Resolving the conflict between generality and plausibility in verified computation

Srinath Setty⋆, Benjamin Braun⋆, Victor Vu⋆, Andrew J. Blumberg⋆, Bryan Parno†, and Michael Walfish⋆

⋆The University of Texas at Austin, Austin, TX †Microsoft Research, Redmond, WA

Abstract. The area of proof-based verified computation (outsourced computation built atop probabilistically checkable proofs and cryptographic machinery) has lately seen renewed interest. Although recent work has made great strides in reducing the overhead of naive applications of the theory, these schemes still cannot be considered practical. A core issue is that the work for the server is immense, in general; it is practical only for hand-compiled computations that can be expressed in special forms.

This paper addresses that problem. Provided one is willing to batch verification, we develop a protocol that achieves the efficiency of the best manually constructed protocols in the literature yet applies to most computations. We show that Quadratic Arithmetic Programs, a new formalism for representing computations efficiently, can yield a particularly efficient PCP that integrates easily into the core protocols, resulting in a server whose work is roughly *linear* in the running time of the computation. We implement this protocol in the context of a system, called Zaatar, that includes a compiler and a GPU implementation. Zaatar is *almost* usable for real problems—without special-purpose tailoring. We argue that many (but not all) of the next research questions in verified computation are questions in secure systems.

1 Introduction

The rise of third-party and cloud computing has led to renewed emphasis on a basic systems security problem: allowing one party to check the output of a computation outsourced to a separate, potentially undependable party. While the first party could in principle carry out the computation itself as a check, that would defeat the purpose of having outsourced the computation in the first place. For example, a typical use of cloud computing is performing large-scale simulations. Given the risk of incorrect program execution, users want to know that the answers that are returned are correct—and local checking is not an option.

There are many approaches to solving this problem that rely on assumptions about the computation or usage model. Replication [4, 19, 32, 41] assumes that failures are uncorrelated. Trusted hardware [48] and attestation [46, 49] can work for any failure model but assume a chain of trust that is inconsistent with many applications. Auditing [43] can be effective for highly structured computations but assumes detailed knowledge of the intermediate results.

In principle, there are solutions that make almost *no* such assumptions. These solutions, which we refer to as *verified computation* protocols, use modern cryptography coupled with seminal work in complexity theory on probabilistically checkable proofs (PCPs) [6, 7]. In these solutions, a remote server returns not only the result of the computation but also a proof of correctness, encoded to enable efficient checking by a client. The theoretical guarantees are very strong: there are no assumptions about the server (besides standard cryptographic ones) or about the computation. Indeed, any program can be compiled into such a protocol—*in principle*.

The stumbling block has been practicality. Although there are many protocols for general-purpose verified computation in the literature [26, 30, 31, 33, 34] and have been for a long time [8], the theory, if implemented naively, is preposterously expensive: hundreds of trillions of CPU-years or more to verify simple computations [52].

Very recently, researchers have started adapting these protocols for use in real systems [23, 50, 52, 54, 55]. The good news is that this work has produced massive performance improvements over naive implementations (e.g., 20 orders of magnitude [52, 54]). The bad news is that these systems are still essentially impractical.

Indeed, the overhead for both verifier and server remains excessive, owing to the proof encoding needed to make the proof efficiently checkable. For general computations, the prover runs in time *quadratic* in the running time of the computation. To support claims to plausibility, existing work introduces a tradeoff between generality and efficiency. Specifically, by restricting themselves to computations that can be expressed in special forms, they achieve better performance; Setty et al. [54] achieve efficiency for hand-tailored protocols for particular computations (e.g., matrix multiplication), and the work of Thaler et al. [23, 55] is restricted a priori to computations that are efficiently parallelizable and can be expressed as concise arithmetic circuits.

This paper resolves this conflict:

(1) Using a new formalism [27], we introduce a novel *linear PCP* that works over *constraints* (these terms are explained in Section 2) and plugs into the framework of Ginger (the verified computation system of [54]) to provide a proof encoding that is now linear, instead of quadratic, in computation running time plus program length (encoded as constraints).[1] As a theoretical matter, this resolves an open

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Eurosys '13 April 15-17, 2013, Prague, Czech Republic.

Copyright © 2013 ACM 978-1-4503-1994-2/13/04. . . $15.00

[1] The observation that the formalism of [27] yields linear PCPs has been

problem posed by Ishai et al. [33] (who speculated that such an encoding was possible) and results in a protocol in which the server's overhead is nearly linear (there are two logarithmic factors, as we explain later). As a practical matter, this demolishes costs: the server's overhead on *all* computations is similar to the overhead for the best manually constructed computations in prior work [23, 54, 55].

(2) Using the Ginger implementation as a base, we have integrated our refined linear PCP into a built system for verified computation, called Zaatar. Zaatar adapts the compiler from Ginger (itself descended from the Fairplay compiler [40]) to produce the constraints needed by the new linear PCP. This allows Zaatar to take a high-level language as input and to leverage Ginger's GPU implementation.

(3) We perform a detailed experimental evaluation of Zaatar. We examine real programs that are expressed in a high-level language and compiled automatically (by contrast, most of the evaluated computations in prior work were manually constructed [23, 52, 54, 55]). For various realistic benchmark computations, including sorting, clustering, and shortest paths, Zaatar's prover is significantly more efficient than that of its base, Ginger. For instance, with the benchmark problems that we experimented with, the prover's costs usually reduce by 3–6 orders of magnitude, depending on the input size. The verifier's costs drop by similar amounts. Therefore the verifier can batch-verify a plausibly small set (thousands) of computations and still gain from outsourcing. As one would expect, Zaatar's measured scaling behavior (linear) is far more favorable than Ginger's (quadratic).

Although we are not quite ready to start a company, these results are encouraging. For one thing, they apply in realistic scenarios: regimes in which batching (verifying multiple instances of the same computation on different inputs) is acceptable and users are willing to pay something for security guarantees. These requirements are consistent with current uses of cloud computing. For instance, large-scale simulations in scientific computing often have repeated structure, as does the map phase of MapReduce computations (however, our system does not yet apply in situations where the verifier does not locally have all of the input).

More importantly, the technical work of this paper reduces costs to the point where the problem of verified computation is now at least partially a problem in secure systems research, realizing the vision of Setty et al. [50]. That is, there is still overhead (which would ideally be smaller), but many of the next steps are systems problems. For instance, there are many integration questions. We need to integrate the verification machinery with (potentially untrusted) storage [17, 18, 36, 38], to handle computations with side-effects. To apply to MapReduce, we need to modify the protocols to handle the case where the inputs are not all available locally. To enable users to write real programs, we need to

independently made in a theoretical context [15], as detailed in Section 6.

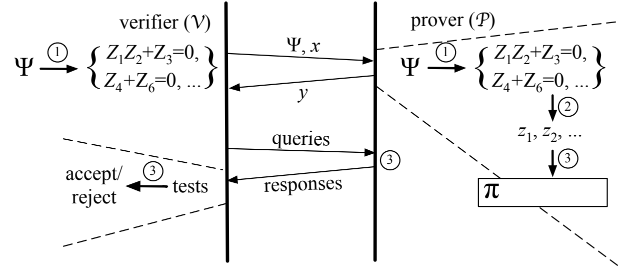

Figure 1—Our framework at a high level. $\mathcal{V}$ specifies a computation, Ψ. Step ①: $\mathcal{V}$ and $\mathcal{P}$ compile Ψ to constraints. Step ②: $\mathcal{P}$ solves the constraints. Step ③: $\mathcal{P}$ produces a proof π, and $\mathcal{V}$ queries $\mathcal{P}$; the responses prove to $\mathcal{V}$ that $\mathcal{P}$ has solved the constraints. This latter step uses an efficient argument system built atop PCPs.

integrate Zaatar with standard libraries. There are also many general systems questions. Most notably, we need a better compiler. For instance, we need to improve the representation of floating-point numbers. Currently, the compiler requires input size bounds, so we need to optimize recompilation to handle variable input sizes [39].

A systems obstacle could be conceptual complexity. Thus, the next section provides a tutorial in the fundamentals. Subsequent sections describe the innovations in this paper.

2 Framework and background

This section describes our framework (§2.1) and the machinery that forms Zaatar's base (§2.2). We intend to provide the necessary context for this paper in a way that is accessible to systems readers. For a more detailed tutorial, see [52, §2].

2.1 Problem statement and approach

Problem statement. A verifier $\mathcal{V}$ sends a description of a computation Ψ and input x to a prover $\mathcal{P}$. $\mathcal{P}$ computes $y = \Psi(x)$, and returns y to $\mathcal{V}$. Then, $\mathcal{V}$ and $\mathcal{P}$ interact in such a way that: (1) if $y = \Psi(x)$, $\mathcal{V}$ accepts y as correct; but (2) if $y \neq \Psi(x)$, $\mathcal{V}$ rejects y, with high probability. The interaction should be cheaper for $\mathcal{V}$ than computing $\Psi(x)$. Furthermore, the guarantees should depend only on $\mathcal{V}$ and not on, say, whether $\mathcal{P}$ obeys the protocol (although we will make standard cryptographic assumptions about the limits of $\mathcal{P}$'s computational power).

Our high-level solution is depicted in Figure 1. The first step is for $\mathcal{V}$ to express its computation in a constraint formalism. For our purposes, a *set of constraints* $\mathcal{C}$ will mean a system of equations that uses field operations (addition and multiplication) over variables (X, Y, Z) in a finite field, $\mathbb{F}$ (for example, $\{Z_1 \cdot Z_2 + Z_3 = 0,\ Z_4 \cdot Z_5 = 0\}$). The variables X and Y will be distinguished input and output variables; for now, we will assume one of each for expositional simplicity. The notation $\mathcal{C}(X{=}x, Y{=}y)$ denotes $\mathcal{C}$ with variable X bound to x and variable Y bound to y. If there exists a setting of the remaining (or *unbound*) variables that makes all of the equations hold simultaneously, then the set of constraints is *satisfiable*; such a setting is called a *satisfying assignment*.

For a given computation Ψ, expressed in a high-level language, we say that a set of constraints $\mathcal{C}$ is *equivalent* to Ψ if: for all x, y, we have $y = \Psi(x)$ if and only if $\mathcal{C}(X{=}x, Y{=}y)$ is satisfiable. As an example, decrement-by-3 is equivalent to $\{X - Z = 0,\ \ Y - (Z - 3) = 0\}$.

In the second step, $\mathcal{P}$ obtains $y = \Psi(x)$ by executing Ψ, and in the process of doing so, identifies z, a satisfying assignment of $\mathcal{C}(X{=}x, Y{=}y)$, where $\mathcal{C}$ is the set of constraints equivalent to Ψ.

The third step is for $\mathcal{P}$ to use the theory of probabilistically checkable proofs (PCPs) to prove to $\mathcal{V}$ that it has a solution. A classical proof for the statement, "These constraints have a solution" would be an actual satisfying assignment, e.g., z, which the verifier could plug into $\mathcal{C}$ to check that each constraint is satisfied. This checking procedure requires the verifier to read the entire proof.

The surprising content of the PCP theorem [6, 7] in our context is that for any set of constraints $\mathcal{C}(X{=}x, Y{=}y)$, there is a randomized verification algorithm, $\mathcal{V}$, that inspects suitably encoded proofs *probabilistically*. Specifically, $\mathcal{V}$ inspects a constant number of randomly chosen locations in a purported proof (the constant is independent of the constraint set), runs some checks against the results, and satisfies:

- **Completeness.** If the constraints are satisfiable—that is, if $y{=}\Psi(x)$—then there exists a suitably encoded proof (a PCP) that makes $\mathcal{V}$ accept.
- **Soundness.** If the constraints are not satisfiable—that is, if $y{\neq}\Psi(x)$—$\mathcal{V}$'s probability of accepting *any* given proof is very small. This probability is over $\mathcal{V}$'s random choices.

Unfortunately, the naturally-implementable "textbook" PCPs [44] are too large to be transferred (and other PCPs, while asymptotically short [11, 12, 24], appear to have prohibitive constants). However, we can handle long PCPs with cryptography: if we assume a standard bound on the prover's computational abilities, we can use *efficient argument systems*. The high-level idea [34] is that the prover *commits* to the contents of the proof by publishing a digest. Then $\mathcal{V}$ asks for the values of the proof at particular locations and verifies that the responses are consistent with the digest. In this way, $\mathcal{P}$ is forced to simulate a fixed proof. That is, an interaction between $\mathcal{V}$ and a fixed proof string becomes an equivalent interactive protocol between two actors, $\mathcal{V}$ and $\mathcal{P}$.

Below, we describe Ginger [54], which is an efficient argument system that implements the idea above, building on the foundations of Ishai et al. [33] and Pepper [52].

2.2 Our starting point: Ginger

Figure 2 depicts Ginger. We first describe its constraints, then its commitment protocol, and then its PCP.

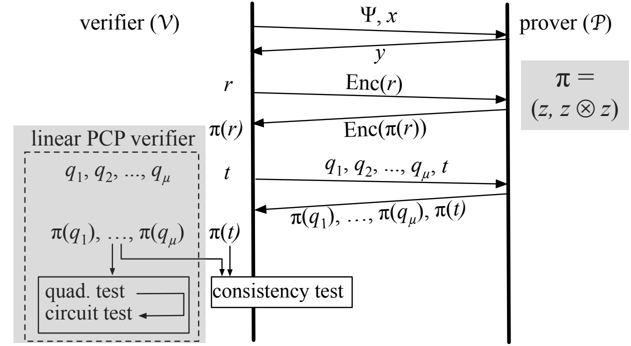

Figure 2—The Ginger protocol [54], which is Zaatar's base; Zaatar modifies the shaded pieces. Ginger and Zaatar verify multiple instances of Ψ in a batch; this is not depicted. Also not depicted are Ψ's equivalent constraints, $\mathcal{C}$.

Computations and constraints. Ginger represents a broad class of computations as degree-2 constraints over $\mathbb{F}$, meaning constraints in which the product terms have no more than two elements. The computations can have field operations, primitive floating-point types, if-then-else constructions, logical tests, logical connectives, and inequality comparisons [54]. As a simple example, the construct `X != Z` can be represented as $\{0 = (X - Z) \cdot M - 1\}$, where M is an auxiliary variable. Ginger's compiler uses this equivalence and others to transform a high-level language into degree-2 constraints. This transformation is detailed in [16]; very broadly speaking, it works as follows. The compiler turns a program (even if it has conditionals and loops) into a list of assignment statements. Then it produces a constraint or *pseudoconstraint* for each statement. Pseudoconstraints abstract certain operations; for instance, order comparisons expand to $O(\log |\mathbb{F}|)$ actual constraints.

Linear commitment. Ginger's commitment protocol leverages PCP constructions [6] in which the proof is encoded as a *linear function*; Ishai et al. call these constructions *linear PCPs* [33].[2] A linear function π is one for which $\pi(a) + \pi(b) = \pi(a + b)$ for all a, b in the domain of π. Note that a linear function $\pi\colon \mathbb{F}^n \mapsto \mathbb{F}$ is determined by a vector. Specifically, there exists a vector $u \in \mathbb{F}^n$ such that $\pi(a) = \langle a, u\rangle$, for all $a \in \mathbb{F}^n$. Here, $\mathbb{F}$ is the same field as above and $\langle a, b\rangle$ is the inner (or dot) product of two vectors.

The protocol has two phases [52]. In the first phase, $\mathcal{V}$ asks $\mathcal{P}$ to apply its function to an encrypted vector;[3] this forces $\mathcal{P}$ to commit to some fixed function π. In the second phase, $\mathcal{V}$ submits the queries given by the PCP protocol (see below) to the function π; note that $\mathcal{P}$ commits to π *before* $\mathcal{P}$ sees these queries. $\mathcal{V}$ then uses the protocol to check that $\mathcal{P}$'s responses are given by π. If not, $\mathcal{V}$ immediately rejects. If so, $\mathcal{V}$ treats the responses as the PCP responses, and runs the PCP checks (described below) on these responses.

[2]To avoid confusion, we note that the term "linear" is relevant in two ways in this paper. First, "*linear* PCPs" refers to the form of the proof (rather than its overhead, which used to be quadratic). Second, the current paper constructs linear PCPs that have *near-linear* overhead.

[3]For this purpose, Ginger requires homomorphic encryption but not fully homomorphic encryption [28]; Ginger uses ElGamal [25].

A linear PCP. A PCP is normally described as an *oracle* π (a fixed function to which $\mathcal{V}$ has access). Since the commitment protocol allows $\mathcal{V}$ to treat $\mathcal{P}$ as implementing that oracle [33, 52, 54], we will be loose about whether $\mathcal{V}$'s queries are submitted to an oracle π or a prover $\mathcal{P}$. Below, we explain the PCP protocol [6], starting with the PCP itself, then $\mathcal{V}$'s queries and checks, and then the costs and guarantees.

Recall that $\mathcal{V}$ begins with Ψ and x, receives y, and wishes to check whether $y = \Psi(x)$; to that end, $\mathcal{V}$ is looking to be convinced that $\mathcal{C}(X{=}x, Y{=}y)$ has a satisfying assignment. A correct prover encodes that satisfying assignment z in a vector $u = (z, z \otimes z)$, where $a \otimes b$ denotes the outer product (the vector consisting of all pairs of products of the components) of a and b. This vector u *is* the linear PCP π. That is, $\pi = (\pi_1, \pi_2)$, where $\pi_1(\cdot) = \langle \cdot, z \rangle$ and $\pi_2(\cdot) = \langle \cdot, z \otimes z \rangle$. We explain the relevance of outer products below; for now, we just note that π will enable $\mathcal{V}$ to probabilistically check whether z is a satisfying assignment. Before proceeding, we give some notation. Let $|\mathcal{C}|$ denote the number of constraints in $\mathcal{C}$. Further, let Z denote the unbound variables in $\mathcal{C}(X{=}x, Y{=}y)$; that is, Z is the set of variables in $\mathcal{C}$ that remain after X and Y are fixed.

The protocol constructs a polynomial $Q(V, Z)$—we say "the *protocol* constructs" because the polynomial is never fully materialized by either party—that depends on $\mathcal{C}$ and (x, y), where $V = (V_1, \ldots, V_{|\mathcal{C}|})$ are variables that $\mathcal{V}$ will set with random choices. We do not give the construction [6, 33, 52] here but state the properties: the construction ensures that Q probabilistically "detects" whether a given assignment z satisfies $\mathcal{C}(X{=}x, Y{=}y)$. Specifically, (1) If z is a satisfying assignment, then $Q(V, z) = 0$ for all possible values of V; but (2) If z is not a satisfying assignment, then for randomly chosen $v \in \mathbb{F}^{|\mathcal{C}|}$, $Q(v, z) \neq 0$, except with probability $1/|\mathbb{F}|$ (and $|\mathbb{F}|$ is large; see Section 5.1).

Thus, $\mathcal{V}$ makes a random choice of $v \in \mathbb{F}^{|\mathcal{C}|}$; if $Q(v, z) = 0$, then $\mathcal{V}$ accepts, and otherwise rejects. This *probabilistic check* of whether the z held by $\mathcal{P}$ is a satisfying assignment is justified because of Q's properties (1) and (2), stated above.

But how, without direct access to z, does $\mathcal{V}$ check whether $Q(v, z) = 0$? The answer is that $Q(V, Z)$, when evaluated at a value $V = v$, is a degree-2 polynomial in Z (see [6, 33, 52] for details), which implies that we can write $Q(v, Z)$ as:

$$Q(v, Z) = \langle \gamma_2, Z \otimes Z \rangle + \langle \gamma_1, Z \rangle + \gamma_0,$$

where the random $\{v_j\}$ determine $\gamma_0 \in \mathbb{F}$, $\gamma_1 \in \mathbb{F}^{|Z|}$, and $\gamma_2 \in \mathbb{F}^{|Z|^2}$. With $Q(v, Z)$ in this form, $\mathcal{V}$ can obtain $Q(v, z)$ by asking $\mathcal{P}$ for $\pi_2(\gamma_2) = \langle \gamma_2, z \otimes z \rangle$ and $\pi_1(\gamma_1) = \langle \gamma_1, z \rangle$, and adding γ_0. Thus, to check whether $Q(v, z) = 0$, $\mathcal{V}$ checks whether $\pi_2(\gamma_2) + \pi_1(\gamma_1) + \gamma_0 = 0$. (Now we can see why $\mathcal{P}$ forms the outer product $z \otimes z$: to answer the "γ_2 queries".)

The above test consists of issuing *circuit queries* (γ_1 and γ_2) and applying a particular check to the results. There are two further sets of queries. $\mathcal{V}$ must check that π indeed is a linear function, and that this linear function has the form $(z, z \otimes z)$ (else, the circuit test is meaningless). For these two purposes, $\mathcal{V}$ issues to π_1 and π_2 *linearity queries* and *quadratic correction* queries, and applies a very efficient check to the results. These tests are run multiple times, with independent random choices; the number depends on the desired soundness (see below).

Notice that constructing queries requires formulating inputs to π_1 and π_2 and that these inputs are at least as large as $|Z|$, which itself is roughly the same size as the computation. For this reason, Ginger amortizes the cost of generating the $\{v_j\}$ and the other queries over a *batch* of size β: a set of computations with the same Ψ but different inputs. Thus, the protocol above should be read as processing in parallel multiple inputs $x^{(1)}, \ldots, x^{(\beta)}$, multiple outputs $y^{(1)}, \ldots, y^{(\beta)}$, and multiple proofs $\pi^{(1)}, \ldots, \pi^{(\beta)}$. Given this batched model, cost reductions for $\mathcal{V}$ will show up in $\mathcal{V}$'s *break-even batch size*, defined as the point at which $\mathcal{V}$ gains from outsourcing: this is the minimum batch size at which the cost of query construction is less than the cost to run the computations locally.

We now state the guarantees. If $\mathcal{P}$ computes correctly, then $\mathcal{V}$ accepts the proof and hence believes that $y = \Psi(x)$. If $\mathcal{P}$ does not compute correctly—if it does not participate in the commitment protocol correctly, if it commits to a function that is not linear, if it commits to a linear function not of the form $(z, z \otimes z)$, or if it commits to a linear function $(z', z' \otimes z')$ where z' is not a satisfying assignment—then $\mathcal{V}$ rejects the proof with probability $\geq 1 - \epsilon$, where ϵ is less than one part in a million, for $|\mathbb{F}| = 2^{128}$ (see [53, Apdx. A.2] for details). This soundness can be made arbitrarily close to 1 by repeating queries (to reduce the PCP error) and increasing the field size (to reduce the commitment error, if needed).

3 Collapsing proof length in Zaatar

Ginger [54] faces a severe obstacle to plausible practicality: the prover's per-instance work and the verifier's query setup work are *quadratic* in the size of the computation. We make this statement more precise below, but for now recall that the server's proof vector u is $(z, z \otimes z)$, so $|u| = |z| + |z|^2$. Ginger addresses this issue by manually tailoring the proof vector and the queries, but this works only for some computations.

In this section and the next, we describe how Zaatar addresses the issue in general. This section describes Zaatar's encoding, in which the proof vector size and the verifier's setup work are *linear* in the size of the computation.[4] However, this encoding imposes costs. Section 4 weighs these costs against the benefits, finding that Zaatar is far more favorable than the alternative.

The high-level idea in Zaatar's encoding is to retain Ginger's structure but to replace the linear PCP of Arora et al. [6]

[4]These costs include an extra log factor applied to the number of steps in the computation. The reason is that the size of our field $\mathbb{F}$ must be larger than the number of steps in the computation, and meanwhile each entry in the proof is $\log |\mathbb{F}|$ bits. However, we neglect this factor in our description by referring to "the size of the computation", which captures the field size.

with a new linear PCP that is based on Quadratic Arithmetic Programs (QAPs), a circuit encoding introduced by Gennaro et al. [27, §7–8]. This substitution works because (a) the linear commitment protocol requires only that the PCP is a linear function; and (b) QAPs have a linear query structure that yields a PCP, a key observation of Zaatar (Bitansky et al. [15] concurrently make a similar observation; see Section 6). To obtain this QAP-based PCP, we extract the essence of Gennaro et al.'s construction (which is more complex because geared to a different regime; see Section 6). Once we do so, we inherit a prover whose proof vector is, to first approximation, the satisfying assignment itself.

Since PCPs "derive their magic" from a highly redundant encoding of the proof, it may seem surprising that we have a protocol in which the proof vector needs little redundancy. However, as Ishai et al. observed [33], a linear PCP contains *implicit* redundancy because the actual proof is *not* the vector but rather the linear function, which is exponentially larger than the classical proof (if written out as a string, it would contain an entry for every point in its domain). For this reason, Ishai et al. speculated that avoiding redundancy in the proof vector might be possible; the construction described below resolves this conjecture.

Loosely speaking, QAPs achieve this compaction by encoding circuits (we adapt the encoding to constraints) as high-degree polynomials, in contrast to the low-degree polynomials of Section 2.2.

Details. As in Ginger, Zaatar takes as a given a constraint set $\mathcal{C}$, over the variables (X,Y,Z), that is equivalent to a computation Ψ. (Recall that X is the set of input variables, Y is the set of output variables, Z are the unbound variables, and let $|\mathcal{C}|$ be the number of constraints in $\mathcal{C}$.)

The protocol constructs two polynomials. The construction is somewhat analogous to the description in Section 2.2; we will mention some of the parallels. The first polynomial, which we call the *divisor polynomial*, $D(t)$, is univariate (and over $\mathbb{F}$) and fixed for all computations Ψ of a given size; $\mathcal{V}$ explicitly materializes $D(t)$. The second polynomial, $P_{x,y}(t,Z)$, depends on the constraints $\mathcal{C}$, the input x, and the purported output y. We write this polynomial as $P(t,Z)$; it is analogous to the polynomial $Q(V,Z)$ in Section 2.2, though here $t \in \mathbb{F}$, versus $V \in \mathbb{F}^{|\mathcal{C}|}$. As with $Q(V,Z)$, neither party fully materializes $P(t,Z)$. We give the complete construction in Appendix A.1 and here state the relevant properties.

The construction ensures that given z, $P(t,z)$ can be factored as $D(t)\cdot H_{x,y,z}(t)$ for some $H_{x,y,z}(t)$ if and only if z satisfies $\mathcal{C}(X{=}x, Y{=}y)$. Meanwhile, this factoring (and hence satisfiability) can be checked *probabilistically*, as follows. Let τ be a random choice from $\mathbb{F}$. Then (1) If z satisfies $\mathcal{C}(X{=}x, Y{=}y)$, then the polynomials factor, so we have $D(\tau)\cdot H_{x,y,z}(\tau) = P(\tau,z)$, for all possible $\tau \in \mathbb{F}$; but (2) If z is not a satisfying assignment, then for *all* polynomials $\tilde{H}(t)$, we have $D(\tau)\cdot\tilde{H}(\tau) \neq P(\tau,z)$, except with probability $2\cdot|\mathcal{C}|/|\mathbb{F}|$. This is because polynomials that are different are equal almost nowhere in their domains (an extreme case is two lines, which cross at most once). Since our fields are large (§5.1), the preceding probability is very small.

The query procedure and $\mathcal{P}$'s proof vector, then, are designed to allow $\mathcal{V}$ to check whether $D(t)\cdot H_{x,y,z}(t) = P(t,z)$, by checking whether this relation holds at a point τ. Specifically, they allow $\mathcal{V}$ to obtain the values $H(\tau) \in \mathbb{F}$ and $P(\tau,z) \in \mathbb{F}$, where: $\mathcal{V}$ chooses τ randomly from $\mathbb{F}$, $\mathcal{P}$ holds the polynomial $H(t)$ and the assignment z ($\mathcal{V}$ has no direct access to either), and neither party materializes $P(t,Z)$. (The analogy here is with the queries that allow $\mathcal{V}$ to obtain $Q(v,z)$, in Section 2.2.) If $D(\tau)\cdot H(\tau) = P(\tau,z)$, then $\mathcal{V}$ accepts and otherwise rejects. (The analogy is with the condition that $Q(v,z)=0$, in Section 2.2.) This procedure probabilistically checks whether z is a satisfying assignment; it is complete and sound because of properties (1) and (2) above.

The proof vector. A correct proof vector u is $(z,\, h)$, where z is a purported satisfying assignment to $\mathcal{C}(X{=}x, Y{=}y)$, and $h = (h_0,\ldots,h_{|\mathcal{C}|}) \in \mathbb{F}^{|\mathcal{C}|+1}$ are the coefficients of the polynomial $H_{x,y,z}(t)$, introduced above. As in Section 2.2, this proof vector u can be regarded as two linear functions, which we denote $\pi_z(\cdot) = \langle\cdot, z\rangle$ and $\pi_h(\cdot) = \langle\cdot, h\rangle$. Thus, the proof vector's length is equal to the number of variables plus the number of constraints, or $|Z| + |\mathcal{C}|$.

The queries and check. To carry out the probabilistic check described above, $\mathcal{V}$ must first ensure that $\mathcal{P}$ is holding a linear function, so $\mathcal{V}$ issues linearity queries (as in Section 2.2).

Next, $\mathcal{V}$ must obtain the value $H_{x,y,z}(\tau) \in \mathbb{F}$. To do so, $\mathcal{V}$ submits $q_h = (1,\tau,\tau^2,\ldots,\tau^{|\mathcal{C}|})$ to $\mathcal{P}$ and asks for $\pi_h(q_h)$, which equals $\langle q_h, h\rangle = \sum_{i=0}^{|\mathcal{C}|} h_i\cdot\tau^i = H_{x,y,z}(\tau)$.

$\mathcal{V}$ also needs the value $P(\tau,z) \in \mathbb{F}$. As shown in Appendix A.1, $P(t,Z)$ is a polynomial in t and Z, with the form

$$P(t,Z) = \left(\sum_{i=1}^{|Z|} Z_i\cdot A_i(t) + A'(t)\right)\cdot\left(\sum_{i=1}^{|Z|} Z_i\cdot B_i(t) + B'(t)\right) - \left(\sum_{i=1}^{|Z|} Z_i\cdot C_i(t) + C'(t)\right),$$

for some polynomials $\{A_i(t), B_i(t), C_i(t)\}_{i=1\ldots|Z|}$ and $\{A'(t), B'(t), C'(t)\}$. Now, observe that evaluating $P(t,Z)$ at $t{=}\tau$ yields $P(\tau,Z)$, a polynomial in Z with the form:

$$P(\tau,Z) = (\langle q_a, Z\rangle + L_a)\cdot(\langle q_b, Z\rangle + L_b) - (\langle q_c, Z\rangle + L_c),$$

where $\{q_a, q_b, q_c\} \in \mathbb{F}^{|Z|}$ depend on τ, and $\{L_a, L_b, L_c\} \in \mathbb{F}$ depend on τ, x, and y. Finally we can say how $\mathcal{V}$ obtains $P(\tau,z)$: it asks $\mathcal{P}$ for $\pi_z(q_a)$, $\pi_z(q_b)$, and $\pi_z(q_c)$.

$\mathcal{V}$'s check is then the following. $\mathcal{V}$ computes $D(\tau)$ and $\{L_a, L_b, L_c\}$, and checks

$$D(\tau)\cdot\pi_h(q_h) \stackrel{?}{=} (\pi_z(q_a) + L_a)\cdot(\pi_z(q_b) + L_b) - (\pi_z(q_c) + L_c).$$

The full protocol and its analysis are in Appendix A (see also [51]); as in Section 2.2, the soundness error is less than one part in a million. We now turn to costs.

	Ginger [54]	Zaatar
proof vector size ($\lvert u_{\text{ginger}}\rvert$ or $\lvert u_{\text{zaatar}}\rvert$)	$\lvert Z_{\text{ginger}}\rvert + \lvert Z_{\text{ginger}}\rvert^2$	$\lvert Z_{\text{zaatar}}\rvert + \lvert \mathcal{C}_{\text{zaatar}}\rvert = 2 \cdot (\lvert Z_{\text{ginger}}\rvert + K_2)$
worst case proof vector size	$\lvert Z_{\text{ginger}}\rvert + \lvert Z_{\text{ginger}}\rvert^2$	$\lvert u_{\text{ginger}}\rvert \cdot (1+\delta)$, where δ is $2/(\lvert Z_{\text{ginger}}\rvert + 1)$
***P*'s per-instance CPU costs**		
Construct proof vector	$T + f \cdot \lvert Z_{\text{ginger}}\rvert^2$	$T + 3f \cdot \lvert \mathcal{C}_{\text{zaatar}}\rvert \cdot \log^2 \lvert \mathcal{C}_{\text{zaatar}}\rvert$
Issue responses	$(h + (\rho \cdot \ell + 1) \cdot f) \cdot \lvert u_{\text{ginger}}\rvert$	$(h + (\rho \cdot \ell' + 1) \cdot f) \cdot \lvert u_{\text{zaatar}}\rvert$
***V*'s per-instance CPU costs**		
Construct computation-specific queries	$\rho \cdot (c \cdot \lvert \mathcal{C}_{\text{ginger}}\rvert + f \cdot K)/\beta$	$\rho \cdot (c + (f_{div} + 5f) \cdot \lvert \mathcal{C}_{\text{zaatar}}\rvert + f \cdot K + 3f \cdot K_2)/\beta$
Construct computation-oblivious queries	$(e+2c+\rho\cdot(2\rho_{\text{lin}}\cdot c+(\ell+1)\cdot f))\cdot\lvert u_{\text{ginger}}\rvert/\beta$	$(e + 2c + \rho \cdot (2\rho_{\text{lin}} \cdot c + \ell' \cdot f)) \cdot \lvert u_{\text{zaatar}}\rvert/\beta$
Process responses	$d + \rho \cdot (2\ell + \lvert x\rvert + \lvert y\rvert) \cdot f$	$d + \rho \cdot (\ell' + 3\lvert x\rvert + 3\lvert y\rvert) \cdot f$

$\mathcal{C}_{\text{ginger}}, \mathcal{C}_{\text{zaatar}}$: Ginger and Zaatar constraint sets for Ψ (§2.1)
$\lvert Z_{\text{ginger}}\rvert$: number of variables in $\mathcal{C}_{\text{ginger}}$ (§2.1)
$\lvert \mathcal{C}_{\text{ginger}}\rvert$ ($= \lvert Z_{\text{ginger}}\rvert$): number of constraints in $\mathcal{C}_{\text{ginger}}$ (§4)
K: number of additive terms in $\mathcal{C}_{\text{ginger}}$ (§4)
K_2 ($\leq K$): number of distinct additive degree-2 terms in $\mathcal{C}_{\text{ginger}}$ (§4)
ρ_{lin}, ρ: number of linearity tests, number of PCP repetitions
e, d: cost of encrypting, decrypting an element in $\mathbb{F}$ (§5.1)
h: cost of ciphertext add plus multiply (§5.1)
c: cost of pseudorandomly generating an element in $\mathbb{F}$ (§5.1)
T: running time of Ψ (§5.2)
$\lvert Z_{\text{zaatar}}\rvert$ ($= \lvert Z_{\text{ginger}}\rvert + K_2$): number of variables in $\mathcal{C}_{\text{zaatar}}$ (§4)
$\lvert \mathcal{C}_{\text{zaatar}}\rvert$ ($= \lvert Z_{\text{ginger}}\rvert + K_2$): number of constraints in $\mathcal{C}_{\text{zaatar}}$ (§4)
$\lvert x\rvert, \lvert y\rvert$: number of elements in input, output (§2.1)
β: batch size (number of instances) (§2.2, §5.2)
$\ell = 3\rho_{\text{lin}} + 2$: number of (high-order) PCP queries in Ginger [53]
$\ell' = 6\rho_{\text{lin}} + 4$: number of (total) PCP queries in Zaatar (§A.1)
f: cost of multiplying in $\mathbb{F}$ (§5.1)
f_{div}: cost of division in $\mathbb{F}$ (§5.1)

Figure 3—Costs to prover and verifier to verify a computation Ψ, under Zaatar and Ginger (the text explains this choice of baseline). Zaatar increases the number of constraints and variables, and requires additional bookkeeping for the prover and verifier, but these effects are dominated by a vast reduction in the proof encoding, so the Zaatar prover and verifier are far more efficient overall. The term K_2 is key to the comparison; this term is large only for degenerate computations (see text). The table assumes that the constraints $\mathcal{C}_{\text{ginger}}$ and $\mathcal{C}_{\text{zaatar}}$ have been generated by our compilers. *Computation-specific* queries refer to those that depend on the structure of the computation and its constraints, while *computation-oblivious* queries depend only on the length of the proof vector.

4 Cost-benefit analysis

Given our goal of removing prover overhead, the Zaatar protocol is very promising. However, we need to consider its benefits against the cost of its additional requirements. This section performs an analysis, summarized in Figure 3.

Our chosen baseline for this analysis is the encoding in Ginger [54]; Ginger was previously the most efficient verified computation scheme that is both a *system* and *general-purpose*. (See Section 6 for discussion of this assertion.)

Summary of the analysis. Zaatar requires more constraints over a larger set of variables than Ginger does for the same computation; all other things being equal, this slight blowup would increase the prover's and the verifier's costs. Also, Zaatar requires additional bookkeeping from the prover (when constructing the proof encoding) and the verifier (when constructing queries). However, these two effects are dwarfed by a vast reduction in the size of the proof encoding under Zaatar. The consequence is a correspondingly vast improvement in both the prover's work and the verifier's query setup work (and hence the break-even batch size, as defined in Section 2.2). Finally, while there are cases when Zaatar is worse than Ginger, they are contrived computations with a particular structure (e.g., evaluation of dense degree-2 polynomials); none of the computations that we have investigated (§5) comes close to this degenerate behavior.[5]

Below, we present the analysis. The comparison depends heavily on the number of constraints and variables in Zaatar versus the alternative, so we begin with these quantities.

Constraints in Zaatar versus Ginger. Whereas Ginger requires only that constraints are degree-2 (§2.2), Zaatar imposes an additional requirement. Under Zaatar, each constraint Q_j must be in the form $p_A \cdot p_B = p_C$, where p_A, p_B, and p_C are degree-1 polynomials over the variables in the constraint set. We call this *quadratic form*; the requirement stems from the way that Zaatar, via QAPs, encodes constraints in polynomials (Appendix A.1 gives detail).

We can obtain constraints $\mathcal{C}_{\text{zaatar}}$ in quadratic form, given a set of Ginger constraints $\mathcal{C}_{\text{ginger}}$; indeed, our compiler first compiles to Ginger constraints and then performs the following transformation. For every constraint in $\mathcal{C}_{\text{ginger}}$, we retain all of the degree-1 terms and replace all degree-2 terms with a new variable. For example, if a constraint in $\mathcal{C}_{\text{ginger}}$ is $\{3 \cdot Z_1Z_2 + 2 \cdot Z_3Z_4 + Z_5 - Z_6 = 0\}$, then $\mathcal{C}_{\text{zaatar}}$ would replace that constraint with the following constraints, which are all in quadratic form: $\{(3 \cdot Z'_1 + 2 \cdot Z'_2 + Z_5) \cdot (1) = Z_6,\ Z_1Z_2 = Z'_1,\ Z_3Z_4 = Z'_2\}$.

We bound the number of variables and constraints in $\mathcal{C}_{\text{zaatar}}$ as follows. Letting $\lvert Z_{\text{zaatar}}\rvert$ (resp., $\lvert Z_{\text{ginger}}\rvert$) equal the number of variables in $\mathcal{C}_{\text{zaatar}}(X{=}x, Y{=}y)$ (resp., $\mathcal{C}_{\text{ginger}}(X{=}x, Y{=}y)$), by construction of $\mathcal{C}_{\text{zaatar}}$ we have $\lvert Z_{\text{zaatar}}\rvert = \lvert Z_{\text{ginger}}\rvert + K_2$, where K_2 is the number of distinct degree-2 terms that appear in all of $\mathcal{C}_{\text{ginger}}$. Similarly, $\lvert \mathcal{C}_{\text{zaatar}}\rvert = \lvert \mathcal{C}_{\text{ginger}}\rvert + K_2$.

We analyze the drop in proof vector size at the end of this section; see also the first two lines of Figure 3.

[5] Also, the degenerate cases are detectable, so the compiler could simply choose to use Ginger (or [23, 55]) over Zaatar; see [57].

The prover's work. Because of the shorter proof vector length, the prover's work to reply to queries drops, usually dramatically (see the "Issue responses" row in Figure 3). However, the prover has an additional cost under Zaatar.

The prover must compute the coefficients of the polynomial $H_{x,y,z}(t) = P(t,z)/D(t)$ (see Section 3). As a starting point in this computation, the prover knows values taken by the polynomials $\{A_i(t), B_i(t), C_i(t)\}$ and $\{A'(t), B'(t), C'(t)\}$ at well-known values of t. Using operations based on the FFT (interpolation [35], polynomial multiplication [21], and polynomial division), the prover obtains the coefficients of $H_{x,y,z}(t)$ in time $\approx 3 \cdot f \cdot (|\mathcal{C}_{\text{zaatar}}| \cdot \log^2 |\mathcal{C}_{\text{zaatar}}|)$, as depicted in Figure 3. The process is detailed in Appendix A.3.

The verifier's work. Like the prover, the verifier in Zaatar also gains from the shorter proof vector; see the "Computation-oblivious queries" line in Figure 3. However, the Zaatar verifier incurs two additional costs, which we summarize immediately below and detail in Appendix A.3.

The first is the cost to construct the query, which is depicted in the "Computation-specific queries" line in the figure. Under Ginger, the verifier must compute γ_1 and γ_2 in order to issue a circuit query (§2.2); this requires generating a pseudorandom number for each constraint and then multiplying it with each term in the given constraint, yielding amortized costs of $\rho \cdot (c \cdot |\mathcal{C}_{\text{ginger}}| + f \cdot K)/\beta$ for a batch of size β. The analog under Zaatar is computing queries to z and to h, which costs for the batch $\rho \cdot (c + (f_{div} + 5f) \cdot |\mathcal{C}_{\text{zaatar}}| + f \cdot K + 3f \cdot K_2)$. The second cost is that Zaatar's verifier requires two more operations per input and output, owing to the details of query construction (see Appendices A.1 and A.3).

Detailed analysis of $|u|$. For a given computation, Ginger's proof vector has length $|u_{\text{ginger}}| = |Z_{\text{ginger}}| + |Z_{\text{ginger}}|^2$ (per Section 2.2). By contrast, Zaatar's proof vector has length $|u_{\text{zaatar}}| = |Z_{\text{zaatar}}| + |\mathcal{C}_{\text{zaatar}}|$ (per Section 3); recalling that $|Z_{\text{zaatar}}| = |Z_{\text{ginger}}| + K_2$ and $|\mathcal{C}_{\text{zaatar}}| = |\mathcal{C}_{\text{ginger}}| + K_2$, we can write $|u_{\text{zaatar}}| = |Z_{\text{ginger}}| + |\mathcal{C}_{\text{ginger}}| + 2K_2$. However, $|\mathcal{C}_{\text{ginger}}| \approx |Z_{\text{ginger}}|$, and we will in fact take $|\mathcal{C}_{\text{ginger}}| = |Z_{\text{ginger}}|$: our compiler, when configured to output Ginger constraints, creates roughly one new variable for each constraint that it introduces.[6] Thus, we get $|u_{\text{zaatar}}| = 2 \cdot (|Z_{\text{ginger}}| + K_2)$.

To compare $|u_{\text{zaatar}}|$ to $|u_{\text{ginger}}|$, we make three points. First, $|u_{\text{zaatar}}|$ is less than $|u_{\text{ginger}}|$ as long as $K_2 < K_2^* \stackrel{\text{def}}{=} (|Z_{\text{ginger}}|^2 - |Z_{\text{ginger}}|)/2$. Indeed, we expect that for most computations, K_2 will be far smaller than K_2^*. Roughly speaking, this fails to occur only when the computation involves adding the product of many multiplications; the reason is that (a) K_2^* corresponds to a computation in which the average number of *distinct* degree-2 terms per Ginger constraint is $(|Z_{\text{ginger}}| - 1)/2$, and (b) our compiler produces a Ginger constraint with more than $(|Z_{\text{ginger}}| - 1)/2$ terms only when compiling a program excerpt that involves summing many terms that are degree-2 (or higher).[7] If most of the constraints have this form, it means that most of the computation involves such sums, which is a degenerate case. An example is degree-2 polynomial evaluation, for which the Ginger encoding is actually very concise [52].

Second, even in the degenerate cases, $|u_{\text{zaatar}}|$ is very close to $|u_{\text{ginger}}|$. The worst case is when K_2 is maximal, which happens when every pair of variables in Z_{ginger} appears as a degree-2 term in $\mathcal{C}_{\text{ginger}}$; that is, the maximum value of K_2 is $K_2 = |Z_{\text{ginger}}| \cdot (|Z_{\text{ginger}}| + 1)/2$. Recalling that $|u_{\text{zaatar}}| = 2 \cdot (|Z_{\text{ginger}}| + K_2)$, we get $|u_{\text{zaatar}}| \leq 2 \cdot |Z_{\text{ginger}}| + |Z_{\text{ginger}}| \cdot (|Z_{\text{ginger}}| + 1) = 3 \cdot |Z_{\text{ginger}}| + |Z_{\text{ginger}}|^2$. But $|u_{\text{ginger}}| = |Z_{\text{ginger}}| + |Z_{\text{ginger}}|^2$, so $|u_{\text{zaatar}}| \leq |u_{\text{ginger}}| \cdot (1 + 2/(|Z_{\text{ginger}}| + 1))$, which is indeed close to $|u_{\text{ginger}}|$.

Third, for all of the computations that we investigate and evaluate, K_2 is far smaller than K_2^* (i.e., we are nowhere close to the degenerate cases), leading to vast improvements in the length of the proof vector, and hence breakeven batch sizes.

5 Evaluation

In this section, we study the empirical effect of Zaatar's refinements on the end-to-end costs (§5.2) and the proof encoding (§5.3). We also discuss the expressibility of constraints and the limitations of Zaatar itself (§5.4).

5.1 Implementation, benchmarks, and method

We implemented Zaatar in C++ (about 6000 lines, per [58]). Both the verifier and the prover can offload their cryptographic operations to GPUs using CUDA [1]; in addition, the prover can be distributed over multiple machines, with each machine computing a subset of a batch. Zaatar's compiler is derived from Ginger's compiler [54]; we added about 1500 lines of Java and about 500 lines of Python (again, per [58]), to translate computations written in SFDL [40] to constraints in quadratic form (§4). For encryption (see Figure 2 and the "linear commitment" paragraphs in Section 2.2), we use ElGamal [25] with 1024-bit keys; for a pseudorandom generator, we use the ChaCha stream cipher [13].

Our experiments examine the CPU cost of Zaatar for a set of benchmark computations: (a) Partitioning Around Medoids (PAM) clustering [56], (b) root finding via bisection, (c) Floyd-Warshall all-pairs shortest paths [22], (d) the Fannkuch benchmark [3], and (e) the longest common subsequence (LCS) problem. Computations (a), (d), and (e) use 32-bit signed integers as inputs with a 128-bit prime as the field modulus. Computation (b) uses *rational number* inputs with 32-bit numerators, 5-bit denominators, and a field modulus of 220 bits. Computation (c) also has rational inputs, with 32-bit numerators, 32-bit denominators, and 128-bit field modulus. The details of rational number handling

[6]A careful analysis of the compiler indicates that the actual bound is $|\mathcal{C}_{\text{ginger}}| \leq (1 + \alpha) \cdot |Z_{\text{ginger}}|$, for $\alpha = 4/(\log_2 |\mathbb{F}| + 3)$. However, our fields are large (§5.1), which is why the text treats α as equal to 0.

[7]The other constructs that produce constraints with degree-2 terms ($<$ and $==$) produce only an average of one or two distinct degree-2 terms per constraint and add at least twice as many new variables.

and representation are given in [54].

In evaluating Zaatar's CPU costs, we compare to two baselines: local computation and Ginger [54]. We report the prover's per-instance costs, including the CPU time to execute the computation and to participate in the verification protocol. For the verifier, we report the *breakeven batch size* (the term is defined in Section 2.2). Note that this quantity captures only the point at which the verifier's CPU is better off verifying a batch versus executing the batch; this quantity does not take into account the prover's CPU costs or the network costs. However, one could apply our measurements and cost models (see Figure 3 and [54, Fig. 2]) to evaluate these quantities and a breakeven point in terms of, say, dollars.

Our comparisons are aided by our cost model (Figure 3), whose parameters we now estimate with microbenchmarks. We run a program that executes each operation 1000 times and report the average CPU time immediately below, for two field sizes (standard deviations are within 5% of the means). The results are immediately below:[8]

field size	e	d	h	f_{lazy}	f	f_{div}	c
128 bits	65 μs	170 μs	91 μs	68 ns	210 ns	2 μs	160 ns
220 bits	88 μs	170 μs	130 μs	90 ns	320 ns	3 μs	260 ns

We use our cost model to validate our experimental results for Zaatar; we find that the empirical CPU costs are 5-15% larger than the model's predictions. We also use the estimates above to parameterize a cost model of Ginger (see [54, Fig. 2]), and use this model to estimate Ginger's end-to-end performance; we use estimates, rather than empirics, because the computations would be too expensive under Ginger.

Our experiments connect the verifier and the prover to a local network. The prover runs in various configurations (single core, distributed over multiple machines, using GPU acceleration); if the configuration is not specified, the prover uses a single core. The machines in our experiments have an Intel Xeon processor E5540 2.53 GHz with 48GB of RAM and two NVIDIA Quadro FX 5800 GPUs (each GPU has 240 CUDA cores and 4GB of device memory). We measure CPU time of the prover and verifier using `getrusage`. Each reported result will be the mean of three experiments (with standard deviations within 8% of the means).

5.2 End-to-end performance

Prover's end-to-end running time. We report on Zaatar's and Ginger's provers with the following combinations of benchmark computations and input sizes: (a) PAM clustering with 2560 data points (m=20 samples with d=128 dimensions clustered into two groups), (b) root finding for degree-2 polynomials with m=256 variables and L=8 iterations (c) Floyd-Warshall with m=25 nodes, (d) Fannkuch with m=100 permutations of $\{1,\ldots,13\}$, and (e) LCS between two strings of length m=300.

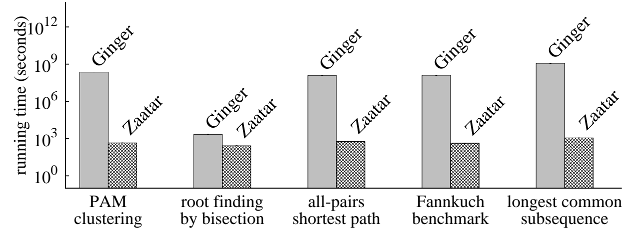

Figure 4—Per-instance running time of the prover under Zaatar and Ginger for various benchmark computations. Zaatar's theoretical refinements improve the running time by 1–6 orders of magnitude compared to the estimated costs of Ginger. The y-axis is log-scaled.

Figure 4 depicts the results. Since Zaatar's proof vectors are much shorter (§4, §5.3), the end-to-end running time of the Zaatar prover is orders of magnitude smaller than the estimated running time of the Ginger prover. For most of the depicted quantities, the difference is 3–6 orders of magnitude. In root finding, the depicted difference is between one and two orders of magnitude because this computation has a relatively efficient representation under Ginger (see Figure 9, Section 5.3). Also, as we increase the number of iterations, L, for instance to 100, the gap between Zaatar and Ginger widens to 3 or more orders of magnitude (however, this configuration requires a higher field size).

We also compare Zaatar's prover to the cost of running the computation locally; see Figure 5. The prover in Zaatar is substantially slower than local computation. We find that about 35% of the work done by the prover is in cryptographic operations, about 40% of the work is in computing the proof vectors, and the remainder is in answering queries.

While this computational burden is heavy, the latency can be tamed. Specifically, we expect that (a) hardware acceleration (e.g., GPUs) reduces latency per-instance, and (b) distributing the prover over more machines makes the latency of a batch not much greater than the latency of a single instance. We experiment by running Zaatar under various hardware configurations (multiple machines, GPUs, etc.), measuring the latency at the verifier. Figure 6 depicts the results. GPU acceleration improves per-instance latency by roughly 20%, and distribution indeed achieves near-linear speedup.

Breakeven batch sizes. We compute Zaatar's breakeven batch sizes by measuring the cost to locally execute a computation instance and to verify one; we decompose the latter into setup costs (which will be amortized) and per-instance costs. We use these quantities to project the breakeven batch sizes. For Ginger, we estimate the breakeven batch sizes using its cost model (as described in Section 5.1).

Figure 7 depicts the results: Zaatar's breakeven batch sizes are several orders of magnitude smaller than Ginger's. This improvement stems from the reduced proof vector size (§5.3), since the verifier's queries are proportional to the length of the prover's proof vector.

[8] The f_{lazy} parameter is not explicitly in Figure 3, but some of the instances of f in the figure should be read as f_{lazy}, which is the cost of a field multiplication that does not require applying "mod p".

computation (Ψ)	local	prover's costs under Zaatar				
		solve constraints	construct u	crypto ops.	answer queries	e2e CPU time
PAM clustering ($m = 20, d = 128$)	51.6 ms	8.6 s	5.0 min	5.3 min	2.4 min	12.8 min
root finding by bisection ($m = 256, L = 8$)	0.8 s	6.3 s	3.3 min	4.1 min	1.4 min	8.9 min
all-pairs shortest path ($m = 25$)	8.1 ms	1.4 s	6.5 min	5.2 min	2.8 min	14.4 min
Fannkuch benchmark ($m = 100$)	0.8 ms	7.5 s	4.8 min	5.1 min	2.3 min	12.3 min
longest common subsequence ($m = 300$)	1.4 ms	13.7 s	12.2 min	10.1 min	5.6 min	28.2 min

Figure 5—Per-instance cost of the Zaatar prover compared to the baseline of local computation (executed with the GMP library [2]), under various computations. The "e2e CPU time" (last column) is decomposed into its contributions (u refers to the proof vector). The end-to-end running time of Zaatar's prover is far more than the cost to execute the computation. However, the costs do not scale up: when batching computations, the latency of the batch is roughly equal to the latency of an instance; see Figure 6.

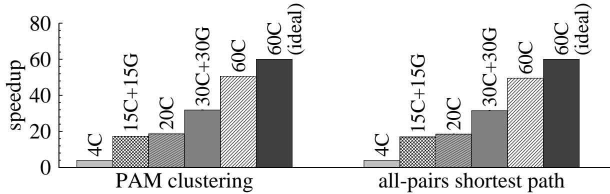

Figure 6—Speedups from parallelizing and distributing the prover. We run with m=10, d=128, and β=60 for PAM clustering, and m=15, β=60 for all-pairs shortest paths. Configurations are denoted with bar labels; for example, 4C means 4 CPU cores, and 15C+15G means 15 CPU cores with 15 GPUs. GPU acceleration improves per-instance latency by about 20%, and Zaatar's prover achieves near-linear speedup as it gets more hardware resources.

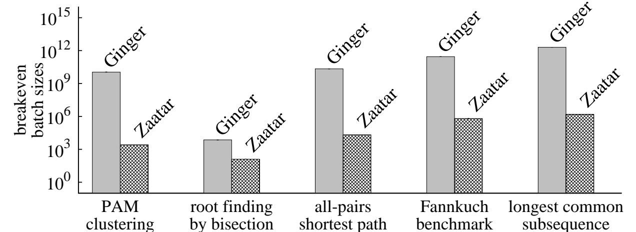

Figure 7—Breakeven batch sizes under Zaatar and Ginger; Zaatar's proof encoding reduces the amount of work done by the verifier, and hence the breakeven batch sizes reduce by several orders of magnitude versus in Ginger. The y-axis is log-scaled.

Scalability. We measure the CPU costs of Zaatar's prover as we vary the input sizes: (a) for PAM clustering, m=$\{5, 10, 20\}$ (and d=128); (b) for root finding, m=$\{64, 128, 256\}$ (and L=8); (c) for Floyd-Warshall, m=$\{5, 10, 20\}$; (d) for Fannkuch, m=$\{25, 50, 100\}$; and (e) for LCS, m=$\{75, 150, 300\}$. We also estimate Ginger's costs at these input sizes. Figure 8 depicts the results. As expected, Zaatar scales better than Ginger.

5.3 Computation encodings

We analyze, as a function of input sizes, the number of variables and constraints in the constraint set representation of our benchmark problems; we perform this analysis for both Zaatar's constraints and Ginger's constraints. We also compute the size of the proof vectors in Zaatar and Ginger. Figure 9 depicts these quantities. The size of Zaatar's proof encoding is linear in the original running time of the computation and is significantly smaller than Ginger's.

5.4 Discussion and limitations

This section has established that (a) Zaatar produces vast performance improvements over the prior state-of-the-art implemented systems, (b) Zaatar's verifier wins only after batching many computations, and (c) Zaatar's prover is substantially more expensive than simply executing the computation. Points (b) and (c) are consequences of the fact that the computation must be encoded as constraints and because of intrinsic costs of the verification machinery.

The constraint formalism is expressive: Degree-2 constraints can represent any computation that terminates in polynomial time (this is implied by Pippenger and Fischer's result [47]). That is, in principle any program for which an upper-bound on running time can be established at compile time can be represented.

However, some programming idioms translate into constraints more efficiently than others. Straight-line arithmetic operations translate directly. Other program structures can produce a large number of constraints for relatively simple operations (for instance, we require $O(\log|\mathbb{F}|)$ constraints for inequality comparisons; see Section 2.2). Worse, under natural translations of computations, indirect memory accesses (for instance, array indices that are not known at compile time) produce an excessive number of constraints.

Less fundamentally, our compiler lacks support for certain program constructs, such as bitwise operations, division, and square root operations. However, this is engineering (e.g., bitwise operations are supported elsewhere [45]).

Given a computation expressed as constraints, the verification machinery imposes further limitations. First, verification requires touching each input and output, so the client saves CPU cycles only when outsourcing computations that take time superlinear in the input size. Second, the sheer size of the queries introduces a substantial setup cost for the verifier; the batched model (§2.2) addresses this cost but the verifier "breaks even" only when it has enough instances to batch. Third, the proof encoding introduces overhead for the prover. Finally, the cryptographic operations are a burden on the verifier and prover, particularly the prover.

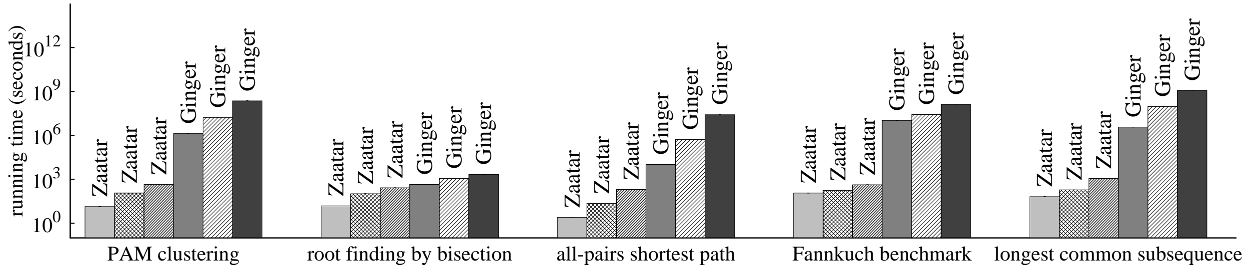

Figure 8—Running time of the prover under Zaatar and Ginger for various benchmark computations with varying input sizes (for each computation and each system, we double the input size twice and report three running times; text has details). Zaatar's prover's work scales linearly; Ginger's, quadratically. The y-axis is log-scaled.

		computation encoding				proof encoding	
computation (Ψ)	$O(\cdot)$	$\lvert Z_{\text{ginger}}\rvert$	$\lvert Z_{\text{zaatar}}\rvert$	$\lvert \mathcal{C}_{\text{ginger}}\rvert$	$\lvert \mathcal{C}_{\text{zaatar}}\rvert$	$\lvert u_{\text{ginger}}\rvert$	$\lvert u_{\text{zaatar}}\rvert$
PAM clustering	$O(m^2d)$	$20m^2d$	$60m^2d$	$20m^2d$	$60m^2d$	$400m^4d^2$	$120m^2d$
root finding by bisection	$O(m^2L)$	$2mL$	m^2L	$2mL$	m^2L	$4m^2L^2$	$2m^2L$
all-pairs shortest path	$O(m^3)$	$84m^3$	$84m^3$	$89m^3$	$89m^3$	$7140m^6$	$173m^3$
Fannkuch benchmark	$O(m)$	$2200m$	$2200m$	$2200m$	$2200m$	$(4.8 \cdot 10^6) \cdot m^2$	$4400m$
longest common subsequence	$O(m^2)$	$43m^2$	$43m^2$	$43m^2$	$43m^2$	$1849m^4$	$86m^2$

Figure 9—Zaatar's and Ginger's computation encodings (i.e., the number of variables and the number of constraints). The $|Z|$ columns give the number of variables, the $|\mathcal{C}|$ columns give the number of constraints, and the $|u|$ columns give the number of terms in the proof encoding. m denotes input size, L denotes the number of iterations in root finding via bisection, and d denotes the number of dimensions in the input sample; see §5.2 for details. For all computations, Zaatar's proof vector is significantly shorter than Ginger's.

6 Related work

Ideally, a protocol for verified computation would be unconditional, general-purpose, and practical. The systems literature contains many practical, implemented systems (see recent surveys [46, 52]); however, these are either special-purpose or conditional (e.g., they require trusted hardware).

Very recently, general-purpose work has emerged that uses tools from cryptography and complexity theory, and relies only on cryptographic assumptions. For example, several protocols [20, 26] employ fully homomorphic encryption (FHE) [28] in protocols for non-interactive verifiable computation. Unfortunately, despite advances in the efficiency of FHE [29], it remains highly impractical.

In a more plausible vein, several authors have worked on succinct arguments that do not rely on FHE [27, 31, 37]. Early work in this area [31, 37] supported only Boolean circuits and suffered from quadratic prover overhead. Gennaro et al. [27] (GGPR) remove these limitations, by coupling QAPs (and a related formalism, called QSPs) with sophisticated cryptography.

Zaatar uses QAPs but without the cryptographic machinery of GGPR; this results in a protocol with different characteristics. For instance, GGPR provides public verifiability (anyone can check a purported proof) while Zaatar does not. And unlike Zaatar, GGPR is *non*-interactive, a property that it achieves by encrypting query-like values and reusing them across computations. This reuse obviates batching and slashes network costs but results in a one-time key setup cost that is likely to be higher (by small constant factors) than Zaatar's per-batch cost. For similar reasons, Zaatar's prover, on a per-instance basis, should be lower overhead than the GGPR prover (Zaatar's prover answers most of its queries in the clear, while the GGPR prover computes over encrypted queries). A more detailed performance comparison to an implemented GGPR refinement [45] is ongoing work.

Independent of us, Bitansky et al. [15] also observe that QAPs yield linear PCPs. However, unlike Zaatar, Bitansky et al. incorporate the resulting linear PCPs into linear interactive proofs and then into succinct non-interactive arguments. This work is part of a theoretical literature that is establishing promising foundations for systems [9, 10].

The work most closely related to Zaatar is several projects aimed at implementing interactive protocols for verified computation, built on different foundations.

One such effort is our prior work [50, 52, 54], which depends on the IKO protocol [33]. IKO coupled a classical linear PCP [6] with a linear commitment primitive to derive an efficient argument. Following a position paper [50], Pepper [52] strengthened this commitment primitive, exploited batching, and applied the approach to arithmetic circuits. Subsequently, Ginger [54] refined the protocols to further reduce costs, extended the computational model to the constraints in Section 2, and implemented the system on GPUs and with a distributed prover.

Another strand is the protocols of Cormode et al. [23, 55], which are based on the work of [30]. These protocols do not need cryptography or batching, and have excellent performance for the verifier. However, results for the prover are more equivocal [54]. Moreover, these protocols require computations to be efficiently parallelizable and expressible as

concise arithmetic circuits, ruling out many programming constructs that Ginger (and Zaatar) contain (e.g., conditionals, order comparisons).

7 Summary and conclusion

The top-level insight in this paper is that the efficient circuit encoding in the QAPs of Gennaro et al. [27] can yield a linear PCP that works with the efficient argument protocol of Setty et al. [52, 54] (itself based on the proposal of Ishai et al. [33]). This observation shows a close relation between QAPs and PCPs, extracting the essence of QAPs—its elegant and efficient encoding—from the cryptographic context in which it was proposed (as observed independently [15]).

This (rather technical) observation leads to the paper's next contributions: a protocol for verified computation whose prover has good overhead—nearly linear—for all computations; an implementation of this protocol in the context of a built system, Zaatar, that includes a compiler and a parallel GPU implementation; and a detailed experimental evaluation, which indicates that Zaatar improves performance by orders of magnitude over the prior state-of-the-art.

Despite these improvements, Zaatar's prover is substantially more expensive than simply executing the computation; the expense enters because the computation must be encoded as constraints (which generalize arithmetic circuits) and because of intrinsic costs of the protocol.

Nevertheless, we expect assurance to have a price, and indeed, there are regimes in which Zaatar's costs are not ridiculous. As an example, consider outsourcing data-parallel computations in the cloud. This setup has (a) an abundance of cheap computing power, meaning that the prover's overhead might be tolerable; and (b) a computation structure that precisely matches the batching requirement of Zaatar's verifier.

In any case, the theoretical underpinnings of Zaatar were thought to be laughably impractical several years ago. As a result of prior work and this paper's work, many of the next steps are now systems problems; we consider that progress!

A A linear PCP protocol based on QAPs

This section describes a new linear PCP [33] protocol, which is used by Zaatar. (Recall that Zaatar's predecessors [33, 52, 54] use the linear PCP protocol of Arora et al. [6].) This protocol is based on Quadratic Arithmetic Programs (QAPs), a construction of Gennaro et al. [27].

Our description will be tailored to our context. In particular, the PCP protocol will check the satisfiability of constraints that are assumed to represent a computation (§2.1). However, this generalizes to checking the satisfiability of any degree-2 constraint set. Since degree-2 constraint satisfaction is an NP-complete problem [5], the PCP that we present here generalizes to checking NP relations. The core idea is to transform a set of constraints to a set of polynomials in such a way that the constraints are satisfiable if and only if the polynomials have a particular algebraic relation.

A.1 The construction

Notation. We are given a constraint set $\mathcal{C}$ over the variables $W = (X, Y, Z)$, where X is the set of distinguished input variables, Y is the set of distinguished output variables, and Z is the set of remaining variables. Let $|\mathcal{C}|$ denote the number of constraints in $\mathcal{C}$. Also, let $n = |W|$ and $n' = |Z|$. The following indexing will be convenient: the variables in Z are labeled as $W_1, \ldots, W_{n'}$, and the variables in X and Y are labeled as $W_{n'+1}, \ldots, W_n$. We will be working over a finite field, $\mathbb{F}$.

Building blocks. The building blocks described in the next several paragraphs are borrowed from QAPs [27], though our notation and phrasing is different, and we work with constraints explicitly.

We require that each constraint is in quadratic form (§4). That is, constraint j has the form $p_{j,A}(W) \cdot p_{j,B}(W) = p_{j,C}(W)$, where $p_{j,A}$, $p_{j,B}$, and $p_{j,C}$ are degree-1 polynomials over W. Now, for variable W_i (which will be a member of X, Y, or Z), let $a_{i,j}$ be the coefficient of W_i in $p_{j,A}$. Similarly, let $b_{i,j}$ be the coefficient of W_i in $p_{j,B}$, and let $c_{i,j}$ be the coefficient of W_i in $p_{j,C}$. Finally, for constraint j, let $a_{0,j}$, $b_{0,j}$, and $c_{0,j}$ be the constant terms in $p_{j,A}, p_{j,B}$, and $p_{j,C}$.

We will construct polynomials that encode these constraints. On the way there, it will be helpful to visualize three $(n+1) \times |\mathcal{C}|$ variable-constraint matrices: A, B, C. In each of these matrices, each row represents a variable in W (as a special case, row $i = 0$ represents the constant terms), and each column represents a constraint in $\mathcal{C}$. In the A (resp., B and C) matrix, the (i, j) cell contains $a_{i,j}$ (resp., $b_{i,j}$ and $c_{i,j}$); thus, this cell is non-zero if variable i appears in constraint j in the $p_{j,A}$ (resp., $p_{j,B}$ and $p_{j,C}$) component. Observe that the matrices A, B, C encode the constraints; we will now turn these matrices into polynomials.

We construct degree-$|\mathcal{C}|$ polynomials $\{A_i(t)\}, \{B_i(t)\}, \{C_i(t)\}$, for $i \in [0..n]$, by interpolation. Take distinguished non-zero points $\sigma_1, \sigma_2, \ldots, \sigma_{|\mathcal{C}|} \in \mathbb{F}$, and for each i require that $A_i(\sigma_j) = a_{i,j}$, $B_i(\sigma_j) = b_{i,j}$, and $C_i(\sigma_j) = c_{i,j}$; at this point, there are $|\mathcal{C}|$ point-value pairs constraining each of the $A_i(t)$, $B_i(t)$, and $C_i(t)$. Finally, require $A_i(0) = B_i(0) = C_i(0) = 0$; this gets us to $|\mathcal{C}| + 1$ points and evaluations, which fully defines the polynomials $A_i(t)$, $B_i(t)$, and $C_i(t)$, by interpolation. Moreover, we can *represent* each $A_i(t)$, $B_i(t)$, and $C_i(t)$ in terms of their evaluations at the values $\{\sigma_j\}$. That is, we can write:

$$A_0(t) = (a_{0,1}, a_{0,2}, \ldots, a_{0,|\mathcal{C}|}) \quad B_0(t) = (b_{0,1}, b_{0,2}, \ldots, b_{0,|\mathcal{C}|})$$
$$A_1(t) = (a_{1,1}, a_{1,2}, \ldots, a_{1,|\mathcal{C}|}) \quad B_1(t) = (b_{1,1}, b_{1,2}, \ldots, b_{1,|\mathcal{C}|})$$
$$\vdots$$
$$A_n(t) = (a_{n,1}, a_{n,2}, \ldots, a_{n,|\mathcal{C}|}) \quad B_n(t) = (b_{n,1}, b_{n,2}, \ldots, b_{n,|\mathcal{C}|})$$

We can do likewise for $C_i(t)$: $C_i(t) = (c_{i,1}, c_{i,2}, \ldots, c_{i,|\mathcal{C}|})$.

Now, construct the *divisor polynomial*, $D(t)$:

$$D(t) = \prod_{j \in [1..|\mathcal{C}|]} (t - \sigma_j).$$

Finally, encode all of the constraints in a single polynomial $P(t,W)$ over t and the constraint variables W:

$$P(t,W) = \left(\sum_{i=0}^{n} W_i \cdot A_i(t)\right)\cdot\left(\sum_{i=0}^{n} W_i \cdot B_i(t)\right) - \left(\sum_{i=0}^{n} W_i \cdot C_i(t)\right)$$

We now give some notation and conventions. We will write $P_{x,y}(t,Z)$ to mean $P(t,W)$ with $X{=}x$ and $Y{=}y$. We will take $w = (x,y,z) \in \mathbb{F}^n$ to mean an assignment to the variables (X,Y,Z); by convention, $w_0 = 1$. Thus, $P_{x,y}(t,z)$ means $P(t, W{=}w)$, for some w; we often write $P(t,W{=}w)$ as $P_w(t)$. Observe that $P_{x,y}(t,Z)$ has the following form, as claimed in Section 3: $P_{x,y}(t,Z) = (\sum_{i=1}^{n'} Z_i \cdot A_i(t) + A'(t)) \cdot (\sum_{i=1}^{n'} Z_i \cdot B_i(t) + B'(t)) - (\sum_{i=1}^{n'} Z_i \cdot C_i(t) + C'(t))$, where $A'(t)$ is a linear combination of $A_0(t), A_{n'+1}(t), \ldots, A_n(t)$, the coefficients given by $1, x, y$, and analogously for $B'(t)$ and $C'(t)$.

Claim A.1. Let $w = (x,y,z)$ be an assignment to the variables (X,Y,Z). Then $D(t)$ divides $P_w(t)$ if and only if z satisfies $\mathcal{C}(X{=}x, Y{=}y)$.

Proof. Assume $D(t)$ divides $P_w(t)$. Fix a constraint $j \in [1..|\mathcal{C}|]$. By definition of $D(t)$, the polynomial $t - \sigma_j$ is a factor of $P_w(t)$, so σ_j is a root of $P_w(t)$. Thus, $0 = P_w(\sigma_j) = \left(\sum_{i=0}^{n} w_i \cdot A_i(\sigma_j)\right) \cdot \left(\sum_{i=0}^{n} w_i \cdot B_i(\sigma_j)\right) - \left(\sum_{i=0}^{n} w_i \cdot C_i(\sigma_j)\right)$. By construction of $\{A_i(t), B_i(t), C_i(t)\}$, we have $\left(\sum_{i=1}^{n} w_i \cdot a_{i,j} + a_{0,j}\right) \cdot \left(\sum_{i=1}^{n} w_i \cdot b_{i,j} + b_{0,j}\right) = \sum_{i=1}^{n} w_i \cdot c_{i,j} + c_{0,j}$. That is, constraint j is satisfied at $w{=}(x,y,z)$, by definition of $a_{i,j}$, $b_{i,j}$, $c_{i,j}$. But we chose j arbitrarily, so z satisfies all constraints in $\mathcal{C}(X{=}x, Y{=}y)$.

For the other direction, if the z "piece" of w satisfies $\mathcal{C}(X{=}x, Y{=}y)$, then every constraint is satisfied, which implies $P_w(\sigma_j) = 0$ for all $\{\sigma_j\}$, so all $\{\sigma_j\}$ are roots of $P_w(t)$, so $P_w(t)$ can be factored into $(t-\sigma_1)\cdots(t-\sigma_{|\mathcal{C}|}) \cdot H_w(t) = D(t) \cdot H_w(t)$, for some $H_w(t)$. □

The QAP-based proof oracle. Let z be the prover's purported assignment to $\mathcal{C}(X{=}x, Y{=}y)$. A correct proof oracle is $\pi = (\pi_z, \pi_h)$, where $\pi_z(\cdot) = \langle \cdot, z\rangle$ and $\pi_h(\cdot) = \langle \cdot, h\rangle$. Here, $h = (h_0, \ldots, h_{|\mathcal{C}|}) \in \mathbb{F}^{|\mathcal{C}|+1}$ represents the coefficients of a polynomial $H(t)$; that is, $H(t) = \sum_{j=0}^{|\mathcal{C}|} h_j \cdot t^j$. In a correct proof oracle for a correct computation, $H(t)$ satisfies $D(t) \cdot H(t) = P_w(t)$.

The PCP protocol. The protocol is depicted in Figure 10. A detail is that queries q_a, q_b, q_c, q_d are self-corrected (see [6, §5] or [44, §7.8.3]). The soundness of the protocol is at least $1 - \kappa^\rho$; Section A.2 establishes this bound and quantifies κ. We cover the verifier's costs in Sections 4 and A.3. For now, note that its costs are proportional to the computation itself, so we amortize them over multiple instances of the computation (§2.2), in the context of the efficient argument system described below.

The efficient argument system. Zaatar is an efficient argument system [33, 34] that composes the PCP in Figure 10 with Ginger's linear commitment primitive [52, 53] (which strengthens a primitive of Ishai et al. [33]). The soundness error of the argument system is upper-bounded by $\kappa^\rho + 9 \cdot \mu \cdot |\mathbb{F}|^{-1/3}$, where μ is the number of PCP queries; see the analysis in [53, Apdx A.2]. The network costs are (a) a full query sent from $\mathcal{V}$ to $\mathcal{P}$, and (b) a random seed from which $\mathcal{V}$ and $\mathcal{P}$ derive the PCP queries pseudorandomly (see [53, Apdx A.3]). The other costs are treated in Section A.3.

A linear PCP based on QAPs [27]

The verifier $\mathcal{V}$ interacts with a proof oracle π as follows. A correct proof oracle encodes z and h, where z satisfies $\mathcal{C}(X{=}x, Y{=}y)$, and h is the coefficients of a polynomial $H_w(t)$ that satisfies $D(t) \cdot H_w(t) = P_w(t)$, for $w = (x,y,z)$.

Loop ρ times:

- Generate linearity queries: Select $q_5, q_6 \in_R \mathbb{F}^{n'}$ and $q_8, q_9 \in_R \mathbb{F}^{|\mathcal{C}|+1}$. Take $q_7 \leftarrow q_5 + q_6$ and $q_{10} \leftarrow q_8 + q_9$. Perform $\rho_{\text{lin}} - 1$ more iterations of this step.
- Generate divisibility correction queries:
  - Select $\tau \in_R \mathbb{F}$.
  - Take $q_a \leftarrow (A_1(\tau), A_2(\tau), \ldots, A_{n'}(\tau))$, and $q_1 \leftarrow (q_a + q_5)$.
  - Take $q_b \leftarrow (B_1(\tau), B_2(\tau), \ldots, B_{n'}(\tau))$, and $q_2 \leftarrow (q_b + q_5)$.
  - Take $q_c \leftarrow (C_1(\tau), C_2(\tau), \ldots, C_{n'}(\tau))$, and $q_3 \leftarrow (q_c + q_5)$.
  - Take $q_d \leftarrow (1, \tau, \tau^2, \ldots, \tau^{|\mathcal{C}|})$, and $q_4 \leftarrow (q_d + q_8)$.
- Issue queries: Send $q_1, \ldots, q_{4+6\rho_{\text{lin}}}$ to oracle π, getting back $\pi(q_1), \ldots, \pi(q_{4+6\rho_{\text{lin}}})$.
- Linearity tests: Check that $\pi(q_5) + \pi(q_6) = \pi(q_7)$ and $\pi(q_8) + \pi(q_9) = \pi(q_{10})$, and likewise for the other $\rho_{\text{lin}} - 1$ iterations. If not, `reject`.
- Divisibility correction test:
  - Take $A_\tau = \left(\pi(q_1) - \pi(q_5) + \sum_{i=n'+1}^{n} w_i \cdot A_i(\tau) + A_0(\tau)\right)$
  - Take $B_\tau = \left(\pi(q_2) - \pi(q_5) + \sum_{i=n'+1}^{n} w_i \cdot B_i(\tau) + B_0(\tau)\right)$
  - Take $C_\tau = \left(\pi(q_3) - \pi(q_5) + \sum_{i=n'+1}^{n} w_i \cdot C_i(\tau) + C_0(\tau)\right)$

  Check $D(\tau) \cdot (\pi(q_4) - \pi(q_8)) = A_\tau \cdot B_\tau - C_\tau$. If not, `reject`.

If $\mathcal{V}$ makes it here, `accept`.

Figure 10—See the text for the definition of $D(t)$ and $P_w(t)$, and the construction of $\{A_i(t)\}$, $\{B_i(t)\}$, $\{C_i(t)\}$, and recall that x and y are labeled as $\{w_{n'+1}, \ldots, w_n\}$.

A.2 Correctness

A verifier $\mathcal{V}$ is given access to a proof oracle π, which is supposed to establish the satisfiability of $\mathcal{C}(X{=}x, Y{=}y)$. The following two claims establish the protocol's correctness. Proofs of these claims are included in [51].

Lemma A.2 (Completeness)**.** If $\mathcal{C}(X{=}x, Y{=}y)$ is satisfiable, if $\pi = (\pi_z, \pi_h)$ is constructed as above, and if $\mathcal{V}$ proceeds according to Figure 10, then $\Pr\{\mathcal{V} \text{ accepts}\} = 1$.

Lemma A.3 (Soundness)**.** There exists a constant $\kappa < 1$ such that if $\mathcal{C}(X{=}x, Y{=}y)$ is not satisfiable and if $\mathcal{V}$ proceeds according to Figure 10, then $\Pr\{\mathcal{V} \text{ accepts}\} < \kappa$ for *all* purported proof oracles $\tilde{\pi}$.

The proof of Lemma A.3 establishes that $\kappa > \max\{(1-3\delta+6\delta^2)^{\rho_{\text{lin}}}, 6\delta + 2\cdot|\mathcal{C}|/|\mathbb{F}|\}$ suffices, for $0 < \delta < \delta^*$, where δ^* is the lesser root of $6\delta^2 - 3\delta + 2/9 = 0$.

As in [53, Apdx A.2], we choose δ to minimize break-even batch sizes. We neglect the ratio $2\cdot|\mathcal{C}|/|\mathbb{F}|$, since $|\mathcal{C}|$ roughly captures the size of the computation and $|\mathbb{F}|$ is astronomical (e.g., $|\mathbb{F}| = 2^{192}$). We choose $\delta = 0.0294$ and $\rho_{\text{lin}} = 20$, and hence $\kappa = 0.177$ suffices. We then take $\rho = 8$ for an upper-bound on soundness error of $\kappa^\rho < 9.6 \times 10^{-7}$.

A.3 Costs in more detail

Earlier in the paper (Figure 3 and Section 4), we stated the costs of Zaatar. This section fleshes out some of those claims.

We repeat the observation of Gennaro et al. [27] that the polynomials $\{A_i(t)\}$, $\{B_i(t)\}$, and $\{C_i(t)\}$ can be represented efficiently, in terms of their evaluations at the $\{\sigma_j\}$. That is, $A_i(t)$ can be written as a list $\{(j, a_{i,j}) \mid a_{i,j} \neq 0, j \in \{1,\ldots,|\mathcal{C}|\}\}$, and similarly for $B_i(t)$ and $C_i(t)$. For convenience, we let $\sigma_0 = 0$ and $a_{i,0} = b_{i,0} = c_{i,0} = 0$.

The prover. To construct the π_h component of its proof vector (§A.1), the prover must compute the coefficients of $H_w(t)$, where $D(t)\cdot H_w(t) = P_w(t)$. Section 4 states that the cost to do so is $3\cdot f\cdot|\mathcal{C}|\cdot\log^2|\mathcal{C}|$. We now detail the process. It is three steps (well-explained in [42, Chapter 1.7]).

Step 1 is writing $P_w(t)$ in the form $A(t)\cdot B(t) - C(t)$, for degree-$|\mathcal{C}|$ polynomials $\{A(t), B(t), C(t)\}$, to obtain the coefficients of $\{A(t), B(t), C(t)\}$. The prover constructs the set $\{(\sigma_j, \sum_i w_i\cdot a_{i,j}) \mid j \in \{0,\ldots,|\mathcal{C}|\}$, which is the evaluations of $A(t)$. The prover then uses multipoint interpolation [35, Chapter 4.6.4] to compute the coefficients of $A(t)$, in time $\approx f\cdot|\mathcal{C}|\cdot\log^2|\mathcal{C}|$. The prover does likewise for $B(t)$ and $C(t)$. Step 2 is computing the coefficients of $P_w(t)$ in time $\approx f\cdot|\mathcal{C}|\cdot\log|\mathcal{C}|$, using multiplication based on the fast Fourier transform (FFT) [21]. Step 3 is computing the coefficients of $P_w(t)/D(t) = H_w(t)$ in time $\approx f\cdot|\mathcal{C}|\cdot\log|\mathcal{C}|$, using FFT-based polynomial division.

The verifier. We will be focused on Figure 10. We stated $\mathcal{V}$'s query setup costs as (§4):

$$c + (f_{div} + 5f)\cdot|\mathcal{C}| + f\cdot K + f\cdot 3K_2.$$

We now explain these costs. Selecting τ costs c. Generating $q_d = (1, \tau, \ldots, \tau^{|\mathcal{C}|})$ costs $f\cdot|\mathcal{C}|$. Most of the remaining costs are generating $(A_0(\tau), A_1(\tau), \ldots, A_n(\tau))$, $(B_1(\tau), \ldots, B_n(\tau))$, and $(C_1(\tau), \ldots, C_n(\tau))$.

Gennaro et al. [27] observe that a Lagrange basis is useful for this purpose; we give the details here. We can write each polynomial $A_i(t)$ as follows (and analogously for $B_i(t), C_i(t)$):

$$A_i(t) = \sum_{j=0}^{|\mathcal{C}|} a_{i,j}\cdot\ell_j(t), \quad \text{where } \ell_j(t) = \prod_{\substack{0\le k\le|\mathcal{C}|\\ k\neq j}} \frac{t-\sigma_k}{\sigma_j-\sigma_k}.$$

We can use Barycentric Lagrange interpolation [14] to write:

$$A_i(t) = \ell(t)\cdot\sum_{j=0}^{|\mathcal{C}|} a_{i,j}\cdot\frac{v_j}{t-\sigma_j}, \text{ where}$$
$$\ell(t) = (t-\sigma_0)(t-\sigma_1)\cdots(t-\sigma_{|\mathcal{C}|}), \text{ and}$$
$$v_j = 1/\prod_{\substack{0\le k\le|\mathcal{C}|\\ k\neq j}}(\sigma_j-\sigma_k).$$

We now explain the remaining costs. Computing $\ell(\tau)$ takes $|\mathcal{C}|$ multiplications; then, computing $D(\tau)$ takes one division and one multiplication, as $D(\tau) = (1/\tau)\cdot\ell(\tau)$. Computing $\{v_j\}$ can be done efficiently via a careful choice of the $\{\sigma_j\}$ (the protocol permits *any* distinct, non-zero values here): if we arrange for $\sigma_1, \ldots, \sigma_{|\mathcal{C}|}$ to follow an arithmetic progression (a convenient choice is $1, 2, \ldots, |\mathcal{C}|$), then computing $1/v_{j+1}$ from $1/v_j$ requires only two operations. Since one can compute $1/v_0$ using $|\mathcal{C}|$ multiplications, the total time to compute the $\{v_j\}$ is $(f_{div} + 3f)\cdot|\mathcal{C}|$ operations.

Finally, given $\{v_j\}$ and $\ell(\tau)$ and using the representation above, one can compute $\{A_i(\tau)\}$, $\{B_i(\tau)\}$, and $\{C_i(\tau)\}$ with a number of multiplications equal to the total number of non-zero $\{a_{i,j}, b_{i,j}, c_{i,j}\}$. This number is computation-dependent (see §A.1), but we can bound it in our framework. Recall that our compiler obtains Zaatar constraints by transforming Ginger constraints (§4). The Zaatar constraints, when written in quadratic form, induce no more than $K + 3K_2$ non-zero $\{a_{i,j}, b_{i,j}, c_{i,j}\}$, where K and K_2 are as defined in Section 4.

In Section 4, we stated that the verifier requires three operations per input and output. This cost comes from computing the following quantities in the divisibility correction test: $\sum_{n'+1}^{n} w_i\cdot A_i(\tau)$, $\sum_{n'+1}^{n} w_i\cdot B_i(\tau)$, and $\sum_{n'+1}^{n} w_i\cdot C_i(\tau)$.

Acknowledgments

Careful reading and constructive suggestions by the anonymous reviewers and our shepherd, Christan Cachin, improved the paper. We thank Sanjeev Arora and Yuval Ishai for helpful conversations. The Texas Advanced Computing Center (TACC) at UT supplied computing resources. The research was supported in part by AFOSR grant FA9550-10-1-0073 and by NSF grants 1055057 and 1040083.

References

[1] CUDA (http://developer.nvidia.com/what-cuda).

[2] The GNU MP bignum library. `http://gmplib.org/`.

[3] Shootout/Fannkuch. `http://www.haskell.org/haskellwiki/Shootout/Fannkuch`.

[4] 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.

[5] S. Arora and B. Barak. *Computational Complexity: A modern approach*. Cambridge University Press, 2009.

[6] 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.

[7] S. Arora and S. Safra. Probabilistic checking of proofs: a new characterization of NP. *J. of the ACM*, 45(1):70–122, Jan. 1998.

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

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

[10] E. Ben-Sasson, A. Chiesa, D. Genkin, and E. Tromer. On the concrete-efficiency threshold of probabilistically-checkable proofs. In *STOC*, 2013. To appear.

[11] E. Ben-Sasson, O. Goldreich, P. Harsha, M. Sudan, and S. Vadhan. Short PCPs verifiable in polylogarithmic time. In *Conference on Computational Complexity (CCC)*, 2005.

[12] E. Ben-Sasson and M. Sudan. Short PCPs with polylog query complexity. *SIAM J. on Comp.*, 38(2):551–607, May 2008.

[13] D. J. Bernstein. ChaCha, a variant of Salsa20. `http://cr.yp.to/chacha.html`.

[14] J.-P. Berrut and L. N. Trefethen. Barycentric Lagrange interpolation. *SIAM Review*, 46(3):501–517, 2004.

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

[16] B. Braun. Compiling computations to constraints for verified computation. UT Austin Honors thesis HR-12-10, Dec. 2012.

[17] C. Cachin. Integrity and consistency for untrusted services. In *Conference on Current Trends in Theory and Practice of Computer Science*, 2011.

[18] C. Cachin, I. Keidar, and A. Shraer. Fail-aware untrusted storage. *SIAM J. on Comp.*, 40(2):493–533, Apr. 2011.

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

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

[21] J. W. Cooley and J. W. Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of Computation*, 19(90):297–301, 1965.

[22] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press, 1990.

[23] G. Cormode, M. Mitzenmacher, and J. Thaler. Practical verified computation with streaming interactive proofs. In *ITCS*, 2012.

[24] I. Dinur. The PCP theorem by gap amplification. *J. of the ACM*, 54(3), June 2007.

[25] T. ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Trans. on Info. Theory*, 31(4):469–472, 1985.

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

[27] R. Gennaro, C. Gentry, B. Parno, and M. Raykova. Quadratic span programs and succinct NIZKs without PCPs. Cryptology ePrint Archive, Report 2012/215, 2012. To appear in EUROCRYPT 2013.

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

[29] C. Gentry, S. Halevi, and N. Smart. Homomorphic evaluation of the AES circuit. In *CRYPTO*, 2012.

[30] S. Goldwasser, Y. T. Kalai, and G. N. Rothblum. Delegating computation: Interactive proofs for muggles. In *STOC*, 2008.

[31] J. Groth. Short pairing-based non-interactive zero-knowledge arguments. In *ASIACRYPT*, 2010.

[32] A. Haeberlen, P. Kouznetsov, and P. Druschel. PeerReview: Practical accountability for distributed systems. In *SOSP*, 2007.

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

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

[35] D. E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming*. Addison-Wesley, third edition, 1997.

[36] J. Li, M. N. Krohn, D. Mazières, and D. Shasha. Secure untrusted data repository (SUNDR). In *OSDI*, 2004.

[37] H. Lipmaa. Progression-free sets and sublinear pairing-based non-interactive zero-knowledge arguments. In *IACR TCC*, 2011.

[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] L. Malka. VMCrypt: Modular software architecture for scalable secure computation. In *ACM CCS*, 2011.

[40] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella. Fairplay—a secure two-party computation system. In *USENIX Security*, 2004.

[41] D. Malkhi and M. Reiter. Byzantine quorum systems. *Distributed Computing*, 11(4):203–213, 1998.

[42] T. Mateer. *Fast Fourier Transform algorithms with applications*. PhD thesis, Clemson University, 2008.

[43] F. Monrose, P. Wycko, and A. D. Rubin. Distributed execution with remote audit. In *NDSS*, 1999.

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

[45] B. Parno, C. Gentry, J. Howell, and M. Raykova. Pinocchio: Nearly practical verifiable computation. In *IEEE Symposium on Security and Privacy*, 2013. To appear.

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

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

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

[49] 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*, 2005.

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

[51] S. Setty, B. Braun, V. Vu, A. J. Blumberg, B. Parno, and M. Walfish. Resolving the conflict between generality and plausibility in verified computation. Cryptology ePrint Archive, Report 2012/622, 2012.

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

[53] S. Setty, V. Vu, N. Panpalia, B. Braun, M. Ali, A. J. Blumberg, and M. Walfish. Taking proof-based verified computation a few steps closer to practicality (extended version). Cryptology ePrint Archive, Report 2012/598, 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*, 2012.

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

[56] S. Theodoridis and K. Koutroumbas. *Pattern Recognition, Third Edition*. Academic Press, Inc., 2006.

[57] V. Vu, S. Setty, A. J. Blumberg, and M. Walfish. A hybrid architecture for interactive verifiable computation. In *IEEE Symposium on Security and Privacy*, 2013. To appear.

[58] D. A. Wheeler. SLOCCount. `http://www.dwheeler.com/sloccount/`.