Taking proof-based verified computation a few steps closer to practicality[1]

Srinath Setty, Victor Vu, Nikhil Panpalia, Benjamin Braun, Andrew J. Blumberg, and Michael Walfish
The University of Texas at Austin

Abstract. We describe GINGER, a built system for unconditional, general-purpose, and nearly practical verification of outsourced computation. GINGER is based on PEPPER, which uses the PCP theorem and cryptographic techniques to implement an *efficient argument* system (a kind of interactive protocol). GINGER slashes query costs via protocol refinements; broadens the computational model to include (primitive) floating-point fractions, inequality comparisons, logical operations, and conditional control flow; and includes a parallel GPU-based implementation that dramatically reduces latency.

1 Introduction

We are motivated by *outsourced computing*: cloud computing (in which clients outsource computations to remote computers), peer-to-peer computing (in which peers outsource storage and computation to each other), volunteer computing (in which projects outsource computations to volunteers' desktops), etc.

Our goal is to build a system that lets a client outsource computation verifiably. The client should be able to send a description of a computation and the input to a server, and receive back the result together with some auxiliary information that lets the client *verify* that the result is correct. For this to be sensible, the verification must be faster than executing the computation locally.

Ideally, we would like such a system to be *unconditional*, *general-purpose*, and *practical*. That is, we don't want to make assumptions about the server (trusted hardware, independent failures of replicas, etc.), we want a setup that works for a broad range of computations, and we want the system to be usable by real people for real computations in the near future.

In principle, the first two properties above have been achievable for almost thirty years, using powerful tools from complexity theory and cryptography. Interactive proofs (IPs) and probabilistically checkable proofs (PCPs) show how one entity (usually called the *verifier*) can be convinced by another (usually called the *prover*) of a given mathematical assertion—without the verifier having to fully inspect a proof [5, 6, 20, 32]. In our context, the mathematical assertion is that a given computation was carried out correctly; though the proof is as long as the computation, the theory implies—surprisingly—that the verifier need only inspect the proof in a small number of randomly-chosen locations or query the prover a relatively small number of times.

The rub has been the third property: practicality. These protocols have required expensive encoding of computations, monstrously large proofs, high error bounds, prohibitive overhead for the prover, and intricate constructions that make the asymptotically efficient schemes challenging to implement correctly.

However, a line of recent work indicates that approaches based on IPs and PCPs are closer to practicality than previously thought [22, 47, 48, 53]. More generally, there has been a groundswell of work that aims for potentially practical verifiable outsourced computation, using theoretical tools [11, 12, 21, 25, 26].

Nonetheless, these works have notable limitations. Only a handful [22, 47, 48, 53] have produced working implementations, all of which impose high costs on the verifier and prover. Moreover, their model of computation is *arithmetic circuits* over finite fields, which represent non-integers awkwardly, control flow inefficiently, and comparisons and logical operations only by degenerating to verbose *Boolean* circuits. Arithmetic circuits are well-suited to integer computations and numerical straight line computations (e.g., multiplying matrices and computing second moments), but the intersection of these two domains leaves few realistic applications.

This paper describes a built system, called GINGER, that addresses these problems, thereby taking general-purpose proof-based verified computation several steps closer to practicality. GINGER is an *efficient argument* system [37, 38]: an interactive proof system that assumes the prover to be computationally bounded. Its starting point is the PEPPER protocol [48] (which is summarized in Section 2). GINGER's contributions are as follows.

(1) GINGER *slashes query costs (§3).* GINGER reduces costs by trading off more queries that are cheap for fewer of a more expensive type (keeping the total number of queries roughly the same); the justification for the soundness of this trade is rooted in a careful revisiting of the PCP's soundness and its sources of overhead. GINGER also slashes network costs by orders of magnitude, by compressing queries.

(2) GINGER *supports a general-purpose programming model (§4).* Although the model does not handle looping concisely, it includes primitive floating-point quantities, inequality comparisons, logical expressions, and conditional control flow. Moreover, we have a compiler (derived from Fairplay [40]) that transforms computations expressed in a general-purpose language to an executable verifier and prover. The core technical challenge is representing computations as additions and multiplications over a finite field (as required by the verification proto-

[1] This version revises the published paper to eliminate an incorrect theoretical claim. We thank Alessandro Chiesa, Yuval Ishai, Nir Bitanksy, and Omer Paneth for noticing the error and bringing it to our attention.

col); for instance, "not equal" and "if/else" do not obviously map to this formalism, inequalities are problematic because finite fields are not ordered, and fractions compound the difficulties. GINGER overcomes these challenges with techniques that, while not deep, require care and detail.[2] These techniques should apply to other protocols that use arithmetic constraints or circuits.

(3) GINGER *exploits parallelism to slash latency (§5).* The prover can be distributed across machines, and some of its functions are implemented in graphics hardware (GPUs). Moreover, GINGER's *verifier* can use a GPU for its cryptographic operations. Allowing the verifier to have a GPU models the present (many computers have GPUs) and a plausible future in which specialized hardware for cryptographic operations is common.[3]

We have implemented and evaluated GINGER (§6). Compared to PEPPER [48], its base, GINGER lowers network costs by 1–2 orders of magnitude (to hundreds of KB or less in our experiments). The verifier's costs drop by multiples, depending on the cost of encryption; if we model encryption as free, the verifier can gain from outsourcing when batch-verifying 740 computations (down from 2800 in PEPPER). The prover's CPU costs drop by about a factor of 2, and our parallel implementation reduces latency with near-linear speedup. Computing with rational numbers in GINGER is roughly two times more expensive than computing with integers, and arithmetic constraints permit far smaller representations than a naive use of Boolean or arithmetic circuits.

Despite all of the above, GINGER is not quite ready for the big leagues. However, PEPPER and GINGER have made argument systems far more practical (in some cases improving costs by 20 or more orders of magnitude over a naive implementation). We are thus optimistic about ultimately achieving true practicality.

2 Problem statement and background

Problem statement. A computer V, known as the *verifier*, has a computation Ψ and some desired input x that it wants a computer P, known as the *prover*, to perform. P returns y, the purported output of the computation, and then V and P conduct an efficient interaction. This interaction should be cheaper for V than locally computing $\Psi(x)$. Furthermore, if P returned the correct answer, it should be able to convince V of that fact; otherwise, V should be able to reject the answer as incorrect, with high probability. (The converse will not hold: rejection does not imply that P returned incorrect output, only that it misbehaved somehow.) Our goal is that this guarantee be *unconditional*: it should hold regardless of whether P obeys the protocol (given standard cryptographic assumptions about P's computational power). If P deviates from the protocol at any point (computing incorrectly, proving incorrectly, etc.), we call it *malicious*.

2.1 Tools

In principle, we can meet our goal using PCPs. The PCP theorem [5, 6] says that if a set of constraints is satisfiable (see below), there exists a *probabilistically checkable* proof (a PCP) and a verification procedure that accepts the proof after querying it in only a small number of locations. On the other hand, if the constraints cannot be satisfied, then the verification procedure rejects *any* purported proof, with probability at least $1 - \epsilon$.

To apply the theorem, we represent the computation as a set of quadratic constraints over a finite field. A *quadratic constraint* is an equation of total degree 2 that uses additions and multiplications (e.g., $A \cdot Z_1 + Z_2 - Z_3 \cdot Z_4 = 0$). A set of constraints is *satisfiable* if the variables can be set to make all of the equations hold simultaneously; such an assignment is called a *satisfying assignment*. In our context, a set of constraints $\mathcal{C}$ will have a designated input variable X and output variable Y (this generalizes to multiple inputs and outputs), and $\mathcal{C}(X = x, Y = y)$ denotes $\mathcal{C}$ with variable X bound to x and Y bound to y.

We say that a set of constraints $\mathcal{C}$ is *equivalent* to a desired computation Ψ if: for all x, y, $\mathcal{C}(X = x, Y = y)$ is satisfiable if and only if $y = \Psi(x)$. As a simple example, increment-by-1 is equivalent to the constraint set $\{Y = Z + 1, Z = X\}$. (For convenience, we will sometimes refer to a given input x and purported output y implicitly in statements such as, "If constraints $\mathcal{C}$ are satisfiable, then Ψ executed correctly".) To verify a computation $y = \Psi(x)$, one could in principle apply the PCP theorem to the constraints $\mathcal{C}(X = x, Y = y)$.

Unfortunately, PCPs are too large to be transferred. However, if we assume a computational bound on the prover P, then *efficient arguments* apply [37, 38]: V issues its PCP queries to P (so V need not receive the entire PCP). For this to work, P must commit to the PCP *before* seeing V's queries, thereby simulating a fixed proof whose contents are independent of the queries. V thus extracts a cryptographic commitment to the PCP (e.g., with a collision-resistant hash tree [42]) and verifies that P's query responses are consistent with the commitment.

This approach can be taken a step further: not even P has to materialize the entire PCP. As Ishai et al. [35] observe, in some PCP constructions, which they call *linear PCPs*, the PCP itself is a linear function: the verifier submits queries to the function, and the function's outputs serve as the PCP responses. Ishai et al. thus design a *linear commitment primitive* in which P can commit to

[2] We elide some of these details for space; they are documented in a longer version of this paper [49].

[3] One may wonder why, if the verifier has this hardware, it needs to outsource. GPUs are amenable only to certain computations (which include the cryptographic underpinnings of GINGER).

a linear function (the PCP) and V can submit function inputs (the PCP queries) to P, getting back outputs (the PCP responses) as if P itself were a fixed function.

PEPPER [48] refines and implements the outline above. In the rest of the section, we summarize the linear PCPs that PEPPER incorporates, give an overview of PEPPER, and provide formal definitions. Additional details are in Appendix A.1.

2.2 Linear PCPs, applied to verifying computations

Imagine that V has a desired computation Ψ and desired input x, and somehow obtains purported output y. To use PCP machinery to check whether $y = \Psi(x)$, V compiles Ψ into equivalent constraints $\mathcal{C}$, and then asks whether $\mathcal{C}(X = x, Y = y)$ is satisfiable, by consulting an *oracle* π: a fixed function (that depends on $\mathcal{C}, x, y$) that V can query. A *correct* oracle π is the proof (or PCP); V should accept a correct oracle and reject an incorrect one.

A correct oracle π has three properties. First, π is a *linear function*, meaning that $\pi(a)+\pi(b) = \pi(a+b)$ for all a, b in the domain of π. A linear function $\pi\colon \mathbb{F}^n \to \mathbb{F}$ is determined by a vector w; i.e., $\pi(a) = \langle a, w\rangle$ for all $a \in \mathbb{F}^n$. Here, $\mathbb{F}$ is a finite field, and $\langle a, b\rangle$ denotes the inner (dot) product of two vectors a and b. The parameter n is the size of w; in general, n is quadratic in the number of variables in $\mathcal{C}$ [5], but we can sometimes tailor the encoding of w to make n smaller [48].

Second, one set of the entries in w must be a redundant encoding of the other entries. Third, w encodes the actual satisfying assignment to $\mathcal{C}(X = x, Y = y)$.

A surprising aspect of PCPs is that each of these properties can be tested by making a small number of queries to π; if π is constructed incorrectly, the probability that the tests pass is upper-bounded by $\epsilon > 0$. For instance, a key test—we return to it in Section 3—is the *linearity test* [17]: V randomly selects q_1 and q_2 from $\mathbb{F}^n$ and checks if $\pi(q_1)+\pi(q_2) = \pi(q_1+q_2)$. The other two PCP tests are the *quadratic correction test* and the *circuit test*.

The completeness and soundness properties of linear PCPs are defined in Section 2.4. A detailed explanation of why the mechanics above satisfy those properties is outside our scope but can be found in [5, 13, 35, 48].

2.3 Our base: PEPPER

We now walk through the three phases of PEPPER [48], which is depicted in Figure 1. The approach is to compose a *linear PCP* and a *linear commitment primitive* that forces the prover to act like an oracle.

Specify and compute. V transforms its desired computation, Ψ, into a set of equivalent constraints, $\mathcal{C}$. V sends Ψ (or $\mathcal{C}$) to P, or P may come with them installed.

To gain from outsourcing, V must amortize the costs of compiling Ψ to $\mathcal{C}$ and generating queries. Thus, V verifies computations in batches [48] (although they need not be

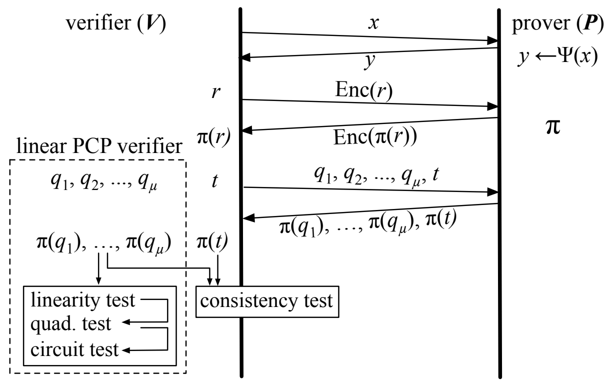

Figure 1—The PEPPER protocol [48], which is GINGER's base. Though not depicted, many of the protocol steps happen in parallel, to facilitate batching.

executed in a batch). A *batch* (of size β) refers to a set of computations in which Ψ is the same but the inputs are different; a member of the batch is called an *instance*. In the protocol, V has inputs $x_1, \ldots, x_\beta$ that it sends to P (not necessarily all at once), which returns $y_1, \ldots, y_\beta$; for each instance i, y_i is supposed to equal $\Psi(x_i)$.

For each instance i, an honest P stores a proof vector w_i that encodes a satisfying assignment to $\mathcal{C}(X = x_i, Y = y_i)$; w_i is constructed as described in Section 2.2. Being a vector, w_i can also be regarded as a linear function π_i—or an oracle of the kind described above.

Extract commitment. V cannot inspect $\{\pi_i\}$ directly (they are functions; written out, they would have an entry for each value in a huge domain). Instead, V extracts a *commitment* to each π_i. To do so, V randomly generates a *commitment vector* $r \in \mathbb{F}^n$. V then homomorphically encrypts each entry of r under a public key pk to get a vector $\mathsf{Enc}(pk, r) = (\mathsf{Enc}(pk, r_1), \mathsf{Enc}(pk, r_2), \ldots, \mathsf{Enc}(pk, r_n))$. We emphasize that $\mathsf{Enc}(\cdot)$ need not be fully homomorphic encryption [28] (which remains unfeasibly expensive); PEPPER uses ElGamal [24, 48].

V sends $(\mathsf{Enc}(pk, r), pk)$ to P. If P is honest, then π_i is linear, so P can use the homomorphism of $\mathsf{Enc}(\cdot)$ to compute $\mathsf{Enc}(pk, \pi_i(r))$ from $\mathsf{Enc}(pk, r)$, without learning r. P replies with $(\mathsf{Enc}(pk, \pi_1(r)), \ldots, \mathsf{Enc}(pk, \pi_\beta(r)))$, which is P's commitment to $\{\pi_i\}$. V then decrypts to get $(\pi_1(r), \ldots, \pi_\beta(r))$.

Verify. V now generates PCP queries $q_1, \ldots, q_\mu \in \mathbb{F}^n$, as described in Section 2.2. V sends these queries to P, along with a *consistency query* $t = r + \sum_{j=1}^{\mu} \alpha_j \cdot q_j$, where each α_j is randomly chosen from $\mathbb{F}$ (here, $\cdot$ represents scalar multiplication).

For ease of exposition, we focus on a single proof π_i; however, the following steps happen β times in parallel, using the same queries for each of the β instances. If P is honest, it returns $(\pi_i(q_1), \ldots, \pi_i(q_\mu), \pi_i(t))$. V checks that $\pi_i(t) = \pi_i(r) + \sum_{j=1}^{\mu} \alpha_j \cdot \pi_i(q_j)$; this is known as the *consistency test*. If P is honest, then this test passes, by the linearity of π. Conversely, if this test passes then, *regardless* of P's honesty, V can treat P's responses as the output of an oracle (this is shown in previous work [35, 48]). Thus, V can use $\{\pi_i(q_1), \ldots, \pi_i(q_\mu)\}$ in the PCP tests described in Section 2.2.

2.4 PCPs and arguments defined more formally

The definitions of PCPs [5, 6] and argument systems [20, 32] below are borrowed from [35, 48].

A *PCP protocol* with soundness error ϵ includes a probabilistic polynomial time verifier V that has access to a constraint set $\mathcal{C}$. V makes a constant number of queries to an oracle π. This process has the following properties:

- **PCP Completeness.** If $\mathcal{C}$ is satisfiable, then there exists a linear function π such that, after V queries π, $\Pr\{V \text{ accepts } \mathcal{C} \text{ as satisfiable}\} = 1$, where the probability is over V's random choices.
- **PCP Soundness.** If $\mathcal{C}$ is not satisfiable, then $\Pr\{V \text{ accepts } \mathcal{C} \text{ as satisfiable}\} < \epsilon$ for *all* purported proof functions $\tilde{\pi}$.

An argument (P, V) with soundness error ϵ comprises P and V, two probabilistic polynomial time (PPT) entities that take a set of constraints $\mathcal{C}$ as input and provide:

- **Argument Completeness.** If $\mathcal{C}$ is satisfiable and P has access to a satisfying assignment z, then the interaction of $V(\mathcal{C})$ and $P(\mathcal{C}, z)$ makes $V(\mathcal{C})$ accept $\mathcal{C}$'s satisfiability, regardless of V's random choices.
- **Argument Soundness.** If $\mathcal{C}$ is not satisfiable, then for every malicious PPT P^*, the probability over V's random choices that the interaction of $V(\mathcal{C})$ and $P^*(\mathcal{C})$ makes $V(\mathcal{C})$ accept $\mathcal{C}$ as satisfiable is less than ϵ.

3 Protocol refinements in GINGER

In principle, PEPPER solves the problem of verified computation. The reality is less attractive: PEPPER's computational burden is high, its network costs are absurd, and its applicability is limited (to straight line numerical computations). Our system, GINGER, mitigates these issues: it lowers costs through protocol refinements (presented in this section), and it applies to a much wider class of computations (as we discuss in Section 4).

GINGER's protocol refinements reduce CPU costs by changing the composition of queries in the PCP protocol, and reduce network costs (by orders of magnitude) by compressing queries. Though not surprising theoretically, these refinements are important to practicality, and are rooted in careful inspection of the theory.

Details. GINGER includes three protocol refinements. The first calls for more linearity tests per PCP run, in return for fewer PCP runs. The trade is favorable because linearity tests are cheaper than the other tests; the trade is permissible because the extra linearity tests decrease soundness error in a single run, necessitating fewer runs.

In more detail, *soundness error* (for example, ϵ in Section 2.4) refers to the probability that a protocol or test succeeds when the condition that it is verifying or testing is actually false; the ideal is to have a small upper-bound on soundness error. Meanwhile, the soundness of the PCP protocol in Section 2.2 and Appendix A.1 is controlled by the soundness of linearity testing [17]. Specifically, the base analysis proves that if the prover returns $y \neq \Psi(x)$, then the prover survives all tests (linearity, quadratic correction, circuit) with probability less than $7/9$, requiring ρ runs to make $(7/9)^\rho$ small; the $7/9$ comes from the soundness of linearity testing.

To simplify slightly, the result of doing more linearity testing per PCP run is to decrease the $7/9$ to something smaller (call it κ), at which point a lower value of ρ is required to make κ^ρ small. The simplification here is that we are ignoring a parameter and some of the subtleties of the analysis; Appendix A.2 contains details. We note that our upper-bound on soundness error is 1 in 2 million, which is somewhat low by cryptographic standards. However, in practice, this failure rate (when the prover is malicious) is reasonable.

The second protocol refinement in GINGER is to reuse queries. Specifically, some of the queries generated during linearity testing can do double-duty as required queries elsewhere in the protocol. This refinement is detailed and justified in Appendix A.2 also.

The third refinement saves network costs by compressing queries; the verifier sends, in the second round, a short seed, or key. At that point, both the verifier and the prover derive the PCP queries from the seed, by applying a pseudorandom generator to the seed to obtain the required "random" bits. (Note that the verifier still sends a full consistency query to the prover.) While a proof of security of this refinement seems difficult in the standard model (similar issues have been noted by others [50]), this approach admits a natural proof in the random oracle model; Appendix A.3 contains details.

Savings. Figure 2 depicts the costs under PEPPER and GINGER. Recall that a PCP run under PEPPER consists of a linearity test, a quadratic correction test, and a circuit test (§2.2). The queries in each of the tests are roughly the same cost to construct (the cost scales with n, the number of high-order terms in the PCP encoding). However, the circuit test is far more expensive to *check*: it requires a linear pass over the input and output (which is reflected in the "Process PCP responses" row).

GINGER saves costs because it does roughly the same number of queries as PEPPER in total (376 high-order queries for GINGER, 385 for PEPPER, for the target soundness of 10^{-6}) but far fewer circuit tests (8 in GIN-

	PEPPER [48]	GINGER
PCP encoding size (n)	s^2+s, in general	s^2+s, in general
***V*'s per-instance CPU costs**		
Issue commit queries	$(e+2c)\cdot n/\beta$	$(e+2c)\cdot n/\beta$
Process commit responses	d	d
Issue PCP queries	$\rho_{\text{pepp}}\cdot(Q+n\cdot(4c+(\ell_{\text{pepp}}+1)\cdot f))/\beta$	$\rho_{\text{ging}}\cdot(Q+n\cdot(2\rho_{\text{lin}}\cdot c+(\ell_{\text{ging}}+1)\cdot f))/\beta$
Process PCP responses	$\rho_{\text{pepp}}\cdot(2\ell_{\text{pepp}}+\lvert x\rvert+\lvert y\rvert)\cdot f$	$\rho_{\text{ging}}\cdot(2\ell_{\text{ging}}+\lvert x\rvert+\lvert y\rvert)\cdot f$
***P*'s per-instance CPU costs**		
Issue commit responses	$h\cdot n$	$h\cdot n$
Issue PCP responses	$(\rho_{\text{pepp}}\cdot\ell_{\text{pepp}}+1)\cdot f\cdot n$	$(\rho_{\text{ging}}\cdot\ell_{\text{ging}}+1)\cdot f\cdot n$
Network cost (per instance)	$((\rho_{\text{pepp}}\cdot\ell_{\text{pepp}}+1)\cdot\lvert p\rvert+\lvert\xi\rvert)\cdot n/\beta$	$(\lvert p\rvert+\lvert\xi\rvert)\cdot n/\beta$
Upper-bound on soundness error	$(7/9)^{\rho_{\text{pepp}}}=9.9\cdot10^{-7}$	$\kappa^{\rho_{\text{ging}}}=5.8\cdot10^{-7}$

$\lvert x\rvert,\lvert y\rvert$: # of elements in input, output
n: # of components in linear function π (§2.2)
s: # of variables in constraint set (§2.1)
χ: # of constraints in constraint set (§2.1)
K: # of additive terms in constraints for Ψ
$\rho_{\text{lin}}=15$: # of linearity tests per PCP run in GINGER (§A.2)
$\ell_{\text{pepp}}=7$: # of high-order PCP queries in PEPPER (§A.1)
$\ell_{\text{ging}}=47$ ($=3\cdot\rho_{\text{lin}}+2$): # of high-order PCP queries in GINGER (§A.2)
$\rho_{\text{pepp}}=55$: # of PCP reps. in base scheme (§A.1)
$\rho_{\text{ging}}=8$: # of PCP reps. in GINGER (§A.2)
β: batch size (# of instances) (§2.3)
e: cost of encrypting an element in $\mathbb{F}$
d: cost of decrypting an encrypted element
f: cost of multiplying in $\mathbb{F}$
h: cost of ciphertext add plus multiply
c: cost of pseudorandomly generating an element in $\mathbb{F}$
$\lvert p\rvert$: length of an element in $\mathbb{F}$
$\lvert\xi\rvert$: length of an encrypted element in $\mathbb{F}$
$Q=\chi\cdot c+K\cdot f$: circuit query setup work (§2.2)
κ: upper-bound on PCP soundness error in GINGER

Figure 2—High-order costs and error in GINGER, compared to its base (PEPPER [48]), for a computation represented as χ constraints over s variables (§2.1). The soundness error depends on field size (Appendix A.2); the table assumes $\lvert\mathbb{F}\rvert=2^{128}$. Many of the cryptographic costs enter through the commitment protocol (see Section 2.3 or Figure 12); Section 6 quantifies the parameters. The "PCP" rows include the consistency query and check. The network costs slightly underestimate by not including query responses. Note that while GINGER does *more* linearity queries than PEPPER, GINGER does fewer of the more expensive PCP queries.

GER, 55 in PEPPER). As a result, GINGER has smaller values of β^*, the minimum batch size (§2.3) at which V gains from outsourcing. As shown in Section 6.1, the reduction in β^* is roughly a factor of 3. Moreover, the lower costs allow the verifier to gain from outsourcing even when the problem size is small. For example, under PEPPER, the fixed costs of the protocol preclude the verifier's breaking even for $m\times m$ matrix multiplication for $m=100$ (because $55\cdot(\lvert x\rvert+\lvert y\rvert)=55\cdot 3m^2>m^3$); GINGER, however, can break even for this computation, with a batch size of 2300.

We note that while both protocols, PEPPER and GINGER, perform hundreds of PCP queries, this cost is dominated by the cost to construct a single encrypted commitment query (because e is orders of magnitude larger than the other parameters; see [49, Appendix E]). In fact, next to that query, the main cost of many PCP queries is in network costs. However, GINGER's protocol refinements save 1-2 orders of magnitude in network costs (if we take $\lvert p\rvert=128$ bits and $\lvert\xi\rvert=2\cdot1024$ bits, and hold β constant); see Section 6.1.

4 Broadening the space of computations

GINGER extends to computations over floating-point fractional quantities and to a restricted general-purpose programming model that includes inequality tests, logical expressions, conditional branching, etc. To do so, GINGER maps computations to the constraint-over-finite-field formalism (§2.1), and thus the core protocol in Section 3 applies. In fact, our techniques[4] apply to the many protocols that use the constraint formalism or arithmetic circuits. Moreover, we have implemented a compiler (derived from Fairplay's [40]) that transforms high-level computations first into constraints and then into verifier and prover executables.

The challenges of representing computations as constraints over finite fields include: the "true answer" to the computation may live outside of the field; sign and ordering in finite fields interact in an unintuitive fashion; and constraints are simply equations, so it is not obvious how to represent comparisons, logical expressions, and control flow. To explain GINGER's solutions, we first present an abstract framework that illustrates how GINGER broadens the set of computations soundly and how one can apply the approach to further computations.

[4]We suspect that many of the individual techniques are known. However, when the techniques combine, the material is surprisingly hard to get right, so we will delve into (excruciating) detail, consistent with our focus on built systems.

Framework to map computations to constraints. To map a computation Ψ over some domain D (such as the integers, $\mathbb{Z}$, or the rationals, $\mathbb{Q}$) to equivalent constraints over a finite field, the programmer or compiler performs three steps, as illustrated and described below:

$$\begin{array}{ccccc} \Psi \text{ over } D & \xrightarrow{(C1)} & \Psi \text{ over } U & \xrightarrow{(C2)} & \theta(\Psi) \text{ over } \mathbb{F} \\ & & & & \big\downarrow{\scriptstyle (C3)} \\ & & & & \mathcal{C} \text{ over } \mathbb{F} \end{array}$$

C1 *Bound the computation.* Define a set $U \subset D$ and restrict the input to Ψ such that the output and intermediate values stay in U.

C2 *Represent the computation faithfully in a suitable finite field.* Choose a finite field, $\mathbb{F}$, and a map $\theta\colon U \to \mathbb{F}$ such that computing $\theta(\Psi)$ over $\theta(U) \subset \mathbb{F}$ is isomorphic to computing Ψ over U. (By "$\theta(\Psi)$", we mean Ψ with all inputs and literals mapped by θ.)

C3 *Transform the finite field version of the computation into constraints.* Write a set of constraints over $\mathbb{F}$ that are equivalent (in the sense of Section 2.1) to $\theta(\Psi)$.

4.1 Signed integers and floating-point rationals

We now instantiate C1 and C2 for integer and rational number computations; the next section addresses C3.

Consider $m \times m$ matrix multiplication over N-bit signed integers. For step C1, each term in the output, $\sum_{k=1}^{m} A_{ik}B_{kj}$, has m additions of $2N$-bit subterms so is contained in $[-m \cdot 2^{2N-1}, m \cdot 2^{2N-1})$; this is our set U.

For step C2, take $\mathbb{F} = \mathbb{Z}/p$ (the integers mod a prime p, to be chosen shortly) and define $\theta\colon U \to \mathbb{Z}/p$ as $\theta(u) = u \bmod p$. Observe that θ maps negative integers to $\{\frac{p+1}{2}, \frac{p+3}{2}, \ldots, p-1\}$, analogous to how processors represent negative numbers with a 1 in the most significant bit (this technique is standard [18, 54]). Of course, addition and multiplication in $\mathbb{Z}/p$ do not "know" when their operands are negative. Nevertheless, the computation over $\mathbb{Z}/p$ is isomorphic to the computation over U, provided that $|\mathbb{Z}/p| > |U|$ (as shown in Appendix B [49]).[5] Thus, for the given U, we require $p > m \cdot 2^{2N}$. Note that a larger p brings larger costs (see Figure 2), so there is a three-way trade-off among p, m, N.

We now turn to rational numbers. For step C1, we restrict the inputs as follows: when written in lowest terms, their numerators are $(N_a + 1)$-bit signed integers, and their denominators are in $\{1, 2, 2^2, 2^3, \ldots, 2^{N_b}\}$. Note that such numbers are (primitive) floating-point numbers: they can be represented as $a \cdot 2^{-q}$, so the decimal point floats based on q. Now, for $m \times m$ matrix multiplication, the computation does not "leave" $U = \{a/b\colon |a| < 2^{N'_a}, b \in \{1, 2, 2^2, 2^3, \ldots, 2^{N'_b}\}\}$, for $N'_a = 2N_a + 2N_b + \log_2 m$ and $N'_b = 2N_b$ [49, Appendix B].

[5] For space, Appendices B–E appear only in the extended version [49].

For step C2, we take $\mathbb{F} = \mathbb{Q}/p$, the quotient field of $\mathbb{Z}/p$. Take $\theta(\frac{a}{b}) = (a \bmod p, b \bmod p)$. For any $U \subset \mathbb{Q}$, there is a choice of p such that the mapped computation over $\mathbb{Q}/p$ is isomorphic to the original computation over $\mathbb{Q}$ [49, Appendix B]. For our U above, $p > 2m \cdot 2^{2N_a+4N_b}$ suffices.

Limitations and costs. To understand the limitations of GINGER's floating-point representation, consider the number $a \cdot 2^{-q}$, where $|a| < 2^{N_a}$ and $|q| \leq N_q$. To represent this number, the IEEE standard requires roughly $N_a + \log N_q + 1$ bits [30] while GINGER requires $N_a + 2N_q + 1$ bits [49, Appendix B]. As a result, GINGER's range is vastly more limited: with 64 bits, the IEEE standard can represent numbers on the order of 2^{1023} and 2^{-1022} (with $N_a = 53$ bits of precision) while 64 bits buys GINGER only numbers on the order of 2^{32} and 2^{-31} (with $N_a = 32$). Moreover, unlike the IEEE standard, GINGER does not support a division operation or rounding.

However, comparing GINGER's floating-point representation to its *integer* representation, the extra costs are not terrible. First, the prover and verifier take an extra pass over the input and output (for implementation reasons; see Appendix B [49] for details). Second, a larger prime p is required. For example, $m \times m$ matrix multiplication with 32-bit integer inputs requires p to have at least $\log_2 m + 64$ bits; if the inputs are rationals with $N_a = N_q = 32$, then p requires $\log_2 m + 193$ bits. The end-to-end costs are about 2× those of the integers case (see Section 6.2). Of course, the actual numbers depend on the computation. (Our compiler computes suitable bounds with static analysis.)

4.2 General-purpose program constructs

Case study: branch on order comparison. We now illustrate C3 with a case study of a computation, Ψ, that includes a less-than test and a conditional branch; pseudocode for Ψ is in Figure 3. For clarity, we will restrict Ψ to signed integers; handling rational numbers requires additional mechanisms [49, Appendix C].

How can we represent the test $x_1 < x_2$ using constraint *equations*? The solution is to use special *range constraints* that decompose a number into its bits to test whether it is in a given range; in this case, $\mathcal{C}_<$, depicted in Figure 3, tests whether $e = \theta(x_1) - \theta(x_2)$ is in the "negative" range of $\mathbb{Z}/p$ (see Section 4.1). Now, under the input restriction $x_1 - x_2 \in U$, $\mathcal{C}_<$ is satisfiable if and only if $x_1 < x_2$ [49, Appendix C]. Analogously, we can construct $\mathcal{C}_{>=}$ that is satisfiable if and only if $x_1 \geq x_2$.

Finally, we introduce a 0/1 variable M that encodes a choice of branch, and then arrange for M to "pull in" the constraints of that branch and "exclude" those of the other. (Note that the prover need not execute the untaken branch.) Figure 3 depicts the complete set of constraints,

```
Ψ :

if (X1 < X2)
  Y = 3
else
  Y = 4
```

$$\mathcal{C}_< = \left\{ \begin{array}{ll} B_0(1-B_0) & = 0, \\ B_1(2-B_1) & = 0, \\ \vdots & \vdots \\ B_{N-2}(2^{N-2}-B_{N-2}) & = 0, \\ \theta(X_1)-\theta(X_2)-(p-2^{N-1})-\sum_{i=0}^{N-2} B_i & = 0 \end{array} \right\} \qquad \mathcal{C}_\Psi = \left\{ \begin{array}{l} M\{\mathcal{C}_<\}, \\ M(Y-3)=0, \\ (1-M)\{\mathcal{C}_{>=}\}, \\ (1-M)(Y-4)=0 \end{array} \right\}$$

Figure 3—Pseudocode for our case study of Ψ, and corresponding constraints $\mathcal{C}_\Psi$. Ψ's inputs are signed integers x_1, x_2; per steps C1 and C2 (§4.1), we assume $x_1 - x_2 \in U \subset [-2^{N-1}, 2^{N-1})$, where $p > 2^N$. The constraints $\mathcal{C}_<$ test $x_1 < x_2$ by testing whether the bits of $\theta(x_1) - \theta(x_2)$ place it in $[p - 2^{N-1}, p)$. $M\{\mathcal{C}\}$ means multiplying all constraints in $\mathcal{C}$ by M and then reducing to degree-2.

$\mathcal{C}_\Psi$; these constraints are satisfiable if and only if the prover correctly computes Ψ [49, Appendix C].

Logical expressions and conditionals. Besides order comparisons and if-else, GINGER can represent `==`, `&&`, and `||` as constraints. An interesting case is `!=`: we can represent `Z1!=Z2` with $\{M \cdot (Z_1 - Z_2) - 1 = 0\}$ because this constraint is satisfiable when $(Z_1 - Z_2)$ has a multiplicative inverse and hence is not zero. These constructs and others are detailed in Appendix D [49].

Limitations and costs. We compile a subset of SFDL, the language of the Fairplay compiler [40]. Thus, our limitations are essentially those of SFDL; notably, loop bounds have to be known at compile time.

How efficient is our representation? The program constructs above mostly have concise constraint representations. Consider, for instance, `comp1==comp2`; the equivalent constraint set $\mathcal{C}$ consists of the constraints that represent `comp1`, the constraints that represent `comp2`, and an additional constraint to relate the outputs of `comp1` and `comp2`. Thus, $\mathcal{C}$ is the same size as its two components, as one would expect.

However, two classes of computations are costly. First, inequality comparisons require variables and a constraint for every bit position; see Figure 3. Second, the constraints for if-else and `||`, as written, seem to be degree-3; notice, for instance, the $M\{C_<\}$ in Figure 3. To be compatible with the core protocol, these constraints must be rewritten to be total degree 2 (§2.1), which carries costs. Specifically, if $\mathcal{C}$ has s variables and χ constraints, an equivalent total degree 2 representation of $M\{\mathcal{C}\}$ has $s + \chi$ variables and $2 \cdot \chi$ constraints [49, Appendix D].

5 Parallelization and implementation

Many of GINGER's remaining costs are in the cryptographic operations in the commitment protocol (see Appendix A.1). To address these costs, we distribute the prover over multiple machines, leveraging GINGER's inherent parallelism. We also implement the prover and verifier on GPUs, which raises two questions. (1) Isn't this just moving the problem? Yes, and this is good: GPUs are optimized for the types of operations that bottleneck GINGER. (2) Why do we assume that the *verifier* has a GPU? Desktops are more likely than servers to have GPUs, and the prevalence of GPUs is increasing. Also, this setup models a future in which specialized hardware for cryptographic operations is common.

Parallelization. To distribute GINGER's prover, we run multiple copies of it (one per host), each copy receiving a fraction of the batch (Section 2.3). In this configuration, the provers use the Open MPI [2] message-passing library to synchronize and exchange data.

To further reduce latency, each prover offloads work to a GPU (see also [53] for an independent study of GPU hardware in the context of [22]). We exploit three levels of parallelism here. First, the prover performs a ciphertext operation for each component in the commitment vector (§2.3); each operation is (to first approximation) separate. Second, each operation computes two independent modular exponentiations (the ciphertext of an ElGamal encryption has two elements). Third, modular exponentiation itself admits a parallel implementation (each input is a multiprecision number encoded in multiple machine words). Thus, in our GPU implementation, a group of CUDA [1] threads computes each exponentiation.

We also parallelize the verifier's encryption work during the commitment phase (§2.3), using the approach above plus an optimization: the verifier's exponentiations are fixed base, letting us memoize intermediate squares. As another optimization, we implement simultaneous multiple exponentiation [41, Chapter 14.4], which accelerates the prover.[6]

We implement exponentiations for the prover and verifier with the `libgpucrypto` library of SSLShader [36], modified to implement the memoization.

Implementation details. Our compiler consists of two stages, which a future publication will detail. The front-end compiles a subset of Fairplay's SFDL [40] to constraints; it is derived from Fairplay and is implemented in 5294 lines of Java, starting from Fairplay's 3886 lines (per [55]). The back-end transforms constraints into C++ code that implements the verifier and prover and then invokes gcc; this component is 1105 lines of Python code.

For efficiency, PEPPER [48] introduced specialized

[6]This last optimization is is an improvement over the originally published version; although the technique is well-known, we were inspired by other works that implement it [33, 39, 45].

GINGER's protocol refinements reduce per-instance network costs by 20–30× (to hundreds of KBs for the computations we study), prover CPU costs by about a factor of 2, and break-even batch size (β^*) by about 3×.	§6.1
With accelerated encryption GINGER breaks even from outsourcing short computations at small batch sizes; for 400×400 matrix multiplication, the verifier gains from outsourcing at a batch size of 740.	§6.1
Rational arithmetic costs roughly 2× integer arithmetic under GINGER (but much more than native floating-point).	§6.2
Parallelizing results in near-linear reduction in the prover's latency.	§6.3

Figure 4—Summary of main evaluation results.

computation (Ψ)	$O(\cdot)$	input domain (see §4.1)	size of $\mathbb{F}$	s	n	default	local
matrix mult.	$O(m^3)$	32-bit signed integers	128 bits	$2m^2$	m^3	$m = 200$	800 ms
matrix mult. ($\mathbb{Q}$)	$O(m^3)$	rationals ($N_a = 32$, $N_b = 32$)	220 bits	$2m^2$	m^3	$m = 100$	5.90 ms
deg-2 poly. eval.	$O(m^2)$	32-bit signed integers	128 bits	m	m^2	$m = 100$	0.40 ms
deg-3 poly. eval.	$O(m^3)$	32-bit signed integers	192 bits	m	m^3	$m = 200$	160 ms
m-Hamming dist.	$O(m^2)$	32-bit unsigned	128 bits	$2m^2 + m$	$2m^3$	$m = 100$	0.90 ms
bisection method	$O(m^2)$	rationals ($N_a = 32$, $N_b = 5$)	220 bits	$16 \cdot (m + \lvert\mathcal{C}_<\rvert)$	$256 \cdot (m + \lvert\mathcal{C}_<\rvert)^2$	$m = 25$	180 ms

Figure 5—Benchmark computations. s is the number of constraint variables; s affects n, which is the size of V's queries and of P's linear function π (see Figure 2). Only high-order terms are reported for n. The latter two columns give our experimental defaults and the cost of local computation (i.e., no outsourcing) at those defaults. In polynomial evaluation, V and P hold a polynomial; the input is values for the m variables. The latter two computations exercise the program constructs in Section 4.2. In m-Hamming distance, V and P hold a fixed set of strings; the input is a length m string, and the output is a vector of the Hamming distance between the input and the set of strings. Bisection method refers to root-finding via bisection: both V and P hold a degree-2 polynomial in m variables, the input is two m-element endpoints that bracket a root, and the output is a small interval that contains the root.

PCP protocols for certain computations. For some experiments we use specialized PCPs in GINGER also; in these cases we write the prover and verifier manually, which typically requires a few hundred lines of C++. Automating the compilation of specialized PCPs is future work.

The verifier and prover are separate processes that exchange data using Open MPI [2]. GINGER uses the ElGamal cryptosystem [24] with 1024-bit keys. For generating pseudorandom bits, GINGER uses the `amd64-xmm6` variant of the Chacha/8 stream cipher [15] in its default configuration as a pseudorandom generator.

6 Experimental evaluation

Our evaluation answers the following questions:

- What is the effect of the protocol refinements (§3)?
- What are the costs of supporting rational numbers and the additional program structures (§4)?
- What is GINGER's speedup from parallelizing (§5)?

Figure 4 summarizes the results.

We use six benchmark computations, summarized in Figure 5 (Appendix E [49] has details). For bisection method and degree-2 polynomial evaluation, V and P were produced by our compiler; for the other computations, we use tailored encodings (see Section 5). We implemented and analyzed other computations (e.g., edit distance and circle packing) but found that V gained from outsourcing only at implausibly large batch sizes.

Method and setup. We measure latency and computing cycles used by the verifier and the prover, and the amount of data exchanged between them. We account for the prover's cost in per-instance terms. Because the verifier amortizes costs over a batch (§2.3), we focus on the *break-even batch size*, β^*: the batch size at which the verifier's CPU cost from GINGER equals the cost of computing the batch locally. We measure local computation using implementations built on the GMP library (except for matrix multiplication over rationals, where we use native floating-point).

For each result that we report, we run at least three experiments and take the averages (the standard deviations are always within 5% of the means). We measure CPU time using `getrusage`, latency using PAPI's real time counter [3], and network costs by recording the number of application-level bytes transferred.

Our experiments use a cluster at the Texas Advanced Computing Center (TACC). Each machine is configured identically and runs Linux on an Intel Xeon processor E5540 2.53 GHz with 48GB of RAM. Experiments with GPUs use machines with an NVIDIA Quadro FX 5800. Each GPU has 240 CUDA cores and 4GB of memory.

Validating the cost model. We will sometimes predict β^*, V's costs, and P's costs by using our cost model (Figure 2), so we now validate this model. We run microbenchmarks to quantify the model's parameters—e is reported in this section; Appendix E [49] quantifies the other parameters—and then compare the parameterized model to GINGER's measured performance. GINGER's empirical results are at most 2%–15% more than are predicted by the model.

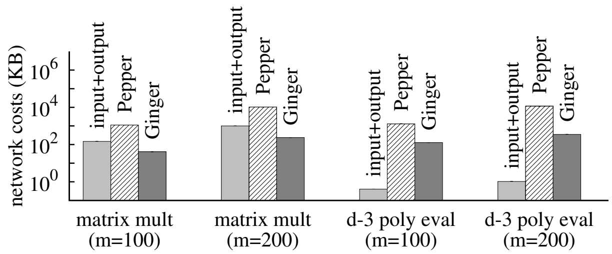

Figure 6—Per-instance network costs of GINGER and its base (PEPPER [48]), compared to the size of the inputs and outputs. At this batch size ($\beta = 5000$), GINGER's refinements reduce per-instance network costs by a factor of 20–30 compared to PEPPER. GINGER's network costs here are hundreds of KB or less. The y-axis is log-scaled.

		PEPPER	GINGER
	local	4.2 s	4.2 s
CPU	β^*	4500	1800
	verifier aggregate	5.3 hr	2.1 hr
	prover aggregate	1.7 yr	140 days
	prover per-instance	3.4 hr	1.8 hr
GPU	β^*	3600	1300
	verifier aggregate	4.3 hr	1.5 hr
	prover aggregate	1.4 yr	97 days
	prover per-instance	3.4 hr	1.8 hr
crypto hardware	β^*	2800	740
	verifier aggregate	3.3 hr	52.3 min
	prover aggregate	1.1 yr	57 days
	prover per-instance	3.4 hr	1.8 hr

Figure 7—Break-even batch sizes (β^*) and estimated running times of prover and verifier at $\beta = \beta^*$, for matrix multiplication ($m = 400$), under three models of the encryption cost. The verifier's per-instance work is not depicted because it equals the local running time, by definition of β^*. The local running time is high in part because the local implementation uses GMP.

6.1 The effect of GINGER's protocol refinements

We begin with $m \times m$ matrix multiplication ($m = 100, 200$) and degree-3 polynomial evaluation ($m = 100, 200$), and batch size of $\beta = 5000$. We report *per-instance* network and CPU costs: the total network and CPU costs over the batch, divided by β.

Figure 6 depicts network costs. In our experiments, these costs for matrix multiplication are about the same as the cost to send the inputs and receive the ouputs; for degree-3 polynomial evaluation, the costs are about 100 times the size of the inputs and outputs (owing to a large problem description, namely all $O(m^3)$ coefficients). Per-instance, GINGER's network costs are hundreds of KB or less, a 20–30× improvement over PEPPER.

In this experiment, GINGER's prover incurs about 2× less CPU time compared to PEPPER (estimated using a cost model from [48]) but still takes tens of minutes per-instance; this is obviously a lot, but we reduce latency by parallelizing (§6.3). For this computation and at this batch size ($\beta = 5000$), GINGER's verifier takes a few hundreds of milliseconds per-instance, less than locally computing using our baseline of GMP.

	mat. mult.	mat. mult. ($\mathbb{Q}$)
local	66.3 ms	5.90 ms
verifier per-instance	66.3 ms	146.7 ms
verifier aggregate	2.5 min	5.5 min
prover per-instance	1.7 min	2.6 min
prover aggregate	2.7 days	4.0 days

Figure 8—Estimated running times of GINGER's verifier and prover for matrix multiplication ($m = 100$), under integer and floating-point inputs, at $\beta = 2200$ (the break-even batch size for this computation over integers). The "local" row refers to GMP arithmetic for $\mathbb{Z}$ and native floating-point for $\mathbb{Q}$. Handling rationals costs 1.5–2.2× (depending on the metric) more than handling integers, but both are still far from native.

computation (Ψ)	# Boolean gates (est.)	# constraint vars.
m-Hamming dist.	$1.3 \cdot 10^6$	$2 \cdot 10^4$
bisection method	$3.0 \cdot 10^8$	1528

Figure 9—GINGER's constraints compared to Boolean circuits, for m-Hamming distance ($m = 100$) and bisection method ($m = 25$). The Boolean circuits are estimated using the unmodified Fairplay [40] compiler. GINGER's constraints are not concise but are far more so than Boolean circuits.

Amortizing the verifier's costs. Batching is both a limitation and a strength of GINGER: GINGER's verifier *must* batch to gain from outsourcing but *can* batch to drive per-instance overhead arbitrarily low. Nevertheless, we want break-even batch sizes (β^*) to be as small as possible. But β^* mostly depends on *e*, the cost of encryption (Figure 2), because after our refinements the verifier's main burden is creating $\mathsf{Enc}(pk, r)$ (see §2.3), the cost of which amortizes over the batch.

What values of *e* make sense? We consider three scenarios: (1) the verifier uses a CPU for encryptions, (2) the verifier offloads encryptions to a GPU, and (3) the verifier has special-purpose hardware that can *only* perform encryptions. (See Section 5 for motivation.) Under scenario (1), we measure $e = 65\mu$s on a 2.53 GHz CPU. Under scenario (3), we take $e = 0\mu$s. What about scenario (2)? Our cost model concerns *CPU* costs, so we need an exchange rate between GPU and CPU exponentations. We make a crude estimate: we measure the number of encryptions per second achievable on an NVIDIA Tesla M2070 (which is 229,000) and on an Intel 2.55 GHz CPU (which is 15,400), normalize by the dollar cost of the chips, and obtain that their throughput-per-dollar ratio is 2×. We thus take $e = 65/2 = 32.5\mu$s.

We plug these three values of *e* into the cost model in Figure 2, set the cost under GINGER equal to the cost of local computing, and solve for β^*. The values of β^* are 1800 (CPU), 1300 (GPU), and 740 (crypto hard-

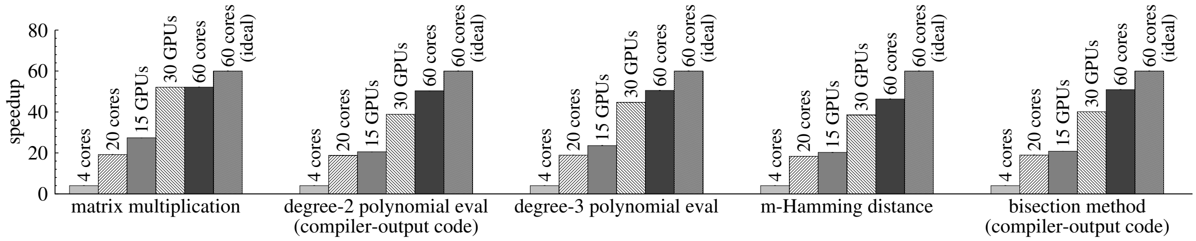

Figure 10—Latency speedup observed by GINGER's verifier when the prover is parallelized. We run with $m = 100, \beta = 120$ for matrix multiplication; $m = 256, \beta = 1200$ for degree-2 polynomial evaluation; $m = 100, \beta = 180$ for degree-3 polynomial evaluation and m-Hamming distance; and $m = 25, \beta = 180$ for bisection method. GINGER's prover achieves near-linear speedups .

ware). We also use the model to predict V's and P's costs at β^*, under PEPPER and GINGER. Figure 7 summarizes. The aggregate verifier computing time drops significantly (2.5–3×) under all three cost models. The prover's per-instance work is mostly unaffected, but as the batch size decreases, so does its aggregate work.

6.2 Evaluating GINGER's computational model

To understand the costs of the floating-point representation (§4.1), we compare it to two baselines: GINGER's signed integer representation and the computation executed locally, using the CPU's floating point unit. Our benchmark application is matrix multiplication ($m = 100$). Figure 8 details the comparison.

We also consider GINGER's general-purpose program constructs (§4). Our baseline is *Boolean* circuits (we are unaware of efficient arithmetic representations of these constructs). We compare the number of Boolean circuit *gates* and the number of GINGER's arithmetic constraint *variables*, since these determine the proving and verifying costs under the respective formalisms (see [5, 48]). Taken individually, GINGER's constructs (`<=`, `&&`, etc.) are the same cost or more than those of Boolean circuits (e.g., `||` introduces auxiliary variables). However, Boolean circuits are in general far more verbose: they represent quantities by their bits (which GINGER does only when computing inequalities). Figure 9 gives a rough end-to-end comparison.

6.3 Scalability of the parallel implementation

To demonstrate the scalability of GINGER's parallelization, we run the prover using many CPU cores, many GPUs, and many machines. We measure end-to-end latency, as observed by the verifier. Figure 10 summarizes the results for various computations. In most cases, the speedup is near-linear.

7 Related work

A substantial body of work achieves two of our goals—it is general-purpose and practical—but it makes strong assumptions about the servers (e.g., trusted hardware). There is also a large body of work on protocols for special-purpose computation. We regard this work as orthogonal to our efforts; for a survey of this landscape, see [48]. Herein, we focus on approaches that are general-purpose and unconditional.

Homomorphic encryption and secure multi-party protocols. Homomorphic encryption (which enables computation over ciphertext) and secure multi-party protocols (in which participants compute over private data, revealing only the result [34, 40, 56]) provide only *privacy* guarantees, but one can build on them for verifiable computation. For instance, the Boneh-Goh-Nissim homomorphic cryptosystem [19] can be adapted to evaluate circuits, Groth uses homomorphic commitments to produce a zero-knowledge argument protocol [33], and Applebaum et al. use secure multi-party protocols for verifying computations [4]. Also, Gentry's fully homomorphic encryption [28] has engendered protocols for verifiable non-interactive computation [21, 25, 27]. However, despite striking improvements [29, 44, 51], the costs of hiding inputs (among other expenses) prevent any of the aforementioned verified computation schemes from getting close to practical (even by our relaxed standards).

PCPs, argument systems, and interactive proofs. Applying proof systems to verifiable computation is standard in the theory community [5–7, 10, 16, 32, 37, 38, 43], and the asymptotics continue to improve [13, 14, 23, 46]. However, none of this work has paid much attention to building systems.

Very recently, researchers have begun to explore using this theory for practical verified outsourced computation. In a recent preprint, Ben-Sasson et al. [12] investigate when PCP protocols might be beneficial for outsourcing. Since many of the protocols require representing computations as constraints, Ben-Sasson et al. [11] study improved reductions to constraints from a RAM model of computation. And Gennaro et al. [26] give a new characterization of NP to provide asymptotically efficient arguments without using PCPs.

However, as far as we know, only two research groups have made serious efforts toward practical systems. Our previous work [47, 48] built upon the efficient argument system of Ishai et al. [35]. In contrast, Cormode, Mitzenmacher, and Thaler [22] (hereafter, CMT) built upon the

m	domain	component	CMT-native	CMT-GMP	GINGER
256	$\mathbb{Z}$	verifier	40 ms	0.6 s	0.6 s
		prover	22 min	2.5 hr	19 min
		network	87 KB	0.3 MB	0.2 MB
128	$\mathbb{Q}$	verifier	–	220 ms	270 ms
		prover	–	41 min	6.1 min
		network	–	0.9 MB	0.2 MB

Figure 11—CMT [22] compared to GINGER, in terms of *amortized* CPU and network costs (GINGER's total costs are divided by a batch size of β=5000 instances), for $m \times m$ matrix multiplication. CMT-native uses native data types but is restricted to small problem sizes and domains. CMT-GMP uses the GMP library for multi-precision arithmetic (as does GINGER).

protocol of Goldwasser et al. [31], and a follow-up effort studies a GPU-based parallel implementation [53].

Comparison of GINGER and CMT [22, 53]. We compared three different implementations: *CMT-native*, *CMT-GMP*, and GINGER. CMT-native refers to the code and configuration released by Thaler et al. [53]; it works over a small field and thereby exploits highly efficient machine arithmetic but restricts the inputs to the computation unrealistically (see Section 4.1). CMT-GMP refers to an implementation based on CMT-native but modified by us to use the GMP library for multi-precision arithmetic; this allows more realistic computation sizes and inputs, as well as rational numbers.

We perform two experiments using $m \times m$ matrix multiplication. Our testbed is the same as in Section 6. In the first one, we run with $m = 256$ and integer inputs. For CMT-GMP and GINGER, the inputs are 32-bit unsigned integers, and the prime (the field modulus) is 128 bits. For CMT-native, the prime is $2^{61} - 1$. In the second experiment, m is 128, the inputs are rational numbers (with $N_a = N_b = 32$; see Section 4.1), the prime is 220 bits, and we experiment only with CMT-GMP and GINGER.

We measure total CPU time and network cost; for CMT, we measure "network" traffic by counting bytes (the CMT verifier and prover run in the same process and hence the same machine). Each reported datum is an average over 3 sample runs; there is little experimental variation (less than 5% of the means).

Figure 11 depicts the results. CMT incurs a significant penalty when moving from native to GMP (and hence to realistic problem sizes). Comparing CMT-GMP and GINGER, the network and prover costs are similar (although network costs for CMT reflect high fixed overhead for their circuit). The *per-instance* verifier costs are also similar, but GINGER is batch verifying whereas CMT does not need to do so (a significant advantage).

A qualitative comparison is as follows. On the one hand, CMT does not require cryptography, has better asymptotic prover and network costs, and for some computations the verifier does not need batching to gain from outsourcing [53]. On the other hand, CMT applies to a smaller set of computations: if the computation is not efficiently parallelizable or does not naturally map to arithmetic circuits (e.g., it has order comparisons or conditionality), then CMT in its current form will be inapplicable or inefficient, respectively. Ultimately, GINGER and CMT should be complementary, as one can likely ease or eliminate some of the restrictions on CMT by incorporating the constraint formalism together with batching [52].

8 Summary and conclusion

This paper is a contribution to the emerging area of practical PCP-based systems for unconditional verifiable computation. GINGER has combined protocol refinements (slashing query costs); a general computational model (including fractions and standard program constructs) with a compiler; and a massively parallel implementation that takes advantage of modern hardware. Together, these changes have brought us closer to a truly deployable system. Nevertheless, much work remains: efficiency depends on tailored protocols, the costs for the prover are still too high, and looping cannot yet be handled concisely.

Acknowledgments

Detailed attention from Edmund L. Wong substantially clarified this paper. Yuval Ishai, Mike Lee, Bryan Parno, Mark Silberstein, Chung-chieh (Ken) Shan, Sara L. Su, Justin Thaler, Brent Waters, and the anonymous reviewers gave useful comments that improved this draft. The Texas Advanced Computing Center (TACC) at UT supplied computing resources. We thank Jane-Ellen Long, of USENIX, for her good nature and inexhaustible patience. The research was supported by AFOSR grant FA9550-10-1-0073 and by NSF grants 1055057 and 1040083.

We thank Alessandro Chiesa, Yuval Ishai, Nir Bitanksy, and Omer Paneth for their careful attention, and for noticing a significant error in a previous version of this paper.

Our code and experimental configurations are available at `http://www.cs.utexas.edu/pepper`

References

[1] CUDA (http://developer.nvidia.com/what-cuda).
[2] Open MPI (http://www.open-mpi.org).
[3] PAPI: Performance Application Programming Interface.
[4] B. Applebaum, Y. Ishai, and E. Kushilevitz. From secrecy to soundness: efficient verification via secure computation. In *ICALP*, 2010.
[5] 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.
[6] S. Arora and S. Safra. Probabilistic checking of proofs: a new characterization of NP. *J. of the ACM*, 45(1):70–122, Jan. 1998.
[7] L. Babai, L. Fortnow, L. A. Levin, and M. Szegedy. Checking computations in polylogarithmic time. In *STOC*, 1991.

[8] 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.
[9] M. Bellare, S. Goldwasser, C. Lund, and A. Russell. Efficient probabilistically checkable proofs and applications to approximations. In *STOC*, 1993.
[10] M. Ben-Or, S. Goldwasser, J. Kilian, and A. Wigderson. Multi-prover interactive proofs: how to remove intractability assumptions. In *STOC*, 1988.
[11] 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.
[12] E. Ben-Sasson, A. Chiesa, D. Genkin, and E. Tromer. On the concrete-efficiency threshold of probabilistically-checkable proofs. *ECCC*, (045), Apr. 2012.
[13] 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.
[14] E. Ben-Sasson and M. Sudan. Short PCPs with polylog query complexity. *SIAM J. on Comp.*, 38(2):551–607, May 2008.
[15] D. J. Bernstein. ChaCha, a variant of Salsa20. `http://cr.yp.to/chacha.html`.
[16] M. Blum and S. Kannan. Designing programs that check their work. *J. of the ACM*, 42(1):269–291, 1995.
[17] 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.
[18] D. Boneh and D. M. Freeman. Homomorphic signatures for polynomial functions. In *EUROCRYPT*, 2011.
[19] D. Boneh, E. J. Goh, and K. Nissim. Evaluating 2-DNF formulas on ciphertexts. In *TCC*, 2005.
[20] G. Brassard, D. Chaum, and C. Crépeau. Minimum disclosure proofs of knowledge. *J. of Comp. and Sys. Sciences*, 37(2):156–189, 1988.
[21] K.-M. Chung, Y. Kalai, and S. Vadhan. Improved delegation of computation using fully homomorphic encryption. In *CRYPTO*, 2010.
[22] G. Cormode, M. Mitzenmacher, and J. Thaler. Practical verified computation with streaming interactive proofs. In *ITCS*, 2012.
[23] I. Dinur. The PCP theorem by gap amplification. *J. of the ACM*, 54(3), June 2007.
[24] T. ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, 31:469–472, 1985.
[25] R. Gennaro, C. Gentry, and B. Parno. Non-interactive verifiable computing: Outsourcing computation to untrusted workers. In *CRYPTO*, 2010.
[26] R. Gennaro, C. Gentry, B. Parno, and M. Raykova. Quadratic span programs and succinct NIZKs without PCPs. Apr. 2012. Cryptology eprint 215.
[27] R. Gennaro and D. Wichs. Fully homomorphic message authenticators. May 2012. Cryptology eprint 290.
[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] D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1):5–48, Mar. 1991.
[31] S. Goldwasser, Y. T. Kalai, and G. N. Rothblum. Delegating computation: Interactive proofs for muggles. In *STOC*, 2008.
[32] S. Goldwasser, S. Micali, and C. Rackoff. The knowledge complexity of interactive proof systems. *SIAM J. on Comp.*, 18(1):186–208, 1989.
[33] J. Groth. Linear algebra with sub-linear zero-knowledge arguments. In *CRYPTO*, 2009.
[34] Y. Huang, D. Evans, J. Katz, and L. Malka. Faster secure two-party computation using garbled circuits. In *USENIX Security*, 2011.
[35] Y. Ishai, E. Kushilevitz, and R. Ostrovsky. Efficient arguments without short PCPs. In *Conference on Computational Complexity (CCC)*, 2007.
[36] K. Jang, S. Han, S. Han, S. Moon, and K. Park. SSLShader: Cheap SSL acceleration with commodity processors. In *NSDI*, 2011.
[37] J. Kilian. A note on efficient zero-knowledge proofs and arguments (extended abstract). In *STOC*, 1992.
[38] J. Kilian. Improved efficient arguments (preliminary version). In *CRYPTO*, 1995.
[39] H. Lipmaa. Progression-free sets and sublinear pairing-based non-interactive zero-knowledge arguments. In *TCC*, 2012.
[40] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella. Fairplay—a secure two-party computation system. In *USENIX Security*, 2004.
[41] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of applied cryptography*. CRC Press, 2001.
[42] R. C. Merkle. Digital signature based on a conventional encryption function. In *CRYPTO*, 1987.
[43] S. Micali. Computationally sound proofs. *SIAM J. on Comp.*, 30(4):1253–1298, 2000.
[44] M. Naehrig, K. Lauter, and V. Vaikuntanathan. Can homomorphic encryption be practical? In *ACM Workshop on Cloud Computing Security*, 2011.
[45] B. Parno, C. Gentry, J. Howell, and M. Raykova. Pinocchio: Nearly practical verifiable computation. In *IEEE Symposium on Security and Privacy*, May 2013. To appear.
[46] A. Polishchuk and D. A. Spielman. Nearly-linear size holographic proofs. In *STOC*, 1994.
[47] S. Setty, A. J. Blumberg, and M. Walfish. Toward practical and unconditional verification of remote computations. In *HotOS*, 2011.
[48] S. Setty, R. McPherson, A. J. Blumberg, and M. Walfish. Making argument systems for outsourced computation practical (sometimes). In *NDSS*, 2012.
[49] 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). Oct. 2012. Cryptology eprint 2012/598.
[50] H. Shacham and B. Waters. Compact proofs of retrievability. In *Asiacrypt*, Dec. 2008.
[51] N. Smart and F. Vercauteren. Fully homomorphic SIMD operations. Aug. 2011. Cryptology eprint 133.
[52] J. Thaler. Personal communication, June 2012.
[53] J. Thaler, M. Roberts, M. Mitzenmacher, and H. Pfister. Verifiable computation with massively parallel interactive proofs. In *USENIX HotCloud Workshop*, June 2012.
[54] C. Wang, K. Ren, J. Wang, and K. M. R. Urs. Harnessing the cloud for securely outsourcing large-scale systems of linear equations. In *Intl. Conf. on Dist. Computing Sys. (ICDCS)*, 2011.
[55] D. A. Wheeler. SLOCCount.
[56] A. C.-C. Yao. How to generate and exchange secrets. In *FOCS*, 1986.

Commit+Multidecommit

The protocol assumes an additive homomorphic encryption scheme (Gen, Enc, Dec) over a finite field, $\mathbb{F}$.

Commit phase

Input: Prover holds a vector $w \in \mathbb{F}^n$, which defines a linear function $\pi\colon \mathbb{F}^n \to \mathbb{F}$, where $\pi(q) = \langle w, q \rangle$.

1. Verifier does the following:
   - Generates public and secret keys $(pk, sk) \leftarrow \mathsf{Gen}(1^k)$, where k is a security parameter.
   - Generates vector $r \in_R \mathbb{F}^n$ and encrypts r component-wise, so $\mathsf{Enc}(pk, r) = (\mathsf{Enc}(pk, r_1), \ldots, \mathsf{Enc}(pk, r_n))$.
   - Sends $\mathsf{Enc}(pk, r)$ and pk to the prover.
2. Using the homomorphism in the encryption scheme, the prover computes $e \leftarrow \mathsf{Enc}(pk, \pi(r))$ without learning r. The prover sends e to the verifier.
3. The verifier computes $s \leftarrow \mathsf{Dec}(sk, e)$, retaining s and r.

Decommit phase

Input: the verifier holds $q_1, \ldots, q_\mu \in \mathbb{F}^n$ and wants to obtain $\pi(q_1), \ldots, \pi(q_\mu)$.

4. The verifier picks μ secrets $\alpha_1, \ldots, \alpha_\mu \in_R \mathbb{F}$ and sends to the prover $(q_1, \ldots, q_\mu, t)$, where $t = r + \alpha_1 q_1 + \cdots + \alpha_\mu q_\mu \in \mathbb{F}^n$.
5. The prover returns $(a_1, a_2, \ldots, a_\mu, b)$, where $a_i, b \in \mathbb{F}$. If the prover behaved, then $a_i = \pi(q_i)$ for all $i \in [\mu]$, and $b = \pi(t)$.
6. The verifier checks: $b \stackrel{?}{=} s + \alpha_1 a_1 + \cdots + \alpha_\mu a_\mu$. If so, it outputs $(a_1, a_2, \ldots, a_\mu)$. If not, it rejects, outputting $\perp$.

Figure 12—The commitment protocol of PEPPER [48], which generalizes a protocol of Ishai et al. [35]. $q_1, \ldots, q_\mu$ are the PCP queries, and n is the size of the proof encoding. The protocol is written in terms of an additive homomorphic encryption scheme, but as stated elsewhere [35, 48], the protocol can be modified to work with a multiplicative homomorphic scheme, such as ElGamal [24].

A Protocol refinements in GINGER

This section describes the base protocols (A.1), states and analyzes GINGER's modifications (A.2), and describes how GINGER compresses queries to save network costs (A.3).

A.1 Base protocols

GINGER uses a linear commitment protocol from PEPPER [48]; this protocol is depicted in Figure 12.[7] As described in Section 2.3, PEPPER composes this protocol and a linear PCP; that PCP is depicted in Figure 13. The purpose of $\{\gamma_0, \gamma_1, \gamma_2\}$ in this figure is to make a maliciously constructed oracle unlikely to pass the circuit test; to generate the $\{\gamma_i\}$, V multiplies each constraint by a random value and collects like terms, a process described in [5, 13, 35, 48]. The completeness and soundness of this PCP are explained in those sources, and our notation is borrowed from [48]. Here we just assert that the soundness error of this PCP is $\epsilon = (7/9)^\rho$; that is, if the proof π is incorrect, the verifier detects that fact with probability greater than $1 - \epsilon$. To make $\epsilon \approx 10^{-6}$, PEPPER requires $\rho = 55$.

A.2 GINGER's PCP modifications

GINGER retains the (P, V) argument system of PEPPER [48] but uses a modified PCP protocol (depicted in Figure 14) that makes the following changes to the base PCP protocol (Figure 13):

- Recycle queries [9].
- Amplify linearity queries and tests.
- Make fewer PCP runs.

We analyze the soundness of these changes below. Although this analysis is not a theoretical contribution (it is an application of well-known techniques), we include it below for two reasons. The first is completeness. The second is that the choice of parameters (e.g., the number of repetitions of each kind of test) requires care, so it is worthwhile to "show our work."

Lemma A.1 (Soundness of GINGER.)**.** The soundness error of GINGER is upper-bounded by

$$\epsilon_G = \kappa^\rho + 2 \cdot \mu \cdot (2\sqrt[3]{9/2} + 1) \cdot \sqrt[3]{\frac{1}{|\mathbb{F}|}} + \epsilon_S,$$

where κ will be constrained below, μ is the number of PCP queries, and ϵ_S is the error from semantic security.

Proof. We begin by bounding the soundness error of GINGER's PCP protocol (Figure 14). To do so, we use the linearity testing results of Bellare et al. [8, 9] and the terminology of [8]. Define $\mathsf{Dist}(f, g)$ to be the fraction of inputs on which f and g disagree. Define $\mathsf{Dist}(f)$ to be the fraction of inputs on which f disagrees with its "closest linear function" [8]. Define $\mathsf{Rej}(f)$ to be the probability, over uniformly random choices of x and y from the domain of f, that $f(x) + f(y) \neq f(x+y)$; $\mathsf{Rej}(f)$ is the probability that f fails the BLR linearity test [17]. As stated by Bellare et al. [8]:

- If $\mathsf{Dist}(f) = \delta$, then $\mathsf{Rej}(f) \geq 3\delta - 6\delta^2$.
- If $\mathsf{Dist}(f) \geq \frac{1}{4}$, then $\mathsf{Rej}(f) \geq \frac{2}{9}$.

[7] Like PEPPER, GINGER verifies in batches (§2.3), which changes the protocols a bit; see [48, Appendix C] for details.

The linear PCP from [5]

Loop ρ times:

- Generate linearity queries: Select $q_1, q_2 \in_R \mathbb{F}^s$ and $q_4, q_5 \in_R \mathbb{F}^{s^2}$. Take $q_3 \leftarrow q_1 + q_2$ and $q_6 \leftarrow q_4 + q_5$.
- Generate quadratic correction queries: Select $q_7, q_8 \in_R \mathbb{F}^s$ and $q_{10} \in_R \mathbb{F}^{s^2}$. Take $q_9 \leftarrow (q_7 \otimes q_8 + q_{10})$.
- Generate circuit queries: Select $q_{12} \in_R \mathbb{F}^s$ and $q_{14} \in_R \mathbb{F}^{s^2}$. Take $q_{11} \leftarrow \gamma_1 + q_{12}$ and $q_{13} \leftarrow \gamma_2 + q_{14}$.
- Issue queries. Send $q_1, \ldots, q_{14}$ to oracle π, getting back $\pi(q_1), \ldots, \pi(q_{14})$.
- 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`.
- Quadratic correction test: Check that $\pi(q_7) \cdot \pi(q_8) = \pi(q_9) - \pi(q_{10})$. If not, `reject`.
- Circuit test: Check that $(\pi(q_{11}) - \pi(q_{12})) + (\pi(q_{13}) - \pi(q_{14})) = -\gamma_0$. If not, `reject`.

If V makes it here, `accept`.

Figure 13—The linear PCP that PEPPER uses. It is from [5]. The notation $x \otimes y$ refers to the outer product of two vectors x and y (meaning the vector or matrix consisting of all pairs of components from the two vectors). The values $\{\gamma_0, \gamma_1, \gamma_2\}$ are described briefly in the text.

Claim A.2. For all $\delta \in \{\delta \mid 3\delta - 6\delta^2 < \frac{2}{9}$ and $0 \leq \delta \leq \frac{1}{4}\}$, if $\mathsf{Rej}(f) \leq 3\delta - 6\delta^2$, then $\mathsf{Dist}(f) \leq \delta$.

Proof. This follows directly from Bellare et al. [8]. Fix δ. Assume to the contrary that $\mathsf{Dist}(f) > \delta$. Case I: $\mathsf{Dist}(f) < \frac{1}{4}$. Then $\mathsf{Rej}(f) \geq 3 \cdot \mathsf{Dist}(f) - 6 \cdot (\mathsf{Dist}(f))^2 > 3\delta - 6\delta^2$. Case II: $\mathsf{Dist}(f) \geq \frac{1}{4}$. Then $\mathsf{Rej}(f) \geq \frac{2}{9} > 3\delta - 6\delta^2$. Thus, both cases lead to $\mathsf{Rej}(f) > 3\delta - 6\delta^2$, which contradicts the given. □

We say that f is δ-close to linear, if $\mathsf{Dist}(f) \leq \delta$. Let δ^* be the lesser root of $3\delta - 6\delta^2 = 2/9$.

Corollary A.3. Let E be the event that f passes the BLR test. For $0 < \delta < \delta^*$, if $\Pr\{E\} > 1 - 3\delta + 6\delta^2$, then f is δ-close to linear.

Corollary A.4. Let T refer to a test that performs ρ_{lin} independent BLR linearity tests. For $\delta < \delta^*$, if $\Pr\{f \text{ passes } T\} > (1 - 3\delta + 6\delta^2)^{\rho_{\text{lin}}}$, then f is δ-close to linear.

Having considered the soundness error of linearity testing, we now consider the soundness error of the PCP.

Claim A.5. Choose $0 < \delta < \delta^*$. Choose $\kappa > \max\{(1 - 3\delta + 6\delta^2)^{\rho_{\text{lin}}}, 4\delta + 2/|\mathbb{F}|\}$. If a purported proof oracle π for constraints $\mathcal{C}$ passes one iteration of the tests in Figure 14 with probability $\geq \kappa$, then $\mathcal{C}$ is satisfiable.

GINGER's PCP protocol

Loop ρ times:

- Generate linearity queries: select $q_4, q_5 \in_R \mathbb{F}^s$ and $q_7, q_8 \in_R \mathbb{F}^{s^2}$. Take $q_6 \leftarrow q_4 + q_5$ and $q_9 \leftarrow q_7 + q_8$. Perform $\rho_{\text{lin}} - 1$ more iterations of this step.
- Generate quadratic correction queries, by reusing randomness of linearity queries: Take $q_1 \leftarrow (q_4 \otimes q_5 + q_7)$.
- Generate circuit queries, again reusing randomness of linearity queries: Take $q_2 \leftarrow \gamma_1 + q_4$. Take $q_3 \leftarrow \gamma_2 + q_8$.
- Issue queries. Send $(q_1, \ldots, q_{3+6\rho_{\text{lin}}})$ to oracle π, getting back $\pi(q_1), \ldots, \pi(q_{3+6\rho_{\text{lin}}})$.
- Linearity tests: Check that $\pi(q_4) + \pi(q_5) = \pi(q_6)$ and $\pi(q_7) + \pi(q_8) = \pi(q_9)$, and likewise for the other $\rho_{\text{lin}} - 1$ iterations. If not, `reject`
- Quadratic correction test: Check that $\pi(q_4) \cdot \pi(q_5) = \pi(q_1) - \pi(q_7)$. If not, `reject`.
- Circuit test: Check that $(\pi(q_2) - \pi(q_4)) + (\pi(q_3) - \pi(q_8)) = -\gamma_0$. If so, `accept`.

If V makes it here, `accept`.

Figure 14—GINGER's PCP protocol, which refines PEPPER's protocol (Figure 13). This protocol recycles queries [9] and amplifies linearity testing.

Proof. (Sketch.) From Corollary A.4 and the given, π is δ-close to linear. We can now apply the proof flow that establishes the soundness of linear PCPs, as in [5]. (A self-contained example is in Appendix D of [48].) Specifically, since π is δ-close to linear and the probability of passing the quadratic correction test is greater than $4\delta + 2/|\mathbb{F}|$, then π's closest linear function is of the right form. Since π's closest linear function is of the right form and since since π's probability of passing the circuit test is greater than $4\delta + 1/|\mathbb{F}|$, then $\mathcal{C}$ is satisfiable.

We are able to recycle queries within a PCP run because, as Bellare et al. [9] observe, the preceding analysis does not require that the quadratic correction and circuit tests are independent of the linearity tests. □

Claim A.6. The soundness of the PCP in Figure 14 is at least $1 - \kappa^{\rho}$.

Proof. Assume $\mathcal{C}$ is not satisfiable. By Claim A.5, the probability that any π passes one iteration is less than κ. This implies that for all π, the probability of passing ρ iterations is less than κ^{ρ}. □

To complete the proof of the lemma, we consider the soundness error of the PCP and the soundness error of the commitment protocol. The analysis is very similar to that of [35, 48]. Lemma B.2 in [48] implies that, in a run of the commitment protocol with μ queries, the GINGER verifier can regard all answers as being given by a fixed oracle, π, except with probability $\epsilon_C = \mu \cdot 2 \cdot (2\sqrt[3]{9/2} + 1) \cdot \sqrt[3]{\epsilon_B}$, where ϵ_B is $1/|\mathbb{F}| + \epsilon_S$.[8]

Claim A.6 implies that if the constraints in question ($\mathcal{C}$) are not satisfiable, then the GINGER verifier passes the PCP checks on π (the fixed function implied by Lemma B.2 in [48]) with probability upper-bounded by κ^ρ. Thus, the total soundness error is upper-bounded by ϵ_G, as claimed. □

We now compute GINGER's soundness for suitable parameter choices. For a given target soundness error, the total number of queries $\rho \cdot (3 + 6\rho_{\text{lin}})$ is roughly constant (this is an empirical claim, not a mathematical one), even as ρ_{lin} and ρ vary. Our approach is to use our cost model (Figure 2) to choose a value of δ that (through its influence on ρ_{lin} and ρ) results in the lowest break-even batch sizes. We obtain low values of ρ and high values of ρ_{lin}; the reason is that linearity queries are much less expensive for the verifier than the other PCP queries (as noted in Section 3), so the optimization favors them.

In more detail, our target upper bound on soundness error is 10^{-6}, and we take $|\mathbb{F}| = 2^{128}$. The cost model leads us to $\delta = 0.041$, which yields $\rho_{\text{lin}} = 15$, and hence $\kappa = 0.166$ suffices. For our target soundness, $\rho = 8$ suffices, and we get $\kappa^\rho < 5.8 \cdot 10^{-7}$ and $\mu = 744$. (Note that although there are hundreds of PCP queries, the high-order cost comes from the encrypted query; see Section 3 and [49, Appendix E].) Following [48], we neglect ϵ_S. Applying Lemma A.1, we get $\epsilon_G < (5.8 \cdot 10^{-7} + 9.2 \cdot 10^{-10}) \approx 5.8 \cdot 10^{-7}$.

A.3 Compressing queries

As stated in Section 3, our protocol compresses queries to save network costs. Specifically, V sends to P a short seed that is used as a key to a pseudorandom generator to generate shared pseudorandomness, and then both parties derive the PCP queries (Figure 14). Intuitively, the fact that the key to generate the randomness is shared is reasonable, since the key is revealed *after* the prover has been bound to a function.

This section establishes the soundness of this protocol in the random oracle model. Our starting point is the idealized version of the protocol that is presented in Sections A.1 and A.2. We call this protocol GINGER-PURE and give it below, after establishing some preliminaries.

We can think of the PCP queries (Figure 14) as being generated by randomness together with an efficient function G that "structures the randomness" to produce the material for the PCP queries and checks. Specifically, let M be the number of random field elements that are required by GINGER's PCP protocol, let $\mu = 3 + 6\rho_{\text{lin}}$ be the number of PCP queries, and let N be the number of field elements in $\{\gamma_0, \gamma_1, \gamma_2, q_1, \ldots, q_\mu\}$ (from Figure 14). Then $G\colon \mathbb{F}^M \to \mathbb{F}^N$ takes as input M field elements $h_1, \ldots, h_M$ (which are random "coin flips") and returns the PCP query and checking material (the q_i and γ_i). As notation, let $G_i(h_1, \ldots, h_M)$ be the ith query (q_i).

Definition A.7 (GINGER-PURE protocol). Let $h\colon \{0,1\}^{\log M} \to \{0,1\}^{\log|\mathbb{F}|}$ be a random function (this models the "coin flips"). Then the GINGER-PURE protocol is as follows:

- V and P have the same input as, and run steps 1–3 of, Commit+Multidecommit (Figure 12).
- V generates random field elements $h_1, \ldots, h_M$ as $h(1), \ldots, h(M)$. Using these values as input to G (which captures the logic in the first part of Figure 14), V derives $\{q_1, \ldots, q_\mu\}$ and the $\{\gamma_i\}$.
- V and P follow steps 4–6 of Commit+Multidecommit.
- If V makes it to here, it uses $(a_1, \ldots, a_\mu)$ from Commit+Multidecommit as inputs to the PCP checks (second part of Figure 14).

The rest of this section focuses on the soundness of our implemented protocol, defined immediately below.[9]

Definition A.8 (GINGER-IMPL protocol). Let $H\colon \{0,1\}^{\log|\mathbb{F}|} \to \{0,1\}^{M\cdot\log|\mathbb{F}|}$ be a hash function (this generates M field elements, from a seed). GINGER-IMPL proceeds as follows:

- V and P have the same input as, and run steps 1–3 of, Commit+Multidecommit (Figure 12).
- V randomly chooses a seed k and breaks $H(k)$ into M values, $H(k)_1, \ldots, H(k)_M$. Using G (which captures the logic in the first part of Figure 14), V derives $\{q_1, \ldots, q_\mu\}$ and the $\{\gamma_i\}$.
- V picks μ secrets $\alpha_1, \ldots, \alpha_\mu \in_R \mathbb{F}$ and computes the consistency query $t = r + \sum_{i=1}^{\mu} \alpha_i q_i \in \mathbb{F}^n$. V sends to the prover the seed k and the consistency query t.
- P uses k to obtain $H(k)_1, \ldots, H(k)_M$ and then derives the PCP queries via G. P responds to the queries, returning $(a_1, \ldots, a_\mu, b)$ as in step 5 in Commit+Multidecommit.
- V follows step 6 of Commit+Multidecommit.
- If V makes it to here, it uses $(a_1, \ldots, a_\mu)$ as inputs to the PCP checks (second part of Figure 14).

[8] We run the commitment protocol twice, for functions $\pi^{(1)}$ and $\pi^{(2)}$, but the number of queries presented to each is, say, $\mu/2$. So a union bound on the commitment error of each oracle again yields ϵ_C.

[9] Our implemented protocol actually differs slightly from GINGER-IMPL: in the encryption step and consistency query construction, the protocol uses pseudorandomness (with a seed different from the revealed one) in place of randomness. However, this use of pseudorandomness is standard, so we ignore it.

Below, we will argue the soundness, in turn, of GINGER-RAW (GINGER-PURE modified to send the random choices themselves, instead of the queries, from V to P) and the idealized variant GINGER-RO (in which V sends a short key, and we work in the random oracle model). GINGER-IMPL is then an instantiation of GINGER-RO.

Soundness of GINGER-RAW

Definition A.9 (GINGER-RAW protocol). The GINGER-RAW protocol is the same as GINGER-PURE, except that steps 4 and 5 in Figure 12 are different. V derives $q_1, \ldots, q_\mu$ just as before, and obtains t just as before, except now V sends $(h_1, \ldots, h_M, t)$ to P, versus sending $(q_1, \ldots, q_\mu, t)$. P uses the h_i to derive the q_i (via G), and then responds to the queries.

We will show that GINGER-RAW is an argument system.

Definition A.10 (CFMD-raw). This is the same as the CFMD (commitment to function with multiple decommitments) in Definition B.1 in PEPPER [48], except that $\mathcal{E}$ generates $h_1, \ldots, h_M$, not Q. There is also a query generation function $G^{(q)}: \mathbb{F}^M \to \mathbb{F}^{n\times\mu}$ where $G^{(q)}(h_1, \ldots, h_M) = (q_1, \ldots, q_\mu) \stackrel{\text{def}}{=} Q$. The other change is that in the setup of the ϵ_B-binding property, the environment produces two M-tuples $(h_1, \ldots, h_M)$ and $(\hat{h}_1, \ldots, \hat{h}_M)$ that generate two queries Q and $\hat{Q}$ (rather than R sending Q and $\hat{Q}$).

We can instantiate the above definition with Commit+Multidecommit-raw, which modifies Commit+Multidecommit as follows: V's input in the decommit phase is an M-tuple $(h_1, \ldots, h_M) \in \mathbb{F}^M$; in step 4, V sends this M-tuple and the consistency query t; and in step 5, P uses these values to derive the $\{q_i\}$, which it responds to as usual.

Lemma A.11. Commit+Multidecommit-raw is a CFMD-raw protocol with $\epsilon_B = 1/|\mathbb{F}| + \epsilon_S$, where ϵ_S comes from the semantic security of the homomorphic encryption scheme.

The proof is nearly the same as the proof of Lemma B.1 in [48], and is omitted.

Claim A.12. GINGER-RAW is an argument system with soundness error upper-bounded by ϵ_G.

Proof. (Sketch.) Commit+Multidecommit-raw works with $G^{(q)}$, which is the the "query part" of the output of G. The binding property of Commit+Multidecommit-raw implies that, after commitment, the prover is bound to a function, $\tilde{f}_v(\cdot)$, from queries to outputs. (The proof is nearly identical to the proof of Lemma B.2 in [48], replacing CFMD with CFMD-raw.) Moreover, we have not altered the PCP from Figure 14. Thus, we can rerun Lemma A.1, applying it to GINGER-RAW instead of GINGER-PURE. □

Soundness of GINGER-RO

Definition A.13 (GINGER-RO protocol). GINGER-RO is nearly the same as GINGER-IMPL. The differences are twofold. First, GINGER-RO includes a random oracle $\mathcal{R}$, and after the commit phase V chooses a key k uniformly at random from $\{0,1\}^{\log|\mathbb{F}|}$. Second, where GINGER-IMPL uses $H(k)_1, \ldots, H(k)_M$, GINGER-RO uses values $h_1, \ldots, h_M$ that are generated by the random oracle, as the expansion of $\mathcal{R}(k)$.

We can define a protocol very similar to CFMD-raw and Commit+Multidecommit-raw to obtain versions of Lemma A.11 and Claim A.12. The claim would need to add some soundness error to bound the probability that the prover guesses k.

GINGER-IMPL is then an instantiation of GINGER-RO. Specifically, we implement the random oracle using a stream cipher as a pseudorandom generator. We note that this kind of query compression (and security proofs in the random oracle model) have been proposed before [50].