# 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. The core issue is that the work for the prover is immense, in general; it is near-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 all computations. Our protocol is built on the observation that the recently-proposed QAPs of Gennaro et al. (ePrint 2012/215) yield a linear PCP that works with the efficient argument protocol of Setty et al. (ePrint 2012/598, Security 2012, NDSS 2012), itself based on the proposal of Ishai et al. (CCC 2007). The consequence is a prover whose total work is not much more than *linear* in the running time of the computation. We implement the protocol in the context of a built system that includes a compiler and a parallel GPU implementation. The result, as indicated by an experimental evaluation, is a system that is *almost* usable for real problems—without special-purpose tailoring.

## 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 data mining on very large data sets. Given the risks associated to corrupted input and incorrect program execution, users need 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 [6, 22, 36, 39, 44] is a classic strategy but assumes that failures are uncorrelated. Trusted hardware [23, 49] and attestation [5, 48, 50, 51] can work for any failure model but assume a chain of trust that is inconsistent with many applications. Auditing [36, 38, 46] 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) [8, 9]. 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 [30, 34, 35, 37, 40] and have been for a long time [10], the theory, if

implemented naively, is preposterously expensive: hundreds of trillions of CPU-years or more to verify simple computations [53].

Very recently, researchers have started adapting these protocols for use in real systems [27, 52–55]. The good news is that this work has produced massive performance improvements over naive implementations (e.g., nearly 23 orders of magnitude [53, 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. [27, 55] is restricted a priori to computations that are efficiently parallelizable and can be expressed as concise arithmetic circuits.

This paper resolves this conflict. Specifically, we make the following contributions:

(1) Using a new formalism [31], 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 length that is now linear, instead of quadratic, in computation running time plus program length (encoded as constraints). As a theoretical matter, this resolves an open problem posed by Ishai et al. [37] 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 [27, 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 earlier compiler from Ginger (itself descended from the Fairplay compiler [43]) to produce the constraints needed by the new linear PCP. This allows Zaatar to take a high-level language as input and also to take advantage of Ginger's parallel GPU implementation.

(3) We perform a detailed experimental evaluation of Zaatar in Section 5. We examine real programs expressed in a high-level language and compiled automatically; this is in contrast to prior work, for which most of the evaluated computations were manually constructed [27, 53–55]. Under reasonable usage models, we achieve numbers that are plausible for various realistic benchmark computations, including sorting, clustering, and shortest path computation. We find that Zaatar's prover is significantly more efficient compared to 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 small set 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). For instance, to sort a vector of size 64 using insertion sort, Ginger's prover takes 15 CPU-days whereas Zaatar's prover takes 50 CPU-seconds.

Although we are not quite ready to start a company, these results are encouraging. They apply in realistic scenarios: our results are practical in a regime in which batching (verifying multiple parallel 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 ways that cloud computation is currently used. For instance, the map phase of computations expressed in the MapReduce idiom fits this description.

The next section is intended to provide a self-contained (but concise) tutorial in the fundamentals, while the sections ahead describe the innovations in this paper.

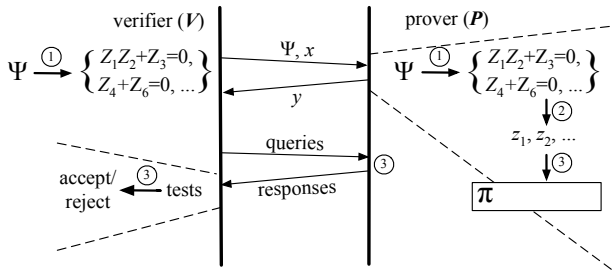



Figure 1—Our framework at a high level. $\mathcal{V}$ specifies a computation, $\Psi$. Step ①: $\mathcal{V}$ and $\mathcal{P}$ compile $\Psi$ to constraints. Step ②: $\mathcal{P}$ solves the constraints. Step ③: $\mathcal{P}$ produces a proof $\pi$, and $\mathcal{V}$ queries $\mathcal{P}$; the responses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

## 2 Framework and background

This section is intended to be a concise and self-contained review of the framework in which we work (§2.1) and the machinery that forms Zaatar's base (§2.2). We discuss the development and evolution of this machinery (see [37, 53, 54]) in Section 6.

### 2.1 Problem statement and approach

*Problem statement.* A verifier $\mathcal{V}$ sends a description of a computation $\Psi$ 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 $y = \Psi(x)$. 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. A *set of constraints* is a system of equations that uses field operations (addition and multiplication) over variables in a finite field, $\mathbb{F}$ (for example, $\{Z_1 \cdot Z_2 + Z_3 = 0, Z_4 \cdot Z_5 = 0\}$). In our context, a set of constraints $\mathcal{C}$ will have distinguished input and output variables, $X$ and $Y$; 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 $\Psi$, expressed in a high-level language, we say that a set of constraints $\mathcal{C}$ is *equivalent* to $\Psi$ if: for all $x, y$, we have $y = \Psi(x)$ if and only if $\mathcal{C}(Y = y, X = x)$ is satisfiable. As an example, decrement-by-3 is equivalent to $\{Z - 3 - Y = 0, Z - X = 0\}$.

Step 2 in the framework: $\mathcal{P}$ obtains $y = \Psi(x)$ by executing $\Psi$, 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 $\Psi$.

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 would allow the verifier to plug in that assignment and check that each constraint is satisfied. This checking procedure requires the verifier to read the entire proof.

The surprising content of the PCP theorem [8, 9] 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 (where the constant is independent of the set of constraints), 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 en-

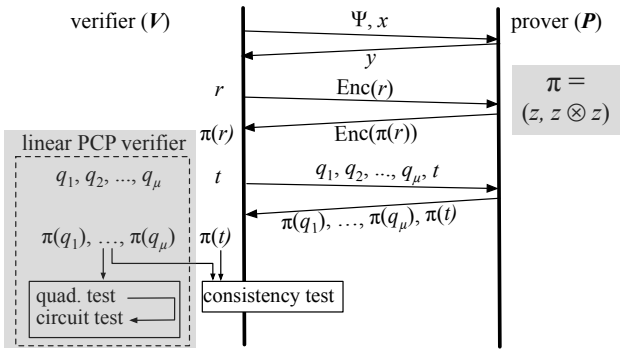



Figure 2—The Ginger protocol [54], which is Zaatar's base; Zaatar modifies the shaded pieces. Ginger and Zaatar verify multiple instances of $\Psi$ in a batch; this is not depicted. Also not depicted are $\Psi$'s equivalent constraints, $\mathcal{C}$.

coded 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 [47] are too large to be transferred (and other PCPs, while asymptotically short [15–17, 28], appear to have prohibitive constants). To handle long PCPs, one can use cryptography. We assume a computational bound on the prover (of the kind made by many cryptographic protocols), and use *efficient argument systems*. The high-level idea [40] is to have the prover *commit* 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, we have turned an interaction between $\mathcal{V}$ and a fixed proof string into 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. [37] and Pepper [53].

### 2.2 Our starting point: Ginger

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

**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. In the rest of this section, we take such a set of constraints, $\mathcal{C}$, as a given.

**Linear commitment.** Ginger's commitment protocol takes advantage of the fact that there are PCP constructions [8] in which the proof is encoded as a *linear function*; Ishai et al. call these constructions *linear PCPs* [37].[1] A linear function $\pi$ is one for which $\pi(a) + \pi(b) = \pi(a + b)$ for all $a, b$ in the domain of $\pi$. 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.

[1] The term "linear" could be confusing here, as it is relevant in two ways in this paper. First, the "linear" in "linear PCPs" refers to the form of the proof, not its overhead (indeed, the construction used by Ishai et al. has *quadratic* costs for the prover). Second, the current paper shows how to make linear PCPs have near-linear costs.

The actual protocol is detailed elsewhere [53]. It has two phases. In the first phase, $\mathcal{V}$ asks $\mathcal{P}$ to apply its function to an encrypted query;[2] this binds $\mathcal{P}$ to some linear function, $\pi$ [54]. In the second phase, $\mathcal{V}$ submits the queries given by the PCP protocol (see below), along with a consistency query. The construction allows $\mathcal{V}$ to check whether $\mathcal{P}$'s responses are given by $\pi$. 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.

**A linear PCP.** A PCP is normally described as an *oracle* $\pi$ (a fixed function to which $\mathcal{V}$ has access). Since the commitment protocol allows $\mathcal{V}$ to treat $\mathcal{P}$ as implementing that oracle [37, 53, 54], we will be loose about whether $\mathcal{V}$'s queries are submitted to an oracle $\pi$ or a prover $\mathcal{P}$. Below, we explain the PCP protocol [8], starting with the PCP itself, then $\mathcal{V}$'s queries and checks, and then the guarantees and costs.

*The construction.* Recall that $\mathcal{V}$ begins with $\Psi$ 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 $\pi$. That is, $\pi = (\pi_1, \pi_2)$, where $\pi_1(\cdot) = \langle \cdot, z \rangle$ and $\pi_2(\cdot) = \langle \cdot, z \otimes z \rangle$.

*The queries and checks.* We begin with 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. $\mathcal{V}$ forms a polynomial $Q(t, Z)$ that depends on $\mathcal{C}$ and $(x, y)$, where $t = (t_1, \ldots, t_{|\mathcal{C}|})$ are variables that $\mathcal{V}$ will set with random choices. The construction is such 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 $Q(t, z) = 0$ for all possible values $t$; but (2) If $z$ is not a satisfying assignment, then for randomly chosen $\tau \in \mathbb{F}$, $Q(\tau, z) \neq 0$, except with probability $1/|\mathbb{F}|$ (and $|\mathbb{F}|$ is large; see Section 5).

$\mathcal{V}$ makes two sets of queries to $\mathcal{P}$. The first set (the *quadratic correction* queries) is designed so that $\mathcal{V}$ can use the responses to check that $\pi$ is indeed in the form $(\tilde{z}, \tilde{z} \otimes \tilde{z})$. If this check passes, the second set (the *circuit* queries) allows $\mathcal{V}$ to obtain the value of $Q(\tau, z)$, for random choice of $\tau$. If $Q(\tau, z) = 0$, then $\mathcal{V}$ accepts, and otherwise rejects. This is a *probabilistic* check of whether the $z$ held by $\mathcal{P}$ is a satisfying assignment; it is justified because of $Q$'s properties (1) and (2), stated above. The details of the queries and checks appear elsewhere [8, 16, 37, 53].

For intuition, we briefly describe how $\mathcal{V}$ checks whether $Q(\tau, z) = 0$ in the circuit test. By the construction of $Q$, $Q(\tau, Z)$ is a degree-2 polynomial in $Z$, so we can write:

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

The random $\{\tau_j\}$ determine $\gamma_0 \in \mathbb{F}$, $\gamma_1 \in \mathbb{F}^n$, and $\gamma_2 \in \mathbb{F}^{n^2}$. Then, $\mathcal{V}$ obtains $Q(\tau, z)$ 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 $\gamma_0$. $\mathcal{V}$'s check is thus whether $\pi_2(\gamma_2) + \pi_1(\gamma_1) + \gamma_0$ equals zero.

*Guarantees and costs.* 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 linear function not of the form $(\tilde{z}, \tilde{z} \otimes \tilde{z})$, or if it commits to a linear function $(\tilde{z}, \tilde{z} \otimes \tilde{z})$ where $\tilde{z}$ is not a satisfying assignment—then $\mathcal{V}$ rejects the proof with probability $\approx \sqrt[6]{1/|\mathbb{F}|}$ (see Setty et al. [54, Apdx. A.2] for a detailed quantification). This soundness can be made arbitrarily close to 1 via repetition of queries (to reduce the PCP error) and a larger field size (to reduce the commitment error).

We will treat costs in detail in the sections ahead. For now, observe that constructing a given query is at least as much work as carrying out the computation. However, these costs can be amortized; Ginger amortizes query generation over a *batch* of size $\beta$: a set of computations with the same $\Psi$ 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 $\pi_1, \ldots, \pi_\beta$.

[2] For this purpose, Ginger requires homomorphic encryption but not fully homomorphic encryption [32], which would be prohibitive; Ginger uses ElGamal [29].

## 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 will 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 only for some computations, via manual tailoring of the proof vector and the queries.

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.[3] 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. [8] with a new linear PCP that is based on Quadratic Arithmetic Programs (QAPs), introduced by Gennaro et al. [31, §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. 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 [37], 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.

**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 $\Psi$. (Recall that $X$ is the set of input variables, $Y$ is the set of output variables, and $Z$ are the unbound variables, and let $|\mathcal{C}|$ be the number of constraints in $\mathcal{C}$.)

$\mathcal{V}$ constructs two polynomials. The first, which we call the *divisor polynomial*, $D(t)$, is fixed for all computations $\Psi$ of a given size. The second, $P_{x,y}(t, Z)$, depends on the constraints $\mathcal{C}$ and on the input $x$ and purported output $y$; we will write this polynomial as $P(t, Z)$. These two polynomials are constructed so that given $z$, $D(t)$ divides $P(t, z)$—meaning that $P(t, z)$ 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 $\mathcal{C}(X = x, Y = y)$. Later in this section, we give the form of $P(t, Z)$; the complete construction is described in Appendix A.1.

Notice that this divisibility condition can be checked *probabilistically*, as follows. Let $\tau$ be a random choice from $\mathbb{F}$. Then (1) If $z$ is a satisfying assignment, $D(\tau) \cdot H_{x,y,z}(\tau) = P(\tau, z)$, for all possible values $\tau$; but (2) If $z$ is not a satisfying assignment, then for *all* polynomials $\tilde{H}(t)$, $D(\tau) \cdot \tilde{H}(\tau) \neq P(\tau, z)$, except with probability $2 \cdot |\mathcal{C}|/|\mathbb{F}|$ (see Appendix A.2). Since our fields are large (§5), this probability is very small.

The prover's vector and the query procedure are described below. They allow $\mathcal{V}$ to obtain the values $H(\tau)$ and $P(\tau, z)$, where $\tau$ is chosen randomly from $\mathbb{F}$ by $\mathcal{V}$, $z$ is an assignment held by $\mathcal{P}$, and $H(t)$ is a polynomial held by $\mathcal{P}$. If $D(\tau) \cdot H(\tau) = P(\tau, z)$, then $\mathcal{V}$ accepts and otherwise rejects. This procedure probabilistically checks whether $z$ is a satisfying assignment; it is complete and sound because of the divisibility condition coupled with properties (1) and (2) above.

*The construction.* 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

[3] 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 our description by referring to "the size of the computation", which captures the field size.

number of constraints, or $|Z| + |\mathcal{C}|$.

*The queries and check.* To carry out the probabilistic check described above, $\mathcal{V}$ must obtain $H_{x,y,z}(\tau) \in \mathbb{F}$. 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 $\langle q_h, h \rangle = \sum_{i=0}^{|\mathcal{C}|} h_i \cdot \tau^i = H_{x,y,z}(\tau)$.

$\mathcal{V}$ must also obtain the value of $P(\tau, z) \in \mathbb{F}$. $P(t, Z)$ 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 $\{A_i(t), B_i(t), C_i(t)\}$ and $\{A'(t), B'(t), C'(t)\}$ (see Appendix A.1). Now, observe that if $P(t, Z)$ is evaluated at $t = \tau$, the result is $P(\tau, Z)$, a polynomial in $Z$ with the following 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 $\{q_a, q_b, q_c\}$ depend on $\tau$, and $\{L_a, L_b, L_c\}$ depend on $\tau, x$, and $y$. Thus, to obtain $P(\tau, z)$, $\mathcal{V}$ 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

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

Appendix A contains the full protocol and proofs of correctness. We now turn to costs.

## 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]; the reason for this choice is that Ginger is currently the most efficient verified computation scheme that is both a *system* and *general-purpose*. (See Section 6 for discussion of this assertion.)

**High-level 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 batch size at which the verifier breaks even). 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) come close to this degenerate behavior.[4]

Below, we present the details of the analysis. The comparison depends heavily on the number of constraints and variables in Zaatar versus the alternative, so we begin by carefully inspecting 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

[4] Also, the degenerate cases are detectable, so the compiler could simply choose to use Ginger (or [27, 55]) over Zaatar; this is ongoing work.

| | Ginger [54] | Zaatar |
|---|---|---|
| proof vector size | $\|u\| = \|Z\| + \|Z\|^2$ | $\|u'\| = \|Z'\| + \|\mathcal{C}'\| = 2 \cdot (\|Z\| + K_2)$ |
| worst case proof vector size | $\|u\| = \|Z\| + \|Z\|^2$ | $\|u'\| \leq \|u\| \cdot \left(1 + \frac{2}{\|Z\|+1}\right)$ |
| ***P*'s per-instance CPU costs** | | |
| Construct proof vector | $T + f \cdot \|Z\|^2$ | $T + 12f \cdot \|\mathcal{C}'\| \cdot \log^2 \|\mathcal{C}'\|$ |
| Issue responses | $(h + (\ell + 1) \cdot f) \cdot \|u\|$ | $(h + (\ell' + 1) \cdot f) \cdot \|u'\|$ |
| ***V*'s per-instance CPU costs** | | |
| Construct computation-specific queries | $(c \cdot \|\mathcal{C}\| + f \cdot K)/\beta$ | $(c + (f_{div} + 5f) \cdot \|\mathcal{C}'\| + f \cdot K + 3f \cdot K_2)/\beta$ |
| Construct computation-oblivious queries | $(e + 3c + (\ell + 1) \cdot f) \cdot \|u\|/\beta$ | $(e + 3c + \ell' \cdot f) \cdot \|u'\|/\beta$ |
| Process responses | $d + (2\ell + \|x\| + \|y\|) \cdot f$ | $d + (\ell' + 3\|x\| + 3\|y\|) \cdot f$ |

| | |
|---|---|
| $\|Z\|$: # of variables in Ginger constraint set for $\Psi$ (§2.1) | $\ell = 3$: # of (high-order) PCP queries in Ginger [54] |
| $\|\mathcal{C}\|$ $(= \|Z\|)$: # of constraints in Ginger constraint set for $\Psi$ (§4) | $\ell' = 5$: # of (total) PCP queries in Zaatar (§A.1) |
| $K$: # of additive terms in Ginger constraints for $\Psi$ (§4) | $\beta$: batch size, meaning # of instances (§2.2, §5.2) |
| $K_2$ $(\leq K)$: # of distinct additive degree-2 terms in Ginger constraints for $\Psi$ (§4) | $e$, $d$: cost of encrypting, decrypting an element in $\mathbb{F}$ (§5) |
| $T$: running time of $\Psi$ (§5.2) | $f$: cost of multiplying in $\mathbb{F}$ (§5) |
| $\|Z'\|$ $(= \|Z\| + K_2)$: # of variables in Zaatar constraints for $\Psi$ (§4) | $f_{div}$: cost of division in $\mathbb{F}$ (§5) |
| $\|\mathcal{C}'\|$ $(= \|Z\| + K_2)$: # of constraints in Zaatar constraints for $\Psi$ (§4) | $h$: cost of ciphertext add plus multiply (§5) |
| $\|x\|$, $\|y\|$: # of elements in input, output (§2.1) | $c$: cost to generate 192-bit pseudorandom # (§5) |

Figure 3—Costs to prover and verifier to verify a computation $\Psi$, 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. Key to the comparison is the term $K_2$, which is large only for degenerate computations (see text). The table assumes that the constraints $\mathcal{C}$ and $\mathcal{C}'$ have been generated by our compiler. *Computation-specific* queries refer to those that depend on the structure of the computation and its constraints $\mathcal{C}$, while *computation-oblivious* queries depend only on the length of the proof vector. The query and response rows include queries from both the commitment (§2.2) and PCP (§2.2, §3) protocols.

*quadratic form*; the requirement stems from the way that Zaatar encodes constraints in polynomials (see Appendix A.1).

We can obtain constraints $\mathcal{C}'$ in quadratic form, given a set of Ginger constraints $\mathcal{C}$; indeed, our compiler first compiles to Ginger constraints and then performs the following transformation. For every constraint in $\mathcal{C}$, we retain all of the degree-1 terms and replace all degree-2 terms with a new variable. For example, if a constraint in $\mathcal{C}$ is $\{3 \cdot Z_1 Z_2 + 2 \cdot Z_3 Z_4 + Z_5 - Z_6 = 0\}$, then $\mathcal{C}'$ would replace that constraint with the following constraints, which are all in quadratic form: $\{(3 \cdot Z'_1 + 2 \cdot Z'_2 + Z_5)(1) = Z_6,\ Z_1 Z_2 = Z'_1,\ Z_3 Z_4 = Z'_2\}$.

We can bound the number of variables and constraints in $\mathcal{C}'$ as follows. Letting $|Z'|$ equal the number of variables in $\mathcal{C}'(X = x, Y = y)$, we have from the construction of $\mathcal{C}'$ that $|Z'| = |Z| + K_2$, where $|Z|$ is the number of variables in $\mathcal{C}(X = x, Y = y)$ and $K_2$ is the number of distinct degree-2 terms in $\mathcal{C}$ that appear in all of the constraints in $\mathcal{C}$. Similarly, $|\mathcal{C}'| = |\mathcal{C}| + K_2$.

Many of the costs in Zaatar and Ginger scale with the length of the proof vector; see the $|u|$-versus-$|u'|$ substitution in some of the rows in Figure 3. For now, we observe that the length of the proof vector drops sharply in Zaatar in comparison to Ginger for generic realistic computations. We give a very careful analysis of this at the end of this section (summarized in the first two lines of Figure 3) and provide experimental validation in Section 5.

**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 $t$ (see Appendix A, and recall that these polynomials are

introduced in Section 3). Using a series of operations based on the fast Fourier transform (interpolation [41], polynomial multiplication [25], and polynomial division), the prover obtains the coefficients of $H_{x,y,z}(t)$ in time $\approx 12 \cdot f \cdot (|\mathcal{C}'| \cdot \log^2 |\mathcal{C}'|)$, as depicted in Figure 3. The process is detailed in Appendix A.3.

Empirically, the cost of this step can be substantial for the prover. However, the cost is outweighed by a vast reduction in query response costs in Zaatar (notably, the cryptographic costs are far lower under Zaatar), owing to the far smaller proof vector. Section 5 provides details.

**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 $\gamma_1$ and $\gamma_2$ 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 constraints, yielding amortized costs of $(c \cdot |\mathcal{C}| + f \cdot K)/\beta$ for a batch of size $\beta$. The analog under Zaatar is computing queries to $z$ and to $h$, which costs for the batch $c + (f_{div} + 5f) \cdot |\mathcal{C}'| + f \cdot K + 3f \cdot K_2$. The second cost is that Zaatar's verifier requires two extra passes over the input and output, owing to the details of query construction (see Appendix A.1).

Nevertheless, the savings from the smaller proof vector are so substantial that the Zaatar verifier has far-lower break-even batch sizes on the computations in our experiments, as we show in Section 5.

**Detailed analysis of $|u'|$ versus $|u|$.** For a given computation, Ginger's proof vector $u$ has length $|u| = |Z| + |Z|^2$ (per Section 2.2). By contrast, Zaatar's proof vector $u'$ has length $|u'| = |Z'| + |\mathcal{C}'|$ (per Section 3). However, $|\mathcal{C}| \approx |Z|$, and we will in fact take $|\mathcal{C}| = |Z|$: our compiler, when configured to output Ginger constraints, introduces roughly one variable for each constraint.[5] Recalling that $|Z'| = |Z| + K_2$ and $|\mathcal{C}'| = |\mathcal{C}| + K_2$, we get $|u'| = 2 \cdot (|Z| + K_2)$.

To compare $|u'|$ to $|u|$, we make three points. First, $|u'|$ is less than $|u|$ as long as $K_2 < K_2^* \stackrel{\text{def}}{=} (|Z|^2 - |Z|)/2$. 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 $(|Z|-1)/2$, and (b) our compiler produces a Ginger constraint with more than $(|Z|-1)/2$ terms only when compiling a program excerpt that involves summing many terms that are degree-2 (or higher).[6] If most of the constraints have this form, it means that most of the computation involved such sums, which is a degenerate case. An example is degree-2 polynomial evaluation, for which the Ginger encoding is actually very concise [53].

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

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, as the next section illustrates.

[5] A careful analysis of the compiler indicates that the actual bound is $|\mathcal{C}| \leq (1 + \delta) \cdot |Z|$, for $\delta = 4/(\log_2 |\mathbb{F}| + 3)$. However, our fields are large (§5), which is why the text treats $\delta$ as equal to 0.

[6] 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.

## 5 Experimental evaluation

Our experimental evaluation studies the effect of Zaatar's refinements on the end-to-end costs and the proof encoding (that is, the proof vector size).

**Computations and parameters.** We use the following benchmark computations: (a) Partitioning Around Medoids (PAM) clustering [56], (b) insertion sort, (c) root finding via bisection, (d) Floyd-Warshall all-pairs shortest paths [26], and (e) the Fannkuch benchmark.[7]

Our implementation (see below) uses ElGamal encryption [29] with 1024-bit keys; for pseudorandomness, we use the ChaCha stream cipher [18]. Most of our experiments use 32-bit signed integers as inputs with a 128-bit prime as the field modulus; for root finding via bisection, we use inputs that are rational numbers with 32-bit numerators and 5-bit denominators, and a 320-bit prime as the field modulus (see [54] for the representation of rational numbers).

### 5.1 Implementation, setup, and method

We implemented Zaatar in C++ (about 6000 lines, per [57]); the verifier and the prover run as separate processes, and they exchange data using OpenMPI [3]. Both the verifier and the prover are multi-threaded via OpenMP threads [4] and can offload their cryptographic operations to GPUs using CUDA [1]. 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 [57]), to translate computations written in SFDL [43] to constraints in quadratic form (§4).

We measure Zaatar's performance by running the verifier and the prover on separate machines, connected by a local network. Each machine is equipped with an Intel Xeon processor E5540 2.53 GHz with 48GB of RAM. For GPU experiments, we use NVIDIA Tesla M2070. Each GPU has 448 CUDA cores and 6GB of device memory.

We measure CPU time of the prover and of the verifier using `getrusage`. Each reported result is the mean of three experiments (and the standard deviations are within 8% of the means).

We use two baselines when we report results; the first baseline is the cost of doing local computation, and the second baseline is Ginger [54]. For the prover, we report the cost to compute the correct output of the computation and the cost to participate in the verification protocol. Since the verifier needs to batch-verify computations to offset its setup costs (see Section 2.2), we report the batch size at which the total cost to the verifier (i.e., the cost to construct queries and process responses; see Figure 3, Section 4) is less than the cost to execute the computation locally; we call this the *breakeven batch size*.

To estimate the parameters in our cost model (Figure 3), we perform microbenchmarks in our testbed. 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):

| field size | $e$ | $d$ | $h$ | $f$ | $f_{div}$ | $c$ |
|---|---|---|---|---|---|---|
| 128 bits | 72 $\mu$s | 170 $\mu$s | 190 $\mu$s | 18 ns | 0.59 $\mu$s | 69 ns |
| 320 bits | 280 $\mu$s | 170 $\mu$s | 560 $\mu$s | 34 ns | 0.72 $\mu$s | 207 ns |

We use these parameters as follows. For Ginger, we use a cost model from [54], parameterized with the estimates above, to estimate the end-to-end performance; we use estimates, rather than empirics, because the computations would be too expensive under Ginger. For Zaatar, we use the cost model in Figure 3, parameterized with the estimates above, to validate our experimental results; we find that our experimental results are about 5–15% more than the model's predictions (perhaps because of cache effects).

### 5.2 End-to-end performance

First, we compare the running time of the Zaatar prover to our two baselines (i.e., the Ginger prover and the cost of executing the computation locally); second, we compare the breakeven batch sizes under Zaatar

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

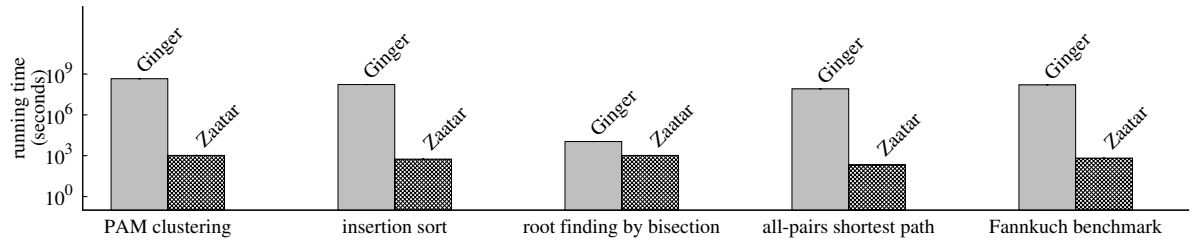



Figure 4—Per-instance running time of the prover under Zaatar and Ginger for various benchmark computations. Zaatar's theoretical refinements improve the running time by 1–6 orders of magnitude compared to the estimated costs of Ginger. The y-axis is log-scaled.

| | | prover's costs under Zaatar | | | | |
|---|---|---|---|---|---|---|
| computation ($\Psi$) | local | solve constraints | construct $u$ | crypto ops. | answer queries | e2e CPU time |
| PAM clustering | 59.0 ms | 8.2 s | 6.9 min | 15.4 min | 2.7 s | 22.5 min |
| insertion sort | 52.6 ms | 1.5 s | 4.9 min | 9.4 min | 2.1 s | 14.4 min |
| root finding by bisection | 88.2 ms | 5.7 s | 5.4 min | 16.1 min | 1.6 s | 21.7 min |
| all-pairs shortest path | 85.3 ms | 0.9 s | 2.4 min | 4.9 min | 1.1 s | 7.3 min |
| Fannkuch benchmark | 42.7 ms | 7.2 s | 4.6 min | 10.8 min | 2.0 s | 15.6 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. Zaatar's prover's end-to-end running time is far more than the cost to execute the computation; however, most of that work is in cryptographic operations so can be parallelized to reduce latency if the prover has access to parallel hardware (e.g., GPUs).

and under Ginger; third, we compare the performance of Zaatar and Ginger when we vary input sizes in the benchmark problems.

**Prover's end-to-end running time.** We report the end-to-end performance of Zaatar's and Ginger's provers with the following combinations of benchmark computations and input sizes: PAM clustering algorithm with 2000 data points (more specifically, $m = 20$ samples are clustered into two groups where each sample has $d = 100$ dimensions), insertion sort with $m = 256$ integers, an algorithm to find roots of a degree-2 polynomial with $m = 256$ variables via bisection (a kind of binary search) for $L = 8$ iterations, Floyd-Warshall's algorithm to compute shortest-paths between all pairs of nodes in a graph with $m = 25$ nodes, and the Fannkuch benchmark with $m = 100$ different permutations of $\{1, \ldots, 13\}$. For Zaatar, we run the experiments and report the data; for Ginger, we estimate (as stated in Section 5.1).

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 between 3–6 orders of magnitude. The root-finding computation, where the depicted difference is between one and two orders of magnitude, is a special case: it has a relatively efficient representation under Ginger. Also, as we increase $L$, 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 70% of the work done by the prover is in cryptographic operations and about 25% of the work is in computing the proof vectors.

Zaatar's prover and verifier can reduce their latency by exploiting parallel hardware (multiple cores, multiple machines, and GPUs) for their cryptographic operations. We have run experiments in which the verifier and the prover have access to multiple CPU cores and GPUs. Since our findings on reducing latency via parallelization replicate those of [54] (i.e., the prover achieves a near-linear reduction in latency for its cryptographic operations as it gets more hardware resources), we do not report them here.

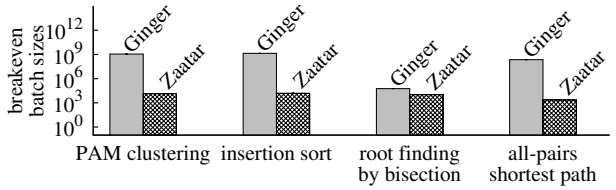



Figure 6—Breakeven batch sizes under Zaatar and Ginger; Zaatar's proof encoding reduces the amount of work done by the verifier, and hence the breakeven batch sizes reduce by several orders of magnitude versus in Ginger. The y-axis is log-scaled.

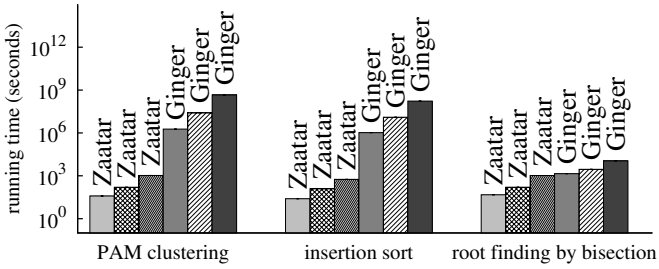



Figure 7—Running time of the prover under Zaatar and Ginger for various benchmark computations with varying input sizes (for each computation and each system, we double the input size twice and report three running times; text has details). Zaatar's prover's work scales linearly; Ginger's, quadratically. The y-axis is log-scaled.

**Breakeven batch sizes.** We compute Zaatar's breakeven batch size by comparing the cost of the verifier's Zaatar-related work for one instance of a computation to the cost of running the computation locally (see above). For Ginger, we estimate the breakeven batch sizes using its cost model (as described in Section 5.1). We perform this comparison with four benchmark problems. (We do not report the breakeven batch size for the Fannkuch benchmark because at the input sizes that we experimented with, there is no breakeven batch size, and our compiler does not yet scale well to larger input sizes.)

Figure 6 depicts the results; we find that 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. As in [54], assuming that the verifier uses a GPU for cryptographic operations reduces batch sizes by a factor of 2.

**Scalability.** We run Zaatar with a subset of our benchmarks and measure the prover's running time as we vary the input sizes: for PAM clustering, we run with $m = 5, m = 10, m = 20$ (and $d = 100$); for insertion sort and root finding by bisection, we run with $m = 64, 128$, and 256 ($L = 8$). (We run with a subset because as noted above our compiler does not yet scale well for other problems.) We also estimate Ginger's costs at these input sizes. Figure 7 depicts the prover's running time under different configurations. Zaatar scales better than Ginger owing to the superior asymptotics of the proof vectors.

### 5.3 Computation encodings

We analyze the size of computation encoding (i.e., the number of variables and constraints) as a function of input sizes for our benchmark problems when they are compiled into Zaatar's constraints and Ginger's constraints; we also compute the size of the proof vectors in Zaatar and Ginger. Figure 8 depicts these quantities along with the running time of the computation in big-O terms. The size of Zaatar's proof encoding is linear in the original running time of the computation and is significantly smaller than Ginger's.

| | | computation encoding | | | | proof encoding | |
|---|---|---|---|---|---|---|---|
| computation ($\Psi$) | $O(\cdot)$ | $\lvert Z\rvert$ | $\lvert Z'\rvert$ | $\lvert \mathcal{C}\rvert$ | $\lvert \mathcal{C}'\rvert$ | $\lvert u\rvert$ | $\lvert u'\rvert$ |
| PAM clustering | $O(m^2d)$ | $20m^2d$ | $60m^2d$ | $20m^2d$ | $60m^2d$ | $400m^4d^2$ | $120m^2d$ |
| insertion sort | $O(m^2)$ | $20m^2$ | $20m^2$ | $22m^2$ | $22m^2$ | $400m^4$ | $42m^2$ |
| root finding by bisection | $O(m^2L)$ | $2mL$ | $m^2L$ | $2mL$ | $m^2L$ | $4m^2L^2$ | $2m^2L$ |
| all-pairs shortest path | $O(m^3)$ | $48m^3$ | $48m^3$ | $52m^3$ | $52m^3$ | $2300m^6$ | $100m^3$ |

Figure 8—Zaatar's and Ginger's computation encodings (i.e., the number of variables and the number of constraints). For Ginger (resp., Zaatar), $|Z|$ (resp., $|Z'|$) denotes the number of variables, and $|\mathcal{C}|$ (resp., $|\mathcal{C}'|$) denotes the number of constraints. $m$ 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 [48, 53]); 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 [24, 30] employ fully homomorphic encryption (FHE) [32] in protocols for verifiable non-interactive computation. Unfortunately, despite advances in the efficiency of FHE [33], it remains highly impractical.

In a more plausible vein, several authors have worked on succinct arguments that rely neither on FHE nor PCPs [31, 35, 42]. Early work in this area [35, 42] supported only Boolean circuits and suffered from quadratic prover overhead. Recent work by Gennaro et al. [31] removes these limitations, and in fact, Zaatar extracts a seminal circuit encoding insight from this work. Note, however, that their protocol is substantially more complicated than Zaatar, as it is designed to avoid batching (and instead pays a one-time setup cost). Moreover, Gennaro et al. require the prover to compute over ciphertext so may in practice have a more expensive prover. Finally, there has been theoretical work that provides new foundations for system building [13, 14]. Systems based on these protocols are still in progress,[8] so it is not yet possible to perform meaningful practical comparisons.

The work most closely related to Zaatar is that on implementing interactive protocols for verified computation. Currently, there are two major implementation efforts, built on different foundations.

One of them is our line of work [52–54], which includes Pepper and Ginger, the basis for Zaatar. This line of work depends on the proposal of Ishai et al. [37], which coupled a classical linear PCP [8] with a linear commitment primitive to derive an efficient argument. Following a position paper [52], Pepper [53] strengthened this commitment primitive (Section 2.2), exploited batching, and applied the approach to arithmetic circuits. Subsequently, Ginger [54] demonstrated that Pepper's commitment protocol binds even a cheating prover to a linear function. Ginger also extends the computational model to the constraints in Section 2.1, and it implements the system on GPUs and with a distributed prover. As discussed earlier, Ginger achieves near-practical performance only for certain manually-tailored computations. Worse, it appears that not all computations can be tailored so. Thus, it is hard to regard Ginger as truly general-purpose.

The second strand is the protocols of Cormode et al. [27, 55], which are based on the work of [34]. These protocols do not need cryptography or batching, and have excellent performance for the verifier. However, results for the prover are more equivocal; measurements suggest comparable performance to Ginger [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).

[8] Personal communication.

## 7 Discussion and conclusion

Zaatar's main contribution is to produce a system for verified computation whose prover has good overhead—nearly linear—for all computations. This is in sharp contrast to the current literature on verified computation; in practice, this gain results in performance improvements of many orders of magnitude over the state-of-the-art implemented systems.

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 the protocol requires cryptographic operations.

Nevertheless, we expect assurance to have a price, and indeed, there are regimes in which Zaatar's costs are not utterly 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 Zaatar's verifier's batching requirement.

We note in closing that, several years ago, the theoretical underpinnings of Zaatar were thought to be laughably impractical. As a result of prior work and the work of this paper, 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 [37] protocol, which is used by Zaatar. (Recall that Zaatar's predecessors [37, 53, 54] use the linear PCP protocol of Arora et al. [8].) This protocol is based on Quadratic Arithmetic Programs (QAPs), a construction of Gennaro et al. [31].

Our description will be tailored to our context of verified computations. In particular, the PCP protocol will check the satisfiability of constraints that are assumed to represent a computation (§2.1). However, this description generalizes to checking the satisfiability of any degree-2 constraint set. Since degree-2 constraint satisfaction is an NP-complete problem [7], 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 $W_{n'+1}, \ldots, W_n$.

*Building blocks.* The building blocks described in the next several paragraphs are borrowed from QAPs [31], 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:

$$\begin{array}{lll}
A_0(t) = (a_{0,1}, a_{0,2}, \ldots, a_{0,|\mathcal{C}|}) & B_0(t) = (b_{0,1}, b_{0,2}, \ldots, b_{0,|\mathcal{C}|}) & C_0(t) = (c_{0,1}, c_{0,2}, \ldots, c_{0,|\mathcal{C}|}) \\
A_1(t) = (a_{1,1}, a_{1,2}, \ldots, a_{1,|\mathcal{C}|}) & B_1(t) = (b_{1,1}, b_{1,2}, \ldots, b_{1,|\mathcal{C}|}) & C_1(t) = (c_{1,1}, c_{1,2}, \ldots, c_{1,|\mathcal{C}|}) \\
\vdots & \vdots & \vdots \\
A_n(t) = (a_{n,1}, a_{n,2}, \ldots, a_{n,|\mathcal{C}|}) & B_n(t) = (b_{n,1}, b_{n,2}, \ldots, b_{n,|\mathcal{C}|}) & C_n(t) = (c_{n,1}, c_{n,2}, \ldots, c_{n,|\mathcal{C}|})
\end{array}$$

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 $W$, as follows:

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

We write $P_{x,y}(t, Z)$ to mean $P(t, W)$ with $X = x$ and $Y = y$. 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 $w = (x, y, z) \in \mathbb{F}^n$ be an assignment to the variables $(X, Y, Z)$, and let $w_0 = 1$. For $P_w(t)$ defined as follows, $D(t)$ divides $P_w(t)$ if and only if $z$ satisfies $\mathcal{C}(X = x, Y = y)$:

$$P_w(t) \stackrel{\text{def}}{=} P(t, w) = \left(\sum_{i=0}^{n} w_i \cdot A_i(t)\right) \cdot \left(\sum_{i=0}^{n} w_i \cdot B_i(t)\right) - \left(\sum_{i=0}^{n} w_i \cdot C_i(t)\right) = P_{x,y}(t, z).$$

*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 $\sigma_j$ 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 $(\sum_{i=1}^{n} w_i \cdot a_{i,j} + a_{0,j}) \cdot (\sum_{i=1}^{n} w_i \cdot b_{i,j} + b_{0,j}) = \sum_{i=1}^{n} w_i \cdot c_{i,j} + c_{0,j}$. That is, constraint $j$ is satisfied at $w = (x, y, z)$, by definition of $a_{i,j}$, $b_{i,j}$, and $c_{i,j}$. But we chose $j$ arbitrarily, so all constraints are satisfied, and hence $z$ satisfies $\mathcal{C}(X = x, Y = y)$.

For the other direction, if the $z$ "piece" of $w$ satisfies $\mathcal{C}(X = x, Y = y)$, then every constraint is satisfied. This implies that $P_w(\sigma_j) = 0$ 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 $D(t) \cdot H(t) = P_w(t)$. We sometimes write the proof vector $u$ as $(z, h)$.

A linear PCP based on QAPs [31]

The verifier $\mathcal{V}$ interacts with a proof oracle $\pi$ 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 $\rho$ times:

- Generate linearity queries: Select $q_1, q_2 \in_R \mathbb{F}^{n'}$ and $q_4, q_5 \in_R \mathbb{F}^{|\mathcal{C}|+1}$. Take $q_3 \leftarrow q_1 + q_2$ and $q_6 \leftarrow q_4 + q_5$.
- Generate self-correction queries: Generate $q_s \in_R \mathbb{F}^{\max\{n', |\mathcal{C}|+1\}}$, and take $q_7 \leftarrow q_s[1 : n']$, $q_8 \leftarrow q_s[1 : |\mathcal{C}| + 1]$. The notation $v[a : b]$ means the vector $v$ at indices $a$ through $b$. (Self-correction queries can be reused soundly; see [54, Apdx. A.3] and [12].)
- 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 $q_9 \leftarrow (q_a + q_7)$.
  - Take $q_b \leftarrow (B_1(\tau), B_2(\tau), \ldots, B_{n'}(\tau))$, and $q_{10} \leftarrow (q_b + q_7)$.
  - Take $q_c \leftarrow (C_1(\tau), C_2(\tau), \ldots, C_{n'}(\tau))$, and $q_{11} \leftarrow (q_c + q_7)$.
  - Take $q_d \leftarrow (1, \tau, \tau^2, \ldots, \tau^{|\mathcal{C}|})$, and $q_{12} \leftarrow (q_d + q_8)$.
- Issue queries: Send $q_1, \ldots, q_{12}$ to oracle $\pi$, getting back $\pi(q_1), \ldots, \pi(q_{12})$.
- Linearity tests: Check that $\pi(q_1) + \pi(q_2) = \pi(q_3)$ and that $\pi(q_4) + \pi(q_5) = \pi(q_6)$. If not, `reject`.
- Divisibility correction test: Return `reject` unless

$$D(\tau) \cdot (\pi(q_{12}) - \pi(q_8)) = \left(\pi(q_9) - \pi(q_7) + \sum_{i=n'+1}^{n} w_i \cdot A_i(\tau) + A_0(\tau)\right) \cdot \left(\pi(q_{10}) - \pi(q_7) + \sum_{i=n'+1}^{n} w_i \cdot B_i(\tau) + B_0(\tau)\right) - \left(\pi(q_{11}) - \pi(q_7) + \sum_{i=n'+1}^{n} w_i \cdot C_i(\tau) + C_0(\tau)\right).$$

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

Figure 9—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 $\{w_{n'+1}, \ldots, w_n\}$. The purpose of queries $q_7$ and $q_8$ is self-correction (see [8, §5] or [47, §7.8.3]). Because Zaatar composes this PCP with linear commitment, it sets $\rho = 1$ and does not include the linearity queries and tests [54].

**The PCP protocol.** The protocol is depicted in Figure 9. Its soundness is 1/8, as established in Section A.2. We state the high-order verifier costs immediately below and give details in Section A.3.

The verifier's running time is proportional to $|u| = |Z| + |\mathcal{C}|$ and so is the randomness that it must generate; the constant of proportionality depends on the cost of field operations. These costs are almost linear in the number of steps in the computation; there is actually an extra logarithmic term (since the field size, which must be larger than the number of steps in the computation, affects the costs). However, to reflect the practical reality, we mostly disregard this term by referring to "the size of the computation", which captures the field size. Of course, costs that are proportional to the computation itself do not save the verifier work; thus, we amortize these costs over multiple instances of the computation (§2.2), in the context of the efficient argument system described immediately below.

**The efficient argument system.** Zaatar is an efficient argument system [37, 40] that composes the PCP in Figure 9 with Ginger's linear commitment primitive [53, 54] (which strengthens a primitive of Ishai et al. [37]). Although the stand-alone PCP includes looping and a linearity test [20], neither is included in Zaatar, owing to the strength of linear commitment [54]. The soundness of the argument system is proportional to $\sqrt[6]{1/|\mathbb{F}|}$. A detailed analysis is given in [54, Apdx. A.2].[9]

[9]To adapt that analysis, one should replace the PCP's soundness of $1 - (4\delta^* + 2/|\mathbb{F}|)$ with $1 - (8\delta^* + 2 \cdot |\mathcal{C}|/|\mathbb{F}|)$.

The verifier incurs additional costs from the linear commitment primitive; these costs are proportional to the computation size. The prover's costs stem from constructing the proof vector (treated in Section A.3) plus responding to queries (which induces costs proportional to the size of the computation).

### A.2 Correctness

A verifier $\mathcal{V}$ is given access to a proof oracle $\pi$, which is supposed to establish the satisfiability of $\mathcal{C}(X = x, Y = y)$.

**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 9, then $\Pr\{\mathcal{V} \text{ accepts}\} = 1$.

*Proof.* If $\pi$ is constructed properly, then it is a linear function, so it passes the linearity test. Next we consider the divisibility test. Let $w = (x, y, z)$. If $\mathcal{C}(X = x, Y = y)$ is satisfiable by $z$, then by Claim A.1, there exists $H_w(t)$ such that $D(t) \cdot H_w(t) = P_w(t)$. Also, if $\pi$ is constructed properly, then $\mathcal{V}$ obtains $\pi_z(q_a)$ as $\pi_z(q_9) - \pi_z(q_7)$, where $\pi_z(q_a) = \sum_{i=1}^{n'} w_i \cdot A_i(\tau)$. Similarly, $\mathcal{V}$ obtains $\pi_z(q_b)$ as $\pi_z(q_{10}) - \pi_z(q_7)$, where $\pi_z(q_b) = \sum_{i=1}^{n'} w_i \cdot B_i(\tau)$. Likewise $\pi_z(q_{11}) - \pi_z(q_7) = \pi_z(q_c) = \sum_{i=1}^{n'} w_i \cdot C_i(\tau)$. Finally, $\mathcal{V}$ obtains $\pi_h(q_d)$ similarly, where $\pi_h(q_d) = \sum_{j=0}^{|\mathcal{C}|} h_j \cdot \tau^j = H_w(\tau)$. Thus, the divisibility test is checking the following (let $w_0 = 1$):

$$D(\tau) \cdot H_w(\tau) \stackrel{?}{=} \left(\sum_{i=0}^{n} w_i \cdot A_i(\tau)\right) \cdot \left(\sum_{i=0}^{n} w_i \cdot B_i(\tau)\right) - \left(\sum_{i=0}^{n} w_i \cdot C_i(\tau)\right) = P_w(\tau), \tag{1}$$

which holds, since $D(t) \cdot H_w(t) = P_w(t)$. Thus, both tests pass, so $\mathcal{V}$ accepts. □

**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 9, then $\Pr\{\mathcal{V} \text{ accepts}\} < \kappa$ for *all* purported proof oracles $\tilde{\pi}$.

*Proof.* Assume for now that $\tilde{\pi}$ is a linear function (this restricts the proof oracle's power to cheat, and we will revisit this in a moment). Consider $\tilde{\pi}(q_a)$, $\tilde{\pi}(q_b)$, and $\tilde{\pi}(q_c)$; these equal $\langle q_a, \tilde{z}\rangle$, $\langle q_b, \tilde{z}\rangle$, and $\langle q_c, \tilde{z}\rangle$, for some $\tilde{z}$ chosen by the prover. Likewise, $\tilde{\pi}(q_d)$ equals $\widetilde{H}(\tau)$, for some polynomial $\widetilde{H}(t)$, chosen by the prover. Thus, the divisibility test is checking $D(\tau) \cdot \widetilde{H}(\tau) \stackrel{?}{=} P_{\tilde{w}}(\tau)$, for $\tilde{w} = (x, y, \tilde{z})$.

However, $\mathcal{C}(X = x, Y = y)$ is not satisfiable, so there is no $\widetilde{H}(t)$ for which $D(t) \cdot \widetilde{H}(t) = P_{\tilde{w}}(t)$ (by Claim A.1). Thus, Equation (1) holds only if $\tau$ is a root of the polynomial $Q(t) = P_{\tilde{w}}(t) - D(t) \cdot \widetilde{H}(t)$. But $\Pr_\tau\{Q(\tau) = 0\} \leq 2 \cdot |\mathcal{C}|/|\mathbb{F}|$, since the degree of $Q(t)$ is bounded by $2 \cdot |\mathcal{C}|$ and $\tau$ is randomly chosen from $\mathbb{F}$.

We now address the possibility that $\tilde{\pi}$ is not a linear function. Take $\kappa > \max\{7/8,\ 8\delta + 2 \cdot |\mathcal{C}|/|\mathbb{F}|\}$ and $\delta = 1/20$. Also, take $\mathbb{F}$ to be large enough that $7/8$ is the larger of the two terms in the max.

**Claim A.4.** If for some $\pi$, the tests pass with probability greater than $\kappa$, then $\mathcal{C}(X = x, Y = y)$ is satisfiable.

*Proof.* If the linearity test passes with probability $> 7/8$, then the linear function is $\delta$-close to linear, where $\delta = 1/20$. This follows from results of Bellare et al. [11, 12]; for similar uses of these results, see Setty et al. [53, Apdx. D][54, Apdx. A]. Next, consider the divisibility correction test (DCT). Assuming the proof oracle is $\delta$-close to linear, the probability of passing the DCT is $> 8\delta + 2 \cdot |\mathcal{C}|/|\mathbb{F}|$, by the given. But the probability that any of the eight queries in this test $(q_7, q_9, q_{10}, \ldots)$ "hit" $\pi$ where it is not linear is upper-bounded by $8\delta$, by the union bound. So with probability $> 2 \cdot |\mathcal{C}|/|\mathbb{F}|$, the tests pass when they query the closest linear function to $\pi$. But if that is the case, then $\mathcal{C}(X = x, Y = y)$ is satisfiable because, as argued above, if $\mathcal{C}(X = x, Y = y)$ is not satisfiable and $\mathcal{V}$ queries a linear function, the probability of passing the tests $\leq 2 \cdot |\mathcal{C}|/|\mathbb{F}|$. □

By Claim A.4, if $\mathcal{C}(X = x, Y = y)$ is not satisfiable, then $\Pr\{\text{tests pass}\} < \kappa$, for all $\tilde{\pi}$, so we have the lemma. □

One bookkeeping detail: Lemma A.3 requires us to take the field large enough so that $2 \cdot |\mathcal{C}|/|\mathbb{F}| \leq 7/8 - 8/20 \approx 1/4$. In practice, this dependency is irrelevant: $|\mathcal{C}|$ roughly captures the size of the computation, and $|\mathbb{F}|$ is astronomical (e.g., $|\mathbb{F}| = 2^{192}$), so their ratio can be neglected.

### 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. [31] 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.** Recall that to construct the $\pi_h$ 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 prover's cost to do so is $12 \cdot f \cdot |\mathcal{C}| \cdot \log^2 |\mathcal{C}|$. We now detail the process. It is three steps (which are well-explained in [45, 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 $\{(\sigma_j, \sum_i z_i \cdot a_{i,j}) \mid j \in \{0, \ldots, |\mathcal{C}|\}\}$, which is the evaluations of $A(t)$. The prover then uses multipoint interpolation [41, Chapter 4.6.4] to compute the coefficients of $A(t)$, in time $\approx 4 \cdot f \cdot |\mathcal{C}| \cdot \log^2 |\mathcal{C}|$. The prover does likewise for $B(t)$ and $C(t)$. Step 2 is computing the coefficients of $P_w(t)$ in time $\approx 4 \cdot f \cdot |\mathcal{C}| \cdot \log |\mathcal{C}|$, using multiplication based on the fast Fourier transform (FFT) [25]. Step 3 is computing the coefficients of $P_w(t)/D(t) = H_w(t)$ in time $\approx 4 \cdot f \cdot |\mathcal{C}| \cdot \log |\mathcal{C}|$, using FFT-based polynomial division.

**The verifier.** We will be focused on Figure 9. 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 $\tau$ 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. [31] observe that a Lagrange basis [21, Chap. 3] is useful for this purpose, but they do not give all of the details; we do so here. We can write each polynomial $A_i(t)$ as follows (and analogously for $B_i(t), C_i(t)$):

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

We can use Barycentric Lagrange interpolation [19] to write:

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

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* values here as long as they are distinct and non-zero). 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 passes over the 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 $\sum_{n'+1}^{n} w_i \cdot C_i(\tau)$.

## References


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

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

[3] Open MPI (http://www.open-mpi.org).

[4] OpenMP. `http://openmp.org/`.

[5] S. Alsouri, S. Katzenbeisser, and S. Biedermann. Trustable outsourcing of business processes to cloud computing environments. In *The Conference on Network and System Security (NSS)*, 2011.

[6] 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.

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

[8] 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.

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

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

[11] M. Bellare, D. Coppersmith, J. Håstad, M. Kiwi, and M. Sudan. Linearity testing in characteristic two. *IEEE Transactions on Information Theory*, 42(6):1781–1795, Nov. 1996.

[12] M. Bellare, S. Goldwasser, C. Lund, and A. Russell. Efficient probabilistically checkable proofs and applications to approximations. In *STOC*, 1993.

[13] E. Ben-Sasson, A. Chiesa, D. Genkin, and E. Tromer. Fast reductions from RAMs to delegatable succinct constraint satisfaction problems. Feb. 2012. Cryptology ePrint 071.

[14] E. Ben-Sasson, A. Chiesa, D. Genkin, and E. Tromer. On the concrete-efficiency threshold of probabilistically-checkable proofs. *ECCC*, (045), Apr. 2012.

[15] 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.

[16] 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.

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

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

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

[20] M. Blum, M. Luby, and R. Rubinfeld. Self-testing/correcting with applications to numerical problems. *J. of Comp. and Sys. Sciences*, 47(3):549–595, Dec. 1993.

[21] R. L. Burden and J. D. Faires. *Numerical Analysis*. Thomson Brooks/Cole, eighth edition, 2005.

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

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

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

[25] 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.

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

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

[28] I. Dinur. The PCP theorem by gap amplification. *J. of the ACM*, 54(3), June 2007.
[29] T. ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Trans. on Info. Theory*, 31:469–472, 1985.
[30] R. Gennaro, C. Gentry, and B. Parno. Non-interactive verifiable computing: Outsourcing computation to untrusted workers. In *CRYPTO*, 2010.
[31] R. Gennaro, C. Gentry, B. Parno, and M. Raykova. Quadratic span programs and succinct NIZKs without PCPs. Apr. 2012. Cryptology ePrint 215.
[32] C. Gentry. *A fully homomorphic encryption scheme*. PhD thesis, Stanford University, 2009.
[33] C. Gentry, S. Halevi, and N. Smart. Homomorphic evaluation of the AES circuit. In *CRYPTO*, 2012.
[34] S. Goldwasser, Y. T. Kalai, and G. N. Rothblum. Delegating computation: Interactive proofs for muggles. In *STOC*, 2008.
[35] J. Groth. Short pairing-based non-interactive zero-knowledge arguments. In *ASIACRYPT*, 2010.
[36] A. Haeberlen, P. Kouznetsov, and P. Druschel. PeerReview: Practical accountability for distributed systems. In *SOSP*, 2007.
[37] Y. Ishai, E. Kushilevitz, and R. Ostrovsky. Efficient arguments without short PCPs. In *Conference on Computational Complexity (CCC)*, 2007.
[38] S. Jha, S. Katzenbeisser, C. Schallhart, H. Veith, and S. Chenney. Semantic integrity in large-scale online simulations. *ACM Transactions on Internet Technology (TOIT)*, 10(1), Feb. 2010.
[39] G. O. Karame, M. Strasser, and S. Capkun. Secure remote execution of sequential computations. In *Intl. Conf. on Information and Communications Security*, 2009.
[40] J. Kilian. A note on efficient zero-knowledge proofs and arguments (extended abstract). In *STOC*, 1992.
[41] D. E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming*. Addison-Wesley, third edition, 1997.
[42] H. Lipmaa. Progression-free sets and sublinear pairing-based non-interactive zero-knowledge arguments. In *IACR TCC*, 2011.
[43] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella. Fairplay—a secure two-party computation system. In *USENIX Security*, 2004.
[44] D. Malkhi and M. Reiter. Byzantine quorum systems. *Distributed Computing*, 11(4):203–213, 1998.
[45] T. Mateer. *Fast Fourier Transform algorithms with applications*. PhD thesis, Clemson University, 2008.
[46] F. Monrose, P. Wycko, and A. D. Rubin. Distributed execution with remote audit. In *NDSS*, 1999.
[47] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.
[48] B. Parno, J. M. McCune, and A. Perrig. *Bootstrapping Trust in Modern Computers*. Springer, 2011.
[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*, June 2010.
[50] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn. Design and implementation of a TCG-based integrity measurement architecture. In *USENIX Security*, 2004.
[51] 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.
[52] S. Setty, A. J. Blumberg, and M. Walfish. Toward practical and unconditional verification of remote computations. In *HotOS*, 2011.
[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*, Aug. 2012.
[55] J. Thaler, M. Roberts, M. Mitzenmacher, and H. Pfister. Verifiable computation with massively parallel interactive proofs. In *USENIX HotCloud Workshop*, June 2012.
[56] S. Theodoridis and K. Koutroumbas. *Pattern Recognition, Third Edition*. Academic Press, Inc., 2006.
[57] D. A. Wheeler. SLOCCount. `http://www.dwheeler.com/sloccount/`.