Cluster Computing in Zero Knowledge

Alessandro Chiesa[1(✉)], Eran Tromer[3], and Madars Virza[2]

[1] ETH Zurich, Zürich, Switzerland
alessandro.chiesa@inf.ethz.ch
[2] MIT, Cambridge, USA
madars@csail.mit.edu
[3] Tel Aviv University, Tel Aviv, Israel
tromer@cs.tau.ac.il

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: Computationally-sound proofs · Proof-carrying data · Zero knowledge · Cluster computing · MapReduce

E. Oswald and M. Fischlin (Eds.): EUROCRYPT 2015, Part II, LNCS 9057, pp. 371–403, 2015.
DOI: 10.1007/978-3-662-46803-6_13

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* [48]. 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

[1] Sometimes a property stronger than soundness is required: *proof of knowledge* [4,48], 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.

the final result 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. In sum: To what extent can one efficiently perform cluster computing in zero knowledge?

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* [35]. We review MapReduce later (in Section 2), and now only say that MapReduce can express many useful computations, including ones used for machine learning [26,67,82], graph mining and processing [52,58], statistical machine translation [20,38,57,70], document similarity [56], and bioinformatics [54,71]. 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 [5,31,50], 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* [12,15,44].
2. **A working prototype.** We design, build, and evaluate a working prototype for the aforementioned construction.

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 [24,25], 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 [13]; a special case was implemented in [8], 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 [48] that are non-interactive [16,17,66]. Specifically, we study non-interactive zero-knowledge proofs that are *succinct*, i.e., short and easy to verify [63]; these are known as *zk-SNARKs* [12,15,44].

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 [3,7,9,15,30,33,39,43,49,53,59–61,69,81,83]; in *fully-succinct zk-SNARKs*, that complexity is independent of computation size [8,11–14,32,36,47,63,64,79]. Working prototypes have been achieved both for preprocessing zk-SNARKs [7,9,30,53,69,83] and for fully-succinct ones [8]. Several works have also explored more in depth various applications of zk-SNARKs [6,21,23,34,41].

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.

Proof-carrying data. *Proof-Carrying Data* (PCD) [24,25] 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 [79] and targeted malleability [19]. Its role in bootstrapping zk-SNARKs was shown in [13], and an implementation of it was achieved in [8].

Outsourcing MapReduce computations. Braun et al. [21] construct (and implement) an interactive protocol for verifiably outsourcing MapReduce computations to untrusted servers. While interacting with the prover, the client has to perform himself the MapReduce shuffling phase; hence, their protocol is neither succinct nor zero knowledge. (In particular, their protocol is not a zk-SNARK and, a fortiori, nor a distributed zk-SNARK.)

Other works on outsourcing computations. Numerous works [2,10,18,21,22,27–29,40,42,46,51,68,73–78,80] 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 [65] 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.

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* [24,25], a cryptographic primitive that enforces a given local invariant, known as the *compliance predicate*, in distributed computations. Such a transformation was described by [13], following [79] and [24]. It was implemented by [8], and

[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.

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 size is the sum of each requirement's size) and entails a large cost in proving time.

We thus extend [8] 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).

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 map node or reduce 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, 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.

2 Preliminaries

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

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). Finally, to simplify notation, we do not make explicit adversaries' auxiliary inputs.

2.1 Commitments

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

- $\mathsf{COMM.Gen}(z) \rightarrow (\mathsf{cm}, \mathsf{trp})$. On input data z, the *commitment generator* COMM.Gen probabilistically samples a commitment cm of z and a corresponding trapdoor trp.
- $\mathsf{COMM.Ver}(z, \mathsf{cm}, \mathsf{trp}) \rightarrow b$. On input data z, commitment cm, and trapdoor trp, the *commitment verifier* COMM.Ver outputs $b = 1$ if cm is a valid commitment of z with respect to the trapdoor trp (and $b = 0$ otherwise).

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.

2.2 Merkle Trees

We use Merkle trees [62] (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 $\mathsf{MERKLE} = (\mathsf{MERKLE.GetRoot}, \mathsf{MERKLE.GetPath},)$ MERKLE.CheckPath with the following syntax:

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

2.3 MapReduce

Overview of MapReduce MapReduce is a programming model for describing data-parallel computations to be run on computer clusters [35]. 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.

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 $\mathtt{x} \in (\mathcal{K}^1 \times \mathcal{V}^1)^*$, the output of the execution of $(\mathsf{Map}, \mathsf{Reduce})$ on $\mathtt{x}$, denoted $[\mathsf{Map}, \mathsf{Reduce}](\mathtt{x})$, is the result $\mathtt{y} \in (\mathcal{K}^3 \times \mathcal{V}^3)^*$ of the following (abstract) computation.

1. **Map step.** For each $i \in \{1, \ldots, |\mathtt{x}|\}$, letting (k_i^1, v_i^1) be the i-th phase-1 record in $\mathtt{x}$, 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 $\mathtt{z} = \big((k_{i,j}^2, v_{i,j}^2)\big)_{i,j}$.

2. **Shuffle step.** Shuffle the list $\mathtt{z}$ so that records with the same key are grouped together. This step induces, for each unique key k^2 appearing in $\mathtt{z}$, a corresponding list $\boldsymbol{v^2}$ of values paired with k^2.
3. **Reduce step.** For each unique phase-2 key k^2 in $\mathtt{z}$ and its corresponding list of phase-2 values $\boldsymbol{v^2}$, compute the phase-3 record $(k^3, v^3) = \mathsf{Reduce}(k^2, \boldsymbol{v^2})$. The output $\mathtt{y}$ 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 $\mathtt{x}$ or in $\mathtt{y}$ is irrelevant.[3] In terms of complexity measures, we say that the execution of $(\mathsf{Map}, \mathsf{Reduce})$ on $\mathtt{x}$ 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 $|\mathtt{x}| \cdot m + |\mathtt{y}| \cdot r \leq p$ (where $\mathtt{y} := [\mathsf{Map}, \mathsf{Reduce}](\mathtt{x})$).[4]

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

Definition 1. *For a MapReduce pair* $(\mathsf{Map}, \mathsf{Reduce})$*, the language* $\mathscr{L}_{(\mathsf{Map},\mathsf{Reduce})}$ *consists of the tuples* $(\mathtt{x}, \mathtt{y})$ *for which* $\mathtt{y} = [\mathsf{Map}, \mathsf{Reduce}](\mathtt{x})$.[5]

In this work, we consider the setting where an input $\mathtt{x}$ is not known to the user, but only its commitment cm is (as $\mathtt{x}$ 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 2.1). In contrast to $\mathscr{L}_{(\mathsf{Map},\mathsf{Reduce})}$, instances in $\mathscr{R}^{\mathsf{COMM}}_{(\mathsf{Map},\mathsf{Reduce})}$ contain cm instead of $\mathtt{x}$, and witnesses are extended to contain decommitment information (i.e., the input and commitment trapdoor). More precisely, we define the relation $\mathscr{R}^{\mathsf{COMM}}_{(\mathsf{Map},\mathsf{Reduce})}$ as follows.

Definition 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* $\big((\mathsf{cm}, \mathtt{y}), (\mathtt{x}, \mathsf{trp})\big)$ *such that* $\mathsf{COMM.Ver}(\mathtt{x}, \mathsf{cm}, \mathsf{trp}) = 1$ *and* $(\mathtt{x}, \mathtt{y}) \in \mathscr{L}_{(\mathsf{Map},\mathsf{Reduce})}$.

MapReduce sequences. A single MapReduce execution is at times 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, $I_i \subseteq \{0, \ldots, i-1\}$ 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 $i = 1, \ldots, d$, 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 $\mathtt{y} := \mathtt{y}^{(d)}$.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

[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 [45].

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

any Map_i takes at most m time, each individual execution of any Reduce_i takes at most r time, and $\sum_{i=1}^{d}(|\mathtt{x}^{(i-1)}| \cdot m + |\mathtt{x}^{(i)}| \cdot r) \leq p$.

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 $\big((I_{N,i}, \mathsf{Map}_{N,i}, \mathsf{Reduce}_{N,i})\big)_{i=1}^{d_N}$.

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 2.3.

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.

- $\mathsf{MR.Prove}(\mathsf{pk}, \mathsf{cm}, \mathtt{y}, \mathtt{x}, \mathsf{trp}) \to \pi_{\mathsf{MR}}$. On input a proving key pk, instance $(\mathsf{cm}, \mathtt{y})$, and witness $(\mathtt{x}, \mathsf{trp})$, the *prover* $\mathsf{MR.Prove}$ outputs a proof π_{MR} for the statement "there is $(\mathtt{x}, \mathsf{trp})$ such that $\big((\mathsf{cm}, \mathtt{y}), (\mathtt{x}, \mathsf{trp})\big) \in \mathscr{R}^{\mathsf{COMM}}_{(\mathsf{Map},\mathsf{Reduce})}$".
- $\mathsf{MR.Verify}(\mathsf{vk}, \mathsf{cm}, \mathtt{y}, \pi_{\mathsf{MR}}) \to b$. On input a verification key vk, commitment cm, output $\mathtt{y}$, and proof π_{MR}, the *verifier* $\mathsf{MR.Verify}$ outputs $b = 1$ if he is convinced that there is $(\mathtt{x}, \mathsf{trp})$ such that $\big((\mathsf{cm}, \mathtt{y}), (\mathtt{x}, \mathsf{trp})\big) \in \mathscr{R}^{\mathsf{COMM}}_{(\mathsf{Map},\mathsf{Reduce})}$.

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 the full version. Here we recall succinctness: an honestly-generated proof π_{MR} has $O_\lambda(1)$ bits, and $\mathsf{MR.Verify}(\mathsf{vk}, \mathsf{cm}, \mathtt{y}, \pi_{\mathsf{MR}})$ runs in time $O_\lambda(|\mathtt{y}|)$.

Costs of key generation. The above implies that $(\mathsf{pk}, \mathsf{vk})$ is generated in time $O_\lambda(1) \cdot \mathsf{poly}(|\mathsf{Map}| + |\mathsf{Reduce}|)$, that $|\mathsf{pk}| = O_\lambda(1) \cdot \mathsf{poly}(|\mathsf{Map}| + |\mathsf{Reduce}|)$,

and that $|\mathsf{vk}| = O_\lambda(1)$ (since MR.Verify runs in time $O_\lambda(|\mathtt{y}|)$ for any $\mathtt{y}$). 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, when producing proofs for the relation $\mathscr{R}^{\mathsf{COMM}}_{(\mathsf{Map},\mathsf{Reduce})}$, $\mathsf{MR.Prove}(\mathsf{pk},\cdot,\cdot,\cdot,\cdot)$ is a family of MapReduce sequences that is $(\mathsf{Map},\mathsf{Reduce})$-*faithful*, a property defined below.

Definition 3. *Given a MapReduce pair* $(\mathsf{Map},\mathsf{Reduce})$*, a family of MapReduce sequences* $(\mathbf{S}_N)_{N\in\mathbb{N}}$ *is* $(\mathsf{Map},\mathsf{Reduce})$*-faithful if, for all* $N \in \mathbb{N}$ *and* $\big((\mathsf{cm},\mathtt{y}),(\mathtt{x},\mathsf{trp})\big) \in \mathscr{R}^{\mathsf{COMM}}_{(\mathsf{Map},\mathsf{Reduce})}$ *with* $|\mathtt{x}|+|\mathtt{y}| \le N$*:*

- *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* $\mathtt{x}$ *is* (m,r,p)*-bounded then the execution of* $\mathbf{S}_N$ *on* $(\mathsf{cm},\mathtt{y},\mathtt{x},\mathsf{trp})$ *is* $(O_\lambda(m),O_\lambda(r),O_\lambda(p))$*-bounded.*

4 Definition of Multi-predicate PCD

Proof-carrying data (PCD) [24,25] 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 [79] and targeted malleability [19].

Extending PCD to multiple predicates. The definition of PCD naturally generalizes to compliance with respect to a *vector* $\boldsymbol{\Pi}$ of compliance predicates (rather than a single predicate). Namely, a msg is $\boldsymbol{\Pi}$*-compliant* if it is consistent with a history in which each node's local computation satisfies some predicate Π

in the vector $\boldsymbol{\Pi}$. 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 $\boldsymbol{\Pi}$ into a single predicate $\Pi^\star$ that (a) reasons about which predicate in $\boldsymbol{\Pi}$ to use at a give node, and (b) enforces a message's type and payload separation. However, this simulation incurs a significant overhead: the size of $\Pi^\star$ is the sum of the sizes of all the predicates in $\boldsymbol{\Pi}$, 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 sizes 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 $\boldsymbol{\Pi}$ of compliance predicates, and multi-predicate PCD.

Transcripts. A *(distributed-computation) transcript* is a tuple $\mathsf{T} = (G, \text{TYPE}, \text{LOC}, \text{PAYLOAD})$, where:
- $G = (V, E)$ is a directed acyclic graph with node set V and edge set $E \subseteq V \times V$;
- $\text{TYPE}\colon V \to \mathbb{N}$ are node labels;
- $\text{LOC}\colon V \to \{0,1\}^*$ are (another kind of) node labels; and
- $\text{PAYLOAD}\colon E \to \{0,1\}^*$ are edge labels.

The *message* of an edge $(u, v) \in E$ is the pair $\text{MSG}(u, v) := (\text{TYPE}(u), \text{PAYLOAD}(u, v))$. The *outputs* of the transcript T, denoted $\text{OUTS}(\mathsf{T})$, is the set of messages $\text{MSG}(\tilde{u}, \tilde{v})$ where $(\tilde{u}, \tilde{v}) \in E$ and $\tilde{v}$ 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 $\mathsf{type}(\Pi)$. Given a vector $\boldsymbol{\Pi}$ of compliance predicates, we say that:
- a transcript $\mathsf{T} = (G, \text{LOC}, \text{TYPE}, \text{PAYLOAD})$ is **$\boldsymbol{\Pi}$-compliant**, denoted $\boldsymbol{\Pi}(\mathsf{T}) = \mathsf{OK}$, if:
  - (i) for each $v \in V$, $\text{TYPE}(v) = 0$ 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 $\Pi \in \boldsymbol{\Pi}$ with $\text{TYPE}(v) = \mathsf{type}(\Pi)$ such that
$$\Pi\Big(\text{MSG}(v, w), \text{LOC}(v), \big(\text{MSG}(u, v)\big)_{u \in \mathsf{parents}(v)}\Big) \text{ accepts.}$$
- a message msg is **$\boldsymbol{\Pi}$-compliant** if there is a transcript T such that $\boldsymbol{\Pi}(\mathsf{T}) = \mathsf{OK}$ and $\mathsf{msg} \in \text{OUTS}(\mathsf{T})$.

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 $\boldsymbol{\Pi}$, and outputs a proving key pk and a verification key vk; these keys allow anyone to prove/verify that a message msg is $\boldsymbol{\Pi}$-compliant. This is achieved by attaching a short and easy-to-verify proof to each message: given pk, input messages $\mathbf{msg}_{\mathsf{in}}$ with proofs $\boldsymbol{\pi}_{\mathsf{in}}$, local data loc, and an output message msg (allegedly, $\boldsymbol{\Pi}$-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 $\boldsymbol{\Pi}$-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 the full version. 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 $O_\lambda(|\mathsf{msg}|)$.

5 Step II: from Multi-predicate PCD to Distributed zk-SNARKs

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 $\boldsymbol{\Pi}^{\mathsf{MR}}$ of predicates.
- **Construction of the proof system** (Section 5.2): suitably invoke the multi-predicate PCD system on the vector $\boldsymbol{\Pi}^{\mathsf{MR}}$ in order to construct a distributed zk-SNARK for MapReduce.

5.1 Compliance Engineering for MapReduce

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

Definition 4. *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* $(\mathsf{cm}, |\mathtt{y}|)$ *and, for each* $i \in \{1, \ldots, |\mathtt{y}|\}$*, a message with type* 2 *and payload* $(\mathsf{cm}, \mathtt{y}_i)$*.*

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

"Given an instance $(\mathsf{cm}, \mathtt{y})$, is there a witness $(\mathtt{x}, \mathsf{trp})$ such that $\big((\mathsf{cm}, \mathtt{y}), (\mathtt{x}, \mathsf{trp})\big)$ 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 $\boldsymbol{\Pi}^{\mathsf{MR}}$-compliant $(\mathsf{cm}, \mathtt{y})$-compatible transcript T?"

More precisely:

Theorem 1. *There exists a commitment scheme* COMM *such that, for every MapReduce pair* $(\mathsf{Map}, \mathsf{Reduce})$*, there exist a vector* $\boldsymbol{\Pi}^{\mathsf{MR}}$ *of compliance predicates and two algorithms* $\mathsf{Eval}, \mathsf{Ext}$ *satisfying the following properties.*

- Efficiency.
  - *The vector* $\boldsymbol{\Pi}^{\mathsf{MR}}$ *consists of* 7 *predicates, with the following sizes:*
    $$|\boldsymbol{\Pi}^{\mathsf{MR}}[1]| = O_\lambda(|\mathsf{Map}|),\ |\boldsymbol{\Pi}^{\mathsf{MR}}[2]| = O_\lambda(|\mathsf{Reduce}|),\ \textit{and}$$
    $$|\boldsymbol{\Pi}^{\mathsf{MR}}[3]|, \ldots, |\boldsymbol{\Pi}^{\mathsf{MR}}[7]| = O_\lambda(1),$$
    where, above, $|\cdot|$ *denotes per-input running time of the underlying algorithm.*
  - *The algorithm* Eval *is* $(\mathsf{Map}, \mathsf{Reduce})$*-faithful.*
  - *The algorithm* Ext *is linear time.*
- Completeness. *For any instance* $(\mathsf{cm}, \mathtt{y})$*, if there is* $(\mathtt{x}, \mathsf{trp})$ *such that* $\big((\mathsf{cm}, \mathtt{y}), (\mathtt{x}, \mathsf{trp})\big)$ *is in* $\mathscr{R}^{\mathsf{COMM}}_{(\mathsf{Map},\mathsf{Reduce})}$*, then there is a* $\boldsymbol{\Pi}^{\mathsf{MR}}$*-compliant* $(\mathsf{cm}, \mathtt{y})$*-compatible transcript* T*; moreover,* $\mathsf{Eval}(\mathsf{cm}, \mathtt{y}, \mathtt{x}, \mathsf{trp})$ *outputs* $\mathrm{OUTS}(\mathsf{T})$ *by dynamically generating* T *"node by node".*
- Proof of knowledge. *For any instance* $(\mathsf{cm}, \mathtt{y})$*, if there is a* $\boldsymbol{\Pi}^{\mathsf{MR}}$*-compliant* $(\mathsf{cm}, \mathtt{y})$*-compatible transcript* T*, then* $\mathsf{Ext}(\mathsf{T})$ *outputs* $(\mathtt{x}, \mathsf{trp})$ *such that* $\big((\mathsf{cm}, \mathtt{y}), (\mathtt{x}, \mathsf{trp})\big)$ *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 $\boldsymbol{\Pi}^{\mathsf{MR}}$ of predicates with the property that, given $(\mathsf{cm}, \mathtt{y})$, if there is a distributed-computation transcript T that is both $\boldsymbol{\Pi}^{\mathsf{MR}}$-compliant and $(\mathsf{cm}, \mathtt{y})$-compatible, then we can find $(\mathtt{x}, \mathsf{trp})$ for which $\mathsf{COMM.Ver}(\mathtt{x}, \mathsf{cm}, \mathsf{trp}) = 1$ and $\mathtt{y} = [\mathsf{Map}, \mathsf{Reduce}](\mathtt{x})$. Intuitively, we achieve proof of knowledge by engineering the predicates in $\boldsymbol{\Pi}^{\mathsf{MR}}$ so that the transcript T is forced 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.

Failed Attempt #1 It is natural to begin by designing two predicates $\Pi_{\mathsf{exe}}^{\mathsf{Map}}$ and $\Pi_{\mathsf{exe}}^{\mathsf{Reduce}}$ that simply capture the correct execution of a mapper and reduce node, respectively, as in Figure 2.

Now suppose that we see a $(\Pi_{\mathsf{exe}}^{\mathsf{Map}}, \Pi_{\mathsf{exe}}^{\mathsf{Reduce}})$-compliant message msg. What can we deduce about the history of computations that led to msg? If $\mathsf{msg.type} = \mathsf{type}(\Pi_{\mathsf{exe}}^{\mathsf{Map}})$, then msg was output by a node at which the predicate $\Pi_{\mathsf{exe}}^{\mathsf{Map}}$ was checked; conversely, if $\mathsf{msg.type} = \mathsf{type}(\Pi_{\mathsf{exe}}^{\mathsf{Reduce}})$, then msg was output by a node at which the predicate $\Pi_{\mathsf{exe}}^{\mathsf{Reduce}}$ was checked. Suppose, for example, that $\mathsf{msg.type} = \mathsf{type}(\Pi_{\mathsf{exe}}^{\mathsf{Reduce}})$. By construction of $\Pi_{\mathsf{exe}}^{\mathsf{Reduce}}$, 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 $\mathbf{msg}_{\mathsf{in}}$ whose payloads contain phase-2 records $\left((k_j^2, v_j^2)\right)_j$ 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", to see what properties each phase-2 record (k_j^2, v_j^2) satisfies, we run into issues not addressed by the above construction of $\Pi_{\mathsf{exe}}^{\mathsf{Map}}$ and $\Pi_{\mathsf{exe}}^{\mathsf{Reduce}}$. Namely,

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

Failed Attempt #2 We augment $\Pi_{\mathsf{exe}}^{\mathsf{Map}}$ and $\Pi_{\mathsf{exe}}^{\mathsf{Reduce}}$ to address these issues. Roughly, we address Issue I by inspecting message types: $\Pi_{\mathsf{exe}}^{\mathsf{Map}}$ ensures that its input messages have type 0 (i.e., are not output by previous nodes); while $\Pi_{\mathsf{exe}}^{\mathsf{Reduce}}$ ensures that they have type $\mathsf{type}(\Pi_{\mathsf{exe}}^{\mathsf{Map}})$. As for Issue II, we augment all messages with a commitment cm to the (overall) input $\mathtt{x}$ and extend $\Pi_{\mathsf{exe}}^{\mathsf{Map}}$ to authenticate the phase-1 record it receives. We now describe 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 1 for more details; we denote the underlying commitment scheme by COMM' and the Merkle-tree scheme by MERKLE (and use notation introduced in Section 2.1 and Section 2.2).

Fig. 1. Choice of commitment scheme COMM (obtained from MERKLE and COMM')

$\mathsf{COMM.Gen}(\boldsymbol{z})$	$\mathsf{COMM.Ver}(\boldsymbol{z}, \mathsf{cm}, \mathsf{trp})$
1. Compute $\mathsf{rt} := \mathsf{MERKLE.GetRoot}(\boldsymbol{z})$. 2. Compute $n := \lvert\boldsymbol{z}\rvert$. 3. Compute $(\mathsf{cm}_{\mathsf{rt}}, \mathsf{trap}_{\mathsf{rt}}) \leftarrow \mathsf{COMM}'.\mathsf{Gen}(\mathsf{rt})$. 4. Compute $(\mathsf{cm}_n, \mathsf{trap}_n) \leftarrow \mathsf{COMM}'.\mathsf{Gen}(n)$. 5. Set $\mathsf{cm} := (\mathsf{cm}_{\mathsf{rt}}, \mathsf{cm}_n)$. 6. Set $\mathsf{trp} := (\mathsf{trap}_{\mathsf{rt}}, \mathsf{trap}_n)$. 7. Output $(\mathsf{cm}, \mathsf{trp})$.	1. Compute $\mathsf{rt} := \mathsf{MERKLE.GetRoot}(\boldsymbol{z})$. 2. Compute $n := \lvert\boldsymbol{z}\rvert$. 3. Parse cm as a pair $(\mathsf{cm}_{\mathsf{rt}}, \mathsf{cm}_n)$. 4. Parse trp as a pair $(\mathsf{trap}_{\mathsf{rt}}, \mathsf{trap}_n)$. 5. Check that $\mathsf{COMM}'.\mathsf{Ver}(\mathsf{rt}, \mathsf{cm}_{\mathsf{rt}}, \mathsf{trap}_{\mathsf{rt}}) = 1$. 6. Check that $\mathsf{COMM}'.\mathsf{Ver}(n, \mathsf{cm}_n, \mathsf{trap}_n) = 1$. 7. Output 1 if the above checks succeeded (else, 0).

Next, in Figure 3, we describe the two (updated) predicates $\Pi_{\mathsf{exe}}^{\mathsf{Map}}$ and $\Pi_{\mathsf{exe}}^{\mathsf{Reduce}}$.

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

Overall, each $(\Pi^{\mathsf{Map}}_{\mathrm{exe}}, \Pi^{\mathsf{Reduce}}_{\mathrm{exe}})$-compliant message msg with $\mathsf{msg.type} = \mathsf{type}(\Pi^{\mathsf{Reduce}}_{\mathrm{exe}})$ 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 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).

Our Approach Unlike previous ones, the above issue is conceptually more complex: tackling it requires ensuring correct shuffling, which is a global computation involving all of the phase-2 (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 other compliance predicates, to ensure correct shuffling in a distributed way.

Further extending $\boldsymbol{\Pi^{\mathsf{Map}}_{\mathrm{exe}}}$ and $\boldsymbol{\Pi^{\mathsf{Reduce}}_{\mathrm{exe}}}$. Roughly, we extend $\Pi^{\mathsf{Map}}_{\mathrm{exe}}$ to store, in the output message, the index i relative to which the phase-1 record, contained in the input message, was authenticated. Subsequently, when receiving several input messages, $\Pi^{\mathsf{Reduce}}_{\mathrm{exe}}$ verifies that all the indices contained in them are distinct. This additional check prevents duplicate messages from being sent to the same reduce node. However, the check does not prevent the same message from being sent to two different reducer nodes, a message from being dropped altogether, or messages with the same key from being sent to two different reducer nodes. 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 $\mathsf{cm}_{k^2}, \mathsf{cm}_{d_{\mathsf{in}}}$ to these are stored, to not violate zero knowledge (by storing information about the internals of the computation in final outputs of the distributed computation). As we now explain, other compliance predicates use the underlying values k^2, d_{in}; for now, in Figure 4, we summarize the changes to $\Pi^{\mathsf{Map}}_{\mathrm{exe}}$ and $\Pi^{\mathsf{Reduce}}_{\mathrm{exe}}$ (highlighted in blue).

$\underline{\Pi^{\mathsf{Map}}_{\mathrm{exe}}(\mathsf{msg}, \mathsf{loc}, \mathbf{msg}_{\mathsf{in}})}$

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

$\underline{\Pi^{\mathsf{Reduce}}_{\mathrm{exe}}(\mathsf{msg}, \mathsf{loc}, \mathbf{msg}_{\mathsf{in}})}$

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

Fig. 2. Summary of the construction of $\Pi^{\mathsf{Map}}_{\mathrm{exe}}$ and $\Pi^{\mathsf{Reduce}}_{\mathrm{exe}}$ for "Failed attempt #1" (see Section 5.1)

$\underline{\Pi^{\mathsf{Map}}_{\mathrm{exe}}(\mathsf{msg}, \mathsf{loc}, \mathbf{msg}_{\mathsf{in}})}$

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

$\underline{\Pi^{\mathsf{Reduce}}_{\mathrm{exe}}(\mathsf{msg}, \mathsf{loc}, \mathbf{msg}_{\mathsf{in}})}$

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

Fig. 3. Summary of the construction of $\Pi^{\mathsf{Map}}_{\mathrm{exe}}$ and $\Pi^{\mathsf{Reduce}}_{\mathrm{exe}}$ for "Failed attempt #2"

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, called $\Pi^{\mathsf{Map}}_{\mathrm{fmt}}$, $\Pi^{\mathsf{Reduce}}_{\mathrm{fmt}}$, $\Pi^{\mathsf{Map}}_{\mathrm{sum}}$, $\Pi^{\mathsf{Reduce}}_{\mathrm{sum}}$, and Π_{fin}, for checking two main distributed computations: a tree-like distributed computation that aggregates information stored by all the messages output by mapper nodes, and another tree-like distributed computation that aggregates information stored by all the messages output by reducer nodes. By comparing the final outputs of these two tree-like distributed computations, we can check if correct shuffling occurred.

$\Pi^{\mathsf{Map}}_{\mathrm{exe}}(\mathsf{msg}, \mathsf{loc}, \mathbf{msg}_{\mathsf{in}})$

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

$\Pi^{\mathsf{Reduce}}_{\mathrm{exe}}(\mathsf{msg}, \mathsf{loc}, \mathbf{msg}_{\mathsf{in}})$

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

Fig. 4. Summary of the construction of $\Pi^{\mathsf{Map}}_{\mathrm{exe}}$ and $\Pi^{\mathsf{Reduce}}_{\mathrm{exe}}$ for "Our approach" (see Section 5.1). The text that is highlighted in blue denotes the differences from the construction in Figure 3.

Aggregating mappers' outputs. We describe each of these tree-like distributed computations, starting with the one for messages output by mapper nodes. Each message output by a mapper node has a payload that looks like $(\mathsf{cm}, i, k^2, v^2)$. We use, for each such message, a node to reformat the message into a new with payload $(\mathsf{cm}, a^\perp, a^\top, b, c)$ where $a^\perp = a^\top = i$ and $b = c = 1$. Afterwards, we use a tree of nodes to aggregate all the resulting messages into a final single one, by pairwise transforming two input messages $(\mathsf{cm}, a^\perp_1, a^\top_1, b_1, c_2)$ and $(\mathsf{cm}, a^\perp_2, a^\top_2, b_2, c_2)$ into the new message $(\mathsf{cm}, a^\perp_1, a^\top_2, b_1 + b_2, c_1 + c_2)$, provided that $a^\top_1 < a^\perp_2$. Intuitively, the second and third components of a message denote the least and largest index seen so far, the fourth component counts the number of mappers, and the fifth counts the number of mapper outputs. If M denotes the number of mappers, the final message, output by the "root node" has payload $(\mathsf{cm}, 1, M, M, M)$. If, however, some messages are either duplicated or dropped, then at least one node will not satisfy its compliance predicate. We realize this idea by designing two new compliance predicates, $\Pi^{\mathsf{Map}}_{\mathrm{fmt}}$ and $\Pi^{\mathsf{Map}}_{\mathrm{sum}}$, respectively for enforcing the reformatting and aggregation of mapper nodes' output messages.

Aggregating reducers' outputs. We now turn to the tree-like distributed computation to aggregate outputs of reducer nodes. Each message output by a reducer node has a payload that looks like $(\mathsf{cm}, k^3, v^3, \mathsf{cm}_{k^2}, \mathsf{cm}_{d_{\mathsf{in}}})$. Similarly to (but not exactly equal to) above, we use a node to reformat the message

into a new with payload $(\mathsf{cm}, a^{\perp}, a^{\top}, b, c)$ where $a^{\perp} = a^{\top} = k^2$, $b = 1$ $c = d_{\mathsf{in}}$ (note that the values k^2 and d_{in} can be obtained by receiving decommitment information as part of the node's local data loc). Afterwards, again similarly to above, we use a tree of nodes to aggregate all the resulting messages into a final single one, by pairwise transforming two input messages $(\mathsf{cm}, a_1^{\perp}, a_1^{\top}, b_1, c_1)$ and $(\mathsf{cm}, a_2^{\perp}, a_2^{\top}, b_2, c_2)$ into a new message $(\mathsf{cm}, a_1^{\perp}, a_2^{\top}, b_1 + b_2, c_1 + c_2)$, provided that $a_1^{\top} < a_2^{\perp}$. The final message, output by the root node, looks like $(\mathsf{cm}, k^2_{\min}, k^2_{\max}, R, M)$, where $k^2_{\min}$ and $k^2_{\max}$ are respectively the least and largest keys encountered, R is the total number of reducer nodes, and M is the total number of inputs received by reducer nodes. Again, we concretely realize the above strategy by designing two new predicates, $\Pi^{\mathsf{Reduce}}_{\mathrm{fmt}}$ and $\Pi^{\mathsf{Reduce}}_{\mathrm{sum}}$, respectively for enforcing the reformatting and aggregation of reducer nodes' output messages; both take $O_\lambda(1)$ to execute.

Consistency between aggregations. After both aggregations have taken place, we are left with two messages $\mathsf{msg}^{\mathsf{Map}}_{\mathsf{sum}}$ and $\mathsf{msg}^{\mathsf{Reduce}}_{\mathsf{sum}}$, respectively with payloads $(\mathsf{cm}, 1, M, M, M)$ and $(\mathsf{cm}, k^2_{\min}, k^2_{\max}, R, M)$, resulting in an output message $\mathsf{msg}_{\mathrm{fin}}$ with payload (cm, R). A simple predicate Π_{fin} performs consistency checks, such as ensuring that the value of M is actually equal between the two messages (and consistency with the commitment cm_M stored in cm). The message $\mathsf{msg}_{\mathrm{fin}}$ that the two messages $\mathsf{msg}^{\mathsf{Map}}_{\mathsf{sum}}$ and $\mathsf{msg}^{\mathsf{Reduce}}_{\mathsf{sum}}$ have been successfully compared, which demonstrates that the outputs of all M mapper nodes were correctly shuffled to R reducer nodes. (We exclude M from $\mathsf{msg}_{\mathrm{fin}}$, for zero-knowledge reasons.)

Throughout, we leverage message types to enforce communication flow between nodes subject to different compliance predicates.

From sketch to proof. The above sketches how the Eval algorithm produces a suitable graph of nodes, culminating in the transcript's output, as stated in Theorem 1. It skims over many details, some of which are provided in the full version. For example, above we have not explained how to handle the case where a mapper node (or even a reducer node) outputs more than one record. Moreover, not only do we work out the details of a solution, but we also bring the solution to efficient implementations of arithmetic circuits for each of the seven compliance predicates.

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 $\mathsf{MR.Verify}$. (We describe the commitment scheme COMM in Figure 1.)

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 1 to deduce, from $(\mathsf{Map}, \mathsf{Reduce})$, the vector $\boldsymbol{\Pi}^{\mathsf{MR}}$ of compliance predicates.
2. Use the PCD key generator $\mathbb{G}$ to compute a PCD key pair for $\boldsymbol{\Pi}^{\mathsf{MR}}$: $(\mathsf{pk}, \mathsf{vk}) := \mathbb{G}(1^\lambda, \boldsymbol{\Pi}^{\mathsf{MR}})$.

3. Set $\mathsf{pk} := (\mathsf{Map}, \mathsf{Reduce}, \mathsf{pk})$ and $\mathsf{vk} := (\mathsf{vk})$; output $(\mathsf{pk}, \mathsf{vk})$.

The prover $\mathbf{MR.Prove(pk, cm, y, x, trp) \to \pi_{MR}}$. On input a proving key pk, an instance $(\mathsf{cm}, \mathtt{y})$, and a witness $(\mathtt{x}, \mathsf{trp})$, the prover MR.Prove computes a non-interactive proof π_{MR} for the statement "I know $(\mathtt{x}, \mathsf{trp})$ such that $\big((\mathsf{cm}, \mathtt{y}), (\mathtt{x}, \mathsf{trp})\big) \in \mathscr{R}^{\mathsf{COMM}}_{(\mathsf{Map},\mathsf{Reduce})}$" as follows. By Theorem 1, we know that there is a $\boldsymbol{\Pi}^{\mathsf{MR}}$-compliant $(\mathsf{cm}, \mathtt{y})$-compatible transcript T and, moreover, that OUTS(T) can be obtained via a $(\mathsf{Map}, \mathsf{Reduce})$-faithful evaluator Eval, which takes as input the the instance $(\mathsf{cm}, \mathtt{y})$ and its witness $(\mathtt{x}, \mathsf{trp})$. Thus, the prover MR.Prove computes π_{MR} by recursively invoking the PCD prover $\mathbb{P}$ on T, following Eval as it computes new nodes of T, by providing to $\mathbb{P}$, at each node, the relevant input messages and their proofs, local data, and output message. At the end of this process, itself $(\mathsf{Map}, \mathsf{Reduce})$-faithful, MR.Prove sets π_{MR} equal to the concatenation of the proofs of all messages in OUTS(T).

The verifier $\mathbf{MR.Verify(vk, cm, y, \pi_{MR}) \to b}$. On input a verification key vk, commitment cm, output $\mathtt{y}$, and proof π_{MR}, the verifier MR.Verify computes a decision bit b as follows.

1. Parse vk as a PCD verification key vk.
2. Use the instance $(\mathsf{cm}, \mathtt{y})$ to construct the following output messages (recall Definition 4):

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

$$\mathsf{msg}_i \begin{cases} .\mathsf{type} := 2 \\ .\mathsf{payload} := (\mathsf{cm}, \mathtt{y}_i) \end{cases} .$$

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

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

As seen above, the combination of compliance engineering and PCD systems provides 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.

Turning to security, we recall that, when invoking a PCD system to produce proofs along a distributed computation, proof of knowledge is achieved by recursively extracting "past proofs" from known ones. This process is technically quite delicate, and a formal treatment of it is in [13]. Here we only note that the distributed computations considered in this paper are shallow (of logarithmic depth) and are thus easily amenable to recursive proof extraction.

6 Step I: Construction of Multi-predicate PCD

We discuss Step I of our bootstrapping theorem: constructing multi-predicate PCD from (preprocessing) zk-SNARKs. As in [8], 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 [55]).

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 [9], a *preprocessing zk-SNARK* [13,15] 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 $O_\lambda(|x|)$.

6.2 Review of the [8] Construction

For efficiency reasons, Ben-Sasson et al. [8] 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 q_α and q_β: (a) $(G_\alpha, P_\alpha, V_\alpha)$ proves/verifies satisfiability of $\mathbb{F}_{q_\beta}$-arithmetic circuits, while V_α is an $\mathbb{F}_{q_\alpha}$-arithmetic circuit; instead, (b) $(G_\beta, P_\beta, V_\beta)$ proves/verifies satisfiability of $\mathbb{F}_{q_\alpha}$-arithmetic circuits, while V_β is an $\mathbb{F}_{q_\beta}$-arithmetic circuit. This property is achieved by instantiating the two zk-SNARKs via a *PCD-friendly 2-cycle of elliptic curves* (see [8] 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 $\mathbb{F}_{q_\beta}$ and $C_{\mathsf{pcd},\beta}$ over the field $\mathbb{F}_{q_\alpha}$. 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 $a_\alpha = (\mathsf{loc}, \mathbf{msg}_{\mathsf{in}}, \boldsymbol{\pi}_{\mathsf{in}})$, use V_β to verify that each input message $\mathbf{msg}_{\mathsf{in}}[j]$ has a valid proof $\boldsymbol{\pi}_{\mathsf{in}}[j]$, and check that Π accepts the output message msg, local data loc, and input messages $\mathbf{msg}_{\mathsf{in}}$.
– $C_{\mathsf{pcd},\beta}$: 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 $C_{\mathsf{pcd},\beta}$, [8] additionally use hashing, pre-computation, and hardcoding. First, the input x_α to $C_{\mathsf{pcd},\alpha}$ is $H(\mathsf{bits}(\mathsf{vk}_\beta) \| \mathsf{bits}(\mathsf{msg}))$, where H is a collision-resistant function mapping $\{0,1\}$-vectors to $\mathbb{F}_{q_\beta}$-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 $\mathbb{F}_{q_\beta}$-vector, and thus cannot be passed as input to $C_{\mathsf{pcd},\beta}$, which is an $\mathbb{F}_{q_\alpha}$-arithmetic circuit. This issue is addressed via two "repacking circuits" that map information content from elements in $\mathbb{F}_{q_\beta}$ to ones in $\mathbb{F}_{q_\alpha}$ 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 [8] for more details.

From the point of view of this paper, the construction of [8] 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 [8]'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 $\boldsymbol{\Pi}$, construct a vector $\boldsymbol{C}_{\mathsf{pcd}}$ of PCD circuits. Roughly, for each $\boldsymbol{\Pi}[i]$ in $\boldsymbol{\Pi}$, we construct two circuits, $C_{\mathsf{pcd},\alpha,i}$ and $C_{\mathsf{pcd},\beta,i}$, tasked with recursive proof composition relative to $\boldsymbol{\Pi}[i]$.
– **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 $\boldsymbol{C}_{\mathsf{pcd}}$; the PCD prover $\mathbb{P}$, to prove compliance relative to $\boldsymbol{\Pi}[i]$, 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 $\boldsymbol{\Pi}[i]$ in $\boldsymbol{\Pi}$, we construct two PCD circuits: a *compliance circuit* $C_{\mathsf{pcd},\alpha,i}$, tasked with checking compliance with $\boldsymbol{\Pi}[i]$; 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 [8]'s translation circuit. Namely, $C_{\mathsf{pcd},\beta,i}$ provides a way to translate a zk-SNARK proof relative to the verification key $\mathbf{vk}_\alpha[i]$ (generated for $C_{\mathsf{pcd},\alpha,i}$ and hardcoded in $C_{\mathsf{pcd},\beta,i}$) to one relative to the verification key $\mathbf{vk}_\beta[i]$ (generated for $C_{\mathsf{pcd},\beta,i}$); the translation has the only goal of matching fields up.

The design of $C_{\mathsf{pcd},\alpha,i}$ extends [8]'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 [8]'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 $\mathbf{vk}_\beta$.[6] 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 $\mathsf{type}(\boldsymbol{\Pi}[i])$) 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 $\boldsymbol{\Pi}$ 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 $(\mathbf{pk}_\alpha[i], \mathbf{vk}_\alpha[i])$ for it, and then
     (b) the translation circuit $C_{\mathsf{pcd},\beta,i}$ (hardcoding $\mathbf{vk}_\alpha[i]$) and generate a zk-SNARK key pair $(\mathbf{pk}_\beta[i], \mathbf{vk}_\beta[i])$ for it.
  2. Compute rt, the root of the Merkle tree whose leaves consist of the vector $\mathbf{vk}_\beta$.
  3. Output the key pair $(\mathsf{pk}, \mathsf{vk})$, where $\mathsf{pk} := (\mathbf{pk}_\alpha, \mathbf{vk}_\alpha, \mathbf{pk}_\beta, \mathbf{vk}_\beta, \mathsf{rt})$ and $\mathsf{vk} = (\mathbf{vk}_\beta, \mathsf{rt})$.
- The PCD prover $\mathbb{P}$, given a proving key pk, output message msg, local data loc, and input messages $\mathbf{msg}_{\mathsf{in}}$ with proofs $\boldsymbol{\pi}_{\mathsf{in}}$, works as follows.
  1. Parse pk as a tuple $(\mathbf{pk}_\alpha, \mathbf{vk}_\alpha, \mathbf{pk}_\beta, \mathbf{vk}_\beta, \mathsf{rt})$.
  2. Let $i^\star$ be the index of the compliance predicate $\Pi[i^\star]$ in $\boldsymbol{\Pi}$ that is satisfied by $(\mathsf{msg}, \mathsf{loc}, \mathbf{msg}_{\mathsf{in}})$.

[6] Merely taking x_α to be $H(\mathsf{bits}(\mathbf{vk}_\beta)\|\mathsf{bits}(\mathsf{msg}))$ would cause $C_{\mathsf{pcd},\alpha,i}$'s to be linear, instead of logarithmic, in the number of predicates.

3. Construct a vector $\mathbf{ap}$ of authentication paths, where each $\mathbf{ap}[j]$ is the authentication path, relative to the root rt, for the leaf $\mathbf{vk}_\beta[\boldsymbol{\pi}_{\mathsf{in}}[j].\mathsf{idx}]$.
4. Use rt, $(\mathsf{msg}, \mathsf{loc}, \mathbf{msg}_{\mathsf{in}})$, and $\mathbf{ap}$ to construct an input x_α and a witness a_α for $C_{\mathsf{pcd},\alpha,i}$.
5. Use $\mathbf{pk}_\alpha[i^\star]$ 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 $\mathbf{pk}_\beta[i^\star]$ 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 $(\mathbf{vk}_\beta, \mathsf{rt})$.
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 $\mathbf{vk}_\beta[i^\star]$ to check that π_β is a valid zk-SNARK proof for x_β.

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. As in [8], 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) $\mathbf{inlen}(\Pi)$, the vector for which $\mathbf{inlen}(\Pi)[j]$ 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 $d \in \{0, \ldots, \mathsf{max\text{-}arity}(\Pi)\}$ 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 $l \in \mathbb{N}$,

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

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 $(1+\mathbf{inlen}(\Pi)[j])$; and the arity d. For notational convenience, we write $\Pi(\mathsf{msg}, \mathsf{loc}, \mathbf{msg}_{\mathsf{in}}, d)$ even when $\mathbf{msg}_{\mathsf{in}}$ contains less than $\mathsf{max\text{-}arity}(\Pi)$ messages (and assume that $\mathbf{msg}_{\mathsf{in}}$ 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 [8] in more detail, so here we review them only at high level.

We use n_α and n_β to denote the size (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; these two sizes are fixed, and they equal $n_\alpha := d_{H,\alpha}$ and $n_\beta := \lceil \frac{n_\alpha \cdot \lceil \log r_\alpha \rceil}{\lfloor \log r_\beta \rfloor} \rceil$, where $d_{H,\alpha}$ is the number of elements output by the collision-resistant function H; n_β is the number of elements in $\mathbb{F}_{r_\beta}$ needed to encode n_α elements in $\mathbb{F}_{r_\alpha}$. We use bits_α to denote a function that, given an input y in $\mathbb{F}_{r_\alpha}^\ell$ (for some ℓ), outputs y's binary representation; the corresponding $\mathbb{F}_{r_\alpha}$-arithmetic circuit is denoted $C_{\mathsf{bits},\alpha}$ 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 re-packing function from $\mathbb{F}_{r_\alpha}$ to $\mathbb{F}_{r_\beta}$; 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 $S_{\alpha\to\beta}$. An $\mathbb{F}_{r_\beta}$-arithmetic circuit $C_{V,\alpha}^{\text{online}}$ implementing V_α^{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 $C_{V,\alpha}^{\text{online}}$ 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 $C_{V,\beta}$ 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 $C_{V,\beta}$ 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 for hashing: $C_{H,\alpha}^{\mathsf{out}}$ implements a collision-resistant function $H_\alpha^{\mathsf{out}}\colon \{0,1\}^{m_{H,\alpha}^{\mathsf{out}}} \to \mathbb{F}_{r_\alpha}^{d_{H,\alpha}}$, and $\boldsymbol{C}_{H,\alpha}^{\mathsf{in}}$ is a vector such that each $\boldsymbol{C}_{H,\alpha}^{\mathsf{in}}[j]$ implements a collision-resistant function $\boldsymbol{H}_\alpha^{\mathsf{in}}[j]\colon \{0,1\}^{m_{H,\alpha,j}} \to \mathbb{F}_{r_\alpha}^{d_{H,\alpha}}$; parameters are such that $m_{H,\alpha}^{\mathsf{out}} = (d_{H,\alpha} + 1 + \mathsf{outlen}(\Pi)) \cdot \lceil \log r_\alpha \rceil$ and $m_{H,\alpha,j} = (d_{H,\alpha} + 1 + \mathbf{inlen}(\Pi)[j]) \cdot \lceil \log r_\alpha \rceil$.

Finally, we use an $\mathbb{F}_{r_\alpha}$-arithmetic circuit for verification of Merkle-tree authentication paths: $C_{\mathsf{CheckPath},\alpha,p}$ implements the function $\mathsf{MERKLE.CheckPath}$ (see Section 2.2) for paths of length $\lceil \log p \rceil$.

Construction of the PCD circuits. In Figure 5 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 6 we provide pseudocode for the PCD generator $\mathbb{G}$, prover $\mathbb{P}$, and verifier $\mathbb{V}$. The construction works for a vector $\boldsymbol{\Pi}$ of $\mathbb{F}_{r_\alpha}$-arithmetic compliance predicates $\boldsymbol{\Pi}$. [7] For convenience, we export $i^\star$, the index of the predicate with respect to which compliance is proved, to $\mathbb{P}$'s interface.

[7] For comparison, [8] consider the following special case: $\boldsymbol{\Pi} = (\Pi)$, $\mathbf{inlen}(\Pi)[j] = \mathsf{outlen}(\Pi)$ 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 $\mathsf{type}(\Pi)$), which is why [8] do not discuss message types, and instead only distinguish between messages that are "base case" or not.

$\mathsf{MakePCDCircuitA}(\boldsymbol{C}^{\mathsf{in}}_{H,\alpha}, C^{\mathsf{out}}_{H,\alpha}, C_{S,\alpha\to\beta}, C_{V,\beta}, C_{\mathsf{CheckPath},\alpha,p}, \Pi)$

Set:
- the input size $n_\alpha := d_{H,\alpha}$; and
- the witness size $h_\alpha := (1 + \mathsf{outlen}(\Pi)) + \mathsf{loclen}(\Pi) + 1 + \sum_{j=1}^{\mathsf{max\text{-}arity}(\Pi)} ((1 + \mathbf{inlen}(\Pi)[j]) + |\pi| + |\mathsf{vk}_\beta(n_\beta)| + \ell_{\mathsf{ap}} + 1)$.

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

$\mathsf{MakePCDCircuitB}(\mathsf{pvk}_\alpha, C_{S,\alpha\leftarrow\beta}, C^{\mathsf{online}}_{V,\alpha})$

Set:
- the input size $n_\beta := \left\lceil \frac{n_\alpha \cdot \lceil \log r_\alpha \rceil}{\lfloor \log r_\beta \rfloor} \right\rceil$; and
- the witness size $h_\beta := |\pi_\alpha|$.

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_β as a zk-SNARK proof π_α.
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 $C^{\mathsf{online}}_{V,\alpha}(\mathsf{pvk}_\alpha, x_\alpha, \pi_\alpha) = 1$.

Fig. 5. Construction of PCD circuits for our 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 MR.KeyGen, prover MR.Prove, and verifier 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 [1], an open-source MapReduce framework, is ongoing.

Integration with `libsnark`. We have integrated our code with `libsnark` [72], a C++ library for zk-SNARKs.

PCD generator $\mathbb{G}$

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

1. Set $n_\alpha := d_{H,\alpha}$ 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 $C_{V,\beta}$, the $\mathbb{F}_{r_\alpha}$-arithmetic circuit implementing V_β for inputs of n_β elements in $\mathbb{F}_{r_\beta}$.
5. Construct $C_{V,\alpha}^{\text{online}}$, the $\mathbb{F}_{r_\beta}$-arithmetic circuit implementing V_α^{online} for inputs of n_α elements in $\mathbb{F}_{r_\alpha}$.
6. Construct $C_{\mathsf{CheckPath},\alpha,p}$, 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 size p: $\mathsf{pk.pk}_\alpha$, $\mathsf{pk.pk}_\beta$, $\mathsf{pk.vk}_\alpha$, $\mathsf{pk.vk}_\beta$.
8. Allocate the verification key vk, consisting of:
   (a) a Merkle tree root $\mathsf{vk.rt}$; and
   (b) one vector of size p: $\mathsf{vk.vk}_\beta$.
9. For $i = 1, \ldots, p$, compute proving and verification keys for $\boldsymbol{\Pi}[i]$ as follows:
   (a) Construct $C_{H,\alpha}^{\mathsf{out}}$, the $\mathbb{F}_{r_\alpha}$-arithmetic circuit implementing $H_\alpha^{\mathsf{out}} \colon \{0,1\}^{m_{H,\alpha}^{\mathsf{out}}} \to \mathbb{F}_{r_\alpha}^{d_{H,\alpha}}$ for $\boldsymbol{\Pi}[i]$.
   (b) Construct $\boldsymbol{C}_{H,\alpha}^{\mathsf{in}}$, the vector of $\mathbb{F}_{r_\alpha}$-arithmetic circuits such that $\boldsymbol{C}_{H,\alpha}^{\mathsf{in}}[j]$ implements $\boldsymbol{H}_\alpha^{\mathsf{in}}[j] \colon \{0,1\}^{m_{H,\alpha,j}} \to \mathbb{F}_{r_\alpha}^{d_{H,\alpha}}$ for $\boldsymbol{\Pi}[i]$.
   (c) Compute $C_{\mathsf{pcd},\alpha,i} := \mathsf{MakePCDCircuitA}(\boldsymbol{C}_{H,\alpha}^{\mathsf{in}}, C_{H,\alpha}^{\mathsf{out}}, C_{S,\alpha\to\beta}, C_{V,\beta}, C_{\mathsf{CheckPath},\alpha,p}, \boldsymbol{\Pi}[i])$.
   (d) Compute $(\mathsf{pk}_{\alpha,i}, \mathsf{vk}_{\alpha,i}) := G_\alpha(C_{\mathsf{pcd},\alpha,i})$.
   (e) Compute $\mathsf{pvk}_{\alpha,i} := V_\alpha^{\text{offline}}(\mathsf{pk}_{\alpha,i})$.
   (f) Compute $C_{\mathsf{pcd},\beta,i} := \mathsf{MakePCDCircuitB}(\mathsf{pvk}_{\alpha,i}, C_{S,\alpha\leftarrow\beta}, C_{V,\alpha}^{\text{online}})$.
   (g) Compute $(\mathsf{pk}_{\beta,i}, \mathsf{vk}_{\beta,i}) := G_\beta(C_{\mathsf{pcd},\beta,i})$.
   (h) Set $\mathsf{pk.pk}_\alpha[i] := \mathsf{pk}_{\alpha,i}$, $\mathsf{pk.pk}_\beta[i] := \mathsf{pk}_{\beta,i}$, $\mathsf{pk.vk}_\alpha[i] := \mathsf{vk}_{\alpha,i}$, $\mathsf{pk.vk}_\beta[i] := \mathsf{vk}_{\beta,i}$, $\mathsf{vk.vk}_\beta[i] := \mathsf{vk}_{\beta,i}$.
10. Compute $\mathsf{rt} := \mathsf{MERKLE.GetRoot}(\mathsf{vk}_\beta)$ 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 $\boldsymbol{\Pi}[i^\star]$ in $\boldsymbol{\Pi}$, with respect to which compliance is proved
- output message $\mathsf{msg} \in \mathbb{F}_{r_\alpha}^{1+\mathsf{outlen}(\boldsymbol{\Pi}[i^\star])}$
- local data $\mathsf{loc} \in \mathbb{F}_{r_\alpha}^{\mathsf{loclen}(\boldsymbol{\Pi}[i^\star])}$
- arity $d \in \{0, \ldots, \mathsf{max\text{-}arity}(\boldsymbol{\Pi}[i^\star])\}$
- d input messages $\mathbf{msg}_{\mathsf{in}}$, each $\mathbf{msg}_{\mathsf{in}}[j] \in \mathbb{F}_{r_\alpha}^{1+\mathsf{inlen}(\boldsymbol{\Pi}[i^\star])[j]}$
- d corresponding proofs $\boldsymbol{\pi}_{\mathsf{in}}$ (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 $\boldsymbol{\Pi}[i^\star]$

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

Fig. 6. Construction of a multi-predicate PCD system

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* [35], 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 $\big((k_1^2, v_1^2), \ldots, (k_\ell^2, v_\ell^2)\big)$, with $\ell \leq 100$, denoting that the word k_i^2 appears v_i^2 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 $\boldsymbol{v^2}$ for k^2, emits the output record $(k^3, v^3) = (k^2, \sum_i \boldsymbol{v^2}[i])$, 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).

Experimental results. We ran our system on the word counting example, on our benchmarking system. Each of the reported times 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" [37], 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, and R, the number of distinct words in the document:

$$M \cdot \big(\mathsf{cost}(\Pi_{\mathrm{exe}}^{\mathsf{Map}}) + \mathsf{cost}(\Pi_{\mathrm{fmt}}^{\mathsf{Map}}) + 2 \cdot \mathsf{cost}(\Pi_{\mathrm{sum}}^{\mathsf{Map}})\big) + R \cdot \big(\mathsf{cost}(\Pi_{\mathrm{exe}}^{\mathsf{Reduce}}) + \mathsf{cost}(\Pi_{\mathrm{fmt}}^{\mathsf{Reduce}}) + 2 \cdot \mathsf{cost}(\Pi_{\mathrm{sum}}^{\mathsf{Reduce}})\big) + \mathsf{cost}(\Pi_{\mathrm{fin}}).$$

The above costs have the following meaning, and the following measured values on our reference node: $\mathsf{cost}(\Pi_{\mathrm{exe}}^{\mathsf{Map}}) \approx 9.3\,\mathrm{s}$ is the cost of proving execution of a mapper node; $\mathsf{cost}(\Pi_{\mathrm{exe}}^{\mathsf{Reduce}}) \approx 45.2\,\mathrm{s}$ is the cost of proving execution of a reducer node; $\mathsf{cost}(\Pi_{\mathrm{fmt}}^{\mathsf{Map}}) \approx 13.6\,\mathrm{s}$ and $\mathsf{cost}(\Pi_{\mathrm{sum}}^{\mathsf{Map}}) \approx 14.2\,\mathrm{s}$, as well as $\mathsf{cost}(\Pi_{\mathrm{fmt}}^{\mathsf{Reduce}}) \approx 13.8\,\mathrm{s}$

and $\mathsf{cost}(\Pi_{\text{sum}}^{\mathsf{Reduce}}) \approx 14.3\,\text{s}$ denote the individual costs in proving the correctness of aggregation of mapper nodes' outputs and reducer nodes' inputs, respectively; and $\mathsf{cost}(\Pi_{\text{fin}}) \approx 14.3\,\text{s}$ is the cost of producing the final proof.

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: (a) the costs of $\Pi_{\text{fmt}}^{\mathsf{Map}}$ and $\Pi_{\text{sum}}^{\mathsf{Map}}$ remain fixed for all MapReduce computations; (b) the costs of $\Pi_{\text{fmt}}^{\mathsf{Reduce}}$, $\Pi_{\text{sum}}^{\mathsf{Reduce}}$ 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; (c) the cost of $\Pi_{\text{exe}}^{\mathsf{Map}}$ changes depending on $N^{\max}$, the maximum number of mapper outputs, and Map's running time. The cost of $\Pi_{\text{exe}}^{\mathsf{Reduce}}$ is dominated by the cost incurred by performing $d_{\mathsf{in}}^{\max}$ proof verifications, each costing $\approx 90{,}000$ gates.

References

1. Apache Hadoop
2. Applebaum, B., Ishai, Y., Kushilevitz, E.: From secrecy to soundness: efficient verification via secure computation. In: Abramsky, S., Gavoille, C., Kirchner, C., Meyer auf der Heide, F., Spirakis, P.G. (eds.) ICALP 2010. LNCS, vol. 6198, pp. 152–163. Springer, Heidelberg (2010)
3. Backes, M., Fiore, D., Reischuk, R.M.: Nearly practical and privacy-preserving proofs on authenticated data (2014)
4. Bellare, M., Goldreich, O.: On defining proofs of knowledge. In: Brickell, E.F. (ed.) CRYPTO 1992. LNCS, vol. 740, pp. 390–420. Springer, Heidelberg (1993)
5. Bellare, M., Palacio, A.: The knowledge-of-exponent assumptions and 3-round zero-knowledge protocols. In: Franklin, M. (ed.) CRYPTO 2004. LNCS, vol. 3152, pp. 273–289. Springer, Heidelberg (2004)
6. Ben-Sasson, E., Chiesa, A., Garman, C., Green, M., Miers, I., Tromer, E., Virza, M.: Zerocash: decentralized anonymous payments from bitcoin. In: SP 2014 (2014)
7. Ben-Sasson, E., Chiesa, A., Genkin, D., Tromer, E., Virza, M.: SNARKs for C: verifying program executions succinctly and in zero knowledge. In: Canetti, R., Garay, J.A. (eds.) CRYPTO 2013, Part II. LNCS, vol. 8043, pp. 90–108. Springer, Heidelberg (2013)
8. Ben-Sasson, E., Chiesa, A., Tromer, E., Virza, M.: Scalable zero knowledge via cycles of elliptic curves. In: Garay, J.A., Gennaro, R. (eds.) CRYPTO 2014, Part II. LNCS, vol. 8617, pp. 276–294. Springer, Heidelberg (2014). http://eprint.iacr.org/2014/595
9. Ben-Sasson, E., Chiesa, A., Tromer, E., Virza, M.: Succinct non-interactive zero knowledge for a von neumann architecture. In: USENIX Security 2014 (2014). http://eprint.iacr.org/2013/879
10. Benabbas, S., Gennaro, R., Vahlis, Y.: Verifiable delegation of computation over large datasets. In: Rogaway, P. (ed.) CRYPTO 2011. LNCS, vol. 6841, pp. 111–131. Springer, Heidelberg (2011)
11. Bitansky, N., Canetti, R., Chiesa, A., Goldwasser, S., Lin, H., Rubinstein, A., Tromer, E.: The hunting of the SNARK. ePrint 2014/580 (2014)
12. Bitansky, N., Canetti, R., Chiesa, A., Tromer, E.: From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again. In: ITCS 2012 (2012)

13. Bitansky, N., Canetti, R., Chiesa, A., Tromer, E.: Recursive composition and bootstrapping for SNARKs and proof-carrying data. In: STOC 2013 (2013)
14. Bitansky, N., Chiesa, A.: Succinct arguments from multi-prover interactive proofs and their efficiency benefits. In: Safavi-Naini, R., Canetti, R. (eds.) CRYPTO 2012. LNCS, vol. 7417, pp. 255–272. Springer, Heidelberg (2012)
15. Bitansky, N., Chiesa, A., Ishai, Y., Ostrovsky, R., Paneth, O.: Succinct non-interactive arguments via linear interactive proofs. In: Sahai, A. (ed.) TCC 2013. LNCS, vol. 7785, pp. 315–333. Springer, Heidelberg (2013)
16. Blum, M., De Santis, A., Micali, S., Persiano, G.: Non-interactive zero-knowledge. SIAM J. Comp. (1991)
17. Blum, M., Feldman, P., Micali, S.: Non-interactive zero-knowledge and its applications. In: STOC 1988 (1988)
18. Blumberg, A.J., Thaler, J., Vu, V., Walfish, M.: Verifiable computation using multiple provers. ePrint 2014/846 (2014)
19. Boneh, D., Segev, G., Waters, B.: Targeted malleability: homomorphic encryption for restricted computations. In: ITCS 2012 (2012)
20. Brants, T., Popat, A.C., Xu, P., Och, F.J., Dean, J.: Large language models in machine translation. In: EMNLP-CoNLL 2007 (2007)
21. Braun, B., Feldman, A.J., Ren, Z., Setty, S., Blumberg, A.J., Walfish, M.: Verifying computations with state. In: SOSP 2013 (2013)
22. Canetti, R., Riva, B., Rothblum, G.N.: Two protocols for delegation of computation. In: Smith, A. (ed.) ICITS 2012. LNCS, vol. 7412, pp. 37–61. Springer, Heidelberg (2012)
23. Chase, M., Kohlweiss, M., Lysyanskaya, A., Meiklejohn, S.: Succinct malleable NIZKs and an application to compact shuffles. In: Sahai, A. (ed.) TCC 2013. LNCS, vol. 7785, pp. 100–119. Springer, Heidelberg (2013)
24. Chiesa, A., Tromer, E.: Proof-carrying data and hearsay arguments from signature cards. In: ICS 2010 (2010)
25. Chiesa, A., Tromer, E.: Proof-carrying data: Secure computation on untrusted platforms (high-level description). The Next Wave: The National Security Agency's review of emerging technologies (2012)
26. Chu, C., Kim, S.K., Lin, Y., Yu, Y., Bradski, G.R., Ng, A.Y., Olukotun, K.: MapReduce for machine learning on multicore. In: NIPS 2004 (2006)
27. Chung, K.-M., Kalai, Y., Vadhan, S.: Improved delegation of computation using fully homomorphic encryption. In: Rabin, T. (ed.) CRYPTO 2010. LNCS, vol. 6223, pp. 483–501. Springer, Heidelberg (2010)
28. Cormode, G., Mitzenmacher, M., Thaler, J.: Practical verified computation with streaming interactive proofs. In: ITCS 2012 (2012)
29. Cormode, G., Thaler, J., Yi, K.: Verifying computations with streaming interactive proofs. In: Proceedings of the VLDB Endowment (2011)
30. Costello, C., Fournet, C., Howell, J., Kohlweiss, M., Kreuter, B., Naehrig, M., Parno, B., Zahur, S.: Geppetto: Versatile verifiable computation. ePrint 2014/976 (2014)
31. Damgård, I.: Towards practical public key systems secure against chosen ciphertext attacks. In: Feigenbaum, J. (ed.) CRYPTO 1991. LNCS, vol. 576, pp. 445–456. Springer, Heidelberg (1992)
32. Damgård, I., Faust, S., Hazay, C.: Secure two-party computation with low communication. In: Cramer, R. (ed.) TCC 2012. LNCS, vol. 7194, pp. 54–74. Springer, Heidelberg (2012)
33. Danezis, G., Fournet, C., Groth, J., Kohlweiss, M.: Square span programs with applications to succinct NIZK arguments. In: Sarkar, P., Iwata, T. (eds.) ASIACRYPT 2014. LNCS, vol. 8873, pp. 532–550. Springer, Heidelberg (2014)

34. Danezis, G., Fournet, C., Kohlweiss, M., Parno, B.: Pinocchio coin: building zerocoin from a succinct pairing-based proof system. In: PETShop 2013 (2013)
35. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. In: OSDI 2014 (2004)
36. Di Crescenzo, G., Lipmaa, H.: Succinct NP proofs from an extractability assumption. In: Beckmann, A., Dimitracopoulos, C., Löwe, B. (eds.) CiE 2008. LNCS, vol. 5028, pp. 175–185. Springer, Heidelberg (2008)
37. Diffie, W., Hellman, M.: New directions in cryptography. IEEE Trans. on Inf. Theory (1976)
38. Dyer, C., Cordova, A., Mont, A., Lin, J.: Fast, easy, and cheap: construction of statistical machine translation models with MapReduce. In: StatMT 2008 (2008)
39. Fauzi, P., Lipmaa, H., Zhang, B.: Efficient modular NIZK arguments from shift and product. In: Abdalla, M., Nita-Rotaru, C., Dahab, R. (eds.) CANS 2013. LNCS, vol. 8257, pp. 92–121. Springer, Heidelberg (2013)
40. Fiore, D., Gennaro, R.: Publicly verifiable delegation of large polynomials and matrix computations, with applications. ePrint 2012/281 (2012)
41. Fredrikson, M., Livshits, B.: Zø: an optimizing distributing zero-knowledge compiler. In: USENIX Security 2014 (2014)
42. Gennaro, R., Gentry, C., Parno, B.: Non-interactive verifiable computing: outsourcing computation to untrusted workers. In: Rabin, T. (ed.) CRYPTO 2010. LNCS, vol. 6223, pp. 465–482. Springer, Heidelberg (2010)
43. Gennaro, R., Gentry, C., Parno, B., Raykova, M.: Quadratic span programs and succinct NIZKs without PCPs. In: Johansson, T., Nguyen, P.Q. (eds.) EUROCRYPT 2013. LNCS, vol. 7881, pp. 626–645. Springer, Heidelberg (2013)
44. Gentry, C., Wichs, D.: Separating succinct non-interactive arguments from all falsifiable assumptions. In: STOC 2011 (2011)
45. Goel, A., Munagala, K.: Complexity measures for Map-Reduce, and comparison to parallel computing. ArXiv abs/1211.6526 (2012)
46. Goldwasser, S., Kalai, Y.T., Rothblum, G.N.: Delegating computation: interactive proofs for muggles. In: STOC 2008 (2008)
47. Goldwasser, S., Lin, H., Rubinstein, A.: Delegation of computation without rejection problem from designated verifier CS-proofs. ePrint 2011/456 (2011)
48. Goldwasser, S., Micali, S., Rackoff, C.: The knowledge complexity of interactive proof systems. SIAM J. Comp. (1989)
49. Groth, J.: Short pairing-based non-interactive zero-knowledge arguments. In: Abe, M. (ed.) ASIACRYPT 2010. LNCS, vol. 6477, pp. 321–340. Springer, Heidelberg (2010)
50. Hada, S., Tanaka, T.: On the existence of 3-round zero-knowledge protocols. In: Krawczyk, H. (ed.) CRYPTO 1998. LNCS, vol. 1462, pp. 408–423. Springer, Heidelberg (1998)
51. Kalai, Y.T., Raz, R.: Probabilistically checkable arguments. In: Halevi, S. (ed.) CRYPTO 2009. LNCS, vol. 5677, pp. 143–159. Springer, Heidelberg (2009)
52. Kang, U., Chau, D.H., Faloutsos, C.: Pegasus: mining billion-scale graphs in the cloud. In: ICASSP 2012 (2012)
53. Kosba, A.E., Papadopoulos, D., Papamanthou, C., Sayed, M.F., Shi, E., Triandopoulos, N.: TRUESET: faster verifiable set computations. In: USENIX Security 2014 (2014)
54. Langmead, B., Schatz, M.C., Lin, J., Pop, M., Salzberg, S.: Searching for SNPs with cloud computing. Genome Biology (2009)
55. Lidl, R., Niederreiter, H.: Finite Fields. Cambridge University Press, second (edn.) (1997)

56. Lin, J.: Brute force and indexed approaches to pairwise document similarity comparisons with mapreduce. In: SIGIR 2009 (2009)
57. Lin, J., Dyer, C.: Data-Intensive Text Processing with MapReduce. Morgan and Claypool Publishers (2010)
58. Lin, J., Schatz, M.C.: Design patterns for efficient graph algorithms in mapreduce. In: MLG 2010 (2010)
59. Lipmaa, H.: Progression-free sets and sublinear pairing-based non-interactive zero-knowledge arguments. In: Cramer, R. (ed.) TCC 2012. LNCS, vol. 7194, pp. 169–189. Springer, Heidelberg (2012)
60. Lipmaa, H.: Succinct non-interactive zero knowledge arguments from span programs and linear error-correcting codes. In: Sako, K., Sarkar, P. (eds.) ASIACRYPT 2013, Part I. LNCS, vol. 8269, pp. 41–60. Springer, Heidelberg (2013)
61. Lipmaa, H.: Efficient NIZK arguments via parallel verification of benes networks. In: Abdalla, M., De Prisco, R. (eds.) SCN 2014. LNCS, vol. 8642, pp. 416–434. Springer, Heidelberg (2014)
62. Merkle, R.C.: A certified digital signature. In: Brassard, G. (ed.) CRYPTO 1989. LNCS, vol. 435, pp. 218–238. Springer, Heidelberg (1990)
63. Micali, S.: Computationally sound proofs. SIAM J. Comp. (2000)
64. Mie, T.: Polylogarithmic two-round argument systems. Journal of Mathematical Cryptology (2008)
65. Miers, I., Garman, C., Green, M., Rubin, A.D.: Zerocoin: anonymous distributed e-cash from bitcoin. In: SP 2013 (2013)
66. Naor, M., Yung, M.: Public-key cryptosystems provably secure against chosen ciphertext attacks. In: STOC 1990 (1990)
67. Panda, B., Herbach, J., Basu, S., Bayardo, R.J.: PLANET: massively parallel learning of tree ensembles with MapReduce. In: Proceedings of the VLDB Endowment (2009)
68. Paneth, O., Rothblum, G.N.: Publicly verifiable non-interactive arguments for delegating computation. ePrint 2014/981 (2014)
69. Parno, B., Gentry, C., Howell, J., Raykova, M.: Pinocchio: nearly practical verifiable computation. In: Oakland 2013 (2013)
70. Pino, J., Waite, A., Byrne, W.: Simple and efficient model filtering in statistical machine translation. Prague Bulletin of Mathematical Linguistics (2012)
71. Schatz, M.C.: CloudBurst: highly sensitive read mapping with MapReduce. Bioinformatics (2009)
72. SCIPR Lab. libsnark: a C++ library for zkSNARK proofs
73. Setty, S., Blumberg, A.J., Walfish, M.: Toward practical and unconditional verification of remote computations. In: HotOS 2011 (2011)
74. Setty, S., Braun, B., Vu, V., Blumberg, A.J., Parno, B., Walfish, M.: Resolving the conflict between generality and plausibility in verified computation. In: EuroSys 2013 (2013)
75. Setty, S., McPherson, M., Blumberg, A.J., Walfish, M.: Making argument systems for outsourced computation practical (sometimes). In: NDSS 2012 (2012)
76. Setty, S., Vu, V., Panpalia, N., Braun, B., Blumberg, A.J., Walfish, M.: Taking proof-based verified computation a few steps closer to practicality. In: USENIX Security 2012 (2012)
77. Thaler, J.: Time-optimal interactive proofs for circuit evaluation. In: Canetti, R., Garay, J.A. (eds.) CRYPTO 2013, Part II. LNCS, vol. 8043, pp. 71–89. Springer, Heidelberg (2013)
78. Thaler, J., Roberts, M., Mitzenmacher, M., Pfister, H.: Verifiable computation with massively parallel interactive proofs. CoRR (2012)

79. Valiant, P.: Incrementally verifiable computation or proofs of knowledge imply time/space efficiency. In: Canetti, R. (ed.) TCC 2008. LNCS, vol. 4948, pp. 1–18. Springer, Heidelberg (2008)
80. Vu, V., Setty, S., Blumberg, A.J., Walfish, M.: A hybrid architecture for interactive verifiable computation. In: Oakland 2013 (2013)
81. Wahby, R.S., Setty, S., Ren, Z., Blumberg, A.J., Walfish, M.: Efficient RAM and control flow in verifiable outsourced computation. ePrint 2014/674 (2014)
82. Wolfe, J., Haghighi, A., Klein, D.: Fully distributed EM for very large datasets. In: ICML 2008 (2008)
83. Zhang, Y., Papamanthou, C., Katz, J.: Alitheia: towards practical verifiable graph processing. In: CCS 2014 (2014)