payload}$ as a phase-2 record $(k^2_j, v^2_j)$.
2. Parse $\mathsf{msg}.\mathsf{payload}$ as a phase-3 record $(k^3, v^3)$.
3. Check that all the $k^2_j$'s are equal, and set $\vec{v^2} := (v^2_j)_j$
4. Check that $(k^3, v^3) = \mathsf{Reduce}(k^2_1, \vec{v^2})$.
```

Figure 6: Summary of the construction of $\Pi^{\mathsf{Map}}_{\mathsf{exe}}$ and $\Pi^{\mathsf{Reduce}}_{\mathsf{exe}}$ for “Failed attempt #1” (see Section 5.1.1).

```
$\Pi^{\mathsf{Map}}_{\mathsf{exe}}(\mathsf{msg}, \mathsf{loc}, \vec{\mathsf{msg}}_{\mathsf{in}})$
1. Check that $\vec{\mathsf{msg}}_{\mathsf{in}}[1].\mathsf{type} = 0$.
2. Parse $\vec{\mathsf{msg}}_{\mathsf{in}}[1].\mathsf{payload}$ as a tuple $(\mathsf{cm}, i, k^1, v^1)$ where:
   • $\mathsf{cm}$ is a commitment (for the scheme $\mathsf{COMM}$);
   • $i$ is an index;
   • $(k^1, v^1)$ is a phase-1 record.
3. Parse $\mathsf{msg}.\mathsf{payload}$ as a tuple $(\mathsf{cm}', k^2, v^2)$ where:
   • $\mathsf{cm}'$ is a commitment (for the scheme $\mathsf{COMM}$);
   • $(k^2, v^2)$ is a phase-2 record.
4. Parse $\mathsf{loc}$ as a tuple $(\mathsf{rt}, M, \mathsf{cr}_{\mathsf{rt}}, \mathsf{cr}_M, \mathsf{ap})$ where:
   • $\mathsf{rt}$ is a commitment (for the scheme $\mathsf{MERKLE}$);
   • $M$ is a positive integer;
   • $\mathsf{cr}_{\mathsf{rt}}, \mathsf{cr}_M$ are randomness (for the scheme $\mathsf{COMM}^*$);
   • $\mathsf{ap}$ is an authentication path (for the scheme $\mathsf{MERKLE}$).
5. Parse $\mathsf{cm}$ as a pair $(\mathsf{cm}_{\mathsf{rt}}, \mathsf{cm}_M)$ where both components are
   commitments for the scheme $\mathsf{COMM}^*$.
6. Check that $\mathsf{COMM}^*.\mathsf{Ver}(\mathsf{rt}, \mathsf{cm}_{\mathsf{rt}}, \mathsf{cr}_{\mathsf{rt}}) = 1$.
7. Check that $\mathsf{COMM}^*.\mathsf{Ver}(M, \mathsf{cm}_M, \mathsf{cr}_M) = 1$.
8. Check that $0 \le i < M$.
9. Check that $\mathsf{MERKLE}.\mathsf{CheckPath}\big(\mathsf{rt}, i, (k^1, v^1), \mathsf{ap}\big) = 1$.
10. Check that $\mathsf{cm}' = \mathsf{cm}$.
11. Check that $\big((k^2, v^2)\big) = \mathsf{Map}(k^1, v^1)$.
```

```
$\Pi^{\mathsf{Reduce}}_{\mathsf{exe}}(\mathsf{msg}, \mathsf{loc}, \vec{\mathsf{msg}}_{\mathsf{in}})$
1. Check that $\vec{\mathsf{msg}}_{\mathsf{in}}[j].\mathsf{type} = \Pi^{\mathsf{Map}}_{\mathsf{exe}}.\mathsf{type}$ for each $j$.
2. Parse each $\vec{\mathsf{msg}}_{\mathsf{in}}[j].\mathsf{payload}$ as a tuple $(\mathsf{cm}'_j, k^2_j, v^2_j)$ where:
   • $\mathsf{cm}'_j$ is a commitment (for the scheme $\mathsf{COMM}$);
   • $(k^2_j, v^2_j)$ is a phase-2 record.
3. Parse $\mathsf{msg}.\mathsf{payload}$ as a tuple $(\mathsf{cm}'', k^3, v^3)$ where:
   • $\mathsf{cm}''$ is a commitment (for the scheme $\mathsf{COMM}$);
   • $(k^3, v^3)$ is a phase-3 record.
4. Check that $\mathsf{cm}'' = \mathsf{cm}'_j$ for each $j$.
5. Check that all the $k^2_j$'s are equal, and set $\vec{v^2} := (v^2_j)_j$.
6. Check that $(k^3, v^3) = \mathsf{Reduce}(k^2_1, \vec{v^2})$.
```

Figure 7: Summary of the construction of $\Pi^{\mathsf{Map}}_{\mathsf{exe}}$ and $\Pi^{\mathsf{Reduce}}_{\mathsf{exe}}$ for “Failed attempt #2” (see Section 5.1.2).

```
$\Pi^{\mathsf{Map}}_{\mathsf{exe}}(\mathsf{msg}, \mathsf{loc}, \vec{\mathsf{msg}}_{\mathsf{in}})$
1. Check that $\vec{\mathsf{msg}}_{\mathsf{in}}[1].\mathsf{type} = 0$.
2. Parse $\vec{\mathsf{msg}}_{\mathsf{in}}[1].\mathsf{payload}$ as a tuple $(\mathsf{cm}, i, k^1, v^1)$ where:
   • $\mathsf{cm}$ is a commitment (for the scheme $\mathsf{COMM}$);
   • $i$ is an index;
   • $(k^1, v^1)$ is a phase-1 record.
3. Parse $\mathsf{msg}.\mathsf{payload}$ as a tuple $(\mathsf{cm}', i', k^2, v^2)$ where:
   • $\mathsf{cm}'$ is a commitment (for the scheme $\mathsf{COMM}$);
   • $i'$ is an index;
   • $(k^2, v^2)$ is a phase-2 record.
4. Parse $\mathsf{loc}$ as a tuple $(\mathsf{rt}, M, \mathsf{cr}_{\mathsf{rt}}, \mathsf{cr}_M, \mathsf{ap})$ where:
   • $\mathsf{rt}$ is a commitment (for the scheme $\mathsf{MERKLE}$);
   • $M$ is a positive integer;
   • $\mathsf{cr}_{\mathsf{rt}}, \mathsf{cr}_M$ are randomness (for the scheme $\mathsf{COMM}^*$);
   • $\mathsf{ap}$ is an authentication path (for the scheme $\mathsf{MERKLE}$).
5. Parse $\mathsf{cm}$ as a pair $(\mathsf{cm}_{\mathsf{rt}}, \mathsf{cm}_M)$ where both components are
   commitments for the scheme $\mathsf{COMM}^*$.
6. Check that $\mathsf{COMM}^*.\mathsf{Ver}(\mathsf{rt}, \mathsf{cm}_{\mathsf{rt}}, \mathsf{cr}_{\mathsf{rt}}) = 1$.
7. Check that $\mathsf{COMM}^*.\mathsf{Ver}(M, \mathsf{cm}_M, \mathsf{cr}_M) = 1$.
8. Check that $0 \le i < M$.
9. Check that $\mathsf{MERKLE}.\mathsf{CheckPath}\big(\mathsf{rt}, i, (k^1, v^1), \mathsf{ap}\big) = 1$.
10. Check that $\mathsf{cm}' = \mathsf{cm}$ and $i' = i$.
11. Check that $\big((k^2, v^2)\big) = \mathsf{Map}(k^1, v^1)$.
```

```
$\Pi^{\mathsf{Reduce}}_{\mathsf{exe}}(\mathsf{msg}, \mathsf{loc}, \vec{\mathsf{msg}}_{\mathsf{in}})$
1. Check that $\vec{\mathsf{msg}}_{\mathsf{in}}[j].\mathsf{type} = \Pi^{\mathsf{Map}}_{\mathsf{exe}}.\mathsf{type}$ for each $j$.
2. Parse each $\vec{\mathsf{msg}}_{\mathsf{in}}[j].\mathsf{payload}$ as a tuple $(\mathsf{cm}'_j, i'_j, k^2_j, v^2_j)$ where:
   • $\mathsf{cm}'_j$ is a commitment (for the scheme $\mathsf{COMM}$);
   • $i'_j$ is an index;
   • $(k^2_j, v^2_j)$ is a phase-2 record.
3. Parse $\mathsf{msg}.\mathsf{payload}$ as a tuple $(\mathsf{cm}'', k^3, v^3, \mathsf{cm}_{k^2}, \mathsf{cm}_{d_{\mathsf{in}}})$ where:
   • $\mathsf{cm}''$ is a commitment (for the scheme $\mathsf{COMM}$);
   • $(k^3, v^3)$ is a phase-3 record;
   • $\mathsf{cm}_{k^2}, \mathsf{cm}_{d_{\mathsf{in}}}$ are commitments (for the scheme $\mathsf{COMM}^*$).
4. Parse $\mathsf{loc}$ as a tuple $(\mathsf{cr}_{k^2}, \mathsf{cr}_{d_{\mathsf{in}}})$ where:
   • $\mathsf{cr}_{k^2}, \mathsf{cr}_{d_{\mathsf{in}}}$ are randomness (for the scheme $\mathsf{COMM}^*$).
5. Check that $\mathsf{cm}'' = \mathsf{cm}'_j$ for each $j$.
6. Check that the $i'_j$ are distinct, and let $d_{\mathsf{in}}$ be their number.
7. Check that all the $k^2_j$'s are equal, and set $\vec{v^2} := (v^2_j)_j$.
8. Check that $\mathsf{COMM}^*.\mathsf{Ver}(k^2_1, \mathsf{cm}_{k^2}, \mathsf{cr}_{k^2}) = 1$.
9. Check that $\mathsf{COMM}^*.\mathsf{Ver}(d_{\mathsf{in}}, \mathsf{cm}_{d_{\mathsf{in}}}, \mathsf{cr}_{d_{\mathsf{in}}}) = 1$.
10. Check that $(k^3, v^3) = \mathsf{Reduce}(k^2_1, \vec{v^2})$.
```

Figure 8: Summary of the construction of $\Pi^{\mathsf{Map}}_{\mathsf{exe}}$ and $\Pi^{\mathsf{Reduce}}_{\mathsf{exe}}$ for “Our approach” (see Section 5.1.3). The text that is highlighted in blue denotes the differences from the construction in Figure 7.

$v^3, \mathsf{cm}_{k^2}, \mathsf{cm}_{d_{\mathsf{in}}})$, where cm is a commitment (for the scheme COMM), (k^3, v^3) is a phase-3 record, and $\mathsf{cm}_{k^2}, \mathsf{cm}_{d_{\mathsf{in}}}$ are commitments (for the scheme COMM^*).

Similarly (but not equal) to above, for each output message, we introduce a node to reformat the message into a new one with payload $(\mathsf{cm}, a^{\perp}, a^{\top}, b, c)$ where $a^{\perp} = a^{\top} = k^2$, $b = 1$, and $c = d_{\mathsf{in}}$;[10] we design a compliance predicate, $\Pi^{\mathsf{Reduce}}_{\mathrm{fmt}}$, to check this reformatting and add it to $\vec{\Pi}^{\mathsf{MR}}$. Intuitively, $a^{\perp}$ and $a^{\top}$ denote the smallest and largest phase-2 keys so far; b counts the number of reducers; and c counts the number of phase-2 records received by reducers.

Then, again similarly to above, we introduce a tree of nodes to aggregate the reformatted messages into a final message, by pairwise transforming two input messages $(\mathsf{cm}, a_1^{\perp}, a_1^{\top}, b_1, c_2)$ and $(\mathsf{cm}, a_2^{\perp}, a_2^{\top}, b_2, c_2)$ to the message $(\mathsf{cm}, a_1^{\perp}, a_2^{\top}, b_1 + b_2, c_1 + c_2)$, provided that $a_1^{\top} < a_2^{\perp}$; we design a compliance predicate, $\Pi^{\mathsf{Reduce}}_{\mathrm{sum}}$, to check this aggregation and add it to $\vec{\Pi}^{\mathsf{MR}}$.

The final message, output by the "root node", has payload $(\mathsf{cm}, k^2_{\min}, k^2_{\max}, R, S)$, where $k^2_{\min}$ and $k^2_{\max}$ are respectively the least and largest phase-2 keys encountered, R is the total number of reducer nodes, and S is the total number of phase-2 records received by all reducers. Crucially, one can see that if any output message of a reducer node is either duplicated or dropped, then it is not possible to obtain a $\vec{\Pi}^{\mathsf{MR}}$-compliant message with type $\Pi^{\mathsf{Reduce}}_{\mathrm{sum}}.\mathsf{type}$ and payload $(\mathsf{cm}, k^2_{\min}, k^2_{\max}, R, S)$.

See Figure 11 and Figure 12 for pseudocode of $\Pi^{\mathsf{Reduce}}_{\mathrm{fmt}}$ and $\Pi^{\mathsf{Reduce}}_{\mathrm{sum}}$, respectively; in terms of efficiency, both predicates are "small" in that they have size and cost $O_{\lambda}(1)$.

Consistency between aggregations. In sum, the above two tree-like distributed sub-computations result into two messages, respectively with payloads $(\mathsf{cm}, 1, M, M, S)$ and $(\mathsf{cm}, k^2_{\min}, k^2_{\max}, R, S)$.

We introduce a final node that compares the two messages and outputs a final message with payload (cm, R).[11] Specifically, we design, and add to $\vec{\Pi}^{\mathsf{MR}}$, a new compliance predicate, Π_{fin}, which checks that: (i) cm is the same in both messages; (ii) S is the same in both messages; and (iii) M is the size of the data committed in cm (this can be done by receiving decommitment information as part of the node's local data loc). These checks ensure, in particular, that the outputs of all mappers were correctly shuffled and then given as input to the reducers. See Figure 13.

Summing up. The vector $\vec{\Pi}^{\mathsf{MR}}$ of compliance predicates is $(\Pi^{\mathsf{Map}}_{\mathrm{exe}}, \Pi^{\mathsf{Reduce}}_{\mathrm{exe}}, \Pi^{\mathsf{Map}}_{\mathrm{fmt}}, \Pi^{\mathsf{Map}}_{\mathrm{sum}}, \Pi^{\mathsf{Reduce}}_{\mathrm{fmt}}, \Pi^{\mathsf{Reduce}}_{\mathrm{sum}}, \Pi_{\mathrm{fin}})$. We fix the types of these predicates to arbitrary but distinct values. Each predicate constrains the types of its input messages as follows: (1) $\Pi^{\mathsf{Map}}_{\mathrm{exe}}$ accepts only input messages with type 0; (2) $\Pi^{\mathsf{Reduce}}_{\mathrm{exe}}$ with type $\Pi^{\mathsf{Map}}_{\mathrm{exe}}.\mathsf{type}$; (3) $\Pi^{\mathsf{Map}}_{\mathrm{fmt}}$ with type $\Pi^{\mathsf{Map}}_{\mathrm{exe}}.\mathsf{type}$; (4) $\Pi^{\mathsf{Map}}_{\mathrm{sum}}$ with type $\Pi^{\mathsf{Map}}_{\mathrm{exe}}.\mathsf{type}$ or $\Pi^{\mathsf{Map}}_{\mathrm{fmt}}.\mathsf{type}$; (5) $\Pi^{\mathsf{Reduce}}_{\mathrm{fmt}}$ with type $\Pi^{\mathsf{Reduce}}_{\mathrm{exe}}.\mathsf{type}$; (6) $\Pi^{\mathsf{Reduce}}_{\mathrm{sum}}$ with type $\Pi^{\mathsf{Reduce}}_{\mathrm{exe}}.\mathsf{type}$ or $\Pi^{\mathsf{Reduce}}_{\mathrm{fmt}}.\mathsf{type}$; (7) Π_{fin} with type $\Pi^{\mathsf{Map}}_{\mathrm{exe}}.\mathsf{type}$ or $\Pi^{\mathsf{Map}}_{\mathrm{fmt}}.\mathsf{type}$ for the first input message, and $\Pi^{\mathsf{Reduce}}_{\mathrm{exe}}.\mathsf{type}$ or $\Pi^{\mathsf{Reduce}}_{\mathrm{fmt}}.\mathsf{type}$ for the second input message. These constraints on message types induce a "data flow" in the distributed computation that we summarized in Figure 9.

If the underlying MapReduce computation results into the output $\mathbb{y} = \big((k^3_1, v^3_1), \ldots, (k^3_R, v^3_R)\big)$, the distributed computation results into $R + 1$ outputs: (i) a message with type $\Pi_{\mathrm{fin}}.\mathsf{type}$ and payload (cm, R); and (ii) for $i = 1, \ldots, R$, a message with type $\Pi^{\mathsf{Reduce}}_{\mathrm{exe}}.\mathsf{type}$ and payload $(\mathsf{cm}, k^3_i, v^3_i, *, *)$ where $*$ denotes arbitrary data. These outputs make the distributed computation $(\mathsf{cm}, \mathbb{y})$-compatible (see Definition 5.1). Moreover, one can see that the outputs of the distributed computation can be computed by a transcript generator TGen that is $(\mathsf{Map}, \mathsf{Reduce})$-complexity-preserving; see Figure 10 for a diagram of the various parts of the distributed computation conducted by TGen. This provides the completeness property claimed in Theorem 5.2.

Conversely, if a message with payload (cm, R) (and suitable type) is $\vec{\Pi}^{\mathsf{MR}}$-compliant, then we can deduce the correct execution of a MapReduce computation relative to the data $\mathbb{x}$ committed in cm and resulting in some output $\mathbb{y}$ with R output records. Moreover, if a message with payload $(\mathsf{cm}, k^3_i, v^3_i, *, *)$ (and suitable type) is $\vec{\Pi}^{\mathsf{MR}}$-compliant, then we can deduce that (k^3_i, v^3_i) is the i-th record of $\mathbb{y}$. In fact, one can see that any transcript T that yields such outputs contains a witness for the instance $(\mathsf{cm}, \mathbb{y})$, which can be obtained in linear time by an extractor TExt. This provides the proof-of-knowledge property claimed in Theorem 5.2.

5.1.4 From sketch to the full construction

The above discussion omits several details. First, we have considered only the (artificial) case where each mapper outputs a single phase-2 record; the case of multiple outputs necessitates further extending the predicates in $\vec{\Pi}^{\mathsf{MR}}$. For

[10]The values k^2 and d_{in} can be obtained by receiving decommitment information as part of the node's local data loc.

[11]To preserve zero knowledge, both M and S (which give information about internals of the computation), are excluded from the final message.

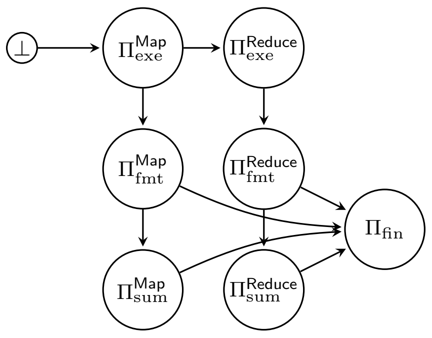
$\perp$
$\Pi^{\mathsf{Map}}_{\mathrm{exe}}$
$\Pi^{\mathsf{Reduce}}_{\mathrm{exe}}$
$\Pi^{\mathsf{Map}}_{\mathrm{fmt}}$
$\Pi^{\mathsf{Reduce}}_{\mathrm{fmt}}$
Π_{fin}
$\Pi^{\mathsf{Map}}_{\mathrm{sum}}$
$\Pi^{\mathsf{Reduce}}_{\mathrm{sum}}$

Figure 9: The compliance predicates constrain the types of input messages. This induces a "data flow" in the distributed computation, which can be summarized via the above diagram.

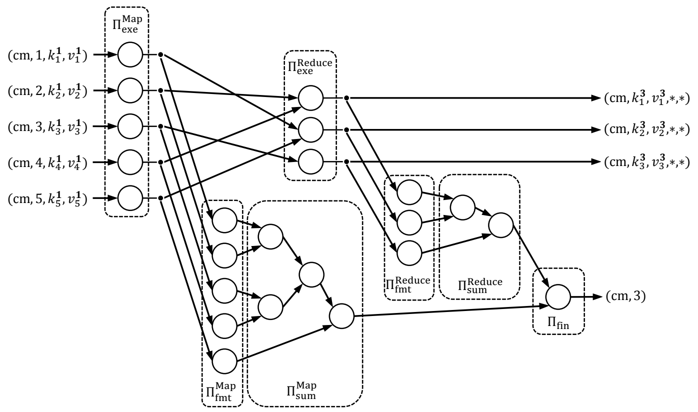
$\Pi^{\mathsf{Map}}_{\mathrm{exe}}$
$(\mathrm{cm}, 1, k_1^1, v_1^1)$
$(\mathrm{cm}, 2, k_2^1, v_2^1)$
$(\mathrm{cm}, 3, k_3^1, v_3^1)$
$(\mathrm{cm}, 4, k_4^1, v_4^1)$
$(\mathrm{cm}, 5, k_5^1, v_5^1)$
$\Pi^{\mathsf{Reduce}}_{\mathrm{exe}}$
$(\mathrm{cm}, k_1^3, v_1^3, *, *)$
$(\mathrm{cm}, k_2^3, v_2^3, *, *)$
$(\mathrm{cm}, k_3^3, v_3^3, *, *)$
$\Pi^{\mathsf{Reduce}}_{\mathrm{fmt}}$
$\Pi^{\mathsf{Reduce}}_{\mathrm{sum}}$
$(\mathrm{cm}, 3)$
Π_{fin}
$\Pi^{\mathsf{Map}}_{\mathrm{fmt}}$
$\Pi^{\mathsf{Map}}_{\mathrm{sum}}$

Figure 10: Diagram of the distributed computation for a MapReduce computation with 5 input records and 3 output records. The dashed regions show which output messages are checked by which compliance predicate. The two tree-like sub-computations in the bottom part correspond to the aggregation of mappers' outputs (left) and the aggregation of reducers' outputs (right).

completeness, in Appendix D we provide details on how to support the case of multiple mapper outputs, as well as on how to support the extensions mentioned in Section 2.4.3 (including multiple reducer outputs).

Moreover, we did not describe the many technicalities that arise when implementing each compliance predicate as an arithmetic circuit, as required by the underlying PCD machinery (see Section 6.4). We invested much effort in obtaining efficient implementations of the 7 compliance predicates as arithmetic circuits, and this involved additional modifications to the predicates in $\vec{\Pi}^{\mathsf{MR}}$.

```
$\Pi^{\mathsf{Map}}_{\mathrm{fmt}}(\mathsf{msg}, \mathsf{loc}, \vec{\mathsf{msg}}_{\mathsf{in}})$
1. Let $d$ be the number of input messages in $\vec{\mathsf{msg}}_{\mathsf{in}}$.
2. Check that $d = 1$.
3. Check that $\vec{\mathsf{msg}}_{\mathsf{in}}[1].\mathsf{type} = \Pi^{\mathsf{Map}}_{\mathrm{exe}}.\mathsf{type}$.
4. Parse $\vec{\mathsf{msg}}_{\mathsf{in}}[1].\mathsf{payload}$ as a tuple $(\mathsf{cm}, i, k^2, v^2)$ where:
   • $\mathsf{cm}$ is a commitment (for the scheme $\mathsf{COMM}$);
   • $i$ is an index;
   • $(k^2, v^2)$ is a phase-2 record.
5. The local data $\mathsf{loc}$ is not used; ignore it.
6. Parse $\mathsf{msg}.\mathsf{payload}$ as a tuple $(\mathsf{cm}', a^{\perp}, a^{\top}, b, c)$ where:
   • $\mathsf{cm}'$ is a commitment (for the scheme $\mathsf{COMM}$);
   • $a^{\perp}, a^{\top}, b, c$ are positive integers.
7. Check that $\mathsf{cm}' = \mathsf{cm}$.
8. Check that $a^{\perp} = a^{\top} = i$.
9. Check that $b = 1$.
10. Check that $c = 1$.
```

```
$\Pi^{\mathsf{Reduce}}_{\mathrm{fmt}}(\mathsf{msg}, \mathsf{loc}, \vec{\mathsf{msg}}_{\mathsf{in}})$
1. Let $d$ be the number of input messages in $\vec{\mathsf{msg}}_{\mathsf{in}}$.
2. Check that $d = 1$.
3. Check that $\vec{\mathsf{msg}}_{\mathsf{in}}[1].\mathsf{type} = \Pi^{\mathsf{Reduce}}_{\mathrm{exe}}.\mathsf{type}$.
4. Parse $\vec{\mathsf{msg}}_{\mathsf{in}}[1].\mathsf{payload}$ as a tuple $(\mathsf{cm}, k^3, v^3, \mathsf{cm}_{k^2}, \mathsf{cm}_{d_{\mathsf{in}}})$ where:
   • $\mathsf{cm}$ is a commitment (for the scheme $\mathsf{COMM}$);
   • $(k^3, v^3)$ is a phase-3 record;
   • $\mathsf{cm}_{k^2}, \mathsf{cm}_{d_{\mathsf{in}}}$ are commitments (for the scheme $\mathsf{COMM}^*$).
5. Parse $\mathsf{loc}$ as a tuple $(k^2, d_{\mathsf{in}}, \mathsf{cr}_{k^2}, \mathsf{cr}_{d_{\mathsf{in}}})$, where:
   • $k^2$ is a phase-2 key;
   • $d_{\mathsf{in}}$ is a positive integer;
   • $\mathsf{cr}_{k^2}, \mathsf{cr}_{d_{\mathsf{in}}}$ are randomness (for the scheme $\mathsf{COMM}^*$).
6. Parse $\mathsf{msg}.\mathsf{payload}$ as a tuple $(\mathsf{cm}', a^{\perp}, a^{\top}, b, c)$ where:
   • $\mathsf{cm}'$ is a commitment (for the scheme $\mathsf{COMM}$);
   • $a^{\perp}, a^{\top}, b, c$ are positive integers.
7. Check that $\mathsf{cm}' = \mathsf{cm}$.
8. Check that $\mathsf{COMM}^*.\mathsf{Ver}(k^2, \mathsf{cm}_{k^2}, \mathsf{cr}_{k^2}) = 1$.
9. Check that $\mathsf{COMM}^*.\mathsf{Ver}(d_{\mathsf{in}}, \mathsf{cm}_{d_{\mathsf{in}}}, \mathsf{cr}_{d_{\mathsf{in}}}) = 1$.
10. Check that $a^{\perp} = a^{\top} = k^2$ (after converting $k^2$ to a positive integer).
11. Check that $b = 1$.
12. Check that $c = d_{\mathsf{in}}$.
```

Figure 11: Summary of the construction of $\Pi^{\mathsf{Map}}_{\mathrm{fmt}}$ and $\Pi^{\mathsf{Reduce}}_{\mathrm{fmt}}$ (see Section 5.1.3).

```
$\Pi^{\mathsf{Map}}_{\mathrm{sum}}(\mathsf{msg}, \mathsf{loc}, \vec{\mathsf{msg}}_{\mathsf{in}})$
1. Let $d$ be the number of input messages in $\vec{\mathsf{msg}}_{\mathsf{in}}$.
2. Check that $d = 2$.
3. Check that each $\vec{\mathsf{msg}}_{\mathsf{in}}[j].\mathsf{type}$ lies in $\{\Pi^{\mathsf{Map}}_{\mathrm{fmt}}.\mathsf{type}, \Pi^{\mathsf{Map}}_{\mathrm{sum}}.\mathsf{type}\}$.
4. Parse each $\vec{\mathsf{msg}}_{\mathsf{in}}[j].\mathsf{payload}$ as a tuple $(\mathsf{cm}_j, a_j^{\perp}, a_j^{\top}, b_j, c_j)$ where:
   • $\mathsf{cm}_j$ is a commitment (for the scheme $\mathsf{COMM}$);
   • $a_j^{\perp}, a_j^{\top}, b_j, c_j$ are positive integers.
5. The local data $\mathsf{loc}$ is not used; ignore it.
6. Parse $\mathsf{msg}.\mathsf{payload}$ as a tuple $(\mathsf{cm}', a^{\perp}, a^{\top}, b, c)$ where:
   • $\mathsf{cm}'$ is a commitment (for the scheme $\mathsf{COMM}$);
   • $a^{\perp}, a^{\top}, b, c$ are positive integers.
7. Check that $\mathsf{cm}' = \mathsf{cm}_1 = \mathsf{cm}_2$.
8. Check that $a_1^{\top} < a_2^{\perp}, a^{\perp} = a_1^{\perp}, a^{\top} = a_2^{\top}$.
9. Check that $b = b_1 + b_2$.
10. Check that $c = c_1 + c_2$.
```

```
$\Pi^{\mathsf{Reduce}}_{\mathrm{sum}}(\mathsf{msg}, \mathsf{loc}, \vec{\mathsf{msg}}_{\mathsf{in}})$
1. Let $d$ be the number of input messages in $\vec{\mathsf{msg}}_{\mathsf{in}}$.
2. Check that $d = 2$.
3. Check that each $\vec{\mathsf{msg}}_{\mathsf{in}}[j].\mathsf{type}$ lies in $\{\Pi^{\mathsf{Reduce}}_{\mathrm{fmt}}.\mathsf{type}, \Pi^{\mathsf{Reduce}}_{\mathrm{sum}}.\mathsf{type}\}$.
4. Parse each $\vec{\mathsf{msg}}_{\mathsf{in}}[j].\mathsf{payload}$ as a tuple $(\mathsf{cm}_j, a_j^{\perp}, a_j^{\top}, b_j, c_j)$ where:
   • $\mathsf{cm}_j$ is a commitment (for the scheme $\mathsf{COMM}$);
   • $a_j^{\perp}, a_j^{\top}, b_j, c_j$ are positive integers.
5. The local data $\mathsf{loc}$ is not used; ignore it.
6. Parse $\mathsf{msg}.\mathsf{payload}$ as a tuple $(\mathsf{cm}', a^{\perp}, a^{\top}, b, c)$ where:
   • $\mathsf{cm}'$ is a commitment (for the scheme $\mathsf{COMM}$);
   • $a^{\perp}, a^{\top}, b, c$ are positive integers.
7. Check that $\mathsf{cm}' = \mathsf{cm}_1 = \mathsf{cm}_2$.
8. Check that $a_1^{\top} < a_2^{\perp}, a^{\perp} = a_1^{\perp}, a^{\top} = a_2^{\top}$.
9. Check that $b = b_1 + b_2$.
10. Check that $c = c_1 + c_2$.
```

Figure 12: Summary of the construction of $\Pi^{\mathsf{Map}}_{\mathrm{sum}}$ and $\Pi^{\mathsf{Map}}_{\mathrm{sum}}$ (see Section 5.1.3).

```
$\Pi_{\mathrm{fin}}(\mathsf{msg}, \mathsf{loc}, \vec{\mathsf{msg}}_{\mathsf{in}})$
1. Let $d$ be the number of input messages in $\vec{\mathsf{msg}}_{\mathsf{in}}$.
2. Check that $d = 2$.
3. Check that $\vec{\mathsf{msg}}_{\mathsf{in}}[1].\mathsf{type} \in \{\Pi^{\mathsf{Map}}_{\mathrm{fmt}}.\mathsf{type}, \Pi^{\mathsf{Map}}_{\mathrm{sum}}.\mathsf{type}\}$.
4. Check that $\vec{\mathsf{msg}}_{\mathsf{in}}[2].\mathsf{type} \in \{\Pi^{\mathsf{Reduce}}_{\mathrm{fmt}}.\mathsf{type}, \Pi^{\mathsf{Reduce}}_{\mathrm{sum}}.\mathsf{type}\}$.
5. Parse each $\vec{\mathsf{msg}}_{\mathsf{in}}[j].\mathsf{payload}$ as a tuple $(\mathsf{cm}_j, a_j^{\perp}, a_j^{\top}, b_j, c_j)$ where:
   • $\mathsf{cm}_j$ is a commitment (for the scheme $\mathsf{COMM}$);
   • $a_j^{\perp}, a_j^{\top}, b_j, c_j$ are positive integers.
6. Parse the local data $\mathsf{loc}$ as a tuple $(M, \mathsf{cr}_M)$ where:
   • $M$ is a positive integer;
   • $\mathsf{cr}_M$ is randomness (for the scheme $\mathsf{COMM}^*$).
7. Parse $\mathsf{msg}.\mathsf{payload}$ as a tuple $(\mathsf{cm}', R)$ where:
   • $\mathsf{cm}'$ is a commitment (for the scheme $\mathsf{COMM}$);
   • $R$ is a positive integer.
8. Check that $\mathsf{cm}' = \mathsf{cm}_1 = \mathsf{cm}_2$.
9. Parse $\mathsf{cm}'$ as a pair $(\mathsf{cm}_{\mathsf{rt}}, \mathsf{cm}_M)$ where both components are commitments
   for the scheme $\mathsf{COMM}^*$.
10. Check that $\mathsf{COMM}^*.\mathsf{Ver}(M, \mathsf{cm}_M, \mathsf{cr}_M) = 1$.
11. Check that $M = b_1$.
12. Check that $R = b_2$.
13. Check that $c_1 = c_2$.
```

Figure 13: Summary of the construction of Π_{fin} (see Section 5.1.3).

5.2 Construction of distributed zk-SNARKs for MapReduce

We give the construction of our distributed zk-SNARK for MapReduce, by describing its key generator $\mathsf{MR.KeyGen}$, prover $\mathsf{MR.Prove}$, and verifier $\mathsf{MR.Verify}$. As for the commitment scheme COMM, it is described in Figure 5.

The key generator $\mathsf{MR.KeyGen}(1^\lambda, \mathsf{Map}, \mathsf{Reduce}) \to (\mathsf{pk}, \mathsf{vk})$. On input a security parameter λ (presented in unary) and a MapReduce pair $(\mathsf{Map}, \mathsf{Reduce})$, the key generator $\mathsf{MR.KeyGen}$ computes a key pair $(\mathsf{pk}, \mathsf{vk})$ as follows.
1. Use Theorem 5.2 to deduce, from $(\mathsf{Map}, \mathsf{Reduce})$, the vector $\vec{\Pi}^{\mathsf{MR}}$ of compliance predicates.
2. Use the PCD key generator $\mathbb{G}$ to compute a PCD key pair for $\vec{\Pi}^{\mathsf{MR}}$: $(\mathsf{pk}_{\mathsf{pcd}}, \mathsf{vk}_{\mathsf{pcd}}) := \mathbb{G}(1^\lambda, \vec{\Pi}^{\mathsf{MR}})$.
3. Set $\mathsf{pk} := (\mathsf{Map}, \mathsf{Reduce}, \mathsf{pk}_{\mathsf{pcd}})$ and $\mathsf{vk} := (\mathsf{vk}_{\mathsf{pcd}})$; output $(\mathsf{pk}, \mathsf{vk})$.

The prover $\mathsf{MR.Prove}(\mathsf{pk}, \mathsf{cm}, \mathbb{y}, \mathbb{x}, \mathsf{cr}) \to \pi_{\mathsf{MR}}$. On input a proving key pk, an instance $(\mathsf{cm}, \mathbb{y})$, and a witness $(\mathbb{x}, \mathsf{cr})$, the prover $\mathsf{MR.Prove}$ computes a non-interactive proof π_{MR} for the statement "I know $(\mathbb{x}, \mathsf{cr})$ such that $\big((\mathsf{cm}, \mathbb{y}), (\mathbb{x}, \mathsf{cr})\big) \in \mathscr{R}^{\mathsf{COMM}}_{(\mathsf{Map},\mathsf{Reduce})}$" as follows.
1. Deduce from the MapReduce pair $(\mathsf{Map}, \mathsf{Reduce})$ the corresponding transcript generator TGen (see Theorem 5.2).
2. Deduce from $\mathsf{TGen}(\mathsf{cm}, \mathbb{y}, \mathbb{x}, \mathsf{cr})$ the corresponding PCD meta prover TPrv (see Section 4).
3. Compute $\vec{\pi} := \mathsf{TPrv}(\mathsf{pk}_{\mathsf{pcd}})$.
4. Set $\pi_{\mathsf{MR}} := \vec{\pi}$ and output π_{MR}.

The verifier $\mathsf{MR.Verify}(\mathsf{vk}, \mathsf{cm}, \mathbb{y}, \pi_{\mathsf{MR}}) \to b$. On input a verification key vk, commitment cm, output $\mathbb{y}$, and proof π_{MR}, the verifier $\mathsf{MR.Verify}$ computes a decision bit b as follows.
1. Parse vk as a PCD verification key $\mathsf{vk}_{\mathsf{pcd}}$.
2. Use the instance $(\mathsf{cm}, \mathbb{y})$ to construct the following output messages (recall Definition 5.1):

$$\mathsf{msg}_0 \begin{cases} .\mathsf{type} := 1 \\ .\mathsf{payload} := (\mathsf{cm}, \mathsf{len}(\mathbb{y})) \end{cases} \quad \text{and, for each } i \in \{1, \ldots, \mathsf{len}(\mathbb{y})\},\ \mathsf{msg}_i \begin{cases} .\mathsf{type} := 2 \\ .\mathsf{payload} := (\mathsf{cm}, \mathbb{y}_i) \end{cases} .$$

3. Parse π_{MR} a vector of PCD proofs $(\pi_0, \pi_1, \ldots, \pi_{\mathsf{len}(\mathbb{y})})$.
4. For each $i \in \{0, 1, \ldots, \mathsf{len}(\mathbb{y})\}$, check that the i-th output message is $\vec{\Pi}^{\mathsf{MR}}$-compliant: $\mathbb{V}(\mathsf{vk}_{\mathsf{pcd}}, \mathsf{msg}_i, \pi_i) = 1$.
5. If all the above steps succeeded, output $b := 1$; otherwise output $b := 0$.

Indeed, if $\mathsf{MR.Verify}$ outputs 1, then we know that the prover that produced π_{MR} knows a $\vec{\Pi}^{\mathsf{MR}}$-compliant $(\mathsf{cm}, \mathbb{y})$-compatible transcript T (by the proof-of-knowledge property of the PCD system), and thus also knows a witness $(\mathbb{x}, \mathsf{cr})$ for the instance $(\mathsf{cm}, \mathbb{y})$ (by Theorem 5.2).

In Appendix E, for additional intuition, we break the abstraction, and describe $\mathsf{MR.Prove}$ directly in terms of the compliance predicates constructed in the proof of Theorem 5.2.

Remark 5.3. In the construction above, the output $\mathbb{y}$ can also be checked one record at a time, rather than all at once as done by $\mathsf{MR.Verify}$. Namely, to check $\mathbb{y}_i$ (the i-th record of $\mathbb{y}$), one only needs to ensure that $\mathbb{V}(\mathsf{vk}_{\mathsf{pcd}}, \mathsf{msg}_0, \pi_0) = 1$ and $\mathbb{V}(\mathsf{vk}_{\mathsf{pcd}}, \mathsf{msg}_i, \pi_i) = 1$. This enables, e.g., to distribute $\mathbb{y}$'s verification.

Remark 5.4. The running time of $\mathsf{MR.Verify}$ is $O_\lambda(|\mathbb{y}|)$, and is dominated by $\mathsf{len}(\mathbb{y}) + 1$ invocations of the underlying PCD verifier $\mathbb{V}$. Since $\mathsf{MR.Verify}$ takes as input $\mathbb{y}$, its asymptotic running time cannot be significantly improved. However, the construction above can be modified so that verification requires only one invocation of $\mathbb{V}$ (on a $O_\lambda(1)$-size input) plus $O(|\mathbb{y}|)$ "light" cryptographic operations — this improves concrete efficiency of verification.

The high-level idea is to extend the proving process with an additional tree-like distributed computation that aggregates the message-proof pairs $(\mathsf{msg}_i, \pi_i)_{i=0}^{\mathsf{len}(\mathbb{y})}$ into a single message-proof pair (msg^*, π^*) such that msg^* has payload $(\mathsf{cm}, \mathsf{len}(\mathbb{y}), \mathsf{cm}_{\mathbb{y}})$ where $\mathsf{cm}_{\mathbb{y}}$ is a Merkle-tree commitment of $\mathsf{cm}_{\mathbb{y}}$. Verification then involves checking that $\mathbb{V}(\mathsf{vk}_{\mathsf{pcd}}, \mathsf{msg}^*, \pi^*) = 1$ and that $\mathbb{y}$ is a valid opening of $\mathsf{cm}_{\mathbb{y}}$. (Moreover, by additionally augmenting π^* with authentication paths, one can recover the "local verification" property discussed in Remark 5.3.)

6 Step I: construction of multi-predicate PCD

We discuss Step I of our bootstrapping theorem: constructing multi-predicate PCD from (preprocessing) zk-SNARKs.

- In Section 6.1, we introduce notation for arithmetic circuits and preprocessing zk-SNARKs.
- In Section 6.2, we review the single-predicate PCD construction of [BCTV14a].
- In Section 6.3, we sketch the ideas of our multi-predicate PCD construction, which extends the one in [BCTV14a].
- In Section 6.4, we provide details of our multi-predicate PCD construction.
- In Section 6.5, we describe some optimizations and extensions.

As in [BCTV14a], we consider compliance predicates Π expressed as $\mathbb{F}$-arithmetic circuits, where $\mathbb{F}$ is a certain field of cryptographically-large prime size (determined by the underlying zk-SNARK). Throughout this section, $\mathbb{F}_n$ denotes the field of size n, and we assume familiarity with finite fields (and, for background on these, see [LN97]).

6.1 Arithmetic circuits and preprocessing zk-SNARKs

Arithmetic circuits. As mentioned, we work with circuits that are arithmetic, rather than boolean. Given a finite field $\mathbb{F}$, an *$\mathbb{F}$-arithmetic circuit* takes inputs that are elements in $\mathbb{F}$, and its gates output elements in $\mathbb{F}$; the circuits we consider only have *bilinear gates*. The *circuit satisfaction problem* of an $\mathbb{F}$-arithmetic circuit $C \colon \mathbb{F}^n \times \mathbb{F}^h \to \mathbb{F}^l$ is defined by the relation $\mathscr{R}_C = \{(x,a) \in \mathbb{F}^n \times \mathbb{F}^h : C(x,a) = 0^l\}$.

Preprocessing zk-SNARKs. As in [BCTV14b], a *preprocessing zk-SNARK* [BCIOP13, BCCT13] for $\mathbb{F}$-arithmetic circuit satisfiability is a triple of polynomial-time algorithms (G, P, V), called *key generator*, *prover*, and *verifier*. The key generator G, given a security parameter λ and an $\mathbb{F}$-arithmetic circuit $C \colon \mathbb{F}^n \times \mathbb{F}^h \to \mathbb{F}^l$, samples a *proving key* pk and a *verification key* vk; these are the proof system's public parameters, and are generated only once per circuit. After that, anyone can use pk to generate non-interactive proofs of knowledge for witnesses in the relation $\mathscr{R}_C$, and anyone can use the vk to check these proofs. Namely, given pk and any $(x,a) \in \mathscr{R}_C$, the honest prover $P(\mathsf{pk}, x, a)$ produces a proof π for the statement "there is a such that $(x,a) \in \mathscr{R}_C$"; the verifier $V(\mathsf{vk}, x, \pi)$ checks that π is a convincing proof for this statement. A proof π is a (computational) proof of knowledge, and a (statistical) zero-knowledge proof. The succinctness property requires that π has length $O_\lambda(1)$ and V runs in time $O_\lambda(|x|)$. See Appendix C for details.

6.2 Review of the [BCTV14a] construction

For efficiency reasons, Ben-Sasson et al. [BCTV14a] construct a PCD system via *two* (preprocessing) zk-SNARKs, $(G_\alpha, P_\alpha, V_\alpha)$ and $(G_\beta, P_\beta, V_\beta)$, that satisfy the following. For two primes r_α and r_β: (a) $(G_\alpha, P_\alpha, V_\alpha)$ proves/verifies satisfiability of $\mathbb{F}_{r_\alpha}$-arithmetic circuits, while V_α is an $\mathbb{F}_{r_\beta}$-arithmetic circuit; instead, (b) $(G_\beta, P_\beta, V_\beta)$ proves/verifies satisfiability of $\mathbb{F}_{r_\beta}$-arithmetic circuits, while V_β is an $\mathbb{F}_{r_\alpha}$-arithmetic circuit. This property is achieved by instantiating the two zk-SNARKs via a *PCD-friendly 2-cycle of elliptic curves* (see [BCTV14a] for details on how to obtain these), and facilitates recursive proof composition.

Specifically, the core of the PCD system construction is the design of two *PCD circuits*: $C_{\mathsf{pcd},\alpha}$ over the field $\mathbb{F}_{r_\alpha}$ and $C_{\mathsf{pcd},\beta}$ over the field $\mathbb{F}_{r_\beta}$. For a given compliance predicate Π, the two circuits work roughly as follows.

- $C_{\mathsf{pcd},\alpha}$: given input $x_\alpha = \mathsf{msg}$ and witness $a_\alpha = (\mathsf{loc}, \vec{\mathsf{msg}}_{\mathsf{in}}, \vec{\pi}_{\mathsf{in}})$, use V_β to verify that each input message $\vec{\mathsf{msg}}_{\mathsf{in}}[j]$ has a valid proof $\vec{\pi}_{\mathsf{in}}[j]$, and check that Π accepts the output message msg, local data loc, and input messages $\vec{\mathsf{msg}}_{\mathsf{in}}$.
- $C_{\mathsf{pcd},\beta}$: given input $x_\beta = \mathsf{msg}$ and witness $a_\beta = (\pi_\alpha)$, uses V_α to verify that the message msg has a valid proof π_α.

The aforementioned property ensures that fields "match up": $C_{\mathsf{pcd},\alpha}$ is defined over the same field as V_β, and similarly for $C_{\mathsf{pcd},\beta}$ and V_α. (Such field matching is not possible when using a single elliptic curve.) The two PCD circuits are used as follows: P_α proves satisfiability of $C_{\mathsf{pcd},\alpha}$, and the resulting proof π_α attests to the compliance of msg; and P_β proves the satisfiability of $C_{\mathsf{pcd},\beta}$, and the resulting proof π_β provides a "translation" of π_α so that π_β can in turn be used as part of a witness to $C_{\mathsf{pcd},\alpha}$. We refer to $C_{\mathsf{pcd},\alpha}$ as the *compliance circuit*, and $C_{\mathsf{pcd},\beta}$ as the *translation* circuit.

The above description omits several details (relevant to later discussions): to reduce the size of the PCD circuits $C_{\mathsf{pcd},\alpha}$ and $C_{\mathsf{pcd},\beta}$, [BCTV14a] additionally use hashing, pre-computation, and hardcoding. First, the input x_α to $C_{\mathsf{pcd},\alpha}$ actually equals to $H(\mathsf{bits}(\mathsf{vk}_\beta) \| \mathsf{bits}(\mathsf{msg}))$, where H is a collision-resistant function mapping $\{0,1\}$-vectors to $\mathbb{F}_{r_\alpha}$-vectors, vk_β is the verification key for $C_{\mathsf{pcd},\beta}$, and msg is the output message to be checked by Π. This ensures that x_α's length equals H's output length, which only depends on λ. However, H's output is an $\mathbb{F}_{r_\alpha}$-vector, and thus cannot be passed as input to $C_{\mathsf{pcd},\beta}$, which is an $\mathbb{F}_{r_\beta}$-arithmetic circuit. This issue is addressed via two "repacking circuits" that map information content from elements in $\mathbb{F}_{r_\alpha}$ to ones in $\mathbb{F}_{r_\beta}$ and back, respectively. Second, a zk-SNARK verifier V can be viewed as two functions, i.e., an "offline" function V^{offline} (given the verification key vk, compute a *processed verification key* pvk) and an "online" function V^{online} (given pvk, an input x, and proof π, compute the decision bit); the tradeoff between V and V^{online} can be exploited. Finally, vk_α, the verification key for $C_{\mathsf{pcd},\alpha}$, is hardcoded in $C_{\mathsf{pcd},\beta}$. See [BCTV14a] for more details.

From the point of view of this paper, the construction of [BCTV14a] in insufficient, because: (i) it supports a single compliance predicate at a time, while our setting calls for multiple ones; and (ii) it requires the compliance predicate to be "rigid" (i.e., accept a fixed number of messages and have input lengths equal output length), while our setting calls for "flexible" predicates.

6.3 Overview of our construction

We overview the construction of our PCD system, which extends [BCTV14a]'s so to achieve native (and thus more efficient) support for multiple compliance predicates, variable message arity, and varying message lengths.

At high level, our construction consists of the following two parts.

- **Part 1:** given a vector of compliance predicates $\vec{\Pi}$, construct a vector $\vec{C}_{\mathsf{pcd}}$ of PCD circuits. Roughly, for each $\vec{\Pi}[i]$ in $\vec{\Pi}$, we construct two circuits, $C_{\mathsf{pcd},\alpha,i}$ and $C_{\mathsf{pcd},\beta,i}$, tasked with recursive proof composition relative to $\vec{\Pi}[i]$.
- **Part 2:** construct the PCD generator, prover, and verifier. Roughly, the PCD generator $\mathbb{G}$ produces a zk-SNARK key pair for each circuit in $\vec{C}_{\mathsf{pcd}}$; the PCD prover $\mathbb{P}$, to prove compliance relative to $\vec{\Pi}[i]$, produces a zk-SNARK proof of satisfiability for $C_{\mathsf{pcd},\alpha,i}$ and then uses it to produce one for $C_{\mathsf{pcd},\beta,i}$; the PCD verifier $\mathbb{V}$ verifies a zk-SNARK proof by using the appropriate verification key.

Below, we elaborate on these two parts. We also note that the above separation is only conceptual, because the two parts are procedurally entangled (due to hardcoding of certain values).

Part 1: the PCD circuits. For each compliance predicate $\vec{\Pi}[i]$ in $\vec{\Pi}$, we construct two PCD circuits: a *compliance circuit* $C_{\mathsf{pcd},\alpha,i}$, tasked with checking compliance with $\vec{\Pi}[i]$; and a *translation circuit* $C_{\mathsf{pcd},\beta,i}$, tasked with checking proofs attesting to the satisfiability of $C_{\mathsf{pcd},\alpha,i}$.

The design of $C_{\mathsf{pcd},\beta,i}$ is similar to [BCTV14a]'s translation circuit. Namely, $C_{\mathsf{pcd},\beta,i}$ provides a way to translate a zk-SNARK proof relative to the verification key $\vec{\mathsf{vk}}_\alpha[i]$ (generated for $C_{\mathsf{pcd},\alpha,i}$ and hardcoded in $C_{\mathsf{pcd},\beta,i}$) to one relative to the verification key $\vec{\mathsf{vk}}_\beta[i]$ (generated for $C_{\mathsf{pcd},\beta,i}$); the translation only has the goal of matching fields up.

The design of $C_{\mathsf{pcd},\alpha,i}$ extends [BCTV14a]'s compliance circuit, so to take into account the fact that input messages may carry proofs relative to different verification keys (depending on which compliance predicate was used to reason about their compliance). So, while the input x_α to [BCTV14a]'s compliance circuit was $H(\mathsf{bits}(\mathsf{vk}_\beta)\|\mathsf{bits}(\mathsf{msg}))$, we now take the input to $C_{\mathsf{pcd},\alpha,i}$ to be $H(\mathsf{bits}(\mathsf{rt})\|\mathsf{bits}(\mathsf{msg}))$ where rt is the root of the Merkle tree whose leaves consist of the vector $\vec{\mathsf{vk}}_\beta$.[12] The circuit $C_{\mathsf{pcd},\alpha,i}$ then receives, as part of the witness, an authentication path for the verification key required of each input message, and checks this authentication path against rt. Additional details of the construction (e.g., checking that the type of the output message equals $\vec{\Pi}[i].\mathsf{type}$) are discussed later.

Part 2: the PCD generator, prover, and verifier. Next, we outline below the PCD generator, prover, and verifier.

- The PCD generator $\mathbb{G}$, given a vector $\vec{\Pi}$ of compliance predicates, works as follows.
  1. For each i, construct:
     (a) the compliance circuit $C_{\mathsf{pcd},\alpha,i}$ and generate a zk-SNARK key pair $(\vec{\mathsf{pk}}_\alpha[i], \vec{\mathsf{vk}}_\alpha[i])$ for it, and then
     (b) the translation circuit $C_{\mathsf{pcd},\beta,i}$ (hardcoding $\vec{\mathsf{vk}}_\alpha[i]$) and generate a zk-SNARK key pair $(\vec{\mathsf{pk}}_\beta[i], \vec{\mathsf{vk}}_\beta[i])$ for it.
  2. Compute rt, the root of the Merkle tree whose leaves consist of the vector $\vec{\mathsf{vk}}_\beta$.
  3. Output the key pair $(\mathsf{pk}, \mathsf{vk})$, where $\mathsf{pk} := (\vec{\mathsf{pk}}_\alpha, \vec{\mathsf{vk}}_\alpha, \vec{\mathsf{pk}}_\beta, \vec{\mathsf{vk}}_\beta, \mathsf{rt})$ and $\mathsf{vk} = (\vec{\mathsf{vk}}_\beta, \mathsf{rt})$.
- The PCD prover $\mathbb{P}$, given a proving key pk, output message msg, local data loc, and input messages $\vec{\mathsf{msg}}_{\mathsf{in}}$ with proofs $\vec{\pi}_{\mathsf{in}}$, works as follows.
  1. Parse pk as a tuple $(\vec{\mathsf{pk}}_\alpha, \vec{\mathsf{vk}}_\alpha, \vec{\mathsf{pk}}_\beta, \vec{\mathsf{vk}}_\beta, \mathsf{rt})$.

[12]Merely taking x_α to be $H(\mathsf{bits}(\vec{\mathsf{vk}}_\beta)\|\mathsf{bits}(\mathsf{msg}))$ causes $C_{\mathsf{pcd},\alpha,i}$'s number of gates to be linear (not logarithmic) in the number of predicates.

2. Let $i^\star$ be the index of the compliance predicate $\Pi[i^\star]$ in $\vec{\Pi}$ that is satisfied by $(\mathsf{msg}, \mathsf{loc}, \vec{\mathsf{msg}}_{\mathsf{in}})$.
3. Construct a vector $\vec{\mathsf{ap}}$ of authentication paths, where each $\vec{\mathsf{ap}}[j]$ is the authentication path, relative to the root rt, for the leaf $\vec{\mathsf{vk}}_\beta[\vec{\pi}_{\mathsf{in}}[j].\mathsf{idx}]$.
4. Use rt, $(\mathsf{msg}, \mathsf{loc}, \vec{\mathsf{msg}}_{\mathsf{in}})$, and $\vec{\mathsf{ap}}$ to construct an input x_α and a witness a_α for $C_{\mathsf{pcd},\alpha,i}$.
5. Use $\vec{\mathsf{pk}}_\alpha[i^\star]$ to generate a zk-SNARK proof π_α attesting that the compliance circuit $C_{\mathsf{pcd},\alpha,i}$ accepts (x_α, a_α).
6. Use rt and msg to construct an input x_β and a witness a_β for $C_{\mathsf{pcd},\beta,i}$.
7. Use $\vec{\mathsf{pk}}_\beta[i^\star]$ to generate a zk-SNARK proof π_β attesting that the translation circuit $C_{\mathsf{pcd},\beta,i}$ accepts (x_β, a_β).
8. Output the proof π, where $\pi.\mathsf{idx} := i^\star$ and $\pi.\mathsf{proof} := \pi_\beta$.

- The PCD verifier $\mathbb{V}$, given a verification key vk, a message msg, and a proof π, works as follows.
  1. Parse vk as a tuple $(\vec{\mathsf{vk}}_\beta, \mathsf{rt})$.
  2. Set $i^\star := \pi.\mathsf{idx}$ and $\pi_\beta := \pi.\mathsf{proof}$.
  3. Use rt and msg to construct the input x_β for $C_{\mathsf{pcd},\beta,i^\star}$.
  4. Use $\vec{\mathsf{vk}}_\beta[i^\star]$ to check that π_β is a valid zk-SNARK proof for x_β.

Remark 6.1. As in other PCD constructions, proof of knowledge is achieved by recursively extracting "past proofs" from known ones. This process is technically delicate, and a formal treatment of it is in [BCCT13]. Here we only note that the distributed computations for MapReduce considered in this paper are shallow (of logarithmic depth) and are thus quite amenable to recursive proof extraction.

6.4 Details of our construction

We provide more details about the construction of our PCD system.

Representation of a compliance predicate. The choice of representation of a compliance predicate (e.g., whether the predicate is expressed via a machine or a circuit) does not impact the main ideas behind the construction of multi-predicate PCD (see Section 6.3). Yet, some efficiency optimizations depend on this choice, and so henceforth we make it explicit: a compliance predicate Π is represented as an arithmetic circuit (and, in particular, this implies that $\mathsf{cost}(\Pi)$ equals Π's number of gates). As in [BCTV14a], this choice is not arbitrary but, rather, is inherited from the "native" model of computation supported by the underlying zk-SNARK.

Notation for predicates as circuits. Arithmetic circuits are a "rigid" computation model, so we introduce additional notation to support a detailed description of our construction. To each $\mathbb{F}$-arithmetic compliance predicate Π, we associate several quantities: (i) $\mathsf{outlen}(\Pi)$, the payload length of an output message; (ii) $\mathsf{loclen}(\Pi)$, the length of local data; (iii) $\mathsf{max\text{-}arity}(\Pi)$, the maximum number of input messages; and (iv) $\vec{\mathsf{inlen}}(\Pi)$, the vector for which $\vec{\mathsf{inlen}}(\Pi)[j]$ is the payload length for the j-th input message. As for the type of a message (which is merely an integer), it will suffice to use a single element of $\mathbb{F}$ to represent it. Moreover, in order for Π (which is a circuit) to "know" the number $d \in \{0, \dots, \mathsf{max\text{-}arity}(\Pi)\}$ of input messages, we let Π receive d explicitly (encoded as a single field element).

In sum, if we view Π as a function, we can write that, for some $l \in \mathbb{N}$,

$$\Pi \colon \mathbb{F}^{(1+\mathsf{outlen}(\Pi))} \times \mathbb{F}^{\mathsf{loclen}(\Pi)} \times \mathbb{F}^{\sum_{j=1}^{\mathsf{max\text{-}arity}(\Pi)}(1+\vec{\mathsf{inlen}}(\Pi)[j])} \times \mathbb{F} \to \mathbb{F}^l .$$

Indeed, Π receives an output message msg of length $(1+\mathsf{outlen}(\Pi))$; local data loc of length $\mathsf{loclen}(\Pi)$; $\mathsf{max\text{-}arity}(\Pi)$ input messages, where the j-th input message has length $(1+\vec{\mathsf{inlen}}(\Pi)[j])$; and the arity d. For notational convenience, we write $\Pi(\mathsf{msg}, \mathsf{loc}, \vec{\mathsf{msg}}_{\mathsf{in}}, d)$ even when $\vec{\mathsf{msg}}_{\mathsf{in}}$ contains less than $\mathsf{max\text{-}arity}(\Pi)$ messages (and assume that $\vec{\mathsf{msg}}_{\mathsf{in}}$ is extended with arbitrary padding to the correct length).

Ingredients. In addition to the two (preprocessing) zk-SNARKs $(G_\alpha, P_\alpha, V_\alpha)$ and $(G_\beta, P_\beta, V_\beta)$ (see Section 6.2), in the construction we make use of certain arithmetic circuits that we now describe. All all of these circuits are discussed in [BCTV14a] in more detail, so here we review them only at high level.

We use n_α and n_β to denote the number of field elements of an input to the PCD circuits $C_{\mathsf{pcd},\alpha,i}$ and $C_{\mathsf{pcd},\beta,i}$ (for any i), respectively; we set $n_\alpha := \ell_H$ and $n_\beta := \lceil \frac{n_\alpha \cdot \lceil \log r_\alpha \rceil}{\lfloor \log r_\beta \rfloor} \rceil$, where ℓ_H is the number of elements output by the collision-resistant function H; note that n_β is the number of $\mathbb{F}_{r_\beta}$-elements needed to encode n_α $\mathbb{F}_{r_\alpha}$-elements. We use bits to denote a function that, given an input γ in $\mathbb{F}_{r_\alpha}^\ell$ (for some fixed ℓ), outputs γ's binary representation; the corresponding $\mathbb{F}_{r_\alpha}$-arithmetic circuit is denoted C_{bits} and has $\ell \cdot \lceil \log r_\alpha \rceil$ gates.

We use the following circuits. An $\mathbb{F}_{r_\alpha}$-arithmetic circuit $C_{S_{\alpha\to\beta}}$ implementing $S_{\alpha\to\beta}\colon \mathbb{F}_{r_\alpha}^{n_\alpha} \to \mathbb{F}_{r_\alpha}^{n_\beta\cdot\lceil\log r_\beta\rceil}$, the repacking function from $\mathbb{F}_{r_\alpha}$ to $\mathbb{F}_{r_\beta}$ (which encodes n_α elements in $\mathbb{F}_{r_\alpha}$ into n_β elements in $\mathbb{F}_{r_\beta}$ and outputs their binary representation over $\mathbb{F}_{r_\alpha}$); and an $\mathbb{F}_{r_\beta}$-arithmetic circuit $C_{S_{\alpha\leftarrow\beta}}$ implementing $S_{\alpha\leftarrow\beta}\colon \mathbb{F}_{r_\beta}^{n_\beta} \to \mathbb{F}_{r_\beta}^{n_\alpha\cdot\lceil\log r_\alpha\rceil}$, the "inverse" of $S_{\alpha\to\beta}$ (which decodes n_β elements in $\mathbb{F}_{r_\beta}$ back into n_α elements in $\mathbb{F}_{r_\alpha}$ and outputs their binary representation over $\mathbb{F}_{r_\beta}$). An $\mathbb{F}_{r_\beta}$-arithmetic circuit $C_{V_\alpha^{\mathrm{online}}}$ implementing $V_\alpha^{\mathrm{online}}$ for inputs of n_α elements in $\mathbb{F}_{r_\alpha}$ (an input $x_\alpha \in \mathbb{F}_{r_\alpha}^{n_\alpha}$ is given to $C_{V_\alpha^{\mathrm{online}}}$ as a string of $n_\alpha\cdot\lceil\log r_\alpha\rceil$ elements in $\mathbb{F}_{r_\beta}$, each carrying a bit of x_α). An $\mathbb{F}_{r_\alpha}$-arithmetic circuit C_{V_β} implementing V_β for inputs of n_β elements in $\mathbb{F}_{r_\beta}$ (an input $x_\beta \in \mathbb{F}_{r_\beta}^{n_\beta}$ is given to C_{V_β} as a string of $n_\beta\cdot\lceil\log r_\beta\rceil$ elements in $\mathbb{F}_{r_\alpha}$, each carrying a bit of x_β).

Moreover, for a given compliance predicate Π, we use various $\mathbb{F}_{r_\alpha}$-arithmetic circuits that implement the collision-resistant function $H\colon \{0,1\}^* \to \mathbb{F}_{r_\alpha}^{\ell_H}$. Namely, setting $m_{H,\Pi,\mathsf{out}} := (\ell_H + 1 + \mathsf{outlen}(\Pi))\cdot\lceil\log r_\alpha\rceil$ and $\vec{m}_{H,\Pi,\mathsf{in}}[j] := (\ell_H + 1 + \vec{\mathsf{inlen}}(\Pi)[j])\cdot\lceil\log r_\alpha\rceil$, the circuit $C_{H,\Pi,\mathsf{out}}$ implements H for $m_{H,\Pi,\mathsf{out}}$-bit inputs, and $\vec{C}_{H,\Pi,\mathsf{in}}$ is a vector such that each circuit $\vec{C}_{H,\Pi,\mathsf{in}}[j]$ implements H for $\vec{m}_{H,\Pi,\mathsf{in}}[j]$-bit inputs.

Finally, we use an $\mathbb{F}_{r_\alpha}$-arithmetic circuit for verification of Merkle-tree authentication paths, for a Merkle tree constructed using H. Namely, $C_{\mathsf{MERKLE},p}$ implements the function $\mathsf{MERKLE.CheckPath}$ (see Section 2.3) for Merkle trees with p leaves (and thus with authentication paths of length $\lceil\log p\rceil$).

Construction of the PCD circuits. In Figure 14 we provide pseudocode for $\mathsf{MakePCDCircuitA}$ and $\mathsf{MakePCDCircuitB}$, the two functions that we use to construct the compliance and translation PCD circuits (i.e., $C_{\mathsf{pcd},\alpha,i}$ and $C_{\mathsf{pcd},\beta,i}$).

Construction of the PCD generator, prover, and verifier. In Figure 15 we provide pseudocode for the PCD generator $\mathbb{G}$, prover $\mathbb{P}$, and verifier $\mathbb{V}$. The construction works for a vector $\vec{\Pi}$ of $\mathbb{F}_{r_\alpha}$-arithmetic compliance predicates $\vec{\Pi}$.[13] For convenience, we export $i^\star$, the index of the predicate for which compliance is proved, to $\mathbb{P}$'s interface.

6.5 Optimizations and extensions

The multi-predicate PCD construction described in the previous sections can be optimized and extended in several ways. Here we give two such examples.

A common special case. For any predicate $\vec{\Pi}[i]$ in $\vec{\Pi}$, the size of the compliance circuit $C_{\mathsf{pcd},\alpha,i}$ can be reduced if there is a set of types T satisfying the following: (a) each $t \in T$ is unique (i.e., $\vec{\Pi}$ has only one predicate of type t); and (b) $\vec{\Pi}[i]$ accepts input messages of only one type from T or of type 0 (i.e., if $\vec{\Pi}[i]$ accepts input messages $\vec{\mathsf{msg}}_{\mathsf{in}}$, then $\{\mathsf{msg.type}\}_{\mathsf{msg}\in\vec{\mathsf{msg}}_{\mathsf{in}}} \subseteq \{0,t\}$ for some $t \in T$). Indeed, the same verification key can be used to verify all the input proofs $\vec{\pi}_{\mathsf{in}}$, so that $C_{\mathsf{pcd},\alpha,i}$ needs only one copy, rather than $\mathsf{max\text{-}arity}(\vec{\Pi}[i])$ copies, of $C_{\mathsf{MERKLE},p}$.[14] For instance, 6 out of 7 predicates in the MapReduce predicate vector $\vec{\Pi}^{\mathsf{MR}}$ (from Section 5.1) benefit from this optimization.

Extending $\vec{\Pi}$ ex post. The multi-predicate PCD construction that we described fixes the given $\vec{\Pi}$ for the entire lifetime of the system. However, the construction can be modified to allow extending $\vec{\Pi}$ ex post. Namely, we can modify the compliance circuits so to authenticate a (translation-step) verification key via a digital signature, relative to a "master" public key, rather than via a Merkle-tree authentication path. The modification enables a trusted party (e.g., the same system administrator that runs the key generator) to dynamically append to $\vec{\Pi}$ a new compliance predicate, by simply generating a key pair for it and signing the corresponding translation-step verification key.

[13] For comparison, [BCTV14a] consider the following special case: $\vec{\Pi} = (\Pi)$, $\vec{\mathsf{inlen}}(\Pi)[j] = \mathsf{outlen}(\Pi)$ for all j, and $d = \mathsf{max\text{-}arity}(\Pi)$. Also note that, in this case, there are only two message types (namely, 0 and $\Pi.\mathsf{type}$), which is why [BCTV14a] do not discuss message types, and instead only distinguish between messages that are "base case" (of type 0) or not.

[14] And if $\vec{\Pi}[i]$ accepts only input messages of type 0, $C_{\mathsf{pcd},\alpha,i}$ does not need to include any circuits for proof or authentication path verification.

$\underline{\mathsf{MakePCDCircuitA}(\vec{C}_{H,\Pi,\mathsf{in}}, C_{H,\Pi,\mathsf{out}}, C_{S_{\alpha\to\beta}}, C_{V_\beta}, C_{\mathsf{MERKLE},p}, \Pi)}$

Set:

- $n_\alpha := \ell_H$; and
- $h_\alpha := (1 + \mathsf{outlen}(\Pi)) + \mathsf{loclen}(\Pi) + 1 + \ell_H + \sum_{j=1}^{\mathsf{max\text{-}arity}(\Pi)} \Big((1 + \vec{\mathsf{inlen}}(\Pi)[j]) + \ell_{\mathsf{vk},\beta} + \ell_{\pi,\beta} + \ell_{\mathsf{ap}} + 2\Big)$ where
  - $\ell_{\mathsf{vk},\beta}$ is the number of $\mathbb{F}_{r_\alpha}$-elements in a verification key vk_β (for inputs of n_β elements in $\mathbb{F}_{r_\beta}$) relative to $(G_\beta, P_\beta, V_\beta)$,
  - $\ell_{\pi,\beta}$ is the number of $\mathbb{F}_{r_\alpha}$-elements in a zk-SNARK proof π_β relative to $(G_\beta, P_\beta, V_\beta)$, and
  - ℓ_{ap} is the number of $\mathbb{F}_{r_\alpha}$-elements in an authentication path for a Merkle tree with p leaves.

Output the $\mathbb{F}_{r_\alpha}$-arithmetic circuit $C_{\mathsf{pcd},\alpha}$ that, given input $x_\alpha \in \mathbb{F}_{r_\alpha}^{n_\alpha}$ and witness $a_\alpha \in \mathbb{F}_{r_\alpha}^{h_\alpha}$, works as follows:

```
1. Parse the witness $a_\alpha$ as $(\mathsf{msg}, \mathsf{loc}, \vec{\mathsf{msg}}_{\mathsf{in}}, d, \vec{\mathsf{vk}}_\beta, \mathsf{rt}, \vec{\mathsf{ap}}, \vec{\pi}_{\mathsf{in}}, \vec{b}_{\mathsf{res}})$.
2. Check that $\mathsf{msg.type} = \Pi.\mathsf{type}$.
3. Check that $0 \le d \le \mathsf{max\text{-}arity}(\Pi)$.
4. For $j = 1, \ldots, d$:
   (a) Check that $C_{\mathsf{MERKLE},p}(\mathsf{rt}, \vec{\pi}_{\mathsf{in}}[j].\mathsf{idx}, C_{\mathsf{bits}}(\vec{\mathsf{vk}}_\beta[j]), \vec{\mathsf{ap}}[j]) = \vec{b}_{\mathsf{res}}[j]$.
   (b) Compute $x_{\alpha,\mathsf{in},j} := \vec{C}_{H,\Pi,\mathsf{in}}[j](C_{\mathsf{bits}}(\mathsf{rt} \| \vec{\mathsf{msg}}_{\mathsf{in}}[j].\mathsf{type} \| \vec{\mathsf{msg}}_{\mathsf{in}}[j].\mathsf{payload})) \in \mathbb{F}_{r_\alpha}^{n_\alpha}$.
   (c) Compute $x_{\beta,\mathsf{in},j} := C_{S_{\alpha\to\beta}}(x_{\alpha,\mathsf{in},j}) \in \mathbb{F}_{r_\alpha}^{n_\beta \cdot \lceil \log r_\beta \rceil}$.
   (d) Check that $C_{V_\beta}\big(\vec{\mathsf{vk}}_\beta[j], x_{\beta,\mathsf{in},j}, \vec{\pi}_{\mathsf{in}}[j].\mathsf{proof}\big) = \vec{b}_{\mathsf{res}}[j]$.
   (e) Check that $\vec{b}_{\mathsf{res}}[j] \in \{0,1\}$ and $\vec{\mathsf{msg}}_{\mathsf{in}}[j].\mathsf{type} \cdot (1 - \vec{b}_{\mathsf{res}}[j]) = 0$ (i.e., either $\vec{\mathsf{msg}}_{\mathsf{in}}[j]$ is a base-case message or its proof verified).
5. Check that $x_\alpha = C_{H,\Pi,\mathsf{out}}(C_{\mathsf{bits}}(\mathsf{rt} \| \mathsf{msg.type} \| \mathsf{msg.payload}))$.
6. Check that $\Pi(\mathsf{msg}, \mathsf{loc}, \vec{\mathsf{msg}}_{\mathsf{in}}, d)$ accepts.
```

$\underline{\mathsf{MakePCDCircuitB}(\mathsf{pvk}_\alpha, C_{S_{\alpha\leftarrow\beta}}, C_{V_\alpha^{\mathrm{online}}})}$

Set:

- $n_\beta := \left\lceil \frac{n_\alpha \cdot \lceil \log r_\alpha \rceil}{\lfloor \log r_\beta \rfloor} \right\rceil$; and
- $h_\beta := \ell_{\pi,\alpha}$ where $\ell_{\pi,\alpha}$ is the number of $\mathbb{F}_{r_\beta}$-elements in a zk-SNARK proof π_α relative to $(G_\alpha, P_\alpha, V_\alpha)$.

Output the $\mathbb{F}_{r_\beta}$-arithmetic circuit $C_{\mathsf{pcd},\beta}$ that, given input $x_\beta \in \mathbb{F}_{r_\beta}^{n_\beta}$ and witness $a_\beta \in \mathbb{F}_{r_\beta}^{h_\beta}$, works as follows:

```
1. Parse the witness $a_\beta$ as a zk-SNARK proof $\pi_\alpha$.
2. Compute $x_\alpha := C_{S_{\alpha\leftarrow\beta}}(x_\beta) \in \mathbb{F}_{r_\beta}^{n_\alpha \cdot \lceil \log r_\alpha \rceil}$.
3. Check that $C_{V_\alpha^{\mathrm{online}}}\big(\mathsf{pvk}_\alpha, x_\alpha, \pi_\alpha\big) = 1$.
```

Figure 14: Construction of PCD circuits for our multi-predicate PCD system.

PCD generator $\mathbb{G}$

- INPUTS: a vector of p compliance predicates $\vec{\Pi} = (\vec{\Pi}[1], \ldots, \vec{\Pi}[p])$, where each compliance predicate $\vec{\Pi}[i]$ is a $\mathbb{F}_{r_\alpha}$-arithmetic circuit
- OUTPUTS: a proving key pk and a verification key vk

1. Set $n_\alpha := \ell_H$ and $n_\beta := \left\lceil \frac{n_\alpha \cdot \lceil \log r_\alpha \rceil}{\lfloor \log r_\beta \rfloor} \right\rceil$.
2. Construct $C_{S_{\alpha\to\beta}}$, the $\mathbb{F}_{r_\alpha}$-arithmetic circuit implementing $S_{\alpha\to\beta} \colon \mathbb{F}_{r_\alpha}^{n_\alpha} \to \mathbb{F}_{r_\alpha}^{n_\beta \cdot \lceil \log r_\beta \rceil}$.
3. Construct $C_{S_{\alpha\leftarrow\beta}}$, the $\mathbb{F}_{r_\beta}$-arithmetic circuit implementing $S_{\alpha\leftarrow\beta} \colon \mathbb{F}_{r_\beta}^{n_\beta} \to \mathbb{F}_{r_\beta}^{n_\alpha \cdot \lceil \log r_\alpha \rceil}$.
4. Construct C_{V_β}, the $\mathbb{F}_{r_\alpha}$-arithmetic circuit implementing V_β for inputs of n_β elements in $\mathbb{F}_{r_\beta}$.
5. Construct $C_{V_\alpha^{\mathrm{online}}}$, the $\mathbb{F}_{r_\beta}$-arithmetic circuit implementing $V_\alpha^{\mathrm{online}}$ for inputs of n_α elements in $\mathbb{F}_{r_\alpha}$.
6. Construct $C_{\mathsf{MERKLE},p}$, the $\mathbb{F}_{r_\alpha}$-arithmetic circuit implementing $\mathsf{MERKLE.CheckPath}$ for depth $\lceil \log p \rceil$.
7. Allocate the proving key pk, consisting of:
   (a) a Merkle tree root $\mathsf{pk.rt}$; and
   (b) four vectors of length p: $\mathsf{pk}.\vec{\mathsf{pk}}_\alpha$, $\mathsf{pk}.\vec{\mathsf{pk}}_\beta$, $\mathsf{pk}.\vec{\mathsf{vk}}_\alpha$, $\mathsf{pk}.\vec{\mathsf{vk}}_\beta$.
8. Allocate the verification key vk, consisting of:
   (a) a Merkle tree root $\mathsf{vk.rt}$; and
   (b) one vector of length p: $\mathsf{vk}.\vec{\mathsf{vk}}_\beta$.
9. For $i = 1, \ldots, p$, compute proving and verification keys for $\vec{\Pi}[i]$ as follows:
   (a) Construct $C_{H,\vec{\Pi}[i],\mathsf{out}}$, the $\mathbb{F}_{r_\alpha}$-arithmetic circuit implementing $H \colon \{0,1\}^* \to \mathbb{F}_{r_\alpha}^{\ell_H}$ for $m_{H,\vec{\Pi}[i],\mathsf{out}}$-bit inputs.
   (b) Construct $\vec{C}_{H,\vec{\Pi}[i],\mathsf{in}}$, the vector of $\mathbb{F}_{r_\alpha}$-arithmetic circuits such that $\vec{C}_{H,\vec{\Pi}[i],\mathsf{in}}[j]$ implements $H \colon \{0,1\}^* \to \mathbb{F}_{r_\alpha}^{\ell_H}$ for $\vec{m}_{H,\vec{\Pi}[i],\mathsf{in}}[j]$-bit inputs.
   (c) Compute $C_{\mathsf{pcd},\alpha,i} := \mathsf{MakePCDCircuitA}(\vec{C}_{H,\vec{\Pi}[i],\mathsf{in}}, C_{H,\vec{\Pi}[i],\mathsf{out}}, C_{S_{\alpha\to\beta}}, C_{V_\beta}, C_{\mathsf{MERKLE},p}, \vec{\Pi}[i])$.
   (d) Compute $(\mathsf{pk}_{\alpha,i}, \mathsf{vk}_{\alpha,i}) := G_\alpha(C_{\mathsf{pcd},\alpha,i})$.
   (e) Compute $\mathsf{pvk}_{\alpha,i} := V_\alpha^{\mathrm{offline}}(\mathsf{pk}_{\alpha,i})$.
   (f) Compute $C_{\mathsf{pcd},\beta,i} := \mathsf{MakePCDCircuitB}(\mathsf{pvk}_{\alpha,i}, C_{S_{\alpha\leftarrow\beta}}, C_{V_\alpha^{\mathrm{online}}})$.
   (g) Compute $(\mathsf{pk}_{\beta,i}, \mathsf{vk}_{\beta,i}) := G_\beta(C_{\mathsf{pcd},\beta,i})$.
   (h) Set $\mathsf{pk}.\vec{\mathsf{pk}}_\alpha[i] := \mathsf{pk}_{\alpha,i}$, $\mathsf{pk}.\vec{\mathsf{pk}}_\beta[i] := \mathsf{pk}_{\beta,i}$, $\mathsf{pk}.\vec{\mathsf{vk}}_\alpha[i] := \mathsf{vk}_{\alpha,i}$, $\mathsf{pk}.\vec{\mathsf{vk}}_\beta[i] := \mathsf{vk}_{\beta,i}$, $\mathsf{vk}.\vec{\mathsf{vk}}_\beta[i] := \mathsf{vk}_{\beta,i}$.
10. Compute $\mathsf{rt} := \mathsf{MERKLE.GetRoot}(\vec{\mathsf{vk}}_\beta)$ and set $\mathsf{pk.rt} := \mathsf{rt}$, $\mathsf{vk.rt} := \mathsf{rt}$.
11. Output $(\mathsf{pk}, \mathsf{vk})$.

PCD prover $\mathbb{P}$

- INPUTS:
  - proving key pk
  - index $i^\star$ of the compliance predicate $\vec{\Pi}[i^\star]$ in $\vec{\Pi}$, with respect to which compliance is proved
  - output message $\mathsf{msg} \in \mathbb{F}_{r_\alpha}^{1+\mathsf{outlen}(\vec{\Pi}[i^\star])}$
  - local data $\mathsf{loc} \in \mathbb{F}_{r_\alpha}^{\mathsf{loclen}(\vec{\Pi}[i^\star])}$
  - arity $d \in \{0, \ldots, \mathsf{max\text{-}arity}(\vec{\Pi}[i^\star])\}$
  - d input messages $\vec{\mathsf{msg}}_{\mathsf{in}}$, each $\vec{\mathsf{msg}}_{\mathsf{in}}[j] \in \mathbb{F}_{r_\alpha}^{1+\vec{\mathsf{inlen}}(\vec{\Pi}[i^\star])[j]}$
  - d corresponding proofs $\vec{\pi}_{\mathsf{in}}$ (some entries may equal $\perp$, denoting that there is no prior proof)
- OUTPUTS: a PCD proof π for the output message msg as attested by $\vec{\Pi}[i^\star]$

1. Compute $x_\alpha := H_\alpha(\mathsf{bits}(\mathsf{pk.rt} \| \mathsf{msg.type} \| \mathsf{msg.payload})) \in \mathbb{F}_{r_\alpha}^{n_\alpha}$ and $x_\beta := S_{\alpha\to\beta}(x_\alpha) \in \mathbb{F}_{r_\alpha}^{n_\beta \cdot \lceil \log r_\beta \rceil}$, and parse x_β as lying in $\mathbb{F}_{r_\beta}^{n_\beta}$.
2. Let $\vec{\mathsf{vk}}_\beta$, $\vec{\mathsf{ap}}$ and $\vec{b}_{\mathsf{res}}$ be three vectors with length d. For $j = 1, \ldots, d$, do the following:
   (a) If $\vec{\mathsf{msg}}_{\mathsf{in}}[j].\mathsf{type} \neq 0$, set $\vec{b}_{\mathsf{res}}[j] := 1$, set $\vec{\mathsf{vk}}_\beta[j] := \mathsf{pk}.\vec{\mathsf{vk}}_\beta[\vec{\pi}_{\mathsf{in}}[j].\mathsf{idx}]$, and compute $\vec{\mathsf{ap}}[j] := \mathsf{MERKLE.GetPath}(\mathsf{pk}.\vec{\mathsf{vk}}_\beta, \vec{\pi}_{\mathsf{in}}[j].\mathsf{idx})$.
   (b) If $\vec{\mathsf{msg}}_{\mathsf{in}}[j].\mathsf{type} = 0$, set $\vec{b}_{\mathsf{res}}[j] := 0$, and let $\vec{\mathsf{vk}}_\beta[j]$ and $\vec{\mathsf{ap}}[j]$ have arbitrary contents of the correct length.
3. Extend $\vec{\mathsf{msg}}_{\mathsf{in}}$ from a vector of length d to a vector with length $\mathsf{max\text{-}arity}(\vec{\Pi}[i^\star])$ using arbitrary padding. Do the same for $\vec{\pi}_{\mathsf{in}}$, $\vec{\mathsf{vk}}_\beta$, $\vec{\mathsf{ap}}$, and $\vec{b}_{\mathsf{res}}$. For simplicity we denote the padded vectors also by $\vec{\mathsf{msg}}_{\mathsf{in}}$, $\vec{\pi}_{\mathsf{in}}$, $\vec{\mathsf{vk}}_\beta$, $\vec{\mathsf{ap}}$, and $\vec{b}_{\mathsf{res}}$.
4. Set $a_\alpha := (\mathsf{msg}, \mathsf{loc}, \vec{\mathsf{msg}}_{\mathsf{in}}, d, \vec{\mathsf{vk}}_\beta, \mathsf{rt}, \vec{\mathsf{ap}}, \vec{\pi}_{\mathsf{in}}, \vec{b}_{\mathsf{res}})$ and compute $\pi_\alpha := P_\alpha(\mathsf{pk}.\vec{\mathsf{pk}}_\alpha[i^\star], x_\alpha, a_\alpha)$.
5. Set $a_\beta := (\pi_\alpha)$ and compute $\pi_\beta := P_\beta(\mathsf{pk}.\vec{\mathsf{pk}}_\beta[i^\star], x_\beta, a_\beta)$.
6. Output a PCD proof π with $\pi.\mathsf{idx} := i^\star$, $\pi.\mathsf{proof} := \pi_\beta$.

PCD verifier $\mathbb{V}$

- INPUTS:
  - verification key vk
  - message $\mathsf{msg} \in \mathbb{F}_{r_\alpha}^*$
  - proof π
- OUTPUTS: decision bit

1. Interpret π as a PCD proof with $i := \pi.\mathsf{idx}$ and $\pi_\beta := \pi.\mathsf{proof}$.
2. Compute $x_\alpha := H_\alpha(\mathsf{bits}(\mathsf{vk.rt} \| \mathsf{msg.type} \| \mathsf{msg.payload})) \in \mathbb{F}_{r_\alpha}^{n_\alpha}$ and $x_\beta := S_{\alpha\to\beta}(x_\alpha) \in \mathbb{F}_{r_\alpha}^{n_\beta \cdot \lceil \log r_\beta \rceil}$, and parse x_β as lying in $\mathbb{F}_{r_\beta}^{n_\beta}$.
3. Compute $b := V_\beta(\mathsf{vk}.\vec{\mathsf{vk}}_\beta[i], x_\beta, \pi_\beta)$ and output b.

Figure 15: Construction of a multi-predicate PCD system.

7 Implementation

Our system. We built a system that implements our constructions. First, we implemented multi-predicate PCD, providing interfaces for the PCD generator $\mathbb{G}$, prover $\mathbb{P}$, and verifier $\mathbb{V}$; this realizes Step I (see Section 6). Next, we used multi-predicate PCD to implement a distributed zk-SNARK for MapReduce, providing interfaces for the zk-SNARK generator $\mathsf{MR.KeyGen}$, prover $\mathsf{MR.Prove}$, and verifier $\mathsf{MR.Verify}$; this realizes Step II (see Section 5).

The prover in our implementation is itself a MapReduce computation, currently running on an ad-hoc MapReduce implementation; integration with Hadoop [Had], an open-source MapReduce framework, is ongoing.

Integration with `libsnark`. We have integrated our code with `libsnark` [SCI], a `C++` library for zk-SNARKs.

Our multi-predicate PCD provides an alternative to the single-predicate PCD that was already part of `libsnark`. In fact, we have harmonized the two PCD interfaces: the object classes for a compliance predicate, messages, and local data are shared across the two. In terms of concrete parameter choices, our multi-predicate PCD uses the two zk-SNARKs (based on PCD-friendly 2-cycles of elliptic curves) that are also used in the single-predicate PCD.

Our distributed zk-SNARK for MapReduce provides an additional choice of proof system in `libsnark`. A MapReduce pair $(\mathsf{Map}, \mathsf{Reduce})$ can be specified via the same "constraint formalism" used throughout `libsnark` (i.e., *rank-1 constraint systems*), thereby facilitating the re-using and sharing of useful constraint systems.

Prototypical MapReduce example: word counting. For evaluation purposes (see Section 8), we wrote a MapReduce pair $(\mathsf{Map}, \mathsf{Reduce})$ that implements the prototypical MapReduce application of *word counting* [DG04], whose goal is to count the number of occurrences of each word in a text (or a collection of texts). Word counting can be cast in the MapReduce framework, e.g., as follows. Each input record (k^1, v^1) represents a slice of, say, 100 words of the document: the key k^1 is the position of the slice in the document, and the value v^1 is the list of words in the slice. The mapper $\mathsf{Map}_{\mathsf{wordcount}}$, when invoked on an input record (k^1, v^1), emits a list of intermediate records $\big((k^2_1, v^2_1), \dots, (k^2_\ell, v^2_\ell)\big)$, with $\ell \leq 100$, denoting that the word k^2_i appears v^2_i times among the words in the slice v^1. The reducer $\mathsf{Reduce}_{\mathsf{wordcount}}$, when invoked on a particular word k^2 and the vector of counts $\vec{v^2}$ for k^2, emits the output record $(k^3, v^3) = (k^2, \sum_i \vec{v^2}[i])$, which reports the total number of occurrences of k^2 in the collection of input records.

8 Evaluation

We evaluated our system by using it to execute the MapReduce application of word counting (see Section 7).

Cost model for word counting. We ran our system on the word counting example, on our benchmarking system. Each of the reported times represents a security level of 80 bits (`libsnark`'s default for PCD), and is relative to a commodity compute node with a 3.40 GHz Intel Core i7-4770 CPU and 16 GB of RAM available and utilizing all 4 cores. We chose the immortal introduction of Diffie and Hellman's pioneering paper "New directions in cryptography" [DH76], divided into slices of 100 words each, as the input to the MapReduce computation.

By analyzing our system's components, we deduced a cost model of the prover's runtime as a function of M, the number of slices the document was divided into (corresponding to the number of mapper nodes), and R, the number of distinct words in the document (corresponding to the number of reducer nodes):

$$\begin{aligned} &M \cdot \big(\mathsf{ptime}(\Pi^{\mathsf{Map}}_{\mathrm{exe}}) + \mathsf{ptime}(\Pi^{\mathsf{Map}}_{\mathrm{fmt}}) + 2 \cdot \mathsf{ptime}(\Pi^{\mathsf{Map}}_{\mathrm{sum}})\big) \\ +&R \cdot \big(\mathsf{ptime}(\Pi^{\mathsf{Reduce}}_{\mathrm{exe}}) + \mathsf{ptime}(\Pi^{\mathsf{Reduce}}_{\mathrm{fmt}}) + 2 \cdot \mathsf{ptime}(\Pi^{\mathsf{Reduce}}_{\mathrm{sum}})\big) \\ +&\mathsf{ptime}(\Pi_{\mathrm{fin}}) \ . \end{aligned}$$

The above costs have the following meaning, and the following measured values on our reference node:

- $\mathsf{ptime}(\Pi^{\mathsf{Map}}_{\mathrm{exe}}) \approx 9.3$ s is the cost of proving execution of a mapper node;
- $\mathsf{ptime}(\Pi^{\mathsf{Reduce}}_{\mathrm{exe}}) \approx 45.2$ s is the cost of proving execution of a reducer node;
- $\mathsf{ptime}(\Pi^{\mathsf{Map}}_{\mathrm{fmt}}) \approx 13.6$ s and $\mathsf{ptime}(\Pi^{\mathsf{Map}}_{\mathrm{sum}}) \approx 14.2$ s are the costs of proving aggregation of mapper nodes' outputs;
- $\mathsf{ptime}(\Pi^{\mathsf{Reduce}}_{\mathrm{fmt}}) \approx 13.8$ s and $\mathsf{ptime}(\Pi^{\mathsf{Reduce}}_{\mathrm{sum}}) \approx 14.3$ s are the costs of proving aggregation of reducer nodes' outputs; and
- $\mathsf{ptime}(\Pi_{\mathrm{fin}}) \approx 14.3$ s is the cost of proving successful comparison of the outputs of the two aggregations.

Extrapolating the cost model. Our cost model accurately characterizes the prover's runtime for the word counting example. When changing the input, the costs change as follows:

(i) the costs of $\Pi^{\mathsf{Map}}_{\mathrm{fmt}}$ and $\Pi^{\mathsf{Map}}_{\mathrm{sum}}$ remain fixed for all MapReduce computations;

(ii) the costs of $\Pi^{\mathsf{Reduce}}_{\mathrm{fmt}}$, $\Pi^{\mathsf{Reduce}}_{\mathrm{sum}}$ and Π_{fin} remain stable as they only exhibit a slight dependency on the length of k^2, but do not otherwise depend on the specific MapReduce computation;

(iii) the cost of $\Pi^{\mathsf{Map}}_{\mathrm{exe}}$ changes depending on $N^{\max}$, the maximum number of mapper outputs, and Map's cost.

The cost of $\Pi^{\mathsf{Reduce}}_{\mathrm{exe}}$ is dominated by the cost incurred by performing $d^{\max}_{\mathsf{in}}$ proof verifications, each costing $\approx 90{,}000$ gates.

9 Conclusion

In this paper we studied the feasibility of cluster computing in zero knowledge, focusing on a concrete distributed architecture: MapReduce. We designed, built, and evaluated a distributed zk-SNARK for proving the correctness of MapReduce computations. Our approach relies on a new bootstrapping theorem for zk-SNARKs, which transforms any zk-SNARK into a distributed zk-SNARK for MapReduce. Several open problems remain.

First, are there approaches to cluster computing in zero knowledge that do not rely on recursive proof composition? This appears to be an interesting direction even without requiring proofs to be non-interactive and succinct.

Second, further reducing the concrete cost of running a PCD prover (beyond the speedup achieved by our multi-predicate PCD) remains an important question. Fast PCD systems would find exciting applications not only to scalable zero knowledge (as in [BCTV14a]) or distributed zero knowledge (as in this work), but also in many other settings [CT10, CT12, CTV13].

Third, while MapReduce is useful for many distributed computations, it does not fit all. For some iterative computations (e.g., deep network training [DCMC+12]), more general architectures such as Spark [ZCFSS10, ZCDD+12] and GraphLab [LGKB+12] are more suitable. It is an interesting question to explore to what extent one can push "compliance engineering" (or, more generally, constructing distributed zero-knowledge proof systems) for other types of parallel distributed computations.

Finally, exploring concrete applications is an interesting direction. Here, existing state-of-the-art circuit generators that compile high-level languages, such as C, into arithmetic circuits can be used to obtain more complex instances of Map and Reduce functions, which can then be plugged into our system.

Acknowledgments

This work was supported by: the Broadcom Foundation and Tel Aviv University Authentication Initiative; the Center for Science of Information (CSoI), an NSF Science and Technology Center, under grant agreement CCF-0939370; the Check Point Institute for Information Security; the Israeli Centers of Research Excellence I-CORE program (center 4/11); the Israeli Ministry of Science and Technology; the Leona M. & Harry B. Helmsley Charitable Trust; MIT Office of the Provost; and the Simons Foundation.

A Formal definition of (non-distributed) zk-SNARKs for MapReduce

We define zk-SNARKs for MapReduce; an informal definition appears in Section 3.1. A (non-distributed) **zk-SNARK for MapReduce** is a tuple of polynomial-time algorithms

$$(\mathsf{COMM}, \mathsf{MR.KeyGen}, \mathsf{MR.Prove}, \mathsf{MR.Verify})$$

where COMM is a commitment scheme and the other three algorithms satisfy the following properties.

Completeness. The honest prover can convince the verifier for any instance in the language. Namely, for every security parameter λ, MapReduce pair $(\mathsf{Map}, \mathsf{Reduce})$, and instance-witness pair $\big((\mathsf{cm}, \mathbb{y}), (\mathbb{x}, \mathsf{cr})\big) \in \mathscr{R}^{\mathsf{COMM}}_{(\mathsf{Map},\mathsf{Reduce})}$,

$$\Pr\left[\mathsf{MR.Verify}(\mathsf{vk}, \mathsf{cm}, \mathbb{y}, \pi_{\mathsf{MR}}) = 1 \;\middle|\; \begin{array}{r} (\mathsf{pk}, \mathsf{vk}) \leftarrow \mathsf{MR.KeyGen}(1^\lambda, \mathsf{Map}, \mathsf{Reduce}) \\ \pi_{\mathsf{MR}} \leftarrow \mathsf{MR.Prove}(\mathsf{pk}, \mathsf{cm}, \mathbb{y}, \mathbb{x}, \mathsf{cr}) \end{array}\right] = 1 \enspace .$$

Succinctness. For every security parameter λ, MapReduce pair $(\mathsf{Map}, \mathsf{Reduce})$, and key pair $(\mathsf{pk}, \mathsf{vk})$ generated by $\mathsf{MR.KeyGen}(1^\lambda, \mathsf{Map}, \mathsf{Reduce})$, (i) an honestly-generated proof π_{MR} has $O_\lambda(1)$ bits, and (ii) $\mathsf{MR.Verify}(\mathsf{vk}, \mathsf{cm}, \mathbb{y}, \pi_{\mathsf{MR}})$ runs in time $O_\lambda(|\mathbb{y}|)$.

Proof of knowledge (and soundness). If the verifier accepts a proof for an instance, the prover "knows" a witness for that instance. (Thus, soundness holds.) Namely, for every constant $c > 0$ and every polynomial-size adversary A there is a polynomial-size witness extractor E such that, for every large-enough security parameter λ, for every MapReduce pair $(\mathsf{Map}, \mathsf{Reduce})$ of size λ^c,

$$\Pr\left[\begin{array}{c} \mathsf{MR.Verify}(\mathsf{vk}, \mathsf{cm}, \mathbb{y}, \pi_{\mathsf{MR}}) = 1 \\ \big((\mathsf{cm}, \mathbb{y}), (\mathbb{x}, \mathsf{cr})\big) \notin \mathscr{R}^{\mathsf{COMM}}_{(\mathsf{Map},\mathsf{Reduce})} \end{array} \;\middle|\; \begin{array}{r} (\mathsf{pk}, \mathsf{vk}) \leftarrow \mathsf{MR.KeyGen}(1^\lambda, \mathsf{Map}, \mathsf{Reduce}) \\ (\mathsf{cm}, \mathbb{y}, \pi_{\mathsf{MR}}) \leftarrow A(\mathsf{pk}, \mathsf{vk}) \\ (\mathbb{x}, \mathsf{cr}) \leftarrow E(\mathsf{pk}, \mathsf{vk}) \end{array}\right] \leq \mathsf{negl}(\lambda) \enspace .$$

Statistical zero knowledge. An honestly-generated proof is statistical zero knowledge.[15] Namely, there is a polynomial-time stateful simulator S such that, for all stateful distinguishers D, the following probabilities are negligibly-close:

$$\Pr\left[\begin{array}{c} \big((\mathsf{cm}, \mathbb{y}), (\mathbb{x}, \mathsf{cr})\big) \in \mathscr{R}^{\mathsf{COMM}}_{(\mathsf{Map},\mathsf{Reduce})} \\ D(\pi_{\mathsf{MR}}) = 1 \end{array} \;\middle|\; \begin{array}{r} (\mathsf{Map}, \mathsf{Reduce}) \leftarrow D(1^\lambda) \\ (\mathsf{pk}, \mathsf{vk}) \leftarrow \mathsf{MR.KeyGen}(1^\lambda, \mathsf{Map}, \mathsf{Reduce}) \\ \big((\mathsf{cm}, \mathbb{y}), (\mathbb{x}, \mathsf{cr})\big) \leftarrow D(\mathsf{pk}, \mathsf{vk}) \\ \pi_{\mathsf{MR}} \leftarrow \mathsf{MR.Prove}(\mathsf{pk}, \mathsf{cm}, \mathbb{y}, \mathbb{x}, \mathsf{cr}) \end{array}\right]$$

and

$$\Pr\left[\begin{array}{c} \big((\mathsf{cm}, \mathbb{y}), (\mathbb{x}, \mathsf{cr})\big) \in \mathscr{R}^{\mathsf{COMM}}_{(\mathsf{Map},\mathsf{Reduce})} \\ D(\pi_{\mathsf{MR}}) = 1 \end{array} \;\middle|\; \begin{array}{r} (\mathsf{Map}, \mathsf{Reduce}) \leftarrow D(1^\lambda) \\ (\mathsf{pk}, \mathsf{vk}) \leftarrow S(1^\lambda, \mathsf{Map}, \mathsf{Reduce}) \\ \big((\mathsf{cm}, \mathbb{y}), (\mathbb{x}, \mathsf{cr})\big) \leftarrow D(\mathsf{pk}, \mathsf{vk}) \\ \pi_{\mathsf{MR}} \leftarrow S(\mathsf{cm}, \mathbb{y}) \end{array}\right] \enspace .$$

[15]Perfect zero knowledge can be achieved at the expense of a negligible probability of error in the completeness property.

B Formal definition of PCD with multiple predicates

We define multi-predicate PCD by giving syntactic and correctness properties of a multi-predicate PCD system; an informal definition appears in Section 4. We assume familiarity with the notions of transcript and compliance (see Section 4). A **multi-predicate PCD system** is a triple of polynomial-time algorithms $(\mathbb{G}, \mathbb{P}, \mathbb{V})$ working as follows.

- $\mathbb{G}(1^\lambda, \vec{\Pi}) \to (\mathsf{pk}, \mathsf{vk})$. On input a security parameter λ (presented in unary) and a vector of compliance predicates $\vec{\Pi}$, the *key generator* $\mathbb{G}$ probabilistically samples a proving key pk and a verification key vk. We assume, without loss of generality, that pk contains (a description of) the vector $\vec{\Pi}$.

The keys pk and vk are published as public parameters and can be used, any number of times, to prove/verify $\vec{\Pi}$-compliance of messages, as follows.

- $\mathbb{P}(\mathsf{pk}, \mathsf{msg}, \mathsf{loc}, \vec{\mathsf{msg}}_{\mathsf{in}}, \vec{\pi}_{\mathsf{in}}) \to \pi$. On input a proving key pk, output message msg, local data loc, and input messages $\vec{\mathsf{msg}}_{\mathsf{in}}$ with proofs $\vec{\pi}_{\mathsf{in}}$, the *prover* $\mathbb{P}$ outputs a proof π for the statement "msg is $\vec{\Pi}$-compliant".
- $\mathbb{V}(\mathsf{vk}, \mathsf{msg}, \pi) \to b$. On input a verification key vk, a message msg, and a proof π, the *verifier* $\mathbb{V}$ outputs $b = 1$ if he is convinced by π that msg is $\vec{\Pi}$-compliant.

The triple $(\mathbb{G}, \mathbb{P}, \mathbb{V})$ satisfies the following properties.

Completeness. The honest prover can convince the verifier that the output of any compliant transcript is indeed compliant. Namely, for every security parameter λ, vector of compliance predicates $\vec{\Pi}$, and transcript generator TGen,

$$\Pr\left[\begin{array}{c|r} \vec{\Pi}(\mathsf{T}) = \mathsf{OK} & (\mathsf{pk}, \mathsf{vk}) \leftarrow \mathbb{G}(1^\lambda, \vec{\Pi}) \\ \mathbb{V}\big(\mathsf{vk}, \mathsf{msg}, \pi\big) \neq 1 & (\mathsf{T}, \mathsf{msg}, \pi) \leftarrow \mathsf{ProofGen}(\mathsf{pk}, \mathsf{TGen}) \end{array}\right] = 0 \ .$$

Above, $\mathsf{ProofGen}(\mathsf{pk}, \mathsf{TGen})$ denotes the following procedure: (i) let T be the transcript generated by TGen; (ii) let TGen be the PCD meta-prover corresponding to TGen; (iii) let $\vec{\pi}$ be the proofs output by $\mathsf{TGen}(\mathsf{pk})$; (iv) let msg be the lexicographically-first message in $\mathrm{OUTS}(\mathsf{T})$, and π the corresponding proof in $\vec{\pi}$; (v) output $(\mathsf{T}, \mathsf{msg}, \pi)$. In other words, completeness requires that if T is $\vec{\Pi}$-compliant then msg's proof (which is the result of invoking $\mathbb{P}$ for each message in T, as T was being constructed by TGen) is accepted by the PCD verifier $\mathbb{V}$.

Succinctness. For every security parameter λ, vector of predicates $\vec{\Pi}$, and $(\mathsf{pk}, \mathsf{vk}) \in \mathbb{G}(1^\lambda, \vec{\Pi})$, (i) an honestly-generated proof π has $O_\lambda(1)$ bits, and (ii) $\mathbb{V}(\mathsf{vk}, \mathsf{msg}, \pi)$ runs in time $O_\lambda(|\mathsf{msg}|)$.

Proof of knowledge (and soundness). If the verifier accepts a proof for a message msg, the prover "knows" a compliant transcript T with output msg. (Thus, soundness holds.) Namely, for every constant $c > 0$ and every polynomial-size adversary A there is a polynomial-size witness extractor E such that, for every large-enough security parameter λ, for every vector of compliance predicates $\vec{\Pi}$ of size λ^c,

$$\Pr\left[\begin{array}{c|r} \mathbb{V}(\mathsf{vk}, \mathsf{msg}, \pi) = 1 & (\mathsf{pk}, \mathsf{vk}) \leftarrow \mathbb{G}(1^\lambda, \vec{\Pi}) \\ \Big(\mathsf{msg} \notin \mathrm{OUTS}(\mathsf{T}) \ \vee \ \vec{\Pi}(\mathsf{T}) \neq \mathsf{OK}\Big) & (\mathsf{msg}, \pi) \leftarrow A(\mathsf{pk}, \mathsf{vk}) \\ & \mathsf{T} \leftarrow E(\mathsf{pk}, \mathsf{vk}) \end{array}\right] \leq \mathsf{negl}(\lambda) \ .$$

Statistical zero knowledge. An honestly-generated proof is statistical zero knowledge.[16] Namely, there is a polynomial-time stateful simulator S such that, for all stateful distinguishers D, the following probabilities are negligibly-close:

$$\Pr\left[\begin{array}{c|r} & \vec{\Pi} \leftarrow D(1^\lambda) \\ \Phi = 1 & (\mathsf{pk}, \mathsf{vk}) \leftarrow \mathbb{G}(1^\lambda, \vec{\Pi}) \\ & (\mathsf{msg}, \mathsf{loc}, \vec{\mathsf{msg}}_{\mathsf{in}}, \vec{\pi}_{\mathsf{in}}) \leftarrow D(\mathsf{pk}, \mathsf{vk}) \\ & \pi \leftarrow \mathbb{P}(\mathsf{pk}, \mathsf{msg}, \mathsf{loc}, \vec{\mathsf{msg}}_{\mathsf{in}}, \vec{\pi}_{\mathsf{in}}) \end{array}\right] \quad \text{and} \quad \Pr\left[\begin{array}{c|r} & \vec{\Pi} \leftarrow D(1^\lambda) \\ \Phi = 1 & (\mathsf{pk}, \mathsf{vk}) \leftarrow S(1^\lambda, \vec{\Pi}) \\ & (\mathsf{msg}, \mathsf{loc}, \vec{\mathsf{msg}}_{\mathsf{in}}, \vec{\pi}_{\mathsf{in}}) \leftarrow D(\mathsf{pk}, \mathsf{vk}) \\ & \pi \leftarrow S(\mathsf{msg}) \end{array}\right] .$$

Above, $\Phi = 1$ if and only if: (i) there is $\Pi \in \vec{\Pi}$ such that $\Pi(\mathsf{msg}, \mathsf{loc}, \vec{\mathsf{msg}}_{\mathsf{in}})$ accepts; (ii) for each $i \in \{1, \ldots, \mathsf{len}(\vec{\mathsf{msg}}_{\mathsf{in}})\}$, either $\vec{\mathsf{msg}}_{\mathsf{in}}[j].\mathsf{type} = 0$ or $\mathbb{V}(\mathsf{vk}, \vec{\mathsf{msg}}_{\mathsf{in}}[j], \vec{\pi}_{\mathsf{in}}[j]) = 1$; and (iii) $D(\pi) = 1$.

[16]Perfect zero knowledge can be achieved at the expense of a negligible probability of error in the completeness property.

C Formal definition of preprocessing zk-SNARKs

The text of this definition is from [BCTV14a]. Given a field $\mathbb{F}$, a **preprocessing zk-SNARK** [BCIOP13, BCCT13] for $\mathbb{F}$-arithmetic circuit satisfiability is a triple of polynomial-time algorithms (G, P, V), with V deterministic, working as follows.

- $G(1^\lambda, C) \to (\mathsf{pk}, \mathsf{vk})$. On input a security parameter λ (presented in unary) and an $\mathbb{F}$-arithmetic circuit C, the *key generator* G probabilistically samples a proving key pk and a verification key vk. We assume, without loss of generality, that pk contains (a description of) the circuit C.

The keys pk and vk are published as public parameters and can be used, any number of times, to prove/verify knowledge of witnesses in the relation $\mathscr{R}_C$, as follows.

- $P(\mathsf{pk}, x, a) \to \pi$. On input a proving key pk and any $(x, a) \in \mathscr{R}_C$, the *prover* P outputs a non-interactive proof π for the statement "there is a such that $(x, a) \in \mathscr{R}_C$".

- $V(\mathsf{vk}, x, \pi) \to b$. On input a verification key vk, an input x, and a proof π, the *verifier* V outputs $b = 1$ if he is convinced by π that there is a such that $(x, a) \in \mathscr{R}_C$.

The triple (G, P, V) satisfies the following properties.

Completeness. The honest prover can convince the verifier for any instance in the language. Namely, for every security parameter λ, $\mathbb{F}$-arithmetic circuit C, and instance-witness pair $(x, a) \in \mathscr{R}_C$,

$$\Pr\left[V(\mathsf{vk}, x, \pi) = 1 \;\middle|\; \begin{array}{r} (\mathsf{pk}, \mathsf{vk}) \leftarrow G(1^\lambda, C) \\ \pi \leftarrow P(\mathsf{pk}, x, a) \end{array} \right] = 1 \ .$$

Succinctness. For every security parameter λ, $\mathbb{F}$-arithmetic circuit C, and $(\mathsf{pk}, \mathsf{vk}) \in G(1^\lambda, C)$, (i) an honestly-generated proof π has $O_\lambda(1)$ bits, and (ii) $V(\mathsf{vk}, x, \pi)$ runs in time $O_\lambda(|x|)$.

Proof of knowledge (and soundness). If the verifier accepts a proof for an instance, the prover "knows" a witness for that instance. (Thus, soundness holds.) Namely, for every constant $c > 0$ and every polynomial-size adversary A there is a polynomial-size witness extractor E such that, for every large-enough security parameter λ, for every $\mathbb{F}$-arithmetic circuit C of size λ^c,

$$\Pr\left[\begin{array}{c} V(\mathsf{vk}, x, \pi) = 1 \\ (x, a) \notin \mathscr{R}_C \end{array} \;\middle|\; \begin{array}{r} (\mathsf{pk}, \mathsf{vk}) \leftarrow G(1^\lambda, C) \\ (x, \pi) \leftarrow A(\mathsf{pk}, \mathsf{vk}) \\ a \leftarrow E(\mathsf{pk}, \mathsf{vk}) \end{array} \right] \leq \mathsf{negl}(\lambda) \ .$$

Statistical zero knowledge. An honestly-generated proof is statistical zero knowledge.[17] Namely, there is a polynomial-time stateful simulator S such that, for all stateful distinguishers D, the following two probabilities are negligibly-close:

$$\Pr\left[\begin{array}{c} (x, a) \in \mathscr{R}_C \\ D(\pi) = 1 \end{array} \;\middle|\; \begin{array}{r} C \leftarrow D(1^\lambda) \\ (\mathsf{pk}, \mathsf{vk}) \leftarrow G(1^\lambda, C) \\ (x, a) \leftarrow D(\mathsf{pk}, \mathsf{vk}) \\ \pi \leftarrow P(\mathsf{pk}, x, a) \end{array} \right] \quad \text{and} \quad \Pr\left[\begin{array}{c} (x, a) \in \mathscr{R}_C \\ D(\pi) = 1 \end{array} \;\middle|\; \begin{array}{r} C \leftarrow D(1^\lambda) \\ (\mathsf{pk}, \mathsf{vk}) \leftarrow S(1^\lambda, C) \\ (x, a) \leftarrow D(\mathsf{pk}, \mathsf{vk}) \\ \pi \leftarrow S(x) \end{array} \right] \ .$$

[17] Perfect zero knowledge can be achieved at the expense of a negligible probability of error in the completeness property.

D More details for the construction of $\vec{\Pi}^{\mathsf{MR}}$

We provide additional details for the construction of $\vec{\Pi}^{\mathsf{MR}} = (\Pi^{\mathsf{Map}}_{\mathrm{exe}}, \Pi^{\mathsf{Reduce}}_{\mathrm{exe}}, \Pi^{\mathsf{Map}}_{\mathrm{fmt}}, \Pi^{\mathsf{Map}}_{\mathrm{sum}}, \Pi^{\mathsf{Reduce}}_{\mathrm{fmt}}, \Pi^{\mathsf{Reduce}}_{\mathrm{sum}}, \Pi_{\mathrm{fin}})$. While in Section 5.1 we consider only the (artificial) case where each mapper outputs a single phase-2 record, the construction below also handles the case where each mapper may output multiple phase-2 records.

Furthermore, the construction below handles the extensions mentioned in Section 2.4.3: (a) a reducer may output multiple phase-3 records; (b) a Map or Reduce function takes as additional input a set of parameters p; and (c) a Map or Reduce function takes as additional input an auxiliary input z. The relation $\mathscr{R}^{\mathsf{COMM}}_{(\mathsf{Map},\mathsf{Reduce})}$ is correspondingly modified to comprise the pairs $\big((p, \mathsf{cm}, \mathbb{y}), (z, \mathbb{x}, \mathsf{cr})\big)$ such that $\mathsf{COMM.Ver}(\mathbb{x}, \mathsf{cm}, \mathsf{cr}) = 1$ and $\mathbb{y} = [\mathsf{Map}_{p,z}, \mathsf{Reduce}_{p,z}](\mathbb{x})$.

In the figures below (Figure 16, Figure 17, Figure 18, Figure 19) we give pseudocode for the upgraded compliance predicates in $\vec{\Pi}^{\mathsf{MR}}$. Beyond the ingredients already used in Section 2.4.3, we also use a collision-resistant function H.

```
$\underline{\Pi^{\mathsf{Map}}_{\mathrm{exe}}(\mathsf{msg}, \mathsf{loc}, \vec{\mathsf{msg}}_{\mathsf{in}})}$
1. Let $d$ be the number of input messages in $\vec{\mathsf{msg}}_{\mathsf{in}}$.
2. Check that $d = 1$.
3. Check that $\vec{\mathsf{msg}}_{\mathsf{in}}[1].\mathsf{type} = 0$.
4. Parse $\vec{\mathsf{msg}}_{\mathsf{in}}[1].\mathsf{payload}$ as a tuple $(\mathsf{cm}, i, k^1, v^1)$ where:
   • $\mathsf{cm}$ is a commitment (for the scheme $\mathsf{COMM}$);
   • $i$ is an index;
   • $(k^1, v^1)$ is a phase-1 record.
5. Parse $\mathsf{msg.payload}$ as a tuple $(\mathsf{sid}, i', N, \rho)$ where:
   • $\mathsf{sid}$ is a session identifier;
   • $i'$ is an index;
   • $N$ is a positive integer;
   • $\rho$ is a root (for the scheme $\mathsf{MERKLE}$).
6. Parse $\mathsf{loc}$ as a tuple
       $\big(p, z, \mathsf{rt}, M, \mathsf{cm}_z, \mathsf{cr}_{\mathsf{rt}}, \mathsf{cr}_M, \mathsf{cr}_z, \mathsf{ap}, ((\mathsf{ap}_l, k^2_l, v^2_l))_{l=1}^N\big)$
   where:
   • $p$ are MapReduce parameters;
   • $z$ is a MapReduce auxiliary input;
   • $\mathsf{rt}$ is a commitment (for the scheme $\mathsf{MERKLE}$);
   • $M$ is a positive integer;
   • $\mathsf{cm}_z$ is a commitment (for the scheme $\mathsf{COMM}^*$);
   • $\mathsf{cr}_{\mathsf{rt}}, \mathsf{cr}_M, \mathsf{cr}_z$ are randomness (for the scheme $\mathsf{COMM}^*$);
   • $\mathsf{ap}, \mathsf{ap}_1, \dots, \mathsf{ap}_N$ are authentication paths (for the scheme $\mathsf{MERKLE}$);
   • $(k^2_1, v^2_1), \dots, (k^2_N, v^2_N)$ are phase-2 records.
7. Check that $\mathsf{sid} = H(\mathsf{cm} \| p \| \mathsf{cm}_z)$.
8. Check that $i' = i$.
9. Check that $0 \le i < M$.
10. Parse $\mathsf{cm}$ as a pair $(\mathsf{cm}_{\mathsf{rt}}, \mathsf{cm}_M)$ where both components are commitments
    for the scheme $\mathsf{COMM}^*$.
11. Check that $\mathsf{COMM}^*.\mathsf{Ver}(\mathsf{rt}, \mathsf{cm}_{\mathsf{rt}}, \mathsf{cr}_{\mathsf{rt}}) = 1$.
12. Check that $\mathsf{COMM}^*.\mathsf{Ver}(M, \mathsf{cm}_M, \mathsf{cr}_M) = 1$.
13. Check that $\mathsf{COMM}^*.\mathsf{Ver}(z, \mathsf{cm}_z, \mathsf{cr}_z) = 1$.
14. Check that $\mathsf{MERKLE.CheckPath}\big(\mathsf{rt}, i, (k^1, v^1), \mathsf{ap}\big) = 1$.
15. For $l = 1, \dots, N$:
        check that $\mathsf{MERKLE.CheckPath}\big(\rho, l, (k^2_l, v^2_l), \mathsf{ap}_l\big) = 1$.
16. Check that $\big((k^2_l, v^2_l)\big)_{l=1}^N = \mathsf{Map}_{p,z}(k^1, v^1)$.
```

```
$\underline{\Pi^{\mathsf{Reduce}}_{\mathrm{exe}}(\mathsf{msg}, \mathsf{loc}, \vec{\mathsf{msg}}_{\mathsf{in}})}$
1. Check that $\vec{\mathsf{msg}}_{\mathsf{in}}[j].\mathsf{type} = \Pi^{\mathsf{Map}}_{\mathrm{exe}}.\mathsf{type}$ for each $j$.
2. Parse each $\vec{\mathsf{msg}}_{\mathsf{in}}[j].\mathsf{payload}$ as a tuple $(\mathsf{sid}'_j, i'_j, N_j, \rho_j)$ where:
   • $\mathsf{sid}'_j$ is a session identifier;
   • $i'_j$ is an index;
   • $N_j$ is a positive integer;
   • $\rho_j$ is a root (for the scheme $\mathsf{MERKLE}$).
3. Parse $\mathsf{msg.payload}$ as a tuple $(\mathsf{sid}'', \rho, \mathsf{cm}_{k^2}, \mathsf{cm}_{d_{\mathsf{in}}})$ where:
   • $\mathsf{sid}''$ is a session identifier;
   • $d_{\mathsf{out}}$ is a positive integer;
   • $\rho$ is a root (for the scheme $\mathsf{MERKLE}$);
   • $\mathsf{cm}_{k^2}, \mathsf{cm}_{d_{\mathsf{in}}}$ are commitments (for the scheme $\mathsf{COMM}^*$).
4. Parse $\mathsf{loc}$ as a tuple
       $\big(p, z, \mathsf{cr}_z, \mathsf{cr}_{k^2}, \mathsf{cr}_{d_{\mathsf{in}}}, (l_j, \mathsf{ap}_{\mathsf{in},j}, k^2_j, v^2_j)_{j=1}^{d_{\mathsf{in}}}, (\mathsf{ap}_{\mathsf{out},o}, k^3_o, v^3_o)_{o=1}^{d_{\mathsf{out}}}\big)$
   where:
   • $p$ are MapReduce parameters;
   • $z$ is a MapReduce auxiliary input;
   • $\mathsf{cm}_z$ is a commitment (for the scheme $\mathsf{COMM}^*$);
   • $\mathsf{cr}_{k^2}, \mathsf{cr}_{d_{\mathsf{in}}}$ are randomness (for the scheme $\mathsf{COMM}^*$).
   • $l_1, \dots, l_{d_{\mathsf{in}}}$ are indices;
   • $\mathsf{ap}_{\mathsf{in},1}, \dots, \mathsf{ap}_{\mathsf{in},d_{\mathsf{in}}}$ are authentication paths (for the scheme $\mathsf{MERKLE}$);
   • $\mathsf{ap}_{\mathsf{out},1}, \dots, \mathsf{ap}_{\mathsf{out},d_{\mathsf{out}}}$ are authentication paths (for the scheme $\mathsf{MERKLE}$);
   • $(k^2_1, v^2_1), \dots, (k^2_{d_{\mathsf{in}}}, v^2_{d_{\mathsf{in}}})$ are phase-2 records;
   • $(k^3_1, v^3_1), \dots, (k^3_{d_{\mathsf{out}}}, v^3_{d_{\mathsf{out}}})$ are phase-3 records.
5. Check that $\mathsf{sid}'' = H(\mathsf{cm} \| p \| \mathsf{cm}_z)$.
6. Check that $\mathsf{sid}'' = \mathsf{sid}'_j$ for each $j$.
7. Check that the $(i'_j, l_j)$ are distinct, and let $d_{\mathsf{in}}$ be their number.
8. Check that all the $k^2_j$'s are equal, and set $\vec{v^2} := (v^2_j)_j$.
9. Check that $\mathsf{COMM}^*.\mathsf{Ver}(k^2_1, \mathsf{cm}_{k^2}, \mathsf{cr}_{k^2}) = 1$.
10. Check that $\mathsf{COMM}^*.\mathsf{Ver}(d_{\mathsf{in}}, \mathsf{cm}_{d_{\mathsf{in}}}, \mathsf{cr}_{d_{\mathsf{in}}}) = 1$.
11. For $j = 1, \dots, d_{\mathsf{in}}$:
        check that $0 \le l_j < N_j$ and
        check that $\mathsf{MERKLE.CheckPath}\big(\rho_j, l_j, (k^2_j, v^2_j), \mathsf{ap}_{\mathsf{in},j}\big) = 1$.
12. For $o = 1, \dots, d_{\mathsf{out}}$:
        check that $\mathsf{MERKLE.CheckPath}\big(\rho, o, (k^3_o, v^3_o), \mathsf{ap}_{\mathsf{out},o}\big) = 1$.
13. Check that $\big((k^3_o, v^3_o)\big)_{o=1}^{d_{\mathsf{out}}} = \mathsf{Reduce}_{p,z}(k^2_1, \vec{v^2})$.
```

Figure 16: Summary of the construction of $\Pi^{\mathsf{Map}}_{\mathrm{exe}}$ and $\Pi^{\mathsf{Reduce}}_{\mathrm{exe}}$.

```
$\Pi^{\mathsf{Map}}_{\mathrm{fmt}}(\mathsf{msg}, \mathsf{loc}, \vec{\mathsf{msg}}_{\mathsf{in}})$
 1. Let $d$ be the number of input messages in $\vec{\mathsf{msg}}_{\mathsf{in}}$.
 2. Check that $d = 1$.
 3. Check that $\vec{\mathsf{msg}}_{\mathsf{in}}[1].\mathsf{type} = \Pi^{\mathsf{Map}}_{\mathrm{exe}}.\mathsf{type}$.
 4. Parse $\vec{\mathsf{msg}}_{\mathsf{in}}[1].\mathsf{payload}$ as a tuple $(\mathsf{sid}, i, N, \rho)$ where:
    • $\mathsf{sid}$ is a session identifier;
    • $i$ is an index;
    • $N$ is a positive integer;
    • $\rho$ is a root (for the scheme $\mathsf{MERKLE}$).
 5. The local data $\mathsf{loc}$ is not used; ignore it.
 6. Parse $\mathsf{msg}.\mathsf{payload}$ as a tuple $(\mathsf{sid}', a^{\perp}, a^{\top}, b, c)$ where:
    • $\mathsf{sid}'$ is a session identifier;
    • $a^{\perp}, a^{\top}, b, c$ are positive integers.
 7. Check that $\mathsf{sid}' = \mathsf{sid}$.
 8. Check that $a^{\perp} = a^{\top} = i$.
 9. Check that $b = 1$.
10. Check that $c = N$.
```

```
$\Pi^{\mathsf{Reduce}}_{\mathrm{fmt}}(\mathsf{msg}, \mathsf{loc}, \vec{\mathsf{msg}}_{\mathsf{in}})$
 1. Let $d$ be the number of input messages in $\vec{\mathsf{msg}}_{\mathsf{in}}$.
 2. Check that $d = 1$.
 3. Check that $\vec{\mathsf{msg}}_{\mathsf{in}}[1].\mathsf{type} = \Pi^{\mathsf{Reduce}}_{\mathrm{exe}}.\mathsf{type}$.
 4. Parse $\vec{\mathsf{msg}}_{\mathsf{in}}[1].\mathsf{payload}$ as a tuple $(\mathsf{sid}, \rho, \mathsf{cm}_{k^2}, \mathsf{cm}_{d_{\mathsf{in}}})$ where:
    • $\mathsf{sid}$ is a session identifier;
    • $\rho$ is a root (for the scheme $\mathsf{MERKLE}$);
    • $\mathsf{cm}_{k^2}, \mathsf{cm}_{d_{\mathsf{in}}}$ are commitments (for the scheme $\mathsf{COMM}^*$).
 5. Parse $\mathsf{loc}$ as a tuple $(k^2, d_{\mathsf{in}}, \mathsf{cr}_{k^2}, \mathsf{cr}_{d_{\mathsf{in}}})$, where:
    • $k^2$ is a phase-2 key;
    • $d_{\mathsf{in}}$ is a positive integer;
    • $\mathsf{cr}_{k^2}, \mathsf{cr}_{d_{\mathsf{in}}}$ are randomness (for the scheme $\mathsf{COMM}^*$).
 6. Parse $\mathsf{msg}.\mathsf{payload}$ as a tuple $(\mathsf{sid}', a^{\perp}, a^{\top}, b, c)$ where:
    • $\mathsf{sid}'$ is a session identifier;
    • $a^{\perp}, a^{\top}, b, c$ are positive integers.
 7. Check that $\mathsf{sid}' = \mathsf{sid}$.
 8. Check that $\mathsf{COMM}^*.\mathsf{Ver}(k^2, \mathsf{cm}_{k^2}, \mathsf{cr}_{k^2}) = 1$.
 9. Check that $\mathsf{COMM}^*.\mathsf{Ver}(d_{\mathsf{in}}, \mathsf{cm}_{d_{\mathsf{in}}}, \mathsf{cr}_{d_{\mathsf{in}}}) = 1$.
10. Check that $a^{\perp} = a^{\top} = k^2$ (after converting $k^2$ to a positive integer).
11. Check that $b = 1$.
12. Check that $c = d_{\mathsf{in}}$.
```

Figure 17: Summary of the construction of $\Pi^{\mathsf{Map}}_{\mathrm{fmt}}$ and $\Pi^{\mathsf{Reduce}}_{\mathrm{fmt}}$.

```
$\Pi^{\mathsf{Map}}_{\mathrm{sum}}(\mathsf{msg}, \mathsf{loc}, \vec{\mathsf{msg}}_{\mathsf{in}})$
 1. Let $d$ be the number of input messages in $\vec{\mathsf{msg}}_{\mathsf{in}}$.
 2. Check that $d = 2$.
 3. Check that each $\vec{\mathsf{msg}}_{\mathsf{in}}[j].\mathsf{type}$ lies in $\{\Pi^{\mathsf{Map}}_{\mathrm{fmt}}.\mathsf{type}, \Pi^{\mathsf{Map}}_{\mathrm{sum}}.\mathsf{type}\}$.
 4. Parse each $\vec{\mathsf{msg}}_{\mathsf{in}}[j].\mathsf{payload}$ as a tuple $(\mathsf{sid}_j, a^{\perp}_j, a^{\top}_j, b_j, c_j)$ where:
    • $\mathsf{sid}_j$ is a session identifier;
    • $a^{\perp}_j, a^{\top}_j, b_j, c_j$ are positive integers.
 5. The local data $\mathsf{loc}$ is not used; ignore it.
 6. Parse $\mathsf{msg}.\mathsf{payload}$ as a tuple $(\mathsf{sid}, a^{\perp}, a^{\top}, b, c)$ where:
    • $\mathsf{sid}$ is a session identifier;
    • $a^{\perp}, a^{\top}, b, c$ are positive integers.
 7. Check that $\mathsf{sid} = \mathsf{sid}_1 = \mathsf{sid}_2$.
 8. Check that $a^{\top}_1 < a^{\perp}_2, a^{\perp} = a^{\perp}_1, a^{\top} = a^{\top}_2$.
 9. Check that $b = b_1 + b_2$.
10. Check that $c = c_1 + c_2$.
```

```
$\Pi^{\mathsf{Reduce}}_{\mathrm{sum}}(\mathsf{msg}, \mathsf{loc}, \vec{\mathsf{msg}}_{\mathsf{in}})$
 1. Let $d$ be the number of input messages in $\vec{\mathsf{msg}}_{\mathsf{in}}$.
 2. Check that $d = 2$.
 3. Check that each $\vec{\mathsf{msg}}_{\mathsf{in}}[j].\mathsf{type}$ lies in $\{\Pi^{\mathsf{Reduce}}_{\mathrm{fmt}}.\mathsf{type}, \Pi^{\mathsf{Reduce}}_{\mathrm{sum}}.\mathsf{type}\}$.
 4. Parse each $\vec{\mathsf{msg}}_{\mathsf{in}}[j].\mathsf{payload}$ as a tuple $(\mathsf{sid}_j, a^{\perp}_j, a^{\top}_j, b_j, c_j)$ where:
    • $\mathsf{sid}_j$ is a session identifier;
    • $a^{\perp}_j, a^{\top}_j, b_j, c_j$ are positive integers.
 5. The local data $\mathsf{loc}$ is not used; ignore it.
 6. Parse $\mathsf{msg}.\mathsf{payload}$ as a tuple $(\mathsf{sid}, a^{\perp}, a^{\top}, b, c)$ where:
    • $\mathsf{sid}$ is a session identifier;
    • $a^{\perp}, a^{\top}, b, c$ are positive integers.
 7. Check that $\mathsf{sid} = \mathsf{sid}_1 = \mathsf{sid}_2$.
 8. Check that $a^{\top}_1 < a^{\perp}_2, a^{\perp} = a^{\perp}_1, a^{\top} = a^{\top}_2$.
 9. Check that $b = b_1 + b_2$.
10. Check that $c = c_1 + c_2$.
```

Figure 18: Summary of the construction of $\Pi^{\mathsf{Map}}_{\mathrm{sum}}$ and $\Pi^{\mathsf{Map}}_{\mathrm{sum}}$.

```
$\Pi_{\mathrm{fin}}(\mathsf{msg}, \mathsf{loc}, \vec{\mathsf{msg}}_{\mathsf{in}})$
 1. Let $d$ be the number of input messages in $\vec{\mathsf{msg}}_{\mathsf{in}}$.
 2. Check that $d = 2$.
 3. Check that $\vec{\mathsf{msg}}_{\mathsf{in}}[1].\mathsf{type} \in \{\Pi^{\mathsf{Map}}_{\mathrm{fmt}}.\mathsf{type}, \Pi^{\mathsf{Map}}_{\mathrm{sum}}.\mathsf{type}\}$.
 4. Check that $\vec{\mathsf{msg}}_{\mathsf{in}}[2].\mathsf{type} \in \{\Pi^{\mathsf{Reduce}}_{\mathrm{fmt}}.\mathsf{type}, \Pi^{\mathsf{Reduce}}_{\mathrm{sum}}.\mathsf{type}\}$.
 5. Parse each $\vec{\mathsf{msg}}_{\mathsf{in}}[j].\mathsf{payload}$ as a tuple $(\mathsf{sid}_j, a^{\perp}_j, a^{\top}_j, b_j, c_j)$ where:
    • $\mathsf{sid}_j$ is a session identifier;
    • $a^{\perp}_j, a^{\top}_j, b_j, c_j$ are positive integers.
 6. Parse the local data $\mathsf{loc}$ as a tuple $(\mathsf{cm}, p, \mathsf{cm}_z, M, \mathsf{cr}_M)$ where:
    • $\mathsf{cm}$ is a commitment (for the scheme $\mathsf{COMM}$);
    • $p$ are MapReduce parameters;
    • $\mathsf{cm}_z$ is a commitment (for the scheme $\mathsf{COMM}^*$);
    • $M$ is a positive integer;
    • $\mathsf{cr}_M$ is randomness (for the scheme $\mathsf{COMM}^*$).
 7. Parse $\mathsf{msg}.\mathsf{payload}$ as a tuple $(\mathsf{sid}, R)$ where:
    • $\mathsf{sid}$ is a session identifier;
    • $R$ is a positive integer.
 8. Check that $\mathsf{sid} = \mathsf{sid}_1 = \mathsf{sid}_2$.
 9. Check that $\mathsf{sid} = H(\mathsf{cm} \| p \| \mathsf{cm}_z)$.
10. Parse $\mathsf{cm}$ as a pair $(\mathsf{cm}_{\mathsf{rt}}, \mathsf{cm}_M)$ where both components are commitments
    for the scheme $\mathsf{COMM}^*$.
11. Check that $\mathsf{COMM}^*.\mathsf{Ver}(M, \mathsf{cm}_M, \mathsf{cr}_M) = 1$.
12. Check that $M = b_1$.
13. Check that $R = b_2$.
14. Check that $c_1 = c_2$.
```

Figure 19: Summary of the construction of Π_{fin}.

E More details for the construction of $\mathsf{MR.Prove}$

The construction of $\mathsf{MR.Prove}$ in Section 5.2 uses the compliance engineering result of Section 5.1 as a black box. For additional intuition, we sketch here the construction of $\mathsf{MR.Prove}$ in terms of the compliance predicates constructed in Theorem 5.2's proof; as in Section 5.1, we focus, for simplicity, on the (artificial) case where each mapper outputs a single phase-2 record. As in the proof, $\vec{\Pi}^{\mathsf{MR}}$ equals the vector $(\Pi^{\mathsf{Map}}_{\mathrm{exe}}, \Pi^{\mathsf{Reduce}}_{\mathrm{exe}}, \Pi^{\mathsf{Map}}_{\mathrm{fmt}}, \Pi^{\mathsf{Reduce}}_{\mathrm{fmt}}, \Pi^{\mathsf{Map}}_{\mathrm{sum}}, \Pi^{\mathsf{Reduce}}_{\mathrm{sum}}, \Pi_{\mathrm{fin}})$.

On input a proving key pk, an instance $(\mathsf{cm}, \mathbb{y})$, and a witness $(\mathbb{x}, \mathsf{cr})$, the prover $\mathsf{MR.Prove}$ computes a non-interactive proof π_{MR} for the statement "I know $(\mathbb{x}, \mathsf{cr})$ such that $\big((\mathsf{cm}, \mathbb{y}), (\mathbb{x}, \mathsf{cr})\big) \in \mathscr{R}^{\mathsf{COMM}}_{(\mathsf{Map},\mathsf{Reduce})}$" as follows.

1. **Initialization.**
   - Parse pk as $(\mathsf{Map}, \mathsf{Reduce}, \mathsf{pk}_{\mathsf{pcd}})$, where $(\mathsf{Map}, \mathsf{Reduce})$ is a MapReduce pair and $\mathsf{pk}_{\mathsf{pcd}}$ a PCD proving key.
   - Set rt to be the root of a Merkle tree on $\mathbb{x}$: $\mathsf{rt} := \mathsf{MERKLE.GetRoot}(\mathbb{x})$. Let ap_i be the authentication path for $\mathbb{x}_i$.
   - Set M to be the number of records in the input $\mathbb{x}$: $M := \mathsf{len}(\mathbb{x})$.
   - Parse cm as $(\mathsf{cr}_{\mathsf{rt}}, \mathsf{cr}_M)$, where $\mathsf{cr}_{\mathsf{rt}}$ and cr_M are commitments for the scheme COMM^*.
   - Allocate six empty lists: $\mathcal{L}_{\mathsf{Map}}$, $\mathcal{L}_{\mathsf{Shuffle}}$, $\mathcal{L}_{\mathsf{Reduce}}$, $\mathcal{L}_{\mathsf{proof}}$, $\mathcal{L}_1$, and $\mathcal{L}_2$.
2. **Prove correctness of execution of all mapper nodes.**
   For each $i \in \{1, \dots, M\}$, prove that Map executes correctly when given the i-th phase-1 record $\mathbb{x}_i = (k^1, v^1)$, as follows.
   - Compute Map's outputs: $(k^2, v^2) = \mathsf{Map}(k^1, v^1)$.
   - Construct inputs for the predicate $\Pi^{\mathsf{Map}}_{\mathrm{exe}}$ (see Figure 8):
     – The (single) input message is $\mathsf{msg}_{\mathsf{in}} \begin{cases} .\mathsf{type} := 0 \\ .\mathsf{payload} := (\mathsf{cm}, i, k^1, v^1) \end{cases}$.
     – The output message is $\mathsf{msg}_{\mathsf{out}} \begin{cases} .\mathsf{type} := \Pi^{\mathsf{Map}}_{\mathrm{exe}}.\mathsf{type} \\ .\mathsf{payload} := (\mathsf{cm}, i, k^2, v^2) \end{cases}$.
     – The local data loc is $(\mathsf{rt}, M, \mathsf{cr}_{\mathsf{rt}}, \mathsf{cr}_M, \mathsf{ap}_i)$.
   - Set $\pi_{\mathsf{in}} := \bot$. (The message $\mathsf{msg}_{\mathsf{in}}$ has no associated proof because it has type 0.)
   - Use the PCD prover $\mathbb{P}$ to prove that $\mathsf{msg}_{\mathsf{out}}$ is $\vec{\Pi}^{\mathsf{MR}}$-compliant: $\pi := \mathbb{P}(\mathsf{pk}, \mathsf{msg}_{\mathsf{out}}, \mathsf{loc}, (\mathsf{msg}_{\mathsf{in}}), (\pi_{\mathsf{in}}))$.
   - Append $(\mathsf{msg}_{\mathsf{out}}, \pi)$ to $\mathcal{L}_{\mathsf{Map}}$.
   - Append (i, k^2, v^2, π) to $\mathcal{L}_{\mathsf{Shuffle}}$.
3. **Prove correctness of execution of all reducer nodes.**
   For each unique phase-2 key k^2 in $\mathcal{L}_{\mathsf{Shuffle}}$, prove that Reduce executes correctly, as follows.
   - Let $(i_j, k^2, v^2_j, \pi_j)_{j=1}^{d_{\mathsf{in}}}$ be the list of tuples in $\mathcal{L}_{\mathsf{Shuffle}}$ that share the same key k^2.
   - Compute Reduce's outputs: $(k^3, v^3) = \mathsf{Reduce}(k^2, (v^2_j)_{j=1}^{d_{\mathsf{in}}})$.
   - Generate a commitment to k^2: $(\mathsf{cm}_{k^2}, \mathsf{cr}_{k^2}) := \mathsf{COMM}^*.\mathsf{Gen}(k^2)$.
   - Generate a commitment to d_{in}: $(\mathsf{cm}_{d_{\mathsf{in}}}, \mathsf{cr}_{d_{\mathsf{in}}}) := \mathsf{COMM}^*.\mathsf{Gen}(d_{\mathsf{in}})$.
   - Construct inputs for the predicate $\Pi^{\mathsf{Reduce}}_{\mathrm{exe}}$ (see Figure 8):
     – The input messages are $\vec{\mathsf{msg}}_{\mathsf{in}}$ where, for each $j \in \{1, \dots d_{\mathsf{in}}\}$, $\vec{\mathsf{msg}}_{\mathsf{in}}[j] \begin{cases} .\mathsf{type} := \Pi^{\mathsf{Map}}_{\mathrm{exe}}.\mathsf{type} \\ .\mathsf{payload} := (\mathsf{cm}, i_j, k^2_j, v^2_j) \end{cases}$.
     – The output message is $\mathsf{msg}_{\mathsf{out}} \begin{cases} .\mathsf{type} := \Pi^{\mathsf{Reduce}}_{\mathrm{exe}}.\mathsf{type} \\ .\mathsf{payload} := (\mathsf{cm}, k^3, v^3, \mathsf{cm}_{k^2}, \mathsf{cm}_{d_{\mathsf{in}}}) \end{cases}$.
     – The local data loc is $(\mathsf{cr}_{k^2}, \mathsf{cr}_{d_{\mathsf{in}}})$.
   - Collect the proofs for the input messages: $\vec{\pi}_{\mathsf{in}} := (\pi_j)_{j=1}^{d_{\mathsf{in}}}$.
   - Use the PCD prover $\mathbb{P}$ to prove that $\mathsf{msg}_{\mathsf{out}}$ is $\vec{\Pi}^{\mathsf{MR}}$-compliant: $\pi := \mathbb{P}(\mathsf{pk}, \mathsf{msg}_{\mathsf{out}}, \mathsf{loc}, \vec{\mathsf{msg}}_{\mathsf{in}}, \vec{\pi}_{\mathsf{in}})$.
   - Append $(\mathsf{msg}_{\mathsf{out}}, \pi)$ to $\mathcal{L}_{\mathsf{Reduce}}$.
   - Append π to $\mathcal{L}_{\mathsf{proof}}$, cm_{k^2} to $\mathcal{L}_1$, and $\mathsf{cm}_{d_{\mathsf{in}}}$ to $\mathcal{L}_2$.
4. **Prove correctness of aggregation of mapper nodes' outputs.**
   The messages in $\mathcal{L}_{\mathsf{Map}}$ are pairwise aggregated via a binary tree with these messages as leaves. The aggregation consists of two parts: reformatting the leaves through $\Pi^{\mathsf{Map}}_{\mathrm{fmt}}$, and then pairwise aggregating internal nodes, one layer at a time, through $\Pi^{\mathsf{Map}}_{\mathrm{sum}}$.

   Part 1: reformat leaves via $\Pi^{\mathsf{Map}}_{\mathrm{fmt}}$. Recall that $\mathcal{L}_{\mathsf{Map}} = \big((\mathsf{msg}_i, \pi_i)\big)_{i=1}^{M}$ and each message msg_i has payload $(\mathsf{cm}, i, k^2_i, v^2_i)$. Initialize $\mathcal{L}_{\mathsf{next\text{-}layer}}$ to be an empty list, and perform the following steps for each $i \in \{1, \dots, M\}$.
   - Construct inputs for the predicate $\Pi^{\mathsf{Map}}_{\mathrm{fmt}}$ (see Figure 11):
     – The input messages are $\vec{\mathsf{msg}}_{\mathsf{in}} := (\mathsf{msg}_i)$.

  - The output message is $\mathsf{msg}_{\mathsf{out}} \begin{cases} .\mathsf{type} := \Pi^{\mathsf{Map}}_{\mathrm{fmt}}.\mathsf{type} \\ .\mathsf{payload} := (\mathsf{cm}, i, i, 1, 1) \end{cases}$.
  - The local data loc is empty (it equals $\perp$).
- Collect the proofs for the input messages: $\vec{\pi}_{\mathsf{in}} := (\pi_i)$.
- Use the PCD prover $\mathbb{P}$ to prove that $\mathsf{msg}_{\mathsf{out}}$ is $\vec{\Pi}^{\mathsf{MR}}$-compliant: $\pi := \mathbb{P}(\mathsf{pk}, \mathsf{msg}_{\mathsf{out}}, \mathsf{loc}, \vec{\mathsf{msg}}_{\mathsf{in}}, \vec{\pi}_{\mathsf{in}})$.
- Append $(\mathsf{msg}_{\mathsf{out}}, \pi)$ to $\mathcal{L}_{\mathsf{next\text{-}layer}}$.

Part 2: aggregate internal nodes via $\Pi^{\mathsf{Map}}_{\mathrm{sum}}$. While $\mathcal{L}_{\mathsf{next\text{-}layer}}$ contains more than one pair, process adjacent pairs in $\mathcal{L}_{\mathsf{next\text{-}layer}}$ as follows. Set $\mathcal{L}_{\mathsf{cur\text{-}layer}} := \mathcal{L}_{\mathsf{next\text{-}layer}}$ and clear $\mathcal{L}_{\mathsf{next\text{-}layer}}$; recall that the i-th message msg_i in $\mathcal{L}_{\mathsf{cur\text{-}layer}}$ has payload $(\mathsf{cm}, a_i^{\perp}, a_i^{\top}, b_i, c_i)$. For each $i \in \{1, \dots, \lceil \mathsf{len}(\mathcal{L}_{\mathsf{cur\text{-}layer}})/2 \rceil\}$, execute the following.
- If $2i \le \mathsf{len}(\mathcal{L}_{\mathsf{cur\text{-}layer}})$, process the $(2i-1)$-th and $(2i)$-th message-proof pairs in $\mathcal{L}_{\mathsf{cur\text{-}layer}}$.
  - Construct inputs for the predicate $\Pi^{\mathsf{Map}}_{\mathrm{sum}}$ (see Figure 12):
    - The input messages are $\vec{\mathsf{msg}}_{\mathsf{in}} := (\mathsf{msg}_{2i-1}, \mathsf{msg}_{2i})$.
    - The output message is $\mathsf{msg}_{\mathsf{out}} \begin{cases} .\mathsf{type} := \Pi^{\mathsf{Map}}_{\mathrm{sum}}.\mathsf{type} \\ .\mathsf{payload} := (\mathsf{cm}, a^{\perp}_{2i-1}, a^{\top}_{2i}, b_{2i-1} + b_{2i}, c_{2i-1} + c_{2i}) \end{cases}$.
    - The local data loc is empty (it equals $\perp$).
  - Collect the proofs for the input messages: $\vec{\pi}_{\mathsf{in}} := (\pi_{2i-1}, \pi_{2i})$.
  - Use the PCD prover $\mathbb{P}$ to prove that $\mathsf{msg}_{\mathsf{out}}$ is $\vec{\Pi}^{\mathsf{MR}}$-compliant: $\pi := \mathbb{P}(\mathsf{pk}, \mathsf{msg}_{\mathsf{out}}, \mathsf{loc}, \vec{\mathsf{msg}}_{\mathsf{in}}, \vec{\pi}_{\mathsf{in}})$.
- If $2i > \mathsf{len}(\mathcal{L}_{\mathsf{cur\text{-}layer}})$, forward the leftover $(2i-1)$-th pair in $\mathcal{L}_{\mathsf{cur\text{-}layer}}$, by setting $(\mathsf{msg}_{\mathsf{out}}, \pi) := (\mathsf{msg}_{2i-1}, \pi_{2i-1})$.
- Add $(\mathsf{msg}_{\mathsf{out}}, \pi)$ to $\mathcal{L}_{\mathsf{next\text{-}layer}}$.

When the above loop terminates, there is a single message-proof pair in $\mathcal{L}_{\mathsf{next\text{-}layer}}$; denote it by $(\mathsf{msg}^{\mathsf{Map}}_{\mathsf{sum}}, \pi^{\mathsf{Map}}_{\mathsf{sum}})$.

5. **Prove correctness of aggregation of mapper nodes' outputs.**
The messages in $\mathcal{L}_{\mathsf{Reduce}}$ are pairwise aggregated via a binary tree with these messages as leaves. Similarly to Step 4, the aggregation consists of two parts: reformatting the leaves through $\Pi^{\mathsf{Reduce}}_{\mathrm{fmt}}$, and then pairwise aggregating internal nodes, one layer at a time, through $\Pi^{\mathsf{Reduce}}_{\mathrm{sum}}$. There are only small differences from Step 4: leaf messages contain the commitments cm_{k^2} and $\mathsf{cm}_{d_{\mathsf{in}}}$ (and $\Pi^{\mathsf{Reduce}}_{\mathrm{fmt}}$ is responsible for ensuring the correctness of their decommitted values), and internal messages maintain not one but two partial sums (counting the number of distinct keys k^2, as well as the sum of all d_{in}). Overall, the computation of the final message-proof pair $(\mathsf{msg}^{\mathsf{Reduce}}_{\mathsf{sum}}, \pi^{\mathsf{Reduce}}_{\mathsf{sum}})$ is quite similar to Step 4, and thus we do not spell out the details.
6. **Prove that the results of the two aggregations are consistent.**
Prove that $\mathsf{msg}^{\mathsf{Map}}_{\mathsf{sum}}$ (aggregating messages in $\mathcal{L}_{\mathsf{Map}}$) and $\mathsf{msg}^{\mathsf{Reduce}}_{\mathsf{sum}}$ (aggregating messages in $\mathcal{L}_{\mathsf{Reduce}}$) are consistent. Consistency of these two implies that all mappers' outputs have been correctly shuffled and then processed by reducers.
   - Construct inputs for the predicate Π_{fin} (see Figure 13).
     - The input messages are $\vec{\mathsf{msg}}_{\mathsf{in}} := (\mathsf{msg}^{\mathsf{Map}}_{\mathsf{sum}}, \mathsf{msg}^{\mathsf{Reduce}}_{\mathsf{sum}})$.
     - The output message is $\mathsf{msg}_{\mathsf{out}} \begin{cases} .\mathsf{type} := \Pi_{\mathrm{fin}}.\mathsf{type} \\ .\mathsf{payload} := (\mathsf{cm}, \mathsf{len}(\mathbb{y})) \end{cases}$.
     - The local data loc is (M, cr_M).
   - Collect the proofs for the input messages: $\vec{\pi}_{\mathsf{in}} := (\pi^{\mathsf{Map}}_{\mathsf{sum}}, \pi^{\mathsf{Reduce}}_{\mathsf{sum}})$.
   - Use $\mathbb{P}$ to prove that $\mathsf{msg}_{\mathsf{out}}$ is $\vec{\Pi}^{\mathsf{MR}}$-compliant: $\pi_{\mathsf{fin}} := \mathbb{P}(\mathsf{pk}, \mathsf{msg}_{\mathsf{out}}, \mathsf{loc}, \vec{\mathsf{msg}}_{\mathsf{in}}, \vec{\pi}_{\mathsf{in}})$.
7. **Construct the final proof.**
   - Set $\pi_{\mathsf{MR}} := (\mathcal{L}_{\mathsf{proof}}, \mathcal{L}_1, \mathcal{L}_2, \pi_{\mathsf{fin}})$ and output π_{MR}.

Note that $\mathsf{MR.Prove}$ consist of: (i) a MapReduce computation of complexity similar to the MapReduce computation being proved (handling Step 2 and Step 3); (ii) a small number of cheap MapReduce computations to prove the correctness of aggregation for mapper and reducer nodes' outputs (handling Step 4 and Step 5); and (iii) a simple local computation (handling Step 6). Overall, one can see that $\mathsf{MR.Prove}$ is indeed $(\mathsf{Map}, \mathsf{Reduce})$-complexity-preserving.

Remark E.1. As described above, $\mathsf{MR.Prove}$ proves compliance over a (distributed-computation) transcript T having the following output messages $\mathrm{OUTS}(\mathsf{T})$:

$$\mathsf{msg}_0 \begin{cases} .\mathsf{type} := \Pi_{\mathrm{fin}}.\mathsf{type} \\ .\mathsf{payload} := (\mathsf{cm}, \mathsf{len}(\mathbb{y})) \end{cases} \quad \text{and, for each } i \in \{1, \dots, \mathsf{len}(\mathbb{y})\},\ \mathsf{msg}_i \begin{cases} .\mathsf{type} := \Pi^{\mathsf{Reduce}}_{\mathrm{exe}}.\mathsf{type} \\ .\mathsf{payload} := (\mathsf{cm}, \mathbb{y}_i, \mathcal{L}_{1,i}, \mathcal{L}_{2,i}) \end{cases}.$$

These output messages are slightly different than what required by $(\mathsf{cm}, \mathbb{y})$-compatibility, because each msg_i additionally contains $\mathcal{L}_{1,i}$ and $\mathcal{L}_{2,i}$ (cf. Definition 5.1). Because both are commitments, the difference does not affect security; we use these merely for improved efficiency (and could eliminate them if needed, by introducing $\mathsf{len}(\mathbb{y})$ new nodes to the distributed computation, each responsible for dropping the extra terms from a message).

References

[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.

[BBFR15] Michael Backes, Manuel Barbosa, Dario Fiore, and Raphael M. Reischuk. ADSNARK: nearly practical and privacy-preserving proofs on authenticated data. In *Proceedings of the 36th IEEE Symposium on Security and Privacy*, S&P '15, pages ???–???, 2015.

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

[BCCG$^+$14] Nir Bitansky, Ran Canetti, Alessandro Chiesa, Shafi Goldwasser, Huijia Lin, Aviad Rubinstein, and Eran Tromer. The hunting of the SNARK. ePrint 2014/580, 2014.

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

[BCCT13] Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. Recursive composition and bootstrapping for SNARKs and proof-carrying data. In *Proceedings of the 45th ACM Symposium on the Theory of Computing*, STOC '13, pages 111–120, 2013.

[BCGG$^+$14] Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from Bitcoin. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, pages 459–474, 2014.

[BCGTV13] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. SNARKs for C: Verifying program executions succinctly and in zero knowledge. In *Proceedings of the 33rd Annual International Cryptology Conference*, CRYPTO '13, pages 90–108, 2013.

[BCIOP13] Nir Bitansky, Alessandro Chiesa, Yuval Ishai, Rafail Ostrovsky, and Omer Paneth. Succinct non-interactive arguments via linear interactive proofs. In *Proceedings of the 10th Theory of Cryptography Conference*, TCC '13, pages 315–333, 2013.

[BCTV14a] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Scalable zero knowledge via cycles of elliptic curves. In *Proceedings of the 34th Annual International Cryptology Conference*, CRYPTO '14, pages 276–294, 2014. Extended version at `http://eprint.iacr.org/2014/595`.

[BCTV14b] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Succinct non-interactive zero knowledge for a von Neumann architecture. In *Proceedings of the 23rd USENIX Security Symposium*, USENIX Security '14, pages 781–796, 2014. Extended version at `http://eprint.iacr.org/2013/879`.

[BDSMP91] Manuel Blum, Alfredo De Santis, Silvio Micali, and Giuseppe Persiano. Non-interactive zero-knowledge. *SIAM Journal on Computing*, 20(6):1084–1118, 1991.

[BFM88] Manuel Blum, Paul Feldman, and Silvio Micali. Non-interactive zero-knowledge and its applications. In *Proceedings of the 20th Annual ACM Symposium on Theory of Computing*, STOC '88, pages 103–112, 1988.

[BFRS$^+$13] Benjamin Braun, Ariel J. Feldman, Zuocheng Ren, Srinath Setty, Andrew J. Blumberg, and Michael Walfish. Verifying computations with state. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles*, SOSP '13, pages 341–357, 2013.

[BG93] Mihir Bellare and Oded Goldreich. On defining proofs of knowledge. In *Proceedings of the 12th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '92, pages 390–420, 1993.

[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.

[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.

[BPXOD07] Thorsten Brants, Ashok C. Popat, Peng Xu, Franz Josef Och, and Jeffrey Dean. Large language models in machine translation. In *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*, EMNLP-CoNLL '07, pages 858–867, 2007.

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

[BTVW14] Andrew J. Blumberg, Justin Thaler, Victor Vu, and Michael Walfish. Verifiable computation using multiple provers. Cryptology ePrint Archive, Report 2014/846, 2014.

[CFHK$^+$15] Craig Costello, Cédric Fournet, Jon Howell, Markulf Kohlweiss, Benjamin Kreuter, Michael Naehrig, Bryan Parno, and Samee Zahur. Geppetto: Versatile verifiable computation. In *Proceedings of the 36th IEEE Symposium on Security and Privacy*, S&P '15, pages ???–???, 2015.

[CKLM13] Melissa Chase, Markulf Kohlweiss, Anna Lysyanskaya, and Sarah Meiklejohn. Succinct malleable NIZKs and an application to compact shuffles. In *Proceedings of the 10th Theory of Cryptography Conference*, TCC '13, pages 100–119, 2013.

[CKLY$^+$06] Cheng-Tao Chu, Sang Kyun Kim, Yi-An Lin, YuanYuan Yu, Gary R. Bradski, Andrew Y. Ng, and Kunle Olukotun. MapReduce for machine learning on multicore. In *Proceedings of the 20th Annual Conference on Neural Information Processing Systems*, NIPS '04, pages 281–288, 2006.

[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.

[CMT12] Graham Cormode, Michael Mitzenmacher, and Justin Thaler. Practical verified computation with streaming interactive proofs. In *Proceedings of the 4th Symposium on Innovations in Theoretical Computer Science*, ITCS '12, pages 90–112, 2012.

[CRR12] Ran Canetti, Ben Riva, and Guy N. Rothblum. Two protocols for delegation of computation. In *Proceedings of the 6th International Conference on Information Theoretic Security*, ICITS 12, pages 37–61, 2012.

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

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

[CTV13] Stephen Chong, Eran Tromer, and Jeffrey A. Vaughan. Enforcing language semantics using proof-carrying data. Cryptology ePrint Archive, Report 2013/513, 2013.

[CTY11] Graham Cormode, Justin Thaler, and Ke Yi. Verifying computations with streaming interactive proofs. *Proceedings of the VLDB Endowment*, 5(1):25–36, 2011.

[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.

[DCMC$^+$12] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V. Le, Mark Z. Mao, Marc'Aurelio Ranzato, Andrew W. Senior, Paul A. Tucker, Ke Yang, and Andrew Y. Ng. Large scale distributed deep networks. In *Proceedings of the 26th Annual Conference on Neural Information Processing Systems*, NIPS '12, pages 1232–1240, 2012.

[DCML08] Christopher Dyer, Aaron Cordova, Alex Mont, and Jimmy Lin. Fast, easy, and cheap: Construction of statistical machine translation models with MapReduce. In *Proceedings of the 3rd Workshop on Statistical Machine Translation*, StatMT '08, pages 199–207, 2008.

[DFGK14] George Danezis, Cédric Fournet, Jens Groth, and Markulf Kohlweiss. Square span programs with applications to succinct NIZK arguments. In *Proceedings of the 20th International Conference on the Theory and Application of Cryptology and Information Security*, ASIACRYPT '14, pages 532–550, 2014.

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

[DFKP13] George Danezis, Cédric Fournet, Markulf Kohlweiss, and Bryan Parno. Pinocchio Coin: building Zerocoin from a succinct pairing-based proof system. In *Proceedings of the 2013 Workshop on Language Support for Privacy Enhancing Technologies*, PETShop '13, 2013. URL: `http://www0.cs.ucl.ac.uk/staff/G.Danezis/papers/DanezisFournetKohlweissParno13.pdf`.

[DG04] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proceedings of the 6th Symposium on Operating System Design and Implementation*, OSDI '04, pages 137–149, 2004.

[DH76] W. Diffie and M.E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, Nov 1976.

[DL08] 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.

[FG12] Dario Fiore and Rosario Gennaro. Publicly verifiable delegation of large polynomials and matrix computations, with applications. Cryptology ePrint Archive, Report 2012/281, 2012.

[FL14] Matthew Fredrikson and Benjamin Livshits. Zø: An optimizing distributing zero-knowledge compiler. In *Proceedings of the 23rd USENIX Security Symposium*, USENIX Security '14, pages 909–924, 2014.

[FLZ13] Prastudy Fauzi, Helger Lipmaa, and Bingsheng Zhang. Efficient modular NIZK arguments from shift and product. In *Proceedings of the 12th International Conference on Cryptology and Network Security*, CANS '13, pages 92–121, 2013.

[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.

[GGPR13] Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. Quadratic span programs and succinct NIZKs without PCPs. In *Proceedings of the 32nd Annual International Conference on Theory and Application of Cryptographic Techniques*, EUROCRYPT '13, pages 626–645, 2013.

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

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

[GM12] Ashish Goel and Kamesh Munagala. Complexity measures for Map-Reduce, and comparison to parallel computing. ArXiv abs/1211.6526, 2012.

[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.

[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.

[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.

[Had] Apache Hadoop. URL: `http://hadoop.apache.org/`.

[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.

[KCF12] U. Kang, Duen Horng Chau, and Christos Faloutsos. Pegasus: Mining billion-scale graphs in the cloud. In *Proceedings of the 2012 IEEE International Conference on Acoustics, Speech and Signal Processing*, ICASSP '12, pages 5341–5344, 2012.

[KPPS$^+$14] Ahmed E. Kosba, Dimitrios Papadopoulos, Charalampos Papamanthou, Mahmoud F. Sayed, Elaine Shi, and Nikos Triandopoulos. TRUESET: Faster verifiable set computations. In *Proceedings of the 23rd USENIX Security Symposium*, USENIX Security '14, pages 765–780, 2014.

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

[LD10] Jimmy Lin and Chris Dyer. *Data-Intensive Text Processing with MapReduce*. Morgan and Claypool Publishers, 2010.

[LGKB$^+$12] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. Distributed graphlab: a framework for machine learning in the cloud. *Proceedings of the VLDB Endowment*, 5(8):716–727, 2012.

[Lin09] Jimmy Lin. Brute force and indexed approaches to pairwise document similarity comparisons with mapreduce. In *Proceedings of the 32nd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '09, pages 155–162, 2009.

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

[Lip13] Helger Lipmaa. Succinct non-interactive zero knowledge arguments from span programs and linear error-correcting codes. In *Proceedings of the 19th International Conference on the Theory and Application of Cryptology and Information Security*, ASIACRYPT '13, pages 41–60, 2013.

[Lip14] Helger Lipmaa. Efficient NIZK arguments via parallel verification of Beneš networks. In *Proceedings of the 9th International Conference on Security and Cryptography for Networks*, SCN '14, pages 416–434, 2014.

[LN97] Rudolf Lidl and Harald Niederreiter. *Finite Fields*. Cambridge University Press, second edition edition, 1997.

[LS10] Jimmy Lin and Michael C. Schatz. Design patterns for efficient graph algorithms in mapreduce. In *Proceedings of the 8th Workshop on Mining and Learning with Graphs*, MLG '10, pages 78–85, 2010.

[LSLPS09] Ben Langmead, Michael C. Schatz, Jimmy Lin, Mihai Pop, and StevenL Salzberg. Searching for SNPs with cloud computing. *Genome Biology*, 10(11), 2009.

[Mer89] Ralph C. Merkle. A certified digital signature. In *Proceedings of the 9th Annual International Cryptology Conference*, CRYPTO '89, pages 218–238, 1989.

[MGGR13] Ian Miers, Christina Garman, Matthew Green, and Aviel D. Rubin. Zerocoin: Anonymous distributed e-cash from bitcoin. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, pages 397–411, 2013.

[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.

[NY90] Moni Naor and Moti Yung. Public-key cryptosystems provably secure against chosen ciphertext attacks. In *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing*, STOC '90, pages 427–437, 1990.

[PGHR13] Brian Parno, Craig Gentry, Jon Howell, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. In *Proceedings of the 34th IEEE Symposium on Security and Privacy*, S&P '13, pages 238–252, 2013.

[PHBB09] Biswanath Panda, Joshua Herbach, Sugato Basu, and Roberto J. Bayardo. PLANET: massively parallel learning of tree ensembles with MapReduce. *Proceedings of the VLDB Endowment*, 2(2):1426–1437, 2009.

[PR14] Omer Paneth and Guy N. Rothblum. Publicly verifiable non-interactive arguments for delegating computation. Cryptology ePrint Archive, Report 2014/981, 2014.

[PWB12] Juan Pino, Aurelien Waite, and William Byrne. Simple and efficient model filtering in statistical machine translation. *Prague Bulletin of Mathematical Linguistics*, 98:5–24, 2012.

[SBVB$^+$13] Srinath Setty, Benjamin Braun, Victor Vu, Andrew J. Blumberg, Bryan Parno, and Michael Walfish. Resolving the conflict between generality and plausibility in verified computation. In *Proceedings of the 8th EuoroSys Conference*, EuroSys '13, pages 71–84, 2013.

[SBW11] Srinath Setty, Andrew J. Blumberg, and Michael Walfish. Toward practical and unconditional verification of remote computations. In *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems*, HotOS '11, pages 29–29, 2011.

[SCFG$^+$15] Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. VC3: Trustworthy data analytics in the cloud. In *Proceedings of the 36th IEEE Symposium on Security and Privacy*, S&P '15, pages ???–???, 2015.

[Sch09] Michael C. Schatz. CloudBurst: highly sensitive read mapping with MapReduce. *Bioinformatics*, 25(11):1363–1369, 2009.

[SCI] SCIPR Lab. libsnark: a C++ library for zkSNARK proofs. URL: `https://github.com/scipr-lab/libsnark`.

[SMBW12] Srinath Setty, Michael McPherson, Andrew J. Blumberg, and Michael Walfish. Making argument systems for outsourced computation practical (sometimes). In *Proceedings of the 19th Network and Distributed System Security Symposium*, NDSS '12, 2012.

[SVPB$^+$12] Srinath Setty, Victor Vu, Nikhil Panpalia, Benjamin Braun, Andrew J. Blumberg, and Michael Walfish. Taking proof-based verified computation a few steps closer to practicality. In *Proceedings of the 21st USENIX Security Symposium*, USENIX Security '12, pages 253–268, 2012.

[Tha13] Justin Thaler. Time-optimal interactive proofs for circuit evaluation. In *Proceedings of the 33rd Annual International Cryptology Conference*, CRYPTO '13, pages 71–89, 2013.

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

[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.

[VSBW13] Victor Vu, Srinath Setty, Andrew J. Blumberg, and Michael Walfish. A hybrid architecture for interactive verifiable computation. In *Proceedings of the 34th IEEE Symposium on Security and Privacy*, S¶ '13, pages 223–237, 2013.

[WHK08] Jason Wolfe, Aria Haghighi, and Dan Klein. Fully distributed EM for very large datasets. In *Proceedings of the 25th International Conference on Machine Learning*, ICML '08, pages 1184–1191, 2008.

[WSRBW15] Riad S. Wahby, Srinath Setty, Zuocheng Ren, Andrew J. Blumberg, and Michael Walfish. Efficient RAM and control flow in verifiable outsourced computation. In *Proceedings of the 22nd Network and Distributed System Security Symposium*, NDSS '15, 2015.

[ZCDD$^+$12] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '12, pages 15–28, 2012.

[ZCFSS10] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2nd USENIX Workshop on Hot Topics in Cloud Computing*, HotCloud '10, 2010.

[ZPK14] Yupeng Zhang, Charalampos Papamanthou, and Jonathan Katz. Alitheia: Towards practical verifiable graph processing. In *Proceedings of the 21st ACM Conference on Computer and Communications Security*, CCS '14, pages 856–867, 2014.