



Verifying computations without reexecuting them: from theoretical possibility to near practicality

Michael Walfish^{*†} and Andrew J. Blumberg[†]

^{*}New York University

[†]The University of Texas at Austin

Abstract

How can we trust results computed by a third party, or the integrity of data stored by such a party? This is a classic question in systems security, and it is particularly relevant in the context of cloud computing.

Various solutions have been proposed that make assumptions about the class of computations, the failure modes of the performing computer, etc. However, deep results in theoretical computer science—interactive proofs, probabilistically checkable proofs (PCPs) coupled with cryptographic commitments, etc.—tell us that a fully general solution exists that makes no assumptions about the third party: the local computer can check the correctness of a remotely executed computation by inspecting a proof returned by the third party. The rub is practicality: if implemented naively, the theory would be preposterously expensive (e.g., trillions of CPU-years or more to verify simple computations).

Over the last several years, a number of projects have reduced this theory to near-practice in the context of implemented systems; we call this field *proof-based verifiable computation*. The pace of progress has been rapid, and there have been many exciting developments. This paper covers the high-level problem, the theory that solves the problem in principle, the work required to bring this theory to near-practicality, the various projects in the area, and open questions. Many of these questions cut across multiple sub-disciplines of computer science: complexity theory, cryptography, systems, parallel programming, and programming languages.

[Note: this survey, which is about applying complexity-theoretic and cryptographic machinery, is written for a general computer science audience. However, we feel that it may be of interest to the ECCC community, so we are posting it here.]

1 INTRODUCTION

In this setup, a single reliable PC can monitor the operation of a herd of supercomputers working with possibly extremely powerful but unreliable software and untested hardware.

—Babai, Fortnow, Levin, and Szegedy, 1991 [6]

How *can* a single PC check a herd of supercomputers with unreliable software and untested hardware? This classic problem is particularly relevant today, as much computation is now outsourced: it is performed by machines that are rented, remote, or both.

For example, service providers (SPs) now offer storage, computation, managed desktops, and more;^{1,2,3} this general arrangement

is known as cloud computing, and it allows relatively weak devices (phones, tablets, laptops, PCs) to offload work (storage, image processing, video encoding, etc.) to banks of machines controlled by someone else. A close cousin of this arrangement is massive cluster computing; here, SPs and others perform computations on terabytes of data, distributed over tens of thousands of machines (a routine example is a computation whose input is the contents of the Web and whose output is the number of occurrences of every English word [16]). SPs themselves run many such computations daily, and in our “Big Data” era, they will only become more common. Another variation of *third-party computing*—as we term any situation in which one computer performs a task on behalf of another—is peer-to-peer computing, in which unknown peers provide computation and storage for each other.

The promise of third-party computation is enormous. A single graduate student (in biology, say) with a particularly intensive analysis (of genome data, say) can now rent a hundred computers for twelve hours at a total cost of less than \$200.⁴ Or consider that many companies now delegate their Web sites to SPs, who can automatically replicate applications to meet demand. Without SPs, these examples would require the company (or the graduate student’s advisor) to buy hundreds of physical machines or more when the demand spikes . . . and then sell them back the next day.

With this promise, however, comes risk: many things can (and do) go wrong in third-party computing scenarios. In cloud computing, one must worry about bugs, misconfigurations, operator error, natural disasters, malice, correlated manufacturing defects, and more [29]. This raises a central question: *How can we ever trust results computed by a third-party, or the integrity of data stored by such a party?*

A common answer is to replicate computations [1, 12, 13, 30]. However, replication assumes that failures are uncorrelated, which may not be a valid assumption. For one thing, the hardware and software platforms in cloud and cluster computing are often homogeneous. Moreover, replication cannot help if the failure is faulty logic. Another answer is auditing—checking the responses in a small sample—but this assumes that incorrect outputs, if they occur, are relatively frequent. Still other solutions involve trusted hardware [35] or attestation [33], but these mechanisms require a chain of trust that may not exist.

But what if the third party could return its results along with a proof that the results had been computed correctly? And what if the proof were inexpensive to check, compared to the cost of re-doing the computation? Then few assumptions would be needed about the kinds of faults that occurred in the cloud: either the proof would check or it wouldn’t. We call this vision *proof-based verifiable computation*, and the question now becomes: *Can this vision be realized for a wide class of computations?*

¹Amazon Web Services, <http://aws.amazon.com>

²Windows Azure, <http://www.windowsazure.com>

³<http://www.vmware.com/products/desktop-virtualization>

⁴Amazon Web Services, <http://aws.amazon.com>

In principle, deep results in theoretical computer science tell us that the answer is “yes”. From complexity theory, interactive proofs (IPs) [5, 22, 27, 40] and probabilistically checkable proofs (PCPs) [3, 4] (coupled with cryptographic commitments [26] in the context of arguments [10]) show how one entity (usually called a *verifier*) can be convinced by another (usually called a *prover*) of a given mathematical assertion. In the context of verifiable computation, the assertion is that a given computation was carried out correctly. In fact, one of the most acclaimed results in complexity theory, the PCP theorem [3, 4] (together with refinements [23]), implies that such an assertion can be checked by inspecting only three bits in a suitably encoded proof! And although the modern significance of the PCP theorem lies elsewhere, this line of research was motivated in part by checking computations efficiently; indeed, the paper quoted in the epigraph [6] was one of the seminal works that led to the PCP theorem.

However, for decades these approaches to verifiable computation were purely theoretical. Interactive protocols were prohibitive (exponential-time) for the prover and did not appear to save the verifier work. The proofs arising from the PCP theorem were so long and complicated that it would have taken trillions of years to generate and check them, and more storage bits than there are atoms in the universe to hold them.

Five years ago, proof-based verifiable computation reemerged in the theory literature [17, 21, 25]. Motivated by the rise of cloud computing, this work specifically emphasized the need to save the verifier work. Ishai et al. [25] explained how to use very simple PCP constructions and a novel cryptographic commitment to verify general-purpose computations; Goldwasser et al., in their Muggles work [21], used a complexity-theoretic interactive proof system that applied to computations expressed as certain kinds of circuits; and a couple of years later, Gentry’s breakthrough protocol for fully homomorphic encryption (FHE) [19, 20] led to a line of work (GGP) on *non-interactive* protocols for general-purpose computations [14, 17]. These developments were exciting, but, as with the earlier work, implementations were thought to be out of the question. So the theory continued to remain theory—until recently.

The last few years have seen a number of projects overturn the conventional wisdom about the hopeless impracticality of proof-based verifiable computation. These projects have aimed squarely at building real systems based on the theory mentioned above, specifically the PCP theorem and Muggles (FHE-based protocols still unfortunately seem too expensive). The improvements over the naive theoretical protocols are dramatic; it is not uncommon in this area to read about factor-of-a-trillion speedups. The projects take different approaches, but broadly speaking, they apply both refinements of the theory and systems techniques. Some of the projects include a full pipeline: a programmer specifies a computation in a high-level language, and then a compiler (a) transforms the computation to the formalism that the verification machinery uses and (b) outputs executables that implement the verifier and prover. This pipeline makes achieving verifiability no harder for the programmer than writing his or her code in the first place.

The goal of this paper is to survey this blossoming area of research. This is an exciting time for work on verifiable computation: while none of the works mentioned above is practical enough for its inventors to raise venture capital for a startup, they merit being referred to as “systems”. Moreover, many of the open problems cut across sub-disciplines of computer science: parallel computing, programming languages, systems engineering, complexity theory and cryptography. The pace of progress has been rapid, and we be-

lieve that real applications of these techniques to cloud computing will appear in the next few years.

We begin with an overview of the underlying technology, then describe progress in the area, and then articulate some of the open questions.

But before following this outline, we note that we are focused on solutions that provide integrity for general-purpose computations and that can in principle save work for the verifier. In particular, this means that we do not treat the exciting work on efficient implementations of secure multi-party protocols [24]. Similarly, we do not discuss the FHE-based approaches descending from GGP [17], since FHE is still impractical. This choice of scope is to make this paper manageable and because none of these algorithms is practical for large problems in the normal sense.

We also exclude a vast body of domain-specific solutions (e.g., Freivalds’s algorithm for matrix multiplication). Indeed, part of the appeal of the theoretical machinery that we turn to now is that it applies to all polynomial-time computations.

2 THE GENERAL APPROACH

We now review the problem that proof-based verifiable computation is solving, together with some of the theory that has been developed to solve it.

The problem statement, and some observations about it

One machine, a *verifier*, specifies a computation f and input x to a *prover*. The prover computes an output y and returns it to the verifier. If $y = f(x)$, then a correct prover should be able to convince the verifier of y ’s correctness, either by answering some questions or by providing a certificate of correctness. Otherwise, the verifier should reject y with high probability.

In any protocol that solves this problem, we desire three things. First, the protocol should be cheaper for the verifier than computing $f(x)$ locally. Second, we do not want to make any assumptions that the prover follows the protocol. Third, f should be general; later, we will have to make some compromises about the class of functions represented by f , but for now, f should be seen as encompassing all C programs that terminate.

Some observations about this setup are in order. To begin with, we are willing to accept some overhead for the prover, as we expect assurance to have a price. Another point is that our requirements contrast with common practices in computer security. In computer security, we often try to reason about what *incorrect* behavior looks like (think of spam detection, for instance). Here, however, we will not reason about every possible failure by the prover. Instead, the solutions will specify *correct* behavior and will ensure that anything other than this behavior is visible as such.

A framework for solving the problem in theory

We now describe a solution to the above problem. Our discussion will be somewhat informal.

The framework is depicted in Figure 1. Because *Boolean circuits* (networks of AND, OR, NOT gates) work naturally with the verification machinery, the first step is for the verifier and prover to transform the computation to such a circuit. This transformation is possible (in principle) because any of our computations f is naturally modeled by a Turing Machine (TM), and meanwhile a TM can be “unrolled” into a Boolean circuit that is not much larger than the number of steps in the computation. (This result is similar to the famous Cook-Levin theorem; see, for instance, the Arora-Barak

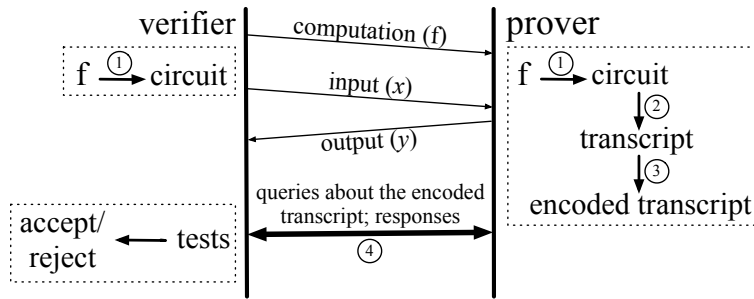


Figure 1—Framework in which a verifier can verify that, for a computation f and desired input x , the prover’s purported output y is correct. There are four steps. Step ①: the verifier and prover compile f , which is expressed in a high-level language (for example, the C programming language), into a Boolean circuit, \mathcal{C} . Step ②: the prover executes the computation, obtaining a transcript for the execution of \mathcal{C} on x . Step ③: the prover encodes the transcript, to make it suitable for efficient querying by the verifier. Step ④: the verifier probabilistically queries the encoded transcript; the structure of this step varies among the protocols (for example, in some of the works [8, 32], explicit queries are established before the protocol begins, and this step requires sending only the prover’s responses).

text [2, Thm 6.6, Thm 6.18].) Essentially, the circuit contains separate gates for each step in the computation, and for each of these steps, the relevant gates compute the TM’s transition function.

Thus, from now on, we will talk only about the circuit \mathcal{C} that represents our computation f (Figure 1, step 1). Consistent with the problem statement above, the verifier supplies the input x , and the prover executes the circuit \mathcal{C} on input x and claims the output is y . In performing this step, the prover is expected to obtain a *valid transcript* for $\{\mathcal{C}, x, y\}$ (Figure 1, step 2). A *transcript* is an assignment of values to the circuit wires; in a *valid transcript* for $\{\mathcal{C}, x, y\}$, the values assigned to the input wires are those of x , the intermediate values correspond to the correct operation of each gate in \mathcal{C} , and the values assigned to the output wires are y . Notice that if the claimed output is incorrect—that is, if $y \neq f(x)$ —then a valid transcript for $\{\mathcal{C}, x, y\}$ simply does not exist.

Therefore, if the prover could establish that a valid transcript exists for $\{\mathcal{C}, x, y\}$, this would convince the verifier of the correctness of the execution. Of course, there is a simple proof that a valid transcript exists: the transcript itself. However, the verifier can check the transcript only by examining all of it, which would be as much work as having executed f in the first place.

Instead, the prover will *encode* the transcript (Figure 1, step 3) into a much longer string, in such a way that different transcripts produce encodings that are different in almost all of their positions. Moreover, the verifier will be able to detect a transcript’s validity by inspecting a small number of randomly-chosen locations in the encoded string, and then applying efficient tests to the contents found at those locations. The powerful machinery of PCPs, for example, allows exactly this (see Sidebars 1 and 2; page 9).

However, we still have a problem. The verifier cannot get its hands on the entire encoded transcript; it’s longer—astronomically longer, in some cases—than the plain transcript, so reading in the whole thing would again require too much work from the verifier. Furthermore, we don’t want the prover to have to write out the whole encoded transcript: that would also be too much work, much of it wasteful, since the verifier looks at only small pieces of the encoding. And unfortunately, we cannot have the verifier just ask the prover point-blank what the encoding holds at particular locations, as the element of surprise is crucial for the protocols to work. That is, in these protocols, if the verifier’s queries are known in advance, then the prover can easily arrange its answers to fool the verifier.

As a result, the verifier has to issue its queries about the encoding carefully (Figure 1, step 4). The literature describes three tech-

niques for this purpose. We summarize them immediately below and discuss their relative merits in the next section.

- *Use the power of interaction.* One set of protocols proceeds in rounds: the verifier queries the prover about the contents of the encoding at a particular location, the prover responds, the verifier makes another query, the prover responds, etc. Just as a lawyer’s questions of a witness are intended to restrict the answers that the witness can give to the next question, until a lying witness is caught in a contradiction, the prover’s answers in each round about what the encoding holds limit the space of valid answers in the next round. This continues until the last round, at which point a prover that has answered perfidiously at any point—by answering based on an invalid transcript or by giving answers that are untethered to any transcript—simply has no valid answers. This approach relies on interactive proof protocols [5, 22, 27, 40], most notably the Muggles protocol [21], which was refined and implemented [15, 41–43].
- *Extract a commitment.* These protocols proceed in two rounds. The verifier first requires the prover to *commit* to the full contents of the encoded transcript. The commitment relies on standard cryptographic primitives, and we call the committed-to contents a *proof*. In the second round, the verifier generates a set of queries—locations in the proof that the verifier is interested in—and then asks the prover what values the proof contains at those locations; the prover is obligated to respond in a way that is consistent with the commitment. To generate queries and validate their responses, the verifier uses PCPs (they enable the kind of probabilistic checking that is described in Sidebar 2). This approach was outlined in theory by Kilian [26], building on the work of the PCP theorem [3, 4]. Later work by Ishai et al. [25] (IKO) gave a drastic simplification, in which the prover does not need to materialize the full proof. IKO led to a series of refinements, and implementation in a system [36–39, 43].
- *Hide the queries.* Instead of extracting a commitment and then revealing its queries, the verifier *pre-encrypts* its queries—as above, the queries describe locations where the verifier wants to inspect an eventual proof, and as above, these locations are chosen by PCP machinery—and sends this description to the prover in advance of their interaction. Then, during the verification phase, the verifier and prover use sophisticated cryptography to achieve the following: the prover answers the queries without being able to tell which locations in the proof are be-

setup costs	applicable computations				
	regular	straightline	pure	stateful, RAM	general loops
none (fast prover)	Thaler				
none	CMT				
low	Allspice				
medium	Pepper	Ginger	Zaatar	Pantry	
high			Pinocchio	Pantry	
very high				BCGTV	BCGTV

Figure 2—Design space of implemented systems for proof-based verifiable computation; there is a three-way trade-off among performance, expressiveness, and functionality. Higher in the figure means better performance (and less cryptography), and rightward means better expressiveness; higher-and-to-the-right is therefore better. The shaded systems achieve zero-knowledge, non-interactivity, etc.

ing queried, and the verifier recovers the prover’s answers. With the prover’s answers in hand, the verifier uses PCP machinery to check the answers, as in the commitment-based protocols. The approach is described in theory in [9, 18] and has been refined and implemented [8, 32].

The approaches summarized above depend on a richly varied set of tools, ranging from complexity-theoretic techniques that do not use more than high-school mathematics, to complexity-theoretic techniques based on abstract algebra, to advanced cryptography, to combinations thereof.

3 PROGRESS: IMPLEMENTED SYSTEMS

The three techniques described above are elegant and powerful, but as we indicated earlier, naive implementations would result in preposterous costs. The research projects that refined and implemented these techniques have applied theoretical innovations and serious systems work to achieve *near* practical performance. There has been a lot of recent activity in this area; in this section, we first explain the structure of the design space, then survey the various efforts, and finally explore their performance (in doing this, we will illustrate what “near practical” means).

We restrict our attention to *implemented systems* with published experimental results. By “system”, we mean code (preferably publicly released) that takes some kind of representation of a computation and produces executables for the verifier and the prover that run on stock hardware. Ideally, this code is a compiler toolchain, and the representation is a program in a high-level language.

The landscape

As depicted in Figure 2, we organize the design space in terms of a three-way trade-off among performance, expressiveness, and functionality. By performance, we mean whether the verifier has a setup cost and, if so, what that cost is; by expressiveness, we mean the class of the computations that the system can handle; and by functionality, we mean whether the works provide properties like zero-knowledge (allowing the prover a measure of privacy) and non-interactivity (setup costs amortize indefinitely). We now walk through the various systems.

CMT, Allspice, and Thaler. One line of work uses “the power of interaction”; it starts from Muggles [21], the interactive proof protocol mentioned in the previous two sections. CMT [15, 42] exploits an algebraic insight to save orders of magnitude, versus a naive implementation of Muggles.

For circuits to which CMT applies, performance is very good (in part because Muggles and CMT do not require cryptographic assumptions). In fact, recent refinements by Thaler [41] provide a prover that is optimal for certain classes of computations: the costs are only a constant factor (roughly 10, which is exceptionally low overhead in this research area) over the cost of executing the computation locally. Moreover, CMT can be applied in (and was originally designed for) a streaming model of computation, in which the verifier processes and discards input as it comes in.

However, CMT’s expressiveness is limited, as it imposes requirements on the circuit’s geometry: the circuit must have structurally similar parallel blocks. Of course, not all computations can be expressed in that form.

Allspice [43] partially relaxes these limitations, under the amortization model of the works that we describe next. (That is, Allspice requires a setup phase, but its cost is far cheaper than the setup phases in the works below.)

Pepper, Ginger, and Zaatar. Another line of work refines the “extract a commitment” technique (referred to in the theory literature as an “efficient argument” [10, 26]). Pepper [38] and Ginger [39] strengthened the commitment primitive of IKO and adapted the protocol to efficiently support “batching”, as described shortly. In addition, both systems represent computations not as circuits but as arithmetic constraints (essentially, a set of equations over a finite field); a valid transcript of the computation corresponds to a solution to the equations, enabling a more concise representation in many cases.

The aforementioned refinements drastically reduce costs for verifier and prover. The protocols leverage *batching*—multiple *instances* of the same computation, on different inputs—to amortize setup costs for the verifier. These setup costs are proportional to running one instance of the computation (and the constant of proportionality is high).

Pepper requires describing constraints manually. Ginger comes with a compiler that targets a larger class of computations. Still, Pepper and Ginger work only with straight-line computations that have repeated structure, and they require special-purpose PCP encodings.

Zaatar [37] removes these restrictions. Zaatar retains the structure of Pepper and Ginger but incorporates a new PCP, using GGPR’s algebraic representation of computations [18]; this PCP applies to pure computations (no side effects). As a result, Zaatar achieves the performance of Pepper and Ginger but on a much larger class of computations.

Pinocchio. Pinocchio [32] instantiates the approach of hiding the queries (Pinocchio is what is known as a *SNARG* and a *SNARK*). Pinocchio not only uses the PCP at the core of GGPR [18] (as does Zaatar) but also implements GGPR’s sophisticated cryptography.⁵

A key benefit to the cryptography is that because queries are hidden, they can be reused. The result is a protocol with minimal interaction (after a per-computation setup phase, the verifier sends only an instance’s input to the prover) and thus qualitatively better amortization behavior. Specifically, Pinocchio amortizes per-computation setup costs over all future instances of a given computation; by contrast, recall that Zaatar and Allspice amortize their per-computation costs only over a batch. The cryptography also buys zero-knowledge and public verifiability; the latter means that

⁵Our perspective here [9, 37] is that GGPR can be understood as sophisticated cryptography layered atop an ingeniously concise linear PCP [3, 25], and that the two components are separable.

anyone (not just a particular verifier) can check a proof, provided that the party who generated the queries is trusted.

The cryptography brings some expense relative to Zaatat (in the prover’s costs and the verifier’s setup costs), though heroic optimizations have resulted in surprisingly small overhead. Pinocchio’s compiler initiated the use of C syntax in this area, and its underlying computational model is the same as Ginger’s and Zaatat’s [37, 39].

Although the systems described above have made tremendous progress, they have done so within a computational model that is not reflective of real-world computations. First, these systems require loop bound bounds to be known at compile time. Second, they do not support indirect memory references, ruling out RAM and thus general-purpose programming. Third, these systems do not support external state: the verifier has to handle all inputs and outputs, ruling out MapReduce, queries against remote databases, etc. The next two projects address subsets of these issues.

BCGTV. In the context of a query-hiding approach, BCGTV [8] compiles programs to an innovative circuit representation [7]. This representation verifies the transition function of a simple processor (called TinyRAM). As a result, BCGTV’s approach is truly general-purpose: it supports all of C (including data-dependent looping and RAM), not just a subset. BCGTV combines its circuit representation with a “backend” that is much like Pinocchio’s (applying insights from [9, 18, 37]). And much like Pinocchio, BCGTV uses sophisticated cryptography (with the accompanying costs and benefits) and has per-computation setup costs that must amortize.

Unfortunately, BCGTV’s generality comes at a cost: its circuit representation is often orders of magnitude larger than the representation in Pinocchio and Zaatat, leading to orders of magnitude worse performance.

Pantry. Pantry [11] extends the computational model of Zaatat and Pinocchio, and works with both systems. Pantry provides a general-purpose approach to state, which yields a RAM abstraction, verifiable MapReduce, verifiable queries on remote databases, and—using Pinocchio’s zero-knowledge variant—computations that keep the prover’s state private. Currently, Pantry is the only system for verifiable computation that handles computations for which the verifier does not have all of the input. In Pantry’s technique—which was known to folklore, though not previously realized—the verifier’s explicit input includes a *digest* (for example, a Merkle hash) of the full input or state, and the prover is obliged to work over state that matches this digest.

Under Pantry, every operation against state compiles into the evaluation of a cryptographic hash function. Since hash functions are comparatively expensive to compute, a memory access is tens of thousands of times more costly than, for example, a basic arithmetic operation. However, compared to BCGTV, Pantry performs better for all but the most memory-intensive programs.

A brief look at performance

We want to understand performance differences among the systems. However, these distinctions are overshadowed by the general nature of costs in this area. Thus, these general costs will be our central concern. We will answer three questions:

1. *How do the verifier’s variable (per-instance) costs compare to local, native execution?* In some (but not all) cases, an alternative to verifiable outsourcing is local execution.
2. *What are the verifier’s setup costs, and how do they amortize?* Recall that in many of the systems, the setup costs are signifi-

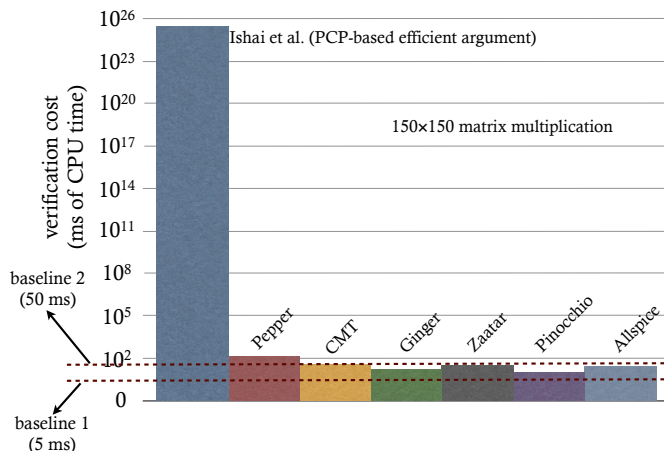


Figure 3—Per-instance verification costs, not including setup costs, for the various systems, applied to 150×150 matrix multiplication of 32-bit numbers. The first baseline, of 5 ms, is the CPU time to execute this computation natively. The second, of 50 ms, is the CPU time to execute this computation using a multiprecision library.

cant and are paid for only over multiple instances of the *same* computation (same logic, same input sizes).

3. What is the prover’s overhead?

We will focus only on CPU costs. On the one hand, this focus is conservative: it rules out cases when verifiable outsourcing is motivated. Specifically, if downloading the computation’s input incurs prohibitive network cost, then the hypothetical CPU cost of local execution is irrelevant as a baseline. On the other hand, CPU costs provide a good sense of the overall expense of the protocols. (For evaluations that take additional resources into account, see [11].)

In performing this evaluation, we include all systems in this area with published experimental results as of July 2013.⁶ The data that we present are from re-implementations, by members of our research group at UT, of the various systems, and match or exceed their published results. All experiments are run on the same hardware platform (Intel Xeon E5-2680 processor, 2.7GHz, 32GB RAM), with the prover on one machine and the verifier on another. We perform three runs per experiment; the experimental variation is minor, so we simply report the average. Our benchmarks are 150×150 matrix multiplication (of 32-bit quantities) and PAM clustering of 20 vectors, each of dimension 128.

Figure 3 depicts per-instance verification costs, for matrix multiplication, compared to two baselines. The first baseline is an optimized local implementation of the standard $O(m^3)$ algorithm, which costs 5 ms, and which beats all of the systems.⁷ The second baseline is an implementation of the algorithm using a multiprecision library; this models the case that local computation needs complete precision.

We evaluate setup costs by asking about the *cross-over point*: how many instances of a computation are required to amortize the setup cost in the sense that the verifier spends fewer CPU cycles on outsourcing versus executing locally? Figure 4 plots total cost lines

⁶In particular, we do not include Thaler [41], BCGTV [8], or Pantry [11]. Thaler does not affect the verifier but results in a vastly faster prover; earlier, we gave a sense of BCGTV’s comparative costs for the computations evaluated here; and Pantry’s innovations do not apply to these computations.

⁷The systems that report verification costs as beating local execution choose very expensive baselines for local computation [32, 37–39].

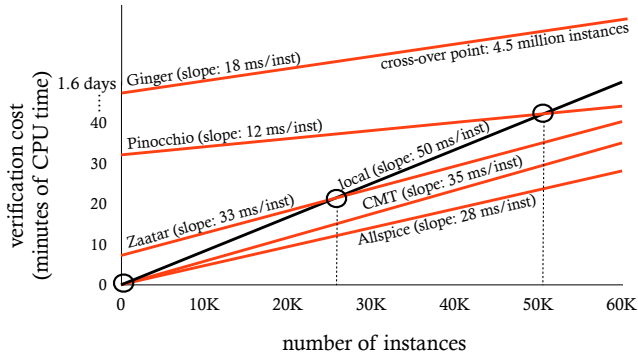


Figure 4—Total costs and cross-over points (extrapolated), for 150×150 matrix multiplication. The slope of each line is the per-instance cost (depicted in Figure 3); the y-intercepts are equal to the setup costs and equal 0 for local and CMT; and the cross-over point for a system is the point on the x-axis at which that system’s total cost line crosses the “local” line. The cross-over points for the general-purpose schemes (Zaatar and Pinocchio) are in the tens of thousands; the special-purpose approaches do far better, but they do not apply to all computations. While Zaatar’s cross-over point is somewhat better than Pinocchio’s, Pinocchio’s amortization regime is superior, as noted earlier: once Pinocchio passes its cross-over point, its setup cost for that computation is forever paid for, whereas Zaatar, Ginger, and Allspice incur the cost for each batch.

and cross-over points, versus the second baseline above.

To evaluate prover overhead, Figure 5 normalizes the prover’s cost to the baseline of native execution.

Summary and discussion. The verifier is practical if its computation is amenable to one of the less expensive (but more restricted) protocols, or if there are a large number of instances of the computation that will be run (on different inputs). And with computations over remote state, we are not obliged to make the verifier faster than local computation because it would be difficult—or impossible, if the remote state is private—for the verifier to perform the computation itself (such applications are evaluated elsewhere [11]).

The prover, of course, has terrible overhead: several orders of magnitude (though as noted previously, this still represents tremendous progress versus the prior costs). The prover’s practicality thus depends on your ability to construct appropriate scenarios. Maybe you’re sending Will Smith and Jeff Goldblum into space to save Earth; then you don’t care about costs. More prosaically, there is a scenario with an abundance of server CPU cycles, many instances of the same computation to verify, and remotely stored inputs: data-parallel cloud computing. Verifiable MapReduce [11] is therefore an encouraging application.

4 OPEN QUESTIONS AND NEXT STEPS

The biggest issue in this research area is performance, and the biggest performance issue is the prover’s overhead. Of course, the verifier’s costs are quantitatively higher than we would like. And qualitatively, we would ideally be able to eliminate the verifier’s setup phase in the context of a practical or near-practical protocol.

The computational model is also a critical area of focus. Only BCGTV [8] handles data-dependent loops, and only BCGTV and Pantry [11] handle computations that work with RAM. Unfortunately, BCGTV adds high overhead to the circuit representation for every operation in the given computation; Pantry, on the other hand, adds even higher overhead to its constraint representation but only to operations that interact with state. While improving the overhead

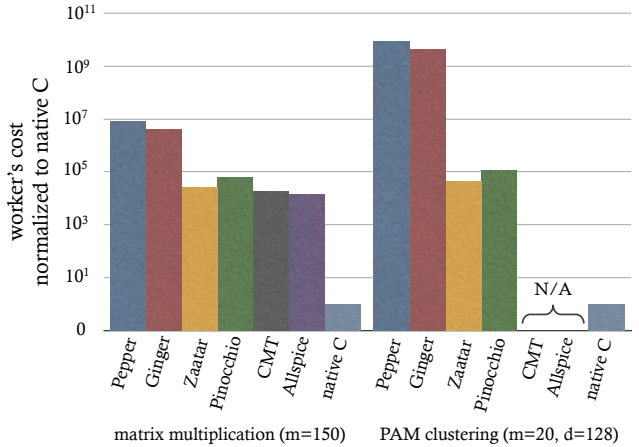


Figure 5—Prover overhead normalized to native execution cost for two computations. Prover overheads are enormous. Thaler’s approach [41] would do far better for matrix multiplication but (like CMT and Allspice) would not apply to PAM clustering.

of either representation would be worthwhile, a more general research direction is to move beyond the circuit and constraint model.

The above issues are likely to be addressed by refining theory. But there are also questions in systems and programming languages. For instance, can we develop programming languages that are well-tailored to the circuit or constraint formalism? We might also be able to co-design the language, computational model, and verification machinery: many of the protocols naturally work with parallel computations, and the current verification machinery is already amenable to a parallel implementation. Another systems question is to develop a realistic database application, which requires concurrency, access control, relational structures, etc. More generally, an important test for this area—so far unmet—is to run experiments at realistic scale.

Other research directions involve changing the model and goals. What is achievable, if we are willing to relax our requirement of “no assumptions about the prover other than cryptographic ones”? For instance, *multiprover* protocols can potentially reduce expense relative to single-prover protocols; the cost is the assumption of an additional (possibly misbehaving) prover that does not communicate with the first one. In fact, if we are willing to assume that at least one of the two provers computes correctly, a truly practical solution is available today [12]. Another interesting direction concerns privacy requirements. By leveraging Pinocchio [32], Pantry [11] has experimented with simple applications that hide the prover’s state from the verifier, but there is far more work to be done here, and many other notions of privacy that are worth providing.

5 REFLECTIONS AND PREDICTIONS

To finish up our coverage of this area, we reflect on its past and present, and make predictions about its future.

It is worth recalling that the intellectual foundations of this research area really had nothing to do with practice: the aim was to prove profound theoretical results about computational complexity, with no attention to concrete performance. For example, the PCP theorem is a landmark achievement of complexity theory, but if we were to implement the theory as proposed, generating the proofs, even for simple computations, would have taken longer than the age of the universe. In contrast, the projects described in this article have not only built systems from this theory but also performed

experimental evaluations that terminate in our lifetimes (in fact, before publication deadlines).

So that’s the encouraging news. The sobering news, of course, is that these systems are basically toys. Part of the reason we are willing to label them near-practical is painful experience with what the theory *used* to cost. Still, these systems are arguably useful in some scenarios. For example, in a high-assurance regime, we might be willing to pay a lot to know that a remotely deployed machine is executing correctly. As another example, if we are in the streaming context, the verifier may not have space to perform a computation, in which case we could use CMT [15] to check that the outputs are correct, particularly if we use Thaler’s recent refinements [41] to make the prover truly low overhead. Finally, data parallel cloud computations (like MapReduce jobs) perfectly match the regimes in which the general-purpose schemes perform well: abundant CPU cycles for the prover, and many instances of the same computation with different inputs.

Beyond this, we predict that in the near future, real systems will use proof-based verification for cloud computations. Indeed, the gap separating the performance of the current research prototypes and plausible deployment in the cloud is a few orders of magnitude—which is certainly daunting, but, given the current pace of improvement, it could conceivably be bridged in a few years.

More speculatively, if this research area succeeds in making the machinery *truly* low overhead, the effects will go far beyond verifying cloud computations: we will have new ways of building systems. In any situation in which one module performs a task for another, the delegating module will be able to check the answers. This could apply at the micro level (if the CPU had a way to check the results of the GPU, then this could potentially turn up hardware errors) and the macro level (distributed systems could be built under very different trust assumptions).

But even if none of the above comes to pass, there are exciting intellectual currents here. Across computer systems, we are starting to see a new style of work: reducing sophisticated cryptography and other achievements of theoretical computer science to practice [28, 31, 34, 44]. These developments are likely a product of our times: the preoccupation with strong security of various kinds, and the computers powerful enough to run algorithms that were previously “paper-only”. Whatever the cause, proof-based verifiable computation is an excellent example of this tendency: not only does it compose theoretical refinements with systems techniques, it also raises research questions in *other* sub-disciplines of Computer Science. This cross-pollination is the best news of all.

Acknowledgments

We thank Srinath Setty both for his help with this paper, including the experimental aspect, and for his deep influence on our understanding of this area. This paper has also benefited from many productive conversations with Justin Thaler, whose patient explanations have been most helpful. Comments from Boaz Barak, William Blumberg, Alexis Gallagher, Oded Goldreich, Riad Wahby, Eleanor Walfish, and Mark Walfish improved this draft. This work was supported by AFOSR grant FA9550-10-1-0073; NSF grants 1055057 and 1040083; a Sloan Fellowship; and an Intel Early Career Faculty Award.

REFERENCES

- [1] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer. SETI@home: An experiment in public-resource computing. *CACM*, 45(11):56–61, Nov. 2002.
- [2] S. Arora and B. Barak. *Computational Complexity: A modern approach*. Cambridge University Press, 2009.
- [3] S. Arora, C. Lund, R. Motwani, M. Sudan, and M. Szegedy. Proof verification and the hardness of approximation problems. *J. of the ACM*, 45(3):501–555, May 1998. (Prelim. version FOCS 1992).
- [4] S. Arora and S. Safra. Probabilistic checking of proofs: a new characterization of NP. *J. of the ACM*, 45(1):70–122, Jan. 1998. (Prelim. version FOCS 1992).
- [5] L. Babai. Trading group theory for randomness. In *STOC*, 1985.
- [6] L. Babai, L. Fortnow, L. A. Levin, and M. Szegedy. Checking computations in polylogarithmic time. In *STOC*, 1991.
- [7] E. Ben-Sasson, A. Chiesa, D. Genkin, and E. Tromer. Fast reductions from RAMs to delegatable succinct constraint satisfaction problems. In *ITCS*, Jan. 2013.
- [8] E. Ben-Sasson, A. Chiesa, D. Genkin, E. Tromer, and M. Virza. SNARKs for C: Verifying program executions succinctly and in zero knowledge. In *CRYPTO*, Aug. 2013.
- [9] N. Bitansky, A. Chiesa, Y. Ishai, R. Ostrovsky, and O. Paneth. Succinct non-interactive arguments via linear interactive proofs. In *IACR TCC*, Mar. 2013.
- [10] G. Brassard, D. Chaum, and C. Crépeau. Minimum disclosure proofs of knowledge. *J. of Comp. and Sys. Sciences*, 37(2):156–189, Oct. 1988.
- [11] B. Braun, A. Feldman, S. Setty, Z. Ren, A. J. Blumberg, and M. Walfish. Verifying computations with state. In *SOSP*, Nov. 2013.
- [12] R. Canetti, B. Riva, and G. Rothblum. Practical delegation of computation using multiple servers. In *ACM CCS*, 2011.
- [13] M. Castro and B. Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Trans. on Comp. Sys.*, 20(4):398–461, Nov. 2002.
- [14] K.-M. Chung, Y. Kalai, and S. Vadhan. Improved delegation of computation using fully homomorphic encryption. In *CRYPTO 2010*.
- [15] G. Cormode, M. Mitzenmacher, and J. Thaler. Practical verified computation with streaming interactive proofs. In *ITCS*, 2012.
- [16] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. In *OSDI*, 2004.
- [17] R. Gennaro, C. Gentry, and B. Parno. Non-interactive verifiable computing: Outsourcing computation to untrusted workers. In *CRYPTO*, 2010.
- [18] R. Gennaro, C. Gentry, B. Parno, and M. Raykova. Quadratic span programs and succinct NIZKs without PCPs. In *EUROCRYPT*, 2013. Cryptology ePrint 2012/215, Apr. 2012.
- [19] C. Gentry. *A fully homomorphic encryption scheme*. PhD thesis, Stanford University, 2009.
- [20] C. Gentry, S. Halevi, and N. Smart. Homomorphic evaluation of the AES circuit. In *CRYPTO*, 2012.
- [21] S. Goldwasser, Y. T. Kalai, and G. N. Rothblum. Delegating computation: Interactive proofs for muggles. In *STOC*, 2008.
- [22] S. Goldwasser, S. Micali, and C. Rackoff. The knowledge complexity of interactive proof systems. *SIAM J. on Comp.*, 18(1):186–208, 1989. (Prelim. version STOC 1985).
- [23] J. Håstad. Some optimal inapproximability results. *J. of the ACM*, 48(4):798–859, July 2001. (Prelim. version STOC 1997).
- [24] Y. Huang, D. Evans, J. Katz, and L. Malka. Faster secure two-party computation using garbled circuits. In *USENIX Security*, 2011.
- [25] Y. Ishai, E. Kushilevitz, and R. Ostrovsky. Efficient arguments without short PCPs. In *Conference on Computational Complexity (CCC)*, 2007.
- [26] J. Kilian. A note on efficient zero-knowledge proofs and arguments (extended abstract). In *STOC*, 1992.
- [27] C. Lund, L. Fortnow, H. J. Karloff, and N. Nisan. Algebraic methods for interactive proof systems. *J. of the ACM*, 39(4):859–868, Oct. 1992.
- [28] M. Maas, E. Love, E. Stefanov, M. Tiwari, E. Shi, K. Asanović, J. Kubiatowicz, and D. Song. PHANTOM: Practical oblivious computation in a secure processor. In *ACM CCS*, 2013.

- [29] P. Mahajan, S. Setty, S. Lee, A. Clement, L. Alvisi, M. Dahlin, and M. Walfish. Depot: Cloud storage with minimal trust. *ACM Trans. on Comp. Sys.*, 29(4), Dec. 2011.
- [30] D. Malkhi and M. Reiter. Byzantine quorum systems. *Distributed Computing*, 11(4):203–213, Oct. 1998. (Prelim. version STOC 1997).
- [31] A. Narayan and A. Haeberlen. DJoin: Differentially private join queries over distributed databases. In *OSDI*, 2012.
- [32] B. Parno, C. Gentry, J. Howell, and M. Raykova. Pinocchio: Nearly practical verifiable computation. In *IEEE Symposium on Security and Privacy*, May 2013.
- [33] B. Parno, J. M. McCune, and A. Perrig. *Bootstrapping Trust in Modern Computers*. Springer, 2011.
- [34] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan. CryptDB: Protecting confidentiality with encrypted query processing. In *SOSP*, 2011.
- [35] A.-R. Sadeghi, T. Schneider, and M. Winandy. Token-based cloud computing: secure outsourcing of data and arbitrary computations with lower latency. In *TRUST*, June 2010.
- [36] S. Setty, A. J. Blumberg, and M. Walfish. Toward practical and unconditional verification of remote computations. In *HotOS*, 2011.
- [37] S. Setty, B. Braun, V. Vu, A. J. Blumberg, B. Parno, and M. Walfish. Resolving the conflict between generality and plausibility in verified computation. In *EuroSys*, Apr. 2013.
- [38] S. Setty, R. McPherson, A. J. Blumberg, and M. Walfish. Making argument systems for outsourced computation practical (sometimes). In *NDSS*, 2012.
- [39] S. Setty, V. Vu, N. Panpalia, B. Braun, A. J. Blumberg, and M. Walfish. Taking proof-based verified computation a few steps closer to practicality. In *USENIX Security*, 2012.
- [40] A. Shamir. $IP = PSPACE$. *J. of the ACM*, 39(4):869–877, 1992.
- [41] J. Thaler. Time-optimal interactive proofs for circuit evaluation. In *CRYPTO*, Aug. 2013.
- [42] J. Thaler, M. Roberts, M. Mitzenmacher, and H. Pfister. Verifiable computation with massively parallel interactive proofs. In *USENIX HotCloud Workshop*, 2012.
- [43] V. Vu, S. Setty, A. J. Blumberg, and M. Walfish. A hybrid architecture for interactive verifiable computation. In *IEEE Symposium on Security and Privacy*, May 2013.
- [44] D. I. Wolinsky, H. Corrigan-Gibbs, B. Ford, and A. Johnson. Dissent in numbers: Making strong anonymity scale. In *OSDI*, 2012.

This sidebar will demonstrate a connection between *program execution* and *polynomials*. As a warmup, consider an AND gate, with two (binary) inputs, z_1, z_2 . One can represent its execution as a *function*:

$$\text{AND}(z_1, z_2) = z_1 \cdot z_2.$$

Here, the function AND behaves exactly as the gate would: it evaluates to 1 if z_1 and z_2 are both 1, and it evaluates to 0 in the other three cases. Now, consider this function of three variables:

$$\begin{aligned} f_{\text{AND}}(z_1, z_2, z_3) &= z_3 - \text{AND}(z_1, z_2) \\ &= z_3 - z_1 \cdot z_2 \end{aligned}$$

Observe that $f_{\text{AND}}(z_1, z_2, z_3)$ evaluates to 0 when, and only when, z_3 equals the AND of z_1 and z_2 . For example, $f_{\text{AND}}(1, 1, 1) = 0$ and $f_{\text{AND}}(0, 1, 0) = 0$ (both of these cases correspond to correct computation by an AND gate), but $f_{\text{AND}}(0, 1, 1) \neq 0$.

We can do the same thing with an OR gate:

$$f_{\text{OR}}(z_1, z_2, z_3) = z_3 - z_1 - z_2 + z_1 \cdot z_2.$$

For example, $f_{\text{OR}}(0, 0, 0) = 0$, $f_{\text{OR}}(1, 1, 1) = 0$, and $f_{\text{OR}}(0, 1, 0) \neq 0$. In all of these cases, the function is determining whether its third argument (z_3) does in fact represent the OR of its first two arguments (z_1 and z_2). Finally, we can do this with a NOT gate:

$$f_{\text{NOT}}(z_1, z_2) = 1 - z_1 + z_2.$$

The intent of this warmup is to communicate that *the correct execution of a gate can be encoded in whether some function evaluates to 0*. Such a function is known as an *arithmetization* of the gate.

Now, we extend the idea to a line $L(t)$ over a dummy variable, t :

$$L(t) = (z_3 - z_1 \cdot z_2) \cdot t.$$

This line is parameterized by z_1, z_2 , and z_3 : depending on their values, $L(t)$ becomes different lines. A crucial fact is that this line is the 0-line (that is, it covers the horizontal axis, or equivalently, evaluates to 0 for all values of t) if and only if z_3 is the AND of z_1 and z_2 . This is because the y-intercept of $L(t)$ is always 0, and the slope of $L(t)$ is given by the function f_{AND} . Indeed, if $(z_1, z_2, z_3) = (1, 1, 0)$, which corresponds to an incorrect computation of AND, then $L(t) = t$, a line that crosses the horizontal axis only once. On the other hand, if $(z_1, z_2, z_3) = (0, 1, 0)$, which corresponds to a correct computation of AND, then $L(t) = 0 \cdot t$, which is 0 for all values of t .

We can generalize this idea to higher order polynomials (a line is just a degree-1 polynomial). Consider the following degree-2 polynomial, or parabola, $Q(t)$ in the variable t :

$$Q(t) = [z_1 \cdot z_2 (1 - z_3) + z_3 (1 - z_1 \cdot z_2)] t^2 + (z_3 - z_1 \cdot z_2) \cdot t.$$

As with $L(t)$, the parabola $Q(t)$ is parameterized by z_1, z_2 , and z_3 : they determine the coefficients. And as with $L(t)$, this parabola is the 0 parabola (all coefficients are 0, causing the parabola to evaluate to 0 for all values of t) if and only if z_3 is the AND of z_1 and z_2 . For example, if $(z_1, z_2, z_3) = (1, 1, 0)$, which is an incorrect computation of AND, then $Q(t) = t^2 - t$, which crosses the horizontal axis only at $t=0$ and $t=1$. On the other hand, if $(z_1, z_2, z_3) = (0, 1, 0)$, which is a *correct* computation of AND, then $Q(t) = 0 \cdot t^2 + 0 \cdot t$, which of course is 0 for all values of t .

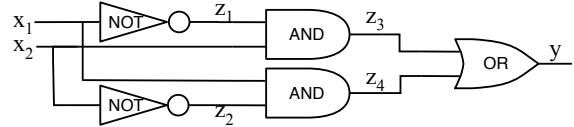
Summarizing, $L(t)$ (resp., $Q(t)$) is the 0-line (resp., 0-parabola) when and only when $z_3 = \text{AND}(z_1, z_2)$. This concept is powerful, for if there is an efficient way to check whether a polynomial is 0, then there is now an efficient check of whether a circuit was executed correctly (here, we have generalized to circuit from gate). And there are indeed such checks of polynomials, as described in Sidebar 2.

Sidebar 1: Encoding a circuit's execution in a polynomial.

This sidebar explains the idea behind a fast probabilistic checks of a transcript's validity. As noted in the text, computations are expressed as Boolean circuits. As an example, consider the following computation, where x_1 and x_2 are bits:

$$\text{if } (x_1 \neq x_2) \{ y = 1 \} \text{ else } \{ y = 0 \}$$

This computation could be represented by a single XOR gate; for illustration, we represent it in terms of AND, OR, NOT:

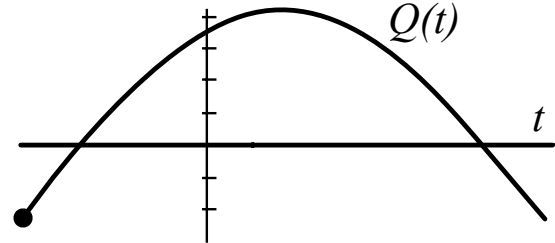


To establish the correctness of the output y , the prover must argue that it has a valid *transcript* (see text) for this circuit. The verifier, of course, cannot receive the transcript, since that would take as much time as (and more memory than) the computation.

Instead the two parties encode the computation as a polynomial $Q(t)$ over a dummy variable t . Sidebar 1 gives an example of this process for a single gate, but the idea generalizes to a full circuit. The result is a polynomial $Q(t)$ that evaluates to 0 for all t if and only if each gate's output in the transcript follows correctly from its inputs.

As with the single-gate case, the coefficients of $Q(t)$ are given by various combinations of $x_1, x_2, z_1, z_2, z_3, z_4, y$. Variables corresponding to inputs x_1, x_2 and output y are hard-coded, ensuring that the polynomial expresses a computation based on the correct inputs and the purported outputs.

Now, the verifier wants a probabilistic and efficient check that $Q(t)$ is 0 everywhere (see Sidebar 1). A key fact is that if a polynomial is *not* the zero polynomial, it has a bounded number of roots (consider a parabola: it crosses the horizontal axis a maximum of two times). For example, if we take $x_1=0, x_2=0, y=1$, which is an *incorrect* execution of the above circuit, then the corresponding polynomial might look like this:



and a polynomial corresponding to a *correct* program execution is simply a horizontal line on the axis.

The check, then, is this: the verifier chooses a random value for t (call it τ) from a pre-existing range (for example, integers between 0 and M , for some M), evaluates Q at τ (this is inexpensive; see below), and then accepts the computation as correct if $Q(\tau) = 0$ and rejects otherwise. This process occasionally produces errors since even a non-zero polynomial Q is zero sometimes (the idea here is a variant of "a stopped clock is right twice per day"), but this event happens rarely and is independent of the prover's actions.

The technical machinery in the various protocols enables the verifier to evaluate $Q(\tau)$ very efficiently (owing to commitment and other ideas discussed in the text). Moreover, neither party has to explicitly materialize the full polynomial Q .

The idea described above is at the heart of all of the approaches discussed in this article. One might wonder: what is the encoded transcript here? It's the polynomial Q , evaluated at every point in the range of possible values for τ : $\{Q(0), Q(1), \dots, Q(M)\}$. However, this encoding need not (and will not) ever be materialized in full.

Sidebar 2: Probabilistically checking a transcript's validity.