Proof-Carrying Data and Hearsay Arguments from Signature Cards

Alessandro Chiesa* Eran Tromer
Massachusetts Institute of Technology
Computer Science and Artificial Intelligence Laboratory
32 Vassar St., Cambridge, MA 02139, USA
{alexch,tromer}@csail.mit.edu

Abstract:
Design of secure systems can often be expressed as ensuring that some property is maintained at every step of a distributed computation among mutually-untrusting 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 circumnavigates the threat of faults and leakage by reasoning about properties of the output *data*, independently of the preceding *computation*. In PCD, the system designer prescribes the desired properties of the 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 specified 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.

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.

Keywords: secure distributed systems; computationally-sound proofs

1 Introduction

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 [11]), leaky (e.g., covert and side channels [48]) or adversarial (e.g., due to components from untrusted sources [12]).

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.

Our approach, *proof-carrying data* (PCD), 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 had the desired property.

---

[*] I dedicate this paper to my father Corrado Chiesa. He was a loving dad and wonderful person.

1.1 Motivation and Goals

Motivation. 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 in the framework of proof-carrying data.

- *Integrity.* Consider parties engaged in a distributed computation. Each party is supposed to transmit messages produced by executing some program on his own inputs and earlier messages received from other parties. Can we obtain evidence that the computation's final output is indeed the result of correctly following the prescribed program in the aforementioned process? For example, if the computation consists of a physics simulation (whether realistic or that of a virtual online world), can we obtain evidence that all parties have "obeyed the laws of physics"?
- *Information flow control.* Confidentiality and privacy are typically expressed as a negative condition forbidding certain effects. However, following the approach of information flow control (IFC) [27][55], 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 [38] and confidentiality [62]. Further details and examples appear in Section 5.

Goals. Generalizing the above, we can state our goal: a compiler that, given a protocol for 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 verifies 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 multiparty 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).

This implies, in particular, that *scalability* is preserved: if the original computation can be jointly conducted by numerous parties, then the compiler produces a secure distributed computation has the same property.

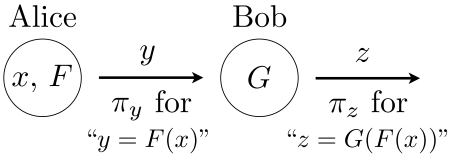

Figure 1: The "F and G" example.

1.2 Our Approach

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 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 $F(x)$, and 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 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.

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. See Figure 2 for a high-level diagram of this idea.[1] We 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.

PCD generalizes the "incrementally verifiable computation" of Valiant [68]. 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 [68], 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 [68], this is not implied by standard sound-

---

[1] Moreover, we obtain a proof-of-knowledge property (see [34, 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 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 [68] offers two constructions: one that assumes the existence of a cryptographic primitive that is nonstandard and arguably implausible [68, Theorem 1], and one whose overall security is conjectured directly without any reduction [68, Section 1.3 under "The Noninteractive CS Knowledge Assumption"]. The difficulty seems inherent; see Section 3.2. In our model, we attain provable security under standard generic cryptographic assumptions.

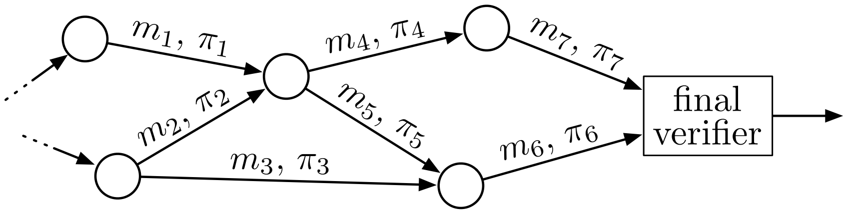

Figure 2: A distributed computation in which parties send messages m_i augmented by proof strings π_i.

ness: 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 [37][14]. Specifically, we use universal arguments [6] which (following [43] and computationally-sound proofs [53]) invoke the PCP theorem [5] to achieve compact proofs and efficient verification. However, such argument systems do not by themselves suffice: where they offer a strong proof-of-knowledge property [30][68], they do so by relying on random oracles, which precludes nesting of proofs since the underlying PCP system does not relativize [31][18]. Even in the restricted case of incrementally-verifiable computation [68], this difficulty precluded a satisfying proof of security.

We address this problem, both in general and for the special case of [68], 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.

1.3 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 [21] for a demonstration). Alternative realizations include obfuscation and multiparty computation; see Section 3.6 for further discussion.

Comparable assumptions have been used in previous works, as setup assumptions to achieve universally-composable functionality that is otherwise impossible [16]. In this context, Hofheinz et al. [39] 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 generator suffices — see Section 3.6), and to use slightly stronger signature schemes (see Section 2).

The more general result of Katz [42] 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 [54][17][25]. 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 [39].

For simplicity, we assume the following setup and trust model. A trusted party generates a signature key pair $(\mathsf{sk}, \mathsf{vk})$ and many signed-input-and-randomness tokens containing sk. Each party is

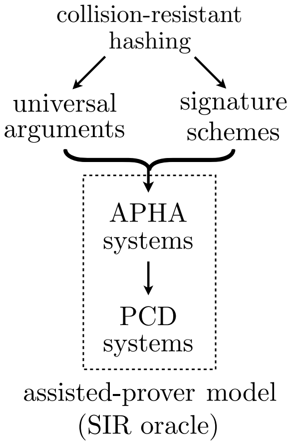

Figure 3: Collision-resistant hashing schemes imply public-coin constant-round universal arguments and secure signature schemes (with the additional property discussed in Section 2). From these two, we derive APHA systems, and then PCD systems.

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]

1.4 Our Results

In summary, we present the following results:

An argument system for hearsay. We define *assisted-prover hearsay-argument (APHA) systems*: non-interactive arguments for NP which 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 standard generic assumptions: collision-resistant hashing schemes and signature schemes (see Figure 3).

Distributed computations and proof-carrying data. 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 (see Figure 3).

Applications. We discuss a number of open problems in the security of real-world applications, where PCD offers a powerful solution approach by circumventing current difficulties.

1.5 Previous Approaches

Proof aggregation As discussed in Section 1.2, our aggregation-of-proofs approach is related to incrementally verifiable computation [68]. Both are built on top of efficient argument systems [37][14]: specifically, CS proofs [53] and universal arguments [6].

Metaproofs [65] also involve recursive aggregation of proofs, but using very different techniques; these seek statistical soundness rather than conciseness and efficient verification.

Signatures of knowledge [19] and their main application of delegatable anonymous credentials [8] yield proofs that are aggregatable, but at the expense of the proof size or the number of times aggregation (in their case, delegation) is allowed.

The problem of ensuring properties of a distributed computation has been previously studied by a variety of approaches.

Secure multiparty computation. Secure multiparty computation [36][9][20] considers the problem of correctly executing multiparty protocols in the presence of adversaries. Our approach follows that of [36] in that parties prove to each other, by cryptographic means, that they have been behaving correctly. The main differences are as follows. First, we address a more general setting, where the computation does not have to be known in advance to the parties. Second, [36][9][20] is unscalable in the sense of not preserving the communication graph of the original computation: even the simple "F and G" example of Section 1.2, would require *everyone on the Internet* to talk to each other. By contrast, in the PCD approach, parties

---

[3] Technically, this variant is realized by tweaking the PCD machine of Section 4.3 to verify the authority's signature on this vk.

perform only local computation to produce proof strings "on the fly", and attach them to outgoing data packets. Conversely, the constructions in this paper are not zero-knowledge.[4]

Distributed algorithms. Distributed algorithms [52] 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 [22] and TrInc [50].

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 [27][55], e.g., at the operating-system level [47][69]. Various invariants can be achieved by statically analyzing programs, and by programming language mechanisms such as type systems [3][26].

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.

Proof-carrying code. Proof-carrying code (PCC) [56] 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 [57], mobile agents [58] and compiled Java programs [23].

PCC and PCD thus address disjoint scenarios, by different techniques (see Table 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 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.

Dynamic analysis. Dynamic analysis monitors the properties of a program's execution at run time (e.g., [59][66][46]). 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 [51] 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.

1.6 Organization

In Section 2, we set up preliminaries. In Section 3, we define and construct hearsay-argument systems, and discuss the inherent difficulties involved as well as their resolution by assisted-prover model. In Section 4, we define proof-carrying data systems and construct them using the results of the previous sections. In Section 5, we discuss some potential applications. In Section 6, we conclude and suggest open problems.

2 Preliminaries

General notation. We let ϵ denote the empty string, and $\mathbb{N}$ the positive integers. For $n \in \mathbb{N}$,

---

[4] Zero-knowledge PCD systems are naturally defined, and necessary for some of our suggested applications. We do not see fundamental barriers to their existence. Their efficient construction is a subject of present investigation.

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

we denote by $[n]$ the set $\{1, \ldots, n\}$. We say that a function $\mu\colon \mathbb{N} \to [0,1]$ is negligible if, for every positive polynomial p, $\mu(n) < 1/p(n)$ for all sufficiently large n.

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 C, then $\langle C \rangle$ is its representation and $|C|$ is its size. For a probability distribution D, we denote by $y \leftarrow D$ drawing an element from D. Similarly, $y \leftarrow M(x)$ denotes the output of the machine or circuit M on input x; if M is a probabilistic machine then y is a random variable.

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

Universal arguments. We use universal arguments [6], a variant of CS proofs [53]. These are an efficient interactive argument system for proving membership into the universal set $S_{\mathcal{U}}$, defined as the set of all tuples $y = (M, x, t)$ for which there exists a witness w such that $M(x, w)$ accepts within t steps. We denote by $R_{\mathcal{U}}$ the witness relation of the universal set, and by $R_{\mathcal{U}}(y)$ the set of valid witnesses for a given instance y.

A universal argument consists of a prover P_{UA} and a verifier V_{UA}. For an instance $y = (M, x, t)$, universal arguments are *efficient* in the sense that the complexity of the verifier V_{UA} is polynomial in $|y|$, i.e., in $\mathsf{poly}(|M| + |x| + \log t)$. Moreover, the complexity of the prover P_{UA} is polynomial in $|M| + |x| + \mathsf{time}_M(x, w)$. Beyond the usual computational soundness required of an argument system, universal arguments also satisfy a *weak proof-of-knowledge property*. This property (defined in [6]) is essential in one of our proofs.

The universal argument construction of Barak and Goldreich [6] is a public-coin, 4-message protocol built from any collision-resistant hashing scheme ([35, Sec. 6.2.2.2]). The aforementioned efficiency comes from the use of a PCP system [5] for compressing proofs (following Micali [53] and Kilian [44]). While PCP constructions are notorious for being efficient only in the asymptotic sense, there are indications [10] that recent progress approaches practicality.

Signature schemes. We denote a signature scheme SIG by a triple $(G_{\mathsf{SIG}}, S_{\mathsf{SIG}}, V_{\mathsf{SIG}})$ consisting of the key generation, signing, and verification algorithms respectively. (See [35, Sec. 6.1].)

We use signature schemes that, beyond satisfying the standard property of security against chosen message attack, also satisfy the (independent) property of *security against signature-only forgery*: it is infeasible for a chosen-message attack to forge a hitherto-unseen signature that is valid for *any* message (the forger is not required to say which one).

It is simple to construct such a scheme: start from a signature scheme that is secure against chosen message attack, and modify its signature algorithm to append the message to the signature (and modify the verification algorithm accordingly). However, the parameters of our construction require concise signatures whose length is independent of the message (i.e., merely polynomial in the security parameter).

This can be achieved using a hash-then-sign approach. Starting with any *super-secure* signature scheme[5] $(G'_{\mathsf{SIG}}, S'_{\mathsf{SIG}}, V'_{\mathsf{SIG}})$ and a collision-

---

[5] A *super-secure signature scheme* (also called a *strongly*

resistant hashing scheme H_s ([35, Sec. 6.2.2.2]), we derive $(G_{\mathsf{SIG}}, S_{\mathsf{SIG}}, V_{\mathsf{SIG}})$ as follows. The key generation algorithm G_{SIG} invokes $(\mathsf{sk}', \mathsf{vk}') \leftarrow G'_{\mathsf{SIG}}$, and generates a public seed s for the hash function. To sign a message m, $S_{\mathsf{SIG}}((\mathsf{sk}', s), m)$ computes $h = H_s(m)$ and $\sigma' = S'_{\mathsf{SIG}}(\mathsf{sk}, h)$, and outputs $\sigma = (h, \sigma')$. To verify an alleged signature $\sigma = (h, \sigma')$ for m, $V_{\mathsf{SIG}}((\mathsf{vk}, s), m, \sigma)$ computes $\tilde{h} = H_s(m)$, verifies $h = \tilde{h}$ and runs $V'_{\mathsf{SIG}}(\mathsf{vk}', h, \sigma')$. Security is easily verified.

The super-secure signature schemes used above are known to exist if one-way functions exist [35, Theorem 6.5.2]. Moreover, there are efficient constructions based on the computational Diffie-Hellman assumption in bilinear groups [13], and generic transformations from regular signature schemes [40].

Therefore, in the rest of this paper, when we mention a signature scheme SIG, we shall assume that it is secure against chosen message attack and against signature-only forgery, and that it produces short signatures. This is without loss of generality, because our constructions already assume the existence of collision-resistant hashing schemes (e.g., to obtain universal arguments).

3 An Argument System for Hearsay

3.1 Overview

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 1.2).

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.*

Next, we explain why achieving the above properties involves a fundamental difficulty, and show how we resolve it by introducing an assisted prover. After that, we define the new argument system, then state which assumptions are sufficient to construct it, and then exhibit a construction for those assumptions. Finally, we discuss the realizability of an assisted prover.

3.2 Difficulties and Our Solution

In constructing an argument system that satisfies the properties discussed in Section 3.1, two opposing requirements arise:

1. **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 [44][53][6], or using a long PCP and homomorphic encryption [41]), all known efficient argument system constructions are based on the PCP theorem, and there is some evidence that this is inherent [63]. Since the PCP theorem does not relativize [31] (not even with respect to a random oracle [18]), 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.
2. **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 1.2, 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 [60][30] knowledge extractor with a tight success probability. The only known approach to

---

unforgeable signature scheme) is one where no new message-signature pair can be forged, even for messages that were already signed by the chosen-message oracle. See [35, Section 6.5.2].

such knowledge extraction is to force the prover to expose the witness in queries to an oracle.

Previous difficulties. The tension between the above two requirements arises in Valiant's work [68]. 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 needs oracle access. Therefore, one can at best conjecture (as done in [68]) 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, he exhibits a procedure that examines a prover's calls to the random oracle. However, once the random oracle has been instantiated, 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 [68] 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 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 1.4 and to be formally defined in Section 3.4), 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 [29]. Extraction of witnesses from oracle queries was used by Pass [60], Fischlin [30] and Valiant [68]. Our approach of using signatures to force oracle queries is similar in spirit to that of Chandran et al. [17].

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 3.6, we argue that the specific oracle that we choose, a signed-input-and-randomness oracle, is reasonable in practice.

3.3 Definition of APHA Systems

We define assisted-prover hearsay-argument (APHA) systems and discuss their properties. 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 $\kappa \in \mathbb{N}$, $G_{\mathsf{APHA}}(1^\kappa)$ outputs the description of a probabilistic[6] stateless oracle O to assist the prover, together with O's verification key vk;

---

[6] While our constructions are given for a probabilistic oracle, in Section 3.6 we discuss how to "derandomize" the oracle and make it deterministic.

- 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 $y \in S_{\mathcal{U}}$.

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 [34, 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 κ 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 κ, 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 $\mathsf{poly}(\kappa)$ such that, for any (possibly cheating) prover circuit $\widetilde{P}$ of size $\mathsf{poly}(\kappa)$ 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 $\langle \widetilde{P}(\mathsf{vk}), O\rangle$ 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. Formally:

Definition 1 (APHA System)**.** An *assisted-prover hearsay-argument system* with security parameter κ 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 $\kappa \in \mathbb{N}$, $(O, \mathsf{vk}) \in G_{\mathsf{APHA}}(1^\kappa)$, instance $y = (M, x, t)$, and proof string π,

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

  In particular, $|\pi| \le p(\kappa + |y|)$, i.e., the proof string length is $\mathsf{poly}(\kappa + |\langle M\rangle| + |x|) + \mathsf{polylog}(t)$.
- *Completeness via a relatively-efficient prover:* For every $\kappa \in \mathbb{N}$ and $(y, w) \in R_{\mathcal{U}}$,

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

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

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

  Note that $\mathsf{time}_M(x, w) \le t$.
- *List extraction:* There exists a *list extractor* circuit LE such that for every (possibly cheating) prover circuit $\widetilde{P}$ of size $\mathsf{poly}(\kappa)$, for all sufficiently large κ, if $\widetilde{P}$ convinces V_{APHA} then LE extracts a list containing a witness:

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

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

Proof of knowledge. The list-extraction property implies the standard proof-of-knowledge property, in which a knowledge extractor directly outputs a witness corresponding to an instance-proof pair that convinces the verifier (indeed, the knowledge extractor need only run the list extractor LE and locate the relevant triple in the list).

Adaptive soundness. As always, proof-of-knowledge implies soundness: if the prover convinces the verifier (with probability better than $1/p(\kappa)$) 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 choose the instance for which he wishes to produce a proof string. In particular, the instance may depend on the oracle and vk.

3.4 Construction of an APHA System

In the assisted-prover model, every party has black-box access to a certain functionality. In our case, the black-box functionality is defined as follows.[7]

Definition 2 (Signed-Input-and-Randomness functionality)**.** Let $\mathsf{SIG} = (G_{\mathsf{SIG}}, S_{\mathsf{SIG}}, V_{\mathsf{SIG}})$ be a signature scheme. Let $\kappa \in \mathbb{N}$ be the security parameter of SIG. Given sk_1 and sk_2 (generated by $G_{\mathsf{SIG}}(1^\kappa)$), 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. $r \leftarrow \{0,1\}^s$
2. If $s = 0$, $\sigma \leftarrow S_{\mathsf{SIG}}(\mathsf{sk}_1, (x, r))$
3. If $s > 0$, $\sigma \leftarrow S_{\mathsf{SIG}}(\mathsf{sk}_2, (x, r))$
4. Output $(r, \sigma)$
```

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

Theorem 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 [6, Theorem 1.1], and likewise for signatures schemes (see Section 2). We thus deduce the existence of APHA systems under a mild, generic assumption:

Corollary 3.2 (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 3.1 by constructing an APHA system, following the intuition presented in Section 3.2. The oracle generator G_{APHA} is constructed as follows.

Algorithm 1 (G_{APHA})**.** The oracle generator G_{APHA}, on input a security parameter $\kappa \in \mathbb{N}$, does the following:

```
1. $(\mathsf{sk}_1, \mathsf{vk}_1) \leftarrow G_{\mathsf{SIG}}(1^\kappa)$
2. $(\mathsf{sk}_2, \mathsf{vk}_2) \leftarrow G_{\mathsf{SIG}}(1^\kappa)$
3. $\mathsf{vk} \equiv (\mathsf{vk}_1, \mathsf{vk}_2)$
4. $\langle O \rangle \equiv \langle O_{\mathsf{sk}_1,\mathsf{sk}_2} \rangle$, where $O_{\mathsf{sk}_1,\mathsf{sk}_2}$ is a SIR oracle
5. Output $(\langle O \rangle, \mathsf{vk})$
```

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 knowledge extraction.) Let us define the subroutine AUG that maps y to y_{aug}:

Algorithm 2 (AUG)**.** Let $p(\kappa, m)$ be a polynomial that bounds the running time of V_{SIG} with security parameter κ on messages of length at most m. Fix a security parameter $\kappa \in \mathbb{N}$ and let $(O, \mathsf{vk}) \in G_{\mathsf{APHA}}(1^\kappa)$ 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 $(\mathsf{vk}_1, \sigma, y)$, does the following:

```
1. $x_{\mathsf{aug}} \equiv (\mathsf{vk}_1, \sigma, y)$
2. $t_{\mathsf{aug}} \equiv t + p(\kappa, m)$ where
   $m \equiv |((\text{“}\texttt{inst-wit}\text{”}, y, 1^t), \epsilon)|$
```

---

[7] The need for two separate keys arises for technical reasons of avoiding thorny dependencies across the transitions in the proof (Section 3.5).

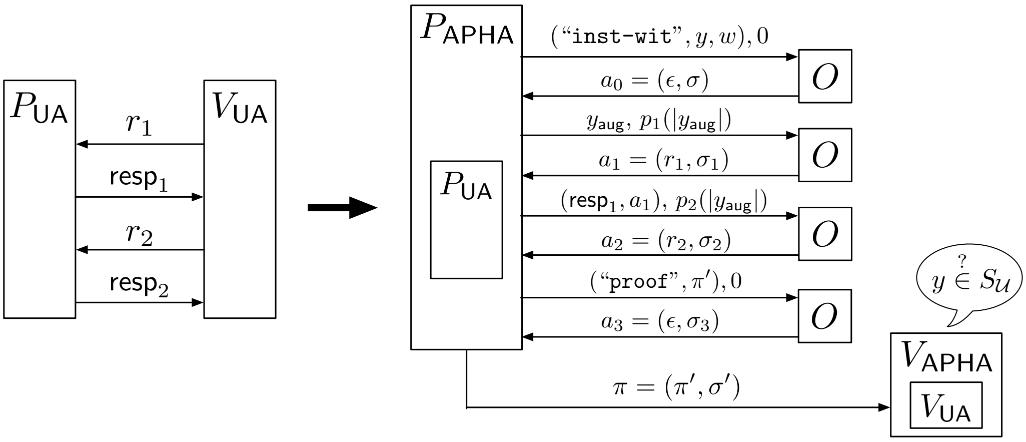

Figure 4: Diagram for the construction of P_{APHA}; recall that $\mathsf{tr} \equiv (a_1, \mathsf{resp}_1, a_2, \mathsf{resp}_2)$ and $\pi' \equiv (\sigma, \mathsf{tr})$.

3. Define M_{aug} to be the machine that, on input (x, w), works as follows
    (a) Let b_1 be the output of $V_{\mathsf{SIG}}(\mathsf{vk}_1, ((\text{“}\texttt{inst-wit}\text{”}, y, w), \epsilon), \sigma)$
    (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 $y_{\mathsf{aug}} \equiv (M_{\mathsf{aug}}, x_{\mathsf{aug}}, t_{\mathsf{aug}})$

We proceed to describe the construction of the prover P_{APHA} and verifier V_{APHA}. 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 $p_2(n)$.[8]

Algorithm 3 (P_{APHA})**.** Fix a security parameter κ and let $(O, \mathsf{vk}) \in G_{\mathsf{APHA}}(1^\kappa)$. 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 $q_0 \equiv ((\text{“}\texttt{inst-wit}\text{”}, y, w), 0)$ to obtain answer $a_0 = (\epsilon, \sigma)$.
2. *Compute the augmented instance.* Parse vk as $(\mathsf{vk}_1, \mathsf{vk}_2)$; compute $y_{\mathsf{aug}} \leftarrow \mathsf{AUG}(\mathsf{vk}_1, \sigma, y)$.
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 resp_1, the prover's first response.
5. *Simulate V_{UA}'s second message.* Call O with query $q_2 = \big((\mathsf{resp}_1, a_1), p_2(|y_{\mathsf{aug}}|)\big)$ 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 resp_2, the prover's second (and last) response.
7. *Package the signature and (part of) the transcript into a preliminary proof string.* Define $\pi' \equiv (\sigma, \mathsf{tr})$, where $\mathsf{tr} \equiv (a_1, \mathsf{resp}_1, a_2, \mathsf{resp}_2)$.
8. *Obtain a signature on the instance and preliminary proof.* Call O with query $q_3 \equiv ((\text{“}\texttt{proof}\text{”}, \pi'), 0)$ to obtain answer $a_3 = (\epsilon, \sigma')$.
9. *Output the signed proof.* Output $\pi \equiv (\pi', \sigma')$.

Algorithm 4 (V_{APHA})**.** Fix a security parameter κ and let $(O, \mathsf{vk}) \in G_{\mathsf{APHA}}(1^\kappa)$. Let $y = (M, x, t)$ be an instance and let π be an (alleged) proof string for “$y \in S_{\mathcal{U}}$”. The verifier $V_{\mathsf{APHA}}(\mathsf{vk}, y, \pi)$ does the following:

1. Parse vk as $(\mathsf{vk}_1, \mathsf{vk}_2)$; parse π as (π', σ'), where $\pi' \equiv (\sigma, \mathsf{tr})$, $\mathsf{tr} \equiv (a_1, \mathsf{resp}_1, a_2, \mathsf{resp}_2)$, $a_1 = (r_1, \sigma_1)$, and $a_2 = (r_2, \sigma_2)$.
2. *Verify that the signature is valid.* Check that $V_{\mathsf{SIG}}(\mathsf{vk}_1, ((\text{“}\texttt{proof}\text{”}, \pi'), 0), \sigma') = 1$.
3. *Compute the augmented instance.* $y_{\mathsf{aug}} \leftarrow \mathsf{AUG}(\mathsf{vk}_1, \sigma, y)$.

---

[8] For convenience, the construction here is specialized to the 2-round (4-message) universal arguments protocol of [6]. It naturally generalizes to any constant-round public-coin protocol.

4. *Verify that the transcript is consistent.* Check that:
    (a) $V_{\mathsf{SIG}}(\mathsf{vk}_2, (y_{\mathsf{aug}}, r_1), \sigma_1) = 1$ and $|r_1| = p_1(|y_{\mathsf{aug}}|)$
    (b) $V_{\mathsf{SIG}}(\mathsf{vk}_2, ((\mathsf{resp}_1, a_1), r_2), \sigma_1) = 1$ and $|r_2| = p_2(|y_{\mathsf{aug}}|)$.
5. *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 resp_1 and resp_2 as the prover's first and second messages, accepts.

3.5 Correctness of the APHA Construction

We complete the proof of Theorem 3.1 by showing that the above construction is indeed an APHA system. Efficient verifiability, as well as completeness via a relatively-efficient prover, follow easily from the construction.

The remaining property, list-extraction, is fulfilled by the following list extractor LE:

Algorithm 5 (LE)**.** Given vk and a prover-oracle interaction transcript $\langle \widetilde{P}(\mathsf{vk}), O\rangle$, $\mathsf{LE}(\langle \widetilde{P}(\mathsf{vk}), O\rangle)$ does the following:

```
1. $\mathsf{extlist} \leftarrow \mathsf{newLIST}()$
2. In the transcript $\langle \widetilde{P}(\mathsf{vk}), O\rangle$, let $(q_1, a_1), \ldots, (q_l, a_l)$ be the query-answer pairs in which the query is of the form $q_i = ((\text{"}\texttt{proof}\text{"}, \pi'_i), 0)$.
3. for $i \in [l]$ do:
    (a) Parse $\pi'_i$ as $(\sigma_i, \mathsf{tr}_i)$ and $a_i$ as $(\epsilon, \sigma'_i)$.
    (b) Find some $(q, a)$ in $\langle \widetilde{P}(\mathsf{vk}), O\rangle$ such that $a = (\epsilon, \sigma_i)$ and $q$ is of the form $q = ((\text{"}\texttt{inst-wit}\text{"}, y, w), 0)$. If none exists, output $\perp$ and abort.
    (c) Add $(y, (\pi'_i, \sigma'_i), w)$ to $\mathsf{extlist}$.
4. Output $\mathsf{extlist}$.
```

Claim 3.3. LE *fulfills the list-extraction property of* $(G_{\mathsf{APHA}}, P_{\mathsf{APHA}}, V_{\mathsf{APHA}})$.

The following is an overview of the proof structure; see [21] for details.

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 full-fledged list extraction). 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_1, V_1) 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_1 that outputs random strings upon request; the prover interacts with O_1, and then outputs an instance y and a proof string π_1 for the claim "$y \in S_{\mathcal{U}}$". When verifying the output of the prover, we allow V_1 to see all the query-answer pairs of the prover to O_1.

V_1 works by requiring a (possibly cheating) prover $\widetilde{P}_1$ 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 $\widetilde{P}_1$, in the right order, as answers from O_1.

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 cheating $\widetilde{P}_{\mathsf{UA}}$ 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_1.

Second step: stateless oracle. Starting from the pair of machines (P_1, V_1), we show how to construct a triple of machines (G_2, P_2, V_2) for which the weak PoK property still holds. This time, the prover has oracle access to a stateless probabilistic oracle O_2 generated by G_2, instead of the aforementioned stateful oracle O_1. On input x, O_2 outputs a random string r together with a signature on (x, r). When verifying the output of the prover, this time V_2 does not see the query-answer pairs of the prover to O_2. Instead, it verifies the signatures in the transcript provided by the prover, to be convinced that the queries were made to O_2.

That is, V_2 requires a (possibly cheating) prover $\widetilde{P}_2$ to produce a proof string that V_1 would have accepted, along with corresponding signatures that are valid under the verification key of O_2.

As before (but by a different technique), we show that whenever a prover $\widetilde{P}_2$ convinces V_2 on some instance y of its choice, $\widetilde{P}_2$ can be converted into a prover $\widetilde{P}_1$ that convinces V_1 on y, from which a witness for "$y \in S_{\mathcal{U}}$" can be extracted using the knowledge extractor E_1. We thus obtain a knowledge extractor E_2.

Third step: list extraction. Starting from (G_2, P_2, V_2), 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 2), 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 the (possibly cheating) prover $\widetilde{P}$ to produce a proof string that V_2 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_2 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 knowledge extractor.

Crucially, the knowledge extractor E_2 is not invoked by the APHA list extractor LE; rather, E_2 is used just in the proof of correctness of LE, in a reduction from failure of LE to forgeability of signatures.[9] 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. $\square$

3.6 Realizability of an Assisted Prover

Our construction attain APHA systems (and eventually PCD systems) assuming black-box access to 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 3.2): linear in the witness size $|w|$, and polynomial in the instance $|y|$ and security parameter κ. 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 which contains the whole t-step execution 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 [39][42][54][17][25][50]. 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 [28]. Similarly, the functionality can be embedded in cryptographic coprocessors, TPM chips, and general-purpose smartcard such as TEMs [24]. Alternatively, one may hope that this specific functionality can be obfuscated, either in the strict virtual-box-box sense [7] 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, applied to the input, whose seed is kept secret. 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

---

[9] This is similar in spirit to the extractor abort lemma of Chandran et al. [17].

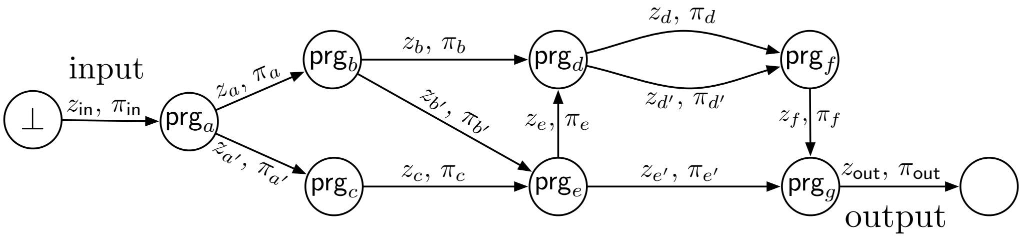

Figure 5: Example of an *augmented* distributed computation transcript. Programs are denoted by prg's, data by z's, and proof strings by π's. The corresponding (non-augmented) distributed computation transcript is with the proof strings omitted.

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 since even with a randomized oracle, adversaries can replay old answers.

4 Proof-Carrying Data Systems

We define and construct proof-carrying data (PCD) systems, which realize the framework of proof-carrying data. The following subsections are organized as follows: in Section 4.1, we define the notion of compliance for distributed computation; in Section 4.2, we define PCD systems and discuss their properties; in Section 4.3, we construct a PCD system, and, in Section 4.4, we sketch its correctness.

4.1 Compliance of Computation

We begin by specifying our notion of distributed computation.

Definition 3 (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 with labeled its program code, denoted $\mathsf{code}(v)$. 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 given as input to the program of v. Each source vertex has a single outgoing edge, carrying an input of the distributed computation; there are no programs at sources, so we set their label to $\perp$. The final output of the distributed computation is the data carried along edges going into sinks.

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.)

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

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 distinct corresponding vertices.

Next, we define what we mean for a distributed computation to be *compliant*, which is our notion of "correctness with respect to some 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 4 ($\mathbf{C}$-compliance)**.** A *compliance predicate* $\mathbf{C}$ is a polynomial-time computable predicate on strings. 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 $\mathbf{C}\big(\mathsf{data}(\mathsf{in}(v)), \mathsf{code}(v), \mathsf{data}(\mathsf{out}(v))\big) = 1$ (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.

4.2 Definition of PCD Systems

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

4.2.1 Structure of PCD systems

A PCD system consists of 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 κ, $G_{\mathsf{PCD}}(1^{\kappa})$ outputs the description of a probabilistic[10] 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} and π_{in_2} (see Figure 6). 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.[11]
- 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.

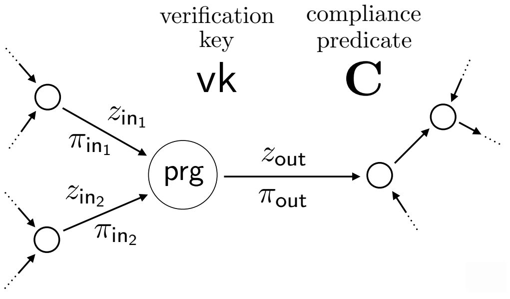

Figure 6: Computation of the new proof string π_{out} for the output data z_{out} using the PCD prover $P^{O}_{\mathsf{PCD}}(\mathsf{vk}, \mathbf{C}, z_{\mathsf{out}}, \mathsf{prg}, z_{\mathsf{in}_1}, \pi_{\mathsf{in}_1}, z_{\mathsf{in}_2}, \pi_{\mathsf{in}_2})$.

Using these algorithms, a distributed computation transcript is dynamically compiled into an *augmented* distributed computation transcript by adding a proof string to each edge (see Figure 5). The process of generating proof strings is defined inductively, by having each (internal) vertex v in DC use 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).

More precisely, focusing on a particular edge $(u, v) \in E$, we recursively define the process of *computing proof strings in* DC *up to* (u, v); this process generates an augmented transcript of DC up to (u, v). Let $\mathsf{DC}' = \mathsf{DC}|_{(u,v)}$ be the transcript of DC up to (u, v), and $\mathsf{proof}' \colon E' \to \{0,1\}^*$ another label on the edges of DC' that carries proof strings (in addition to the label data' that carries the data). 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,[12] let w_{in_1},w_{in_2} be the two parents of w, and let w_{out} be its the 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

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

---

[10] The oracle can be derandomized; see Section 3.6.

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

[12] 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.

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 $z = \mathsf{proof}(u, v)$.

4.2.2 Properties of PCD (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 soundness). These are adapted to the context of distributed computation transcripts.

First, proof strings generated by the prover should be *efficiently verifiable* by the verifier: V_{PCD} halts in time that is polynomial in the security parameter κ, the size of the description of C, the length of z, and the logarithm of the time it took to generate π. (Our parameters are even better; see the analysis in Section 4.2.3.)

Second, the 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 is used 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 process runs in time that is polynomial in the security parameter κ, the size of the description of $\mathbf{C}$ and the time it took to to verify $\mathbf{C}$-compliance at every node.

Third, *soundness* means that 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 $\mathsf{poly}(\kappa)$ can generate a convincing proof string π for z (except with non-negligible probability, over the randomness of the oracle and its verification key).

In order to preserve security for distributed computation that uses cryptographic functionality that is only computationally secure, we actually require a stronger property: *proof of knowledge*. A proof string π augmenting a piece of data z attests to the following. For any (possibly cheating) prover circuit $\widetilde{P}$ of size $\mathsf{poly}(\kappa)$, there exists a knowledge extractor E_{PCD} circuit 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. Also, $|E_{\mathsf{PCD}}|$ is polynomial in $|\widetilde{P}|$ and the security parameter κ.[13]

4.2.3 Properties of PCD (formal)

We proceed to capture the above intuition more formally. First, because provers and verifiers are concatenated in a recursive structure, in order to precisely quantify their complexity we need to define a recursive function over the transcript DC.

The recursive function that characterizes the complexity is as follows:

Definition 5 (Recursive Time up to an Edge)**.** Let p be a positive polynomial, κ 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 $T_p(\kappa, \mathbf{C}, \mathsf{DC}|_{(u,v)})$, where T_p is recursively defined as follows:

- If u is a source vertex,

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

- Otherwise,

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

The essential property of this recursive function is that the cost of past computation decays as an iterated logarithm at every aggregation step, and thus converges very quickly. Hence, the time it takes to generate a proof π_{out} is 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 verification time is logarithmic in that.

We can now state the definition of PCD systems.

Definition 6 (PCD System)**.** A *proof-carrying data system* with security parameter κ is a triple of polynomial-time machines $(G_{\mathsf{PCD}}, P_{\mathsf{PCD}}, V_{\mathsf{PCD}})$,

---

[13] 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.

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 $\kappa \in \mathbb{N}$, $(O, \mathsf{vk}) \in G_{\mathsf{PCD}}(1^{\kappa})$, compliance predicate $\mathbf{C}$, transcript DC, edge $(u, v) \in E$ with label $z = \mathsf{data}(u, v)$, and proof string π,

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

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

- *Completeness via a relatively-efficient prover:* Let A be the process of computing proof strings in DC up to (u, v), described above. For every $\kappa \in \mathbb{N}$, compliance predicate $\mathbf{C}$, $\mathbf{C}$-compliant transcript DC, and edge $(u, v) \in E$ with label $z = \mathsf{data}(u, v)$,

  $$\begin{aligned}\Pr\Big[&V_{\mathsf{PCD}}(\mathsf{vk}, \mathbf{C}, z, \pi) = 1 \;\Big|\; \\ &(O, \mathsf{vk}) \leftarrow G_{\mathsf{PCD}}(1^{\kappa})\ ; \\ &\pi \leftarrow A^{O}(\mathsf{vk}, \mathbf{C}, \mathsf{DC}|_{(u,v)})\Big] = 1\end{aligned}$$

  (where the probability is taken over the internal randomness of G_{PCD} and O).
  Furthermore, there exists a positive polynomial p such that for every $\kappa \in \mathbb{N}$, $(O, \mathsf{vk}) \in G_{\mathsf{PCD}}(1^{\kappa})$, $\mathbf{C}$-compliant computation DC, and edge $(v, w) \in E$ with label $z = \mathsf{data}(v, w)$,

  $$\begin{aligned}&\mathsf{time}_{A^{O}}(\mathsf{vk}, \mathbf{C}, \mathsf{DC}|_{(u,v)}) \le \\ &p\left(\kappa + |\langle \mathbf{C} \rangle| + |z| + T_p\left(\kappa, \mathbf{C}, \mathsf{DC}|_{(u,v)}\right)\right)\ .\end{aligned}$$

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

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

  (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:

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

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

4.3 Construction of a PCD System

We show the following:

Theorem 4.1 (PCD from APHA)**.** *PCD systems can be built from APHA systems (using the same oracle).*

Combining this with Corollary 3.2, we deduce the existence of PCD systems under mild standard assumptions:

Corollary 4.2 (Existence of PCD systems)**.** *Assuming the existence of collision-resistant hashing schemes, there exist PCD systems whose oracle is signed-input-and-randomness.*

Given any APHA system $(G_{\mathsf{APHA}}, P_{\mathsf{APHA}}, V_{\mathsf{APHA}})$, such as those of Section 3, we construct a PCD system $(G_{\mathsf{PCD}}, P_{\mathsf{PCD}}, V_{\mathsf{PCD}})$ as follows.

The oracle generator is the same (i.e., $G_{\mathsf{PCD}} = G_{\mathsf{APHA}}$). The PCD prover and verifier will invoke those of APHA on specially crafted statements "$(M_{\mathsf{PCD}}, x, t) \in S_{\mathcal{U}}$", where M_{PCD} is a fixed *PCD machine* (depending only on the compliance predicate $\mathbf{C}$ and the verification key vk) which specifies how to aggregate proof strings, check $\mathbf{C}$ locally and generate the new proof string.

Specifically, M_{PCD} 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 number of past aggregations, 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

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

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

Algorithm 6 (PCD Machine)**.** Fix $\kappa \in \mathbb{N}$ and let $(O, \mathsf{vk}) \in G_{\mathsf{PCD}}(1^\kappa)$. Let $\mathbf{C}$ be a compliance predicate $\mathbf{C}$, 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 $M_{\mathsf{PCD}}^{\mathsf{vk},\mathbf{C}}$, is defined as follows: $M_{\mathsf{PCD}}^{\mathsf{vk},\mathbf{C}}$, 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 $y_{\mathsf{in}_1} \equiv (M_{\mathsf{PCD}}^{\mathsf{vk},\mathbf{C}}, (z_{\mathsf{in}_1}, d_{\mathsf{in}_1}), t_{\mathsf{in}_1})$.
   (c) Verify that $V_{\mathsf{APHA}}(\mathsf{vk}, y_{\mathsf{in}_1}, \pi'_{\mathsf{in}_1}) = 1$, otherwise reject.
3. Repeat steps 1 and 2 for z_{in_2} and π_{in_2}.
4. Accept iff $\mathbf{C}((z_{\mathsf{in}_1}, z_{\mathsf{in}_2}), \mathsf{prg}, (z_{\mathsf{out}}))$ accepts.

The PCD prover and verifier are then constructed as follows.

Algorithm 7 (P_{PCD})**.** Fix $\kappa \in \mathbb{N}$ and let $(O, \mathsf{vk}) \in G_{\mathsf{PCD}}(1^\kappa)$. 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* $P_{\mathsf{PCD}}^{O}(\mathsf{vk}, \mathbf{C}, z_{\mathsf{out}}, \mathsf{prg}, z_{\mathsf{in}_1}, \pi_{\mathsf{in}_1}, z_{\mathsf{in}_2}, \pi_{\mathsf{in}_2})$ 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. Otherwise, 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. Otherwise, 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 $t \equiv t_{\mathbf{C}} + t_{\mathsf{in}_1} + t_{\mathsf{in}_2}$,
   $d_{\mathsf{out}} \equiv \max\{d_{\mathsf{in}_1}, d_{\mathsf{in}_2}\} + 1$,
   $y \equiv (M_{\mathsf{PCD}}^{\mathsf{vk},\mathbf{C}}, (z_{\mathsf{out}}, d_{\mathsf{out}}), t)$ and
   $w \equiv (\mathsf{prg}, z_{\mathsf{in}_1}, \pi_{\mathsf{in}_1}, z_{\mathsf{in}_2}, \pi_{\mathsf{in}_2})$.
5. Compute $\pi' \leftarrow P_{\mathsf{APHA}}^{O}(\mathsf{vk}, y, w)$.
6. Define $\pi \equiv (\pi', d, t)$.
7. Output π.

Algorithm 8 (V_{PCD})**.** Fix $\kappa \in \mathbb{N}$ and let $(O, \mathsf{vk}) \in G_{\mathsf{PCD}}(1^\kappa)$. 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 $y \equiv (M_{\mathsf{PCD}}^{\mathsf{vk},\mathbf{C}}, (z, d), t)$, and output $V_{\mathsf{APHA}}(\mathsf{vk}, y, \pi')$.

4.4 Correctness of the PCD Construction

To complete the proof of Theorem 4.1, there remains to show that the above construction is indeed a PCD system. Efficient verifiability, as well as completeness via a relatively-efficient prover, follow easily from the construction.

In the following, we sketch the proof of the PCD proof-of-knowledge property. For further details see the full version of this paper [21].

The PCD knowledge extractor E_{PCD} for a (cheating) prover $\widetilde{P}$, on input $(\mathsf{vk}, \mathbf{C}, z)$ 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 to record its oracle queries and answers, $\langle \widetilde{P}(\mathsf{vk}, \mathbf{C}, z), O \rangle$.
2. Apply the APHA list extractor LE to the recorded interaction $\langle \widetilde{P}(\mathsf{vk}), O \rangle$, to extract a list, extlist, of triples (y_i, π_i, w_i).
3. Apply an *offline reconstruction procedure* which outputs a transcript of the "past" distributed computation by looking only at extlist and (z, π) (see below).

All our work thus far was aimed at making such 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.

Offline reconstruction procedure. The procedure performs a depth-first traversal of the implicit history represented by extlist, starting from the root implied by (z, π). It maintains the following data structures:

- An *augmented distributed computation transcript* ADC, initially containing just the output edge.
- An *exploration stack*, denoted expstack, containing the set of edges of G 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 expstack. 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_i) in the extracted list extlist. (From the APHA list-extraction property, this succeeds, and moreover w_i is a valid witness with all but negligible probability.) If we have already seen the instance-witness pair (y_e, w_i) 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_i is a witness that uses the base case of the PCD machine, then v is a source vertex and we are done for his iteration. Otherwise v is a new internal vertex, and we add the edges leading to its (yet unknown) parents to expstack. The labels on ADC are updated accordingly.

5 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 a some examples of domains where we envision applicability. We stress that this is intended as a glimpse of things to come; full realizations, and evaluation of concrete practicality, exceed the present scope.

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 [5] and CS proofs [53], 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 original (long) proofs of all the lemmas, and only then then use (for example) CS proofs to compress them. Our results imply 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 [15] or perhaps via literal eavesdropping [49][4][67]).

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. 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 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 [11]. However, hardware and software components are often produced in far-away lands from parts of uncertain origin. This IT supply chain issue forms risks to users and organizations [1][12][45][64]. 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.

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 [61][33]. 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 naive (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 money 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., [59]) cannot securely span mutually untrusting platforms. Since tainting rules are easily expressed by a compliance predicate that observes the computation of the program, PCD can maintain tainting across a distributed computation.

Distributed type safety. Language-based type-safety mechanisms have tremendous expressive power, 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* [32] 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.

6 Conclusions and Open Problems

We envision proof-carrying data as a framework for achieving security properties in a nonconventional way, which circumvents many difficulties with current approaches. In PCD, faults and leakage are acknowledged as an expected occurrence, and rendered inconsequential by reasoning about properties of *data* which are independent of the preceding *computation*. The system designer prescribes the desired properties of the computation's output; proofs of these properties are attached to the data flowing through the system, and are mutually verified by the system's components.

This work shows explicit constructions of proof-carrying data, under standard assumptions, in the 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. A PCD system with the additional property of zero-knowledge [37][34, Chap. 4] would be useful in many applications. Of particular interest is surmounting the current inefficiency of the underlying argument systems and obtaining a fully practical realization.

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.

Acknowledgments

We are indebted to Ron Rivest for his insight and support during this investigation. 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, Greg Morrisett and Jeff Vaughan shared their perspective on applications of PCD in type safety. We thank Shafi Goldwasser, Frans Kaashoek, Nancy Lynch, Silvio Micali, Nickolai Zeldovich and the anonymous reviewers for valuable feedback.

This work was supported by NSF grant NSF-CNS-0808907 and AFRL grant FA8750-08-1-0088. Views and conclusions contained here are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either express or implied, of AFRL, NSF, the U.S. Government or any of its agencies.

References

[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] 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.

[5] 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.

[6] 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.

[7] 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.

[8] Mira Belenkiy, Jan Camenisch, Melissa Chase, Markulf Kohlweiss, Anna Lysyanskaya, and Hovav Shacham. Randomizable proofs and delegatable anonymous credentials. In *CRYPTO '09: Proceedings of the 29th Annual International Cryptology Conference on Advances in Cryptology*, pages 108–125, Berlin, Heidelberg, 2009. Springer-Verlag.

[9] 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.

[10] Arnab Bhattacharyya. Implementing probabilistically checkable proofs of proximity. Technical Report MIT-CSAIL-TR-2005-051, MIT, 2005. Available at http://dspace.mit.edu/handle/1721.1/30562.

[11] 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.

[12] 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.

[13] Dan Boneh, Emily Shen, and Brent Waters. Strongly unforgeable signatures based on computational diffie-hellman. In *PKC '06: Proceedings of the 9th International Workshop on Practice and Theory in Public Key Cryptography*, pages 229–240, London, UK, 2006. Springer-Verlag.

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

[15] 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.

[16] Ran Canetti and Marc Fischlin. Universally com-

posable commitments. In *CRYPTO '01: Proceedings of the 21st Annual International Cryptology Conference on Advances in Cryptology*, pages 19–40, London, UK, 2001. Springer-Verlag.

[17] 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.

[18] 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.

[19] Melissa Chase and Anna Lysyanskaya. On signatures of knowledge. In *CRYPTO '06: Proceedings of the 26th Annual International Cryptology Conference on Advances in Cryptology*, pages 78–96, London, UK, 2006. Springer-Verlag. Full version available at http://eprint.iacr.org/2006/184.

[20] 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.

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

[22] 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.

[23] 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.

[24] 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.

[25] 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.

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

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

[28] 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.

[29] 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.

[30] 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.

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

[32] 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.

[33] 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.

[34] Oded Goldreich. *Foundations of Cryptography: Volume 1, Basic Tools*. Cambridge University Press, New York, NY, USA, 2000.

[35] Oded Goldreich. *Foundations of Cryptography: Volume 2, Basic Applications*. Cambridge University Press, New York, NY, USA, 2004.

[36] 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.

[37] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof systems. *SIAM Journal on Computing*, 18 (1):186–208, 1989.

[38] 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.

[39] 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.

[40] Qiong Huang, Duncan S. Wong, and Yiming Zhao. Generic transformation to strongly unforgeable signatures. In *ACNS '07: Proceedings of the 5th International Conference on Applied Cryptography and Network Security*, pages 1–17, Berlin, Heidelberg, 2007. Springer-Verlag.

[41] 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.

[42] 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.

[43] Joe Kilian. Zero-knowledge with log-space verifiers. In *SFCS '88: Proceedings of the 29th Annual Symposium on Foundations of Computer Science*, pages 25–35, Washington, DC, USA, 1988. IEEE Computer Society.

[44] Joe Kilian. A note on efficient zero-knowledge proofs and arguments (extended abstract). In *STOC '92: Proceedings of the 24th Annual ACM Symposium on Theory of Computing*, pages 723–732, New York, NY, USA, 1992. ACM.

[45] 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.

[46] 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.

[47] 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.

[48] Butler W. Lampson. A note on the confinement problem. *Communications of the ACM*, 16(10): 613–615, 1973.

[49] 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.

[50] 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.

[51] 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.

[52] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, San Mateo, CA, 1996.

[53] Silvio Micali. Computationally sound proofs. *SIAM Journal on Computing*, 30(4):1253–1298, 2000.

[54] 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.

[55] 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.

[56] 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.

[57] George C. Necula and Peter Lee. Safe kernel extensions without run-time checking. *ACM SIGOPS Operating Systems Review*, 30(SI):229–243, 1996.

[58] George C. Necula and Peter Lee. Safe, un-

trusted agents using proof-carrying code. In *Mobile Agents and Security*, pages 61–91, London, UK, 1998. Springer-Verlag.

[59] 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.

[60] 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.

[61] Jeff Plummer. A flexible and expandable architecture for computer games. Master's thesis, Arizona State University, 2004.

[62] 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.

[63] 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.

[64] 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.

[65] Alfredo De Santis and Moti Yung. Cryptograpic applications of the non-interactive metaproof and many-prover systems. In *CRYPTO '90: Proceedings of the 10th Annual International Cryptology Conference on Advances in Cryptology*, pages 366–377, London, UK, 1991. Springer-Verlag.

[66] 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.

[67] Eran Tromer and Adi Shamir. Acoustic cryptanalysis, 2004. Eurocryt 2004 rump session; see http://people.csail.mit.edu/tromer/acoustic.

[68] 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.

[69] 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.