Proof-Carrying Data

by

Alessandro Chiesa

S.B., Mathematics (2009)
S.B., Computer Science and Engineering (2009)
Massachusetts Institute of Technology

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June, 2010

Author . . . .
Department of Electrical Engineering and Computer Science
May 21, 2010

Certified by . . . .
Ronald L. Rivest
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Certified by . . . .
Eran Tromer
Postdoctoral Associate
Thesis Supervisor

Accepted by . . . .
Christopher J. Terman
Chairman, Department Committee on Graduate Students

Proof-Carrying Data
by
Alessandro Chiesa

Submitted to the
Department of Electrical Engineering and Computer Science

May 21, 2010

In Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

The security of systems can often be expressed as ensuring that some property is maintained at every step of a distributed computation conducted by untrusted parties. Special cases include integrity of programs running on untrusted platforms, various forms of confidentiality and side-channel resilience, and domain-specific invariants.

We propose a new approach, *proof-carrying data* (PCD), which sidesteps the threat of faults and leakage by reasoning about properties of a computation's output *data*, regardless of the *process* that produced it. In PCD, the system designer prescribes the desired properties of a computation's outputs. Corresponding proofs are attached to every message flowing through the system, and are mutually verified by the system's components. Each such proof attests that the message's data *and all of its history* comply with the prescribed properties.

We construct a general protocol compiler that generates, propagates, and verifies such proofs of compliance, while preserving the dynamics and efficiency of the original computation. Our main technical tool is the cryptographic construction of short non-interactive arguments (computationally-sound proofs) for statements whose truth depends on "hearsay evidence": previous arguments about other statements. To this end, we attain a particularly strong proof-of-knowledge property.

We realize the above, under standard cryptographic assumptions, in a model where the prover has black-box access to some simple functionality — essentially, a signature card.

Thesis Supervisor: Ronald L. Rivest
Title: Professor of Electrical Engineering and Computer Science

Thesis Supervisor: Eran Tromer
Title: Postdoctoral Associate

Alla memoria del mio caro Papà,
alla mia meravigliosa Mamma,
e a Silvia e Marco.

Acknowledgments

This research was carried out in the Computer Science and Artificial Intelligence Laboratory at MIT, under the supervision of Professor Ronald L. Rivest and Doctor Eran Tromer, and was supported by NSF grant NSF-CNS-0808907 and AFRL grant FA8750-08-1-0088.

I would like to thank Eran Tromer for being an amazing advisor. Eran's guidance and teaching profoundly contributed to my academic growth throughout my first major research endeavor.

I am indebted to Ron Rivest for providing his valuable insight and support during this investigation.

I am grateful to Silvio Micali for teaching me cryptography; Silvio's superb lectures truly ignited my passion for the field. Since then Silvio has remained an encouraging mentor.

My beloved girlfriend Raluca Ada Popa was a constant source of support, both emotional and technical; even in the darkest times, she managed to make me see light again.

My years at MIT would not have been as exciting, intense, and full of personal and technical growth had it not been for my best friend Andre Yohannes Wibisono. Since freshman year, Andre was always the best person to talk to about my most recent worry or the most recent problem set in one of our classes.

Scott Aaronson, Andrew Drucker, and Paul Valiant provided valuable pointers about the PCP vs. oracle difficulty. Boaz Barak, Arnab Bhattacharyya, and Or Meir helped in the evaluation of argument systems and the underlying PCPs. Stephen Chong, Fred Schneider, Greg Morrisett, and Jeff Vaughan shared their perspective on applications of PCD in type safety. I would also like to thank Shafi Goldwasser, Frans Kaashoek, Nancy Lynch, and Nickolai Zeldovich for valuable feedback.

Overview

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Proof systems lie at the heart of modern cryptography and complexity theory; they have demonstrated tremendous expressive power and flexibility, yielding both surprising theoretical results and finding powerful applications. In this thesis, we give evidence that, when three particular properties of proof systems come together (low communication complexity, non-interactivity, and aggregability of proofs), proof systems acquire a new level of expressiveness and flexibility.

Our main technical contribution is a proof system with these three properties. Specifically, we realize short non-interactive computationally-sound proofs for statements whose truth depends on "hearsay evidence" (previous proofs of other statements), under standard cryptographic assumptions, in a model where the prover has black-box access to some simple functionality. We call a proof system with these properties an *assisted-prover hearsay argument*. The main ingredient of the construction is achieving a particularly strong proof-of-knowledge property in a model that allows for "aggregation of proofs".

As the main implication, we show how assisted-prover hearsay arguments imply the existence of a protocol compiler that can "magically" enforce invariant properties on a distributed computation. The given invariant is encoded as an efficiently computable predicate called the *compliance predicate*, which takes as input a party's incoming messages, local program, and outgoing messages; the predicate is required to hold for every party that takes part in the distributed computation. The protocol compiler enforces the invariant by enabling parties to generate and attach to each message flowing through the computation a concise proof attesting to the message's "compliance"; these proofs are dynamically composed and propagated, while preserving the dynamics and efficiency of the original distributed computation. We call our protocol compiler a *proof-carrying data system*.

Finally, we argue that a proof-carrying data system enables a new solution approach to solving problems in security. We observe that the security of systems can often be expressed as ensuring that some property is maintained at every step of a distributed computation conducted by untrusted parties. Special cases include integrity of programs running on untrusted platforms, various forms of confidentiality and side-channel resilience, and domain-specific invariants. We thus propose a new approach, *proof-carrying data*, which sidesteps the threat of faults and leakage by reasoning about properties of a computation's output *data*, regardless of the *process* that produced it. The system designer prescribes the desired properties of a computation's outputs by specifying the compliance predicate for the proof-carrying data system. Then, corresponding proofs are attached to every message flowing through the system, and are mutually verified by the system's components. Each such proof attests that the message's data *and all of its history* comply with the prescribed properties.

1.1 Motivation

An important motivation for our work is exploring the power of proof systems. We show that, under plausible cryptographic and setup assumptions, *efficient distributed theorem proving* is possible. Roughly, our results imply that every NP statement has a very concise proof string for its correctness, and this proof string can be used "at no cost" as a "lemma" in a concise proof string for any other NP statement (which did not have to be known when the proof string to the first NP statement was generated). As a result, we learn that proof systems for NP can be essentially as efficient and as flexible as one could hope for. Further details about this interpretation of our results can be found in Chapter 6.

However, the primary motivation for our work originates from its main application: using a proof system to provide *a new solution approach to security problems* where the idea is to enforce an invariant through a distributed computation.

Indeed, security in distributed systems typically requires maintaining properties across the computation of multiple, potentially malicious, parties. Even when human participants are honest, the computational devices they use may be faulty (due to bugs or transient errors [22]), leaky (e.g., suffering from covert and side channels [99]) or adversarial (e.g., due to components from untrusted sources [23]).

Let us consider a few examples of security properties whose attainment, in the general case and under minimal assumptions, is a major open problem — and how they can be approached using our framework of *proof-carrying data* (PCD).

- **Integrity.** Consider parties engaged in a distributed computation. Each party receives input messages from other parties, executes some program on his own local inputs and the input messages, and then produces some output messages to be sent out to other parties.

  Can we obtain evidence that the distributed computation's final output is indeed the result of correctly following a prescribed program in the aforementioned process? For example, if the computation consists of a physics simulation (whether realistic or that of an online virtual world), can we obtain evidence that all parties have "obeyed the laws of physics"?

- **Information flow control.** Confidentiality and privacy are typically expressed as negative conditions forbidding certain effects. However, following the approach of information flow control (IFC) [47][111], one may instead reason about what computation is *allowed* and on what inputs.

  Thus, within a distributed computation, we can define the security property of intermediate results as being "consistent with a distributed computation that follows the IFC rules". In IFC, intermediate results are labeled according to their confidentiality; PCD augments these with a proof string attesting to the validity of the label. Ultimately, a censor at the system perimeter lets through only the "non-secret" outputs, by verifying their associated label and proof string. Because verification inspects only the (augmented) output, it is inherently unaffected by anomalies (faults and leakage) in the preceding computation; only the censor needs to be trusted to properly verify proof strings.

- **Fault isolation and accountability.** Consider a distributed system consisting of numerous unreliable components. Let any communication across component boundaries carry a concise proof of correctness, and let each component verify the proofs of its inputs and generate proofs for its outputs. Whenever verification of a proof fails, the computation is locally aborted and outputs a proof of the wrongdoing. Damage is thus controlled and attributed. In principle this may be realized at any scale, from individual chips to whole organizational units.

Many applications involve multiple such goals. For example, in *cloud computing*, clients are typically interested in both integrity [77] and confidentiality [125]. Further details and proposed examples appear in Chapter 6.

Thus, the framework of proof-carrying data is based on augmenting every message passed in the distributed computation with a short proof string attesting to the fact that the message's data, along with all of the distributed computation leading to that message, satisfies the desired property. These proofs are efficiently produced, verified and aggregated at every node. Ultimately, the proof string attached to the system's final output attests that the whole computation satisfied the desired property.

1.2 Goals

Generalizing the discussion in the previous section, we address the general problem of *secure distributed computation when all parties are mutually untrusting and potentially malicious.* Computation may be dynamic and interactive, and "secure" may be any property that is expressible as a predicate that efficiently checks each party's actions.

We thus wish to construct a compiler that, given a protocol for a distributed computation, and a security property (in the form of a predicate to be verified at every node of the computation), yields an augmented protocol that enforces the security property. We wish this compiler to *respect the original distributed computation*, i.e., it should preserve communication, dynamics and efficiency:

- **Preserve the communication graph.** Parties should not be required to engage in additional communication channels beyond those of the original distributed computation. For example: protecting the distributed computation carried out by a system of hardware components should not require each chip to continuously communicate with all other chips; agents executing in the "cloud" should remain trustworthy even when their owners are offline; and parties should be able to conduct joint computation on a remote island and later re-join a larger multi-party computation.

– **Allow dynamic computations.** The compiler should allow for inputs that are provided on the fly (e.g., determined by human interaction, random processes, or nondeterministic choices).

– **Minimize the blowup in communication and computation.** The induced overhead in communication between parties, and computation within parties, should be kept at a minimum (e.g., at most a local polynomial blowup).

The above properties imply, for example, that *scalability* is preserved: if the original computation can be jointly conducted by numerous parties, then the compiler produces a secure distributed computation that has the same property.

1.3 Our approach

Use a proof system. In our approach, *proof-carrying data*, every piece of data flowing through a distributed computation is augmented by a short proof string that certifies the data as compliant with some desired property. These proofs can be propagated and aggregated "on the fly", as the computation proceeds.

Let us illustrate our approach by a simple scenario. Alice has some input x and a function F. She computes $y := F(x)$ at a great expense, along with a proof string π_y for the claim "$y = F(x)$", and then publishes the pair ("$y = F(x)$", π_y) on her webpage. A week later, Bob comes across Alice's webpage, notices the usefulness of y, and wants to use it as part of his computations: he picks a function G and computes $z := G(y)$. To convince others that the combined result is correct, Bob also generates a new proof string π_z for the claim "$z = G(F(x))$", using both the transcript of his own computation of G on y, and Alice's proof string π_y. (See Figure 1-1 for a diagram.) Crucially, Bob does not have to recompute $F(x)$. The size of π_z is merely polylogarithmic in Bob's own work (i.e., the time to compute G on y and the size of the statement "$z = G(F(x))$"), and is essentially independent of the past work by Alice.

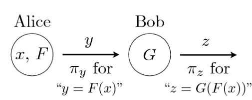

Figure 1-1: The "F and G" example.

We generalize the above scenario to any distributed computation. Also, we generalize "correctness" to be any property that should hold at every node of the computation. More precisely, we consider properties that can be expressed as a requirement that every step in the computation satisfies some *compliance predicate* $\mathbf{C}$ computable in polynomial time; we call this notion $\mathbf{C}$-*compliance*. Thus, each party receives inputs that are augmented with proof strings, computes some outputs, and augments each of the outputs with a new proof string that will convince the next party (or the verifier of the ultimate output) that the output is consistent with a $\mathbf{C}$-compliant computation. We stress that each party does not have to commit in advance to what computation it will have to perform; rather, parties may decide what do to in real time (as if they were not computing any proof strings) and will generate proof strings based on their decision of what computation to perform. See Figure 1-2 for a high-level diagram of this idea.[1] We

[1] In addition, we obtain a proof-of-knowledge property (see Goldreich [60, Sec. 4.7] for the definition), which implies that not only does there *exist* a $\mathbf{C}$-compliant computation consistent with the output, but moreover this computation was

thus define and construct a *proof-carrying data (PCD) system* primitive that fully encapsulates the proof system machinery, and provides a simple but very general "interface" to be used in applications.

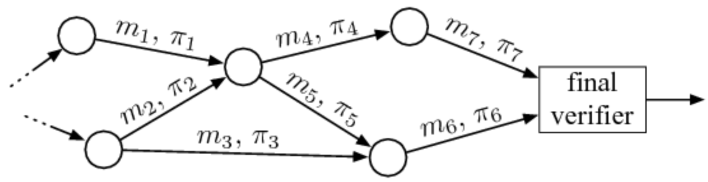

Figure 1-2: A distributed computation in which parties send messages m_i that are augmented by proof strings π_i. The distributed computation does not have to be known in advance, i.e., parties may decide (possibly based on their input messages) which computation to perform and to whom send messages — in other words, the graph representing the distributed computation through time is allowed to be completely dynamic.

PCD generalizes the "incrementally-verifiable computation" of Valiant [134]. The latter compiles a (possibly super-polynomial-time) machine into a new machine that always maintains a proof for the correctness of its internal state. PCD extends this in several essential ways: allowing for the computation to be dynamic (interactive and nondeterministic); allowing for multiple parties and arbitrary communication graphs; and allowing for an arbitrary compliance predicate, instead of considering only the special case of correctness. These greatly expand expressibility, but entail significant technical challenges (for example, dynamic computation forces us to recursively aggregate proofs in polynomially-long chains, instead of the logarithmically-deep trees of [134], and this requires a much stronger knowledge extractor). Crucially, our construction circumvents a major barrier which precluded a satisfying proof of security even for the simpler functionality of incrementally verifiable computation.[2]

Construction and tools. Our main technical tool, potentially of independent interest, is *assisted-prover hearsay-argument (APHA) systems*. These are short non-interactive arguments (computationally-sound proofs) for statements whose truth depends on "hearsay evidence" from previous arguments, in the sense of the above "F and G" example. As pointed out by Valiant [134], this is not implied by standard soundness: the latter merely says that if the verifier for a statement "$z = G(F(x))$" is convinced then there exists a witness for that statement. But if the witness is supposed to contain a proof string π_y for another statement $y = F(x)$, the mere *existence* of π_y (that would be accepted by the verifier) is useless: such π_y may exist regardless of the truth of the statement "$y = F(x)$", since the soundness of the argument is merely computational. We actually need to show that if the proof string for "$z = G(F(x))$" was generated efficiently, then a valid proof string for "$y = F(x)$" can be generated with essentially the same efficiency (and acceptance probability) and is thus also convincing. Technically, this is captured by a particularly strong proof-of-knowledge property.

Our construction of APHA systems is built on argument systems [76][26]. Specifically, we use universal arguments [15] which (following [92] and computationally-sound proofs [107]) invoke the PCP theorem [12] to achieve concise proofs and efficient verification.

However, such argument systems do not by themselves suffice: where they offer a strong proof-of-knowledge property [53][134], they do so by relying on random oracles, which precludes nesting of proofs since the underlying PCP system does not relativize [54][33]. Even in the restricted case of incrementally-verifiable computation [134], this difficulty precluded a satisfying proof of security.

We address this problem, both in general and for the special case of [134], by extending the model with a new assumption: an oracle that is invoked by the prover, but not by the verifier. The former facilitates knowledge extraction, while the latter allows for aggregation of proof strings. The oracle provides a simple *signed-input-and-randomness* functionality: for every invocation, it augments the input x with some fresh randomness r, and outputs r along with a signature on (x, r) under a secret key sk embedded in the oracle. This is discussed next.

---

actually "known" to whoever produced the proof. This is essential for applications that employ cryptographic functionality that is secure only against computationally-bounded adversaries, since an efficient cheating prover can only "know" efficient $\mathbf{C}$-compliant computation.

[2] Valiant [134] offers two constructions: one that assumes the existence of a cryptographic primitive that is nonstandard and arguably implausible [134, Theorem 1], and one whose overall security is conjectured directly without any reduction [134, Sec. 1.3 under "The Noninteractive CS Knowledge Assumption"]. The difficulty seems inherent; see Section 4.1. In our model, we attain provable security under standard generic cryptographic assumptions.

1.4 Model and trust

We assume that all parties have black-box access to the aforementioned signed-input-and-randomness functionality. Concretely, we think of this oracle as realized by hardware tokens, such as existing signature cards, TPM chips or smartcards. It can also be implemented by a trusted Internet service (see [35] for a demonstration). Alternative realizations include obfuscation and multi-party computation; see Section 4.5 for further discussion.

Comparable assumptions have been used in previous works, as setup assumptions to achieve universally-composable functionality that is otherwise impossible [29]. In this context, Hofheinz et al. [83] assume signature cards similar to ours. The main differences in the requisite functionality is that we require the card to generate random strings and include them in its output and signature (a pseudorandom function suffices — see Section 4.5), and to use signature schemes where the signature length can only depend on the security parameter (see Section 2.3.5).

The more general result of Katz [90] assumes that parties can embed functionality of their choice in secure tokens and send it to each other; follow-up works in similar models include [110][32][45]. However, in our case we cannot afford a model where parties generate tokens and send them to all other parties, since this does not preserve the communication graph of the original computation. Thus, our model is closer to that of [83].

For simplicity, we assume the following setup and trust model. A trusted party generates a signature key pair $(\mathsf{vk}, \mathsf{sk})$ and many signed-input-and-randomness tokens containing sk. Each party is told vk and receives a token. All parties trust the manufacturer and the tokens, in the sense that each party, upon seeing a signature on some (x, r) that verifies under vk, believes that the signature was produced by some token queried on $(x, |r|)$.

One can easily adapt this to a certificate-authority model where each token uses its own secret key sk, and publishes the corresponding public key vk along with a certificate for vk (i.e., a signature under the key of a trusted certificate authority).[3]

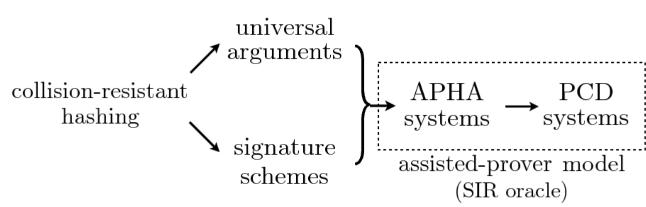

Figure 1-3: Collision-resistant hashing schemes imply public-coin constant-round universal arguments and secure concise signature schemes. From these two, we derive APHA systems, and then PCD systems, in the model where provers are assisted with signed-input-and-randomness (SIR) oracles.

1.5 Contributions of this thesis

In summary, we present the following results, discussed in the indicated chapters:

[3]Technically, this variant is realized by tweaking the PCD machine of Section 5.3 to verify the authority's signature on this vk.

An argument system for hearsay. (Chapter 4) We define *assisted-prover hearsay-argument (APHA) systems*: non-interactive arguments for NP that can efficiently prove statements that recursively rely on earlier APHA proof strings, using a very strong proof-of-knowledge property. We construct these in a model where the prover has black-box access to a simple stateless functionality, namely signing (under a secret key) every input along with fresh randomness. Our construction relies on public-coin constant-round universal arguments [15] and concise[4] secure signature schemes. Both exist under the standard generic assumption that collision-resistant hashing schemes exist. See Figure 1-3.

Distributed computations and proof-carrying data. (Chapter 5) We propose proof-carrying data (PCD) as a framework for expressing and enforcing security properties, and formally define *proof-carrying data (PCD) systems* that capture the requisite protocol compiler and computationally-sound proof system. We construct this primitive under the same assumptions as above (in fact, we present a *generic* transformation from APHA systems to PCD systems). See Figure 1-3.

Proposed applications. (Chapter 6) We discuss a number of open problems in the security of real-world applications, where PCD potentially offers a powerful solution approach by circumventing current difficulties.

Also, in Chapter 2 we introduce the basic notation, notions, and cryptographic primitives used throughout this thesis, and in Chapter 3 we discuss prior work relating to our main technical tool, i.e., proof systems.

1.6 Previous approaches to security design

Security design has been approached from a variety of perspectives. We review here the most important of those approaches.

Distributed algorithms. Distributed algorithms [104] typically address achieving *specific* properties of a *global* nature (e.g., consensus). By contrast, we offer a *general* protocol compiler for ensuring *local* properties of individual steps in the distributed computation. In this sense the problems are complementary. Indeed, trusted tokens turn out to be a powerful tool for global properties as well, as shown by A2M [37] and TrInc [101].

Platforms, languages, and static analysis. Integrity can be achieved by running on suitable fault-tolerant systems. Confidentiality can be achieved by platforms with suitable information flow control mechanisms [47][111], e.g., at the operating-system level [98][136]. Various invariants can be achieved by statically analyzing programs, and by programming language mechanisms such as type systems [122][3][46].

The inherent limitations of these approaches (beside their difficulty) is that the output of such computation can be trusted only if one trusts the whole platform that executed it; this renders them ineffective in the setting of mutually-untrusting distributed parties.

Cryptographic approaches. Proof systems with certain properties can be used to achieve a variety of goals in security design. We discuss proof systems at length in Chapter 3, as the results in this thesis are attained through tools that fall in this category.

Proof-carrying code. Proof-carrying code (PCC) [114][123, Ch. 5] addresses scenarios in which a host wishes to execute code received from untrusted producers, and would like to ascertain that the code adheres to some rules (e.g., because the execution environment is not inherently confining). In the PCC approach, the producer augments the code with formal, efficiently-checkable proofs of the desired properties — typically, using the aforementioned language or static analysis techniques. Such systems have been built for scenarios such as packet filter code [115], mobile agents [116] and compiled Java programs [39].

Indeed, PCC inspired the name of our approach, "proof-carrying data" (PCD). The difference lies in that we reason about properties of *data*, as opposed to properties of *code*. PCC and PCD thus address disjoint scenarios, by different techniques (see Table 1.1 for a summary). However, the two approaches can be composed: a potentially powerful way to express security properties is to require messages to be correctly produced by some program prg that has desired properties (e.g., type safety), and then prove

[4] The signature length should depend only on the security parameter.

these properties of prg using proof-carrying code. Here, the PCD compliance predicate $\mathbf{C}$ consists of running the PCC verifier on prg and then executing prg.

	Proof-carrying data	**Proof-carrying code**
Message	data	executable code
Statement about	specific past history	all future executions
Proof method	cryptography + compliance predicate	formal methods
Main computation executed by	prover (sender)	verifier (host)
Recursively aggregatable	yes	n/a

Table 1.1: Comparison between proof-carrying data and proof-carrying code.

Dynamic analysis. Dynamic analysis monitors the properties of a program's execution at run time (e.g., [117][131][95]). Our approach can be interpreted as extending dynamic analysis to the distributed setting, by allowing parties to (implicitly) monitor the program execution of all prior parties without actually being present during the executions.

Fabric. The Fabric system [102] is similar to PCD in motivation, but takes a very different approach. Fabric addresses execution in a network of nodes which have *partial* trust in each other. Nodes express their information flow and trust policies, and the Fabric platform (through a combination of static and runtime techniques) ensures that computation and data will be delegated across nodes only when requisite trust relations exist for preserving the information flow policy. Thus, Fabric is a practical system that allows "as much delegation as we are sure is safe" across a system of partially-trusting nodes (where a violated trust relation will undermine security). In contrast, PCD allows (somewhat different) security properties to be preserved across an arbitrary network of fully-mistrustful nodes, but with a much higher overhead.

Chapter 2

Preliminaries

We review basic notation, notions, and cryptographic primitives. An excellent comprehensive treatment of all of these can be found in Goldreich [60] and Goldreich [61].

2.1 Basic notation

We use standard notation for functions, algorithms, and machines, as summarized in the next few subsections.

2.1.1 Strings, relations, and functions

We take all strings to be over the binary alphabet $\{0,1\}$; the empty string is denoted ϵ. Often we will talk about objects that are not binary strings or tuples of strings, however all are easy to encode in some canonical way as binary strings, so we assume throughout that such an encoding has been fixed. The set of positive integers is denoted $\mathbb{N}$; all integers are presented in binary; an integer k presented in unary will be specially denoted as 1^k; for $n \in \mathbb{N}$, we denote by $[n]$ the set $\{1, \dots, n\}$.

Let $R \subset \{0,1\}^* \times \{0,1\}^*$ be a binary relation. The *language of a relation* R, denoted $L(R)$, is defined to be the set $\{x \in \{0,1\}^* : \exists\, y \text{ s.t. } (x,y) \in R\}$. The *witness set of a string* x *in* R, denoted $R(x)$, is defined to be the set $\{y \in \{0,1\}^* : (x,y) \in R\}$; a string $y \in R(x)$ is called a *witness* for x in R. We say that a relation R is *polynomially bounded* if there exists a positive polynomial such that for all (x,y) it holds that $|y| \leq p(|x|)$. The familiar case of an NP relation R requires that the relation be both polynomially bounded and there exists a (deterministic) polynomial-time algorithm for deciding membership in $L(R)$.

A function f with domain D and co-domain C is denoted $f\colon C \to D$. For a subset S of D, the restriction of f to S is denoted $f|_S$. For an element y in C, the (possibly empty) subset of elements in D that map to y is denoted $f^{-1}(y)$.

Definition 2.1.1 (Function Ensemble)**.** *Let $\ell\colon \mathbb{N} \to \mathbb{N}$. An* ℓ-bit function ensemble *is a sequence of random variables $\mathcal{F} = \{F_k\}_{k \in \mathbb{N}}$, where F_k takes on values that are functions from $\ell(k)$-bit strings to $\ell(k)$-bit strings. The* uniform ℓ-bit function ensemble*, denoted $\mathcal{H} = \{H_k\}_{k \in \mathbb{N}}$ is the ensemble where H_k is uniform over all functions from $\ell(k)$-bit strings to $\ell(k)$-bit strings.*

2.1.2 Distributions

If A is a probabilistic algorithm, then for any input x to A we let $A(x)$ denote the probability space that assigns to any string σ the probability that $A(x)$ outputs σ. If we want to specify the random tape of A, then we will write $A(x;r)$ to denote the output of the (now deterministic) algorithm A on input x and random tape r. If S is a probability space, then $x \leftarrow S$ denotes that x is randomly chosen from S; if S is a finite set, then $x \leftarrow S$ denotes that x is randomly chosen from the set S.

An *experiment* is a probability distribution over strings that are the result of an ordered execution of probabilistic algorithms. For example,

$$\Big\{ (x,y,z) \;\Big|\; x \leftarrow A \;;\; y \leftarrow B(x) \;;\; z \leftarrow C(x,y) \Big\}$$

denotes the probability distribution over the triples (x, y, z) as generated by first running algorithm A to obtain x, then running algorithm B on input x to obtain y, and then running algorithm C on inputs x and y to obtain z. Similarly, for a predicate Π on strings, the probability that the above experiment yields a triple (x, y, z) that satisfies Π is denoted as

$$\Pr\left[\Pi(x,y,z) = 1 \,\middle|\, x \leftarrow A \; ; \; y \leftarrow B(x) \; ; \; z \leftarrow C(x,y)\right] .$$

2.1.3 Computational models

We use the standard notions of Turing machines and circuits. (Details such as how many tapes a Turing machine has or which universal set of gates is allowed in a circuit are inconsequential for the results in this thesis.) If M is a Turing machine, then $\langle M \rangle$ is its description (on occasion identified with M) and $\mathsf{time}_M(x)$ is the time that M takes to halt on input a string x. If C is a circuit, then $\langle C \rangle$ is its representation and $|C|$ is its size. For a more detailed discussion of Turing machines and circuits, see any introductory textbook on complexity theory, such as Papadimitriou [119], Sipser [130], Goldreich [62], or Arora and Barak [5].

An *interactive Turing machine* is a Turing machine that has additional tapes for the purpose of "communicating" with other Turing machines. For a formal definition, see Goldwasser, Micali, and Rackoff [74][76]. More generally, we can consider the notion of interactive *circuits*. For two interactive machines (or circuits) A and B, we denote the output of A, after interacting with B on common input x, and each with private inputs y and z, by $\langle A(y), B(z) \rangle(x)$.

2.1.4 Feasible and infeasible computation

We adopt the standard postulate that feasible computations correspond to those that can be performed by probabilistic polynomial-time Turing machines. Thus, infeasible computations correspond to those that require a super-polynomial amount of probabilistic time to perform.

The adversarial model should then include at least all feasible strategies (i.e., those which are implementable in probabilistic polynomial-time). In this thesis, we will use the stronger (and overly cautious) adversarial model that allows adversaries to be of *non-uniform* polynomial-time, i.e., to be of probabilistic polynomial-time with access to a polynomial-size "advice" string for each input length (these are simply families of polynomial-size circuits).

We will often need to show that certain events "never happen". The standard way to capture that is to require that they occur with a probability that vanishes faster than the inverse of any polynomial; in particular, that implies that these events would "never" be noticed by any polynomial-time strategy. Such a probability, and, more in general, such a function, is said to be *negligible*.

Definition 2.1.2 (Negligible Function)**.** *A function $f : \mathbb{N} \to \mathbb{R}$ is said to be* negligible *if for every positive constant c there exists an k_0 in $\mathbb{N}$ such that for every $k \geq k_0$ it holds that $f(k) < k^{-c}$.*

Functions that are not negligible are said to be *non-negligible*.

Definition 2.1.3 (Non-Negligible Function)**.** *A function $f : \mathbb{N} \to \mathbb{R}$ is said to be* non-negligible *if it is not negligible, i.e., there is a positive constant c such that for infinitely many k in $\mathbb{N}$ it holds that $f(k) \geq k^{-c}$.*

We warn that in some works the terminology "non-negligible" is reserved for a "strong negation" of negligible, i.e., for functions f for which there is a positive constant c such that, *for all sufficiently large* k, $f(k) \geq k^{-c}$; in those works, functions defined as in Definition 2.1.3 are instead called *not negligible*.

2.2 Basic notions

We review basic notions that often appear in cryptography.

2.2.1 Computational indistinguishability

We will often be concerned with asymptotic behavior of random variables.

Definition 2.2.1 (Probability Ensemble). *Let S be a subset of $\{0,1\}^*$. A* probability ensemble indexed by S *is a sequence of random variables $\{X_\sigma\}_{\sigma \in S}$.*

Goldwasser and Micali [72] first suggested (in the context of defining security for encryption schemes) that the notion of equivalence between probability ensembles can be usefully relaxed from a requirement of *equality* to a requirement of *indistinguishability under any probabilistic polynomial-time test.* Yao [135] first independently considered and formulated this notion, which is called *computational indistinguishability.*

Definition 2.2.2 (Computational Indistinguishability). *We say that two ensembles $X = \{X_\sigma\}_{\sigma \in S}$ and $Y = \{Y_\sigma\}_{\sigma \in S}$ are* computationally indistinguishable *if for every family of polynomial-size distinguisher circuits $\{D_k\}_{k \in \mathbb{N}}$, every positive constant c, and all sufficiently large k and every $\sigma \in S \cap \{0,1\}^k$,*

$$\left| \Pr\left[D_k(\alpha) = 1 \;\middle|\; \alpha \leftarrow X_\sigma \right] - \Pr\left[D_k(\alpha) = 1 \;\middle|\; \alpha \leftarrow Y_\sigma \right] \right| < \frac{1}{k^c} \ .$$

We note that the above definition would not have been stronger if we were to provide D_k with the index σ of the distribution being tested; in fact, it would not have been stronger even if we were to consider a different distinguisher D_σ for each index σ. Finally, the above definition refers only to distinguishing distributions by a single sample, but it is equivalent to distinguishing distributions by any polynomial-number of samples [60, Ch. 3, Ex. 9].[1]

We also recall the basic fact that computational indistinguishability is preserved under efficient transformations [60, Ch. 3, Ex. 2].

2.2.2 Black-box subroutines and oracles

We often use a machine A as a *subroutine* of a second machine B: the description of machine B contains the description of machine A in order to enable B to run A on inputs of its choice. In particular, B uses the description of A only for the purpose of running A, and in this case we say that A is a *black-box* subroutine of B, i.e., the functionality of B could have been similarly achieved by restricting B to have only oracle access to the functionality of A.

We note that, in most occasions, results are established with techniques that involve only the use of black-box subroutines. Notable exceptions are the results of Canetti, Goldreich, and Halevi [30][31] and Barak, Goldreich, Impagliazzo, Rudich, Sahai, Vadhan, and Yang [16] who used the code of the adversary to obtain certain negative results, and the results of Barak [13] and Barak [14] who used the code of the adversary as part of a proof of security.

When we want to *enforce* black-box access of a machine B to a certain functionality A, we will say that B is an oracle machine (or circuit) with *oracle access* to A. We denote the transcript of B on input x and oracle access to A by $[\![B(x), A]\!]$. For two (possibly) probabilistic machines or circuits A and B, we use $B^{\uparrow A}$ to denote that B has *black-box rewinding access* to A, i.e., B can run A multiple times, each time choosing A's random tape; moreover, if both A and B are interactive machines, then B can also choose the messages that it sends to A.

2.3 Basic cryptographic primitives

Modern cryptography captures cryptographic tasks and their security requirements as *cryptographic primitives*; these are tuples of algorithms that satisfy certain conditions expressing how these algorithms should be used (i.e., expressing their *functionality*) and what adversarial manipulations are almost impossible to achieve (i.e., expressing their *security guarantees*).

The cryptographic notions defined in this section are defined to be secure against families of polynomial-size circuits, as that is the adversarial model that we consider in this work. (See Section 2.1.4.) Analogous notions definitions can be obtained for different adversarial models (e.g., probabilistic polynomial-time Turing machines, families of subexponential-size circuits).

[1] The same equivalence does not hold when considering computational indistinguishability against *probabilistic polynomial-time Turing machines* [65].

2.3.1 One-way functions

The ability to easily sample hard instances of an efficiently-verifiable problem, along with a solution to the problem instance, is one of the simplest and most basic notions in modern cryptography. Formally, this ability is captured by a cryptographic primitive called a *one-way function* (see Diffie and Hellman [49] and Yao [135] for its origins).

Definition 2.3.1 (One-Way Function)**.** *A function* $f\colon \{0,1\}^* \to \{0,1\}^*$ *is said to be* one-way *if it satisfies the following two conditions:*

1. Easy to compute: *There exists a deterministic polynomial-time evaluator Turing machine* E *such that* $E(x) = f(x)$ *for every string* x*.*
2. Hard to invert: *For every family of polynomial-size inverter circuits* $\{I_k\}_{k\in\mathbb{N}}$*, every positive constant* c*, and for all sufficiently large* k*,*
$$\Pr\left[I_k(y) \in f^{-1}(x) \;\middle|\; x \leftarrow \{0,1\}^k \;;\; y \leftarrow f(x) \right] < \frac{1}{k^c} \ .$$

Additionally, we will say that f is a *one-way permutation* if f is one-way and $f|_{\{0,1\}^k}$ is a permutation of $\{0,1\}^k$ to itself for each k in $\mathbb{N}$.

2.3.2 Pseudo-random generators

The notion of a randomness *amplifier* is what is formally captured by a pseudo-random generator. Loosely speaking, on input a random *seed* that is kept secret, a pseudo-random generator is a *deterministic* algorithm that outputs a longer string that "looks" random: no polynomial-time procedure is able to tell whether a given string is truly random or only a random output of a pseudo-random generator.

Definition 2.3.2 (Pseudo-Random Generator)**.** *A function* $f\colon \{0,1\}^* \to \{0,1\}^*$ *is said to be a* pseudo-random generator *if it satisfies the following two conditions:*

1. Easy to compute: *There exists a deterministic polynomial-time evaluator Turing machine* E *such that* $E(x) = f(x)$ *for every string* x*.*
2. Expansion: *There exists a function* $\ell\colon \mathbb{N} \to \mathbb{N}$ *such that* $\ell(k) > k$ *for all* k *in* $\mathbb{N}$ *and* $|f(x)| = \ell(|x|)$ *for all strings* x*. The function* ℓ *is called the* expansion factor *of* f*.*
3. Pseudo-randomness: *The probability ensembles* $\{f(U_k)\}_{k\in\mathbb{N}}$ *and* $\{U_{\ell(k)}\}_{k\in\mathbb{N}}$ *are computationally indistinguishable.*

Blum and Micali [24] originally considered *unpredictable* sequences of bits, and showed how to construct them assuming the difficulty of taking discrete logarithms. Yao [135] then proved that unpredictable sequences of bits are in fact *pseudo-random* (i.e., are computationally indistinguishable from truly random strings), thus showing that Blum and Micali had in fact constructed a pseudo-random generator; Yao further showed how to construct pseudo-random generators assuming the existence of one-way permutations. His results were later improved, and eventually Håstad, Impagliazzo, Levin, and Luby [84] proved that pseudo-random generators exist if (and only if) one-way functions exist; this last result is not an efficient construction, so most practical pseudo-random generators are based on specific number-theoretic assumptions and follow the paradigm of Yao's construction. For more details on pseudo-random generators, see Goldreich [59, Ch. 3] or Goldreich [60, Ch. 3].

2.3.3 Pseudo-random functions

The property of "looking" random can be meaningfully defined for functions as well. Indeed, if a pseudo-random string is one that cannot be distinguished by any polynomial-time procedure form a truly random string, then it is natural to define a pseudo-random function as one that cannot be distinguished from a truly-random function. However, while for strings the distinguisher is a probabilistic polynomial-time algorithm that takes a string as input, for functions the distinguisher is a statistical test: a probabilistic polynomial time oracle machine that adaptively queries its oracle in an attempt to figure out if the oracle was drawn from the pseudo-random function ensemble or the uniform ensemble (see Definition 2.1.1). Of course, pseudo-random functions are further required to be efficiently computable.

Definition 2.3.3 (Pseudo-Random Function Ensemble)**.** *We say that* $\mathcal{F} = \{F_k\}_{k\in\mathbb{N}}$ *is a* pseudo-random function ensemble *if it satisfies the following conditions:*

1. Indexing: *Each function in* F_k *has a unique* k*-bit* seed s *associated with it:*

$$F_k = \left\{ f_s \colon \{0,1\}^k \to \{0,1\}^k \ : \ s \in \{0,1\}^k \right\} \ .$$

Thus drawing a function from F_k *is easy: simply toss* k *coins.*

2. Efficiently computable: *There is an efficient algorithm* E *(the "evaluator") such that* $E(s,x) = f_s(x)$ *for all* $x, s \in \{0,1\}^k$*.*

3. Pseudo-randomness: *For all efficient statistical tests* T *and all positive constants* c*, for all sufficiently large* k*,*

$$\left| \Pr\left[T^f(1^k) = 1 \,\middle|\, f \leftarrow F_k \right] - \Pr\left[T^f(1^k) = 1 \,\middle|\, f \leftarrow H_k \right] \right| < \frac{1}{k^c} \ .$$

In other words, pseudo-random function families "look like" truly random functions to any efficient statistical test. (Note that in the experiment the outputs of the oracle are consistent, i.e., they belong to the same function.)

Pseudo-random function families constitute a very powerful tool in cryptographic settings: the functions in such families are easy to select and compute, and yet retain all the desired statistical properties of truly random functions (with respect to polynomial-time algorithms).

Goldreich, Goldwasser, and Micali [67] introduced the notion of a pseudo-random function ensembles, and showed how to construct them using any pseudo-random generator. Naor and Reingold [112] exhibit more practical constructions based on number-theoretic assumptions.

2.3.4 Collision-resistant hashing schemes

The notion of a collection of functions for which it is hard to find collisions was first introduced by Damgård [44]. Formally, such a collection is captured by a pair of machines $(G_{\mathsf{CRH}}, E_{\mathsf{CRH}})$, the *hashing-key generator* and the *evaluator*. On input a security parameter 1^k, $G_{\mathsf{CRH}}(1^k)$ outputs a *hashing key* hk that identifies a function $H_{\mathsf{hk}} \colon \{0,1\}^* \to \{0,1\}^{|\mathsf{hk}|}$ that can be computed efficiently using $E_{\mathsf{CRH}}(\mathsf{hk}, \cdot)$.

Definition 2.3.4 (Collision-Resistant Hashing Scheme)**.** *A pair of polynomial-time machines* $(G_{\mathsf{CRH}}, E_{\mathsf{CRH}})$*, where* G_{CRH} *is probabilistic and* E_{CRH} *is deterministic, is said to be a* collision-resistant hashing scheme *if it satisfies the following two conditions:*

1. Shrinking: *For each* $\mathsf{hk} \in G_{\mathsf{CRH}}(1^k)$*,* $E_{\mathsf{CRH}}(\mathsf{hk}, \cdot)$ *is a function that maps* $\{0,1\}^*$ *to* $\{0,1\}^{|\mathsf{hk}|}$*.*

2. Hard to find collisions: *For every family of polynomial-size collision-finding circuits* $\{C_k\}_{k\in\mathbb{N}}$*, every positive constant* c*, and for all sufficiently large* k*,*

$$\Pr\left[(x \neq y) \wedge \big(E_{\mathsf{CRH}}(\mathsf{hk}, x) = E_{\mathsf{CRH}}(\mathsf{hk}, y)\big) \,\middle|\, \mathsf{hk} \leftarrow G_{\mathsf{CRH}}(1^k) \ ; \ (x,y) \leftarrow C_k(\mathsf{hk}) \right] < \frac{1}{k^c} \ .$$

Collision-resistant hashing schemes are known to be implied by claw-free collections of permutations [44][75], or length-restricted collision-free hashing schemes [43][105]; they easily imply one-way functions, but unfortunately they are not known to follow from the existence of one-way functions, one-way permutations, or even trapdoor permutations — they remain a strong and powerful existential assumption. See Goldreich [61, Sec. 6.2.2.2] for more details.

2.3.5 Signature schemes

Signature schemes [61, Sec. 6.1] capture the notion of a *public-key authentication scheme*. Suppose that some party wants to "sign" a message of his choice, in a way that enables anyone to verify his signature and thus be sure that the message originated from him. Signing the message would be meaningless if anyone could do it, so it is required that no one (other than the party who knows some secret) is able to produce valid-looking signatures (for any message).

More precisely, a signature scheme SIG is a triple of polynomial-time machines $(G_{\mathsf{SIG}}, S_{\mathsf{SIG}}, V_{\mathsf{SIG}})$ with the following syntactic properties:

– The *key generator algorithm* G_{SIG}, on input a security parameter 1^k, outputs a key pair $(\mathsf{vk}, \mathsf{sk})$ where vk is known as the *verification key* and sk as the *signing key*.

– The *signing algorithm* S_{SIG}, on input a signing key sk and a message $m \in \{0,1\}^*$, outputs a *signature* σ of m with respect to the verification key vk corresponding to sk.

– The *signature verification algorithm* V_{SIG}, on input a verification key vk, a message $m \in \{0,1\}^*$ and a signature $\sigma \in \{0,1\}^*$, decides whether σ is valid for m with respect to vk.

The basic requirement is that the verification algorithm recognize as valid all signatures that are legitimately generated by the signing algorithm. Namely, the triple $(G_{\mathsf{SIG}}, S_{\mathsf{SIG}}, V_{\mathsf{SIG}})$ is required to satisfy the following *completeness* property: for any $k \in \mathbb{N}$ and $m \in \{0,1\}^*$,

$$\Pr\left[V_{\mathsf{SIG}}(\mathsf{vk}, m, \sigma) = 1 \;\middle|\; (\mathsf{vk}, \mathsf{sk}) \leftarrow G_{\mathsf{SIG}}(1^k) \;;\; \sigma \leftarrow S_{\mathsf{SIG}}(\mathsf{sk}, m)\right] = 1 \;.$$

The security requirement is that no efficient adversary should be able to generate a valid signature for *any* message, after adaptively querying a signing oracle on messages of his choice (the message output by the adversary should of course be different than every message that was queried to the signing oracle). This security notion is known as *existential unforgeability against chosen-message attack*.

Definition 2.3.5 (Existential Unforgeability against Chosen-Message Attack)**.** *A signature scheme* $(G_{\mathsf{SIG}}, S_{\mathsf{SIG}}, V_{\mathsf{SIG}})$ *is said to be* existentially unforgeable against chosen-message attack *if for every family of polynomial-size circuits* $\{A_k\}_{k\in\mathbb{N}}$*, for every positive constant* c*, and for all sufficiently large* k*,*

$$\Pr\left[\left(V_{\mathsf{SIG}}(\mathsf{vk}, m, \sigma) = 1\right) \;\wedge\; \left(A_k \textit{ did not query } m\right) \;\middle|\; (\mathsf{vk}, \mathsf{sk}) \leftarrow G_{\mathsf{SIG}}(1^k) \;;\; (m, \sigma) \leftarrow A_k^{S_{\mathsf{SIG}}(\mathsf{sk}, \cdot)}(\mathsf{vk})\right] < \frac{1}{k^c} \;.$$

The notion of existential unforgeability against chosen-message attack was introduced by Goldwasser, Micali, and Rivest [75], who first exhibited a signature scheme secure under this definition, using the assumption that claw-free collections of permutations[2] exist. The result was improved by Bellare and Micali [18], who showed how to construct secure signature schemes using any trapdoor permutation. The assumption was further weakened by Naor and Yung [113] to any universal one-way hash function,[3] which they showed how to obtain using any one-to-one one-way function. Finally, Rompel [126] showed how to construct universal one-way hash functions using any one-way function (Katz and Koo [91] provide a full proof of this result, as the original conference paper had some errors).

In this thesis, we use an existentially-unforgeable signature scheme $\mathsf{SIG} = (G_{\mathsf{SIG}}, S_{\mathsf{SIG}}, V_{\mathsf{SIG}})$ that has "concise" signatures, i.e., the length of signatures is polynomial in the security parameter (and, in particular, is independent of the message length). This can be achieved by using a "hash-then-sign" approach [61, Sec. 6.2.2.2], using collision-resistant hashing schemes. This is without loss of generality, because our constructions already assume the existence of collision-resistant hashing schemes (e.g., to obtain universal arguments, see Section 3.3.4 under the paragraph entitled "Universal arguments").

[2] See Goldreich [60, Sec. 2.4.5] for a definition of claw-free collections of permutations; they are known to exist under number-theoretic assumptions such as the difficulty of factoring or the difficulty of taking discrete logarithms.

[3] See Goldreich [61, Sec. 6.4.3] for a definition of universal one-way hash functions.

Chapter 3

Related Work

In this chapter we review prior works that are most related to our results. See the Complexity Zoo [40] for definitions of the complexity classes that we use.

3.1 Proof systems

Central to modern cryptography and complexity theory is the notion of a *proof system*. Traditionally, a proof system is a finite alphabet (e.g., $\{0,1\}$), together with a finite set of *axioms* and *inference rules*. *Statements* are strings of symbols from the alphabet; *true statements* are strings for which there exists a *proof*, i.e., a finite sequence of strings each of which is either an axiom or a string obtained via an inference rule from previous strings.

To put it another way, given some statement x (e.g., "the graph G has a Hamiltonian cycle", "the Boolean formula ϕ is satisfiable", or "the two graphs G and H are not isomorphic"), a prover writes down a proof π in some canonical format; a verifier can examine π and decide whether it is a convincing proof for the fact that x is true.

To demonstrate our notation, let us define, using the above "proof system" setting, the standard complexity class NP:

Definition 3.1.1 (Class NP)**.** *The complexity class* NP *is the set of all languages* L *for which there exists a deterministic polynomial-time verifier* V_L *and an all-powerful prover* P_L*, as well as a positive polynomial* q_L*, that satisfy the following two conditions:*

1. Completeness: *For every* x *in* L*, the prover* P_L *can write a proof* π *of length at most* $q_L(|x|)$ *that the verifier* V_L *will accept as a valid proof for* x*.*
2. Soundness: *For every* x *not in* L*, no (possibly cheating) prover* $\widetilde{P}$ *can write down a proof* $\widetilde{\pi}$ *of length at most* $q_L(|x|)$ *that the verifier* V_L *will accept as a valid proof for* x*.*

Thus, NP captures all statements whose proofs are efficiently verifiable by a *deterministic* procedure, i.e., all statements that have a short and easy to verify proof — in essence, all true statements that are relevant to us in practice. Nonetheless, other interpretations of the notion of "proof system" have been studied, yielding a very rich, and often surprising, set of results. We discuss one interpretation, *probabilistically-checkable proofs*, in Section 3.2, and another interpretation, *interactive proofs*, in Section 3.3.

3.2 Probabilistically-checkable proofs

Suppose that a prover writes down a short (i.e., polynomial-size) proof π for some statement x. Instead of considering a deterministic verifier that must read the whole proof π in order to decide whether he is convinced or not of the veracity of x, let us consider a *probabilistic* verifier with *bounded randomness and "attention"*, i.e., the verifier tosses at most r coins and reads at most q bits of π. We say that a language L has a *probabilistically-checkable proof* if statements in L have proof π that can be checked by the kind of verifiers we just described.

Definition 3.2.1 (Class PCP)**.** *Given functions $r, s\colon \mathbb{N} \to \mathbb{R}$, the complexity class* $\mathsf{PCP}[r, q]$ *consists of those languages L for which there is a probabilistic polynomial-time verifier V_L that satisfies the following conditions:*

1. Verifier restrictions: *On input a string x and with oracle access to another string π, the verifier V_L tosses at most $r(|x|)$ coins and reads at most $q(|x|)$ bits of π.*
2. Completeness: *For every x in L there is a proof π, called a* PCP oracle*, such that $V_L^\pi(x)$ accepts with probability one.*
3. Soundness: *For every x not in L and every proof π, $V_L^\pi(x)$ accepts with probability at most* $1/2$.

Arora and Safra [6] and Arora, Lund, Motwani, Sudan, and Szegedy [7] showed all of NP has a probabilistically checkable proof where the verifier tosses logarithmically many coins and queries a constant number of bits of the PCP oracle; this result is known as the PCP Theorem.

Theorem 3.2.1 (PCP Theorem)**.** $\mathsf{NP} = \mathsf{PCP}[O(\log n), O(1)]$.

There is a long sequence of results leading up to the PCP theorem [118], as well as a large and thriving area of research born out of it, especially in light of the surprising connection between PCPs and hardness of approximation [51]. Even a brief overview of the many results in this area is beyond the scope of this thesis,[1] so we will only mention that: Arora [4] provides an accessible survey to the main results; references and treatment of more recent results can be found in the book of Arora and Barak [5] or, e.g., in the scribed notes of classes taught by Guruswami and O'Donnell [78] and Ben-Sasson [21].

3.2.1 Alternative models of PCP

An important efficiency measure in the area of PCPs is the length of the PCP oracle. For example, can the satisfiability of a size-n formula in k variables be proved by a PCP oracle of size $\mathsf{poly}(k)$, rather than $\mathsf{poly}(n)$? The existence PCP oracles of size polynomial in the number of variables would be very useful in a variety of cryptographic applications, as discussed by Harnik and Naor [82]. Unfortunately, Fortnow and Santhanam [55] show that such PCP oracles do not exist unless $\mathsf{NP} \subset \mathsf{coNP/poly}$.

Interactive PCP. Kalai and Raz [88] consider the alternative setting where a verifier is given access to a PCP oracle *and* an interactive proof (and not just a PCP oracle); the resulting model is called *interactive PCP*. In other words, an interactive PCP is a proof that can be verified by reading only a small number of its bits, with the help of an interactive proof (see Section 3.3). In this new model, they show the following:

Theorem 3.2.2 ([88])**.** *For any constant ε, the satisfiability of a constant-depth formula can be proved by an interactive PCP, where one query is made to a PCP oracle that is polynomial in the number of variables and is followed by an interactive proof that has communication complexity that is polylogarithmic in the formula size, with completeness $1 - \varepsilon$ and soundness $1/2 + \varepsilon$.*

The above theorem finds several cryptographic applications, including succinct zero-knowledge proofs.

Probabilistically-checkable arguments. Kalai and Raz [89] consider another alternative setting, this time by relaxing the soundness property of PCPs to only hold computationally; the resulting model is called *probabilistically-checkable argument* (PCA). Specifically, they consider "one-round arguments" where the verifier sends a message depending on his private coins (but not the statement to be proved) to the prover; at a later time, the prover uses the verifier's message to generate an oracle string, which the verifier will access only at a few bits.

Kalai and Raz [89] give a general reduction from any efficient interactive PCP into a short PCA oracle. Hence, using their result on interactive PCPs (improved by Goldwasser, Kalai, and Rothblum [77]), they show the following:

Theorem 3.2.3 ([89])**.** *For any security parameter t, any formula of size n and depth d in $k \geq \log n$ variables can be proved by a PCA oracle (with an efficient prover) of size* $\mathsf{poly}(k, d, t)$ *by querying a number of queries that is* $\mathsf{poly}(d, t)$*, achieving completeness $1 - 2^{-t}$ and soundness 2^{-t}.*

Hence, at least in the model of PCA, succinct oracle proof strings can be achieved.

[1] The results in this thesis only use the PCP theorem implicitly, through the universal arguments of Barak and Goldreich [15], which form one of the building blocks of one of our constructions.

3.3 Interactive proofs

Goldwasser, Micali, and Rackoff [74][76] first extended the traditional notion of a "written-down" proof to the interactive setting, where the action of proving a theorem is an interactive protocol between an all-powerful prover and a probabilistic polynomial-time verifier.

Definition 3.3.1 (Interactive Proofs and Class IP)**.** *A pair of interactive Turing machines (P, V) is called an* interactive proof system *for a language $L \subset \{0,1\}^*$ if the following conditions hold:*

1. Efficient verification: *The verifier V is a probabilistic polynomial-time machine. (In particular, the number of messages and the total length of all messages exchanged between the prover P and the verifier V are polynomially bounded in the length of the common input.)*

2. Completeness: *For every string x in L, the verifier V always accepts after interacting with the prover P on common input x.*

3. Soundness: *There exists some positive polynomial p such that, for every x not in L and every (possibly cheating) prover strategy $\widetilde{P}$, the verifier V rejects with probability at least $p(|x|)^{-1}$ after interacting with $\widetilde{P}$ on common input x.*

The complexity class IP denotes the set of all languages L that have an interactive proof system. (At times, we consider the subset of IP induced by restricting the number of messages between the prover and the verifier to be bounded by some polynomial $r\colon \mathbb{N} \to \mathbb{N}$ in the length of the common input; this subset is denoted by $\mathsf{IP}[r]$.)

Interactive proof systems thus integrate both *interaction* and *randomness* into the notion of a proof.

Note that *no complexity requirement* is placed on the honest prover P or any cheating prover $\widetilde{P}$.

The power of interactive proofs. When Goldwasser, Micali, and Rackoff [74][76] first introduced interactive proof systems, they observed that NISO, the language of pairs of non-isomorphic graphs, is in IP (recall that NISO is not known, nor believed, to be in NP); indeed, Lund, Fortnow, Karloff, and Noam [103] then gave interactive proof systems for all of coNP. Later, Shamir [129] proved that every language that can be decided in polynomial-time by a Turing machine is also in IP, i.e., that $\mathsf{PSPACE} \subset \mathsf{IP}$ (the other inclusion is trivial, so that $\mathsf{IP} = \mathsf{PSPACE}$ — in particular, this shows that IP is closed under complementation, as PSPACE is); recently, Jain, Ji, Upadhyay, and Watrous [87] showed that $\mathsf{QIP} = \mathsf{IP}$, i.e., that *quantum* interactive proof systems are no more powerful than classical ones. Indeed, the class IP has shown extraordinary expressive power. (Also, Fürer, Goldreich, Mansour, Sipser, and Zachos [56] observe that allowing two-sided error probability does not increase the power of interactive proof systems.)

We note that both interaction and randomness seem essential to the power of interactive proofs: indeed, any language with an interactive proof in which the verifier is deterministic is in NP, and *non*-interactive proofs in which the verifier may toss coins are conjectured to be contained in NP (see the discussion below in the paragraph entitled "Non-interactive proofs") — if one removes *all* interaction then we get BPP. Finally, Goldreich, Mansour, and Sipser [68] showed that the soundness error is essential for the expressive power of interactive proofs: if the verifier never accepts when the common input x is not in L, then L is in NP; that is, interactive proofs with *perfect soundness* exist only for languages in NP.

Public coins. A special case of interactive proof system is when all of the verifier's messages are random strings; we refer to such a protocol as a *pubic-coin* (or *Arthur-Merlin* [11]) interactive proof system. Such protocols were first considered by Babai [10], and were shown by Goldwasser and Sipser [73] to have essentially the same power as the general case (where the verifier may send any kind of message, and, in particular, toss coins and not reveal their outcomes to the prover). An interactive proof system that is not of the public-coin type is of the *private-coin* type.

Arguments (computationally-sound proofs). An important relaxation of the definition of interactive proof systems, first considered by Brassard, Chaum, and Crépeau [26], is to only require that soundness hold against *efficient prover strategies*, i.e., the quantification is only over feasible strategies for $\widetilde{P}$ rather than over all strategies. The resulting condition is called *computational soundness* and is as follows

Computational soundness: There exists some positive polynomial p such that, for every family of polynomial-size circuits $\{\widetilde{P}_k\}_{k\in\mathbb{N}}$ and for all sufficiently large k, for all $x \in \{0,1\}^k \cap L$, the verifier V rejects with probability at least $p(|x|)^{-1}$ after interacting with $\widetilde{P}_{|x|}$ on common input x.

An interactive protocol that is a computationally-sound proof system is called an *interactive argument system*.

Non-interactive proofs. An important special case of an interactive proof system is when the interaction between the prover and the verifier consists of only one message from the prover to the verifier; we call such a proof system a *non-interactive proof system*. *Non-interactive argument systems* are similarly defined for the analogous special case of interactive argument systems. Non-interactive proof systems are also known as *Merlin-Arthur protocols*, and the complexity class of languages solved by such protocols is denoted by MA, which contains NP and BPP and is contained in AM. Impagliazzo and Wigderson [85] proved that, assuming that $\mathsf{DTIME}(2^{O(n)})$ requires exponential-size circuits, $\mathsf{PromiseBPP} = \mathsf{PromiseP}$, implying that $\mathsf{MA} = \mathsf{NP}$. Boppana, Håstad, and Zachos [25] proved that if $\mathsf{coNP} \subset \mathsf{AM}$ then the polynomial hierarchy collapses, implying that if $\mathsf{coNP} \subset \mathsf{MA}$ then the same conclusion holds.

Decreasing the soundness error probability. The soundness error of both interactive proof systems and interactive argument systems can always be reduced to a negligible amount by sequential repetition of the interactive protocol.[2] Parallel repetition always reduces the soundness error of interactive proof systems [59, Appendix C.1], but that is not always the case for interactive argument systems (see Bellare, Impagliazzo, and Naor [19], Haitner [81], and Chung and Liu [38]).

Efficient prover strategies. For a language L in NP, we say that (P, V) is an interactive proof system (resp., interactive argument system) with *efficient prover strategy* if the completeness condition can be satisfied by a probabilistic polynomial-time algorithm that, beyond getting $x \in L$ as input, also gets a witness for x as an auxiliary input. When specifically constructing interactive proof systems for languages in NP, this property follows naturally in most occasions.

Additional properties. Some prover strategies satisfy the additional property that, while they are able to convince the prescribed verifier that some statement is true, they do not "leak" to any verifier any information beyond the fact that the statement is true; this property is called *zero knowledge*, and we discuss it in Section 3.3.1. Some verifier strategies satisfy the additional property that whenever they are convinced by a prover that some statement is true, they also have the confidence that not only there exists a valid proof for the statement, but the particular prover that convinced them *knows* a valid proof; this property is called *proof of knowledge*, and we discuss it in Section 3.3.2.

Efficiency measures. When studying interactive proof systems (and interactive argument systems), parameters of special interest often include:

1. *Round complexity:* The *number of rounds* of an interactive protocol is the total number of messages sent between the two parties during the interaction. Of course, since the verifier is required to run in probabilistic polynomial-time, the round complexity is always polynomial in the common input to the prover and verifier. Nonetheless, one can still ask the question what happens if the number of rounds is required to be very low, e.g., constant — the class of languages having constant-round interactive proofs is denoted AM. In Section 3.3.3, we discuss the known results about this efficiency measure.

2. *Communication complexity:* Interactive proofs may be performed across a channel with costly communication, so it becomes important to find interactive proofs where the number of bits sent between the prover and the verifier is very low. In Section 3.3.4, we discuss the known results about this efficiency measure.

3. *Verifier complexity:* In certain settings, the computational power of a verifier taking part in an interactive proof may be severely bounded, so it becomes important to find interactive proofs where the effort required of a verifier to take part in an interactive proof is minimal. In Section 3.3.5, we discuss the known results about weak verifiers.

[2] Indeed, as Goldwasser and Sipser [73] observed, the class IP does not change if the completeness condition is modified to require that the the acceptance probability of the verifier is bounded below by some function $c\colon \mathbb{N} \to [0,1]$ and the soundness condition is modified to require that the acceptance probability is bounded above by some function $r\colon \mathbb{N} \to [0,1]$, as long as the function $c - s$ is *bounded below by the inverse of some positive polynomial*.

3.3.1 Zero knowledge

Goldwasser, Micali, and Rackoff [74] introduced the notion of *zero knowledge*. Informally, given a proof system (P, V) for a language L, zero-knowledge is a property of the honest prover P that guarantees that *any* feasible (possibly cheating) verifier strategy, by interacting with P, learns nothing about the common input x beyond the fact that $x \in L$. This intuition is captured by a *simulation-based definition* where it is required that the view of a (possibly cheating) verifier can be re-constructed by an efficient algorithm, called the *simulator*, that is given as inputs only the string x, the verifier's strategy, and the verifier's private inputs.[3]

Definition 3.3.2 (Zero Knowledge)**.** *A prover strategy P is* (auxiliary-input) zero knowledge *for a language L if for every probabilistic polynomial-time (possibly cheating) verifier strategy $\widetilde{V}$ and every positive polynomial p there exists a probabilistic polynomial-time simulator algorithm S such that the following two probability ensembles are computationally indistinguishable:*

1. $\{\langle P, \widetilde{V}(z)\rangle(x)\}_{x \in L, z \in \{0,1\}^{\mathsf{poly}(|x|)}}$, *i.e., the output of $\widetilde{V}$, given auxiliary input z, after interacting with P on common input x in L; and*
2. $\{S(x, z)\}_{x \in L, z \in \{0,1\}^{\mathsf{poly}(|x|)}}$, *i.e., the output of the simulator S on input $x \in L$ and the verifier's auxiliary input z.*

We will say that an interactive proof system (resp., interactive argument system) for a language L is auxiliary-input zero knowledge *if the honest prover strategy is auxiliary-input zero knowledge for the language L.*

There is a vast literature on zero-knowledge, whose main results we do not discuss in this thesis as our constructions do not make use of it. An excellent introductory tutorial to the research directions and open questions in zero-knowledge is by Goldreich [63].

3.3.2 Proof of knowledge

The proof-of-knowledge property is a property possessed by some verifiers that, roughly, says that whenever the verifier is convinced then the prover responsible for convincing the verifier "knows" something — that something is usually taken to be a witness for membership of some common input in some NP language.

Goldwasser, Micali, and Rackoff [74] first informally suggested an outline for a definition that could capture the above intuition: a prover "knows" something (e.g., the witness to an NP statement) if there is some polynomial-time Turing machine, called the *knowledge extractor*, with complete control over the prover, that prints that something as a result of interacting with it. In other words, a prover knows something if it can be easily modified so that it outputs the satisfying assignment; and by "easily modified" it is meant any efficient algorithm that uses the prover as an oracle.

Feige, Fiat, and Shamir [50] proposed a formalization of proofs of knowledge by defining *interactive proof systems of knowledge*, for which they proved the following results:

Theorem 3.3.1 ([50, Theorem 1])**.** *If secure public-key encryption schemes exist, every language in* NP *has a zero-knowledge interactive proof system of knowledge.*

Theorem 3.3.2 ([50, Theorem 2])**.** *If secure public-key encryption schemes exist, every language in* $\mathsf{NP} \cap \mathsf{coNP}$ *has an* unrestricted-input *zero-knowledge interactive proof system of knowledge. (Unrestricted input refers to the simulator works on all inputs, and not just inputs in the language.)*

In addition, Feige, Fiat, and Shamir [50] noted that possession of knowledge may be non-trivial even for trivial languages. (For example, deciding the language for the relation "$(x, y) \in \{0,1\}^*$ such that y is the prime factorization of x" is easy, because one only has to check that x is an integer. However, proving knowledge of the prime factorization of x is non-trivial, as finding said factorization is believed hard.) They then show how to use the above results to construct efficient identification schemes.

Tompa and Woll [132], using a definition of interactive proof systems of knowledge similar to that of Feige, Fiat, and Shamir [50], proved that:

[3] The definition that we present here is not the original one of Goldwasser, Micali, and Rackoff [74]; the alternative (and stronger) "auxiliary-input" definition proposed by Goldreich and Oren [66] is required in most practical applications. For example, zero knowledge of the latter kind is sequentially composable, while zero knowledge of the former kind is not.

Theorem 3.3.3 ([132, Theorem 4]). *Random self-reducible languages have zero-knowledge interactive proof system of knowledge.*

Both definitions by Feige et al. [50] and Tompa and Woll [132] follow the intuitive definition of Goldwasser et al. [74].

The definition that we state here (and which is nowadays regarded as the "correct" one) is that of Bellare and Goldreich [17], who first noted that the definitions adopted by Feige et al. [50] and Tompa and Woll [132] are inadequate, both for some of the applications in which they ended up being used (e.g., see Haber and Pinkas [80] and Haber [79]) and also at the intuitive level. For example, Bellare and Goldreich [17] observe that the lack of any requirements of provers that convince the verifier with non-negligible probability is problematic; moreover, at the conceptual level, Feige et al. [50] and Tompa and Woll [132] carelessly combined the (independent) notions of soundness and knowledge, and consider only the case of efficient prover strategies.

Thus, the definition of Bellare and Goldreich [17] refers to *all* provers, independently of their complexity or their probability of convincing the verifier; moreover it makes no distinction between provers that convince the verifier with non-negligible probability or not, but, rather, the knowledge extractor is required to have a running time that is polynomially inversely related to the the probability that the prover convinces the verifier.

Definition 3.3.3 (Proof of Knowledge). *Let $R \subset \{0,1\}^* \times \{0,1\}^*$ be a binary relation and $\kappa\colon \{0,1\}^* \to [0,1]$ a function. We say that (P, V) is an* interactive proof system of knowledge *for the relation R with knowledge error κ if the following two conditions hold:*

1. Completeness: *For every $x \in L(R)$, the verifier V always accepts after interacting with the prover P on common input x.*
2. Knowledge with error κ: *There exists a probabilistic oracle machine E, the* knowledge extractor*, and a positive constant c such that for every (possibly cheating) interactive prover strategy $\widetilde{P}$ and every $x \in L(R)$: if*
$$p(x) \equiv \Pr\left[\left\langle \widetilde{P}, V \right\rangle (x) = 1\right] > \kappa(x) \ ,$$
then, on input x and with oracle access to $\widetilde{P}$, the knowledge extractor E outputs a witness for $x \in L(R)$ within an expected number of steps that is bounded by $|x|^c \cdot (p(x) - \kappa(x))^{-1}$.[4]

Bellare and Goldreich [17] showed how to reduce the knowledge error with sequential repetition (and by parallel repetition in a special case); also, if the knowledge error is sufficiently small, it can be eliminated.

In this thesis, we *depart* from the above definition: we loosely follow Barak and Goldreich [15] in that we consider various forms of proof of knowledge that always imply (computational) soundness. In other words, we will not be concerned with uniformly treating the knowledge of all provers, but will only be concerned with characterizing the knowledge of sufficiently convincing ones. (See Section 4.2.3 and Section 5.2.3 for our two definitions that consider proof of knowledge properties.)

3.3.3 Round efficiency

Adopting a similar notation as was done for IP, we let $\mathsf{AM}[r]$ denote the class of languages with r-round *public-coin* interactive proofs, and let AM denote the class of languages with constant-round public-coin interactive proofs.

Goldwasser and Sipser [73] proved that every interactive proof can always be transformed into a *public-coin* interactive proof at the cost of adding two extra messages:

Theorem 3.3.4 ([73]). *For every positive polynomial r, $\mathsf{IP}[r] \subset \mathsf{AM}[r+2]$.*

In other words, allowing the verifier to toss private coins does not give additional power to interactive proofs. Babai [10] proved a "collapse" theorem for constant-round interactive proofs:

Theorem 3.3.5 ([10]). *For every constant c, $\mathsf{AM}[c] \subset \mathsf{AM}[2]$. (I.e., $\mathsf{AM} = \mathsf{AM}[2]$.)*

The above result was later strengthened by Babai and Moran [11] into a "speed-up" theorem:

[4]Equivalently [17], there exists a probabilistic expected *polynomial-time* oracle machine E such that for every (possibly cheating) interactive prover strategy $\widetilde{P}$ and every $x \in L(R)$ it holds that $\Pr\left[E^P(x) \in R(x)\right] > \Pr\left[\langle P, V\rangle(x) = 1\right] - \kappa(x)$.

Theorem 3.3.6. *For every positive polynomial r, $\mathsf{AM}[2r] \subset \mathsf{AM}[r+1]$.*

Thus, the round complexity of interactive proofs can always be reduced by a constant factor, so that constant-round interactive proofs can always be transformed into two-message interactive proofs.

However, interactive proofs with any polynomial number of rounds are believed to be much more powerful than constant-round interactive proofs. Indeed, all of PSPACE has an interactive proof [129], but constant-round interactive proofs are contained in the second level of the polynomial-time hierarchy. In fact, Boppana, Håstad, and Zachos [25] show that coNP does not have constant-round interactive proofs unless the polynomial-hierarchy collapses:

Theorem 3.3.7 ([25])**.** *If* $\mathsf{coNP} \subset \mathsf{AM}$*, then* $\mathsf{PH} = \Pi_2^{\mathsf{P}}$.

Moreover, languages with constant-round interactive proofs are contained in NP under plausible circuit complexity assumptions. (See Arvind and Köbler [8], Klivans and van Melkebeek [96][97], and Miltersen and Vinodchandran [108][109].)

In summary, constant-round interactive proofs are known (and probably exist only) for NP.

3.3.4 Communication efficiency

The communication complexity of an interactive proof system is a natural efficiency measure. One can therefore ask the question: how low can the communication complexity of proof systems and argument systems be?

Specifically, suppose that a language L is in $\mathsf{NTIME}(t(n))$ for some function $t\colon \mathbb{N} \to \mathbb{N}$. Then, if t is bounded by a polynomial (i.e., $L \in \mathsf{NP}$), there is a natural interactive proof system for L with communication complexity $t(n)$: the prover sends to the verifier the $t(|x|)$-bit witness for the common input $x \in L$. How much better than $t(n)$ can the communication complexity of an interactive proof system or argument system for L be?

Negative results for interactive proofs

Unfortunately, several conditional results indicate that it is unlikely that one can do much better. Goldreich and Håstad [64] proved the following:

Theorem 3.3.8 ([64, Theorem 2])**.** *If a language L has an interactive proof with communication complexity $c(n)$, then L is in* $\mathsf{BPTIME}(2^{O(c(n))} \cdot \mathsf{poly}(n))$.

In particular, if we let $c(n) = \mathsf{polylog}(n)$, then we obtain that $\mathsf{NP} \subset \mathsf{QP}$, where QP is *Quasi-Polynomial Time*; however, QP is widely believed not to contain NP. In fact, Goldreich and Håstad [64] also show that, if we only require a bound on the *prover-to-verifier* communication complexity (thus considering what are called *laconic* provers), a similar result holds for *public-coin* interactive proofs:

Theorem 3.3.9 ([64, Theorem 3])**.** *If a language L has a public-coin interactive proof with prover-to-verifier communication complexity $c(n)$, then L is in* $\mathsf{BPTIME}(2^{O(c(n) \cdot \log c(n))} \cdot \mathsf{poly}(n))$.

Goldreich and Håstad [64] also obtain another result that characterizes languages "just outside of NP", such as Quadratic Non-Residuosity and Graph Non-Isomorphism:

Theorem 3.3.10 ([64, Theorem 4])**.** *If a language L has an interactive proof with prover-to-verifier communication complexity $c(n)$, then L is in* $\mathsf{BPTIME}(2^{O(c(n) \cdot \log c(n))} \cdot \mathsf{poly}(n))^{\mathsf{NP}}$.

In a subsequent paper, Goldreich, Vadhan, and Wigderson [70] provide the first evidence that NP-complete languages cannot have low communication complexity interactive proofs, by showing that if NP has constant-round interactive proofs with logarithmic prover-to-verifier communication complexity, then $\mathsf{coNP} \subset \mathsf{AM}$, which is believed to be unlikely. Recently, Pavan, Selman, Sengupta, and Vinodchandran [121] proved that if coNP has interactive proofs with a polylogarithmic number of rounds, then the exponential-time hierarchy collapses to the second level.

Positive results for interactive arguments

On the other hand, if one relaxes the soundness condition to only hold *computationally*, the situation dramatically changes: unlike for interactive proof systems, interactive argument systems with very low communication complexity for languages in NP have been constructed, and all such constructions are based on the PCP theorem (see Section 3.2).

How does one use the PCP theorem to construct efficient arguments? In a PCP system (with an efficient prover strategy) the PCP prover, on input a witness to the NP statement under consideration, outputs a polynomially long string, called the PCP oracle; the PCP verifier is then given oracle access to the PCP oracle, which he will query at a few places (usually, polylogarithmically many for negligible soundness). Here is a naïve scheme to use the PCP system: the verifier runs the PCP verifier and then sends the queries it produces to the prover; the prover runs the PCP prover on input those queries and replies with the answers. However, note that such a scheme will not work: the soundness of the PCP system is guaranteed only if the PCP oracle is fixed in advance (independent of the coin tosses of the PCP verifier); in the scheme we just described, a malicious prover may easily cheat by choosing the replies depending on the queries he receives. (Also, the PCP oracle is *longer* than the NP witness, so if the prover simply sends to the verifier the whole PCP oracle, then the communication complexity will not be low!)

A construction that works was first invented by Kilian [93] and, since its main idea is essentially preserved in all later works, we will call it a *Kilian-type construction*; it works as follows:[5]

> **Kilian-type construction:** The prover first commits to a PCP oracle by sending a commitment to the verifier. Then, the verifier runs the PCP verifier and sends to the prover the queries for the PCP oracle. The prover sends to the verifier the de-commitments for the requested bits of the PCP oracle. Finally, the verifier checks that the PCP verifier accepts on input these bits.

In order for the communication complexity to be low, the construction cannot use an arbitrary commitment scheme;[6] rather, the commitment scheme must satisfy two properties: it should be *concise*, so that the commitment to the PCP oracle is much smaller than the PCP oracle itself; and it should allow for *local de-commitment*, so that the prover does not have to send the whole PCP oracle as a de-commitment, when the verifier is only interested in the de-commitments of a few of its bits. Assuming that collision-resistant hashing schemes exist, it is easy to construct such a commitment scheme: use the tree hashing technique of Merkle [105]. (See Section 2.3.4.)

Thus, Kilian [93] shows the following:

Theorem 3.3.11 ([93]). *If strong (i.e., sub-exponentially secure) collision-resistant hash functions exist, then* NP *has a public-coin (zero-knowledge) argument system where the verifier tosses polylogarithmically many coins and the communication complexity (in both directions) is polylogarithmic.*

The above result, when combined with the results of Goldreich and Håstad [64] and Goldreich, Vadhan, and Wigderson [70], implies that, under a quite plausible cryptographic assumption, there is a strong separation between the communication-efficiency of interactive arguments for NP and interactive proofs for NP; the separation still holds even if the requirement of public coin and verifier-to-prover communication are dropped, and only the prover-to-verifier communication is counted.

Computationally-sound proofs. Micali [106][107] generalizes the ideas of Kilian [93] and defines the notion of a *computationally-sound proof* (CS proof). Loosely speaking, a CS proof attempts to capture the notion of a certificate for a computation's correctness, as well as the efficiency requirement that convincing someone else that a computation is correct should be much easier than convincing yourself in the first place (as evidenced by the results of Kilian [93]).

In order to be able to "talk" about all computations, he defines a language that can handle all computations in a uniform manner.

Definition 3.3.4 (CS Language). *The* CS language*, denoted* $\mathcal{L}_{\mathsf{CS}}$*, is the set of all quadruples* $q = (M, x, y, t)$ *such that* M *is (the description of) a Turing machine,* x *and* y *are binary strings, and* t *is a binary integer such that* $|x|, |y| \leq t$*,* $M(x) = y$*, and* $\mathsf{time}_M(x) = t$*.*

Thus, a CS proof is an interactive proof system for the CS language.

[5] We disregard here the zero-knowledge aspect, which was one of the goals of Kilian [93].

[6] Note that we are using the terminology "commitment" somewhat loosely, because we do not require that the commitment be hiding (but, of course, it should be computationally binding).

Definition 3.3.5 (Interactive CS Proof)**.** *Let $\mathcal{L}_{\mathsf{CS}}$ be the CS language. A pair of machines $(P_{\mathsf{CS}}, V_{\mathsf{CS}})$ is said to be an* interactive CS proof system *if the following conditions hold:*

1. Efficient verification: *There exists a polynomial p such that for any $k \in \mathbb{N}$ and $q = (M, x, y, t)$, the total time spent by the (probabilistic) verifier strategy V_{CS}, on common input $(1^k, q)$, is at most $p(k+|q|) = p(k+|M|+|x|+|y|+\log t)$. In particular, all messages exchanged during the interaction have length that is at most $p(k+|q|)$.*

2. Completeness via a relatively-efficient prover: *For every $k \in \mathbb{N}$ and $q = (M, x, y, t) \in \mathcal{L}_{\mathsf{CS}}$,*

$$\Pr\left[\langle P_{\mathsf{CS}}, V_{\mathsf{CS}} \rangle \left(1^k, q\right) = 1 \right] = 1 \ .$$

   Furthermore, there exists a positive constant c such that for every $q = (M, x, y, t) \in \mathcal{L}_{\mathsf{CS}}$ the total time spent by P, on common input $(1^k, q)$, is at most $(|q|kt)^c$.

3. Computational soundness: *There exist positive constants b, c, d such that for every $\widetilde{q} \notin \mathcal{L}_{\mathsf{CS}}$, every k with $2^k > |q|^b$, and every prover circuit $\widetilde{P}$ of size at most 2^{ck},*

$$\Pr\left[\left\langle \widetilde{P}, V_{\mathsf{CS}} \right\rangle \left(1^k, q\right) = 1 \right] < \frac{1}{2^{dk}} \ .$$

Using a Kilian-type construction, Micali [106][107] showed the following:

Theorem 3.3.12 ([106][107])**.** *If strong (i.e., sub-exponentially secure) collision-resistant hashing schemes exist, then there exist (four-message, public coin) interactive CS proof systems. Moreover, under a plausible number-theoretic assumption (the Φ-Hiding Assumption introduced by Cachin, Micali, and Stadler [28]), there exist two-message interactive CS proof systems.*

The above theorem, when combined with the Fiat-Shamir heuristic [52], one obtains the following:

Theorem 3.3.13. *In the random-oracle model, there exist* non-interactive *CS proof systems.*

It is an open problem how to construct non-interactive CS proof systems in the plain model, or even in the common random string model.

Universal arguments. Barak and Goldreich [15] relax the computational soundness condition of the (interactive) CS proofs of Micali [106][107], and show how to construct *universal arguments* under the assumption that *standard* collision-resistant hashing schemes exist (improving over the results of Kilian [93] and Micali [106][107], where *strong* collision-resistant hashing schemes were needed instead).

Specifically, Barak and Goldreich [15] consider a language analogous to the CS language $\mathcal{L}_{\mathsf{CS}}$, defined as follows:

Definition 3.3.6 (Universal Set)**.** *The* universal set*, denoted $S_{\mathcal{U}}$, is the set of all triples $y = (M, x, t)$ such that M is (the description of) a non-deterministic Turing machine, x is a binary string, and t is a binary integer such that M accepts x within t steps. We denote by $R_{\mathcal{U}}$ the witness relation of the universal set $S_{\mathcal{U}}$, and by $R_{\mathcal{U}}(y)$ the set of valid witnesses for a given triple y.*

The name "universal" comes from the fact that every language L in NP is linear-time reducible to $S_{\mathcal{U}}$ by mapping every instance x to the triple $(M_L, x, 2^{|x|})$, where M_L a non-deterministic Turing machine that decides L, so $S_{\mathcal{U}}$ can uniformly handle all NP statements. Moreover, $S_{\mathcal{U}}$ is in NE; also, any language in NE is linear-time reducible to $S_{\mathcal{U}}$ by mapping an instance $x \in L$, with L in $\mathsf{NTIME}(2^{cn})$, to the instance $(M_L, x, 1, 2^{c|x|})$; thus, so that $S_{\mathcal{U}}$ is NE-complete. In fact, every language in NEXP is polynomial-time reducible to $S_{\mathcal{U}}$.

A *universal argument* is then an efficient interactive argument system (with a weak proof of knowledge) for the universal set.

Definition 3.3.7 (Universal Arguments)**.** *A* universal argument system *is a pair of machines $(P_{\mathsf{UA}}, V_{\mathsf{UA}})$ that satisfies the following conditions:*

1. Efficient verification: *There exists a polynomial p such that for any $y = (M, x, t)$, the total time spent by the (probabilistic) verifier strategy V_{UA}, on common input y, is at most $p(|y|) = p(|M|+|x|+\log t)$. In particular, all messages exchanged during the interaction have length that is at most $p(|y|)$.*

2. Completeness via a relatively-efficient prover: *For every* $((M,x,t),w) \in R_{\mathcal{U}}$,

$$\Pr\Big[\langle P_{\mathsf{UA}}(w), V_{\mathsf{UA}}\rangle(M,x,t)=1\Big]=1 \ .$$

Furthermore, there exists a polynomial p *such that for every* $((M,x,t),w) \in R_{\mathcal{U}}$ *the total time spent by* $P_{\mathsf{UA}}(w)$*, on common input* (M,x,t)*, is at most*

$$p\Big(|M|+|x|+\mathsf{time}_M(x,w)\Big) \le p\big(|M|+|x|+t\big) \ .$$

3. Computational soundness: *For every family of polynomial-size prover circuits* $\{\widetilde{P}_k\}_{k\in\mathbb{N}}$ *and every positive constant* c*, for all sufficiently large* $k \in \mathbb{N}$*, for every* $(M,x,t) \in \{0,1\}^k - S_{\mathcal{U}}$,

$$\Pr\Big[\Big\langle \widetilde{P}_k, V_{\mathsf{UA}}\Big\rangle(M,x,t)=1\Big] < \frac{1}{k^c} \ .$$

4. Weak proof of knowledge:[7] *For every positive polynomial* p*, there exists a positive polynomial* p' *and a probabilistic polynomial-time knowledge extractor* E_{UA} *such that the following holds: for every family of polynomial-size prover circuits* $\{\widetilde{P}_k\}_{k\in\mathbb{N}}$*, for all sufficiently large* $k\in\mathbb{N}$*, for every instance* $y=(M,x,t)\in\{0,1\}^k$*, if* $\widetilde{P}_k$ *convinces* V_{UA} *with non-negligible probability,*

$$\Pr\Big[\Big\langle \widetilde{P}_k, V_{\mathsf{UA}}\Big\rangle(y)=1\Big] > \frac{1}{p(k)}$$

(where probability is taken over the internal randomness of V_{UA}*), then* E_{UA}*, with oracle access to* $\widetilde{P}_k$ *and on input* y*, is an implicit representation of a valid witness for* y *with non-negligible probability,*

$$\Pr\Big[\exists\, w=w_1\cdots w_t \in R_{\mathcal{U}}(y) \,:\, \forall\, i\in[t]\,,\ E_{\mathsf{UA}}^{\widetilde{P}_k}(y,i)=w_i\Big] > \frac{1}{p'(k)}$$

(where the probability is taken over the internal randomness of E_{UA}*).*

Note that, in the definition of the weak proof-of-knowledge property, the knowledge extractor E_{UA} is required to run in probabilistic polynomial-time, while, on the other hand, the size of the witness for a particular instance $y=(M,x,t)$ may be exponential; therefore, we can only require that the knowledge extractor is an "implicit representation" of a valid witness. Moreover, both E_{UA} and p' may depend on p, so that the proof of knowledge is "weak" in the sense that it does not imply the standard (or, "strong") proof of knowledge, as defined in Section 3.3.2.

Barak and Goldreich [15] use a Kilian-type construction to prove the following:

Theorem 3.3.14 ([15])**.** *If (standard) collision-resistant hashing schemes exist, then there exist (four-message, public coin) universal arguments.*

Incrementally-verifiable computation. Valiant [134] introduces the notion of *incrementally-verifiable computation* (IVC), in the common random string model. Roughly, IVC is a compiler C_{IVC}, together with a verifier V_{IVC}, that works as follows. Given a security parameter $k\in\mathbb{N}$, for any (possibly superpolynomial-time) Turing machine M with $|M|\le k$, $C(1^k,M)$ outputs another Turing machine M' that carries out the same computation as M does, and, moreover, it is *incrementally verifiable*, i.e., every $\mathsf{poly}(k)$ steps outputs a proof string attesting to the correctness of its computation so far — the verifier V_{IVC} can verify these proof strings.

In order to argue the existence of IVC in the common random string model, Valiant [134] first considers the random oracle model. In the random oracle model, he exhibits an *online* knowledge extractor for non-interactive CS proofs; then, he uses the resulting proof of knowledge property of non-interactive CS proofs to show that non-interactive CS proofs can be "composed" (roughly, a non-interactive CS proof can be used as a witness to an NP statement that will be itself made into a CS proof); such composition easily yields a construction for IVC in the random oracle model.

Unfortunately, the construction for IVC in the random oracle model does not quite carry through, because the PCP theorem does not relativize [54], not even with respect to a random oracle [33]. Therefore, one

[7] The weak proof-of-knowledge property implies computational soundness.

cannot prove statements that are decided by a procedure that accesses an oracle; in particular, including a non-interactive CS proof as part of a witness to an NP statement (whose truth depends on the verification of the CS proof) and then make a CS proof out of that is an ill-defined operation.

Thus, Valiant [134] is forced to *assume* the existence of IVC in the common random string model, and argue that the flawed construction of the random oracle model can be taken as evidence for its existence in the common random string model. Nonetheless, the plausibility of such evidence is limited by the fact that the construction of the knowledge extractor for non-interactive CS proofs seems to depend quite essentially on the random oracle, and it is unclear that a knowledge extractor would exist in the common random string model as well.

PCP theorem and efficient arguments. Thus, the PCP theorem has allowed the construction of the asymptotically most efficient way of proving NP statements. The resulting efficient argument systems have immediate applications to delegation of computation, but have also found unexpected applications, as found by Canetti, Goldreich, and Halevi [30][31] and Barak [13].

On the necessity of PCPs in efficient arguments

Observing that all known constructions of efficient arguments make use of a *polynomial-size* PCP, Ishai, Kushilevitz, and Ostrovsky [86] raised the question of whether the use of such PCPs is inherent to efficient arguments. Constructions of polynomial-size PCPs are often complex, and it seems unlikely that they could be used in practice.

Ishai, Kushilevitz, and Ostrovsky [86] show that, if one is interested only in the efficiency of the prover-to-verifier communication, then, assuming the stronger assumption of the existence of homomorphic encryption, the full machinery of polynomial-size PCPs is not needed. Using the idea of commitment with linear de-commitment, they show:

Theorem 3.3.15 ([86]). *There is a compiler that, on input a linear PCP and an additively homomorphic encryption scheme, constructs an argument system where the prover-to-verifier communication consists of only a constant-number of encrypted field elements.*

For example, the *exponential-size* linear PCP based on Hadamard codes of Arora, Lund, Motwani, Sudan, and Szegedy [7] and a homomorphic encryption such as the one of Goldwasser and Micali [72] could be plugged into the compiler. We stress that the verifier-to-prover communication complexity is polynomial, and that additively homomorphic encryption is a stronger assumption than collision-resistant hashing schemes. Moreover, the protocol is not public-coin (as the interaction includes a commit phase and a de-commit phase).

Nonetheless, Rothblum and Vadhan [127] provide evidence that suggests that PCPs *are* inherent to efficient arguments. Roughly, they show that any argument system whose soundness follows from a black-box reduction to the security of a cryptographic primitive yields a PCP with parameters that are related to the efficiency of the argument system. In particular, they show that argument systems base on collision-resistant hashing schemes, the RSA assumption, and homomorphic encryption (all of which have been used to construct efficient argument systems) yield related PCP systems.

3.3.5 Verifier efficiency

While the verifier in an interactive proof is always efficient (i.e., must run in probabilistic polynomial time), there remains the question of studying the power of certain classes of verifiers with severe complexity restrictions.

Goldwasser, Kalai, and Rothblum [77] study public-coin interactive proofs with space-bounded verifiers. They prove the existence of public-coin interactive proofs for the setting in which the verifier is very weak, i.e., the verifier is allowed only a logarithmic amount of space. Their main result is the following.

Theorem 3.3.16 ([77]). *Let s and d be functions from positive integers to reals. If a language L is decided by a $O(\log s)$-space uniform circuit of size s and depth d, then it has a* public-coin interactive proof *with completeness equal to* 1 *and soundness equal to* $1/2$*, with the following additional properties:*

- *the prover runs in time* $\mathsf{poly}(s)$*;*
- *the verifier runs in time* $(n+d)\cdot\mathsf{polylog}(s)$ *and space* $O(\log s)$*; and*
- *the communication complexity is* $d\cdot\mathsf{polylog}(s)$.

Note that the result puts no requirements on the complexity of cheating provers, as opposed to computationally sounds proofs (and related work discussed in Section 3.3.4) where soundness holds only computationally. On the other hand, the result is particularly interesting only for shallow circuits (i.e., when d is much smaller than s, for example $d = O(\log s)$) because the verifier complexity is much smaller than the size of the circuit; computationally sound proofs instead yield very efficient verifiers for all of NP (and, in fact, for all languages in the CS language $\mathcal{L}_{\mathsf{CS}}$).

3.4 Secure multi-party computation

Secure multi-party computation [69][20][34] considers the problem of *secure function evaluation*: given a function f with n inputs, how can n parties in the real world realize the ideal setting in which they each send to a trusted party their respective inputs and the trusted party returns the correct evaluation of f on these inputs?

Realizing this ideal setting is usually stated as ensuring a property of correctness (each party learns the correct evaluation of f) and one of secrecy (each party learns nothing about other parties' inputs, other than what can be learned from f's output), and these properties should be achieved even under some kind of adversarial behavior (e.g., there is an honest majority).

A survey of the results in this area is beyond the scope of this thesis; an overview of the main theoretical results is offered by Goldwasser [71]; for more details, see the recent set of notes from Cramer, Damgård, and Nielsen [42].

Proof-carrying data vs. secure multi-party computation. Recall that proof-carrying data considers the problem of *secure distributed computations*: given a compliance predicate $\mathbf{C}$, how can parties in the real world compute and exchange messages in a way that realizes the ideal setting in which the only allowed messages are ones that are consistent with $\mathbf{C}$-compliant computations?

Unlike secure multi-party computation, proof-carrying data is only concerned with correctness, i.e., messages sent between parties are forced to be consistent with $\mathbf{C}$-compliant computations; indeed, secrecy as understood in secure multi-party computation does not hold — later parties are expected to choose their inputs based on messages received from previous parties.[8] Our setting is not *one* function evaluation, but ensuring a single invariant (i.e., $\mathbf{C}$-compliance) through *many* interactions and computations between parties.

With regard to techniques, our approach follows that of [69] in that parties prove to each other, by cryptographic means, that they have been behaving correctly.

Finally, we remark that secure multi-party computation protocols are unscalable in the sense of not preserving the communication graph of the original computation: even the simple "F and G" example of Section 1.3 would require *everyone on the Internet* to talk to each other. By contrast, in our approach, parties perform only local computation to produce proof strings "on the fly", and attach them to outgoing data packets.

[8] We note that zero-knowledge PCD systems are naturally defined (and necessary for some of our suggested applications). The secrecy guarantees that they would provide is to ensure that the proof string attached to the output of a party leaks no information about the local inputs of that party. We stress that this notion of secrecy is different than the notion of secrecy considered in secure multi-party computation. We do not see fundamental barriers to the existence of zero-knowledge PCD systems, and their efficient construction is a subject of present investigation.

Chapter 4

An Argument System for Hearsay

We introduce a new argument system for NP, which can prove statements based on "hearsay evidence", i.e., statements expressed by a decision procedure that itself relies on proofs generated by earlier, recursive invocations of the proof system (as in the "F and G" example of Section 1.3).

At a high level, our goal is a proof system with the following features:

- *Non-interactive*, so that (i) its proof strings can be forwarded and included as part of the "hearsay evidence" for subsequent proofs, and so that (ii) its proof strings can be used to augment unidirectional communication in proof-carrying data.
- *Efficient*, so that proof strings (and their verification) are much shorter than the time to decide statements they attest to.
- *Aggregatable*, which means that it can generate an argument for a statement decided by a procedure that verifies "hearsay evidence" that is the aggregation of at most polynomially many arguments.

We call an argument system that satisfies the above set of properties a *hearsay-argument system*. In our construction the prover is assisted by an oracle, so we define and obtain an *assisted-prover hearsay-argument system* (APHA system).

The following sections are organized as follows:

Section 4.1 Discussion of a fundamental difficulty in achieving the above properties, and of how we resolve it by introducing an assisted prover.

Section 4.2 Definition of APHA systems and discussion of their properties.

Section 4.3 Generic construction of APHA systems, based on collision-resistant hashing schemes.

Section 4.4 Proof sketch of the construction's correctness (for full proof see Appendix A).

Section 4.5 Discussion of the realizability of an assisted prover.

4.1 Difficulties and our solution

In constructing an argument system that satisfies the properties discussed above, two opposing requirements arise:

- **We must not use oracles.** While we know how to construct efficient argument systems using different approaches (using a short PCP and a Merkle tree [93][107][15], or using a long PCP and homomorphic encryption [86]), all known efficient argument system constructions are based on the PCP theorem, and there is some evidence that this is inherent [127]. Since the PCP theorem does not relativize [54] (not even with respect to a random oracle [33]), these systems cannot prove statements that are decided by a procedure that accesses an oracle. Thus, to allow recursive aggregation of proofs, it seems the system cannot rely on oracles.

– **We must use oracles.** Efficient non-interactive argument systems for NP are only known to exist in the random oracle model, where the verifier needs access to the random oracle. Moreover and more fundamentally, in order to prove statements involving "hearsay evidence", we need a proof-of-knowledge property — as discussed in Section 1.3, mere soundness does not suffice. To support repeated aggregation of such proofs, the proof-of-knowledge must be of a very strong form: a very efficient online [120][53] knowledge extractor with a tight success probability. The only known approach to such knowledge extraction is to force the prover to expose the witness in queries to an oracle.

Prior attempts to aggregate proofs. The tension between the above two requirements arises already in Valiant's work [134] (see preliminary discussion on Valiant's incrementally-verifiable computation in Section 1.3). On one hand, he uses CS proofs as non-interactive arguments. Hence, his construction is ill-defined: it requires generating (PCP-based) CS proofs for statements decided by a procedure that uses oracle access, which, as discussed above, is impossible in general. Therefore, one can at best conjecture (as done in [134]) that the construction, once the random oracle has been instantiated by an appropriate function ensemble, is secure.

Moreover, in order to prove the existence of an efficient knowledge extractor with a tight success probability, Valiant exhibits a procedure that examines a prover's calls to the random oracle. However, once the random oracle has been instantiated by an efficient hash function, the procedure fails since there are no oracle calls to examine.

This difficulty seems inherent: Valiant's construction uses an online knowledge extractor that observes an execution of a prover only through its inputs, outputs, and oracle calls (of which there are none after instantiation), and the online knowledge extractor must be able to extract a witness of size $3n$ given a proof string of size only n. The existence of such a procedure would imply that for any NP language, the witnesses can be compressed by a factor of 3, which seems unlikely.

Lastly, note that the proof-of-knowledge property we require is even stronger than [134] aimed for, in terms of the knowledge extractor's tightness. This is because incrementally verifiable computation allows proofs to be aggregated in a logarithmically-deep tree, so a multiplicative blowup can be tolerated at every extraction step. Conversely, PCD systems must handle polynomially-long chains of proofs, and can thus tolerate an *additive* blowup per extraction step; hence the knowledge extractor can do little more than merely run the prover.

Our solution. We manage to simultaneously satisfy the above requirements, by requiring the prover to access an oracle but not requiring the verifier to do so. A high-level description of our construction follows.

We start with the interactive protocol for public-coin, constant-round universal arguments. By granting the prover access to a signed-input-and-randomness oracle (informally defined in Section 1.5 and to be formally defined in Section 4.3), we turn this into a non-interactive protocol: the prover obtains the public-coin challenges from the oracle instead of the verifier (in a way that also enforces the proper temporal dependence).

The oracle signs its answers using a public-key signature scheme, so that the oracle's random answers are verifiable without access to the oracle. This asymmetry breaks the tension of the two requirements above, i.e., it breaks the "PCP vs. oracles" tension.

Additionally, we require the prover to obtain a signature for the witness that he uses to generate an argument, thus forcing the prover to query the oracle with the witness. This yields a very strong form of proof-of-knowledge property.

We exploit two (related) properties of the oracle: explicitness and temporal dependence. Seeing the oracle's signature on (x, r) implies that r was drawn at random *after* x was *explicitly* written down. In the construction, x will be (for example) a purported prover message in an interactive argument, and r will be the verifier's (public-coin) response. Such forcing of temporal ordering is reminiscent of the Fiat-Shamir heuristic [52]. Extraction of witnesses from oracle queries was used by Pass [120], Fischlin [53] and Valiant [134]. Our approach of using signatures to force oracle queries is similar in spirit to that of Chandran et al. [32].

The introduction of an oracle accessible by the prover is, of course, an extra requirement of our model. Yet given the discussion above, it seems inevitable. In Section 4.5, we argue that the specific oracle that we choose, a signed-input-and-randomness oracle, is reasonable in practice.

4.2 Definition of APHA systems

We proceed to define assisted-prover hearsay-argument (APHA) systems, starting with their structure and an informal description of their properties, and then following with a formal definition. Throughout, we will uniformly handle all NP instances by considering theorems about membership into the universal set $S_{\mathcal{U}}$, introduced in Section 3.3.4 under the paragraph entitled "Universal arguments".

4.2.1 Structure of APHA systems

An APHA system is a triple of machines $(G_{\mathsf{APHA}}, P_{\mathsf{APHA}}, V_{\mathsf{APHA}})$ that works as follows:

- The *oracle generator* G_{APHA}: for a security parameter $k \in \mathbb{N}$, $G_{\mathsf{APHA}}(1^k)$ outputs the description of a probabilistic[1] stateless oracle O to assist the prover, together with O's verification key vk;
- The *prover* P_{APHA}: for a verification key vk, an instance $y = (M, x, t)$, and a string w such that (y, w) is in the witness relation $R_{\mathcal{U}}$ of the universal set $S_{\mathcal{U}}$ (i.e., the machine M, on input x and w, accepts within t steps), $P^{O}_{\mathsf{APHA}}(\mathsf{vk}, y, w)$ outputs a proof string π for the claim that $y \in S_{\mathcal{U}}$; and
- The *verifier* V_{APHA}: for a verification key vk, an instance y, and a proof string π, $V_{\mathsf{APHA}}(\mathsf{vk}, y, \pi)$ accepts if π convinces him that $y \in S_{\mathcal{U}}$.

4.2.2 Properties of APHA systems (intuitive)

The triple $(G_{\mathsf{APHA}}, P_{\mathsf{APHA}}, V_{\mathsf{APHA}})$ must satisfy three properties — the first two are essentially the verifying and proving complexity requirements of computationally-sound proofs and universal arguments, and the third one is a form of proof-of-knowledge property (that is strictly stronger than the regular one [60, Sec. 4.7]).

First, proof strings generated by the prover should be *efficiently verifiable* by the verifier: V_{APHA} halts in time that is polynomial in the security parameter k and the length of the instance y; in particular, the length of a proof string π is also so bounded.

Second, the prover should be able to *prove true theorems using a reasonable amount of resources*: whenever it is indeed the case that $(y, w) \in R_{\mathcal{U}}$, $P^{O}_{\mathsf{APHA}}(\mathsf{vk}, y, w)$ always convinces V_{APHA}; moreover, P_{APHA} halts in time that is polynomial in the security parameter k, the size of the description of M, the length of x, and $\mathsf{time}_M(x, w)$. (Note that $\mathsf{time}_M(x, w)$ is the *actual* time it takes for M to halt on input x and w, and not the upper bound t.)

Third, there exists a fixed *list extractor* circuit LE of size $\mathsf{poly}(k)$ such that, for any (possibly cheating) prover circuit $\widetilde{P}$ of size $\mathsf{poly}(k)$ that outputs an instance y and proof π that convince V_{APHA}, LE produces a valid witness for y in the following sense. By examining only the oracle query-answer transcript $[\![\widetilde{P}(\mathsf{vk}), O]\!]$ of $\widetilde{P}$, LE produces a list of triples $\{(y_i, \pi_i, w_i)\}_i$ with the property that there exists some triple (y_j, π_j, w_j) for which $y_j = y$, $\pi_j = \pi$, and for every such triple w_j is a valid witness for y. This implication holds with all but negligible probability (over the output of G_{APHA}). Note that LE is not explicitly told which y or π to look for.

4.2.3 Properties of APHA systems (formal)

Capturing the above definition more formally, we define APHA systems as follows:

Definition 4.2.1 (APHA System)**.** *An* assisted-prover hearsay-argument system *with security parameter k is a triple of polynomial-time machines $(G_{\mathsf{APHA}}, P_{\mathsf{APHA}}, V_{\mathsf{APHA}})$, where G_{APHA} is a probabilistic, P_{APHA} is deterministic with oracle access, and V_{APHA} is a deterministic, that satisfies the following conditions:*

- Efficient verification: *There exists a polynomial p such that for any $k \in \mathbb{N}$, $(O, \mathsf{vk}) \in G_{\mathsf{APHA}}(1^k)$, instance $y = (M, x, t)$, and proof string π,*

$$\mathsf{time}_{V_{\mathsf{APHA}}}(\mathsf{vk}, y, \pi) \leq p(k + |y|) \ .$$

In particular, $|\pi| \leq p(k + |y|)$, i.e., the proof string length is $\mathsf{poly}(k + |\langle M \rangle| + |x|) + \mathsf{polylog}(t)$.

[1] While our constructions are given for a probabilistic oracle, in Section 4.5 we discuss how to "derandomize" the oracle and make it deterministic.

– Completeness via a relatively-efficient prover: *For every $k \in \mathbb{N}$ and $(y, w) \in R_{\mathcal{U}}$,*

$$\Pr\left[V_{\mathsf{APHA}}(\mathsf{vk}, y, \pi) = 1 \;\middle|\; (O, \mathsf{vk}) \leftarrow G_{\mathsf{APHA}}(1^k) \;;\; \pi \leftarrow P^O_{\mathsf{APHA}}(\mathsf{vk}, y, w)\right] = 1$$

(where the probability is taken over the internal randomness of G_{APHA} and O). Furthermore, there exists a polynomial p such that for every $k \in \mathbb{N}$, $(O, \mathsf{vk}) \in G_{\mathsf{APHA}}(1^k)$, and $((M, x, t), w) \in R_{\mathcal{U}}$,

$$\mathsf{time}_{P^O_{\mathsf{APHA}}}(\mathsf{vk}, (M, x, t), w) \leq p\big(k + |\langle M \rangle| + |x| + \mathsf{time}_M(x, w)\big) \;.$$

Note that $\mathsf{time}_M(x, w) \leq t$.

– List extraction: *There exists a* list extractor *circuit* LE *such that for every (possibly cheating) prover circuit $\widetilde{P}$ of size $\mathsf{poly}(k)$, for all sufficiently large k, if $\widetilde{P}$ convinces V_{APHA} then LE extracts a list containing a witness:*

$$\begin{aligned}\Pr\Big[V_{\mathsf{APHA}}(\mathsf{vk}, y, \pi) = 1 \;\longrightarrow\; &\Big(\big(\exists\, (y_i, \pi_i, w_i) \in \mathsf{extlist} \;\; s.t. \;\; y_i = y, \pi_i = \pi\big) \;\; and \\ &\big(\forall\, (y_i, \pi_i, w_i) \in \mathsf{extlist} \;\; s.t. \;\; y_i = y, \pi_i = \pi : (y_i, w_i) \in R_{\mathcal{U}}\big)\Big) \;\Big|\; \\ (O, \mathsf{vk}) \leftarrow G_{\mathsf{APHA}}(1^k) \;;\; (y, \pi) \leftarrow \widetilde{P}^O(\mathsf{vk}) \;;\; &\mathsf{extlist} \leftarrow \mathsf{LE}\big([\![\widetilde{P}(\mathsf{vk}), O]\!]\big)\Big] > 1 - \mu(k)\end{aligned}$$

(where the probability is taken over the internal randomness of G_{APHA} and O), for some negligible function μ. Furthermore, $|\mathsf{LE}|$ is $\mathsf{poly}(k)$.

Proof of knowledge. Note that the list-extraction property implies a proof-of-knowledge property, in which a knowledge extractor directly outputs a witness corresponding to an instance-proof pair that convinces the verifier. More precisely, the APHA proof-of-knowledge property is given by the following definition:

Definition 4.2.2 (APHA Proof-of-Knowledge Property)**.** *Let $k \in \mathbb{N}$. For every (possibly cheating) prover circuit $\widetilde{P}$ of size $\mathsf{poly}(k)$, there exists a* knowledge extractor *circuit E_{APHA} of size $\mathsf{poly}(k)$ such that, for every polynomial p, and for sufficiently large k: if $\widetilde{P}$ convinces V_{APHA} with non-negligible probability,*

$$\Pr\left[V_{\mathsf{APHA}}(\mathsf{vk}, y, \pi) = 1 \;\middle|\; (O, \mathsf{vk}) \leftarrow G_{\mathsf{APHA}}(1^k) \;;\; (y, \pi) \leftarrow \widetilde{P}^O(\mathsf{vk})\right] > \frac{1}{p(k)}$$

(where the probability is taken over the internal randomness of G_{APHA} and O), then E_{APHA} extracts a valid witness from $\widetilde{P}$ with almost the same probability,

$$\Pr\left[(y, w) \in R_{\mathcal{U}} \;\middle|\; (O, \mathsf{vk},) \leftarrow G_{\mathsf{APHA}}(1^k) \;;\; (y, \pi) \leftarrow \widetilde{P}^O(\mathsf{vk}) \;;\; (y, w) \leftarrow E^O_{\mathsf{APHA}}\big(\widetilde{P}, \mathsf{vk}\big)\right] > \frac{1}{p(k)} - \mu(k)$$

(where the probability is taken over the internal randomness of G_{APHA} and O), for some negligible function μ.

Indeed, it is easy to use the list extractor circuit to define the circuit of the knowledge extractor: the knowledge extractor need only run the list extractor and locate the relevant triple in the list. Specifically, the knowledge extractor E_{APHA}, with oracle access to O and on input $(\widetilde{P}, \mathsf{vk})$, does the following:

1. Run $\widetilde{P}^O(\mathsf{vk})$ to obtain its output (y, π) and its oracle query-answer transcript $[\![\widetilde{P}(\mathsf{vk}), O]\!]$.
2. Run $\mathsf{LE}\big([\![\widetilde{P}(\mathsf{vk}), O]\!]\big)$ to obtain its output $\mathsf{extlist} = \{(y_i, \pi_i, w_i)\}_i$.
3. Search $\mathsf{extlist}$ for a pair (y_i, π_i, w_i) such that $y_i = y$ $\pi_i = \pi$, and, if such a pair is found, output w_i.
4. If Step 3 is unsuccessful, output $\perp$.

Correctness of E_{APHA} easily follows from the list-extraction property.

Adaptive soundness. As always, proof-of-knowledge implies computational soundness: if a (possibly cheating) efficient prover convinces the verifier with probability better than $1/p(k)$, then a witness can be *extracted* with nonzero probability and thus *exists*. Moreover, APHA systems are *adaptively* sound, i.e., soundness holds even when the prover chooses the instance for which he wishes to produce a proof string. In particular, the instance may depend on the oracle O and the verification key vk.

4.3 Construction of an APHA system

We now construct an APHA system. In the assisted-prover model, every party has black-box access to a certain functionality and, in our case, the black-box functionality is defined as follows:[2]

Definition 4.3.1 (Signed-Input-and-Randomness functionality)**.** *Let* $\mathsf{SIG} = (G_{\mathsf{SIG}}, S_{\mathsf{SIG}}, V_{\mathsf{SIG}})$ *be a signature scheme with security parameter* $k \in \mathbb{N}$.[3] *Given* sk_1 *and* sk_2 *(generated by* $G_{\mathsf{SIG}}(1^k)$*), the* signed-input-and-randomness (SIR) *functionality with respect to* sk_1 *and* sk_2*, denoted* $O_{\mathsf{sk}_1,\mathsf{sk}_2}$*, is given by the probabilistic machine defined as follows.*

On input (x, s) *where* $x \in \{0,1\}^*$ *and* $s \geq 0$*,* $O_{\mathsf{sk}_1,\mathsf{sk}_2}$ *does the following:*

1. *If* $s > 0$, $\sigma \leftarrow S_{\mathsf{SIG}}(\mathsf{sk}_1, (x, r))$*, where* $r \leftarrow \{0,1\}^s$.
2. *If* $s = 0$, $\sigma \leftarrow S_{\mathsf{SIG}}(\mathsf{sk}_2, (x, \epsilon))$.
3. *Output* (r, σ).

Our main technical result is constructing APHA systems from constant-round public-coin universal arguments and signature schemes:

Theorem 4.3.1 (APHA from universal arguments and signatures)**.** *APHA systems whose oracle is signed-input-and-randomness can be built from any signature scheme and (public-coin, constant-round) universal arguments.*

Such public-coin, constant-round universal arguments are known to exist if collision-resistant hashing schemes exist [15, Theorem 1.1], and likewise for signatures schemes (see Section 2.3.5). We thus deduce the existence of APHA systems under a mild, generic assumption:

Corollary 1 (Existence of APHA systems)**.** *Assuming the existence of collision-resistant hashing schemes, there exist APHA systems whose oracle is signed-input-and-randomness.*

Let us proceed to prove Theorem 4.3.1 by constructing an APHA system, following the intuition presented in Section 4.1. The oracle generator G_{APHA} is constructed as follows.

Construction 4.3.2 (G_{APHA})**.** The oracle generator G_{APHA}, on input a security parameter $k \in \mathbb{N}$, does the following:

1. $(\mathsf{vk}_1, \mathsf{sk}_1) \leftarrow G_{\mathsf{SIG}}(1^k)$.
2. $(\mathsf{vk}_2, \mathsf{sk}_2) \leftarrow G_{\mathsf{SIG}}(1^k)$.
3. $\mathsf{vk} \equiv (\mathsf{vk}_1, \mathsf{vk}_2)$.
4. $O \equiv O_{\mathsf{sk}_1,\mathsf{sk}_2}$, where $O_{\mathsf{sk}_1,\mathsf{sk}_2}$ is (the description of) the SIR functionality with respect to sk_1 and sk_2.
5. Output (O, vk).

To prove $y \in S_{\mathcal{U}}$, we will not invoke universal arguments directly on the instance $y = (M, x, t)$, but rather on an a slightly larger *augmented instance* $y_{\mathsf{aug}} = (M_{\mathsf{aug}}, x_{\mathsf{aug}}, t_{\mathsf{aug}})$. The augmented decider machine M_{aug} invokes M to check an (alleged) witness w for y, and also verifies an (alleged) signature on y and w. (The prover will be forced to query the oracle on w in order to obtain such a signature, and this will facilitate extraction of the witness.) Let us define the subroutine AUG that maps y to y_{aug}:

Construction 4.3.3 (AUG)**.** Let $p(k, m)$ be a polynomial that bounds the running time of V_{SIG} with security parameter k on messages of length at most m. Fix a security parameter $k \in \mathbb{N}$ and let $(O, \mathsf{vk}) \in G_{\mathsf{APHA}}(1^k)$ and parse vk as $(\mathsf{vk}_1, \mathsf{vk}_2)$. Let $y = (M, x, t)$ be an instance, and let σ be an (alleged) signature on a witness for y. The subroutine AUG, on input $(\mathsf{vk}_2, \sigma, y)$, does the following:

1. $x_{\mathsf{aug}} \equiv (\mathsf{vk}_2, \sigma, y)$.
2. $t_{\mathsf{aug}} \equiv t + p(k, m)$ where $m \equiv \left|((\text{“}\texttt{inst-wit}\text{”}, y, 1^t), \epsilon)\right|$.
3. Define M_{aug} to be the machine that, on input (x_{aug}, w), works as follows
   (a) Let b_1 be the output of $V_{\mathsf{SIG}}(\mathsf{vk}_2, ((\text{“}\texttt{inst-wit}\text{”}, y, w), \epsilon), \sigma)$.
   (b) Let b_2 be the output of $M(x, w)$ after running for t steps.
   (c) Output $b_1 \wedge b_2$.
4. Output $y_{\mathsf{aug}} \equiv (M_{\mathsf{aug}}, x_{\mathsf{aug}}, t_{\mathsf{aug}})$.

---

[2]The need for two separate keys arises for technical reasons of avoiding thorny dependencies across the transitions in the proof; see Section 4.4 and Appendix A.

[3]As mentioned in Section 2.3.5, in this thesis we always consider signature schemes where the signature length depends only on the security parameter.

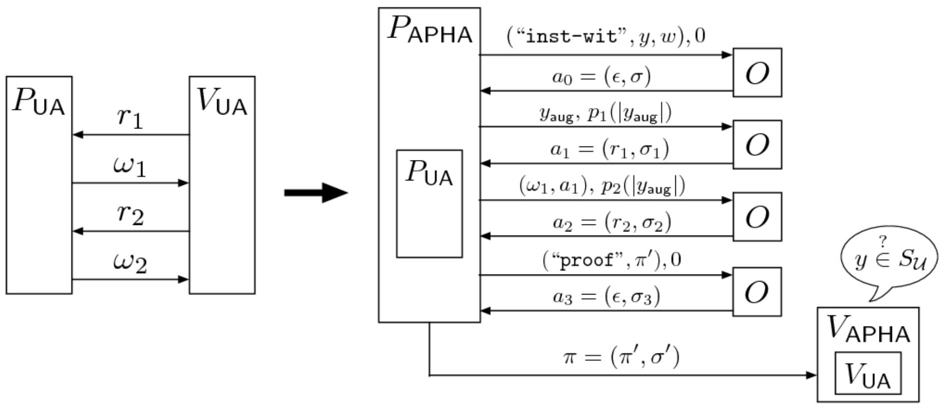

Figure 4-1: Diagram for the construction of P_{APHA}; recall that $\tau \equiv (a_1, \omega_1, a_2, \omega_2)$ and $\pi' \equiv (\sigma, \tau)$.

We proceed to describe the construction of the prover P_{APHA} and verifier V_{APHA}, given a universal-argument system $(P_{\mathsf{UA}}, V_{\mathsf{UA}})$. Let p_1 and p_2 be polynomials such that, given an instance y of length n, the first message of V_{UA} has length $p_1(n)$ and the second message of V_{UA} has length $p_2(n)$. (For convenience, the construction here is specialized to the 4-message universal arguments protocol of Barak and Goldreich [15]. It naturally generalizes to any *constant-round public-coin* protocol.)

Construction 4.3.4 (P_{APHA})**.** Fix a security parameter k and let $(O, \mathsf{vk}) \in G_{\mathsf{APHA}}(1^k)$. Let $y = (M, x, t)$ be an instance and w be a string, supposedly such that $(y, w) \in R_{\mathcal{U}}$. The prover $P^O_{\mathsf{APHA}}(\mathsf{vk}, y, w)$ does the following:

1. *Obtain a signature of the witness.* Call O with query $q_0 \equiv \big((\text{“}\texttt{inst-wit}\text{”}, y, w), 0\big)$ to obtain answer $a_0 = (\epsilon, \sigma)$.
2. *Compute the augmented instance.* Parse vk as $(\mathsf{vk}_1, \mathsf{vk}_2)$; compute $y_{\mathsf{aug}} \leftarrow \mathsf{AUG}(\mathsf{vk}_2, \sigma, y)$.
3. *Simulate V_{UA}'s first message.* Call O with query $q_1 \equiv \big(y_{\mathsf{aug}}, p_1(|y_{\mathsf{aug}}|)\big)$ to obtain answer $a_1 = (r_1, \sigma_1)$.
4. *Compute P_{UA}'s first message.* Execute the first step of $P_{\mathsf{UA}}(y_{\mathsf{aug}}, w)$, using r_1 as the verifier's first message, to obtain ω_1, the prover's first response.
5. *Simulate V_{UA}'s second message.* Call O with query $q_2 = \big((\omega_1, a_1), p_2(|y_{\mathsf{aug}}|)\big)$ to obtain answer $a_2 = (r_2, \sigma_2)$.
6. *Compute P_{UA}'s second message.* Continue the above execution of $P_{\mathsf{UA}}(y_{\mathsf{aug}}, w)$, using r_2 as the verifier's second message, to obtain ω_2, the prover's second (and last) response.
7. *Package the signature and (part of) the transcript into a preliminary proof string.* Define $\pi' \equiv (\sigma, \tau)$, where $\tau \equiv (a_1, \omega_1, a_2, \omega_2)$.
8. *Obtain a signature on the instance and preliminary proof.* Call O with query $q_3 \equiv \big((\text{“}\texttt{proof}\text{”}, \pi'), 0\big)$ to obtain answer $a_3 = (\epsilon, \sigma')$.
9. *Output the signed proof.* Output $\pi \equiv (\pi', \sigma')$.

Construction 4.3.5 (V_{APHA})**.** Fix a security parameter k and let $(O, \mathsf{vk}) \in G_{\mathsf{APHA}}(1^k)$. Let $y = (M, x, t)$ be an instance and let π be an (alleged) proof string for the claim “$y \in S_{\mathcal{U}}$”. The verifier $V_{\mathsf{APHA}}(\mathsf{vk}, y, \pi)$ does the following:

1. Parse vk as $(\mathsf{vk}_1, \mathsf{vk}_2)$.
2. Parse π as (π', σ'), where $\pi' = (\sigma, \tau)$, $\tau = (a_1, \omega_1, a_2, \omega_2)$, $a_1 = (r_1, \sigma_1)$, and $a_2 = (r_2, \sigma_2)$.
3. *Verify that the proof signature is valid.* Check that $V_{\mathsf{SIG}}\big(\mathsf{vk}_2, ((\text{“}\texttt{proof}\text{”}, \pi'), \epsilon), \sigma'\big) = 1$.
4. *Compute the augmented instance.* Compute $y_{\mathsf{aug}} \leftarrow \mathsf{AUG}(\mathsf{vk}_2, \sigma, y)$.
5. *Verify that the transcript is consistent.* Check that:
   (a) $V_{\mathsf{SIG}}\big(\mathsf{vk}_1, (y_{\mathsf{aug}}, r_1), \sigma_1\big) = 1$ and $|r_1| = p_1(|y_{\mathsf{aug}}|)$;
   (b) $V_{\mathsf{SIG}}\big(\mathsf{vk}_1, ((\omega_1, a_1), r_2), \sigma_2\big) = 1$ and $|r_2| = p_2(|y_{\mathsf{aug}}|)$.
6. *Verify that the transcript is convincing.* Check that the third step of $V_{\mathsf{UA}}(y_{\mathsf{aug}})$, using r_1 and r_2 as the verifier's first and second messages, and using ω_1 and ω_2 as the prover's first and second messages, accepts.

4.4 Correctness of the APHA construction

We complete the proof of Theorem 4.3.1 by showing that the constructions presented in Section 4.3 indeed constitute an APHA system.

The properties of efficient verifiability and completeness via a relatively-efficient prover easily follow by construction. The remaining property, list extraction, is fulfilled by the following list extractor LE:

Construction 4.4.1 (LE). The list extractor LE, on input a prover-oracle transcript $[\![\widetilde{P}(\mathsf{vk}), O]\!]$, does the following:

1. Create an empty list $\mathsf{extlist}$.
2. In the transcript $[\![\widetilde{P}(\mathsf{vk}), O]\!]$, let $(q_1, a_1), \ldots, (q_l, a_l)$ be the query-answer pairs for which the query is of the form $q_i = ((\text{"}\mathtt{proof}\text{"}, \pi'_i), 0)$.
3. For each $i \in [l]$, do the following:
   (a) Parse π'_i as (σ_i, τ_i) and a_i as (ϵ, σ'_i).
   (b) Find some (q, a) in $[\![\widetilde{P}(\mathsf{vk}), O]\!]$ such that $a = (\epsilon, \sigma_i)$ and $q = ((\text{"}\mathtt{inst\text{-}wit}\text{"}, y, w), 0)$. (If none exists, abort and output $\perp$.)
   (c) Add $(y, (\pi'_i, \sigma'_i), w)$ to $\mathsf{extlist}$.
4. Output $\mathsf{extlist}$.

Claim 4.4.2. *The list extractor* LE *from Construction 4.4.1 fulfills the list-extraction property for the triple* $(G_{\mathsf{APHA}}, P_{\mathsf{APHA}}, V_{\mathsf{APHA}})$ *constructed in Section 4.3.*

The following is an overview of the proof structure; the full proof is given in Appendix A.

Proof sketch. To prove the success of LE, we define a sequence of intermediate constructions of increasing power, starting from universal-argument systems (with a weak proof-of-knowledge property) and ending at APHA systems (with a full-fledged list-extraction property). Each construction is built via black-box access to the functionality proved for the preceding one.

First construction: adaptivity. Starting from a universal-argument system $(P_{\mathsf{UA}}, V_{\mathsf{UA}})$, which has a weak proof-of-knowledge (PoK) property, we show how to construct a pair of machines (P_0, V_0) for which the weak PoK property holds even when the prover itself adaptively chooses the claimed instance y. The prover has oracle access to a functionality O' that outputs random strings upon request; the prover interacts with O', and then outputs an instance y and a proof string π_0 for the claim "$y \in S_{\mathcal{U}}$". When verifying the output of the prover, we allow V_0 to see all the query-answer pairs of the prover to the functionality O'.

V_0 works by requiring a (possibly cheating) prover $\widetilde{P}_0$ to produce a transcript of the universal-argument protocol which V_{UA} would have accepted, and, moreover, by verifying that the public-coin challenges in the transcript were obtained by $\widetilde{P}_0$, in the right order, as answers from O'.

We show that whenever a prover $\widetilde{P}_0$ convinces V_0 on some instance y of its choice, $\widetilde{P}_0$ can be converted into a cheating prover $\widetilde{P}_{\mathsf{UA}}$ that convinces V_{UA} on y, from which a witness for "$y \in S_{\mathcal{U}}$" can be extracted using the universal-argument knowledge extractor E_{UA}. We thus obtain a knowledge extractor E_0 for (P_0, V_0).

Second step: stateless oracle. Starting from the pair of machines (P_0, V_0), we show how to construct a triple of machines (G_1, P_1, V_1) for which the weak PoK property still holds. This time, the prover has oracle access to a stateless probabilistic oracle O'' generated by G_1, instead of the aforementioned stateful oracle O'. On input x, O'' outputs a random string r together with a signature on (x, r). When verifying the output of the prover, this time V_1 does not see the query-answer pairs of the prover to O''. Instead, it verifies the signatures in the transcript provided by the prover, to be convinced that the queries were indeed made to O''.

That is, V_1 requires a (possibly cheating) prover $\widetilde{P}_1$ to produce a proof string that V_0 would have accepted, along with corresponding signatures that are valid under the verification key of O''.

As before (but by a different technique), we show that whenever a prover $\widetilde{P}_1$ convinces V_1 on some instance y of its choice, $\widetilde{P}_1$ can be converted into a prover $\widetilde{P}_0$ that convinces V_0 on y, from which a witness for "$y \in S_{\mathcal{U}}$" can be extracted using the knowledge extractor E_0. We thus obtain a knowledge extractor E_1 for (G_1, P_1, V_1).

Third step: list extraction. Starting from (G_1, P_1, V_1), we show how to construct a triple of machines $(G_{\mathsf{APHA}}, P_{\mathsf{APHA}}, V_{\mathsf{APHA}})$ that is an APHA system. Similarly to the previous step, provers for V_{APHA} have access to a stateless signed-input-and-randomness oracle O (following Definition 4.3.1), generated by G_{APHA}; however, $(G_{\mathsf{APHA}}, P_{\mathsf{APHA}}, V_{\mathsf{APHA}})$ satisfies a PoK property in a much stronger sense, specified by the APHA list-extraction property and its list-extractor LE. This "knowledge boosting" relies on forcing the prover to explicitly state its witness in some query to O.

V_{APHA} works by requiring a (possibly cheating) prover $\widetilde{P}$ to produce a proof string that V_1 would have accepted; however, the proof string should not be about the claim "$y \in S_{\mathcal{U}}$" (for some instance y chosen by the prover), but about some related claim "$y_{\mathsf{aug}} \in S_{\mathcal{U}}$", where y_{aug} is derived from y. Essentially, the prover can convince V_1 that "$y_{\mathsf{aug}} \in S_{\mathcal{U}}$" only if it knows a signature, that verifies under the verification key of O, for a valid witness that "$y \in S_{\mathcal{U}}$". Thus, the prover is forced to explicitly query O on such a witness — and this query can be found by the list extractor.

Crucially, the knowledge extractor E_1 is not invoked by the APHA list extractor LE; rather, E_1 is used just in the proof of correctness of LE, in a reduction from failure of LE to forgeability of signatures.[4] Since signatures are forgeable with negligible probability, the polynomial loss of the weak PoK amounts to just a small increase in the security parameter.

Thus, we show that whenever V_{APHA} accepts the output of $\widetilde{P}$ we can (with all but for negligible probability) efficiently find a valid witness for the instance output by $\widetilde{P}$ among the queries of $\widetilde{P}$ to O, which is the main ingredient of the proof of correctness of the list extractor LE. $\square$

4.5 Realizability of an assisted prover

Our construction of APHA systems (and eventually PCD systems) assumes black-box access to a single, fixed functionality: signed-input-and-randomness. This functionality is stateless, and is parametrized by a single concise secret (the signing key sk).

Communication. The communication between the prover and the oracle O is as low as one could hope for given our approach to knowledge extraction (see Section 4.1): linear in the witness size $|w|$, and polynomial in the instance $|y|$ and security parameter k. Moreover, only four queries are needed. Note that the total communication is linear in the length of the original witness w for the statement $y = (M, x, t) \in S_{\mathcal{U}}$, rather than (as in non-interactive CS proofs) a much longer PCP witness, whose length is polynomial in the halting time of $M(x, w)$.

Computation. Using the hash-then-sign approach, and typical hash function constructions, the computational complexity of the signed-input-and-randomness functionality is essentially linear in its communication complexity size and polynomial in the security parameter.

Realization. How would such an oracle be provided in reality? As noted earlier, similar requisites arose in related works [83][90][110][32][45][101]. One well-studied option is to use a secure hardware token that is tamper-proof and leak-proof. Indeed, similar signing tokens are already prescribed by German law [48]. Similarly, the functionality can be embedded in cryptographic coprocessors, TPM chips, and general-purpose smartcard such as TEMs [41]. Alternatively, one may hope that this specific functionality can be obfuscated, either in the strict virtual-box-box sense [16] or (for real-world security applications) in some heuristic sense. Lastly, the functionality can be realized via standard MPC techniques between multiple parties, tokens, or services, if the requisite fraction of honest participants is available.

Removing randomness. The randomness of the signed-input-and-randomness functionality is not essential: one could replace the fresh random bits with pseudorandom bits obtained by a pseudorandom function (whose seed is kept secret) applied to the input. In this way, one only has to trust the token to hide its secret bits (the signing key and the seed) and to operate correctly, but not to also generate random bits. Indeed, our constructions do not require the randomness from the token to be fresh for repeated queries with the same input, and security holds even if the randomness comes from a pseudorandom function. (Intuitively, this holds because, even with a truly randomized oracle, adversaries can always replay old answers.)

[4] This is similar in spirit to the extractor abort lemma of Chandran et al. [32].

Chapter 5

Proof-Carrying Data Systems

We define and construct proof-carrying data (PCD) systems, the cryptographic primitive that realizes the framework of proof-carrying data.

This chapter is organized as follows:

- Section 5.1 Formal definition of compliance for distributed computations.
- Section 5.2 Definition of PCD systems and discussion of their properties.
- Section 5.3 Generic construction of PCD systems from APHA systems.
- Section 5.4 Proof sketch of the construction's correctness (for full proof see Appendix B).

5.1 Compliance of computation

We begin by specifying our notion of a distributed computation, which we call a distributed computation transcript.

For a directed graph $G = (V, E)$, and vertex $v \in V$, $\mathsf{in}(v)$ denotes the incoming edges of v, $\mathsf{out}(v)$ denotes its outgoing edges, $\mathsf{parents}(v)$ denotes its neighbors across $\mathsf{in}(v)$, and $\mathsf{children}(v)$ denotes its neighbors across $\mathsf{out}(v)$.

Definition 5.1.1 (Distributed computation transcript)**.** *A* distributed computation transcript *(abbreviated* transcript*) is a triple* $\mathsf{DC} = (G, \mathsf{code}, \mathsf{data})$ *representing a directed acyclic multi-graph* $G = (V, E)$ *with labels* code *on the vertices and labels* data *on the edges.*

Vertices represent the computation of programs, and edges represent messages sent between these programs. Each non-source vertex v *is labeled by* $\mathsf{code}(v)$*, which is the code of the program at* v*. Each edge* (u, v) *is labeled by* $\mathsf{data}(u, v)$*, which is the data that is (allegedly) output by the program of* u *and is sent by* u *to* v *as one of the inputs to the program of* v*. Each source vertex has a single outgoing edge, carrying an input to the distributed computation; there are no programs at sources, so we set their label to* $\perp$*. The final outputs of the distributed computation is the data carried along edges going into sinks. (See Figure 5-1 for an illustration.)*

A transcript captures the propagation of information via messages in the distributed computation, and thus the graph is acyclic by definition. A party performing several steps of computations on different inputs at different times is represented by corresponding distinct vertices.

A transcript where messages are augmented by proof strings is called an *augmented* distributed computation transcript.

Definition 5.1.2 (Augmented distributed computation transcript)**.** *An* augmented distributed computation transcript *(abbreviated* augmented transcript*) is a quadruple* $\mathsf{ADC} = (G, \mathsf{code}, \mathsf{data}, \mathsf{proof})$ *such that* $(G, \mathsf{code}, \mathsf{data})$ *is a transcript, and* proof *is an additional labeling on the edges of* G*, specifying proof strings carried along those edges. (See Figure 5-1 for an illustration.)*

Given a distributed computation transcript $\mathsf{DC} = (G, \mathsf{code}, \mathsf{data})$, at times we need to consider the part of the transcript up to a certain edge.

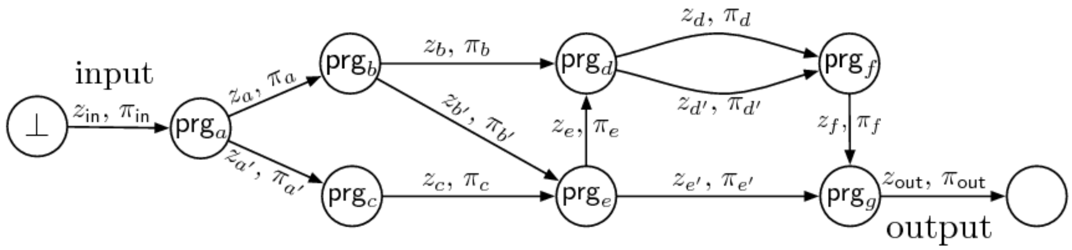

Figure 5-1: Example of an *augmented* distributed computation transcript. Programs are denoted by prg's, data by z's, and proof strings by π's. The corresponding (non-augmented) distributed computation transcript is given by the same illustration, with the proof strings omitted.

Definition 5.1.3. *For an edge $(u, v) \in E$, we define the* transcript of DC up to (u, v), *denoted* $\mathsf{DC}|_{(u,v)} = (G', \mathsf{code}', \mathsf{data}')$*, to be the labeled subgraph induced by the subset of vertices consisting of v, u, and all ancestors of u.*

Next, we define what we mean for a distributed computation to be *compliant*, which is our notion of "correctness with respect to a given specification". We capture compliance via an efficiently computable predicate $\mathbf{C}$ that is required to hold true at each vertex, when given the program of the vertex together with its inputs and (alleged) outputs.

Definition 5.1.4 ($\mathbf{C}$-compliance)**.** *A* compliance predicate $\mathbf{C}$ *is a polynomial-time computable predicate on strings (we will abuse notation and identify $\mathbf{C}$ with the polynomial-time Turing machine that computes it). A distributed computation transcript $\mathsf{DC} = (G, \mathsf{code}, \mathsf{data})$ is* $\mathbf{C}$-compliant *if, for every vertex $v \in V$, it holds that*

$$\mathbf{C}\Big(\mathsf{data}\big(\mathsf{in}(v)\big), \mathsf{code}(v), \mathsf{data}\big(\mathsf{out}(v)\big)\Big) = 1 \ ,$$

where $\mathsf{data}(\mathsf{in}(v))$ denotes the list of data labels on v's parents, and analogously for $\mathsf{data}(\mathsf{out}(v))$.

Alternatives. One may consider stronger forms of compliance. For example, the compliance predicate could get as extra inputs the graph G and the identity of the vertex v (so that the compliance predicate "knows" which vertex in the graph it is examining). Stronger still, the compliance predicate could be *global*, and get as input the whole transcript $\mathsf{DC} = (G, \mathsf{code}, \mathsf{data})$.

However, our goal is to realize PCD in a dynamic setting, where future computations have not happened yet (and might even be unknown) and past computations have been long forgotten, so that compliance must indeed be decided locally. Therefore, we choose a *local* compliance predicate, which only gets as input the information that is locally available at a vertex, i.e., the program of the vertex together with its inputs and (alleged) outputs. (Of course, the preceding discussion does not prevent one from choosing a compliance predicate $\mathbf{C}$ that forces vertices to keep track of certain information, such as history of past computations or network topology, for the purpose of deciding compliance. Such additional information would be appended as "meta-data" to the messages sent between vertices.)

5.2 Definition of PCD systems

We proceed to define proof-carrying data systems, starting with their structure and an informal description of their properties, and then following with a formal definition.

5.2.1 Structure of PCD systems

A PCD system is a triple of machines $(G_{\mathsf{PCD}}, P_{\mathsf{PCD}}, V_{\mathsf{PCD}})$ that works as follows:

– The *PCD oracle generator* G_{PCD}: for a security parameter k, $G_{\mathsf{PCD}}(1^k)$ outputs the description of a probabilistic[1] stateless oracle O, together with O's verification key vk.
– The *PCD prover* P_{PCD}: let vk be a verification key, let $\mathbf{C}$ be a compliance predicate, and let prg be a program with (alleged) output z_{out} and two inputs z_{in_1} and z_{in_2} with corresponding proof strings π_{in_1}

[1] The oracle can be derandomized; see Section 4.5.

and π_{in_2} (see Figure 5-2); then $P^O_{\mathsf{PCD}}(\mathsf{vk}, \mathbf{C}, z_{\mathsf{out}}, \mathsf{prg}, z_{\mathsf{in}_1}, \pi_{\mathsf{in}_1}, z_{\mathsf{in}_2}, \pi_{\mathsf{in}_2})$ outputs a proof string π_{out} for the claim that z_{out} is an output consistent with a $\mathbf{C}$-compliant transcript.[2]

– The *PCD verifier* V_{PCD}: for a verification key vk, a compliance predicate $\mathbf{C}$, an output z_{out}, and a proof string π_{out}, $V_{\mathsf{PCD}}(\mathsf{vk}, \mathbf{C}, z_{\mathsf{out}}, \pi_{\mathsf{out}})$ accepts if π_{out} convinces him that z_{out} is an output consistent with a $\mathbf{C}$-compliant transcript.

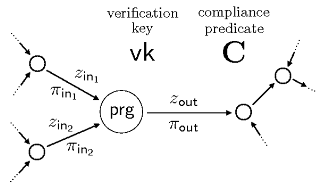

Figure 5-2: The new proof string π_{out} for the output data z_{out} is computed using the PCD prover P_{PCD}, with oracle access to O and input $(\mathsf{vk}, \mathbf{C}, z_{\mathsf{out}}, \mathsf{prg}, z_{\mathsf{in}_1}, \pi_{\mathsf{in}_1}, z_{\mathsf{in}_2}, \pi_{\mathsf{in}_2})$.

Using these algorithms, a distributed computation transcript DC is dynamically compiled into an *augmented* distributed computation transcript ADC by generating and adding "on the fly" a proof string to each edge (see Figure 5-1). The process of generating proof strings is defined inductively, by having each (internal) vertex v in the transcript DC use the PCD prover P_{PCD} to produce a new proof string π_{out} for its output z_{out} (given its inputs, their inductively generated proof strings, its program, and output). The proof strings generated in this way form the additional label proof on the edges in the resulting augmented transcript ADC.

More precisely, focusing on a particular edge $(u, v) \in E$ in the transcript $\mathsf{DC} = (G, \mathsf{code}, \mathsf{data})$, we recursively define the process of *computing proof strings in* DC *up to* (u, v), and informally denote this process by $\mathsf{PROOFGEN}(\mathsf{DC}|_{(u,v)})$ (or $\mathsf{PROOFGEN}^O(\mathsf{vk}, \mathbf{C}, \mathsf{DC}|_{(u,v)})$ when being precise); this process generates the proof string label $\mathsf{proof}' \colon E' \to \{0,1\}^*$ for $\mathsf{DC}' = \mathsf{DC}|_{(u,v)}$, the transcript of DC up to (u, v).

Specifically, the process $\mathsf{PROOFGEN}(\mathsf{DC}')$ is defined as follows. Initially, proof strings on the outgoing edges of sources are set to $\perp$. Then, taking each non-source non-sink vertex $w \in V'$ in some topological order,[3] let w_{in_1} and w_{in_2} be the two parents of w, and let w_{out} be its single child in DC'. Let $z_{\mathsf{in}_1} = \mathsf{data}'(w_{\mathsf{in}_1}, w)$, $\pi_{\mathsf{in}_1} = \mathsf{proof}'(w_{\mathsf{in}_1}, w)$, $z_{\mathsf{in}_2} = \mathsf{data}'(w_{\mathsf{in}_2}, w)$, $\pi_{\mathsf{in}_2} = \mathsf{proof}'(w_{\mathsf{in}_2}, w)$, $\mathsf{prg} = \mathsf{code}'(w)$, and $z_{\mathsf{out}} = \mathsf{data}'(w, w_{\mathsf{out}})$. Then, recursively compute

$$\pi_{\mathsf{out}} \leftarrow P^O_{\mathsf{PCD}}(\mathsf{vk}, \mathbf{C}, z_{\mathsf{out}}, \mathsf{prg}, z_{\mathsf{in}_1}, \pi_{\mathsf{in}_1}, z_{\mathsf{in}_2}, \pi_{\mathsf{in}_2}) \ ,$$

and define $\pi_{\mathsf{out}} \equiv \mathsf{proof}'(w, w_{\mathsf{out}})$. The final output z of $\mathsf{DC}' = \mathsf{DC}|_{(u,v)}$ has the proof string $\pi = \mathsf{proof}'(u, v)$, and now all the values of proof' have been assigned.

5.2.2 Properties of PCD systems (intuitive)

The triple $(G_{\mathsf{PCD}}, P_{\mathsf{PCD}}, V_{\mathsf{PCD}})$ must satisfy three properties. Analogously to APHA systems, the first two bound the complexity of proving and verifying, and the third is a strong proof-of-knowledge property (which, in particular, implies computational soundness). These are adapted to the context of distributed computation transcripts.

First, proof strings generated by the PCD prover should be *efficiently verifiable* by the PCD verifier: V_{PCD} halts in time that is polynomial in the security parameter k, the size of the description of $\mathbf{C}$, the length of z, and the logarithm of the time it took to generate π. (Our parameters are even better; see Definition 5.2.1.)

[2] Without loss of generality, we restrict our attention to transcripts for which programs have exactly two inputs and one output.

[3] Formally, since G is acyclic, we are oblivious to the choice of temporal order. In reality the proof strings are computed on-the-fly according to the temporal order by which the data messages are generated; by causality, this order is topological.

Second, the PCD prover should be able to *prove true statements using a reasonable amount of time.* Whenever it is indeed the case that a transcript DC is $\mathbf{C}$-compliant, if the above recursive process repeatedly uses P_{PCD} to generate a proof string π for the data z on some edge, then (z, π) are indeed accepted by V_{PCD}. Moreover, the above recursive step runs in time that is polynomial in the security parameter k, the size of the description of $\mathbf{C}$, and the time it took to to verify $\mathbf{C}$-compliance at each node.

Third, *no efficient prover should be able to prove false statements*: given a compliance predicate $\mathbf{C}$ and an output string z that is not consistent with *any* $\mathbf{C}$-compliant transcript, no cheating prover circuit $\widetilde{P}$ of size $\mathsf{poly}(k)$ can generate a convincing proof string π for z (except with negligible probability, over the randomness of the oracle and its verification key).

In order to preserve security for distributed computations that use cryptographic functionality that is only computationally secure, we actually require a stronger property: a *proof-of-knowledge* property. A proof string π augmenting a piece of data z not only attests to the fact that z_{out} is consistent with the output of *some* $\mathbf{C}$-compliant transcript, but also guarantees the following. For any (possibly cheating) prover circuit $\widetilde{P}$ of size $\mathsf{poly}(k)$, there exists a knowledge extractor E_{PCD} circuit of size $\mathsf{poly}(k)$ such that, for any output string z, if $\widetilde{P}$ produces a sufficiently convincing proof string π for z, then E_{PCD} can *extract* from $\widetilde{P}$ a $\mathbf{C}$-compliant transcript DC that has final output z.[4]

5.2.3 Properties of PCD systems (formal)

We proceed to capture the above intuition more formally.

First, we discuss how to characterize the PCD prover's complexity. Recall that PCD provers are invoked in a recursive structure that follows a given distributed computation transcript. At any given vertex in the transcript, the PCD prover's complexity *essentially* depends (polynomially) only in the size of the local inputs (i.e., the vertex's input data, program's code, and alleged output data) and the time it takes to verify compliance of these local inputs.

However, the exact complexity of the PCD prover *does* depend on the particular computation leading up to the input data (and the data's proof strings) to the given vertex.[5] Therefore, in order to precisely quantify the PCD prover's complexity at that vertex, we define a recursive function over the history preceding the vertex. The recursive function is defined as follows:

Definition 5.2.1 (Recursive Time up to an Edge)**.** *Let p be a positive polynomial, k a security parameter, $\mathbf{C}$ a compliance predicate, and DC a transcript. Given $(u, v) \in E$, we define the* recursive time of $\mathsf{DC}|_{(u,v)}$, *denoted $T_p(k, \mathbf{C}, \mathsf{DC}|_{(u,v)})$, to be the following recursively-defined function:*

– *If u is a source vertex,*

$$T_p\left(k, \mathbf{C}, \mathsf{DC}|_{(u,v)}\right) \equiv p\big(k + |\langle \mathbf{C} \rangle|\big) \ .$$

– *Otherwise,*

$$T_p\left(k, \mathbf{C}, \mathsf{DC}|_{(u,v)}\right) \equiv \mathsf{time}_{\mathbf{C}}\Big(\mathsf{data}(\mathsf{in}(u)), \mathsf{code}(u), \mathsf{data}(\mathsf{out}(u))\Big) + \sum_{u' \in \mathsf{parents}(u)} p\Big(k + |\langle \mathbf{C} \rangle| + |\mathsf{data}(u', u)| + \log T_p\big(k, \mathbf{C}, \mathsf{DC}|_{(u',u)}\big)\Big) \ .$$

Roughly, $T_p\left(k, \mathbf{C}, \mathsf{DC}|_{(u,v)}\right)$ indicates the time that the node u spends to locally verify compliance (corresponding to the $\mathsf{time}_{\mathbf{C}}$ term in Definition 5.2.1) *and* the time necessary to verify the proof strings received from previous vertices (corresponding to the summation term in Definition 5.2.1).

Note that since the PCD prover's complexity will be phrased in terms of the above recursive function, the PCD verifier's complexity will also be, because the PCD verifier is supposed to examine proof strings output by PCD provers. See below in Definition 5.2.2 for how exactly the above recursive function affects the complexity of the PCD prover and PCD verifier.

We observe that the essential property of the above recursive function is that the cost of past computation decays as an *iterated logarithm* at every aggregation step, and thus converges very quickly. This means

[4]Our construction attains a stronger definition, where a *fixed* knowledge extractor can extract from *any* convincing prover by observing only its output and its interaction with the oracle (analogously to the APHA list extraction property). We use the above weaker definition for convenience of presentation.

[5]The PCD prover's complexity at a given vertex depends on the length of the (concise) proof strings received from the parent vertices (besides depending on the time needed to check compliance). The length of these proof strings will itself essentially depend (this time *logarithmically*) on the time it took to locally check compliance at the previous vertices.

that the time it takes to generate a proof π_{out} is indeed essentially polynomial in the time it takes to merely locally check compliance, i.e., to compute $\mathbf{C}((z_{\mathsf{in}_1}, z_{\mathsf{in}_2}), \mathsf{prg}, (z_{\mathsf{out}}))$; and the verification time of a proof π_{out} is logarithmic in that.

We can now state the definition of PCD systems.

Definition 5.2.2 (PCD System)**.** *A* proof-carrying data system *with security parameter k is a triple of polynomial-time machines $(G_{\mathsf{PCD}}, P_{\mathsf{PCD}}, V_{\mathsf{PCD}})$, where G_{PCD} is probabilistic, P_{PCD} is deterministic with oracle access, and V_{PCD} is deterministic, that satisfies the following conditions:*

– Efficient verification: *There exists a positive polynomial p such that for every $k \in \mathbb{N}$, $(O, \mathsf{vk}) \in G_{\mathsf{PCD}}(1^k)$, compliance predicate $\mathbf{C}$, transcript DC, edge $(u, v) \in E$ with label $z = \mathsf{data}(u, v)$, and proof string π,*

$$\mathsf{time}_{V_{\mathsf{PCD}}}(\mathsf{vk}, \mathbf{C}, z, \pi) \le p\left(k + |\langle \mathbf{C} \rangle| + |z| + \log T_p\left(k, \mathbf{C}, \mathsf{DC}|_{(u,v)}\right)\right) \ .$$

In particular, the proof string is short: $|\pi| \le p\left(k + |\langle \mathbf{C} \rangle| + |z| + \log T_p\left(k, \mathbf{C}, \mathsf{DC}|_{(u,v)}\right)\right)$.

– Completeness via a relatively-efficient prover: *For every $k \in \mathbb{N}$, compliance predicate $\mathbf{C}$, $\mathbf{C}$-compliant transcript DC, and edge $(u, v) \in E$ with label $z = \mathsf{data}(u, v)$,*

$$\Pr\Big[V_{\mathsf{PCD}}(\mathsf{vk}, \mathbf{C}, z, \pi) = 1 \;\Big|\; (O, \mathsf{vk}) \leftarrow G_{\mathsf{PCD}}(1^k) \; ; \; \mathsf{proof}' \leftarrow \mathsf{PROOFGEN}^{O}(\mathsf{vk}, \mathbf{C}, \mathsf{DC}|_{(u,v)}) \; ; \; z \leftarrow \mathsf{proof}'(u, v)\Big] = 1$$

(where the probability is taken over the internal randomness of G_{PCD} and O); recall that $\mathsf{PROOFGEN}$ *is the process of computing proof strings in DC up to (u, v), described above. Furthermore, there exists a positive polynomial p such that for every $k \in \mathbb{N}$, $(O, \mathsf{vk}) \in G_{\mathsf{PCD}}(1^k)$, $\mathbf{C}$-compliant computation DC, and edge $(u, v) \in E$ with label $z = \mathsf{data}(u, v)$, letting u_1 and u_2 be the two parents of u, $\mathsf{proof}' \leftarrow \mathsf{PCD}^{O}(\mathsf{vk}, \mathbf{C}, \mathsf{DC}|_{(u,v)})$, $z_1 = \mathsf{data}(u_1, u)$, $z_2 = \mathsf{data}(u_2, u)$, $\pi_1 = \mathsf{proof}'(u_1, u)$, $\pi_2 = \mathsf{proof}'(u_2, u)$, and $\mathsf{prg} = \mathsf{code}(u)$,*

$$\mathsf{time}_{P^{O}_{\mathsf{PCD}}}(\mathsf{vk}, \mathbf{C}, z, \mathsf{prg}, z_1, \pi_1, z_2, \pi_2) \le p\left(k + |\langle \mathbf{C} \rangle| + |z| + T_p\left(k, \mathbf{C}, \mathsf{DC}|_{(u,v)}\right)\right) \ .$$

– Proof-of-knowledge property: *Let $k \in \mathbb{N}$. For every (possibly cheating) prover circuit $\widetilde{P}$ of size $\mathsf{poly}(k)$, there exists a* knowledge extractor *circuit E_{PCD} of size $\mathsf{poly}(k)$ such that, for every polynomial p, compliance predicate $\mathbf{C}$, output string $z \in \{0,1\}^*$, and for sufficiently large k: if $\widetilde{P}$ convinces V_{PCD} to accept z with non-negligible probability,*

$$\Pr\Big[V_{\mathsf{PCD}}(\mathsf{vk}, \mathbf{C}, z, \pi) = 1 \;\Big|\; (O, \mathsf{vk}) \leftarrow G_{\mathsf{PCD}}(1^k) \; ; \; \pi \leftarrow \widetilde{P}^{O}(\mathsf{vk}, \mathbf{C}, z)\Big] > \frac{1}{p(k)}$$

(where the probability is taken over the internal randomness of G_{PCD} and O), then E_{PCD} extracts a $\mathbf{C}$-compliant distributed computation transcript DC consistent with the final output z (i.e., $z = \mathsf{data}(u, v)$ and (u, v) is the unique incoming edge to the unique sink vertex v) with almost the same probability,

$$\Pr\Big[\mathsf{DC} \textit{ is } \mathbf{C}\textit{-compliant} \wedge u, v \in V \wedge \mathsf{DC} = \mathsf{DC}|_{(u,v)} \wedge z = \mathsf{data}(u, v) \;\Big|\; (O, \mathsf{vk}) \leftarrow G_{\mathsf{PCD}}(1^k) \; ; \; \mathsf{DC} \leftarrow E^{O}_{\mathsf{PCD}}(\mathsf{vk}, \mathbf{C}, z)\Big] > \frac{1}{p(k)} - \mu(k)$$

(where the probability is taken over the internal randomness of G_{PCD} and O), for some negligible function μ.

5.2.4 More on recursive time and PCD prover's complexity

When using a proof-carrying data system to enforce a compliance predicate $\mathbf{C}$, parties in the distributed computation incur in the computational overhead of having to generate proof strings. We briefly discuss this overhead.

From Definition 5.2.2, we know that the PCD prover's complexity is:

$$\mathsf{time}_{P^{O}_{\mathsf{PCD}}}(\mathsf{vk}, \mathbf{C}, z, \mathsf{prg}, z_1, \pi_1, z_2, \pi_2) \le p\left(k + |\langle \mathbf{C} \rangle| + |z| + T_p\left(k, \mathbf{C}, \mathsf{DC}|_{(u,v)}\right)\right) \ .$$

But how does a party that uses the PCD prover at some vertex u actually know how much work it will have to do? After all, the that party may not have been present throughout the computation leading up to vertex u, and therefore will not know the exact value of $T_p\left(k, \mathbf{C}, \mathsf{DC}|_{(u,v)}\right)$. Definition 5.2.1 already tells us that $T_p\left(k, \mathbf{C}, \mathsf{DC}|_{(u,v)}\right)$ "essentially" depends only on the time it takes to check compliance at u, because the effect of the past decays quickly. However, can we get a better handle on what "essentially" concretely means for a party a vertex u?

In general, it is hard to quantify "essentially" in simpler terms than what is already captured by Definition 5.2.1, because even asymptotic behavior of $T_p\left(k, \mathbf{C}, \mathsf{DC}|_{(u,v)}\right)$ really depends on the particular distributed computation transcript $\mathsf{DC}|_{(u,v)}$ under consideration.

Nonetheless, we can consider situations in which we do have some partial information about the computation preceding vertex u; using that information, we can then derive concrete upper bounds on the overhead required of the party at vertex u to generate proof strings. Specifically, we consider the following two scenarios:

- *Bound on total verification time is known:* We know that the *total* amount of time that it would take to verify the $\mathbf{C}$-compliance of all nodes in the computation preceding vertex u is at most S.

- *Bound on each party's verification time is known:* We know that all parties that took part in the computation preceding vertex u *each* performed a computation that can be verifier by $\mathbf{C}$ in time at most s (however, we may not know of an upper bound on the number of such parties) and, moreover, we know that the in-degree of messages to any vertex is at most d.

We derive an upper bound on $T_p\left(k, \mathbf{C}, \mathsf{DC}|_{(u,v)}\right)$ for each of the above.

Bound on total verification time is known. Suppose that we know that the $\mathbf{C}$-compliance of the whole preceding computation can be verified in time at most S.[6] Then, we can derive an upper bound on $T_p\left(k, \mathbf{C}, \mathsf{DC}|_{(u,v)}\right)$ by computing an upper bound for it when $\mathsf{DC}|_{(u,v)}$ is chosen to yield the worst asymptotic behavior.

First, note that we are actually only concerned with upper bounding $T_p\left(k, \mathbf{C}, \mathsf{DC}|_{(u',u)}\right)$ for every parent u' of u, because $\mathsf{time}_{\mathbf{C}}\big((z_1, z_2), \mathsf{prg}, (z)\big)$, which is the first term of $T_p\left(k, \mathbf{C}, \mathsf{DC}|_{(u,v)}\right)$, will be known only at computation time, and will depend on what the party at vertex u decides to do. Furthermore, without loss of generality, we can assume that u has a single parent u' that carries a proof string (and the other parent is a "base case").

We observe that $T_p\left(k, \mathbf{C}, \mathsf{DC}|_{(u',u)}\right)$ is largest when the graph $\mathsf{DC}|_{(u',u)}$ is the graph with edges $\{(u'', u'), (u', u)\}$ (i.e., u' has as its single parent a source vertex, so that the data along (u'', u') is an input to the distributed computation transcript) and u performs a single computation that takes time S. Intuitively, this is because of the iterated logarithm. In this case, $T_p\left(k, \mathbf{C}, \mathsf{DC}|_{(u',u)}\right) = \mathsf{poly}(k + |\mathbf{C}| + S)$, and we deduce that the PCD prover's complexity at vertex u is:

$$\mathsf{time}_{P^{\mathcal{O}}_{\mathsf{PCD}}}(\mathsf{vk}, \mathbf{C}, z, \mathsf{prg}, z_1, \pi_1, z_2, \pi_2) \le p\left(k + |\langle\mathbf{C}\rangle| + |z| + |z_1| + |z_2| + \mathsf{time}_{\mathbf{C}}\big((z_1, z_2), \mathsf{prg}, (z)\big) + \log S\right) \quad ,$$

which is the upper bound on the computational overhead incurred by the party at vertex u by generating the proof string.

Moreover, if we also know that vertex u can be verified to be $\mathbf{C}$-compliant in time at most S_u, then we are guaranteed to spend time at most

$$\begin{aligned}\mathsf{time}_{V_{\mathsf{PCD}}}(\mathsf{vk}, \mathbf{C}, z, \pi) &\le p\left(k + |\langle\mathbf{C}\rangle| + |z| + \log|z_1| + \log|z_2| + \log \mathsf{time}_{\mathbf{C}}\big((z_1, z_2), \mathsf{prg}, (z)\big) + \log\log S\right) \\ &\le p\left(k + |\langle\mathbf{C}\rangle| + |z| + \log S_u + \log\log S\right) \quad .\end{aligned}$$

in order to verify the proof string for message z along edge (u, v).

Bound on each party's verification time is known. Suppose that we know that all parties that took part in the computation preceding vertex u *each* performed a computation that can be verified by $\mathbf{C}$ in time at most s,[7] and that any vertex received at most d messages.

Similarly to the above discussion, we are only concerned with upper bounding $T_p\left(k, \mathbf{C}, \mathsf{DC}|_{(u',u)}\right)$ for

[6] This, assuming that the compliance predicate $\mathbf{C}$ has complexity that is at least linear. If $\mathbf{C}$ has sublinear complexity, then the bound S would have to account for the sum of the lengths of all parties' messages and programs as well.

[7] Again, if $\mathbf{C}$ has sublinear complexity, the bound s must account for a party's input messages, local program, and output messages as well.

every parent u' of u. By induction on the graph $\mathsf{DC}|_{(u',u)}$, we prove that, for sufficiently large k,

$$T_p\left(k, \mathbf{C}, \mathsf{DC}|_{(u',u)}\right) \leq q(k+ds) \ ,$$

for some polynomial q such that

$$q(k+sd) \geq s + d \cdot p\Big(k + s + \log\left(q(k+ds)\right)\Big) \ .$$

So now consider any given parent u' of u. If u' is a source vertex, then $T_p\left(k, \mathbf{C}, \mathsf{DC}|_{(u',u)}\right) = p(k+|\langle\mathbf{C}\rangle|) \leq p(k+s) \leq q(k+ds)$. If u' is not a source vertex, then let $w_1, \ldots, w_d$ be its parents. Then,

$$\begin{aligned}
T_p\left(k, \mathbf{C}, \mathsf{DC}|_{(u',u)}\right) &= \mathsf{time}_{\mathbf{C}}\Big(\mathsf{data}(\mathsf{in}(u')), \mathsf{code}(u'), \mathsf{data}(\mathsf{out}(u'))\Big) \\
&\qquad + \sum_{i=1}^{d} p\Big(k + |\langle\mathbf{C}\rangle| + |\mathsf{data}(w_i, u')| + \log T_p\big(k, \mathbf{C}, \mathsf{DC}|_{(w_i,u')}\big)\Big) \\
&\leq s + \sum_{i=1}^{d} p\Big(k + s + \log T_p\big(k, \mathbf{C}, \mathsf{DC}|_{(w_i,u')}\big)\Big) \\
&\leq s + \sum_{i=1}^{d} p\Big(k + s + \log\left(q(k+ds)\right)\Big) \qquad \text{(by the inductive assumption)} \\
&= s + d \cdot p\Big(k + s + \log\left(q(k+ds)\right)\Big) \\
&\leq q(k+sd) \ .
\end{aligned}$$

Therefore, we deduce that

$$\begin{aligned}
T_p\left(k, \mathbf{C}, \mathsf{DC}|_{(u,v)}\right) &\leq \mathsf{time}_{\mathbf{C}}\Big(\mathsf{data}(\mathsf{in}(u)), \mathsf{code}(u), \mathsf{data}(\mathsf{out}(u))\Big) \\
&\qquad + \sum_{u' \in \mathsf{parents}(u)} p\Big(k + |\langle\mathbf{C}\rangle| + |\mathsf{data}(u', u)| + \log T_p\big(k, \mathbf{C}, \mathsf{DC}|_{(u',u)}\big)\Big) \\
&\leq \mathsf{time}_{\mathbf{C}}\Big(\mathsf{data}(\mathsf{in}(u)), \mathsf{code}(u), \mathsf{data}(\mathsf{out}(u))\Big) + d \cdot p\Big(k + s + \log\left(q(k+ds)\right)\Big) \\
&\leq \mathsf{time}_{\mathbf{C}}\Big(\mathsf{data}(\mathsf{in}(u)), \mathsf{code}(u), \mathsf{data}(\mathsf{out}(u))\Big) + q(k+ds) \ .
\end{aligned}$$

Thus, the PCD prover's complexity at vertex u is at most

$$\mathsf{poly}\left(k + |z| + ds + \mathsf{time}_{\mathbf{C}}\Big(\mathsf{data}(\mathsf{in}(u)), \mathsf{code}(u), \mathsf{data}(\mathsf{out}(u))\Big)\right) \ .$$

Moreover, if we also know that the bound s applies to vertex u as well, then we are guaranteed to spend time at most

$$\mathsf{poly}\left(k + |z| + \log(ds) + \log \mathsf{time}_{\mathbf{C}}\Big(\mathsf{data}(\mathsf{in}(u)), \mathsf{code}(u), \mathsf{data}(\mathsf{out}(u))\Big)\right) \ .$$

in order to verify the proof string for message z along edge (u, v).

Practicality. We recall the bounds on the PCD prover's computational complexity are only asymptotic. As the construction of the PCD prover involves the use of a PCP system, it is an open problem to investigate the extent to which the PCD prover can be made practical.

5.3 Construction of a PCD system

We show how to use APHA systems to construct PCD systems, thus obtaining the following result:

Theorem 5.3.1 (PCD from APHA)**.** *PCD systems can be built from APHA systems (using the same oracle).*

Combining the above with Corollary 1, our result that gives sufficient conditions for the existence of APHA systems, we deduce the existence of PCD systems under mild standard assumptions:

Corollary 2 (Existence of PCD systems)**.** *Assuming the existence of collision-resistant hashing schemes, there exist PCD systems whose oracle is signed-input-and-randomness.*

We now show how to construct a PCD system $(G_{\mathsf{PCD}}, P_{\mathsf{PCD}}, V_{\mathsf{PCD}})$ using any given APHA system $(G_{\mathsf{APHA}}, P_{\mathsf{APHA}}, V_{\mathsf{APHA}})$, as defined in Section 4.2.

First, we define the PCD oracle generator to be the APHA oracle generator, i.e., $G_{\mathsf{PCD}} \equiv G_{\mathsf{APHA}}$.

Next, we discuss the construction of the PCD prover and PCD verifier. Intuitively, the PCD prover (resp., verifier) will invoke the APHA prover (resp., verifier) on specially-crafted instances "$(M_{\mathsf{PCD}}, x, t) \in S_{\mathcal{U}}$",[8] where M_{PCD} is a fixed machine (depending only on the compliance predicate $\mathbf{C}$ and the verification key vk) called the *PCD machine*; this machines specifies how to aggregate proof strings and locally check $\mathbf{C}$. We discuss the PCD machine next.

At high level, $M_{\mathsf{PCD}}(\cdot,\cdot)$ gets as input a string $x = (z_{\mathsf{out}}, d_{\mathsf{out}})$, where z_{out} is the alleged output of the current vertex and d_{out} is the depth of proof aggregation, and a witness $w = (\mathsf{prg}, z_{\mathsf{in}_1}, \pi_{\mathsf{in}_1}, z_{\mathsf{in}_2}, \pi_{\mathsf{in}_2})$, containing the program prg of the current vertex together with its inputs and their corresponding proof strings. The PCD machine will accept only if

- it verifies, by invoking V_{APHA}, that the proof strings of the inputs are valid, and
- $\mathbf{C}$-compliance holds, i.e., $\mathbf{C}\Big((z_{\mathsf{in}_1}, z_{\mathsf{in}_2}), \mathsf{prg}, (z_{\mathsf{out}})\Big) = 1$.

For the "base case" $d_{\mathsf{out}} = 1$, M_{PCD} does not have previous proof strings to verify, so it will only have to check that $\mathbf{C}$-compliance holds. Formally, the PCD machine is defined as follows:

Construction 5.3.2 (M_{PCD})**.** Fix $k \in \mathbb{N}$ and let $(O, \mathsf{vk}) \in G_{\mathsf{PCD}}(1^k)$. Let $\mathbf{C}$ be a compliance predicate, z_{out} the (alleged) output of a program prg with inputs z_{in_1} and z_{in_2}, and π_{in_1} and π_{in_2} proof strings. Define $x \equiv (z_{\mathsf{out}}, d_{\mathsf{out}})$ and $w \equiv (\mathsf{prg}, z_{\mathsf{in}_1}, \pi_{\mathsf{in}_1}, z_{\mathsf{in}_2}, \pi_{\mathsf{in}_2})$.

The *PCD machine* with respect to $\mathbf{C}$ and vk, denoted $M_{\mathsf{PCD}}^{\mathsf{vk},\mathbf{C}}$, on input (x, w), does the following:

1. *Base case.* If $\pi_{\mathsf{in}_1} = \perp$, verify that $d_{\mathsf{out}} = 1$ and $\mathbf{C}(\perp, \perp, z_{\mathsf{in}_1}) = 1$, otherwise reject.
2. *Recursive case.* If $\pi_{\mathsf{in}_1} \neq \perp$, parse π_{in_1} as $(\pi'_{\mathsf{in}_1}, d_{\mathsf{in}_1}, t_{\mathsf{in}_1})$, and do the following:
   (a) Verify that $d_{\mathsf{out}} > d_{\mathsf{in}_1} > 0$.
   (b) Define $y_{\mathsf{in}_1} \equiv \Big(M_{\mathsf{PCD}}^{\mathsf{vk},\mathbf{C}}, (z_{\mathsf{in}_1}, d_{\mathsf{in}_1}), t_{\mathsf{in}_1}\Big)$.
   (c) Verify that $V_{\mathsf{APHA}}(\mathsf{vk}, y_{\mathsf{in}_1}, \pi'_{\mathsf{in}_1}) = 1$, otherwise reject.
3. Repeat Step 1 and Step 2 for z_{in_2} and π_{in_2}.
4. Accept if and only if $\mathbf{C}\Big((z_{\mathsf{in}_1}, z_{\mathsf{in}_2}), \mathsf{prg}, (z_{\mathsf{out}})\Big)$ accepts.

Having defined the PCD machine, we may now proceed to describe the PCD prover and PCD verifier. We do so first at high level. The PCD prover P_{PCD}, with oracle access to O and on input $(\mathsf{vk}, \mathbf{C}, z_{\mathsf{out}}, \mathsf{prg}, z_{\mathsf{in}_1}, \pi_{\mathsf{in}_1}, z_{\mathsf{in}_2}, \pi_{\mathsf{in}_2})$, does the following:

1. Construct an instance $y \equiv \Big(M_{\mathsf{PCD}}, (z_{\mathsf{out}}, d_{\mathsf{out}}), t_{\mathsf{out}}\Big)$ for some appropriate numbers d_{out} and t_{out}.
2. Construct a witness $w \equiv (\mathsf{prg}, z_{\mathsf{in}_1}, \pi_{\mathsf{in}_1}, z_{\mathsf{in}_2}, \pi_{\mathsf{in}_2})$.
3. Use the APHA prover $P_{\mathsf{APHA}}^{O}(\mathsf{vk}, y, w)$ to generate a proof string π'.
4. Output the proof string $\pi \equiv (\pi', d_{\mathsf{out}}, t_{\mathsf{out}})$.

Then, the PCD verifier V_{PCD}, on input (vk, z, π), does the following:

1. Parse π as the triple (π', d, t).
2. Construct an instance $y \equiv (M_{\mathsf{PCD}}, (z, d), t)$.
3. Run the APHA verifier $V_{\mathsf{APHA}}(\mathsf{vk}, y, \pi')$ and accept if it accepts.

Formally, the PCD prover and verifier are constructed as follows.

[8]Recall that $S_{\mathcal{U}}$ is the universal set, introduced by Barak and Goldreich [15]. See Section 3.3.4 under the paragraph entitled "Universal arguments" for details.

Construction 5.3.3 (P_{PCD}). Fix $k \in \mathbb{N}$ and let $(O, \mathsf{vk}) \in G_{\mathsf{PCD}}(1^k)$. Let $\mathbf{C}$ be a compliance predicate, z_{out} the (alleged) output of a program prg with inputs z_{in_1} and z_{in_2} (and corresponding proof strings π_{in_1} and π_{in_2}).

The *PCD prover* $P^O_{\mathsf{PCD}}(\mathsf{vk}, \mathbf{C}, z_{\mathsf{out}}, \mathsf{prg}, z_{\mathsf{in}_1}, \pi_{\mathsf{in}_1}, z_{\mathsf{in}_2}, \pi_{\mathsf{in}_2})$ does the following:

1. If $\pi_{\mathsf{in}_1} = \perp$, run $\mathbf{C}(\perp, \perp, z_{\mathsf{in}_1})$ and let t_{in_1} be the time $\mathbf{C}$ takes to halt; else, parse π_{in_1} as $(\pi'_{\mathsf{in}_1}, d_{\mathsf{in}_1}, t_{\mathsf{in}_1})$.
2. If $\pi_{\mathsf{in}_2} = \perp$, run $\mathbf{C}(\perp, \perp, z_{\mathsf{in}_2})$ and let t_{in_2} be the time $\mathbf{C}$ takes to halt; else, parse π_{in_2} as $(\pi'_{\mathsf{in}_2}, d_{\mathsf{in}_2}, t_{\mathsf{in}_2})$.
3. Run $\mathbf{C}((z_{\mathsf{in}_1}, z_{\mathsf{in}_2}), u, (z_{\mathsf{out}}))$ and let $t_{\mathbf{C}}$ be the time $\mathbf{C}$ takes to halt.
4. Define $t_{\mathsf{out}} \equiv t_{\mathbf{C}} + t_{\mathsf{in}_1} + t_{\mathsf{in}_2}$ and $d_{\mathsf{out}} \equiv \max\{d_{\mathsf{in}_1}, d_{\mathsf{in}_2}\} + 1$.
5. Define $y \equiv \left(M^{\mathsf{vk},\mathbf{C}}_{\mathsf{PCD}}, (z_{\mathsf{out}}, d_{\mathsf{out}}), t\right)$ and $w \equiv (\mathsf{prg}, z_{\mathsf{in}_1}, \pi_{\mathsf{in}_1}, z_{\mathsf{in}_2}, \pi_{\mathsf{in}_2})$.
6. Compute $\pi' \leftarrow P^O_{\mathsf{APHA}}(\mathsf{vk}, y, w)$.
7. Define $\pi \equiv (\pi', d_{\mathsf{out}}, t_{\mathsf{out}})$.
8. Output π.

Construction 5.3.4 (V_{PCD}). Fix $k \in \mathbb{N}$ and let $(O, \mathsf{vk}) \in G_{\mathsf{PCD}}(1^k)$. Let $\mathbf{C}$ be a compliance predicate, z an output string, and π a proof string.

The *PCD verifier* $V_{\mathsf{PCD}}(\mathsf{vk}, \mathbf{C}, z, \pi)$ does the following:

1. If $\pi = \perp$, output $\mathbf{C}(\perp, \perp, z)$.
2. If $\pi = (\pi', d, t)$, define $y \equiv \left(M^{\mathsf{vk},\mathbf{C}}_{\mathsf{PCD}}, (z, d), t\right)$, and output $V_{\mathsf{APHA}}(\mathsf{vk}, y, \pi')$.

5.4 Correctness of the PCD construction

To complete the proof of Theorem 5.3.1, we must show that the construction given above is indeed a PCD system, according to Definition 5.2.2.

The two properties of efficient verifiability and completeness via a relatively-efficient prover easily follow from the construction. In the following, we sketch the proof of the PCD proof-of-knowledge property, while the full proof is given in Appendix B.

Proof sketch of the PCD proof-of-knowledge property. Let $\widetilde{P}$ be a (possibly cheating) efficient prover. We need to exhibit a knowledge extractor with the requisite properties. Hence, consider the PCD knowledge extractor E_{PCD} that, with oracle access to O and on input $(\mathsf{vk}, \mathbf{C}, z)$, does the following:

1. Run the prover $\widetilde{P}^O(\mathsf{vk}, \mathbf{C}, z)$ to record its oracle queries and answers $[\![\widetilde{P}(\mathsf{vk}, \mathbf{C}, z), O]\!]$ and to get its output (z, π).
2. Apply the APHA list extractor LE to the recorded interaction $[\![\widetilde{P}(\mathsf{vk}, \mathbf{C}, z), O]\!]$ to extract a list, $\mathsf{extlist}$, of triples (y_i, π_i, w_i); each triple contains an instance y_i, a proof string π_i, and a witness w_i.
3. Apply an *offline reconstruction procedure* which builds a transcript of the "past" distributed computation by examining only $\mathsf{extlist}$ and (z, π). (See below.)

All our work thus far was aimed at making such an offline reconstruction possible. The fact that the transcript can be reconstructed from a *single* invocation of $\widetilde{P}$ is essential: had we used a recursive approach requiring multiple invocations, we would have experienced an exponential blowup as aggregated proofs are recursively extracted.

Specifically, the offline reconstruction procedure performs a depth-first traversal of the implicit history represented by $\mathsf{extlist}$, starting from the root implied by the prover's output (z, π). It maintains the following data structures:

- An *augmented distributed computation transcript* ADC, initially containing just the output edge.
- An *exploration stack* $\mathsf{expstack}$, containing the set of edges in the distributed computation that we have discovered but not yet explored.

At a high level, the procedure operates iteratively as follows. At every iteration, we pop the next edge e to explore from $\mathsf{expstack}$ (initialized with the final output edge). Then, we check ADC to see what is the

APHA instance and proof string pair (y_e, π_e) on the edge e, and look for a corresponding triple of the form (y_e, π_e, w) in the extracted list $\mathsf{extlist}$. (From the APHA list-extraction property, this succeeds, and, moreover, w is a valid witness with all but negligible probability.) If we have already seen the instance-witness pair (y_e, w) on some edge edge e', we grow the graph of ADC by making the (hitherto unknown) source vertex of e the same as the source of e'. Otherwise, we grow ADC by making the source of e a new vertex v. If w is a witness that uses the base case of the PCD machine, then v is a source vertex and we are done for this iteration. Otherwise v is a new internal vertex, and we add the edges leading to its (yet unknown) parents to $\mathsf{expstack}$. The labels on ADC are updated accordingly. Termination is ensured by the monotonicity of the counters tracking the depth of proof aggregation. $\square$

Chapter 6

Applications and Design Patterns

Proof-carrying data is a flexible and powerful framework that can be applied to security goals in many problem domains. Below are some examples of domains where we envision applicability. We stress that the following discussion is intended to give a glimpse of things to come; full realizations, and evaluation of concrete practicality, exceed the scope of the present thesis.

Distributed theorem proving. Proof-carrying data can be interpreted as a new result in the theory of proofs: *"distributed theorem proving" is feasible.* It was previously known, via probabilistically-checkable proofs [12] and CS proofs [107], that one can be convinced of a theorem much quicker than by inspecting the theorem's proof. However, consider a theorem whose proof is built on various (possibly nested) lemmas proved by different people. In order to quickly convince a verifier of the theorem's truth, in previous techniques, we would have to obtain and concatenate the full (long) proofs of all the lemmas, and only then use (for example) CS proofs to compress them. PCD implies that compressed proofs for the lemmas can be directly used to obtain a compressed proof of the reliant theorem, and moreover the latter's length is essentially independent of the length of the lemmas' proofs.

Multilevel security. As mentioned in Section 1.1, PCD may be used for information flow control. For example, consider enforcing multilevel security [2, Chap. 8.6] in a room full of data-processing machines. We want to publish outputs labeled "non-secret", but are concerned that they may have been tainted by "secret" information (e.g., due to bugs, via software side channel attacks [27], or perhaps via literal eavesdropping [100][9][133]).

Suppose every "non-secret" input entering the system is digitally signed as such, by some classifier, under a verification key $\mathsf{vk}_{\mathsf{ns}}$. Suppose moreover (for simplicity) that the scheduling of which-program-to-apply-on-what-data is fully specified in advance and that the prescribed programs are deterministic. Then we can define the compliance predicate $\mathbf{C}$ as verifying that, in the distributed computation transcript, the output of every vertex is either properly signed under $\mathsf{vk}_{\mathsf{ns}}$, or is the result of correctly executing some program prg on the vertex's inputs and this is indeed the prescribed program according to the schedule. Then, every $\mathbf{C}$-compliant distributed computation transcript consists of applying the scheduled programs to "non-secret" inputs. Thus, its final output is independent of secret inputs.

The PCD system augments every message in the system with a proof string that attests this $\mathbf{C}$-compliance. Eventually, a censor at the system perimeter inspects the final output by verifying its associated proof, and lets out only properly-verified messages (as in Figure 1-2). Because verification is concerned with properties of the output per se, security is unaffected by anomalies (faults and leakage) in the preceding computation.

Bug attacks and IT supply chain. Faults can be devastating to security [22]. However, hardware and software components are often produced in far-away lands from parts of uncertain origin. This information technology (IT) supply chain issue forms risks to users and organizations [1][23][94][128]. Using PCD, one can achieve fault isolation and accountability at the level of system components, e.g., chips or software modules, by having each component augment every output with a proof that its computation, *including all history it relied on*, were correct. This requires the components to have a trusted functional specification (to be encoded into the compliance predicate), but their actual realization need not be trusted.

Simulations and MMO. Consider a simulation such as massively multiplayer online (MMO) worlds. These typically entail certain invariants ("laws of physics"), together with inputs chosen at human users' discretion. A common security goal is to ensure that a particular player does not cheat (e.g., by modifying the game code). Today, this is typically enforced by a centralized server, which is unscalable. Attempts at secure peer-to-peer architectures have seen very limited success [124][58]. PCD offers a potential solution approach when the underlying information flow has sufficient locality (as is it the case for most simulations): start with a naïve (insecure) peer-to-peer system, and enforce the invariants by augmenting every message with a proof of the requisite properties.

Financial systems. As a special case of the above, one can think of financial systems as a "game" where parties perform local transactions subject to certain rules. For example, in any transaction, the total amount of currency held by the parties must not increase (unless the government is involved). We conjecture that interesting financial settings can be thus captured and allowed to proceed in a secure distributed fashion. Potentially, this may capture financial processes that are much richer than the consumer-vendor relations of traditional e-cash.

Distributed dynamic program analysis. Consider, for example, taint propagation — a popular dynamic program analysis technique which tracks propagation of information inside programs. Current systems (e.g., [117]) cannot securely span mutually untrusting platforms. Since tainting rules are easily expressed by a compliance predicate that observes the computation of a program, PCD can maintain tainting across a distributed computation.

Distributed type safety. Language-based type-safety mechanisms have tremendous expressive power [122][123], but are targeted at the case where the underlying execution platform can be trusted to enforce type rules. Thus, they typically cannot be applied across distributed systems consisting of multiple mutually-untrusting execution platforms. This barrier can be surmounted by using PCD to augment typed values passing between systems with proofs for the correctness of the type.

Generalizing: design patterns. The PCD approach allows a system designer to "program in" the security requirement into a compliance predicate, and have it "magically" enforced by the PCD system. As gleaned from the above examples, this programming can be nontrivial and requires various tricks. This is somewhat similar to the world of software engineering, and indeed we can borrow some meta-techniques from that world. In particular, *design patterns* [57] are a very useful method for capturing common problems and solution techniques in a loosely-structured way. A number of such design patterns are already evident in the above examples (e.g., using signatures to designate parties or properties). We envision, and are exploring, a library of such patterns to aid system designers.

Chapter 7

Conclusions and Open Problems

Conclusions. We showed how to construct *assisted-prover hearsay arguments*: concise non-interactive arguments that allow for "aggregability" of proof strings. Our construction assumes only the existence of collision-resistant hashing schemes, and is in a model where parties have black-box access to a simple stateless trusted functionality.

Using these, we showed how to construct a *proof-carrying data system*, i.e., a protocol compiler that can enforce any efficiently-computable local invariant, called the *compliance predicate*, on distributed computations performed by mutually-untrusting parties.

Such a protocol compiler enables a new solution approach to security design that we call *proof-carrying data*. This solution approach provides a framework for achieving certain security properties in a non-conventional way, which circumvents many difficulties with current approaches. In proof-carrying data, faults and leakage are acknowledged as an expected occurrence, and rendered inconsequential by reasoning about properties of *data* that are independent of the preceding *computation*. The system designer prescribes the desired properties of the computation's output by choosing a compliance predicate; proofs of these properties are attached to the data flowing through the system, and are mutually verified by the system's components. We believe that in this way many security engineering problems are reduced to *compliance engineering*, the task of choosing a "good" compliance predicate for the given problem; this task is significantly simpler because it enables the system designer to only reason about properties of data, and not worry about which components of a possibly large and complex system may be untrusted, faulty, or leaky.

Open problems. The work of this thesis leaves open several future research directions:

- *Assumptions.* We showed how to construct a proof-carrying data system in a model where parties have black-box access to some functionality (e.g., a simple hardware token). The problem of weakening this requirement, or formally proving that it is (in some sense) necessary, remains open.

- *Practicality.* Of particular interest is surmounting the current inefficiency of the underlying argument systems and obtaining a fully practical realization.

- *Applications.* In this work we briefly touched upon potential applications; this leaves many opportunities for fleshing out the details, devising design patterns, and implementing real systems.

- *Extensions.* A proof-carrying data system with the additional property of zero-knowledge would be useful in many applications.

Appendix A

Full Proof of Security for APHA Systems

We now give the details for the proof of Claim 4.4.2; a sketch of the proof was already given in Section 4.4. Recall that we need to prove the list-extraction property of APHA systems, which will complete the proof of Theorem 4.3.1 (i.e., that APHA systems whose oracle is signed-input-and-randomness can be built from any signature scheme and public-coin constant-round universal arguments).

In order to show that the list-extractor LE given in Section 4.4 works, we define a sequence of intermediate constructions of increasing power, starting from universal-argument systems, and ending with APHA systems. Throughout, we will make use of the notion of *black-box rewinding access*, which is defined in Section 2.2.2 and denoted $A^{\uparrow B}$.

As a preliminary, we prove a (standard) probability lemma that we will use throughout our proofs.

A.1 A probability lemma

Roughly, we prove that any predicate over a product probability distribution can be usefully decomposed into two predicates over the two probability distributions of the product probability distribution.

Lemma A.1.1. *Let A and B be sets. If X is a predicate over $A \times B$ such that*

$$\Pr_{(\rho_1,\rho_2)\in A\times B}[X(\rho_1,\rho_2)] > 2\varepsilon \ ,$$

then

$$\Pr_{\rho_1\in A}\left[\Pr_{\rho_2\in B}[X(\rho_1,\rho_2)] > \varepsilon\right] > \varepsilon \ . \tag{A.1}$$

Proof. Let S be the set of "good" ρ_1, i.e.,

$$S \equiv \left\{\rho_1 \in A \colon \Pr_{\rho_2\in B}[X(\rho_1,\rho_2)] > \varepsilon\right\} \ .$$

Suppose (A.1) is violated. Then the premise is contradicted:

$$\begin{aligned}\Pr_{(\rho_1,\rho_2)\in A\times B}[X(\rho_1,\rho_2)] &= \Pr_{\rho_1\in A}[\rho_1\in S]\cdot \Pr_{\rho_2\in B}[X(\rho_1,\rho_2)] + \Pr_{\rho_1\in A}[\rho_1\notin S]\cdot \Pr_{\rho_2\in B}[X(\rho_1,\rho_2)] \\ &\le \varepsilon\cdot 1 + 1\cdot\varepsilon = 2\varepsilon \ .\end{aligned}$$

□

The above lemma will be useful to us in the case of $\varepsilon \equiv 1/\mathsf{poly}(k)$, because it tells us that if we randomly condition a predicate with non-negligible support size 2ε, then, with non-negligible probability ε, we obtain a conditional predicate whose support size is ε, which is still non-negligible.

A.2 Universal arguments with adaptive provers

First, starting from a universal argument system $(P_{\mathsf{UA}}, V_{\mathsf{UA}})$, which satisfies a weak proof-of-knowledge (PoK) property,[1] we show how to construct a pair of machines (P_0, V_0) for which the weak PoK holds even when the prover itself adaptively chooses a claimed instance y. The prover has oracle access to a functionality O' (informally called a "query box", and denoted O'_ρ when specifying its randomness) that outputs random strings upon request; the prover interacts with O', and then outputs an instance y and a proof string π_0 for the claim "$y \in S_{\mathcal{U}}$". When verifying the output of the prover, we allow V_0 to see all the query-answer pairs contained in the transcript trans' between the prover and O'. The weak PoK property in this setting is as follows:

Claim A.2.1. *Let $k \in \mathbb{N}$. For every (possibly cheating) prover circuit $\widetilde{P}_0$ of size $\mathsf{poly}(k)$ and every positive polynomial p, there exists a positive polynomial p' and a probabilistic polynomial-time knowledge extractor E_0 (which can be efficiently found) such that the following holds: if $\widetilde{P}_0$ convinces V_0 with non-negligible probability,*

$$\Pr_{\rho \in \{0,1\}^{\mathsf{poly}(k)}} \left[V_0(\mathsf{trans}', y, \pi_0) = 1 \;\middle|\; (y, \pi_0) \leftarrow \widetilde{P}_0^{O'_\rho} \; ; \; \mathsf{trans}' \leftarrow [\![\widetilde{P}_0, O'_\rho]\!] \right] > \frac{1}{p(k)} \enspace ,$$

then E_0, with input the randomness ρ used by O' and with rewinding oracle access to $\widetilde{P}_0$, extracts a valid witness w for y with non-negligible probability,

$$\Pr_{\rho \in \{0,1\}^{\mathsf{poly}(k)}} \left[(y, w) \in R_{\mathcal{U}} \;\middle|\; (y, \pi_0) \leftarrow \widetilde{P}_0^{O'_\rho} \; ; \; w \leftarrow E_0^{\uparrow \widetilde{P}_0}(1^t, \rho) \right] > \frac{1}{p'(k)} \enspace .$$

(The input 1^t to E_0 is a technical requirement to ensure that E_0 is able to output a full witness for y.)

We now construct (P_0, V_0) from $(P_{\mathsf{UA}}, V_{\mathsf{UA}})$. The idea of the construction is to make V_0 accept any (y, π_0) for which π_0 is an accepting universal-argument transcript $(r_1, \omega_1, r_2, \omega_2)$ for y, as long as V_0 confirms, by looking at the query-answers pairs of a prover to O', that indeed r_1 and r_2 are random and that indeed ω_1 had been "committed to" before r_2 was generated. Then, P_0 is constructed from P_{UA} in the obvious way. More precisely, the machines of P_0 and V_0 are as follows:

Construction A.2.2 (P_0)**.** The prover $P_0^{O'}(y, w)$ is defined as follows:

1. Query O' with $q_1 = (y, p_1(|y|))$ to obtain random string r_1.
2. Run $P_{\mathsf{UA}}(y, w; r_1)$, the universal-argument prover with r_1 as V_{UA}'s first message, and let ω_1 be its output (which is the first message that would have been sent to V_{UA}).
3. Query O' with $q_2 = ((y, \omega_1), p_2(|y|))$ to obtain random string r_2.
4. Run $P_{\mathsf{UA}}(y, w; r_1, r_2)$, the universal-argument prover with r_1 and r_2 as V_{UA}'s first and second messages, and let ω_2 be its output (which is the second message that would have been sent to V_{UA}).
5. Define $\pi_0 \equiv (r_1, \omega_1, r_2, \omega_2)$, and output π_0.

Construction A.2.3 (V_0)**.** The verifier $V_0(\mathsf{trans}', y, \pi_0)$ is defined as follows:

1. Parse π_0 as $(r_1, \omega_1, r_2, \omega_2)$.
2. Verify that $|r_1| = p_1(|y|)$ and $|r_2| = p_2(|y|)$.
3. Verify that trans' contains the query-answer pairs (y, r_1) and $((y, \omega_1), r_2)$.
4. Verify that $V_{\mathsf{UA}}(y; r_1, \omega_1, r_2, \omega_2) = 1$.

We now prove that the pair of machines (P_0, V_0) from Construction A.2.2 and Construction A.2.3 satisfies the weak proof-of-knowledge property of Claim A.2.1.

Proof of Claim A.2.1. Let $\widetilde{P}_0$ be a (possibly cheating) prover circuit of size $\mathsf{poly}(k)$ that, with oracle access to a query box O', convinces the verifier V_0 (to accept some instance and proof string of his choosing) with some non-negligible probability δ.

First, we exhibit a procedure $\mathsf{makeUAprover}$ that uses $\widetilde{P}_0$ to probabilistically generate an instance y and a universal-argument prover $\widetilde{P}_{\mathsf{UA}}$; moreover, we prove that with non-negligible probability (over the generation of $\widetilde{P}_{\mathsf{UA}}$), $\widetilde{P}_{\mathsf{UA}}$ convinces V_{UA} that $y \in S_{\mathcal{U}}$ with non-negligible probability.

Fix some topological order on the gates of $\widetilde{P}_0$, and note that $\widetilde{P}_0$ always makes at most $l \equiv |\widetilde{P}_0|$ queries. The procedure $\mathsf{makeUAprover}$, on input the randomness ρ of the query box, is defined as follows:

[1] See Section 3.3.4, under paragraph entitled "Universal arguments", for the definition of universal arguments.

1. Compute $(y, \pi_0) \leftarrow \widetilde{P}_0^{O'_\rho}$, by simulating the query box O'_ρ.

2. Draw query indices $i, j \leftarrow [l]$ such that $i < j$.

3. Parse ρ as answers to the l queries of $\widetilde{P}_0$. Define $\rho_{[i,j]}$ as ρ with the i-th and j-th answers omitted.

4. Run $\widetilde{P}_0$ with query box O'_ρ until $\widetilde{P}_0$'s i-th query q_i.

5. Verify that $q_i = (y, p_1(|y|))$; otherwise, abort.

6. Define $\widetilde{P}_{\mathsf{UA}} = \widetilde{P}_{\mathsf{UA}}(y, i, j, \rho_{[ij]})$ to be the following (universal-argument) prover:

$\widetilde{P}_{\mathsf{UA}} \equiv$
   1. Run $\widetilde{P}_0$ until its i-th query, using $\rho_{[ij]}$ to answer queries.
   2. Upon receiving V_{UA}'s first message r_1, give r_1 as answer to the i-th query of $\widetilde{P}_0$.
   3. Continue running $\widetilde{P}_0$ until its j-th query q_j, using $\rho_{[ij]}$ to answer queries.
   4. Verify that $q_j = ((y, \omega_1), p_2(|y|))$, for some ω_1, and send ω_1 to V_{UA}; otherwise, abort.
   5. Upon receiving V_{UA}'s second message r_2, give r_2 as answer to the j-th query of $\widetilde{P}_0$.
   6. Continue running $\widetilde{P}_0$ using $\rho_{[ij]}$ to answer queries, until it outputs (y', π_0).
   7. If $y' \neq y$ or π_0 is not of the form $(r_1, \omega_1, r_2, \omega_2)$, abort.
   8. Send ω_2 to V_{UA}.

7. Output $\big(y, \langle \widetilde{P}_{\mathsf{UA}} \rangle\big)$.

By construction, whenever V_0 accepts some instance y and proof $\pi_0 = (r_1, \omega_1, r_2, \omega_2)$ from some prover $\widetilde{P}_0$, it is the case that $\widetilde{P}_0$ queried O' with queries (y, r_1) and $((y, \omega_1), r_2)$, in this order. If on query answers ρ the prover $\widetilde{P}_0$, using query box O'_ρ, outputs (y, π_0) and its i-th and j-th queries are of the above form, we say that (i, j) are *good indices for* ρ (if there are multiple such good indices, always choose the least such indices). Intuitively, the good indices are those that contain the relevant public-coin challenges from the UA verifier, and the procedure $\mathsf{makeUAprover}$ tries to guess these good indices (i, j) for ρ.

Consider the event ξ (over i and j drawn by $\mathsf{makeUAprover}$ and over a random choice of its input ρ) that V_0 accepts the output of $\widetilde{P}_0$ (when using query box O'_ρ) and (i, j) are good indices for ρ. This event has non-negligible probability, at least δ/l^2.

Then, invoking Lemma A.1.1 with $\varepsilon = \delta/(2l^2)$, $X = \xi$, A consisting of possible values of $(i, j, \rho_{[ij]})$, and B consisting of possible values of (ρ_i, ρ_j), we get:

$$\Pr_{(i,j,\rho_{[ij]})}\left[\Pr_{(\rho_i,\rho_j)}\left[\xi(i, j, \rho_{[ij]}, \rho_i, \rho_j)\right] > \varepsilon\right] > \varepsilon$$

and thus

$$\Pr_{(i,j,\rho_{[ij]})}\left[\Pr_{(\rho_i,\rho_j)}\left[V_0 \text{ accepts } \widetilde{P}_0^{O'_\rho} \text{ and } (i,j) \text{ are good indices for } \rho_{[i,j]}, \rho_i, \rho_j\right] > \varepsilon\right] > \varepsilon \ .$$

By construction, the innermost event implies that when V_{UA} produces challenges $(r_1, r_2) = (\rho_i, \rho_j)$, it is convinced by $\widetilde{P}_{\mathsf{UA}}(y, i, j, \rho_{[ij]})$ that $y \in S_{\mathcal{U}}$. Hence,

$$\Pr_{(i,j,\rho_{[ij]})}\left[\Pr\left[V_{\mathsf{UA}} \text{ is convinced by } \widetilde{P}_{\mathsf{UA}} \text{ that } y \in S_{\mathcal{U}}\right] > \varepsilon \;\middle|\; \big(y, \langle \widetilde{P}_{\mathsf{UA}} \rangle\big) \leftarrow \mathsf{makeUAprover}(\rho)\right] > \varepsilon \ .$$

Next, we exhibit a strategy for the knowledge extractor E_0. Essentially, E_0 invokes the procedure $\mathsf{makeUAprover}$ as a subroutine, and then uses the universal-argument knowledge extractor E_{UA} to recover a witness for the given instance. The knowledge extractor $E_0^{\uparrow \widetilde{P}_0}(1^t, y, \rho)$ is defined as follows:

$E_0^{\uparrow \widetilde{P}_0}(1^t, \rho) \equiv$
   1. $\big(y, \langle \widetilde{P}_{\mathsf{UA}} \rangle\big) \leftarrow \mathsf{makeUAprover}^{\uparrow \widetilde{P}_0}(\rho)$.
   2. For $i \in [t]$, $w_i \leftarrow E_{\mathsf{UA}}^{\uparrow\left(\widetilde{P}_{\mathsf{UA}}^{\uparrow \widetilde{P}_0}\right)}(y, i)$.
   3. Output $w = w_1 \ldots w_t$.

We have shown above that, with non-negligible probability, $\mathsf{makeUAprover}$ generates a $\widetilde{P}_{\mathsf{UA}}$ that convinces V_{UA} that $y \in S_{\mathcal{U}}$ with non-negligible probability. We can now invoke the weak proof-of-knowledge property of universal arguments, to deduce that, with non-negligible probability, $E_{\mathsf{UA}}(y, \cdot)$ is an implicit representation of a valid witness for y. Thus, with non-negligible probability, we can recover a valid witness w for y by asking each bit w_i of the witness to E_{UA}.

□

We observe that the proof above still holds for any prover $\widetilde{P}_0$ that has oracle access to a uniformly-generated polynomial-time oracle. This fact is used implicitly in the proofs of the constructions that follow.

A.3 Achieving non-interaction through a SIR oracle

Next, we show how to replace the (stateful and probabilistic) query box O' with a (stateless and probabilistic) oracle O'', and obtain another argument system. In doing so, we avoid the need for the verifier to know the transcript between the prover and O' (as was the case for V_0); instead, we use signatures to force the prover to include a genuine transcript in its proof string. The resulting proof system is non-interactive in the assisted prover model, where the oracle is O'' (as opposed to a SIR oracle, which will appear in the next construction).

The new argument system is a triple (G_1, P_1, V_1), because this time we need an oracle generator G_1 that outputs O'' together with its corresponding verification key vk_1. The weak PoK property in this new setting takes the following form:

Claim A.3.1. *Let $k \in \mathbb{N}$. For every (possibly cheating) prover circuit $\widetilde{P}_1$ of size $\mathsf{poly}(k)$ and every positive polynomial p, there exists a positive polynomial p' and a probabilistic knowledge extractor E_1 (which can be efficiently found) such that the following holds: if $\widetilde{P}_1$ convinces V_1 with non-negligible probability,*

$$\Pr_{\rho \in \{0,1\}^{\mathsf{poly}(k)}}\left[V_1(\mathsf{vk}_1, y, \pi_1) = 1 \;\middle|\; (O''_{\rho,\mathsf{sk}_1}, \mathsf{vk}_1, S^{\mathsf{sk}_1}_{\mathsf{SIG}}) \leftarrow G_1(1^k, \rho)\;;\; (y, \pi_1) \leftarrow \widetilde{P}_1^{O''_{\rho,\mathsf{sk}_1}}(\mathsf{vk}_1)\right] > \frac{1}{p(k)} \;,$$

then E_1, on input the verification key vk_1 and the randomness ρ used by O'', with rewinding oracle access to $\widetilde{P}_1$, and with oracle access to the signing oracle S_{SIG}, extracts a valid witness w for y with non-negligible probability,

$$\Pr_{\rho \in \{0,1\}^{\mathsf{poly}(k)}}\Big[(y, w) \in R_{\mathcal{U}} \;\Big|\; (O''_{\rho,\mathsf{sk}_1}, \mathsf{vk}_1, S^{\mathsf{sk}_1}_{\mathsf{SIG}}) \leftarrow G_1(1^k, \rho)\;;$$
$$(y, \pi_1) \leftarrow \widetilde{P}_1^{O''_{\rho,\mathsf{sk}_1}}(\mathsf{vk}_1)\;;\; w \leftarrow E_1^{\uparrow\widetilde{P}_1,\, S^{\mathsf{sk}_1}_{\mathsf{SIG}}}(1^t, \mathsf{vk}_1, \rho)\Big] > \frac{1}{p'(k)} \;.$$

(The input 1^t to E_1 is a technical requirement to ensure that E_1 is able to output a full witness for y.)

Starting from (P_0, V_0), and using the signature scheme SIG defined in Section 2.3.5, we construct (G_1, P_1, V_1) in the following way:

Construction A.3.2 (G_1)**.** The oracle generator $G_1(1^k, \rho)$ is defined as follows:

1. $(\mathsf{vk}_1, \mathsf{sk}_1) \leftarrow G_{\mathsf{SIG}}(1^k)$.
2. Let O''_{ρ,sk_1} be the oracle that, on input (x, s), does the following:
   (a) $r \leftarrow \{0,1\}^s$, using the next s bits of the random string ρ.
   (b) $\sigma \leftarrow S_{\mathsf{SIG}}(\mathsf{sk}_1, (x, r))$.
   (c) Output (r, σ).
3. Define $S^{\mathsf{sk}_1}_{\mathsf{SIG}} \equiv S_{\mathsf{SIG}}(\mathsf{sk}_1, \cdot)$.
4. Output $\left(O''_{\rho,\mathsf{sk}_1}, \mathsf{vk}_1, S^{\mathsf{sk}_1}_{\mathsf{SIG}}\right)$.

Construction A.3.3 (P_1)**.** The prover $P_1^{O''}(y, w)$ is defined as follows:

1. Run $P_0^{O'}(y, w)$ by simulating its query box O' as follows: for each query $q = (x, s)$, query the oracle O'' with q to obtain $a = (r, \sigma)$, and return r as answer.
2. Let π_0 be the output of $P_0(y, w)$ and let $\mathsf{trans}'' = \{(q_i, a_i)\}_i$ be the transcript of its interaction with the oracle O''.
3. Output $\pi_1 = (\mathsf{trans}'', \pi_0)$.

Construction A.3.4 (V_1)**.** The verifier $V_1(\mathsf{vk}_1, y, \pi_1)$ is defined as follows:

1. Parse π_1 as $(\mathsf{trans}'', \pi_0)$.
2. Verify that $\mathsf{trans}'' = \{(q_i, a_i)\}_i$ is of the correct form and has valid signatures, i.e., that $q_i = (x_i, d_i)$, $a_i = (r_i, \sigma_i)$, $d_i = |r_i|$, and $V_{\mathsf{SIG}}(\mathsf{vk}_1, ((x_i, d_i), r_i), \sigma_i) = 1$, for some strings x_i and r_i.
3. Define the query-answer set trans' to be the set $\{(x_i, r_i)\}_i$.
4. Verify that $V_0(\mathsf{trans}', y, \pi_0) = 1$.

We now prove that the construction for (G_1, P_1, V_1) given above satisfies the required weak PoK property.

Proof of Claim A.3.1. Let $\widetilde{P}_1$ be a (possibly cheating) prover circuit that, with oracle access to O'', convinces the verifier V_1 with non-negligible probability.

By construction, whenever V_1 is convinced by $\widetilde{P}_1$, it holds that the transcript trans'' contained in the proof π_1 output by $\widetilde{P}_1$ is of the correct form with valid signatures (as specified in Step 2 of V_1). By the existential unforgeability property of the signature scheme $\mathsf{SIG} = (G_{\mathsf{SIG}}, S_{\mathsf{SIG}}, V_{\mathsf{SIG}})$, except with negligible probability, every query-answer pair (q_i, a_i) contained in trans'' is genuine, i.e., $\widetilde{P}_1$ did query O'' with query q_i and received a corresponding answer a_i;[2] so condition on this event happening.

First, we exhibit a procedure $\mathsf{make0prover}$ that uses $\widetilde{P}_1$ to generate a prover $\widetilde{P}_0$ for V_0. The procedure $\mathsf{make0prover}$, on input vk_1, outputs the prover $\widetilde{P}_0 = \widetilde{P}_0(\mathsf{vk}_1)$ defined as follows:

1. Run $\widetilde{P}_1^{O''}(\mathsf{vk}_1)$ by simulating O'' as follows: for each query $q = (x, s)$, query O' with q to obtain randomness r, then query $S_{\mathsf{SIG}}^{\mathsf{sk}_1}$ with query (x, r) to obtain signature σ, and return (r, σ).
2. Let the output of $\widetilde{P}_1^{O''}(\mathsf{vk}_1)$ be (y, π_1) and parse π_1 as $(\mathsf{trans}'', \pi_0)$.
3. Output (y, π_0).

(Note $\widetilde{P}_0$ will be given oracle access to O' when challenged to convince a verifier V_0; also, it will be given oracle access to $S_{\mathsf{SIG}}^{\mathsf{sk}_1}$ by the extractor E_1 that will use it.)

Now consider the event ξ (over the random string ρ and key pairs $(\mathsf{vk}_1, \mathsf{sk}_1) \leftarrow G_{\mathsf{SIG}}(1^k)$) that V_1 accepts the output of $P_1^{O''_{\rho,\mathsf{sk}_1}}(\mathsf{vk}_1)$. By assumption, this event has non-negligible probability δ. Then, invoking Lemma A.1.1 with $\varepsilon = \delta/2$, $X = \xi$, A consisting of possible values of $(\mathsf{vk}_1, \mathsf{sk}_1)$, and B consisting of possible values of ρ, we get:

$$\Pr_{(\mathsf{vk}_1,\mathsf{sk}_1)}\left[\Pr_{\rho}\left[\xi(\mathsf{sk}_1, \mathsf{vk}_1, \rho)\right] > \varepsilon\right] > \varepsilon$$

and thus

$$\Pr_{(\mathsf{vk}_1,\mathsf{sk}_1)}\left[\Pr_{\rho}\left[V_1 \text{ accepts } \widetilde{P}_1^{O''_{\rho,\mathsf{sk}_1}}(\mathsf{vk}_1)\right] > \varepsilon\right] > \varepsilon \ .$$

By construction, the innermost event is equal to the event that $\widetilde{P}_0$ convinces the verifier V_0 when having oracle access to a query box O'_ρ. Hence,

$$\Pr_{\langle\widetilde{P}_0\rangle \leftarrow \mathsf{make0prover}}\left[\Pr_{\rho}\left[V_0 \text{ accepts } \widetilde{P}_0^{O'_\rho}\right] > \varepsilon\right] > \varepsilon \ .$$

Next, we exhibit a strategy for the knowledge extractor E_1. Essentially, E_1 invokes the procedure $\mathsf{make0prover}$ as a subroutine, and then uses the knowledge extractor E_0 to recover a witness for the given instance. The knowledge extractor $E_1^{\uparrow\widetilde{P}_1, S_{\mathsf{SIG}}^{\mathsf{sk}_1}}(1^t, \mathsf{vk}_1, y, \rho)$ is defined as follows:

$$E_1^{\uparrow\widetilde{P}_1, S_{\mathsf{SIG}}^{\mathsf{sk}_1}}(1^t, \mathsf{vk}_1, \rho) \quad \equiv \quad \begin{array}{l} 1.\ \langle\widetilde{P}_0\rangle \leftarrow \mathsf{make0prover}^{\uparrow\widetilde{P}_1}(\mathsf{vk}_1). \\ 2.\ w \leftarrow E_0^{\uparrow\left(\widetilde{P}_0^{S_{\mathsf{SIG}}^{\mathsf{sk}_1}, \uparrow\widetilde{P}_1}\right)}(1^t, \rho). \\ 3.\ \text{Output } w. \end{array}$$

By now invoking the weak PoK property of (P_0, V_0), we deduce that E_0 outputs, with non-negligible probability, a valid witness w for the instance y in the output of $\widetilde{P}_1^{\prime O'_\rho}$ (which is the same instance as the one in the output of $\widetilde{P}_1^{O''_{\rho,\mathsf{sk}_1}}(\mathsf{vk}_1)$), and this witness is output by E_1, as desired. $\square$

[2] This argument can be made more precise by considering a forger for each query-answer pair of the transcript; each such forger attempts to forge by outputting his corresponding query-answer pair; then, unforgeability implies that the probability of success of each such forger is negligible.

A.4 Forcing a prover to query the witness

Finally, we show how we can "boost" the weak PoK property of (G_1, P_1, V_1) into the much stronger APHA list-extraction property, thereby obtaining an APHA system $(G_{\mathsf{APHA}}, P_{\mathsf{APHA}}, V_{\mathsf{APHA}})$. The main idea is to require any prover to provide a convincing proof string to V_1, with the modification that the proof string should not refer to the instance y, but instead to some *augmented instance* y_{aug}; this new instance y_{aug} is constructed in a way that "forces" the prover to explicitly include the instance and the witness in a query to a signing oracle in order to obtain a signature σ_{iw}. Intuitively, to see a witness, an extractor would not need to rewind, but would just need to look at the prover's oracle queries and find the one resulting in σ_{iw}.

Consider the following constructions for $(G_{\mathsf{APHA}}, P_{\mathsf{APHA}}, V_{\mathsf{APHA}})$, equivalent to those that we presented in Section 4.3.

Construction A.4.1 (G_{APHA}). The oracle generator $G_{\mathsf{APHA}}(1^k)$ is defined as follows:

1. $(O'', \mathsf{vk}_1, S_{\mathsf{SIG}}^{\mathsf{sk}_1}) \leftarrow G_1(1^k)$.
2. $(\mathsf{vk}_2, \mathsf{sk}_2) \leftarrow G_{\mathsf{SIG}}(1^k)$.
3. Define $S_{\mathsf{SIG}}^{\mathsf{sk}_2} \equiv S_{\mathsf{SIG}}(\mathsf{sk}_2, \cdot)$.
4. Define $\mathsf{vk} \equiv (\mathsf{vk}_1, \mathsf{vk}_2)$.
5. Define $O = O(\mathsf{sk}_2, O'')$ to be the following:
   $O(x, s) \equiv$ 1. If $s > 0$, then output $O''(x, s)$;
   2. If $s = 0$, then output $(\epsilon, S_{\mathsf{SIG}}^{\mathsf{sk}_2}(x, \epsilon))$.
6. Output $(O, \mathsf{vk}, S_{\mathsf{SIG}}^{\mathsf{sk}_2})$.

Construction A.4.2 (P_{APHA}). The prover $P_{\mathsf{APHA}}^{O}(\mathsf{vk}, y, w)$ is defined as follows:

1. Parse vk as $(\mathsf{vk}_1, \mathsf{vk}_2)$.
2. $\sigma_{\mathsf{iw}} \leftarrow O\big((\text{"}\texttt{inst-wit}\text{"}, y, w), 0\big)$.
3. $y_{\mathsf{aug}} \leftarrow \mathsf{AUG}(\mathsf{vk}_2, \sigma, y)$.
4. $\pi_1 \leftarrow P_1^O(y_{\mathsf{aug}}, w)$.
5. Define $\pi' \equiv (\sigma, \pi_1)$.
6. $\sigma' \leftarrow O((\text{"}\texttt{proof}\text{"}, \pi'), 0)$.
7. Output $\pi \equiv (\pi', \sigma')$.

Construction A.4.3 (V_{APHA}). The verifier $V_{\mathsf{APHA}}(\mathsf{vk}, y, \pi)$ is defined as follows:

1. Parse vk as $(\mathsf{vk}_1, \mathsf{vk}_2)$.
2. Parse π as (π', σ') and π' as $(\sigma_{\mathsf{iw}}, \pi_1)$.
3. Verify that $V_{\mathsf{SIG}}(\mathsf{vk}_2, ((\text{"}\texttt{proof}\text{"}, \pi'), \epsilon), \sigma') = 1$.
4. $y_{\mathsf{aug}} \leftarrow \mathsf{AUG}(\mathsf{vk}_2, \sigma_{\mathsf{iw}}, y)$.
5. Verify that $V_1(\mathsf{vk}_1, y_{\mathsf{aug}}, \pi_1) = 1$.

We now prove the APHA list-extraction property, thereby completing the proof of Theorem 4.3.1.

Proof of APHA list-extraction property. Fix a prover circuit $\widetilde{P}$. First, we present a strategy for the APHA list extractor LE. For $(O, \mathsf{vk}) \in G_{\mathsf{APHA}}(1^k)$, consider the following strategy:

LE, on input $[\![\widetilde{P}(\mathsf{vk}), O]\!]$, does the following:

1. Create an empty list $\mathsf{extlist}$.
2. Let $(q_1, a_1), \ldots, (q_l, a_l)$ be the oracle query-answer pairs in $[\![\widetilde{P}(\mathsf{vk}), O]\!]$ for which q_i is of the form $((\text{"}\texttt{proof}\text{"}, \pi_i'), 0)$.
3. For each $i \in [l]$, do the following:
   (a) Parse π_i' as $(\sigma_{\mathsf{iw}}^{(i)}, \pi_1^{(i)})$ and a_i as (ϵ, σ_i'), and find some query-answer pair (q, a) in $[\![\widetilde{P}(\mathsf{vk}), O]\!]$ such that $a = (\epsilon, \sigma_{\mathsf{iw}}^{(i)})$ and q is of the form $q = ((\text{"}\texttt{inst-wit}\text{"}, y, w), 0)$.
   (b) Add $(y, (\pi_i', \sigma_i'), w)$ to $\mathsf{extlist}$.
4. Output $\mathsf{extlist}$.

Note that, indeed, $|\mathsf{LE}| = \mathsf{poly}(|[\![\widetilde{P}(\mathsf{vk}), O]\!]|) = \mathsf{poly}(k)$.

Now we prove that the list extractor LE fulfills the list-extraction property. Note that, whenever V_{APHA} accepts some proof string $\pi = (\pi', \sigma')$ for some claim "$y \in S_{\mathcal{U}}$", it holds that $V_{\mathsf{SIG}}(\mathsf{vk}_2, ((\text{"}\texttt{proof}\text{"}, \pi'), \epsilon), \sigma') =$

1 (see Step 3 in V_{APHA}); hence, by the existential unforgeability property of SIG, with all but negligible probability, there is at least one query-answer pair (q, a) where q is of the form $((\text{“}\mathtt{proof}\text{”}, \pi'), 0)$ and a is of the form (ϵ, σ'). Let S be the set of those coin tosses (of the oracle O) for which this is the case and for which the (possibly cheating) prover circuit $\widetilde{P}$ convinces the verifier V_{APHA}.

Recalling that π' is parsed as $(\sigma_{\mathsf{iw}}, \pi_1)$ (see Step 2 in V_{APHA}), let $\mathcal{M}_{\sigma_{\mathsf{iw}}}$ denote the (possibly empty) subset of $[\![\widetilde{P}(\mathsf{vk}), O]\!]$ of those oracle query-answer pairs whose signature contained in the oracle answer is equal to σ_{iw}, i.e.,

$$\mathcal{M}_{\sigma_{\mathsf{iw}}} = \left\{ (q, a) \in [\![\widetilde{P}(\mathsf{vk}), O]\!] \colon a = (r, \sigma_{\mathsf{iw}}) \text{ for some string } r \right\} .$$

Given some query q to the oracle and some instance $\widetilde{y}$, we will say that q is *valid with respect to* $\widetilde{y}$ if q is of the form $((\text{“}\mathtt{inst\text{-}wit}\text{”}, \widetilde{y}, \widetilde{w}), 0)$ and $(\widetilde{y}, \widetilde{w}) \in S_{\mathcal{U}}$ for some string $\widetilde{w}$.

We now split S into different subsets. For oracle coin tosses in S, it is the case that π convinces V_{APHA} that $y \in S_{\mathcal{U}}$, so, in particular, it holds that $V_1(\mathsf{vk}_1, y_{\mathsf{aug}}, \pi_1) = 1$ (see Step 5 in V_{APHA}). We will show that most of the probability mass in S goes into a “good” subset, for which the list extractor LE described above succeeds. For each “bad” subset, we prove that it has negligible probability mass.

Case 1: $A \subset S$ is the set of those coin tosses for which $y_{\mathsf{aug}} = \mathsf{AUG}(\mathsf{vk}_2, \sigma_{\mathsf{iw}}, y) \notin S_{\mathcal{U}}$. This set has negligible probability mass by the adaptive computational soundness of (G_1, P_1, V_1), which is implied by its weak proof-of-knowledge property (see Claim A.3.1).

Case 2: $B \subset S$ is the set of those coin tosses for which $y_{\mathsf{aug}} = \mathsf{AUG}(\mathsf{vk}_2, \sigma_{\mathsf{iw}}, y) \in S_{\mathcal{U}}$ and in (possibly empty) $\mathcal{M}_{\sigma_{\mathsf{iw}}} = \{(q_1, a_1), \ldots, (q_d, a_d)\}$ all the q_i's are *not* valid with respect to y. Suppose by way of contradiction that the probability mass of B is non-negligible. In such a case, we show how to construct a prover $\widetilde{P}_1$ that convinces V_1 with non-negligible probability, so that, using E_1, we can extract (with non-negligible probability) a valid witness w for y_{aug}; the message-signature pair

$$\Big(((\text{“}\mathtt{inst\text{-}wit}\text{”}, y, w), \epsilon), \sigma_{\mathsf{iw}} \Big) \tag{A.2}$$

will then contradict the existential unforgeability property of SIG.

We exhibit a procedure $\mathsf{make1prover}$ that uses $\widetilde{P}$ to generate a prover $\widetilde{P}_1$ for V_1. The procedure $\mathsf{make1prover}^{\uparrow\widetilde{P}}$, on input vk_2, outputs the prover $\widetilde{P}_1 = \widetilde{P}_1(\mathsf{vk}_2)$ that, on input vk_1 and with oracles O'' and $S_{\mathsf{SIG}}^{\mathsf{sk}_2}$, does the following:

1. Define $\mathsf{vk} = (\mathsf{vk}_1, \mathsf{vk}_2)$.
2. Run $\widetilde{P}^O(\mathsf{vk})$ by simulating its oracle O as follows: for each query $q = (x, s)$, if $s = 0$ then return the answer $(\epsilon, S_{\mathsf{SIG}}^{\mathsf{sk}_2}(x, \epsilon))$; if $s > 0$, then give the circuit the answer $O''(q)$.
3. Let (y, π) be the output of $\widetilde{P}^O(\mathsf{vk})$.
4. Parse $\pi = (\pi', \sigma')$ and π' as $(\sigma_{\mathsf{iw}}, \pi_1)$.
5. $y_{\mathsf{aug}} \leftarrow \mathsf{AUG}(\mathsf{vk}_2, \sigma_{\mathsf{iw}}, y)$.
6. Output $(y_{\mathsf{aug}}, \pi_1)$.

(Note $\widetilde{P}_1$ will be given oracle access to O'' when challenged to convince a verifier V_1; also, it will be given oracle access to $S_{\mathsf{SIG}}^{\mathsf{sk}_2}$ by the extractor $\mathsf{WE}_{\mathsf{APHA}}$ that will use it.)

By the same kind of argument made in the proof of Claim A.3.1, we deduce that $\widetilde{P}_1$, with non-negligible probability, convinces the verifier V_1 with non-negligible probability. Hence, similarly, we can construct a weak proof-of-knowledge extractor $\mathsf{WE}_{\mathsf{APHA}}^{\uparrow\widetilde{P}, S_{\mathsf{SIG}}^{\mathsf{sk}_1}, S_{\mathsf{SIG}}^{\mathsf{sk}_2}}$ that, on input $(1^t, \mathsf{vk}, \pi, \rho)$ is defined as follows:

$\mathsf{WE}_{\mathsf{APHA}}^{\uparrow\widetilde{P}, S_{\mathsf{SIG}}^{\mathsf{sk}_1}, S_{\mathsf{SIG}}^{\mathsf{sk}_2}}(1^t, \mathsf{vk}, \pi, \rho) \equiv$

1. $\langle \widetilde{P}_1 \rangle \leftarrow \mathsf{make1prover}^{\uparrow\widetilde{P}}(\mathsf{vk})$.
2. Parse vk as $(\mathsf{vk}_1, \mathsf{vk}_2)$.
3. Parse $\pi = (\pi', \sigma')$ and π' as $(\sigma_{\mathsf{iw}}, \pi_1)$.
4. $y_{\mathsf{aug}} \leftarrow \mathsf{AUG}(\mathsf{vk}_2, \sigma_{\mathsf{iw}}, y)$.
5. $w \leftarrow E_1^{\uparrow\left(\widetilde{P}_1^{S_{\mathsf{SIG}}^{\mathsf{sk}_2}, \uparrow\widetilde{P}}\right), S_{\mathsf{SIG}}^{\mathsf{sk}_1}}(1^t, \mathsf{vk}_1, y_{\mathsf{aug}}, \rho)$.
6. Output w.

By the weak PoK property of (P_1, V_1), the knowledge extractor E_1 outputs a valid witness w for y_{aug} with non-negligible probability, and thus (A.2) contradicts the existential unforgeability of SIG.[3]

[3]The argument in this section needs adjusting to correct for the potential (anti)correlation between the success of $\mathsf{WE}_{\mathsf{APHA}}$ and the failure of LE, since forgery is achieved only in the intersection of the two events. Unfortunately, this gap in the argument was discovered after the thesis text was finalized. Please contact the author directly to obtain a revised discussion

Case 3: $C \subset S$ is the set of those coin tosses for which $\mathcal{M}_{\sigma_{\mathsf{iw}}} = \{(q_1, a_1), \ldots, (q_d, a_d)\}$ and there exist two different pairs $(q_i, a_i) \neq (q_j, a_j)$. This set has negligible probability mass, because the existential unforgeability property of SIG implies that no efficient adversary can find two messages with the same signature:

Lemma A.4.4. *Let F be a $\mathsf{poly}(k)$-size circuit that is given oracle access to a signing oracle from* SIG. *Then, with all but negligible probability, the transcript of interaction of F with the signing oracle contains no two different queries q_i and q_j that have as answer the same signature.*

Proof. Take $i < j$ and let both be the least such indices. Let h be a bound on the number of queries of F. (Note that h is polynomial in k.) Consider the circuit F' that, on input a signature verification key and with oracle access to a signing oracle, does the following:

1. Draw $i, j \in [h]$ as guesses for i and j. (If $j \geq i$ abort.)
2. Run F on input the signature verification key, forwarding its queries to the signing oracle, until F queries its j-th query q_j. Let σ_i be the answer to the i-th query q_i of F.
3. Output (q_j, σ_i).

The circuit F' forges a valid signature for an unqueried message with success probability that is at least $1/h^2$ times the probability that F's transcript contains two different queries with the same signature. Therefore, since SIG is existentially unforgeable, F's transcript contains two different queries with the same signature with negligible probability. □

Case 4: $D \subset S$ is the set of those coin tosses for which $\mathcal{M}_{\sigma_{\mathsf{iw}}} = \{(q_1, a_1)\}$ and all the q_1 is valid with respect to y. This is the *good subset*, the above a_1 would be identified in Step 3a of LE, and the corresponding triple will be added to extlist in Step 3b.

We conclude that the coin tosses for which the extractor works, i.e., those in the subset D of S, have most of the probability mass in S, thus proving the list extraction property. □

We remark that we actually proved a slightly stronger property than the APHA list-extraction property, because the query answered with the signature σ_{iw} is unique. However, this additional requirement is not needed for constructing PCD systems using APHA systems, so we do not incorporate it into the definition of APHA systems.

---

of this point.

Appendix B

Full Proof of Security for PCD Systems

We prove the PCD proof-of-knowledge property for the PCD system $(G_{\mathsf{PCD}}, P_{\mathsf{PCD}}, V_{\mathsf{PCD}})$ constructed in Section 5.3, thus completing the proof of Theorem 5.3.1, which was sketched in Section 5.4.

We briefly review the property that we want to prove, and then give the details of the proof. As before, we restrict our attention to distributed computation transcripts for which programs have exactly two inputs and one (alleged) output.

B.1 Review of goals

Given a compliance predicate $\mathbf{C}$ and an output string z, consider a (possibly cheating) prover circuit $\widetilde{P}$ that, on input $(\mathsf{vk}, \mathbf{C}, z)$ and with oracle access to O, outputs a convincing proof string π for z with some non-negligible probability $\delta(k)$ in the security parameter k. The probability is taken over the internal coin tosses of the PCD oracle generator G_{PCD} and oracle O.

Our goal is to construct a knowledge extractor circuit E_{PCD} that, on input $\left(\widetilde{P}, \mathsf{vk}, \mathbf{C}, z\right)$ and with oracle access to O, outputs a $\mathbf{C}$-compliant distributed computation transcript DC with final output z, with probability equal to $\delta(k) - \mu(k)$, for some negligible function $\mu(k)$. Moreover, the size of E_{PCD} should be $\mathsf{poly}(k)$.

B.2 Details of proof

Fix a prover circuit $\widetilde{P}$, a compliance predicate $\mathbf{C}$, and an output string z. We give the strategy for the PCD knowledge extractor E_{PCD}, and then prove that the strategy works. Note that, as we describe the strategy, we assume that it is possible to perform the prescribed steps; if ever the strategy cannot perform one of the steps as indicated, it simply aborts and outputs $\perp$.

Following the intuition sketched in Section 5.4, the PCD knowledge extractor E_{PCD}, on input $(\widetilde{P}, \mathsf{vk}, \mathbf{C}, z)$ and with oracle access to O, does the following:

1. Run $\widetilde{P}^O(\mathsf{vk}, \mathbf{C}, z)$ to get its output (z, π) and the query-answer transcript of its oracle accesses $[\![\widetilde{P}(\mathsf{vk}, \mathbf{C}, z), O]\!]$.
2. Make a new augmented distributed computation transcript $\mathsf{ADC} = (G, \mathsf{code}, \mathsf{data}, \mathsf{proof})$ by defining the following quantities:
   (a) $V := \{1, 2\}$ and $E := \{(2, 1)\}$.
   (b) $\mathsf{code}(1) := \perp$ and $\mathsf{code}(2) := \perp$.
   (c) $\mathsf{data}(2, 1) := z$.
   (d) $\mathsf{proof}(2, 1) := \pi$.
3. Make a new exploration stack and push on it the "first" edge:

(a) Create an empty stack $\mathsf{expstack}$.

(b) Add the edge $(2, 1)$ to $\mathsf{expstack}$.

4. Make a new vertex label counter: $i \leftarrow 2$.

5. Make a new cache cache. (Its entries will be the instance-witness pairs that we found, together with their corresponding edges.)

6. Using the APHA list extractor, extract the list from the prover circuit: $\mathsf{extlist} \leftarrow \mathsf{LE}\big([\![\widetilde{P}(\mathsf{vk}, \mathbf{C}, z), O]\!]\big)$.

7. `while` $\mathsf{expstack}$ is not empty `do`:

(a) Pop off the next edge from the exploration stack: $(j, k) \leftarrow \mathsf{expstack.pop}()$.

(b) Construct the instance $y_{(j,k)}$ and the APHA proof string $\pi'_{(j,k)}$ corresponding to the edge (j, k):

i. $z_{(j,k)} \leftarrow \mathsf{data}(j, k)$.

ii. $\pi_{(j,k)} \leftarrow \mathsf{proof}(j, k)$.

iii. Parse $\pi_{(j,k)}$ as $(\pi'_{(j,k)}, d_{(j,k)}, t_{(j,k)})$ and define $y \equiv \big(M^{\mathsf{vk},\mathbf{C}}_{\mathsf{PCD}}, (z_{(j,k)}, d_{(j,k)}), t_{(j,k)}\big)$.

(c) Find a triple (y, π', w) in $\mathsf{extlist}$ such that $y = y_{(j,k)}$ and $\pi' = \pi'_{(j,k)}$; if there is none, abort.

(d) If $(y_{(j,k)}, w)$ already appears in cache as some entry $((y_{(j,k)}, w), (j', k'))$, then add an edge from vertex j' to vertex k with $\mathsf{data}(j', k) = z_{(j,k)}$ and $\mathsf{proof}(j', k) = \pi_{(j,k)}$, and then delete vertex j from the graph.

(e) If $(y_{(j,k)}, w)$ does not appear in cache as part of any entry, then:

i. Put $(y_{(j,k)}, w)$ in cache as entry $((y_{(j,k)}, w), (j, k))$.

ii. Parse the witness w as $(\mathsf{prg}, z_{\mathsf{in}_1}, \pi_{\mathsf{in}_1}, z_{\mathsf{in}_2}, \pi_{\mathsf{in}_2})$.

iii. Extend the augmented transcript ADC with the two newly discovered edges and their labels:

A. $V := V \cup \{i + 1, i + 2\}$ and $E := E \cup \{(i + 1, j), (i + 2, j)\}$.

B. $\mathsf{code}(j) := \mathsf{prg}$.

C. $\mathsf{data}(i + 1, j) := z_{\mathsf{in}_1}$ and $\mathsf{data}(i + 2, j) := z_{\mathsf{in}_2}$.

D. $\mathsf{proof}(i + 1, j) := \pi_{\mathsf{in}_1}$ and $\mathsf{proof}(i + 2, j) := \pi_{\mathsf{in}_2}$.

iv. `if` $\pi_{\mathsf{in}_2} \neq \perp$, `do`:

A. Parse π_{in_2} as $(\pi'_{\mathsf{in}_2}, d_{\mathsf{in}_2}, t_{\mathsf{in}_2})$.

B. If $d_{\mathsf{in}_2} < d$, add the new edge $(i + 2, j)$ to the exploration stack $\mathsf{expstack}$.

v. `if` $\pi_{\mathsf{in}_1} \neq \perp$, `do`:

A. Parse π_{in_1} as $(\pi'_{\mathsf{in}_1}, d_{\mathsf{in}_1}, t_{\mathsf{in}_1})$.

B. If $d_{\mathsf{in}_1} < d$, add the new edge $(i + 1, j)$ to the exploration stack $\mathsf{expstack}$.

vi. Increase the vertex label counter by two: $i \leftarrow i + 2$.

8. Strip from ADC the label proof, to obtain a distributed computation transcript DC, and output DC.

We now argue that E_{PCD}, defined as the strategy above converted to a circuit, works. First, we show that we can usefully bound the number of loop iterations.

Lemma B.2.1. *The number of* `while` *loop iterations is* $\mathsf{poly}(k)$.

Proof. In every iteration of the `while` loop, an edge is popped off from $\mathsf{expstack}$ (Step 7a), a corresponding witness w is found in the extracted list $\mathsf{extlist}$ from the APHA list extractor LE (Step 7c), and then either:

– the corresponding instance-witness pair already appears in cache (Step 7d), so that an additional edge is added to ADC to reflect that; in this case, the size of $\mathsf{expstack}$ decreases by one while the size of cache remains unchanged;

– the corresponding instance-witness pair is not in cache (Step 7e), so it is added there and then some more work is done, which also involves pushing on $\mathsf{expstack}$ two new edges; in this case, the size of $\mathsf{expstack}$ increases by one (one pop and two pushes) while the size of cache increases by one.

In summary, in every iteration of the `while` loop, either the size of $\mathsf{expstack}$ decreases by one, or the sizes of *both* $\mathsf{expstack}$ and cache increase by one.

However, every instance-witness pair (y, w) that is added to cache comes from some triple (y, π', w) in $\mathsf{extlist}$, and $\mathsf{extlist}$ is a list of at most l triples, where l is at most $|\mathsf{LE}([\![\widetilde{P}(\mathsf{vk}, \mathbf{C}, z), O]\!])| = \mathsf{poly}(k)$. Hence, cache has at most l entries. Therefore, there are at most $2l = \mathsf{poly}(k)$ iterations before $\mathsf{expstack}$ becomes empty, and the strategy halts. □

Next, we observe that each iteration does not take up too much work.

Lemma B.2.2. *Every iteration of the* `while` *loop takes* $\mathsf{poly}(k)$ *time to complete.*

Proof. Before the first iteration of the `while` loop, all the data structures have size that is $\mathsf{poly}(k)$. In each iteration, the sizes of all the data structures increase by only an additive factor that is $\mathsf{poly}(k)$. By Lemma B.2.1, there are at most $\mathsf{poly}(k)$ iterations of the `while` loop. We conclude that all data structures have size bounded by $\mathsf{poly}(k)$. All steps, being (polynomial-time) operations on the data structures, take time that is $\mathsf{poly}(k)$. □

Hence, we can now deduce that the size of E_{PCD} is indeed what we want.

Corollary 3. $|E_{\mathsf{PCD}}|$ *is* $\mathsf{poly}(k)$.

Finally, we prove that E_{PCD} extracts a $\mathbf{C}$-compliant augmented transcript from any sufficiently convincing prover circuit, except with negligible probability.

Lemma B.2.3. *If*

$$\Pr\left[V_{\mathsf{PCD}}(\mathsf{vk}, \mathbf{C}, z, \pi) = 1 \;\middle|\; (O, \mathsf{vk}) \leftarrow G_{\mathsf{PCD}}(1^k)\ ;\ \pi \leftarrow \widetilde{P}^O(\mathsf{vk}, \mathbf{C}, z)\right] = \delta(k)$$

(where the probability is taken over the internal randomness of G_{PCD} and O) for some non-negligible function δ, then

$$\Pr\Big[\mathsf{DC} \text{ is } \mathbf{C}\text{-compliant} \ \wedge\ u, v \in V \ \wedge\ \mathsf{DC} = \mathsf{DC}|_{(u,v)} \ \wedge\ z = \mathsf{data}(u, v) \ \Big| \ (O, \mathsf{vk}) \leftarrow G_{\mathsf{PCD}}(1^k)\ ;\ \mathsf{DC} \leftarrow E^O_{\mathsf{PCD}}(\widetilde{P}, \mathsf{vk}, \mathbf{C}, z)\Big] = \delta(k) - \mu(k)$$

(where the probability is taken over the internal randomness of G_{PCD} and O) for some negligible function μ.

Proof. We want to show that, whenever $\widetilde{P}$ convinces V_{PCD} to accept some output z and proof string π, except with negligible probability, E_{PCD} successfully extracts a $\mathbf{C}$-compliant transcript with output z. By construction of E_{PCD}, this holds if *every time* Step 7c is executed, the desired triple (y, π', w) is found in $\mathsf{extlist}$ and the witness w is such that $(y_{(j,k)}, w) \in R_{\mathcal{U}}$. We now argue that, whenever $\widetilde{P}$ convinces V_{PCD}, this is the case, except with negligible probability.

Define $\widetilde{P}'$ to be the following circuit:

> $\widetilde{P}'$, on input vk and with oracle access to O, does the following:
> 1. $\pi \leftarrow \widetilde{P}(\mathsf{vk}, \mathbf{C}, z)$.
> 2. Parse π as (π', d, t) and define $y \equiv \big(M^{\mathsf{vk},\mathbf{C}}_{\mathsf{PCD}}, (z, d), t\big)$.
> 3. Output (y, π').

Observe that, whenever $\widetilde{P}$ convinces a PCD verifier V_{PCD}, $\widetilde{P}'$ convinces an APHA verifier, because $\widetilde{P}'$ is simply an interface around $\widetilde{P}$ that "peels off" the PCD layer to expose the APHA layer underneath. Moreover, observe that, for any $(O, \mathsf{vk}) \in G_{\mathsf{PCD}}(1^k)$, it holds that $[\![\widetilde{P}'(\mathsf{vk}), O]\!] = [\![\widetilde{P}(\mathsf{vk}, \mathbf{C}, z), O]\!]$ and $|[\![\widetilde{P}(\mathsf{vk}, \mathbf{C}, z), O]\!]| \leq m \equiv |\langle \widetilde{P} \rangle|$.

Next, for $i \in [m]$, consider the circuit C_i defined as follows:

> C_i, on input vk and oracle access to O, does the following:
> 1. Run $\widetilde{P}'^O(\mathsf{vk})$ to obtain $[\![\widetilde{P}'(\mathsf{vk}), O]\!]$.
> 2. $\mathsf{extlist} \leftarrow \mathsf{LE}\big([\![\widetilde{P}'(\mathsf{vk}), O]\!]\big)$.

3. Let (y_i, π'_i, w_i) be the i-th entry of $\mathsf{extlist}$, and output (y_i, π'_i, w_i).

By the APHA list-extraction property, whenever (y_i, π'_i) convinces V_{APHA}, then (y_i, w_i) is a valid instance-witness pair, except with negligible probability. Thus, by a simple union bound, we obtain:

$$\Pr\left[\forall\, i \in [m]\,,\ \big(V_{\mathsf{APHA}}(\mathsf{vk}, y_i, \pi'_i) = 1\big) \longrightarrow \big((y_i, w_i) \in R_{\mathcal{U}}\big) \;\middle|\; (O, \mathsf{vk}) \leftarrow G_{\mathsf{PCD}}(1^k)\ ;\ (y_i, \pi'_i, w_i) \leftarrow C_i^O(\mathsf{vk})\right] > 1 - \mu(k)\ ,$$

i.e., except with negligible probability, *for every* $i \in [m]$, whenever (y_i, π'_i) convinces V_{APHA}, it holds that (y_i, w_i) is a valid instance-witness pair.

Next, for $i \in [m]$ and $b \in \{1, 2\}$, consider the circuit $D_{i,b}$ defined as follows:

$D_{i,b}$, on input vk and oracle access to O, does the following:

1. Run $\widetilde{P}'^O(\mathsf{vk})$ to obtain $[\![\widetilde{P}'(\mathsf{vk}), O]\!]$.
2. $\mathsf{extlist} \leftarrow \mathsf{LE}\big([\![\widetilde{P}'(\mathsf{vk}), O]\!]\big)$.
3. Let (y_i, π'_i, w_i) be the i-th entry of $\mathsf{extlist}$.
4. Parse w_i as $(\mathsf{prg}, z_{\mathsf{in}_1}, \pi_{\mathsf{in}_1}, z_{\mathsf{in}_2}, \pi_{\mathsf{in}_2})$.
5. Parse $\pi_{\mathsf{in_b}}$ as $(\pi'_{\mathsf{in_b}}, d_{\mathsf{in_b}}, t_{\mathsf{in_b}})$ and define $y_{\mathsf{in_b}} \equiv \big(M^{\mathsf{vk},\mathbf{C}}_{\mathsf{PCD}}, (z_{\mathsf{in_b}}, d_{\mathsf{in_b}}), t_{\mathsf{in_b}}\big)$.
6. Output $(y_{\mathsf{in_b}}, \pi'_{\mathsf{in_b}})$.

By the APHA list-extraction property, whenever $(y_{\mathsf{in_b}}, \pi'_{\mathsf{in_b}})$ convinces V_{APHA}, then $(y_{\mathsf{in_b}}, \pi'_{\mathsf{in_b}})$ appears as part of a triple in $\mathsf{extlist}$ (output by $\mathsf{LE}\big([\![\widetilde{P}'(\mathsf{vk}), O]\!]\big)$), except with negligible probability. Similarly to above, by a simple union bound, we obtain:

$$\Pr\left[\forall\, i \in [m], b \in \{1, 2\}\,,\ \big(V_{\mathsf{APHA}}(\mathsf{vk}, y_{\mathsf{in_b}}, \pi'_{\mathsf{in_b}}) = 1\big) \longrightarrow \big((y_{\mathsf{in_b}}, \pi'_{\mathsf{in_b}}) \text{ is part of some triple in } \mathsf{extlist}\big) \;\middle|\; (O, \mathsf{vk}) \leftarrow G_{\mathsf{PCD}}(1^k)\ ;\ (y_{\mathsf{in_b}}, \pi'_{\mathsf{in_b}}) \leftarrow D^O_{i,b}(\mathsf{vk})\ ;\ \mathsf{extlist} \leftarrow \mathsf{LE}\big([\![\widetilde{P}'(\mathsf{vk}), O]\!]\big)\right] > 1 - \mu(k)\ ,$$

i.e., except with negligible probability, *for every* $i \in [m]$ *and* $b \in \{1, 2\}$, whenever $(y_{\mathsf{in_b}}, \pi'_{\mathsf{in_b}})$ convinces V_{APHA}, it holds that $(y_{\mathsf{in_b}}, \pi'_{\mathsf{in_b}})$ can be found as some triple in $\mathsf{extlist}$.

We can put the above two arguments together, to obtain the *for every* $i \in [m]$ *and* $b \in \{1, 2\}$, except with negligible probability, whenever (y_i, π'_i) convinces V_{APHA}, it holds that $(y_i, w_i) \in R_{\mathcal{U}}$ and, moreover, each of the two pairs $(y_{\mathsf{in}_1}, \pi'_{\mathsf{in}_1})$ and $(y_{\mathsf{in}_2}, \pi'_{\mathsf{in}_2})$, which can be obtained from w_i, are convincing (because the PCD machine $M^{\mathsf{vk},\mathbf{C}}_{\mathsf{PCD}}$ in instance y_i verifies them, and w_i is a valid witness for y_i) and can be found in $\mathsf{extlist}$.

Hence, by an inductive argument, it follows that, except with negligible probability, *every time* Step 7c is executed, a valid witness is found for the instance. $\square$

Statement of Originality

The results contained in this thesis (specifically, Chapter 4, Chapter 5, and Chapter 6) are original research that is joint work with Eran Tromer, and are contained in the following paper:

– Alessandro Chiesa and Eran Tromer, *Proof-Carrying Data and Hearsay Arguments from Signature Cards.* Proceedings of the 1st Symposium on Innovations in Computer Science, 310-331, Tsinghua University Press, 2010. [36]

Bibliography

[1] Dakshi Agrawal, Selcuk Baktir, Deniz Karakoyunlu, Pankaj Rohatgi, and Berk Sunar. Trojan detection using IC fingerprinting. In *SP '07: Proceedings of the 2007 IEEE Symposium on Security and Privacy*, pages 296–310, Washington, DC, USA, 2007. IEEE Computer Society.

[2] Ross J. Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems*. Wiley Publishing, 2nd edition, 2008.

[3] Gregory R. Andrews and Richard P. Reitman. An axiomatic approach to information flow in programs. *ACM Transactions on Programming Languages and Systems*, 2(1):56–76, 1980.

[4] Sanjeev Arora. How NP got a new definition: a survey of probabilistically checkable proofs. In *ICM '02: Proceedings of the 2002 International Congress of Mathematicians*, volume 3, pages 637–648, 2002.

[5] Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, New York, NY, USA, 2009.

[6] Sanjeev Arora and Shmuel Safra. Probabilistic checking of proofs: a new characterization of NP. *Journal of the ACM*, 45(1):70–122, 1998.

[7] Sanjeev Arora, Carsten Lund, Rajeev Motwani, Madhu Sudan, and Mario Szegedy. Proof verification and the hardness of approximation problems. *Journal of the ACM*, 45(3):501–555, 1998.

[8] Vikraman Arvind and Johannes Köbler. On resource-bounded measure and pseudorandomness. In *Proceedings of the 17th Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 235–249, London, UK, 1997. Springer-Verlag.

[9] Dmitri Asonov and Rakesh Agrawal. Keyboard acoustic emanations. In *SP '04: Proceedings of the 2004 IEEE Symposium on Security and Privacy*, pages 3–11, Washington, DC, USA, 2004. IEEE Computer Society.

[10] László Babai. Trading group theory for randomness. In *STOC '85: Proceedings of the 17th Annual ACM Symposium on Theory of Computing*, pages 421–429, New York, NY, USA, 1985. ACM.

[11] László Babai and Shlomo Moran. Arthur-Merlin games: a randomized proof system, and a hierarchy of complexity class. *Journal of Computer and System Sciences*, 36(2):254–276, 1988.

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

[13] Boaz Barak. How to go beyond the black-box simulation barrier. In *FOCS '01: Proceedings of the 42nd Annual IEEE Symposium on Foundations of Computer Science*, pages 106–115, Washington, DC, USA, 2001. IEEE Computer Society.

[14] Boaz Barak. Constant-round coin-tossing with a man in the middle or realizing the shared random string model. In *FOCS '02: Proceedings of the 43rd Annual IEEE Symposium on Foundations of Computer Science*, pages 345–355, Washington, DC, USA, 2002. IEEE Computer Society.

[15] Boaz Barak and Oded Goldreich. Universal arguments and their applications. In *CCC '02: Proceedings of the 17th IEEE Annual Conference on Computational Complexity*, pages 194–203, Washington, DC, USA, 2002. IEEE Computer Society.

[16] Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil P. Vadhan, and Ke Yang. On the (im)possibility of obfuscating programs. In *CRYPTO '01: Proceedings of the 21st Annual International Cryptology Conference on Advances in Cryptology*, pages 1–18, London, UK, 2001. Springer-Verlag.

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

[18] Mihir Bellare and Silvio Micali. How to sign given any trapdoor function. In *STOC '88: Proceedings of the 20th Annual ACM Symposium on Theory of Computing*, pages 32–42, New York, NY, USA, 1988. ACM.

[19] Mihir Bellare, Russell Impagliazzo, and Moni Naor. Does parallel repetition lower the error in computationally sound protocols? In *FOCS '97: Proceedings of the 38th Annual IEEE Symposium on Foundations of Computer Science*, pages 374–383, Washington, DC, USA, 1997. IEEE Computer Society.

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

[21] Eli Ben-Sasson. Probabilistically checkable proofs, 2007. Available online at http://www.cs.technion.ac.il/~eli/courses/2007_Fall/.

[22] Eli Biham and Adi Shamir. Differential fault analysis of secret key cryptosystems. In *CRYPTO '97: Proceedings of the 17th Annual International Cryptology Conference on Advances in Cryptology*, pages 513–525, London, UK, 1997. Springer-Verlag.

[23] Eli Biham, Yaniv Carmeli, and Adi Shamir. Bug attacks. In *CRYPTO '08: Proceedings of the 28th Annual International Cryptology Conference on Advances in Cryptology*, pages 221–240, Berlin, Heidelberg, 2008. Springer-Verlag.

[24] Manuel Blum and Silvio Micali. How to generate cryptographically strong sequences of pseudo-random bits. *SIAM Journal on Computing*, 13(4):850–864, 1984.

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

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

[27] David Brumley and Dan Boneh. Remote timing attacks are practical. *Computer Networks: The International Journal of Computer and Telecommunications Networking*, 48(5):701–716, 2005.

[28] Christian Cachin, Silvio Micali, and Markus Stadler. Computationally private information retrieval with polylogarithmic communication. In *EUROCRYPT '99: Proceedings of the 18th Annual International Conference on Advances in Cryptology*, pages 402–414, Berlin, Heidelberg, 1999. Springer-Verlag.

[29] Ran Canetti and Marc Fischlin. Universally composable commitments. In *CRYPTO '01: Proceedings of the 21st Annual International Cryptology Conference on Advances in Cryptology*, pages 19–40, London, UK, 2001. Springer-Verlag.

[30] Ran Canetti, Oded Goldreich, and Shai Halevi. The random oracle methodology, revisited. In *STOC '98: Proceedings of the 30th Annual ACM Symposium on Theory of Computing*, pages 209–218, New York, NY, USA, 1998. ACM.

[31] Ran Canetti, Oded Goldreich, and Shai Halevi. The random oracle methodology, revisited. *Journal of the ACM*, 51(4):557–594, 2004.

[32] Nishanth Chandran, Vipul Goyal, and Amit Sahai. New constructions for UC secure computation using tamper-proof hardware. In *EUROCRYPT '08: Proceedings of the 27th Annual International Conference on Advances in Cryptology*, pages 545–562, Berlin, Heidelberg, 2008. Springer-Verlag.

[33] Richard Chang, Suresh Chari, Desh Ranjan, and Pankaj Rohatgi. Relativization: a revisionistic retrospective. *Bulletin of the European Association for Theoretical Computer Science*, 47:144–153, 1992.

[34] David Chaum, Claude Crépeau, and Ivan Damgård. Multiparty unconditionally secure protocols. In *STOC '88: Proceedings of the 20th Annual ACM Symposium on Theory of Computing*, pages 11–19, New York, NY, USA, 1988. ACM.

[35] Alessandro Chiesa and Eran Tromer. Proof-carrying data, 2009. Web site at http://projects.csail.mit.edu/pcd.

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

[37] Byung-Gon Chun, Petros Maniatis, Scott Shenker, and John Kubiatowicz. Attested append-only memory: making adversaries stick to their word. *ACM SIGOPS Operating Systems Review*, 41(6): 189–204, 2007.

[38] Kai-Min Chung and Feng-Hao Liu. Parallel repetition theorems for interactive arguments. In *TCC '10: Proceedings of the 7th Theory of Cryptography Conference on Theory of Cryptography*, pages 19–36, Berlin, Heidelberg, 2010. Springer-Verlag.

[39] Christopher Colby, Peter Lee, and George C. Necula. A proof-carrying code architecture for Java. In *CAV '00: Proceedings of the 12th International Conference on Computer Aided Verification*, pages 557–560, London, UK, 2000. Springer-Verlag.

[40] Complexity Zoo. http://www.complexityzoo.com.

[41] Victor Costan, Luis F. Sarmenta, Marten Dijk, and Srinivas Devadas. The trusted execution module: Commodity general-purpose trusted computing. In *CARDIS '08: Proceedings of the 8th IFIP WG 8.8/11.2 International Conference on Smart Card Research and Advanced Applications*, pages 133–148, Berlin, Heidelberg, 2008. Springer-Verlag.

[42] Ronald Cramer, Ivan Damgård, and Jesper Buus Nielsen. Multiparty computation, an introduction, 2010. Available online at http://www.daimi.au.dk/~ivan/smc.pdf.

[43] Ivan Damgård. A design principle for hash functions. In *CRYPTO '89: Proceedings of the 9th Annual International Cryptology Conference on Advances in Cryptology*, pages 416–427, London, UK, 1990. Springer-Verlag.

[44] Ivan Damgård. Collision free hash functions and public key signature schemes. In *EUROCRYPT '87: Proceedings of the 6th Annual International Conference on Advances in Cryptology*, pages 203–216, Berlin, Heidelberg, 1988. Springer-Verlag.

[45] Ivan Damgård, Jesper Buus Nielsen, and Daniel Wichs. Universally composable multiparty computation with partially isolated parties. In *TCC '09: Proceedings of the 6th Theory of Cryptography Conference on Theory of Cryptography*, pages 315–331, Berlin, Heidelberg, 2009. Springer-Verlag.

[46] Dorothy E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19 (5):236–243, 1976.

[47] Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, 1977.

[48] Bundesministerium der Justiz. Gesetz über Rahmenbedingungen für elektronische Signaturen. *Bundesgesetzblatt I 2001, 876*, May 2001. online at http://bundesrecht.juris.de/bundesrecht/sigg_2001/inhalt.html.

[49] Whit Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory"*, IT-22(6):644–654, 1976.

[50] Uriel Feige, Amos Fiat, and Adi Shamir. Zero knowledge proofs of identity. In *STOC '87: Proceedings of the 19th Annual ACM Symposium on Theory of Computing*, pages 210–217, New York, NY, USA, 1987. ACM.

[51] Uriel Feige, Shafi Goldwasser, Laszlo Lovász, Shmuel Safra, and Mario Szegedy. Interactive proofs and the hardness of approximating cliques. *Journal of the ACM*, 43(2):268–292, 1996.

[52] Amos Fiat and Adi Shamir. How to prove yourself: practical solutions to identification and signature problems. In *CRYPTO '86: Proceedings of the 6th Annual International Cryptology Conference on Advances in Cryptology*, pages 186–194, London, UK, 1987. Springer-Verlag.

[53] Marc Fischlin. Communication-efficient non-interactive proofs of knowledge with online extractors. In *CRYPTO '05: Proceedings of the 25th Annual International Cryptology Conference on Advances in Cryptology*, pages 152–168, London, UK, 2005. Springer-Verlag.

[54] Lance Fortnow. The role of relativization in complexity theory. *Bulletin of the European Association for Theoretical Computer Science*, 52:229–244, 1994.

[55] Lance Fortnow and Rahul Santhanam. Infeasibility of instance compression and succinct PCPs for NP. In *STOC '08: Proceedings of the 40th Annual ACM Symposium on Theory of Computing*, pages 133–142, New York, NY, USA, 2008. ACM.

[56] Martin Fürer, Oded Goldreich, Yishay Mansour, Michael Sipser, and Stathis Zachos. On completeness and soundness in interactive proof systems. *Advances in Computing Research: A Research Annual*, 5 (Randomness and Computation, S. Micali, ed.:429–442, 1989.

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

[58] Chris GauthierDickey, Daniel Zappala, Virginia Lo, and James Marr. Low latency and cheat-proof event ordering for peer-to-peer games. In *NOSSDAV '04: Proceedings of the 14th International Workshop on Network and Operating Systems Support for Digital Audio and Video*, pages 134–139, New York, NY, USA, 2004. ACM.

[59] Oded Goldreich. *Modern Cryptography, Probabilistic Proofs, and Pseudorandomness*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1998.

[60] Oded Goldreich. *Foundations of Cryptography: Volume 1, Basic Tools*. Cambridge University Press, New York, NY, USA, 2000.

[61] Oded Goldreich. *Foundations of Cryptography: Volume 2, Basic Applications*. Cambridge University Press, New York, NY, USA, 2004.

[62] Oded Goldreich. *Computational Complexity: A Conceptual Perspective*. Cambridge University Press, New York, NY, USA, 2008.

[63] Oded Goldreich. A short tutorial of zero-knowledge, 2010. Available online at http://www.wisdom.weizmann.ac.il/~oded/zk-tut02.html.

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

[65] Oded Goldreich and Bernd Meyer. Computational indistinguishability: algorithms vs. circuits. *Theoretical Computer Science*, 191(1-2):215–218, 1998.

[66] Oded Goldreich and Yair Oren. Definitions and properties of zero-knowledge proof systems. *Journal of Cryptology*, 7(1):1–32, December 1994.

[67] Oded Goldreich, Shafi Goldwasser, and Silvio Micali. How to construct random functions. *Journal of the ACM*, 33(4):792–807, 1986.

[68] Oded Goldreich, Yishay Mansour, and Michael Sipser. Interactive proof systems: Provers that never fail and random selection. In *FOCS '87: Proceedings of the 28th Annual IEEE Symposium on Foundations of Computer Science*, pages 462–471, Washington, DC, USA, 1987. IEEE Computer Society.

[69] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play ANY mental game. In *STOC '87: Proceedings of the 19th Annual ACM Symposium on Theory of Computing*, pages 218–229, New York, NY, USA, 1987. ACM.

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

[71] Shafi Goldwasser. Multi party computations: past and present. In *PODC '97: Proceedings of the 16th Annual ACM Symposium on Principles of Distributed Computing*, pages 1–6, New York, NY, USA, 1997. ACM.

[72] Shafi Goldwasser and Silvio Micali. Probabilistic encryption. *Journal of Computer and System Sciences*, 28(2):270–299, 1984.

[73] Shafi Goldwasser and Michael Sipser. Private coins versus public coins in interactive proof systems. In *STOC '86: Proceedings of the 18th Annual ACM Symposium on Theory of Computing*, pages 59–68, New York, NY, USA, 1986. ACM.

[74] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof-systems. In *STOC '85: Proceedings of the 17th Annual ACM Symposium on Theory of Computing*, pages 291–304, New York, NY, USA, 1985. ACM.

[75] Shafi Goldwasser, Silvio Micali, and Ronald L. Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM Journal on Computing*, 17(2):281–308, 1988.

[76] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof systems. *SIAM Journal on Computing*, 18(1):186–208, 1989.

[77] Shafi Goldwasser, Yael Tauman Kalai, and Guy N. Rothblum. Delegating computation: interactive proofs for muggles. In *STOC '08: Proceedings of the 40th Annual ACM Symposium on Theory of Computing*, pages 113–122, New York, NY, USA, 2008. ACM.

[78] Venkatesan Guruswami and Ryan O'Donnell. The PCP theorem and hardness of approximation, 2005. Available online at http://www.cs.washington.edu/education/courses/533/05au/.

[79] Stuart Haber. *Multi-Party Cryptographic Computations: Techniques and Applications*. PhD thesis, Columbia University, Computer Science Department, November 1987.

[80] Stuart Haber and Benny Pinkas. Securely combining public-key cryptosystems. In *CCS '01: Proceedings of the 8th ACM Conference on Computer and Communications Security*, pages 215–224, New York, NY, USA, 2001. ACM.

[81] Iftach Haitner. A parallel repetition theorem for any interactive argument. In *FOCS '09: Proceedings of the 50th Annual IEEE Symposium on Foundations of Computer Science*, pages 241–250, Washington, DC, USA, 2009. IEEE Computer Society.

[82] Danny Harnik and Moni Naor. On the compressibility of NP instances and cryptographic applications. In *FOCS '06: Proceedings of the 47th Annual IEEE Symposium on Foundations of Computer Science*, pages 719–728, Washington, DC, USA, 2006. IEEE Computer Society.

[83] Dennis Hofheinz, Jörn Müller-Quade, and Dominique Unruh. Universally composable zero-knowledge arguments and commitments from signature cards. In *MoraviaCrypt '05: Proceedings of the 5th Central European Conference on Cryptography*, pages 93–103, 2005.

[84] Johan Håstad, Russell Impagliazzo, Leonid A. Levin, and Michael Luby. A pseudorandom generator from any one-way function. *SIAM Journal on Computing*, 28(4):1364–1396, 1999.

[85] Russell Impagliazzo and Avi Wigderson. P = BPP if E requires exponential circuits: derandomizing the XOR lemma. In *STOC '97: Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 220–229, New York, NY, USA, 1997. ACM.

[86] Yuval Ishai, Eyal Kushilevitz, and Rafail Ostrovsky. Efficient arguments without short PCPs. In *CCC '07: Proceedings of the Twenty-Second Annual IEEE Conference on Computational Complexity*, pages 278–291, Washington, DC, USA, 2007. IEEE Computer Society.

[87] Rahul Jain, Zhengfeng Ji, Sarvagya Upadhyay, and John Watrous. QIP = PSPACE. In *STOC '10: Proceedings of the 42nd Annual ACM Symposium on Theory of Computing*, New York, NY, USA, 2010. ACM.

[88] Yael Tauman Kalai and Ran Raz. Interactive PCP. In *ICALP '08: Proceedings of the 35th International Colloquium on Automata, Languages and Programming, Part II*, pages 536–547, Berlin, Heidelberg, 2008. Springer-Verlag.

[89] Yael Tauman Kalai and Ran Raz. Probabilistically checkable arguments. In *CRYPTO '09: Proceedings of the 29th Annual International Cryptology Conference on Advances in Cryptology*, pages 143–159, London, UK, 2009. Springer-Verlag.

[90] Jonathan Katz. Universally composable multi-party computation using tamper-proof hardware. In *EUROCRYPT '07: Proceedings of the 26th Annual International Conference on Advances in Cryptology*, pages 115–128, Berlin, Heidelberg, 2007. Springer-Verlag.

[91] Jonathan Katz and Chiu-Yuen Koo. On constructing universal one-way hash functions from arbitrary one-way functions. Cryptology ePrint Archive, Report 2005/328, 2005. Available at http://eprint.iacr.org/2005/328.

[92] Joe Kilian. Zero-knowledge with log-space verifiers. In *SFCS '88: Proceedings of the 29th Annual IEEE Symposium on Foundations of Computer Science*, pages 25–35, Washington, DC, USA, 1988. IEEE Computer Society.

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

[94] Samuel T. King, Joseph Tucek, Anthony Cozzie, Chris Grier, Weihang Jiang, and Yuanyuan Zhou. Designing and implementing malicious hardware. In *LEET'08: Proceedings of the 1st USENIX Workshop on Large-Scale Exploits and Emergent Threats*, pages 1–8, Berkeley, CA, USA, 2008. USENIX Association.

[95] Vladimir Kiriansky, Derek Bruening, and Saman P. Amarasinghe. Secure execution via program shepherding. In *Proceedings of the 11th USENIX Security Symposium*, pages 191–206, Berkeley, CA, USA, 2002. USENIX Association.

[96] Adam R. Klivans and Dieter van Melkebeek. Graph nonisomorphism has subexponential size proofs unless the polynomial-time hierarchy collapses. In *STOC '99: Proceedings of the 31st Annual ACM Symposium on Theory of Computing*, pages 659–667, New York, NY, USA, 1999. ACM.

[97] Adam R. Klivans and Dieter van Melkebeek. Graph nonisomorphism has subexponential size proofs unless the polynomial-time hierarchy collapses. *SIAM Journal on Computing*, 31(5):1501–1526, 2002.

[98] Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M. Frans Kaashoek, Eddie Kohler, and Robert Morris. Information flow control for standard os abstractions. In *SOSP '07: Proceedings of the 21st ACM SIGOPS Symposium on Operating Systems Principles*, pages 321–334, New York, NY, USA, 2007. ACM.

[99] Butler W. Lampson. A note on the confinement problem. *Communications of the ACM*, 16(10): 613–615, 1973.

[100] Michael LeMay and Jack Tan. Acoustic surveillance of physically unmodified PCs. In *SAM '06: Proceedings of the 2006 International Conference on Security and Management*, pages 328–334. CSREA Press, 2006.

[101] Dave Levin, John R. Douceur, Jacob R. Lorch, and Thomas Moscibroda. TrInc: small trusted hardware for large distributed systems. In *NSDI'09: Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, pages 1–14, Berkeley, CA, USA, 2009. USENIX Association.

[102] Jed Liu, Michael D. George, K. Vikram, Xin Qi, Lucas Waye, and Andrew C. Myers. Fabric: a platform for secure distributed computation and storage. In *SOSP '09: Proceedings of the 22nd ACM SIGOPS Symposium on Operating Systems Principles*, pages 321–334, New York, NY, USA, 2009. ACM.

[103] Carsten Lund, Lance Fortnow, Howard Karloff, and Nisan Noam. Algebraic methods for interactive proof systems. *Journal of the ACM*, 39(4):859–868, 1992.

[104] Nancy A. Lynch. *Distributed Algorithms.* Morgan Kaufmann Publishers, San Mateo, CA, 1996.

[105] Ralph C. Merkle. A certified digital signature. In *CRYPTO '89: Proceedings of the 9th Annual International Cryptology Conference*, pages 218–238, New York, NY, USA, 1989. Springer-Verlag New York, Inc.

[106] Silvio Micali. CS Proofs. In *FOCS '94: Proceedings of the 35th Annual IEEE Symposium on Foundations of Computer Science*, pages 436–453, Washington, DC, USA, 1994. IEEE Computer Society.

[107] Silvio Micali. Computationally sound proofs. *SIAM Journal on Computing*, 30(4):1253–1298, 2000.

[108] Peter Bro Miltersen and Variyam N. Vinodchandran. Derandomizing Arthur-Merlin games using hitting sets. In *FOCS '99: Proceedings of the 40th Annual IEEE Symposium on Foundations of Computer Science*, pages 71–80, Washington, DC, USA, 1999. IEEE Computer Society.

[109] Peter Bro Miltersen and Variyam N. Vinodchandran. Derandomizing Arthur-Merlin games using hitting sets. *Computational Complexity*, 14(3):256–279, 2005.

[110] Tal Moran and Gil Segev. David and Goliath commitments: UC computation for asymmetric parties using tamper-proof hardware. In *EUROCRYPT '08: Proceedings of the 27th Annual International Conference on Advances in Cryptology*, pages 527–544, Berlin, Heidelberg, 2008. Springer-Verlag.

[111] Andrew C. Myers and Barbara Liskov. A decentralized model for information flow control. In *SOSP '97: Proceedings of the 16th ACM SIGOPS Symposium on Operating Systems Principles*, pages 129–142, New York, NY, USA, 1997. ACM.

[112] Moni Naor and Omer Reingold. Number-theoretic constructions of efficient pseudo-random functions. In *FOCS '97: Proceedings of the 38th Annual IEEE Symposium on Foundations of Computer Science*, pages 458–467, Washington, DC, USA, 1997. IEEE Computer Society.

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

[114] George C. Necula. Proof-carrying code. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119, New York, NY, USA, 1997. ACM.

[115] George C. Necula and Peter Lee. Safe kernel extensions without run-time checking. *ACM SIGOPS Operating Systems Review*, 30(SI):229–243, 1996.

[116] George C. Necula and Peter Lee. Safe, untrusted agents using proof-carrying code. In *Mobile Agents and Security*, pages 61–91, London, UK, 1998. Springer-Verlag.

[117] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 89–100, New York, NY, USA, 2007. ACM.

[118] Ryan O'Donnell. A history of the PCP theorem. Available online at http://www.cs.washington.edu/education/courses/533/05au/pcp-history.pdf.

[119] Christos H. Papadimitriou. *Computational Complexity.* Addison-Wesley, Reading, MA, USA, 1994.

[120] Rafael Pass. On deniability in the common reference string and random oracle model. In *CRYPTO '03: Proceedings of the 23rd Annual International Cryptology Conference on Advances in Cryptology*, pages 316–337, London, UK, 2003. Springer-Verlag.

[121] A. Pavan, Alan L. Selman, Samik Sengupta, and Variyam N. Vinodchandran. Polylogarithmic-round interactive proofs for conp collapse the exponential hierarchy. *Theoretical Computer Science*, 385(1-3):167–178, 2007.

[122] Benjamin C. Pierce. *Types and Programming Languages.* MIT Press, Cambridge, MA, USA, 2002.

[123] Benjamin C. Pierce. *Advanced Topics in Types and Programming Languages.* The MIT Press, Cambridge, MA, USA, 2004.

[124] Jeff Plummer. A flexible and expandable architecture for computer games. Master's thesis, Arizona State University, 2004.

[125] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud! Exploring information leakage in third-party compute clouds. In *CCS '09: Proceedings of the 16th ACM Conference on Computer and Communications Security*, pages 199–212, New York, NY, USA, 2009. ACM.

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

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

[128] Jarrod A. Roy, Farinaz Koushanfar, and Igor L. Markov. Circuit CAD tools as a security threat. In *HOST '08: Proceedings of the 1st IEEE International Workshop on Hardware-Oriented Security and Trust*, pages 65–66, Washington, DC, USA, 2008. IEEE Computer Society.

[129] Adi Shamir. IP = PSPACE. *Journal of the ACM*, 39(4):869–877, 1992.

[130] Michael Sipser. *Introduction to the Theory of Computation.* Thomson Course Technology, Boston, MA, USA, 2005.

[131] G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. Secure program execution via dynamic information flow tracking. In *ASPLOS '04: Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 85–96, New York, NY, USA, 2004. ACM.

[132] Martin Tompa and Heather Woll. Random self-reducibility and zero knowledge interactive proofs of possession of information. In *FOCS '87: Proceedings of the 28th Annual IEEE Symposium on Foundations of Computer Science*, pages 472–482, Washington, DC, USA, 1987. IEEE Computer Society.

[133] Eran Tromer and Adi Shamir. Acoustic cryptanalysis, 2004. Eurocryt 2004 rump session; see http://people.csail.mit.edu/tromer/acoustic.

[134] Paul Valiant. Incrementally verifiable computation or proofs of knowledge imply time/space efficiency. In *TCC '08: Proceedings of the 5th Theory of Cryptography Conference on Theory of Cryptography*, pages 1–18, Berlin, Heidelberg, 2008. Springer-Verlag.

[135] Andrew C. Yao. Theory and application of trapdoor functions. In *FOCS '82: Proceedings of the 23rd Annual IEEE Symposium on Foundations of Computer Science*, pages 80–91, Washington, DC, USA, 1982. IEEE Computer Society.

[136] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in HiStar. In *OSDI '06: Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, pages 19–19, Berkeley, CA, USA, 2006. USENIX Association.