Electronic Colloquium on Computational Complexity, Revision 3 of Report No. 165 (2013)

# 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

> 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, Szegedy, 1991 [5]

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. As a result, relatively weak devices (phones, tablets, laptops, PCs) can run computations (storage, image processing, data analysis, video encoding, etc.) on banks of machines controlled by someone else.

This arrangement is known as cloud computing, and its promise is enormous. A lone graduate student with an intensive analysis of genome data can now rent a hundred computers for twelve hours for less than $200. And many companies now run their core computing tasks (Web sites, application logic, storage, etc.) on machines owned by SPs, which automatically replicate applications to meet demand. Without cloud computing, these examples would require buying hundreds of physical machines when demand spikes . . . and then selling them back the next day.

But with this promise comes risk. SPs are complex and large-scale, making it unlikely that execution is always correct. Moreover, SPs do not necessarily have strong incentives to ensure correctness. Finally, SPs are black boxes, so faults—which can include misconfigurations, corruption of data in storage or transit, hardware problems, malicious operation, and more [38]—are unlikely to be detectable. This raises a central question, which goes beyond cloud computing: *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 [16, 17, 39]. However, replication assumes that failures are uncorrelated, which may not be a valid assumption: the hardware and software platforms in cloud computing are often homogeneous. 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 [19, 49] or attestation [40, 45, 50], but these mechanisms require a chain of trust and assumptions that the hardware or a hypervisor works correctly.

But what if the third party returned its results along with a proof that the results were 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 can occur: either the proof would check or it wouldn't. We call this vision *proof-based verifiable computation*,[1] and the question now becomes: *Can this vision be realized for a wide class of computations?*

Deep results in complexity theory and cryptography tell us that in principle the answer is "yes". Probabilistic proof systems [28, 56]—which include interactive proofs (IPs) [4, 30, 37, 55], probabilistically checkable proofs (PCPs) [2, 3, 56], and argument systems [14] (PCPs coupled with cryptographic commitments [35])—consist of two parties: a *verifier* and a *prover*. In these protocols, the prover can efficiently convince the verifier of a mathematical assertion. In fact, the acclaimed PCP theorem [2, 3], together with refinements [31], implies that a verifier only has to check three randomly chosen bits in a suitably encoded proof!

Meanwhile, the claim "*this* program executed on *this* input produces *that* output" can be represented as a mathematical assertion of the necessary form. The only requirement is that the verifier knows the program, the input (or at least a digest, or fingerprint, of the input), and the purported output. And this requirement is met in many uses of outsourced computing; examples include MapReduce-style text processing, scientific computing and simulations, database queries, and Web request-response.[2] Indeed, although the modern significance of PCPs lies elsewhere, an original motivation was verifying the correctness of remotely executed computations: the paper quoted in our epigraph [5] 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 (despite asymptotic improvements [9, 23]) were so long and complicated that it would have taken thousands of years to generate and check them, and more storage bits than there are atoms in the universe.

But beginning around 2007, a number of theoretical works achieved results that were especially relevant to the problem of verifying cloud computations. Goldwasser et al., in their influential Muggles paper [29], refocused the theory community's attention on verifying outsourced computations, in the context of an interactive proof system that required only *polynomial* work from the prover, and that applied to computations expressed as certain kinds of circuits; Ishai et al. [34] proposed a novel cryptographic commitment to an entire linear function, and used this primitive to apply *simple* PCP constructions to verifying general-purpose outsourced computations; and a couple of years later, Gentry's breakthrough protocol for fully homomorphic encryption (FHE) [26] led to work (GGP) on

* This is a (slightly) expanded version of a forthcoming article for Communications of the ACM (CACM).

[1] The term *verifiable computation* was defined formally by Gennaro et al. [24], but our usage here is considerably broader.

[2] The condition does not hold for "proprietary" computations whose logic is concealed from the verifier. However, the theory can be adapted to this case too, as we discuss near the end of the article.

ISSN 1433-8092

*non-interactive* protocols for general-purpose computations [20, 24]. 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 aim squarely at building real systems based on the theory mentioned above, specifically the PCP theorem and Muggles (FHE-based protocols still seem too expensive). The improvements over naive theoretical protocols are dramatic; it is not uncommon 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 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. As a result, achieving verifiability is no harder for the programmer than writing the code in the first place.

The goal of this article is to survey this blossoming area of research. This is an exciting time for work on verifiable computation: while none of the works that we will discuss 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 believe that real applications of these techniques to cloud computing will appear in the next few years.

*Roadmap.* The next section gives an overview of the underlying technology. The following sections describe progress in this research area and articulate some of the open questions.

*A note about scope.* We focus on solutions that provide integrity to, and in principle can save work for, the verifier. Thus, we do not treat exciting work on efficient implementations of secure multiparty protocols (e.g., Huang et al. [32] and Kreuter et al. [36]). We also exclude FHE-based approaches based on GGP [24] (as noted above, these techniques seem too expensive) and the vast body of domain-specific solutions (surveyed elsewhere [44, 53, 59]).

## A PROBLEM AND THEORETICAL SOLUTIONS

**The problem statement, and some observations about it.** A *verifier* sends the specification of a computation $p$ (e.g., the text of a program) and input $x$ to a *prover*. The prover computes an output $y$ and returns it to the verifier. If $y = p(x)$, 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 provide some advantage to the verifier: either the protocol should be cheaper for the verifier than executing $p(x)$ locally, or else the protocol should handle computations $p$ that the verifier could not execute itself (e.g., operations on state private to the prover). Second, we do not want to make any assumptions that the prover follows the protocol. Third, $p$ should be general; later, we will have to make some compromises, but for now, $p$ should be seen as encompassing all C programs whose running time can be statically bounded given the input size.

Some reflections 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. Something else to note is that whereas some approaches to computer security attempt to reason about what *incorrect* behavior looks like (think of spam detection, for instance), we will specify *correct* behavior and ensure that anything other than this behavior is visible as such; this frees us from having to enumerate, or reason about, the possible failures of the prover.

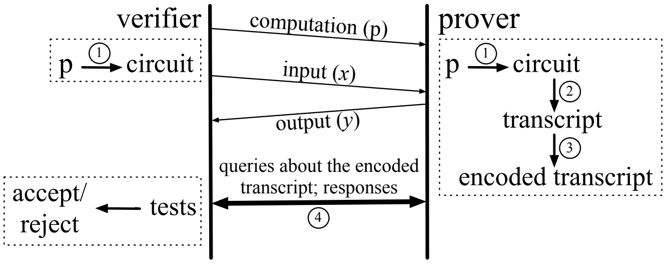


**Figure 1**—Framework in which a verifier can check that, for a computation $p$ and desired input $x$, the prover's purported output $y$ is correct. There are four steps. Step ①: the verifier and prover compile $p$, which is expressed in a high-level language (for example, the C programming language), 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 [7, 44], explicit queries are established before the protocol begins, and this step requires sending only the prover's responses).

Finally, one might wonder: how does our problem statement relate to NP-complete problems, which are easy to check but believed to be hard to solve? The answer is that the "check" of an NP solution requires the checking entity to do work polynomial in the length of the solution, whereas our verifier will do far less work than that! (Randomness makes this possible.) Another answer is that many computations (e.g., those that run in deterministic polynomial time) do not admit an asymmetric checking structure—unless one invokes the fascinating body of theory that we turn to now.

**A framework for solving the problem in theory** 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 because any of our computations $p$ 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 [46].

Thus, from now on, we will talk only about the circuit $\mathcal{C}$ that represents our computation $p$ (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 $y$.[3] 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

[3] The framework also handles circuits where the prover supplies some of the inputs and receives some of the outputs (enabling computations over remote state inaccessible to the verifier). However, the techniques for ensuring the validity of these auxiliary inputs and outputs are mostly beyond our scope (we will briefly mention them later). Thus, for simplicity we are treating $p$ as a pure computation.

the claimed output is incorrect—that is, if $y \neq p(x)$—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 $p$ in the first place.

Instead, the prover will *encode* the transcript (Figure 1, step 3) into a longer string. The encoding lets the verifier 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 therein. The machinery of PCPs, for example, allows exactly this (see Sidebars 1–3; pages 8–9).

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 separate techniques for this purpose. They draw on a richly varied set of tools from complexity theory and cryptography, and are summarized below. Their relative merits are discussed 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 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 [4, 30, 37, 55], most notably Muggles [29], which was refined and implemented [21, 57–59].
- *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 commited-to contents a *proof*. In the second round, the verifier generates 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 forced to respond consistent with the commitment. To generate queries and validate answers, the verifier uses PCPs (they enable probabilistic checking, as described in Sidebar 3). This approach was outlined in theory by Kilian [35], building on the PCP theorem [2, 3]. Later, Ishai et al. [34] (IKO) gave a drastic simplification, in which the prover can commit to a proof without materializing the whole thing. IKO led to a series of refinements, and implementation in a system [51–54, 59].

| | applicable computations | | | | |
|---|---|---|---|---|---|
| setup costs | regular | straight-line | pure | stateful | general loops |
| none (fast prover) | Thaler | | | | |
| none | CMT | | | | |
| low | | Allspice | | | |
| medium | Pepper | Ginger | Zaatar,<br>Pinocchio | Pantry | |
| high | | | | | TinyRAM |

**Figure 2—**Design space of implemented systems for proof-based verifiable computation; there is a three-way trade-off among cost, expressiveness, and functionality. Higher in the figure means lower cost, and rightward generally means better expressiveness. The shaded systems achieve non-interactivity, zero-knowledge, etc. (Pantry achieves these properties by leveraging Pinocchio.) Here, "regular" means structurally similar parallel blocks; "straight-line" means not many conditional statements; and "pure" means computations without side effects. The setup costs increase by factors of roughly 100 from low to medium to high.

- *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 prior to their interaction. Then, during the verification phase, powerful cryptography achieves the following: the prover answers the queries without being able to tell which locations in the proof are being queried, and the verifier recovers the prover's answers. The verifier then uses PCP machinery to check the answers, as in the commitment-based protocols. The approach was described in theory by Gennaro et al. [25] (see also Bitansky et al. [12]), and refined and implemented in two projects [7, 8, 44].

## PROGRESS: IMPLEMENTED SYSTEMS

The three techniques described above are elegant and powerful, but as noted, naive implementations would result in preposterous costs. The research projects that implemented these techniques have applied theoretical innovations and serious systems work to achieve *near* practical performance. We now explain the structure of the design space, survey the various efforts, and 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 publically 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 cost, expressiveness, and functionality. Here, cost mainly refers to *setup costs* for the verifier; as we will see, this cost is the verifier's largest expense, and affects whether a system meets the goal of saving the verifier work. (This setup cost also correlates with the prover's cost for most of the systems discussed.) By expressiveness, we mean the class of computations that the system can handle while providing a benefit to the verifier. By functionality, we mean whether the works provide properties like non-interactivity (setup costs amortize indefinitely), *zero knowledge* [28, 30] (the computation transcript is hidden from the verifier, giving the prover some privacy), and public verifiability (anyone, not just a particular verifier, can check a proof, provided that the party who generated the queries is trusted).

**CMT, Allspice, and Thaler.** One line of work uses "the power of interaction"; it starts from Muggles [29], the interactive proof protocol mentioned in the previous two sections. CMT [21, 58] exploits an algebraic insight to save orders of magnitude for the prover, versus a naive implementation of Muggles.

For circuits to which CMT applies, performance is very good, in part because Muggles and CMT do not use cryptographic operations. In fact, refinements by Thaler [57] provide a prover that is optimal for certain classes of computations: the costs are only a constant factor (10–100×, depending on choice of baseline) over executing the computation locally. Moreover, CMT applies in (and was originally designed for) a streaming model, in which the verifier processes and discards input as it comes in.

However, CMT's expressiveness is limited. First, 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. Second, the computation cannot use order comparisons (less-than, etc.).

Allspice [59] has CMT's low costs but achieves greater expressiveness (under the amortization model described next).

**Pepper, Ginger, and Zaatar.** Another line of work builds on the "extract a commitment" technique (called an "efficient argument" in the theory literature [14, 35]). Pepper [53] and Ginger [54] refine the protocol of IKO. To begin with, they represent computations as arithmetic constraints (i.e., a set of equations over a finite field); a solution to the constraints corresponds to a valid transcript of the computation. This representation is often far more concise than Boolean circuits (used by IKO and in the proof of the PCP theorem [2]) or arithmetic circuits (used by CMT). Pepper and Ginger also strengthen IKO's commitment primitive, explore low-overhead PCP encodings for certain computations, and apply a number of systems techniques (parallelization on distributed systems, etc.).

Pepper and Ginger dramatically reduce costs for the verifier and prover, compared to IKO. However, as in IKO, the verifier incurs setup costs. Both systems address this issue via amortization, reusing the setup work over a *batch*: multiple *instances* of the same computation, on different inputs, verified simultaneously.

Pepper requires describing constraints manually. Ginger has a compiler that targets a larger class of computations; also, the constraints can have auxiliary variables set by the prover, allowing for efficient representation of not-equal-to checks and order comparisons. Still, both handle only straight-line computations with repeated structure, and both require special-purpose PCP encodings.

Zaatar [52] composes the commitment protocol of Pepper and Ginger with a new linear PCP [2, 34]; this PCP adapts an ingenious algebraic encoding of computations from GGPR [25] (see below). The PCP applies to all pure computations; as a result, Zaatar achieves Ginger's performance but with far greater generality.

**Pinocchio.** Pinocchio [44] instantiates the technique of hiding the queries; Pinocchio is an implementation of GGPR (which is a *non-interactive* argument, or more technically, a SNARG [27] and a SNARK [10]). GGPR can be viewed as a probabilistically checkable encoding of computations that is akin to a PCP (this is the piece that Zaatar adapts) and, on top of that, a layer of sophisticated cryptography.[4] GGPR's encoding is substantially more concise than prior approaches, yielding major reductions in overhead.

The cryptography also provides many benefits. It hides the queries, which allows them to 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. GGPR's and Pinocchio's cryptography also yield zero knowledge and public verifiability.

Compared to Zaatar, Pinocchio brings some additional expense in the prover's costs and the verifier's setup costs, though heroic optimizations have resulted in surprisingly small overhead. Pinoccho's compiler initiated the use of C syntax in this area, and includes some program constructs not present in prior work. The underlying computational model (unrolled executions) is essentially the same as Ginger's and Zaatar's [52, 54].

Although the systems described above have made tremendous progress, they have done so within a programming model that is not reflective of real-world computations. First, these systems require loop bounds to be known at compile time. Second, they do not support indirect memory references scalably and efficiently, ruling out RAM and thus general-purpose programming. Third, the verifier must handle all inputs and outputs, a limitation that is at odds with common uses of the cloud. For example, it is unreasonable to insist that the verifier materialize the entire (massive) input to a MapReduce job. The next two projects address these issues.

**TinyRAM.** Using the query-hiding technique, TinyRAM [7, 8] compiles programs in C (not just a subset) to an innovative circuit representation [6]. This circuit consists of the unrolled execution of a general-purpose processor that can issue RAM operations, together with a compact routing network that (roughly speaking) verifies the consistency of RAM operations. Applying prior insights [12, 25, 52], TinyRAM combines these circuits with proof machinery (transcript encoding, queries, etc.) from Pinocchio.

TinyRAM's circuits have several appealing aspects. First, the technique for verifying memory consistency is efficient and elegant. Second, the computational model (a processor execution) provides a natural representation for high-level language features like data-dependent looping, control flow, and self-modifying code. Finally, as a result of its universality—all program executions with the same number of steps use the same circuit—TinyRAM has the best amortization behavior in the literature. Specifically, setup costs amortize over all future computations of a given length.

On the other hand, the generality brings a steep price: TinyRAM's circuits are orders of magnitude larger than Pinocchio's and Zaatar's for the same high-level programs. As a result, the verifier's setup work and the prover's costs are orders of magnitude higher, and TinyRAM is restricted to very short executions. At this scale, its advantage in expressiveness is largely theoretical. Nevertheless, TinyRAM's approach is very promising.

**Pantry.** Pantry [15] extends the computational model of Zaatar and Pinocchio, and works with both systems. Pantry provides a general-purpose approach to state, yielding 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. To date, Pantry is the only system to extensively use the capability of argument protocols (the "extract a commitment" and "hide the queries" techniques) to handle computations for which the verifier does not have all the input. In Pantry's approach—which instantiates folklore techniques [6, 10, 25, 33]—the verifier's explicit input includes a *digest* (for example, a Merkle hash [13, 41]) of the full input or state, and the prover is obliged to

[4] GGPR does not explicitly invoke PCPs, but the *key* in their work can be viewed as encrypted PCP queries [12, 52].

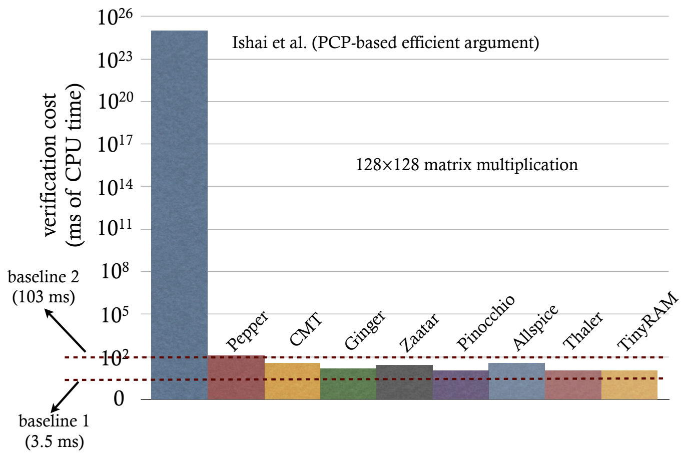


**Figure 3**—Per-instance verification costs, excluding setup costs, applied to 128×128 matrix multiplication of 64-bit numbers. (Data for Ishai et al. and TinyRAM are extrapolated.) The first baseline, of 3.5 ms, is the CPU time to execute natively, using floating-point arithmetic. The second, of 103 ms, uses multi-precision arithmetic.

work over state that matches this digest.

Under Pantry, every operation against state compiles into the evaluation of a cryptographic hash function. As a result, a memory access is tens of thousands of times more costly than a basic arithmetic operation. However, compared to TinyRAM, Pantry performs better for all but the most memory-intensive programs.[5]

## A brief look at performance

We will answer three questions:

1. *How do the verifier's variable (per-instance) costs compare to the baseline of local, native execution?* For some computations, this baseline is an alternative to verifiable outsourcing.
2. *What are the verifier's setup costs, and how do they amortize?* In many of the systems, setup costs are significant and are paid for only over multiple instances of the *same* circuit.
3. *What is the prover's overhead?*

We will focus only on CPU costs. On the one hand, this focus is conservative: verifiable outsourcing is motivated by more than CPU savings for the verifier. For example, if inputs are large or inaccessible, verifiable outsourcing saves network costs (the naive alternative is to download the inputs and locally execute); in this case, the CPU cost of local execution is irrelevant. 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 Braun et al. [15].)

The data that we present are from re-implementations (by members of our research group) of the various systems, and match or exceed their published results.[6] All experiments are run on the same hardware (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; experimental variation is minor, so we just report the average. For TinyRAM, we report extrapolated results since, as noted earlier, TinyRAM on current hardware is restricted to executions much smaller than the benchmarks that we use. These benchmarks are 128×128 matrix multiplication (of 64-bit quantities, with full precision arithmetic) and PAM clustering of 20 vectors, each of dimension 128.

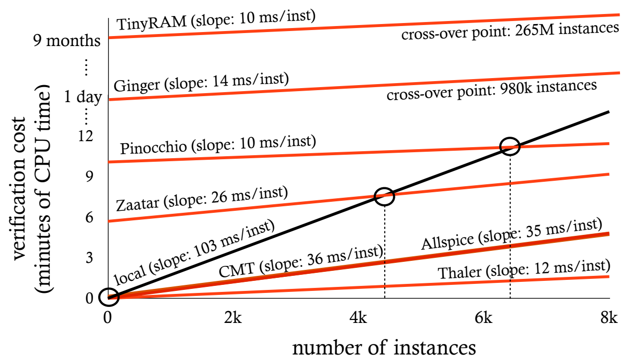


**Figure 4**—Total costs and cross-over points (extrapolated), for 128×128 matrix multiplication. The slope of each line is the per-instance cost (depicted in Figure 3); the y-intercepts are the setup costs and equal 0 for local, CMT, and Thaler. The cross-over point is the x-axis point at which a system's total cost line crosses its "local" line. The cross-over points for Zaatar and Pinocchio are in the thousands; the special-purpose approaches do far better but do not apply to all computations. Zaatar's cross-over point is somewhat better than Pinocchio's, which is much better than TinyRAM's; however, TinyRAM has the best amortization regime, followed by Pinocchio and then Zaatar (see text).

Figure 3 depicts per-instance verification costs, for matrix multiplication, compared to two baselines. The first is a native execution of the standard algorithm, implemented with floating-point operations; it costs 3.5 ms, and beats all of the systems at the given input size.[7] (At larger input sizes, the verifier would do better than native execution: the verifier's costs grow linearly in the input size, which is only $O(m^2)$; local execution is $O(m^3)$.) The second is an implementation of the algorithm using a multi-precision library; this baseline models a situation in which complete precision is required.

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 and cross-over points, versus the second baseline above. Notice that setup costs (y-intercepts) are in minutes, whereas per-instance execution and verification costs (slopes) are in milliseconds; thus, amortization requires thousands of instances or more.

To evaluate prover overhead, Figure 5 normalizes the prover's cost to the floating-point baseline.

**Summary and discussion.** Performance differences among the systems are overshadowed by the general nature of costs in this area. 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 that will be run (on different inputs). And when state is remote, the verifier doesn't need to be 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 [15]).

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

[5] Preliminary experiments suggest that the best option might be to combine the RAM abstraction from TinyRAM with Pantry's approach to execution verification and remote state.

[6] The exception is TinyRAM. Our reimplementation results in circuits (and hence verifier setup costs and prover costs) that are twice as large as the hand-optimized circuits that TinyRAM reports.

[7] Systems that report verification costs beating local execution choose very expensive baselines for local computation [44, 52–54].

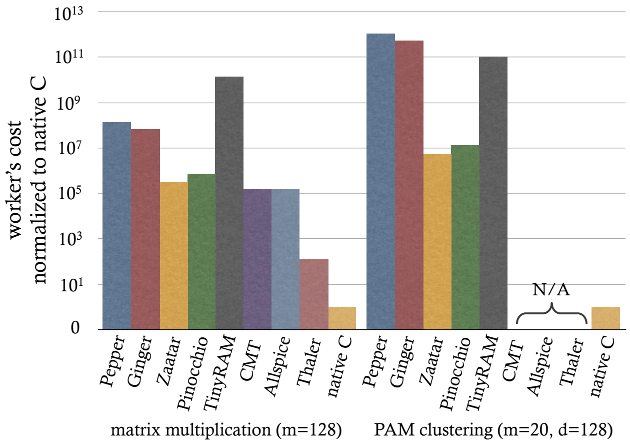


**Figure 5—**Prover overhead normalized to native execution cost for two computations. Prover overheads are generally enormous.

you're sending Will Smith and Jeff Goldblum into space to save Earth; then you care a lot more about correctness than costs (a calculus that applies to ordinary satellites too). 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 [15] is therefore an encouraging application.

## Open questions and next steps

The main issue in this area is performance, and the biggest problem is the prover's overhead. The verifier's costs are also quantitatively higher than ideal; qualitatively, we would ideally eliminate the verifier's setup phase while retaining expressivity (such schemes exist in theory [11] but have very high overheads, even in principle).

The computational model is also a critical area of focus. Only TinyRAM handles data-dependent loops, only Pantry handles remotely stored inputs, and only TinyRAM and Pantry handle computations that work with RAM. Unfortunately, TinyRAM 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 of either representation would be worthwhile, a more general research direction is to move beyond the circuit and constraint model.

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, relational structures, etc. More generally, an important test for this area—so far unmet—is to run experiments at realistic scale.

Another interesting area of investigation concerns privacy. By leveraging Pinocchio, Pantry has experimented with simple applications that hide the prover's state from the verifier, but there is more work to be done here and other notions of privacy that are worth providing. For example, we can provide verifiability while concealing the program that is executed (by composing techniques from Pantry, Pinocchio, and TinyRAM). A speculative application is to produce versions of Bitcoin in which transactions can be conducted anonymously, in contrast to the status quo [18, 22].

## Reflections and predictions

It is worth recalling that the intellectual foundations of this research area really had nothing to do with practice. 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 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 that we are willing to label them near-practical is painful experience with what the theory *used* to cost. (As a rough analogy, imagine a graduate student's delight in discovering hexadecimal machine code after years spent programming one-tape Turing machines.)

Still, these systems are arguably useful in some scenarios. In high-assurance contexts, we might be willing to pay a lot to know that a remotely deployed machine is executing correctly. In the streaming context, the verifier may not have space to compute locally, so we could use CMT [21] to check that the outputs are correct, in concert with Thaler's refinements [57] to make the prover truly inexpensive. 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.

Moreover, 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, might be bridged in a few years.

More speculatively, if the machinery becomes *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 could check the results of the GPU, this would expose 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 [32, 43, 47, 60]. These developments are likely a product of our times: the preoccupation with strong security of various kinds, and the computers powerful enough to run previously "paper-only" algorithms. 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 article, including the experimental aspect, and for his deep influence on our understanding of this area. This article has also benefited from many productive conversations with Justin Thaler, whose patient explanations have been most helpful. This draft was improved by experimental assistance from Riad Wahby; by detailed feedback from Alexis Gallagher and the anonymous CACM reviewers; and by comments from Boaz Barak, William Blumberg, Oded Goldreich, Yuval Ishai, Guy Rothblum, Riad Wahby, Eleanor Walfish, and Mark Walfish. 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] S. Arora and B. Barak. *Computational Complexity: A modern approach*. Cambridge University Press, 2009.

[2] 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).

[3] 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).

[4] L. Babai. Trading group theory for randomness. In *STOC*, pages 421–429, 1985.

[5] L. Babai, L. Fortnow, L. A. Levin, and M. Szegedy. Checking computations in polylogarithmic time. In *STOC*, 1991.

[6] E. Ben-Sasson, A. Chiesa, D. Genkin, and E. Tromer. Fast reductions from RAMs to delegatable succinct constraint satisfaction problems. In *ITCS*, pages 401–414, Jan. 2013.

[7] 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*, pages 90–108, Aug. 2013.

[8] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza. Succinct non-interactive zero knowledge for a von Neumann architecture. In *USENIX Security*, Aug. 2014.

[9] E. Ben-Sasson, O. Goldreich, P. Harsha, M. Sudan, and S. Vadhan. Robust PCPs of proximity, shorter PCPs and applications to coding. *SIAM J. on Comp.*, 36(4):889–974, Dec. 2006.

[10] N. Bitansky, R. Canetti, A. Chiesa, and E. Tromer. From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again. In *ITCS*, pages 326–349, Jan. 2012.

[11] N. Bitansky, R. Canetti, A. Chiesa, and E. Tromer. Recursive composition and bootstrapping for SNARKs and proof-carrying data. In *STOC*, pages 111–120, June 2013.

[12] N. Bitansky, A. Chiesa, Y. Ishai, R. Ostrovsky, and O. Paneth. Succinct non-interactive arguments via linear interactive proofs. In *IACR TCC*, pages 315–333, Mar. 2013.

[13] M. Blum, W. Evans, P. Gemmell, S. Kannan, and M. Naor. Checking the correctness of memories. In *FOCS*, pages 90–99, Oct. 1991.

[14] G. Brassard, D. Chaum, and C. Crépeau. Minimum disclosure proofs of knowledge. *J. of Comp. and Sys. Sciences*, 37(2):156–189, Oct. 1988.

[15] B. Braun, A. J. Feldman, Z. Ren, S. Setty, A. J. Blumberg, and M. Walfish. Verifying computations with state. In *SOSP*, pages 341–357, Nov. 2013.

[16] R. Canetti, B. Riva, and G. Rothblum. Practical delegation of computation using multiple servers. In *ACM CCS*, pages 445–454, 2011.

[17] M. Castro and B. Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Trans. on Comp. Sys.*, 20(4):398–461, Nov. 2002.

[18] A. Chiesa, C. Garman, E. Ben-Sasson, I. Miers, M. Green, M. Virza, and E. Tromer. Zerocash: Practical decentralized anonymous e-cash from Bitcoin. In *IEEE Symposium on Security and Privacy*, May 2014.

[19] A. Chiesa and E. Tromer. Proof-carrying data and hearsay arguments from signature cards. In *ICS*, 2010.

[20] K.-M. Chung, Y. Kalai, and S. Vadhan. Improved delegation of computation using fully homomorphic encryption. In *CRYPTO 2010*.

[21] G. Cormode, M. Mitzenmacher, and J. Thaler. Practical verified computation with streaming interactive proofs. In *ITCS*, pages 90–112, 2012.

[22] G. Danezis, C. Fournet, M. Kohlweiss, and B. Parno. Pinocchio coin: Building zerocoin from a succinct pairing-based proof system. In *Workshop on Language Support for Privacy-enhancing Technologies*, Nov. 2013.

[23] I. Dinur. The PCP theorem by gap amplification. *J. of the ACM*, 54(3):12:1–12:44, June 2007.

[24] R. Gennaro, C. Gentry, and B. Parno. Non-interactive verifiable computing: Outsourcing computation to untrusted workers. In *CRYPTO*, pages 465–482, 2010.

[25] R. Gennaro, C. Gentry, B. Parno, and M. Raykova. Quadratic span programs and succinct NIZKs without PCPs. In *EUROCRYPT*, pages 626–645, May 2013.

[26] C. Gentry. *A fully homomorphic encryption scheme*. PhD thesis, Stanford University, 2009.

[27] C. Gentry and D. Wichs. Separating succinct non-interactive arguments from all falsifiable assumptions. In *STOC*, pages 99–108, June 2011.

[28] O. Goldreich. Probabilistic proof systems – a primer. *Foundations and Trends in Theoretical Computer Science*, 3(1):1–91, 2007.

[29] S. Goldwasser, Y. T. Kalai, and G. N. Rothblum. Delegating computation: Interactive proofs for muggles. In *STOC*, pages 113–122, May 2008.

[30] S. Goldwasser, S. Micali, and C. Rackoff. The knowledge complexity of interactive proof systems. *SIAM J. on Comp.*, 18(1):186–208, 1989.

[31] J. Håstad. Some optimal inapproximability results. *J. of the ACM*, 48(4):798–859, July 2001. (Prelim. version STOC 1997).

[32] Y. Huang, D. Evans, J. Katz, and L. Malka. Faster secure two-party computation using garbled circuits. In *USENIX Security*, 2011.

[33] Y. Ishai. Personal communication, June 2012.

[34] Y. Ishai, E. Kushilevitz, and R. Ostrovsky. Efficient arguments without short PCPs. In *Conference on Computational Complexity (CCC)*, pages 278–291, 2007.

[35] J. Kilian. A note on efficient zero-knowledge proofs and arguments (extended abstract). In *STOC*, pages 723–732, 1992.

[36] B. Kreuter, a. shelat, and C.-H. Shen. Billion-gate secure computation with malicious adversaries. In *USENIX Security*, Aug. 2012.

[37] C. Lund, L. Fortnow, H. J. Karloff, and N. Nisan. Algebraic methods for interactive proof systems. *J. of the ACM*, 39(4):859–868, 1992.

[38] 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.

[39] D. Malkhi and M. Reiter. Byzantine quorum systems. *Distributed Computing*, 11(4):203–213, Oct. 1998. (Prelim. version STOC 1997).

[40] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: An execution infrastructure for TCB minimization. In *EuroSys*, 2008.

[41] R. C. Merkle. A digital signature based on a conventional encryption function. In *CRYPTO*, pages 369–378, Aug. 1987.

[42] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, Cambridge, United Kingdom, 1995.

[43] A. Narayan and A. Haeberlen. DJoin: Differentially private join queries over distributed databases. In *OSDI*, 2012.

[44] B. Parno, C. Gentry, J. Howell, and M. Raykova. Pinocchio: Nearly practical verifiable computation. In *IEEE Symposium on Security and Privacy*, pages 238–252, May 2013.

[45] B. Parno, J. M. McCune, and A. Perrig. *Bootstrapping Trust in Modern Computers*. Springer, 2011.

[46] N. Pippenger and M. J. Fischer. Relations among complexity measures. *J. of the ACM*, 26(2):361–381, Apr. 1979.

[47] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan. CryptDB: Protecting confidentiality with encrypted query processing. In *SOSP*, 2011.

[48] G. N. Rothblum. *Delegating Computation Reliably: Paradigms and Constructions*. PhD thesis, Massachusetts Institute of Technology, 2009.

[49] A.-R. Sadeghi, T. Schneider, and M. Winandy. Token-based cloud computing: secure outsourcing of data and arbitrary computations with lower latency. In *TRUST*, pages 417–429, 2010.

[50] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. van Doorn, and P. Khosla. Pioneer: Verifying integrity and guaranteeing execution of code on legacy platforms. In *SOSP*, pages 1–16, 2005.

[51] S. Setty, A. J. Blumberg, and M. Walfish. Toward practical and unconditional verification of remote computations. In *HotOS*, May 2011.

[52] 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*, pages 71–84, Apr. 2013.

[53] S. Setty, R. McPherson, A. J. Blumberg, and M. Walfish. Making argument systems for outsourced computation practical (sometimes). In *NDSS*, 2012.

[54] 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*, pages 253–268, Aug. 2012.

[55] A. Shamir. IP = PSPACE. *J. of the ACM*, 39(4), Oct. 1992.

[56] M. Sudan. Probabilistically checkable proofs. *Communications of the ACM*, 52(3):76–84, Mar. 2009.

[57] J. Thaler. Time-optimal interactive proofs for circuit evaluation. In *CRYPTO*, pages 71–89, Aug. 2013.

[58] J. Thaler, M. Roberts, M. Mitzenmacher, and H. Pfister. Verifiable computation with massively parallel interactive proofs. In *USENIX HotCloud Workshop*, June 2012.

[59] V. Vu, S. Setty, A. J. Blumberg, and M. Walfish. A hybrid architecture for interactive verifiable computation. In *IEEE Symposium on Security and Privacy*, pages 223–237, May 2013.

[60] 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*:

$$\textsc{and}(z_1, z_2) = z_1 \cdot z_2.$$

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:

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

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 $f_{\text{AND}}$. 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 $t$.

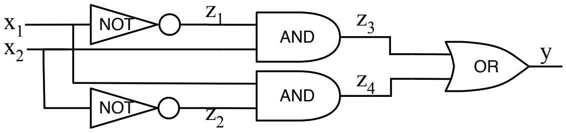
Summarizing, $L(t)$ (resp., $Q(t)$) is the 0-line (resp., 0-parabola) when and only when $z_3 = \textsc{and}(z_1, z_2)$. This concept is powerful, for if there is an efficient way to check whether a polynomial is 0, then there is now an efficient check of whether a circuit was executed correctly (here, 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 check 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:

```
if (x1 != x2) { y = 1 } else { y = 0 }
```

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

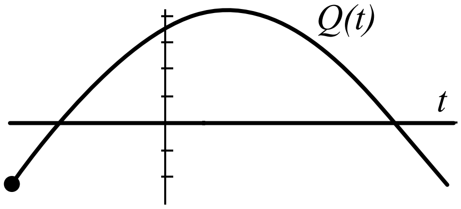


To establish the correctness of a purported output $y$ given inputs $x_1, x_2$, the prover must demonstrate to the verifier that it has a *valid transcript* (see text) for this circuit. A naive way to do this is for the prover to simply send the transcript to the verifier, and for the verifier to check it step-by-step. However, that would take as much time as 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.

Generalizing 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 output.

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 few 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* execution is simply a horizontal line on the axis.

The check, then, is the following. The verifier chooses a random value for $t$ (call it $\tau$) from a pre-existing range (for example, integers between 0 and $M$, for some $M$), and evaluates $Q$ at $\tau$. The verifier 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.

But how does the verifier actually evaluate $Q(\tau)$? Recall that our setup, for now, is that the prover sends a (possibly long) *encoded transcript* to the verifier. The next sidebar will explain what is in the encoded transcript, and how it allows the verifier to evaluate $Q(\tau)$.

**Sidebar** 2: Probabilistically checking a transcript's validity.

This sidebar answers the following question: how does the prover encode its transcript, and how does the verifier use this encoded transcript to evaluate $Q$ at a randomly chosen point? (The encoded transcript is known as a probabilistically checkable proof, or PCP. For the purposes of this sidebar, we assume that the prover sends the PCP to the verifier; in the main text, we will ultimately avoid this transmission, using commitment and other techniques.)

A naive solution is a protocol in which: the prover claims that it is sending $\{Q(0),\ldots,Q(M)\}$ to the verifier, the verifier chooses one of these values at random, and the verifier checks whether the randomly-chosen value is 0. However, this protocol does not work: even if there is no valid transcript, the prover could cause the verifier's "check" to always pass, by sending a string of zeroes.

Instead, the prover will encode the transcript, $z$, and the verifier will impose structure on this encoding; in this way, both parties together form the required polynomial $Q$. This process is detailed in the rest of this sidebar, which will be somewhat more technical than Sidebars 1 and 2. Nevertheless, we will be simplifying heavily; readers who want the full picture are referred to tutorials on PCP techniques [42, §7.8][1, Ch. 13, Ch. 22][9, Apdx A][53, §2.1][52, §2-3, Apdx. A] and interactive proof techniques [1, §8.3][48, §3.2-3.3].

As a warmup, observe that we can rewrite the polynomial $Q$ by regarding the "$z$" variables as unknowns. For example, the polynomial $Q(t)$ in Sidebar 1 can be written as:

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

An important fact is that for any circuit, the polynomial $Q$ that encodes its execution can be represented as a linear combination of the components of the transcript and pairwise products of components of the transcript. We will now state this fact using notation. Assume that there are $n$ circuit wires, labeled $(z_1,\ldots,z_n)$, and arranged as a vector $\vec{z}$. Further, let $\langle\vec{a},\vec{b}\rangle$ denote the dot product between two vectors, and let $\vec{a}\otimes\vec{b}$ denote a vector whose components are all pairs $a_ib_j$. Then we can write $Q$ as

$$Q(t,\vec{z}) = g_0(t) + \langle\vec{g}_1(t),\vec{z}\rangle + \langle\vec{g}_2(t),\vec{z}\otimes\vec{z}\rangle\,,$$

where $g_0$ is a function from $t$ to scalars, and $\vec{g}_1$ and $\vec{g}_2$ are functions from $t$ to vectors.

Now, what if the verifier had a table that contained $\langle\vec{b},\vec{z}\rangle$ for all vectors $\vec{b}$ (in a finite vector space), and likewise a table that contained $\langle\vec{c},\vec{z}\otimes\vec{z}\rangle$ for all vectors $\vec{c}$? Then, the verifier could evaluate $Q(\tau)$ by inspecting the tables in only one location each. Specifically, the verifier would randomly choose $\tau$; then compute $g_0(\tau)$, $\vec{g}_1(\tau)$, and $\vec{g}_2(\tau)$; then use the two tables to look up $\langle\vec{g}_1(\tau),\vec{z}\rangle$ and $\langle\vec{g}_2(\tau),\vec{z}\otimes\vec{z}\rangle$; and add these values to $g_0(\tau)$. If the tables were produced correctly, this final sum (of scalars) will yield $Q(\tau,\vec{z})$.

However, a few issues remain. The verifier cannot know that it actually received tables of the correct form, or that the tables are consistent with each other. So the verifier performs additional spot checks; the rough idea is that if the tables deviate too heavily from the correct form, then the spot checks will pick up the divergence with high probability (and if the tables deviate from the correct form but only mildly, the verifier's checks continue to work).

At this point, we have answered the question at the beginning of this sidebar: the correct encoding of a valid transcript $z$ *is* the two tables of values. In other words, these two tables form the probabilistically checkable proof, or PCP.

Notice that the two tables are exponentially larger than the transcript. Therefore, the prover cannot send them to the verifier or even materialize them. The purpose of the three techniques discussed next in the text—interactivity, commitment, hide the queries—is, roughly speaking, to allow the verifier to query the prover about the tables without either party having to materialize or handle them.

**Sidebar** 3: Probabilistically checkable proofs (simplified).