Taking proof-based verified computation a few steps closer to practicality (extended version)[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 the query size and costs via theoretical refinements that are of independent interest; 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, 19, 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 [21, 44, 45, 49]. More generally, there has been a groundswell of work that aims for potentially practical verifiable outsourced computation, using theoretical tools [11, 12, 20, 24, 25].

Nonetheless, these works have notable limitations. Only a handful [21, 44, 45, 49] 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 [45] (which is summarized in Section 2). GINGER's contributions are as follows.

(1) GINGER *demonstrates the strength of linear commitment (§3).* This paper proves that PEPPER's commitment primitive [45], which generalizes the commitment primitive of Ishai et al. [35], is surprisingly powerful: it not only commits an untrusted entity to a function and extracts evaluations of that function (as previously shown) but also ensures that the function is linear. (The primitive embeds a *strong linearity test*.) This result sharply reduces the required number of queries (from 500 to 3) and a key error bound, and hence overhead.

(2) GINGER *supports a general-purpose programming model (§4).* Although the model does not handle looping concisely, it includes primitive floating-point quantities,

[1]This paper, UT CS technical report 12-14, is an extended version of a publication at Usenix Security [46]. This version includes four Appendices (B–E) that were elided for space.

inequality comparisons, logical expressions, and conditional control flow. Moreover, we have a compiler (derived from Fairplay [39]) 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 protocol); 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. 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.[2]

We have implemented and evaluated GINGER (§6). Compared to PEPPER [45], 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 and possibly orders of magnitude, depending on the cost of encryption; if we model encryption as free, the verifier can gain from outsourcing when batch-verifying as few as 20 computations (down from 3900 in PEPPER). The prover's CPU costs drop by 10–15%, which is not much, but our parallel implementation reduces latency with near-linear speedup. Computing with rational numbers in GINGER is roughly three 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 23 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 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 [40]) 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]

[2] 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).

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 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 [45] 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.

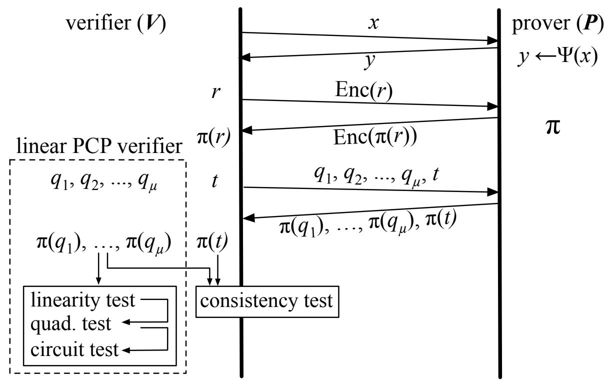

Figure 1—The PEPPER protocol [45], which is GINGER's base. Though not depicted, many of the protocol steps happen in parallel, to facilitate batching.

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 [45].

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 $\epsilon > 0$. A key test for us—we return to it in Section 3—is the *linearity test* [16]: 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, 45].

2.3 Our base: PEPPER

We now walk through the three phases of PEPPER [45], 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 [45] (although they need not be 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 [27] (which remains unfeasibly expensive); PEPPER uses ElGamal [23, 45].

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, 45]). 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 [19, 32] below are borrowed from [35, 45].

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 refinements eliminate many queries, by relying on a new analysis of the base commitment primitive. To motivate this analysis, we note that there is something seemingly redundant in the base machinery (see Figure 1): why does the linear PCP require a linearity test (§2.2) if an honest prover depends on the linearity of its function π to pass the linear commitment protocol's consistency test (§2.3)? Can we remove one of these tests, or combine them somehow? The reason that PEPPER appears to need both tests is that their guarantees are (so far) subtly different:

- *Consistency test (§2.3)*: First, an honest prover is guaranteed to pass this test. Second, if the prover—even a cheating one—passes this test, then it is very likely bound to *some* function (as shown in [35, 45]).
- *Linearity test (§2.2)*: This test is needed in case the prover cheats—it establishes that π is linear (as required by the rest of the PCP protocol). More accurately, if π is far from being linear, the test is somewhat likely to catch that case.

Yet, it seems unsatisfying that both tests are required when composing linear commitment and the linear PCP: can a prover really pass the consistency test systematically with a function that the linearity test would reject? In fact, our intuitive dissatisfaction is well-founded: this paper proves that the commitment primitive (which includes the consistency test) is far stronger than the linearity test. Put simply, even a cheating prover is very likely bound to a function that is linear, or almost so.

Practically, this result saves query generation and response costs. For one thing, we can eliminate linearity tests from the protocol. More significantly, we eliminate *amplification*: PEPPER needed to repeat the protocol to turn the linearity test's guarantee of "somewhat likely" into "very likely". In contrast, our result already gives a guarantee of "very likely", so no repetition is required.

More broadly, this result means that the commitment primitive is considerably more powerful than was realized—it efficiently commits an untrusted entity to a linear function and extracts evaluations of that function—and may apply elsewhere.

Details. The protocol refinements are rooted in a strengthened soundness analysis. *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.

The soundness of the PCP protocol in Section 2.2 and Appendix A.1 is connected to the soundness of linearity testing [16]. 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 $7/9$ derives from a result [8] that if the proof oracle is not "somewhat close" to linear, then the linear-

	PEPPER [45]	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	$\rho\cdot(\chi\cdot f+\ell'\cdot f+5c)\cdot n/\beta$	$(\chi\cdot f+\ell\cdot f+2c)\cdot n/\beta$
Process PCP responses	$\rho\cdot(2\ell'+\lvert x\rvert+\lvert y\rvert)\cdot f$	$(2\ell+\lvert x\rvert+\lvert y\rvert)\cdot f$
***P*'s per-instance CPU costs**		
Issue commit responses	$h\cdot n$	$h\cdot n$
Issue PCP responses	$(\rho\cdot\ell'+1)\cdot f\cdot n$	$(\ell+1)\cdot f\cdot n$
Network cost (per instance)	$((\rho\cdot\ell'+1)\cdot\lvert p\rvert+\lvert\xi\rvert)\cdot n/\beta$	$((\ell+1)\cdot\lvert p\rvert+\lvert\xi\rvert)\cdot n/\beta$
PCP soundness error	$(7/9)^\rho = 2.3\cdot 10^{-8}$	$\kappa = 2.6\cdot 10^{-6}$
Overall soundness error	$2.4\cdot 10^{-8}$	$4.5\cdot 10^{-6}$

$|x|$, $|y|$: # 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)
$\ell = 3$: # of high-order PCP queries in GINGER (§A.2, §A.3)
$\ell' = 7$: # of high-order PCP queries in PEPPER (§A.1)
$\rho = 70$: # of PCP reps. in base scheme (§A.1)
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 to generate 192-bit pseudorandom #
$|p|$: length of an element in $\mathbb{F}$
$|\xi|$: length of an encrypted element in $\mathbb{F}$

Figure 2—High-order costs and error in GINGER, compared to its base (PEPPER [45]), 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 $|\mathbb{F}| = 2^{128}$. Many of the cryptographic costs enter through the commitment protocol (see Section 2.3 or Figure 12); Section 6 quantifies the parameters. The "PCP" row include the consistency query and check. The network costs slightly underestimate by not including query responses.

ity test passes with probability $< 7/9$ (though fascinating, this result is inconveniently weak in our context).

Our analysis, detailed in Appendix A.2, establishes that the commitment protocol binds the prover to a function that is extremely close to linear (otherwise, the prover could break the semantic security of the homomorphic encryption used by GINGER and PEPPER). This results in the PCP soundness error improving from $7/9$ to κ, where $\kappa \approx 4\sqrt[6]{1/|\mathbb{F}|}$; this analysis does not depend on linearity tests, so they can be dropped.

The soundness error is somewhat low by cryptographic standards, but in practice, a failure rate (when the prover is malicious) of 1 in 200,000 is reasonable.

A further optimization. GINGER reuses some queries across the quadratic correction and circuit tests; this refinement is detailed and justified in Appendix A.3.

Savings. Most significantly, V can take advantage of the lower soundness error to run $\rho = 1$ instead of $\rho = 70$ repetitions of the PCP protocol. Also, per repetition, V's work to generate pseudorandom queries decreases by $3/5$ ($2/5$ coming from the elimination of linearity tests and $1/5$ from reusing queries). These gains are depicted in Figure 2, most notably in the reduction from $\rho\cdot\ell' \approx 500$ to $\ell = 3$ total PCP queries.

The total savings for the verifier depend on the relative cost of pseudorandom number generation (encapsulated by c) and encryption (encapsulated by e). These savings show up in β^*, the minimum batch size (§2.3) at which V gains from outsourcing. As shown in Section 6.1, the reduction in β^* can be several orders of magnitude (when e is small). Finally, taking $|p| = 128$ bits and $|\xi| = 2\cdot 1024$ bits, the savings in network costs are 1–2 orders of magnitude (holding β constant).

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[3] apply to the many protocols that use the constraint formalism or arithmetic circuits. Moreover, we have implemented a compiler (derived from Fairplay's [39]) 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.

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

[3] 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.

three steps, as illustrated and described below:

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

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 [17, 50]). 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). 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 $N'_b = 2N_b$ (shown in Appendix B).

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 $\mathbb{Q}$ (shown in Appendix B). For our U above, $p > (m+1)^2 \cdot 2^{4(N_a+N_b)}$ 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 $N_a + \log N_q$ bits [29] while GINGER requires $2 \cdot (\max(N_a, N_q) + 1)$ bits (shown in 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^{-32} (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 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 $2\log_2(m+1) + 256$ bits. Roughly speaking, the end-to-end costs are $3\times$ 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 (see 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 $x_1 < x_2$ (shown in 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, $\mathcal{C}_\Psi$; these constraints are satisfiable if and only if the prover correctly computes Ψ (shown in Appendix C).

```
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.

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.

Limitations and costs. We compile a subset of SFDL, the language of the Fairplay compiler [39]. 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 degree-2 (§2.1), which carries costs. Specifically, if $\mathcal{C}$ has s variables and χ constraints, an equivalent degree-2 representation of $M\{\mathcal{C}\}$ has $s + \chi$ variables and $2 \cdot \chi$ constraints (shown in 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 [49] for an independent study of GPU hardware in the context of [21]). 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. 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 [39] to constraints; it is derived from Fairplay and is implemented in 5294 lines of Java, starting from Fairplay's 3886 lines (per [51]). 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 [45] introduced specialized 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 [23] with 1024-bit keys.

GINGER's protocol refinements reduce per-instance network costs by 25–30× (to hundreds of KBs for the computations we study), prover CPU costs by about 10–14% (leaving them still high), and break-even batch size (β^*) by about 4×.	§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 20 (tens of seconds of computation).	§6.1
Rational arithmetic costs roughly 3× 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 ($N_a = 32$, $N_b = 32$)	320 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 ($N_a = 32$, $N_b = 5$)	320 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.

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 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 Tesla M2070. Each GPU has 448 CUDA cores and 6GB 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 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. However, local computation costs about 1.2–4.0 times more than is predicted; we think that the divergence results from adverse caching effects that increase the cost of a multiplication. Thus, we expect the verifier to break even at batch sizes that are about a factor of 1.2–4.0 smaller than predicted by the model.

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. For matrix multiplication, these are about the same as the cost to send the

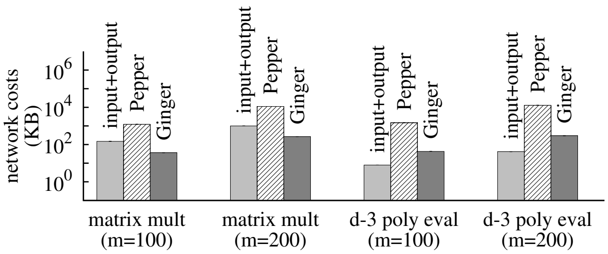

Figure 6—Per-instance network costs of GINGER and its base (PEPPER [45]), compared to the size of the inputs and outputs. At this batch size ($\beta = 5000$), GINGER's refinements reduce per-instance network costs by a factor of 25–30 compared to PEPPER. GINGER's network costs here are hundreds of KB or less. The y-axis is log-scaled.

		PEPPER	GINGER
	local	1.1 s	1.1 s
CPU	β^*	13000	4100
	verifier aggregate	3.9 hr	1.3 hr
	prover aggregate	5.0 yr	1.6 yr
	prover per-instance	3.5 hr	3.3 hr
GPU	β^*	8700	2300
	verifier aggregate	2.7 hr	43.4 min
	prover aggregate	3.5 yr	320 days
	prover per-instance	3.5 hr	3.3 hr
crypto hardware	β^*	3900	20
	verifier aggregate	1.2 hr	22.3 s
	prover aggregate	1.6 yr	2.8 days
	prover per-instance	3.5 hr	3.3 hr

Figure 7—Break-even batch sizes (β^*) and predicted 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.

inputs and receive the outputs; for polynomial evaluation, these are about 10 times the size of the inputs and outputs. Also, GINGER improves on PEPPER by 20–30×.

In this experiment, GINGER's prover incurs about 10–14% less CPU time compared to PEPPER (estimated using a cost model from [45]) 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.

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

	mat. mult.	mat. mult. ($\mathbb{Q}$)
local	17.6 ms	5.90 ms
verifier per-instance	17.6 ms	80.2 ms
verifier aggregate	76.1 s	5.7 min
prover per-instance	3.1 min	9.4 min
prover aggregate	9.3 days	28 days

Figure 8—Predicted running times of GINGER's verifier and prover for matrix multiplication ($m = 100$), under integer and floating-point inputs, at $\beta = 4300$ (the break-even batch size for this computation over integers). The "local" row refers to GMP arithmetic for $\mathbb{Z}$ and native floating-point arithmetic for $\mathbb{Q}$. Handling rationals costs GINGER roughly 3× 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 [39] compiler. GINGER's constraints are not concise but are far more so than Boolean circuits.

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 $e = 72.1\mu s$ on a 2.5 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 180,000) and on an Intel 2.5 GHz CPU (which is 13,700), normalize by the dollar cost of the chips, and obtain that their throughput-per-dollar ratio is 1.8×. We thus (very conservatively) take $e = 72.1/1.8 = 40\mu$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 4150 (CPU), 2300 (crude GPU estimate), and 20 (crypto hardware). We also use the model to predict *V*'s and *P*'s costs at β^*, under PEPPER and GINGER. Figure 7 summarizes. GINGER is very sensitive to the value of *e* because its refinements have eliminated many of the other costs. Moreover, the aggregate verifier computing time drops significantly 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.

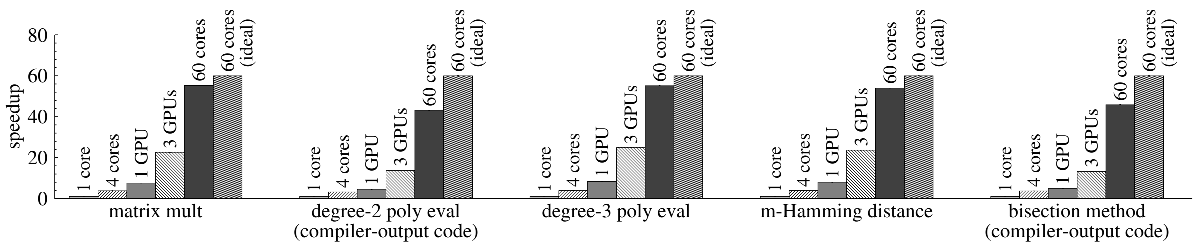

Figure 10—Latency speedup observed by GINGER's verifier when the prover is parallelized. We run with $m = 100, \beta = 150$ for matrix multiplication and degree-3 polynomial evaluation; $m = 100, \beta = 1500$ for degree-2 polynomial evaluation; $m = 100, \beta = 15$ for m-Hamming distance; and $m = 25, \beta = 15$ for bisection method. GINGER's prover achieves near-linear speedups except when the problem sizes are small and hence the overhead from parallelizing is significant (e.g., degree-2 polynomial evaluation).

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, 45]). 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 [45]. 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, 39, 52]) provide only *privacy* guarantees, but one can build on them for verifiable computation. For instance, the Boneh-Goh-Nissim homomorphic cryptosystem [18] 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 [27] has engendered protocols for verifiable non-interactive computation [20, 24, 26]. However, despite striking improvements [28, 42, 47], 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, 15, 32, 37, 38, 41], and the asymptotics continue to improve [13, 14, 22, 43]. 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. [25] 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 [44, 45] built upon the efficient argument system of Ishai et al. [35]. In contrast, Cormode, Mitzenmacher, and Thaler [21] (hereafter, CMT) built upon the protocol of Goldwasser et al. [31], and a follow-up effort studies a GPU-based parallel implementation [49].

Comparison of GINGER and CMT [21, 49]. We compared three different implementations: *CMT-native*,

m	domain	component	CMT-native	CMT-GMP	GINGER
256	$\mathbb{Z}$	verifier	40 ms	0.6 s	0.3 s
		prover	22 min	2.5 hr	36 min
		network	88 KB	0.3 MB	1.1 MB
128	$\mathbb{Q}$	verifier	–	260 ms	190 ms
		prover	–	1.0 hr	21 min
		network	–	1.8 MB	1.4 MB

Figure 11—CMT [21] 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).

CMT-GMP, and GINGER. CMT-native refers to the code and configuration released by Thaler et al. [49]; 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 320 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 [49]. 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 [48].

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 theoretical refinements (slashing query costs and network overhead); 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: the efficiency of the verifier depends on special hardware, 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, 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.

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] M. Blum and S. Kannan. Designing programs that check their work. *J. of the ACM*, 42(1):269–291, 1995.
[16] 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.
[17] D. Boneh and D. M. Freeman. Homomorphic signatures for polynomial functions. In *EUROCRYPT*, 2011.
[18] D. Boneh, E. J. Goh, and K. Nissim. Evaluating 2-DNF formulas on ciphertexts. In *TCC*, 2005.
[19] G. Brassard, D. Chaum, and C. Crépeau. Minimum disclosure proofs of knowledge. *J. of Comp. and Sys. Sciences*, 37(2):156–189, 1988.
[20] K.-M. Chung, Y. Kalai, and S. Vadhan. Improved delegation of computation using fully homomorphic encryption. In *CRYPTO*, 2010.
[21] G. Cormode, M. Mitzenmacher, and J. Thaler. Practical verified computation with streaming interactive proofs. In *ITCS*, 2012.
[22] I. Dinur. The PCP theorem by gap amplification. *J. of the ACM*, 54(3), June 2007.
[23] T. ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, 31:469–472, 1985.
[24] R. Gennaro, C. Gentry, and B. Parno. Non-interactive verifiable computing: Outsourcing computation to untrusted workers. In *CRYPTO*, 2010.
[25] R. Gennaro, C. Gentry, B. Parno, and M. Raykova. Quadratic span programs and succinct NIZKs without PCPs. Apr. 2012. Cryptology eprint 215.
[26] R. Gennaro and D. Wichs. Fully homomorphic message authenticators. May 2012. Cryptology eprint 290.
[27] C. Gentry. *A fully homomorphic encryption scheme*. PhD thesis, Stanford University, 2009.
[28] C. Gentry, S. Halevi, and N. Smart. Homomorphic evaluation of the AES circuit. In *CRYPTO*, 2012.
[29] D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1):5–48, Mar. 1991.
[30] O. Goldreich. *Foundations of Cryptography: II Basic Applications*. Cambridge University Press, 2004.
[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] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella. Fairplay—a secure two-party computation system. In *USENIX Security*, 2004.
[40] R. C. Merkle. Digital signature based on a conventional encryption function. In *CRYPTO*, 1987.
[41] S. Micali. Computationally sound proofs. *SIAM J. on Comp.*, 30(4):1253–1298, 2000.
[42] M. Naehrig, K. Lauter, and V. Vaikuntanathan. Can homomorphic encryption be practical? In *ACM Workshop on Cloud Computing Security*, 2011.
[43] A. Polishchuk and D. A. Spielman. Nearly-linear size holographic proofs. In *STOC*, 1994.
[44] S. Setty, A. J. Blumberg, and M. Walfish. Toward practical and unconditional verification of remote computations. In *HotOS*, 2011.
[45] S. Setty, R. McPherson, A. J. Blumberg, and M. Walfish. Making argument systems for outsourced computation practical (sometimes). In *NDSS*, 2012.
[46] 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.
[47] N. Smart and F. Vercauteren. Fully homomorphic SIMD operations. Aug. 2011. Cryptology eprint 133.
[48] J. Thaler. Personal communication, June 2012.
[49] J. Thaler, M. Roberts, M. Mitzenmacher, and H. Pfister. Verifiable computation with massively parallel interactive proofs. In *USENIX HotCloud Workshop*, June 2012. Full paper at `http://arxiv.org/abs/1202.1350`, Feb. 2012.
[50] 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.
[51] D. A. Wheeler. SLOCCount.
[52] A. C.-C. Yao. How to generate and exchange secrets. In *FOCS*, 1986.

A Efficient arguments with linear PCPs but no linearity tests

Whereas previous work [35, 45] established that the commitment protocol in phases 2 and 3 of PEPPER (§2.3) binds the prover to a particular function, there were no constraints on that function. The principal result of this section is that the prover is actually bound to a function that is linear, or very nearly so. As a consequence, we can eliminate linearity testing from the PCP protocol. Furthermore, the error bound from one run of this modified PCP protocol is far stronger (lower) than was known.

This section describes the base protocols (A.1), states the refinements and proves their soundness (A.2), and describes a few other optimizations (A.3).

Commit+Multidecommit

The protocol assumes an additive homomorphic encryption scheme $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ 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 [45], 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, 45], the protocol can be modified to work with a multiplicative homomorphic scheme, such as ElGamal [23].

A.1 Base protocols

GINGER uses a linear commitment protocol that is borrowed from PEPPER [45]; this protocol is depicted in Figure 12.[4] 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, 45]. The completeness and soundness of this PCP are explained in those sources, and our notation is borrowed from [45]. 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 ϵ small, PEPPER takes $\rho = 70$.

A.2 Stronger soundness analysis and consequences

GINGER retains the (P, V) argument system of PEPPER [45] but uses a modified PCP protocol (depicted in Figure 14) that makes the following changes to the base PCP protocol (Figure 13):

- Remove the linearity queries and tests.
- Set $\rho = 1$.

Theorem A.1. The (P, V) described above is an argument system with soundness $\epsilon_G \approx \sqrt[6]{1/|\mathbb{F}|}$. (The exact value of ϵ_G depends on intermediate lemmas and will be given at the end of the section.)

[4] Like PEPPER, GINGER verifies in batches (§2.3), which changes the protocols a bit; see [45, Appendix C] for details.

We will prove this theorem at the end of this section. To build up to the proof, we first strengthen the definition of a linear commitment primitive. We note that only the third property (linearity) in the definition is new; the rest is taken from [45, Appendix B], which itself heavily borrows framing, notation, and text from Ishai et al. [35].

Definition A.1 (Commitment to a function with multiple decommitments (CFMD)). Define a two-phase experiment between two probabilistic polynomial time actors (S, R) (a sender and receiver, which correspond to our prover and verifier) in an environment $\mathcal{E}$ that generates $\mathbb{F}$, w and $Q = (q_1, \ldots, q_\mu)$. In the first phase, the *commit phase*, S has w, and S and R interact, based on their random inputs. In the *decommit phase*, $\mathcal{E}$ gives Q to R, and S and R interact again, based on further random inputs. At the end of this second phase, R outputs $A = (a_1, \ldots, a_\mu) \in \mathbb{F}^\mu$ or $\perp$. A CFMD meets the following properties:

- **Correctness**. At the end of the decommit phase, R outputs $\pi(q_i) = \langle w, q_i \rangle$ (for all i), if S is honest.
- **ϵ_B-Binding**. Consider the following experiment. The environment $\mathcal{E}$ produces two (possibly distinct) μ-tuples of queries: $Q = (q_1, \ldots, q_\mu)$ and $\hat{Q} = (\hat{q}_1, \ldots, \hat{q}_\mu)$. R and a cheating S^* run the commit phase once and two independent instances of the decommit phase. In the two instances R presents the queries as Q and $\hat{Q}$, respectively. We say that S^* *wins binding* if R's outputs at the end of the respective decommit phases are $A = (a_1, \ldots, a_\mu)$ and $\hat{A} = (\hat{a}_1, \ldots, \hat{a}_\mu)$, and for some i, j, we have $q_i = \hat{q}_j$ but $a_i \neq \hat{a}_j$. We say that the protocol meets the ϵ_B-Binding property if for all $\mathcal{E}$ and for all efficient S^*, the proba-

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.

bility of S^* winning binding is less than ϵ_B. The probability is taken over three sets of independent randomness: the commit phase and the two runnings of the decommit phase.

- ϵ_L**-Linearity**. Consider the same experiment above. We say that S^* *wins linearity* if R's outputs at the end of the respective decommit phases are $A = (a_1, \ldots, a_\mu)$ and $\hat{A} = (\hat{a}_1, \ldots, \hat{a}_\mu)$, and for some i, j, k, we have $\hat{q}_k = q_i + q_j$ but $\hat{a}_k \neq a_i + a_j$. We say that the protocol meets the ϵ_L-linearity property if for all $\mathcal{E}$ and for all efficient S^*, the probability of S^* winning linearity is less than ϵ_L. As with the prior property, the probability is taken over three sets of independent randomness: the commit phase and the two runnings of the decommit phase.

Prior work proved that Commit+Multidecommit (Figure 12) meets the first two properties above [45]. We will now show that it also meets the third property.

Lemma A.1. Commit+Multidecommit meets the definition of ϵ_L-linearity, with $\epsilon_L = 1/|\mathbb{F}| + \epsilon_S$, where ϵ_S comes from the semantic security of the homomorphic encryption scheme.

Proof. We will show that if S^* can systematically cheat, then an adversary $\mathcal{A}$ could use S^* to break the semantic security of the encryption scheme.

Let $r \in_R \mathbb{F}^n$ and $Z_1, Z_2 \in_R \mathbb{F}$ (we use $\in_R$ to mean "drawn uniformly at random from"). Semantic security

GINGER's PCP protocol

- Generate quadratic correction queries: Select $q_1, q_2 \in_R \mathbb{F}^s$ and $q_4 \in_R \mathbb{F}^{s^2}$. Define $q_3 \leftarrow (q_1 \otimes q_2 + q_4)$. Note that q_3 will not travel, as P can derive it.
- Generate circuit queries: Take $q_5 \leftarrow \gamma_1 + q_1$. Take $q_6 \leftarrow \gamma_2 + q_4$.
- Issue queries. Send $(q_1, q_2, q_4, q_5, q_6)$ to oracle π, getting back $\pi(q_1), \pi(q_2), \pi(q_3), \pi(q_4), \pi(q_5), \pi(q_6)$.
- Quadratic correction test: Check that $\pi(q_1) \cdot \pi(q_2) = \pi(q_3) - \pi(q_4)$. If not, `reject`.
- Circuit test: Check that $(\pi(q_5) - \pi(q_1)) + (\pi(q_6) - \pi(q_4)) = -\gamma_0$. If so, `accept`.

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

(see [30], definitions 5.2.2, 5.2.8 and Exercise 17) implies that for all PPT $\mathcal{A}$ ($\mathcal{A}$ can be non-uniform),

$$\Pr_{\mathsf{Gen},\mathsf{Enc},r,Z_1,Z_2}\{\mathcal{A}(pk, \mathsf{Enc}(pk, r), r + Z_1 q, r + Z_2 q) = Z_1\} < 1/|\mathbb{F}| + \epsilon_S. \quad (1)$$

This holds for all $q \in \mathbb{F}^n$.[5]

Now, assume to the contrary that Commit+Multidecommit does not meet the definition of ϵ_L-linearity. Then there exists an environment $\mathcal{E}$ producing $q_i, q_j, i, j, k, Q, \hat{Q}, S^*$ (where Q has q_i, q_j in the ith and jth positions and $\hat{Q}$ has $q_i + q_j$ in the kth position) such that $\Pr_{\text{all 3 phases}}\{S^* \text{ wins linearity under } \mathcal{E}\} > 1/|\mathbb{F}| + \epsilon_S$. Let $q' \triangleq \hat{q}_k = q_i + q_j$.

We now describe an algorithm $\mathcal{A}$ that, when given input $I = (pk, \mathsf{Enc}(pk, r), r + Z_1 q', r + Z_2 q')$, can recover Z_1 with probability more than $1/|\mathbb{F}| + \epsilon_S$. $\mathcal{A}$ has $Q, \hat{Q}, q_i, q_j, i, j, k$ hard-wired (because it is working under environment $\mathcal{E}$) and works as follows:

(a) $\mathcal{A}$ gives $(pk, \mathsf{Enc}(pk, r))$ to S^* and ignores the reply.

(b) $\mathcal{A}$ randomly generates $\alpha_1, \ldots, \alpha_\mu$ and sends to S^* the input $(Q, r + \alpha_1 q_1 + \cdots + (\alpha_i + Z_1) q_i + \cdots + (\alpha_j + Z_1) q_j + \cdots + \alpha_\mu q_\mu)$. $\mathcal{A}$ is able to construct this input because $\mathcal{A}$ was given $r + Z_1 q' = r + Z_1 q_i + Z_1 q_j$. In response, S^* returns $(b, a_1, \ldots, a_i, \ldots, a_j, \ldots, a_\mu)$.

(c) $\mathcal{A}$ randomly generates $\hat{\alpha}_1, \ldots, \hat{\alpha}_\mu$. $\mathcal{A}$ sends to S^* the input $(\hat{Q}, r + \hat{\alpha}_1 \hat{q}_1 + \cdots + Z_2 \hat{q}_k + \cdots + \hat{\alpha}_\mu \hat{q}_\mu)$. $\mathcal{A}$ is able to construct this input because $\mathcal{A}$ was given $r + Z_2 q' = r + Z_2 \hat{q}_k$. $\mathcal{A}$ gets back $(\hat{b}, \hat{a}_1, \ldots, \hat{a}_k, \ldots, \hat{a}_\mu)$.

At this point, $\mathcal{A}$ assumes that the responses from S^* pass the decommitment phase; that is, $\mathcal{A}$ acts as if $b = s + \alpha_1 a_1 + \cdots + (\alpha_i + Z_1) a_i + \cdots + (\alpha_j + Z_1) a_j + \cdots + \alpha_\mu a_\mu$ and $\hat{b} = s + \hat{\alpha}_1 \hat{a}_1 + \cdots + Z_2 \hat{a}_k + \cdots + \hat{\alpha}_\mu \hat{a}_\mu$. $\mathcal{A}$ can write

[5] We are being loose here. Under the actual definition of semantic security, (a) ϵ_S should be replaced with a negligible function of n, and (b) the claim holds only for n sufficiently large.

$$K_1 = Z_2\hat{a}_k - Z_1(a_i + a_j), \quad (2)$$

where $\mathcal{A}$ can derive $K_1 = \hat{b} - b - \sum_{\iota \neq k} \hat{\alpha}_\iota \hat{a}_\iota + \sum_\iota \alpha_\iota a_\iota$. Now, let $t = r + Z_1 q'$ and let $\hat{t} = r + Z_2 q'$ (both of these were supplied as input to $\mathcal{A}$). These two equations concern vectors. However, by choosing an index ι in the vector q' where q' is not zero (if the vector is zero everywhere, then r is revealed), $\mathcal{A}$ can derive

$$K_2 = Z_2 - Z_1, \quad (3)$$

where $K_2 = (\hat{t}^{(\iota)} - t^{(\iota)})/q'^{(\iota)}$.

Now, observe that if $\hat{a}_k \neq a_i + a_j$ (as happens when S^* wins), then $\mathcal{A}$ can recover Z_1 by solving equations (2) and (3). Thus,

$$\begin{aligned} &\Pr_{\mathsf{Gen},\mathsf{Enc},r,Z_1,Z_2,\vec{\alpha},\vec{\hat{\alpha}}}\{\mathcal{A}(I) = Z_1\} \\ &\geq \Pr_{\mathsf{Gen},\mathsf{Enc},r,Z_1,Z_2,\vec{\alpha},\vec{\hat{\alpha}}}\{S^* \text{ wins linearity under } \mathcal{E}\} \\ &= \Pr_{\text{all 3 phases}}\{S^* \text{ wins linearity under } \mathcal{E}\} \\ &> 1/|\mathbb{F}| + \epsilon_S. \quad (4) \end{aligned}$$

The equality holds because the distribution of $(\alpha_1, \ldots, \alpha_i + Z_1, \ldots, \alpha_j + Z_1, \ldots, \alpha_\mu)$ and $(\hat{\alpha}_1, \ldots, Z_2, \ldots, \hat{\alpha}_\mu)$ is equivalent to the distribution from which R selects in the decommit phases of the three-phase experiment, under Commit+Multidecommit. Meanwhile, inequality (4) contradicts inequality (1). □

The lemmas ahead show that, under Commit+Multidecommit, S is bound to a *nearly linear* function, $\tilde{f}(\cdot)$; specifically, $\tilde{f}(\cdot)$ is δ^*-close to linear for small δ^*. By contrast, previous work [35, 45] showed only that S was bound to *some* function $\tilde{f}(\cdot)$.

We now give some notation and restate two claims from [45]. Let ζ be the event that R's output is a vector $(a_1, \ldots, a_\mu)$; equivalently, ζ is the event that R's output is non-$\perp$. Below, we sometimes write $\Pr_{\text{comm}}\{\cdot\}$ or $\Pr_{\text{decomm}}\{\cdot\}$ to mean the probability over the random choices of the commit or decommit phases.

Lemma A.2 (Existence of an extractor function [45]). Let (S, R) be a CFMD protocol with binding error ϵ_B. Let $\epsilon_C = \mu \cdot 2 \cdot (2\sqrt[3]{9/2} + 1) \cdot \sqrt[3]{\epsilon_B}$. Let $v = (v_{S^*}, v_R)$ represent the views of S^* and R after the commit phase (v captures the randomness of the commit phase). For every efficient S^* and for every v, there exists a function $\tilde{f}_v : \mathbb{F}^n \to \mathbb{F}$ such that the following holds.[6] For any environment $\mathcal{E}$, the output of R at the end of the decommit phase is, except with probability ϵ_C, either $\perp$ or satisfies $a_i = \tilde{f}_v(q_i)$ for all $i \in [\mu]$, where $(q_1, \ldots, q_\mu)$ are the decommitment queries generated by $\mathcal{E}$, and the probability is over the random inputs of S^* and R in both phases.

[6]Note that after the commit phase, $\tilde{f}_v(\cdot)$ is deterministic. ($\tilde{f}_v(\cdot)$ is defined [35, 45] to map q to the value that R is most likely to successfully output in the decommit phase.)

Lemma A.3. Let $\epsilon_3 = (2\sqrt[3]{9/2} + 1) \cdot \sqrt[3]{\epsilon_B}$. Label the ith query in Q as q_i and the ith response as a_i. For all Q, i, we have $\Pr_{\text{comm,decomm}}\{\zeta \cap \{a_i \neq \tilde{f}_v(q_i)\}\} < 2\epsilon_3$.

Proof. Follows from a claim in [45] (Claim B.4). □

Lemma A.4. For all $q_1, q_2 \in \mathbb{F}^n$, $\Pr_{\text{comm}}\{\tilde{f}_v(q_1) + \tilde{f}_v(q_2) \neq \tilde{f}_v(q_1 + q_2)\} < \epsilon_F \triangleq \epsilon_L + 6\epsilon_3$.

Proof. Assume otherwise. Then for some q_1 and q_2, we have $\Pr_{\text{comm}}\{\tilde{f}_v(q_1) + \tilde{f}_v(q_2) \neq \tilde{f}_v(q_1 + q_2)\} \geq \epsilon_F$, which implies $\Pr_{\text{all 3 phases}}\{\tilde{f}_v(q_1) + \tilde{f}_v(q_2) \neq \tilde{f}_v(q_1 + q_2)\} \geq \epsilon_F$, since we can "add coin flips that don't matter", namely those of the two decommit phases.

Now, consider the game in the definition of ϵ_L-linearity, and set $Q = (q_1, q_2, \ldots)$ and $\hat{Q} = (q_1 + q_2, \ldots)$. Let η be the event that S^* wins in this game. Let ν be the event that the outputs $a_1, a_2, \hat{a}_1$ are given by the function $\tilde{f}_v(\cdot)$. Then $\Pr_{\text{all 3 phases}}\{\neg\nu\} < 6\epsilon_3$, by Lemma A.3, by the union bound, and by (again) "adding coin flips that don't matter" to get from a probability over two phases to one over three phases. Now, note that $\Pr_{\text{all 3 phases}}\{\eta | \nu\} \geq \epsilon_F$, by the contrary hypothesis. This implies that $\Pr_{\text{all 3 phases}}\{\eta\} \geq \epsilon_F - 6\epsilon_3 = \epsilon_L$, which contradicts the definition of ϵ_L-linearity. □

Lemma A.4 almost talks about a linearity test [16]! But linearity testing theory [8] relates (a) the probability *over randomly chosen queries* that the test fails and (b) the closeness-to-linearity of the tested function. Thus, to apply the theory, we line up Lemma A.4 and (a).

Lemma A.5. With probability greater than $1 - \sqrt{\epsilon_F}$ over the commit phase, the fraction of (q_1, q_2) pairs that cause $\tilde{f}_v(\cdot)$ to fail the linearity test is $\leq \sqrt{\epsilon_F}$.

Proof. Let I_{v,q_1,q_2} be an indicator random variable that equals 1 if, in view v (that is, given the randomness of the commit phase), $\tilde{f}_v(q_1 + q_2) \neq \tilde{f}_v(q_1) + \tilde{f}_v(q_2)$. The lemma is equivalent to the statement

$$\Pr_{\text{comm}}\{\Pr_{q_1,q_2}\{I_{v,q_1,q_2} = 1\} > \sqrt{\epsilon_F}\} < \sqrt{\epsilon_F}.$$

Now, define a random variable $Y_v = \frac{1}{Q^2}\sum_{q_1,q_2} I_{v,q_1,q_2}$, where $Q = |\mathbb{F}|^n$ is the number of possibilities for each of q_1 and q_2. By linearity of expectation, $E_{\text{comm}}[Y_v] = \frac{1}{Q^2} \cdot (E[I_{v,1}] + \cdots + E[I_{v,Q^2}])$, where $E[I_{v,i}]$ is the probability, over the commit phase, that a particular (q_j, q_k) pair causes $\tilde{f}_v(\cdot)$ to fail the linearity test. Lemma A.4 implies that $E[I_{v,i}] < \epsilon_F$ for all i; hence, $E_{\text{comm}}[Y_v] < \epsilon_F$. We now apply a Markov bound to Y_v:

$$\Pr_{\text{comm}}\{Y_v > \sqrt{\epsilon_F}\} < \frac{E_{\text{comm}}[Y_v]}{\sqrt{\epsilon_F}} < \frac{\epsilon_F}{\sqrt{\epsilon_F}} = \sqrt{\epsilon_F}.$$

But Y_v is equivalent to $\Pr_{q_1,q_2}\{I_{v,q_1,q_2} = 1\}$; making this substitution immediately above yields the lemma. □

Lemma A.6. Let δ^* be the lesser root of $6\delta^2 - 3\delta + \sqrt{\epsilon_F} = 0$. If $\sqrt{\epsilon_F} < \frac{2}{9}$, then with probability greater than $1 - \sqrt{\epsilon_F}$ over the commit phase, $\tilde{f}_v(\cdot)$ is δ^*-close to linear.

Proof. 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 linearity test. 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 $\mathsf{Rej}(f) \geq \frac{2}{9}$.

The above implies the following claim: for all $\delta' \in \{\delta' \mid 3\delta' - 6\delta'^2 < \frac{2}{9} \text{ and } 0 \leq \delta' \leq \frac{1}{4}\}$, if $\mathsf{Rej}(f) \leq 3\delta' - 6\delta'^2$, then $\mathsf{Dist}(f) \leq \delta'$. (To see this, fix δ'. Assume to the contrary that $\delta = \mathsf{Dist}(f) > \delta'$. There are two cases, and both contradict the given. If $\delta < \frac{1}{4}$, then $\mathsf{Rej}(f) \geq 3\delta - 6\delta^2 > 3\delta' - 6\delta'^2$. If $\delta \geq \frac{1}{4}$, then $\mathsf{Rej}(f) \geq \frac{2}{9} > 3\delta' - 6\delta'^2$.)

From lemma A.5, the probability is greater than $1 - \sqrt{\epsilon_F}$ over the commit phase that $\mathsf{Rej}(\tilde{f}_v) \leq \sqrt{\epsilon_F}$. We call such commit phases *usual*. Under a *usual* commit phase, we can apply the claim just above. To do so, we assume that $\sqrt{\epsilon_F} < \frac{2}{9}$, and we set δ^* so that $\sqrt{\epsilon_F} = 3\delta^* - 6\delta^{*2}$ and $\delta^* \leq \frac{1}{4}$ (such a δ^* is guaranteed to exist because the parabola is symmetric about $\delta = \frac{1}{4}$). The claim implies that $\mathsf{Dist}(\tilde{f}_v) \leq \delta^*$, or that $\tilde{f}_v$ is δ^*-close to linear. □

Lemma A.7. If the PCP oracle π is known to be δ^*-close to linear, then the linear PCP (Section A.1) with linearity testing removed has soundness error $\kappa > \max\{4\delta^* + \frac{2}{|\mathbb{F}|}, 4\delta^* + \frac{1}{|\mathbb{F}|}\}$.

Proof. This follows from the proof flow that establishes the soundness of linear PCPs, as in [5]. (A self-contained example is in Appendix D of [45].) Those proofs first establish that if the linearity test passes with probability higher than the soundness error, then π is δ-close to linear, for some δ. However, if we are *given* that π is δ^*-close to linear, then we can start those proofs midway and obtain the soundness of π as κ. □

Proof of Theorem A.1. Lemma A.2 implies that there exists an extractor function that determines a (possibly incorrect) oracle $\tilde{\pi}$ such that, if V' does not reject during decommit, then with all but probability ϵ_C, V' receives back $\tilde{\pi}(q_1), \ldots, \tilde{\pi}(q_\mu)$. We can thus "pay" probability ϵ_C in the union bound (below) to assume that V' hears back from $\tilde{\pi}$ itself. This allows us to apply Lemma A.6, at which point we can "pay" $\sqrt{\epsilon_F}$ more probability (again in the union bound below) to get that $\tilde{\pi}$ is δ^*-close to linear. (Applying the lemma requires that $\sqrt{\epsilon_F} < \frac{2}{9}$, and we will verify below that this bound holds.) Now, we can apply Lemma A.7 to ρ runs of the PCP protocol, giving a PCP soundness error of κ^ρ. Thus, the probability that V' wrongly accepts a proof is bounded from above by:

$$\epsilon_G = \epsilon_C + \sqrt{\epsilon_F} + \kappa^\rho.$$

By inspection (of the lemmas), the dominant contributor to ϵ_G, namely $\sqrt{\epsilon_F}$, is proportional to $\sqrt[6]{1/|\mathbb{F}|}$. □

We compute a bound on ϵ_G as follows.

- ϵ_C is given in Lemma A.2. We take $\mu = 6$ (per Figure 14). We also take $\epsilon_B = 1/|\mathbb{F}|$ (following [45]; this amounts to ignoring the error from the semantic security of the homomorphic encryption scheme) and $|\mathbb{F}| = 2^{128}$, giving $\epsilon_C < 7.4 \cdot 10^{-12}$.
- $\epsilon_F = \epsilon_L + 6\epsilon_3$ (from Lemma A.4). ϵ_3 is given in Lemma A.3. We set $\epsilon_L = 1/|\mathbb{F}|$ (which again amounts to ignoring ϵ_S). Again taking $|\mathbb{F}| = 2^{128}$, we get $\sqrt{\epsilon_F} < 1.9 \cdot 10^{-6}$. Thus, $\sqrt{\epsilon_F} < 2/9$, as required.
- $\kappa = 4\delta^* + \frac{2}{|\mathbb{F}|}$, where δ^* is the lesser root of $6\delta^2 - 3\delta + \sqrt{\epsilon_F}$. This gives $\delta^* = 6.4 \cdot 10^{-7}$ and $\kappa = 2.6 \cdot 10^{-6}$.

Since κ and $\sqrt{\epsilon_F}$ are roughly the same, there is not much point to taking $\rho > 1$. Thus, we take $\rho = 1$, giving $\epsilon_G < 4.5 \cdot 10^{-6}$ when $|\mathbb{F}| = 2^{128}$. When $|\mathbb{F}| = 2^{192}$, we get $\epsilon_G < 2.8 \cdot 10^{-9}$.

A.3 Optimizing out queries

GINGER's PCP protocol includes two further refinements. First, the protocol reuses q_4 and q_1 from test to test. This reuse is sound because the PCP soundness lemma [5] is of the form, "if all tests pass with probability greater than X, then the proof oracle π has a certain desired property"; meanwhile, as Bellare et al. [9] observe, the tests need not be independent! One can observe the savings by comparing Figure 13 (minus the linearity queries) to Figure 14. The protocol goes from 8 queries (the original 14 minus 6 linearity queries) to 6 queries, though the real savings for the prover is in reducing the 4 high-order queries (that is, queries to the $\mathbb{F}^{s^2}$ component of π) to 3. Moreover, the verifier saves because it goes from generating pseudorandomness for 3 high-order queries (including γ_2) to 2. Second, V avoids transmitting a query (q_3) that P can generate for itself. This optimization offsets the consistency query, which is computed over $\mathbb{Z}$ not $\mathbb{Z}/p$ (owing to the details of our use of ElGamal [45, Appendix E]) and thus has roughly twice as many bits as a PCP query.

B Signed integers, floating-point rationals

In this appendix and the two ahead, we describe how GINGER applies to general-purpose computations. This appendix describes GINGER's representation of signed integers and its representation of primitive floating-point quantities (this treatment expands on Section 4.1). The next appendix details the case study (from Section 4.2) of an inequality test and a conditional branch. Appendix D describes other program constructs.

Our goal in these appendices is to show how to map computations to equivalent constraints over finite fields (according to the definition of equivalent given in Section 2.1). To do so, we follow the framework from Section 4. Recall that the three steps in that framework are: (C1) Bound the computation Ψ, which starts out over some domain D (such as $\mathbb{Z}$ or $\mathbb{Q}$), to ensure that Ψ stays within some set $U \subset D$; (C2) Establish a map between U and a finite field $\mathbb{F}$ such that computing Ψ in $\mathbb{F}$ is equivalent to computing Ψ in U. (C3) Transform the finite field version of the computation into constraints.

B.1 Signed integers

To illustrate step C1, consider $m \times m$ matrix multiplication over signed integers, with inputs of N bits (where the top bit is the sign): the computation does not "leave" $U = [-m \cdot 2^{2N-1}, m \cdot 2^{2N-1})$, where $U \subset \mathbb{Z}$.

For step C2, we take $\mathbb{F} = \mathbb{Z}/p$ and define θ between U and $\mathbb{Z}/p$ as follows:

$$\begin{aligned}\theta\colon U &\to \mathbb{Z}/p\\ u &\mapsto u \bmod p.\end{aligned}$$

Note that:

(1) If U is an interval $[a,b]$ and $|\mathbb{Z}/p| > |U|$, then θ is 1-to-1.

(2) If $x_1, x_2 \in U$ and $x_1 + x_2 \in U$, then $\theta(x_1) + \theta(x_2) = \theta(x_1 + x_2)$.

(3) If $x_1, x_2 \in U$ and $x_1 x_2 \in U$, then $\theta(x_1)\theta(x_2) = \theta(x_1 x_2)$.

To argue property (1), take $x \bmod p = y \bmod p$, which means $x = y + p \cdot k$, for $k \in \mathbb{Z}$. If $x \neq y$ (so θ is not 1-to-1), then $|y - x| \geq p$, which implies that there must be at least $p+1$ elements in U, since U is an interval that includes x and y. But $|U| < |\mathbb{Z}/p| = p$, a contradiction. Properties (2) and (3) follow because θ is a restriction of the usual reduction map, so it preserves addition and multiplication.

Thus, computation in $\mathbb{Z}/p$ is isomorphic to computation in U: the constraint representation uses only field operations to represent computations (see Section 2.1), and for the purposes of field operations, $\mathbb{Z}/p$ acts like U.

B.2 Floating-point rational numbers

Step C1

To illustrate this step, we again consider $m \times m$ matrix multiplication and this time require the input entries to be in the set $T = \{a/b : |a| \leq 2^{N_a}, b \in \{1, 2, 2^2, 2^3, \ldots, 2^{N_b}\}\}$. To bound the computation to a set U, we use the claim below.

Claim B.1. For the computation of matrix multiplication, with input entries restricted to T, the computation of matrix multiplication is restricted to $U = \{a/b\colon |a| < 2^{N'_a}, b \in \{1, 2, 2^2, 2^3, \ldots, 2^{2N_b}\}\}$, for $N'_a = 2N_a + 2N_b + \log_2 m$.

Proof. Consider an entry in the output; it is of the form $\sum_{k=1}^{m} A_{ik}B_{kj}$, where each $A_{ik}B_{kj}$ is contained in $S = \{a/b\colon |a| < 2^{2N_a}, b \in \{1, 2, 2^2, 2^3, \ldots, 2^{2N_b}\}\}$. Thus, we can write each output entry as $\sum_{k=1}^{m} a_k/b_k$, the sum of m numbers from the set S. Writing each b_k as 2^{e_k}, and letting $e^* = \max_k e_k$, we can write the sum as

$$\frac{\sum_k a_k 2^{e^* - e_k}}{2^{e^*}}.$$

The denominator of this sum is contained in $\{1, 2, 2^2, 2^3, \ldots, 2^{2N_b}\}$. The absolute value of each summand in the numerator, $a_k 2^{e^* - e_k}$, is no larger than $2^{2N_a + 2N_b}$, and there are m summands, so the absolute value of the numerator is no larger than $m \cdot 2^{2N_a + 2N_b} = 2^{2N_a + 2N_b + \log_2 m}$. A fortiori, the intermediate sums are contained in U (they have fewer than m terms). □

Step C2

We must identify a field $\mathbb{F}$ where the computation can be mapped; that is, we need a field that behaves something like $\mathbb{Q}$. For this purpose, we take $\mathbb{F} = \mathrm{Frac}(\mathbb{Z}/p)$, the quotient field of $\mathbb{Z}/p$, which we denote $\mathbb{Q}/p$.

This paragraph reviews the definition and properties of $\mathbb{Q}/p$ because we will need these details later. As a quotient field, $\mathbb{Q}/p$ is the set of *equivalence classes* on the set $\mathbb{Z}/p \times (\mathbb{Z}/p \setminus \{0\})$, under the equivalence relation $\sim$, where $(a,b) \sim (c,d)$ if $ad = bc \bmod p$; the field operations are $(a,b) + (c,d) = (ad + bc \bmod p, bd \bmod p)$ and $(a,b) \cdot (c,d) = (ac \bmod p, bd \bmod p)$, where a pair (x,y) represents its equivalence class. Note that although elements of $\mathbb{Q}/p$ are *represented* as having two components, each of which seems able to take p or $p-1$ values, the cardinality of $\mathbb{Q}/p$ is only p. In fact, $\mathbb{Q}/p$ is isomorphic to $\mathbb{Z}/p$, via the map $f((a,b)) = a \cdot b^{-1}$.

We must now define a map from U to $\mathbb{Q}/p$; in doing so, we will take U to be an arbitrary subset of $\mathbb{Q}$:

$$\begin{aligned}\theta\colon U &\to \mathbb{Q}/p\\ \frac{a}{b} &\mapsto (a \bmod p, b \bmod p).\end{aligned}$$

Note that θ is well-defined. (This fact is standard, but for completeness, we briefly argue it. Take $q_1 = \frac{a_1}{b_1}$, $q_2 = \frac{a_1 x}{b_1 x}$. Then $\theta(q_1) = (a_1 \bmod p, b_1 \bmod p)$ and $\theta(q_2) = (a_1 x \bmod p, b_1 x \bmod p)$. But we have $(a_1 \bmod p)(b_1 x \bmod p) \equiv (b_1 \bmod p)(a_1 x \bmod p) \pmod p$, so $\theta(q_1) \sim \theta(q_2)$.)

As mentioned above, $\mathbb{Q}/p$ does not have as much "room" as one might guess. To make θ 1-to-1, we must choose p carefully. The lemma below says how to do so, but we need a definition first. Define the *s-value of an element* $q \in \mathbb{Q}$ as follows. Write q as a/b, where $a, b \in \mathbb{Z}$, $b > 0$, and a and b are co-prime. Then the s-value of q, written $s(q)$, is $s(q) = |a| + b$. We write the *s-value of a set* $U \subset \mathbb{Q}$ as $s(U)$, and we define it as the largest s-value of U's elements: $s(U) = \max_{u \in U} s(u)$.

Lemma B.2. For any $U \subset \mathbb{Q}$, if $p \geq s(U)^2$, then θ is 1-to-1.

Proof. Take $q_1, q_2 \in U$, where $\theta(q_1) \sim \theta(q_2)$. We need to show that $q_1 = q_2$. Write $q_1 = a_1/b_1$ and $q_2 = a_2/b_2$ in reduced form (that is, a_i and b_i are co-prime). Note that by definition of s-value and choice of p, p is greater than each of a_1, b_1, a_2, b_2, so we can write $\theta(q_1)$ as (a_1, b_1) and $\theta(q_2)$ as (a_2, b_2). Since $\theta(q_1) \sim \theta(q_2)$, we have $a_1 b_2 \equiv a_2 b_1 \pmod p$. But then if $a_1 b_2 \neq a_2 b_1$ (as would be implied by $q_1 \neq q_2$) we would have:

$$
\begin{aligned}
p &\leq |a_1 b_2 - a_2 b_1| \\
&\leq |a_1 b_2| + |a_2 b_1| \\
&< |a_1 b_2| + |a_2 b_1| + |a_1 a_2| + |b_1 b_2| \\
&= (|a_1| + |b_1|)(|a_2| + |b_2|) \\
&\leq s(U) \cdot s(U),
\end{aligned}
$$

making $p < s(U)^2$, a contradiction. □

Representing computations over $\mathbb{Q}/p$ faithfully. Note that:

(1) If $q_1, q_2 \in U$ and $q_1 + q_2 \in U$, then $\theta(q_1) + \theta(q_2) \sim \theta(q_1 + q_2)$.

(2) If $q_1, q_2 \in U$ and $q_1 q_2 \in U$, then $\theta(q_1)\theta(q_2) \sim \theta(q_1 q_2)$.

These properties can be verified by inspection. Since θ preserves addition and multiplication, and since θ is 1-to-1 by choice of p, the computation in $\mathbb{Q}/p$ is isomorphic to the computation in $\mathbb{Q}$.

Examples. If U is defined as in Claim B.1, then the s-value is upper-bounded by

$$m \cdot 2^{2(N_a+N_b)} + 2^{2N_b} < m \cdot 2^{2(N_a+N_b)} + 2^{2(N_a+N_b)}.$$

Applying Lemma B.2, if we take $p > (m+1)^2 \cdot 2^{4(N_a+N_b)}$, then computation over $\mathbb{Q}/p$ is isomorphic to computation over U, as claimed in Section 4.1. As another example, consider numbers of the form $a \cdot 2^{-q}$, where $|a| < 2^{N_a}$ and $|q| \leq N_q$. Then the prime requires at least $2\log_2(2^{N_a}+2^{N_q}) \leq 2 \cdot (\max\{N_a, N_q\}+1)$ bits, as claimed in Section 4.1.

Canonical forms and θ^{-1}

Later, it will be convenient to have defined θ^{-1} explicitly—and to have expressed this definition in terms of a particular representation of elements of $\mathbb{Q}/p$. This may seem strange because the whole concept of equivalence class is that, within a class, all representations are equivalent. However, our constraints for certain computations, such as less-than, will require assumptions about the representation of an element (see Appendix C.2). Thus, we define a canonical representation below; we focus on the case when U is of the form $\{a/b\colon |a| < 2^{N_a}, b \in \{1, 2, 2^2, 2^3, \ldots, 2^{N_b}\}\}$.

Definition B.1 (Canonical form in $\mathbb{Q}/p$). An element $(a, b) \in \theta(U)$ is a *canonical form* or *canonical representation* of its equivalence class if $a \in [0, 2^{N_a}] \cup [p - 2^{N_a}, p)$ and $b \in \{1, 2, 4, \ldots, 2^{N_b}\}$. Every element in $\theta(U)$ has such a representation, by definition of U and θ.

We now define θ^{-1}; let (e_a, e_b) denote a canonical form of e:

$$
\begin{aligned}
&\theta^{-1}\colon \theta(U) \to U \\
&\qquad e \mapsto \begin{cases} e_a/e_b, & 0 \leq e_a \leq 2^{N_a} \\ (e_a - p)/e_b, & p - 2^{N_a} \leq e_a < p \end{cases}
\end{aligned}
$$

Note that when e_a is in the "upper" part of the range, θ^{-1} maps e to a negative number in $\mathbb{Q}$. Note also that the canonical form for an equivalence class may not be unique. However, the following two claims establish that this non-uniqueness is not an issue in our context.

Claim B.3. θ^{-1} is well-defined.

Proof. For $e \in \theta(U)$, let $e = (a, b) \sim (c, d)$, where (a, b) and (c, d) are both canonical forms. We wish to show that $\theta^{-1}((a, b)) = \theta^{-1}((c, d))$.

We have $\theta^{-1}((a, b)) \in U$ and $\theta^{-1}((c, d)) \in U$, by definition of θ^{-1} and U. Also, we have $\theta(\theta^{-1}((a, b))) \sim (a, b)$, as follows. If $a \in [0, 2^{N_a}]$, then $\theta^{-1}((a, b)) = a/b$ and $\theta(a/b) = (a, b)$. If $a \in [p - 2^{N_a}, p)$, then $\theta^{-1}((a, b)) = (a - p)/b$ and $\theta((a - p)/b) = (a - p \bmod p, b) \sim (a, b)$. Likewise, $\theta(\theta^{-1}((c, d))) \sim (c, d)$. Now, let $u_1 = \theta^{-1}((a, b))$ and $u_2 = \theta^{-1}((c, d))$. Assume toward a contradiction that $u_1 \neq u_2$; then $\theta(u_1) \not\sim \theta(u_2)$, by Lemma B.2. Thus $(a, b) \sim \theta(\theta^{-1}((a, b)) \not\sim \theta(\theta^{-1}((c, d))) \sim (c, d)$, a contradiction. □

Claim B.4. An element in $\theta(U)$ cannot have two canonical representations (a, b) and (c, d) with $a \in [0, 2^{N_a}]$ and $c \in [p - 2^{N_a}, p)$.

Proof. Take $(a,b) \sim (c,d)$ where $a \in [0, 2^{N_a}]$ and $c \in [p - 2^{N_a}, p)$ (note that $b, d > 0$). Because θ^{-1} is a function (Claim B.3), $\theta^{-1}((a,b)) = \theta^{-1}((c,d))$. However, $\theta^{-1}((a,b)) = a/b \geq 0$ and $\theta^{-1}((c,d)) = (c-p)/d < 0$, which is a contradiction. □

Discussion

Most of our work above presumed a restriction on U: that the denominators of its elements are powers of 2. We defined U this way because, without this restriction, we would need a much larger prime p, per Lemma B.2. However, this restriction is not fundamental, and our framework does not require it. On the other hand, the restriction yields primitive floating-point numbers with acceptable precision at acceptable cost (see Section 4.1).

Implementation detail

When working with computations over $\mathbb{Q}$, we express them over the finite field $\mathbb{Q}/p$. However, our implementation (source code, etc.) assumes that the finite field is represented as $\mathbb{Z}/p$. Fortunately, as noted above, $\mathbb{Q}/p$ is isomorphic to $\mathbb{Z}/p$ via the following map:

$$\begin{aligned} f\colon \mathbb{Q}/p &\to \mathbb{Z}/p \\ (a,b) &\mapsto ab^{-1}. \end{aligned}$$

We take advantage of this isomorphism to reuse our implementation over $\mathbb{Z}/p$. Specifically, when computing over $\mathbb{Q}/p$, V and P follow the protocol below.

Definition B.2 (GINGER-Q protocol). Let Ψ be a computation over $\mathbb{Q}/p$, and let Ψ' be the same computation, expressed over $\mathbb{Z}/p$. The GINGER-Q protocol for verifying Ψ is defined as follows:

1. $V \to P$: a vector x, over the domain $\mathbb{Q}/p$.
2. $P \to V$: $y = \Psi(x)$.
3. $P \to V$: x' and y'. P obtains x', y' (which are vectors in $\mathbb{Z}/p$) by applying f elementwise to x and y.
4. V checks that for all $(a,b) \in \{x \cup y\}$ and the corresponding element $c \in \{x' \cup y'\}$, $cb \equiv a \bmod p$. This confirms that P has applied f correctly. If the check fails, V rejects.
5. V engages P using the existing GINGER implementation, to verify that $y' = \Psi'(x')$.

The implementation convenience carries a cost: P must compute b^{-1} for each element a/b in the input and output of Ψ (as part of computing f), and V must check that P applied f correctly. On the other hand, some cost seems unavoidable. In fact, it might cost even more if the implementation worked in $\mathbb{Q}/p$ directly: arithmetic is roughly twice as expensive using the $\mathbb{Q}/p$ representation versus the $\mathbb{Z}/p$ representation.

C Case study: branch and inequalities

Below, we will give constraints for a computation that branches based on a less-than test. (This will instantiate step C3 for the case study in Section 4.2.) Most of the work is in representing the less-than test; we do so with *range constraints* that take apart a number and interrogate its bits.

C.1 Order comparisons over the integers

Preliminaries

We will assume that the programmer or compiler has applied steps C1 and C2 to bound the inputs, x_1 and x_2, and to choose $\mathbb{F}$; thus, their difference is bounded too. Specifically, we assume $x_1 - x_2 \in U \subset [-2^{N-1}, 2^{N-1})$, $\mathbb{F} = \mathbb{Z}/p$ for some $p > 2^N$, and $\theta(x) = x \bmod p$. (See Appendix B.1.)

With these restrictions, $x_1 < x_2$ if and only if $x_1 - x_2 \in [-2^{N-1}, 0)$, which holds if and only if $\theta(x_1) - \theta(x_2) \in [p - 2^{N-1}, p)$; the second equivalence follows because θ is 1-to-1 and preserves addition and multiplication, as shown in the previous appendix.

Step C3

To instantiate step C3, we write a set of constraints, $\mathcal{C}_<$:[7]

$$\mathcal{C}_< = \left\{ \begin{array}{l} B_0(1 - B_0) = 0, \\ B_1(2 - B_1) = 0, \\ \quad \vdots \\ B_{N-2}(2^{N-2} - B_{N-2}) = 0, \\ \\ \theta(X_1) - \theta(X_2) - (p - 2^{N-1}) - \sum_{i=0}^{N-2} B_i = 0 \end{array} \right\}$$

Lemma C.1. $\mathcal{C}_<$ is satisfiable if and only if $\theta(x_1) - \theta(x_2) \in [p - 2^{N-1}, p)$.

Proof. Assume $\theta(x_1) - \theta(x_2) \in [p - 2^{N-1}, p)$. Let $X_3 = \theta(x_1) - \theta(x_2) - (p - 2^{N-1})$. Observe that $X_3 \in [0, 2^{N-1})$, so X_3's binary representation has bits $z_0, z_1, \ldots, z_{N-2}$. Now, set $B_i = z_i \cdot 2^i$ for $i \in \{0, 1, \ldots, N-2\}$. This will satisfy all but the last constraint because B_i is set equal to either 0 or 2^i. And the last constraint is satisfied from the definition of X_3 and because we set the $\{B_i\}$ so that $\sum_{i=0}^{N-2} B_i = X_3$. For the other direction, if the constraints are satisfiable, then $\theta(x_1) - \theta(x_2) = p - 2^{N-1} + \sum_{i=0}^{N-2} B_i$, where the $\{B_i\}$ are powers of 2, or 0. This means that $\theta(x_1) - \theta(x_2) \in [p - 2^{N-1}, p)$. □

Corollary C.2. For x_1, x_2 as restricted above, $\mathcal{C}_<$ is satisfiable if and only if $x_1 < x_2$.

In other words, assuming the input restrictions, $\mathcal{C}_<$ is equivalent to the logical test of $<$ over $\mathbb{Z}$.

[7] Step C3 in the body text (§4) calls for "equivalent" constraints, but the definition of "equivalent" in Section 2.1 presumes a designated output variable, which the constraints for logical tests do not have. However, one can extend the definition of "equivalent" to logical tests.

C.2 Order comparisons over the rationals

When dealing with the rationals, extra preliminary work is required to apply step C3; the core reason is that each element in $\mathbb{Q}/p$ has multiple representations (recall that $\mathbb{Q}/p$ is isomorphic to $\mathbb{Z}/p$).

Preliminaries

We assume that the programmer or compiler has applied steps C1 and C2 to restrict the inputs, x_1 and x_2, so that $x_1 - x_2 \in U$, for $U = \{a/b\colon |a| < 2^{N_a}, b \in \{1, 2, 2^2, \ldots, 2^{N_b}\}\}$. Similarly, we assume that $\mathbb{F}$ is $\mathbb{Q}/p$, p is chosen according to Lemma B.2, and $\theta(a/b) = (a \bmod p, b \bmod p)$. (See Appendix B.2.)

At this point, we need the $x_1 < x_2$ test to be in a form suitable for representation in $\mathbb{Q}/p$. Observe that $x_1 < x_2$ if and only if $x_1 - x_2 \in S = \{a/b\colon -2^{N_a} \le a < 0, b \in \{1, 2, 2^2, \ldots, 2^{N_b}\}\}$, which holds if and only if $\theta(x_1) - \theta(x_2) \in \theta(S)$; as with the integers case, the second biconditional follows because θ is 1-to-1, and preserves addition and multiplication. However, we wish to represent this condition in a way that explicitly refers to the representation of $\theta(x_1) - \theta(x_2)$.

Claim C.3. $\theta(x_1) - \theta(x_2) \in \theta(S)$ if and only if the numerator in the canonical representation (see Definition B.1) of $\theta(x_1) - \theta(x_2)$ is contained in $[p - 2^{N_a}, p)$.

Proof. We will use the definition of θ^{-1} in the previous appendix. Let $e = \theta(x_1) - \theta(x_2)$. If $e \in \theta(S)$, then $\theta^{-1}(e) = a/b$, where $a \in [-2^{N_a}, 0)$ and $b \in \{1, 2, 2^2, \ldots, 2^{N_b}\}$. Thus, $\theta(a/b) = (p + a, b)$, where $p + a \in [p - 2^{N_a}, p)$, and $\theta(a/b) = \theta(\theta^{-1}(e)) \sim e$, so e has a canonical representation of the required form. On the other hand, if $e \sim (a, b)$, where $a \in [p - 2^{N_a}, p)$, then $\theta^{-1}(e) = (a - p)/b \in S$, so $e \sim \theta(\theta^{-1}(e)) \in \theta(S)$. ☐

Step C3

We instantiate step C3 with the following constraints $\mathcal{C}_<$:

$$\mathcal{C}_< = \left\{ \begin{array}{ll} A_0((1,1) - A_0) & = (0,1), \\ A_1((2,1) - A_1) & = (0,1), \\ \vdots & \vdots \\ A_{N_a-1}((2^{N_a-1},1) - A_{N_a-1}) & = (0,1), \\ A - (p - 2^{N_a}, 1) - \sum_{i=0}^{N_a-1} A_i & = (0,1), \\ B_0((1,1) - B_0) & = (0,1), \\ B_1((1,1) - B_1) & = (0,1), \\ \vdots & \vdots \\ B_{N_b}((1,1) - B_{N_b}) & = (0,1), \\ \sum_{i=0}^{N_b} B_i - (1,1) & = (0,1), \\ B - \sum_{i=0}^{N_b} B_i \cdot (1, 2^i) & = (0,1), \\ \theta(X_1) - \theta(X_2) - A \cdot B & = (0,1) \end{array} \right\}$$

Lemma C.4. $\mathcal{C}_<$ is satisfiable if and only if the numerator in the canonical representation (see Definition B.1) of $\theta(x_1) - \theta(x_2)$ is contained in $[p - 2^{N_a}, p)$.

Proof. Assume that $X_3 = \theta(x_1) - \theta(x_2)$ has the required form (a, b). We have $k = \log_2 b \in \{0, 1, 2, \ldots, N_b\}$ and $a \in [p - 2^{N_a}, p)$. Now, take $B_k = (1,1)$ and all other $B_j = (0,1)$; this satisfies all of the B_i constraints, including $\sum_{i=0}^{N_b} B_i - (1,1) = (0,1)$, which requires that exactly one B_i be equal to $(1,1)$. For B, take $B = (1, b) = (1, 2^k)$, to satisfy $B - \sum_{i=0}^{N_b} B_i \cdot (1, 2^i) = (0,1)$.

Now, let $a' = a - (p - 2^{N_a})$. The binary representation of a' has bits $z_0, z_1, \ldots, z_{N_a-1}$. Set $A_i = (z_i, 1)(2^i, 1)$ for $i \in \{0, 1, \ldots, N_a - 1\}$. This will satisfy all of the individual A_i constraints. And, since $\sum_{i=0}^{N_a-1} A_i = (a', 1)$, we can take $A = (a, 1)$ to satisfy $A - (p - 2^{N_a}, 1) - \sum_{i=0}^{N_a-1} A_i = (0,1)$. The remaining constraint is the last one in the list. It is satisfiable because we took $B = (1, b)$ and $A = (a, 1)$, giving $X_3 - (a,1) \cdot (1,b) = (0,1)$.

For the other direction, if the constraints are satisfiable, then $X_3 = \theta(x_1) - \theta(x_2)$ can be written as $(a,1)(1,b)$, where $b \in \{1, 2, \ldots, 2^{N_b}\}$ and where $a = p - 2^{N_a} + \sum_{i=0}^{N_a-1} z_i 2^i$, for $z_i \in \{0, 1\}$. This implies that $a \in [p - 2^{N_a}, p)$. ☐

In analogy with the integers case, notice that the lemma, together with the reasoning in "Preliminaries", implies the following corollary.

Corollary C.5. If the input restrictions are met, then $\mathcal{C}_<$ is satisfiable if and only if $x_1 < x_2$.

That is, $\mathcal{C}_<$ is equivalent to $<$ over $\mathbb{Q}$.

C.3 Branching

We now return to the case study in Section 4.2. We will abstract the domain ($\mathbb{Z}/p$ or $\mathbb{Q}/p$): when we write 0 in constraints below, it denotes the additive identity, which is $(0,1)$ in $\mathbb{Q}/p$, and when we write 1, it denotes the multiplicative identity, which is $(1,1)$ in $\mathbb{Q}/p$. Recall the computation Ψ and the constraints $\mathcal{C}_\Psi$ (Figure 3):

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

$$\mathcal{C}_\Psi = \left\{ \begin{array}{l} M\{\mathcal{C}_<\}, \\ M(Y-3) = 0, \\ (1-M)\{\mathcal{C}_{>=}\}, \\ (1-M)(Y-4) = 0 \end{array} \right\}$$

We now argue that $\mathcal{C}_\Psi$ is equivalent to Ψ. (The definition of "equivalent" is given in Section 2.1.)

Lemma C.6. The constraints $\mathcal{C}_\Psi(X_1 = x_1, X_2 = x_2, Y = y)$ are satisfiable if and only if $y = \Psi(x_1, x_2)$.

Proof. Assume $\mathcal{C} = \mathcal{C}_\Psi(X_1 = x_1, X_2 = x_2, Y = y)$ is satisfiable. Since $\mathcal{C}_<$ and $\mathcal{C}_{>=}$ cannot be simultaneously satisfiable (that would imply opposing logical conditions), then $M = 0$ or $1 - M = 0$. If $1 - M = 0$, then $y = 3$, since we are given that the constraint $M(Y - 3) = 0$ is satisfiable when $Y = y$. Moreover, $\mathcal{C}_{<}$ must be satisfiable, implying that $x_1 < x_2$ (see Corollaries C.2 and C.5). On the other hand, by analogous reasoning, if $M = 0$, then $y = 4$, $\mathcal{C}_{>=}$ is satisfiable, and $x_1 \geq x_2$. Thus, we have two cases: (1) $x_1 < x_2$ and $y = 3$ or (2) $x_1 \geq x_2$ and $y = 4$. But this means that $y = \Psi(x_1, x_2)$ for all x_1, x_2 in the permitted input.

Now assume that $y = \Psi(x_1, x_2)$. If $x_1 < x_2$, then $y = 3$. Take $M = 1$ to satisfy the constraints $(1 - M)\{\mathcal{C}_{>=}\} = 0$ and $(1-M)(Y-4) = 0$. Also, $M(Y-3) = 0$ is satisfied, because $y = 3$. Last, $\mathcal{C}_{<}$ can be satisfied, because $x_1 < x_2$. Thus, the constraints are satisfiable if $x_1 < x_2$. Similar reasoning establishes that the constraints are satisfiable if $x_1 \geq x_2$. □

We can generalize the computation Ψ. For instance, let Ψ_1, Ψ_2 be sub-computations, which we abbreviate in code as `comp1` and `comp2`. Let $\mathcal{C}_{\Psi_1}$ and $\mathcal{C}_{\Psi_2}$ denote the constraints that are equivalent to Ψ_1 and Ψ_2, and rename the distinguished output variables in $\mathcal{C}_{\Psi_1}$ and $\mathcal{C}_{\Psi_2}$ to be Y_1 and Y_2, respectively. Below, Ψ and $\mathcal{C}_\Psi$ are equivalent:

```
Ψ :

if (X1 < X2)
  Y = comp1
else
  Y = comp2
```

$$\mathcal{C}_\Psi = \left\{ \begin{array}{l} M\{\mathcal{C}_{<}\}, \\ M\{\mathcal{C}_{\Psi_1}\}, \\ M(Y - Y_1) = 0, \\ (1 - M)\{\mathcal{C}_{>=}\}, \\ (1 - M)\{\mathcal{C}_{\Psi_2}\}, \\ (1 - M)(Y - Y_2) = 0, \end{array} \right\}$$

The reasoning that establishes the equivalence is very similar to the proof of Lemma C.6. (The differences are as follows. In the forward direction, take $M = 1$. Then, since $Y = y$ and $M(Y - Y_1)$ is satisfied, $Y_1 = y$; meanwhile, $\mathcal{C}_{\Psi_1}(Y_1 = y, X_1 = x_1, X_2 = x_2)$ must be satisfied, which implies $y = \Psi_1(x_1, x_2)$ and hence $y = \Psi(x_1, x_2)$. In the reverse direction, take $x_1 < x_2$. Then we have $y = \Psi_1(x_1, x_2)$. But this implies that when $Y_1 = y$, we can satisfy $\mathcal{C}_{\Psi_1}$, so set $Y_1 = y$. Furthermore, set $Y = y$, and we thus satisfy $M(Y - Y_1)$ and hence all constraints.)

We can generalize further. First, the logical test in the "if" can be an arbitrary test constructed from `==`, `!=`, `&&`, `||`, `>`, `>=`, `<`, `<=`; in this case, we must also construct the negation of the test (just as we need constraints that represent both $\mathcal{C}_{<}$ and $\mathcal{C}_{>=}$). Second, we need not capture the result of the conditional in Y; we can assign the result to an intermediate variable Z. In that case, we would replace the constraints $M(Y - Y_1) = 0$ and $(1 - M)(Y - Y_2) = 0$ with $M(Z - Y_1)$ and $(1 - M)(Z - Y_2)$, respectively, and of course we would need other constraints that capture the flow from Z to the ultimate output, Y.

D Program constructs and costs

This appendix describes further program constructs; as with the case study, the work here corresponds to step C3 in our framework. However, in this appendix, we will not delve into as much detail as in the previous appendices; a more precise syntax and semantics is future work. Below, we describe how we map program constructs to constraints and then briefly consider the costs of doing so.

D.1 Program constructs

Aside from order comparisons, the computations and constraints below are independent of the domain of the computation; as in Appendix C.3, 0 and 1 denote the additive and multiplicative identities in the field in question.

Tests

`==`. Consider the fragment `(comp1) == (comp2)`, where `comp1` and `comp2` are computations Ψ_1 and Ψ_2. Renaming the output variables in Ψ_1 and Ψ_2 to be Y_1 and Y_2, respectively, we can represent the fragment with the constraint $Y_1 - Y_2 = 0$.

`!=`. Consider the program fragment `Z1 != Z2`. An equivalent constraint is $M \cdot (Z_1 - Z_2) - 1 = 0$, where M is a new auxiliary variable. This constraint is satisfiable if and only if $Z_1 - Z_2$ has a multiplicative inverse; that is, it is satisfiable if and only if $Z_1 - Z_2 \neq 0$, or $Z_1 \neq Z_2$. As above, we can represent `(comp1) != (comp2)`; the constraint would be $M \cdot (Y_1 - Y_2) - 1 = 0$.

`<`, `<=`, `>`, `>=`. Appendix C described in detail the constraints that represent `<`. A similar approach applies for the other three order comparisons. For example, for `X1 <= X2` over the rationals, we want to enforce that the canonical numerator (see Definition B.1) of $X_1 - X_2 \in [p - 2^{N_a}, p) \cup \{0\}$. To do so, we modify $\mathcal{C}_{<}$ in Appendix C.2 as follows. First, we add a constraint $A'_0(A'_0 - (1, 1)) = (0, 1)$. Second, we change the A constraint from

$$A - (p - 2^{N_a}, 1) - \sum_{i=0}^{N_a - 1} A_i = (0, 1)$$

to

$$A - (p - 2^{N_a}, 1) - \sum_{i=0}^{N_a - 1} A_i - A'_0 = (0, 1).$$

Composing tests into expressions

To compose logical expressions, we provide `&&` and `||`. We do not provide logical negation explicitly, but our computational model includes inverses for all tests (for example, `==` and `!=`), so the programmer or compiler can use DeMorgan's laws to write the negation of any logical expression in terms of `&&` and `||`.

`||`. Consider the expression `(cond1) || (cond2)`, and let $\mathcal{C}_1$ and $\mathcal{C}_2$ be the constraints that are equivalent to `cond1` and `cond2` respectively. The expression is equivalent to the following constraints, where M_1, M_2 are new variables:

$$\mathcal{C}_{||} = \left\{ \begin{array}{l} (M_1 - 1)(M_2 - 1) = 0, \\ M_1\{\mathcal{C}_1\}, \\ M_2\{\mathcal{C}_2\} \end{array} \right\}$$

We now argue that $\mathcal{C}_{||}$ is equivalent to the original expression. If `cond1` holds, then $\mathcal{C}_1$ is satisfiable; choose $M_1 = 1$ and $M_2 = 0$ to satisfy all constraints. Note that if `cond2` also holds, then setting $M_2 = 1$ also works, but the prover might wish to avoid "executing" (i.e., finding a satisfying assignment for) $\mathcal{C}_2$. On the other hand, if $\mathcal{C}_{||}$ is satisfiable, then $M_1 = 1$ or $M_2 = 1$, or both. If $M_1 = 1$, then $\mathcal{C}_1$ is satisfied, which implies that `cond1` holds. The identical reasoning applies if $M_2 = 1$.

`&&`. To express `(cond1) && (cond2)`, the programmer simply includes $\mathcal{C}_1$ and $\mathcal{C}_2$.

Conditionals

We covered conditional branching in detail in Appendix C.3. Below we describe two other conditional constructs, EQUALS-ZERO and NOT-EQUALS-ZERO, that are useful as "type casts" from integers to 0-1 values.

NOT-EQUALS-ZERO. The computation Ψ is `Y = (X != 0) ? 1 : 0`, and it can be represented with the following constraints:

$$\mathcal{C}_{\text{NOT-EQUALS-ZERO}} = \left\{ \begin{array}{l} X \cdot M - Y = 0, \\ (1 - Y) \cdot X = 0 \end{array} \right\}.$$

One can verify by inspection that $\mathcal{C}_{\text{NOT-EQUALS-ZERO}}(Y = y, X = x)$ is satisfiable if and only if $y = \Psi(x)$. Note that we could implement NOT-EQUALS-ZERO by using a conditional branch (see Appendix C.3) together with a `!=` test (see above). However, relative to that option, the constraints above are more concise (fewer constraints, fewer variables). They are also more concise than the representation given by Cormode et al. [21]. Roughly speaking, Cormode et al. represent NOT-EQUALS-ZERO with a constraint like $X^{p-1} - Y = 0$, where p is the modulus of $\mathbb{Z}/p$ (the approach works because Fermat's Little Theorem says that for any non-zero X, $X^{p-1} \equiv 1 \pmod{p}$); this approach requires $\log p$ intermediate variables.

EQUALS-ZERO. This computation is the inverse of the previous; the constraint representation of $\mathcal{C}_{\text{EQUALS-ZERO}}$ is the same as $\mathcal{C}_{\text{NOT-EQUALS-ZERO}}$ but with Y replaced by $1-Y$.

D.2 Costs

As mentioned in Section 4.2, there are two main costs of the constructs above. First, the order comparisons require a variable and a constraint for each bit position, Second, the constraints for conditional branching and `||` appear to be degree-3 or higher—notice the $M\{\mathcal{C}\}$ notation in these constructs—but must be reduced to be degree-2, as required by the protocol (see Sections 2.1–2.2). Below, we describe this reduction and its costs.

We will start with the degree-3 case and then generalize. Let $\mathcal{C}$ be a constraint set over variables $\{Z_1, \ldots, Z_n, M\}$, and let $\mathcal{C}$ have a degree-3 constraint, $Q(Z_1, \ldots, Z_n, M)$. Q has the form $R(M) \cdot S(Z_1, \ldots, Z_n)$, where $R(M)$ is M or $(1-M)$; this follows because higher-degree constraints only ever emerge from multiplication by an auxiliary variable. We reduce Q by constructing a $\mathcal{C}'$ that is the same as $\mathcal{C}$ except that Q is replaced with the following two constraints, using a new variable M':

$$\begin{array}{l} M' - S(Z_1, \ldots, Z_n) = 0, \\ R(M) \cdot M' = 0. \end{array}$$

Claim D.1. $\mathcal{C}$ is satisfiable if and only if $\mathcal{C}'$ is satisfiable.

Proof. Abbreviate $Z = Z_1, \ldots, Z_n$. Assume $\mathcal{C}$ is satisfied by assignment $Z = z, M = m$. Use this same setting for $\mathcal{C}'$. So far, all constraints other than the two new ones are satisfied in $\mathcal{C}'$. To satisfy the two new ones, set $M' = S(z)$. This satisfies the first new constraint. It also satisfies the second new constraint because either $M' = 0$ or $M' \neq 0$, in which case $S(z) \neq 0$, which implies (because $\mathcal{C}$ is satisfied and hence Q is too) that $R(m) = 0$.

Now assume that $\mathcal{C}'$ is satisfiable with assignment $Z = z, M = m, M' = m'$. In $\mathcal{C}$, set $Z = z, M = m$. Now, in this assignment, in $\mathcal{C}'$, $R(m) = 0$ or $m' = 0$. If $R(m) = 0$, then $Q(z,m) = 0$. If $m' = 0$, then $S(z) = 0$, so $Q(z,m) = 0$ again. □

Since applying a single transformation of the kind above does not change the satisfiability of the resulting set, we can transform all of the constraints this way, to make $M\{\mathcal{C}\}$ degree-2. The costs of doing so are as follows. Each of the χ constraints in $\mathcal{C}$ causes us to add another constraint and a new variable. Thus, if $\mathcal{C}$ has s variables and χ constraints, then our representation of $M\{\mathcal{C}\}$ has $s + \chi$ variables and $2 \cdot \chi$ constraints.

The approach above generalizes to higher degrees. Higher-degree constraints emerge from nesting of branches or `||` operations. If there are k levels of nesting somewhere in the computation, then the computation's constraints have a subset of the form

$$\mathcal{C}_{\text{nested}} = M_k\{M_{k-1}\{\cdots M_1\{\mathcal{C}\}\cdots\}\}.$$

(Each of the M_i could also be $1 - M_i$.) Then there is a set of equivalent degree-2 constraints that uses in total $s + k \cdot \chi$ variables and $(k+1) \cdot \chi$ constraints.

The details are as follows. Consider a single constraint in $\mathcal{C}_{\text{nested}}$; it has the form $R(M_k) \cdots R(M_2)R(M_1)S(Z) = 0$, where $S(Z)$ is degree-2. Replace this constraint with

```
input:
  Xa: a string of size m
  Xb: two dimensional matrix of size m x m.
      Each row is a string of size m

output:
  Y: a vector of size m, where each entry is an
     unsigned integer

m-Hamming-distance(Xa, Xb):
  for (i = 1; i <= m; i++) {
    Y[i] = Hamming-distance(Xa, Xb[i]);
  }

unsigned int Hamming-distance(U1, U2):
  unsigned int D = 0;
  for (i = 1; i <= m; i++) {
    D += (U1[i] != U2[i]);
  }
  return D;
```

```
// constraint-friendly version
m-Hamming-distance(Xa, Xb):
  unsigned int Za[m];
  unsigned int Zd[m][m]; // 0-1 variables

  for (pos = 1; pos <= m; pos++) {
    Za[pos] = Xa[pos];
  }

  for (row = 1; row <= m; row++) {
    for (pos = 1; pos <= m; pos++) {
      Zd[row][pos] =
        (Za[pos] - Xb[row][pos] != 0) ? 1 : 0;
    }
  }

  for (row = 1; row <= m; row++) {
    Y[row] = Zd[row][1] + ... + Zd[row][m];
  }
```

Figure 15—Pseudocode for *m*-Hamming distance computation, presented two ways. The second presentation permits a more natural translation to constraints (see Figure 16).

the following ones, to form $\mathcal{C}'_{\text{nested}}$:

$$\begin{aligned}
&M'_0 - S(Z) = 0,\\
&M'_1 - R(M_1)\cdot M'_0 = 0,\\
&M'_2 - R(M_2)\cdot M'_1 = 0,\\
&\quad\vdots\\
&M'_{k-1} - R(M_{k-1})\cdot M'_{k-2} = 0,\\
&R(M_k)\cdot M'_{k-1} = 0.
\end{aligned}$$

The proof that $\mathcal{C}'_{\text{nested}}$ and $\mathcal{C}_{\text{nested}}$ are equivalent is similar to the proof of Claim D.1 and is omitted for the sake of brevity. Observe that this construction introduces, per constraint in $\mathcal{C}$, k new variables and k new constraints, leading to the costs stated above.

E Evaluation benchmarks

This appendix describes some of the benchmark computations from Section 6 and reports microbenchmarks to quantify our cost model (Figure 2).

E.1 Benchmark computations

Below, we cover in detail *m*-Hamming distance and bisection method; the other benchmark computations in Section 6 (Figure 5) are described elsewhere [45].

m-Hamming distance

Recall that the Hamming distance between two strings of the same length is the number of positions at which they differ. We define the *m-Hamming distance* computation as follows. The input is a string, x_a, of length m, and there is also a predefined set of m strings, x_b; the computation is to find the Hamming distance between the input string and every string in the predefined set (i.e., the output is a vector of length m containing integers). This formulation makes the work super-linear (motivating the use of GINGER) and is similar to a suggested use of Apache Mahout, namely computing "the pairwise similarity between all documents ... in a corpus".[8]

$$\left\{\begin{array}{ll}
Z_i^{(a)} - X_i^{(a)} = 0 & (1 \le i \le m)\\
(Z_j^{(a)} - X_{i,j}^{(b)})\cdot M_{i,j} - Z_{i,j}^{(d)} = 0 & (1 \le i,j \le m)\\
(1 - Z_{i,j}^{(d)})\cdot(Z_j^{(a)} - X_{i,j}^{(b)}) = 0 & (1 \le i,j \le m)\\
Y_i - \sum_{j=1}^{m} Z_{i,j}^{(d)} = 0 & (1 \le i \le m)
\end{array}\right\}$$

Figure 16—Constraints for the *m*-Hamming distance computation. The pseudocode for this computation is in Figure 15.

Figure 15 gives the pseudocode for the computation, in two forms. Notice that in the second form, there are three groups of "loops"; each group corresponds to an array of constraints of a given type. The three types are input handling, NOT-EQUALS-ZERO (see the previous appendix), and a summation at the end. The constraints themselves are listed in Figure 16.

We now describe a specialized PCP for this computation (such tailoring is discussed in Section 5 and [45]). Because the input variables $\{X\}$ will be assigned based on the user's input (§2.1), the only degree-2 terms in the constraints have the form $Z_j^{(a)}\cdot M_{i,j}$ and $Z_{i,j}^{(d)}\cdot Z_j^{(a)}$. Our linear PCP captures these terms. Specifically, the PCP π is given by a vector w of the following form:

$$(Z^{(a)}, M, Z^{(d)}, Z^{(a)}\otimes M, Z^{(d)}\otimes Z^{(a)}).$$

[8] https://cwiki.apache.org/confluence/display/MAHOUT/Algorithms

$$\left\{\begin{array}{ll} Z^{(a)}_{1,j} - X^{(a)}_j = (0,1) & (1 \le j \le m) \\ Z^{(b)}_{1,j} - X^{(b)}_j = (0,1) & (1 \le j \le m) \\ Z^{(c)}_i - F((1,2)\cdot(Z^{(a)}_i + Z^{(b)}_i)) = (0,1) & (1 \le i \le L) \\ \mathcal{C}_{i,\texttt{is>}}, \text{constraints for a sub-computation: ``}M_i = (Z^{(c)}_i > (0,1))\ ?\ (1,1) : (0,1)\text{''} & (1 \le i \le L) \\ Z^{(a)}_{i+1,j} - M_i \cdot Z^{(a)}_{i,j} - (1-M_i)\cdot(1,2)\cdot(Z^{(a)}_{i,j} + Z^{(b)}_{i,j}) = (0,1) & (1 \le i < L, 1 \le j \le m) \\ Z^{(b)}_{i+1,j} - (1-M_i)\cdot Z^{(b)}_{i,j} - M_i\cdot(1,2)\cdot(Z^{(a)}_{i,j} + Z^{(b)}_{i,j}) = (0,1) & (1 \le i < L, 1 \le j \le m) \\ Y^{(a)}_j - M_L \cdot Z^{(a)}_{L,j} - (1-M_L)\cdot(1,2)\cdot(Z^{(a)}_{L,j} + Z^{(b)}_{L,j}) = (0,1) & (1 \le j \le m) \\ Y^{(b)}_j - (1-M_L)\cdot Z^{(b)}_{L,j} - M_L\cdot(1,2)\cdot(Z^{(a)}_{L,j} + Z^{(b)}_{L,j}) = (0,1) & (1 \le j \le m) \end{array}\right\}$$

Figure 17—Constraints for the bisection method computation, as output by our compiler; the SFDL source code is in Figure 18. The constraints $\mathcal{C}_{i,\texttt{is>}}$ are not unpacked above; they use inequality comparisons (see Appendix C).

If the PCP is correct, the setting of $Z^{(a)}, M, Z^{(d)}$ in w satisfies the constraints. (Here, $Z^{(a)}$ refers to all $Z^{(a)}_j$, M refers to all $M_{i,j}$, and $Z^{(d)}$ refers to all $Z^{(d)}_{i,j}$, for $1 \le i,j \le m$.) The size of this encoding is $n = 2m^3 + 2m^2 + m$ elements.

In formulating the PCP tests, let $\pi^{(1)}, \ldots, \pi^{(5)}$ denote the linear functions that correspond to the 5 components of w above. The first PCP tests are two quadratic correction tests (since there are two outer products):

$$\begin{array}{l} \pi^{(1)}(q_1)\cdot\pi^{(2)}(q_2) \stackrel{?}{=} \pi^{(4)}(q_1 \otimes q_2 + q_3) - \pi^{(4)}(q_3) \\ \pi^{(1)}(q_4)\cdot\pi^{(3)}(q_5) \stackrel{?}{=} \pi^{(5)}(q_4 \otimes q_5 + q_6) - \pi^{(5)}(q_6), \\ \text{where} \quad q_1, q_4 \in_R \mathbb{F}^m \quad q_2, q_5 \in_R \mathbb{F}^{m^2} \quad q_3, q_6 \in_R \mathbb{F}^{m^3}. \end{array}$$

The final PCP test is a circuit test. Recall that to test a satisfying assignment, V constructs a polynomial, $Q(\cdot)$, as a random linear combination of its constraints (see [45, §2]). This $Q(\cdot)$ can be written in the following form:

$$\begin{array}{l} Q\left(Z^{(a)}, M, Z^{(d)}\right) = \\ \qquad \gamma_0 + \langle\gamma_1, Z^{(a)}\rangle + \langle\gamma_2, M\rangle + \langle\gamma_3, Z^{(d)}\rangle + \\ \qquad \langle\gamma_4, Z^{(a)} \otimes M\rangle + \langle\gamma_5, Z^{(d)} \otimes Z^{(a)}\rangle. \end{array}$$

(For similar constructions, see [45, §2 and AppendixD].) The circuit test, then, is to construct self-correcting queries from the γ_i (for instance, q_s and $\gamma_i + q_s$), to supply the self-correcting γ_i to $\pi^{(i)}$, and to check that the sum of the results equals $-\gamma_0$.

In our experiments, the input "characters" (in the "strings") are 32-bit unsigned quantities (see Figure 5), and we take $m = 100$. Thus, the number of constraints is $2m^2 + 2m = 20{,}200$. Also, the number of variables is $s = 2m^2 + m$ (per Figure 5), which is 20,100 (as stated in Figure 9). Note that n, the size of the encoding, is $O(m^3)$; with the standard (non-tailored) encoding, n would be $O(s^2) = O(m^4)$.

```
type Y = struct {float[m] Ya, float[m] Yb};
function Y output (float[m] Xa, float[m] Xb) {
  var int i; var int j;
  var float[m] Za;  var float[m] Zb;

  Za = Xa;  Zb = Xb;
  for (i = 0 to L-1) {
    if (fAtMidpt(Za, Zb) > 0) {
      for (j = 0 to m-1) {
        Za[j] = 1/2*(Za[j] + Zb[j]);
      }
    } else {
      for (j = 0 to m-1) {
        Za[j] = 1/2*(Za[j] + Zb[j]);
      }
    }
  }
  output.Ya = Za;  output.Yb = Zb;
}
```

Figure 18—Partial SFDL code for the bisection method applied to a degree-2 polynomial, F, in m variables. The function $\texttt{fAtMidpt}(a,b)$ evaluates the polynomial F at the midpoint of a and b.

Root finding by bisection

This computation identifies roots of a function. Root-finding of course has many uses (in optimization, signal processing, etc.). The "bisection method", as we call it, finds a root of a continuous function in a given interval, provided the interval brackets a root. The method repeatedly bisects the search interval, searching in one of the two subintervals. As the number of bisections $L \to \infty$, the search interval is guaranteed to converge to a root of the function inside the given interval.

Our benchmark computation applies this algorithm to find roots of a degree-2 polynomial in m variables, and we assume that the verifier knows the endpoints of the search space. Note that this is a minor restriction since the computation can be modified slightly, with little additional cost, to have the prover supply those endpoints (say by randomly evaluating the polynomial to identify valid endpoints). Figure 17 depicts the constraints produced by our compiler; the SFDL source code is depicted in Figure 18. The `float` and `int` types in our SFDL are an extension to Fairplay's SFDL [39].

In our experiments, we take the number of iterations, L, to be 8. The inputs are vectors of $m = 25$ floating-point numbers (using our primitive representation) with $N_a = 32$ and $N_b = 5$ (see Section 4.1). Thus, we can bound the computation to the subset $U = \{a/b : |a| < 2^{100}, b \in \{1, 2, 2^2, \ldots, 2^{30}\}\}$. This allows us to implement $\mathcal{C}_{i,\texttt{is>}}$ with 145 constraints and 140 variables (including the $\{M_i\}$). Adding up the constraints and variables in Figure 17, we get 1618 constraints and 1528 variables (as stated in Figure 9).

E.2 Microbenchmarks

To quantify the parameters in the cost model, we run a program that executes each operation (encryption, decryption, etc.) at least 5000 times using 1024-bit keys with inputs chosen from different finite fields. Here are the mean execution times of the operations (standard deviations are within 5% of the means):

field size	e	d	h	f	c
128 bits	72 μs	170 μs	190 μs	18 ns	69 ns
192 bits	85 μs	170 μs	280 μs	45 ns	69 ns
320 bits	280 μs	170 μs	560 μs	180 ns	69 ns