Making Argument Systems for Outsourced Computation Practical (Sometimes)

Srinath Setty, Richard McPherson, Andrew J. Blumberg, and Michael Walfish
The University of Texas at Austin

Abstract

This paper describes the design, implementation, and evaluation of a system for performing verifiable outsourced computation. It has long been known that (1) this problem can be solved in theory using probabilistically checkable proofs (PCPs) coupled with modern cryptographic tools, and (2) these solutions have wholly impractical performance, according to the conventional (and well-founded) wisdom. Our goal is to challenge (2), with a built system that implements an *argument system* based on PCPs. We describe a general-purpose system that builds on work of Ishai et al. (CCC '07) and incorporates new theoretical work to improve performance by 20 orders of magnitude. The system is (arguably) practical in some cases, suggesting that, as a tool for building secure systems, PCPs are not a lost cause.

1 Introduction

This paper describes progress toward the goal of practical and general-purpose verifiable outsourced computation. This broad area has seen renewed interest, owing to cloud computing (a computationally limited device offloads processing to the cloud but does not assume the cloud's correctness [36]), volunteer computing (some 30 projects on the BOINC [1] software platform use volunteers' spare cycles, but some "volunteers" return wrong answers [6]), peer-to-peer computing, and high assurance computing (in the latter cases, remote peers or components are likely to be untrusted). Research has resulted in practical solutions, which depend on various conditions particular to the operating regime (e.g., they require trusted hardware [27, 62], or assume independent failures and replicate [6, 25, 42, 52, 56]), and theoretical solutions, which tend toward specialized algorithms for the computation at hand. There is, however, a strand of theory that is *unconditional* and *general-purpose*. Unfortunately, this theory has so far been totally impractical, which leads to the question: can we incorporate this powerful and enticing theory into a built system?

Our focus is on *argument systems*, which are interactive protocols with two actors, a *prover* and a *verifier*. Assuming that the prover is computationally bounded, such protocols can convince the verifier that the prover executed a given computation correctly and that the prover holds a proof to that effect. This theory dates to the 1980s [24, 39], and starting with Kilian [48, 49] *efficient* argument systems [45] have been based on probabilistically checkable proofs (PCPs)—which themselves are astonishing. Informally, the central result is that a verifier can—with a suitably encoded proof and with a negligible chance of regarding a wrong answer as correct—check an answer's correctness by probing a constant number of locations in a proof [8, 9]. Unfortunately, PCPs and hence arguments are wildly impractical: traditional PCPs are too expensive to instantiate at the prover or query from the verifier. While state-of-the-art PCP schemes are asymptotically efficient [15–17, 30], the constants on their running times are large, and they seem too intricate to be implemented easily.

This brings us to our animating question: can we construct an argument system for outsourced computation that has practical performance and is simple? Specifically, we require, first, that the verifier do less work than if it executed the outsourced computation locally. Some of the theoretical schemes meet this requirement asymptotically, but we want to meet it for reasonable computation sizes. Second, we require that the prover not do much more work than it would without verification. Last, we strongly favor protocols that are simple for both verifier and prover: a simple protocol is easier to implement, check for correctness, and optimize for performance.

Our starting point in answering the above question is a powerful and elegant insight of Ishai, Kushilevitz, and Ostrovsky [45]: some PCPs, though hopelessly impractical to materialize explicitly, are linear functions for which evaluation at any given point is inexpensive. Ishai et al. use such *linear PCPs* to build argument systems, via a novel *linear commitment* protocol: the prover commits to a linear function (the PCP), after which the verifier can inexpensively "query the proof" by asking the prover to evaluate the linear function at a verifier-chosen point. Since linear PCPs are simple (as PCPs go), the resulting argument systems are too. Nonetheless, they are still not practical. For example, for $m \times m$ matrix multiplication, the verifier would have to do $10^9 \cdot m^6$ work—a factor of $10^9 \cdot m^3$ more than executing the computation locally. This overhead comes from the encoding of the computation, a large number of cryptographic commitments, and substantial setup costs.

This paper describes a built system, called PEPPER, that goes a long way toward making argument systems practical. PEPPER refines the protocol of Ishai et al. to shrink program encoding (via a concise representation that generalizes arithmetic circuits), reduce the cost of commitment (via a stronger and streamlined commitment primitive), and amortize the verifier's costs (via batching). We

prove the soundness of these refinements. The result is a simple constant-round protocol that gains a factor of over 10^{17} (no joke) versus a naive implementation of [45].

Much (but not all) of PEPPER's remaining costs stem from the computational complexity of the prover's work. This problem is shared by all PCP schemes, so a protocol in which the prover's total work is not much greater than the cost of executing the computation would be a major improvement. In fact, a key innovation in PEPPER is to meet this goal for computations of real interest (matrix multiplication, polynomial evaluation, etc.) by tailoring the PCP encoding to reduce overhead. The result is roughly three more orders of magnitude saved, for a total of 20. We note that the tailoring is systematic, and we believe that it will lend itself to automation (see §3.6, §4.2, and §6).

The tailored protocols for our examples are close to practical: at problem sizes of $m = 500$ variables, for instance, the verifier benefits from outsourcing degree-2 polynomial evaluation when working over $25,000$ batched computations, and the verification takes minutes. And with two heuristic optimizations, we can get these numbers down to a batch size of 118 and sub-second verification time.

Of course, PEPPER is not yet ready for production deployment, but its remaining obstacles, while significant, do not seem insurmountable. Modular exponentiation is a bottleneck, but this primitive occurs in many cryptographic protocols and will benefit as researchers optimize it. Another obstacle is the computation encoding. One might guess that arithmetic circuits are too inefficient to be useful, but we discovered that we could tailor circuits until they imposed no overhead above a C++ program (§3.3). Of course, in the real world, people perform computations besides self-contained numerical functions. However, our work suggests that it will be possible to build a compiler that transforms a broader class of computations to efficient tailored protocols. Also, we had to start somewhere, to move PCPs from theory to technology. Our hope now is that PEPPER opens the door to an exciting area of systems research.

To explain this paper's organization, we note that there is a contrast between the "what" and the "why" of PEPPER. The "what" is surprisingly simple: the verifier submits vectors to the prover and expects dot products in return. The "why", however, requires some notation and time to explain, as PEPPER builds on the theory of PCPs, which is subtle and counter-intuitive. For this reason, Section 2 is a fairly technical overview of the underlying machinery. While this background is necessary for the precise specification of our protocols, the trusting reader can probably skip to Section 3, where we discuss the innovations and details of PEPPER. We experimentally evaluate PEPPER in Section 4. We survey related work in Section 5. Here, we just note that while there has been much theoretical work on unconditional, general-purpose verifiable computation, these works are not yet practical, and there have been few concerted efforts to make them so.

We originally proposed this research program in a position paper [64]. However, that paper gave only a high-level sketch that we have since heavily modified; it did not give a concrete protocol, proofs, an implementation, or an evaluation. The specific contributions of this paper are (1) a built PCP-based system that is near practical; (2) a series of refinements, with proofs, to the protocol of Ishai et al. [45] that save 20 orders of magnitude; (3) tailored PCP protocols that result in a prover whose overhead is only a constant factor; and (4) the implementation and experimental evaluation of our system, PEPPER.

2 Preliminaries and base protocol

2.1 A crash course in PCPs

Consider a *verifier* V that wishes to be convinced that a given problem instance X (e.g., a given 3-CNF logical formula) is in an NP language L (e.g., the set of satisfiable 3-CNF formulas). By definition of NP, if X is in L, then there is some witness z that V can check in polynomial time to be convinced of X's membership in L. Remarkably, there also exists a proof π that convinces V of X's membership but only needs to be inspected in a *constant number* of places—yet if X is not in L, then for any purported proof, the probability that V is wrongly convinced of X's membership can be arbitrarily close to zero. This remarkable statement is the rough content of the PCP theorem [8, 9], and the rest of this subsection describes the machinery from this theorem that is relevant to PEPPER.

Following [8], we take L to be Boolean circuit satisfiability: the question of whether the input wires of a given Boolean circuit $\mathcal{C}$ can be set to make $\mathcal{C}$ evaluate to 1.[1] It suffices to consider this problem because L is NP-complete; any other problem in NP can be reduced to it. Of course, a satisfying assignment z—a setting of all wires in $\mathcal{C}$ such that $\mathcal{C}$ evaluates to 1—constitutes an (obvious) proof that $\mathcal{C}$ is satisfiable: V could check z against every gate in $\mathcal{C}$. Note that this check requires inspecting all of z. In contrast, the PCP theorem yields a V that makes only a *constant number* of queries to an oracle π and satisfies:

- **Completeness.** If $\mathcal{C}$ is satisfiable, then there exists a linear function π (called a proof oracle) 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.
- **Soundness.** If $\mathcal{C}$ is not satisfiable, then $\Pr\{V \text{ accepts } \mathcal{C} \text{ as satisfiable}\} < \epsilon$ for *all* purported proof functions $\tilde{\pi}$. Here, ϵ is a constant that can be driven arbitrarily low.

[1] A Boolean circuit is a set of interconnected gates, each with input wires and an output wire, with wires taking 0/1 values.

Note that we are requiring a correct proof π to be a linear function over finite fields.[2] Following [45], we call such proofs *linear PCPs*. By linear function, we mean that $\pi(q_1+q_2) = \pi(q_1)+\pi(q_2)$. A linear function $\pi\colon \mathbb{F}^n \mapsto \mathbb{F}^b$ can be regarded as a $b \times n$ matrix M, where $\pi(q) = M \cdot q$; in the case $b = 1$, π returns a dot product with the input. Below we describe a linear PCP that is used by [8] as a building block; this linear PCP is also presented in [16, 45], and we borrow some of our notation from these three sources.

To motivate this linear PCP—that is, the encoding of π, and how V interacts with it—recall that we must avoid V's having to check a purported assignment, z, against every gate, as that would be equivalent to the polynomial-time check of membership. Instead, V will construct a polynomial $P(Z)$ that represents $\mathcal{C}$, and π will be carefully constructed to allow evaluation of this polynomial. For each of the s gates, V creates a variable $Z_i \in \{0,1\}$ that represents the output of gate i. V also creates a set of arithmetic constraints, as follows. If gate i is the AND of Z_j and Z_k, then V adds the constraint $Z_i - Z_j \cdot Z_k = 0$; if gate i is the NOT of Z_j, then V adds the constraint $1 - (Z_i + Z_j) = 0$; if gate i is an input gate for the jth input, V adds the constraint $Z_i - in_j = 0$; and finally, for the last gate, representing the output of the circuit, we also have $Z_s - 1 = 0$. V then obtains the polynomial $P(Z)$ by combining all of the constraints: $P(Z) = \sum_i v_i \cdot Q_i(Z)$, where $Z = (Z_1, \ldots, Z_s)$, each $Q_i(Z)$ is given by a constraint (e.g., $Z_i - Z_j Z_k$), and V chooses each v_i uniformly and independently at random from a finite field $\mathbb{F}$. The reason for the randomness is given immediately below.

Notice that $P(z)$ detects whether z is a satisfying assignment: (1) if z is a satisfying assignment to the circuit, then it also satisfies all of the $\{Q_i(Z)\}$, yielding $P(z) = 0$; but (2) if z is not a satisfying assignment to the circuit, then the randomness of the $\{v_i\}$ makes $P(z)$ unlikely to equal 0 (as illustrated in the next paragraph). Thus, the proof oracle π must encode a purported assignment z in such a way that V can quickly evaluate $P(z)$ by making a few queries to π. To explain the encoding, let $\langle x, y\rangle$ represent the inner (dot) product between two vectors x and y, and $x \otimes y$ represent the outer product $x \cdot y^T$ (that is, all pairs of components from the two vectors). Observe that V can write

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

The $\{\gamma_0, \gamma_1, \gamma_2\}$ are determined by the $\{Q_i(Z)\}$ and the choice of $\{v_i\}$, with $\gamma_2 \in \mathbb{F}^{s^2}$, $\gamma_1 \in \mathbb{F}^s$, and $\gamma_0 \in \mathbb{F}$. The reason that V can write $P(Z)$ this way is that all of the $\{Q_i(Z)\}$ are degree-2 functions.

Given this representation of $P(Z)$, V can compute $P(z)$ by asking for $\langle \gamma_2, z \otimes z\rangle$ and $\langle \gamma_1, z\rangle$. This motivates the form of a correct proof, π. We write $\pi = (z, z \otimes z)$, by which we mean $\pi = (\pi^{(1)}, \pi^{(2)})$, where $\pi^{(1)}(\cdot) = \langle \cdot, z\rangle$ and $\pi^{(2)}(\cdot) = \langle \cdot, z \otimes z\rangle$. At this point, we have our first set of queries: V checks whether $\pi^{(2)}(\gamma_2) + \pi^{(1)}(\gamma_1) + \gamma_0 = 0$. If z is a satisfying assignment and π is correctly computed, the check passes. Just as important, if z' is *not* a satisfying assignment—which is always the case if $\mathcal{C}$ is not satisfiable—then V is not likely to be convinced. To see this, first assume that V is given a syntactically correct but non-satisfying π'; that is, $\pi' = (z', z' \otimes z')$, where z' is a non-satisfying assignment. The test above—that is, checking whether $\pi'^{(2)}(\gamma_2) + \pi'^{(1)}(\gamma_1) + \gamma_0 = 0$—checks whether $P(z') = 0$. However, there must be at least one i' for which $Q_{i'}(z')$ is not 0, which means that the test passes if and only if $v_{i'} \cdot Q_{i'}(z') = -\sum_{i \neq i'} v_i \cdot Q_i(z')$. But the $\{v_i\}$ are conceptually chosen *after* z', so the probability of this event is upper-bounded by $1/|\mathbb{F}|$.

The above test is called the *circuit test*, and it has so far been based on an assumption: that if π' is invalid, it encodes *some* (non-satisfying) assignment. In other words, we have been assuming that $\pi'^{(1)}$ and $\pi'^{(2)}$ are linear functions that are consistent with each other. But of course a malevolently constructed oracle might not adhere to this requirement. To relax the assumption, we need two other checks. First, with *linearity tests* [12, 21], V makes three queries to $\pi^{(1)}$ and three to $\pi^{(2)}$, and checks the responses. If the checks pass, V develops a reasonable confidence that $\pi^{(1)}$ and $\pi^{(2)}$ are linear functions, which is another way of saying that $\pi^{(1)}(\cdot)$ is returning $\langle \cdot, z\rangle$ for some z and that $\pi^{(2)}(\cdot)$ is returning $\langle \cdot, u\rangle$ for some $u \in \mathbb{F}^{s^2}$. In the second test, the *quadratic correction test*, V makes four queries total and checks their responses; if the checks pass, V develops reasonable confidence that these two linear functions have the required relationship, meaning that $u = z \otimes z$. Once these tests have passed, the circuit test above is valid.

In all, V makes $\ell = 14$ queries. The details of the queries and tests, and a formal statement of their completeness and soundness, are in [8] and Appendix A. Here, we just informally state that if $\mathcal{C}$ is satisfiable, then V will always be convinced by π, and if $\mathcal{C}$ is not satisfiable, then V's probability of passing the tests is upper bounded by a constant κ (for any $\tilde{\pi}$). If the scheme is repeated ρ times, for $\mu = \ell \cdot \rho$ total queries, the error probability ϵ becomes $\epsilon = \kappa^\rho$.

2.2 Arguments

The theory above says that V can be convinced of a circuit's satisfiability by making only a constant number of queries to the proof oracle π. But how does V query π? Clearly, π cannot be available for direct inspection by V because it is tremendous: written out, π would be a list of the function's values at every point in an exponentially-sized domain. The idea of an efficient argument (due to Kilian [48, 49]) is to use a PCP in an interactive protocol: a separate *prover* P computes a PCP and responds to V's queries. However, P

[2] The PCP theorem permits, but does not impose, this restriction; we need it for reasons elucidated in the next subsection.

must be forced to "act like" a fixed proof—P must be prevented from adjusting answers to later queries based on earlier answers. Kilian's observation is that this can be done with cryptographic commitments.

In the sections ahead, we build on an elegant scheme by Ishai, Kushilevitz, and Ostrovsky [45]. They (1) observe that π is a linear function (determined by z and $z \otimes z$) and (2) develop a *commitment to a linear function* primitive. In this primitive, P commits to a linear function by pre-evaluating the function at a point chosen by V and hidden from P; then, V submits one query, and the response must be consistent with the pre-evaluation. Roughly speaking, V can now proceed as if P's responses are given by an oracle π. (More accurately, V can proceed as if P's responses are given by a set of non-colluding oracles, one per PCP query.)

In more detail, V obtains a commitment from P by homomorphically encrypting a random vector r and asking P to compute $\mathsf{Enc}(\pi(r))$; P can do this without seeing r, by the linearity of π and the homomorphic properties of the encryption function (we do not need or assume fully homomorphic encryption [37]). V can then apply the decryption function to recover $\pi(r)$. To submit a PCP query q and obtain $\pi(q)$, V asks P for $\pi(q)$ and $\pi(r + \alpha q)$, for α randomly chosen from $\mathbb{F}$. V then requires that $\pi(r + \alpha q) = \pi(r) + \alpha\pi(q)$, or else V rejects $\pi(q)$. By running parallel instances of the commitment protocol (one for each time that V wants to inspect π), Ishai et al. convert any PCP protocol that uses linear functions into an *argument system* [24, 39].

Arguments are defined as follows; we borrow some of our notation and phrasing from [45], and we restrict the definition to Boolean circuits. An *argument* (P, V) *with soundness error* ϵ comprises two probabilistic polynomial time entities, P and V, that take a Boolean circuit $\mathcal{C}$ as input and meet two properties:

- **Completeness.** If $\mathcal{C}$ is satisfiable and P has access to the satisfying assignment z, then the interaction of $V(\mathcal{C})$ and $P(\mathcal{C}, z)$ always makes $V(\mathcal{C})$ accept $\mathcal{C}$'s satisfiability.
- **Soundness.** If $\mathcal{C}$ is not satisfiable, then for every efficient malicious 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 $< \epsilon$.

3 Design and details of PEPPER

Problem statement and threat model. We wish to implement the following protocol. A computer that we control, the *verifier*, sends a program Ψ and an input x to a remote computer, the *prover*. The prover returns a result y, and the verifier issues queries over a small number of interaction rounds to the prover, whose responses either establish for the verifier that Ψ was run correctly and that y is the result of running Ψ on x, or else cause the verifier to reject. If y is incorrect, the verifier should reject with high probability. We do not provide the converse; that is, rejection in our context implies only that the prover has not given a correct proof, not that $y \neq \Psi(x)$. We require the following: (1) this protocol should be cheaper for the verifier, in computational resources, than computing y locally, and (2) the guarantee that the protocol provides to the verifier should make no assumptions about the prover, other than standard cryptographic ones. That is, the prover can subvert the protocol, can choose not to follow it, can return wrong answers, etc.

3.1 The promise and perils of PCPs

In principle, the problem above can be solved with PCPs. In practice, a number of severe obstacles arise, as can be seen by the following attempt to use PCPs. There is a Boolean circuit $\mathcal{C}$ (which depends on Ψ, x, and y) such that $\mathcal{C}$ is satisfiable (that is, evaluates to 1) if and only if y is the correct output of Ψ run on x. Assume that the prover and verifier can efficiently derive $\mathcal{C}$, given Ψ, x, and y. Then, the prover can issue a PCP (as in §2.1) of $\mathcal{C}$'s satisfiability. At that point, if y is the correct output, the verifier can efficiently inspect the PCP to be convinced both that Ψ was executed correctly and that Ψ produces y when run on x; if y is not the correct output, the probability that the verifier accepts $\mathcal{C}$ as satisfiable is upper-bounded by a negligible constant—for *any* purported PCP. This approach sounds promising; unfortunately, it is totally impractical:

- *The proof is too long.* It is much too large for the verifier to handle or for the prover to write out in full.
- *The protocol is too complicated.* State-of-the-art PCP protocols [15–17, 30] partially address the concern about proof length, but they are intricate to the point of making a bug-free implementation difficult. Unfortunately, in this context, even small bugs can be security-critical. Complexity also hinders optimization.
- *The phrasing of the computation is too primitive.* Even if they have simple control flow, most general-purpose computations are far longer when expressed as Boolean circuits than in, say, C++.
- *The preparatory work is too high for the verifier.* Deriving $\mathcal{C}$ and generating queries to the proof take at least as much work for the verifier as executing the computation.
- *The prover's overhead is too high.* In the base PCP protocol (§2.1), the prover's work is at least quadratic in the number of circuit wires. While this overhead can be reduced to a polylogarithmic factor [15–17, 30] (at the cost of intricacy, as noted above), we ideally want to limit the prover's overhead to a small constant factor.

The first two obstacles above can be addressed by the argument system of Ishai et al. [45]. As discussed in Section 2.2, their approach addresses proof length (since the proof is not materialized) and proof complexity (since the proof is sim-

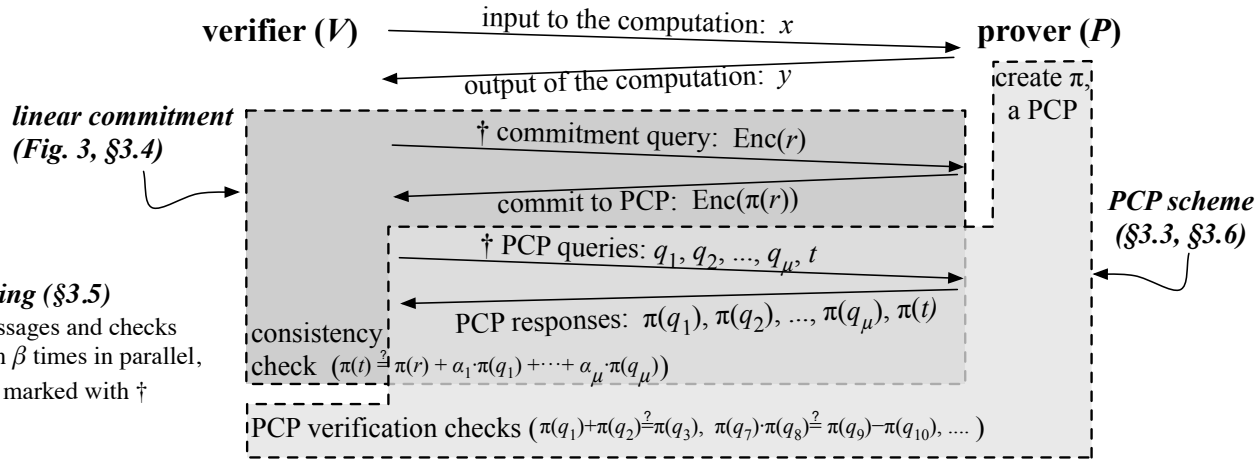

FIGURE 1—High-level depiction of PEPPER, a PCP-based argument system for verified outsourced computation between a verifier and a prover. We formalize this picture and prove its soundness in the appendices.

ply a linear function to which the prover commits). Proof length and proof complexity in exchange for commitment: this is a great trade! However, it brings another obstacle:

- *The parallel commitments are too expensive.* Commitment requires cryptographic operations and hence multiprecision arithmetic, and the scheme of [45] invokes these operations far too much (by several orders of magnitude) to be practical.

We now turn to our system, PEPPER, which builds on [45] and addresses all of the items above.

3.2 Overview of PEPPER

Figure 1 depicts PEPPER. In the computation phase, V selects a computation[3] Ψ and different inputs $x_1, \ldots, x_\beta$, and P returns outputs $y_1, \ldots, y_\beta$. These computations can happen iteratively or in a batch. For each computation, P creates and stores a proof oracle π_i establishing that $\Psi(x_i) = y_i$. In the proof commitment phase, V asks P to commit to all of its proofs. Then V moves to the proof verification phase and issues PCP queries to P, whose responses must be consistent with the commitment phase. Then, V runs the PCP verification checks on P's responses. At that point, V outputs `accept` or `reject` for each computation. An output of `accept` on y_i means that $y_i = \Psi(x_i)$, with high probability.

The rest of this section describes how PEPPER overcomes the obstacles mentioned in Section 3.1. PEPPER addresses proof length and complexity by inheriting from [45]; it shrinks the program encoding by using arithmetic circuits instead of Boolean circuits (§3.3); it reduces commitment costs by requiring fewer commitments while offering better security (§3.4); it uses batching to reduce V's costs (§3.5); and it reduces P's overhead for some problems by reducing redundancy in the PCP encoding (§3.6). Figure 2 summarizes the costs under these refinements; we explain this figure over the course of the section.

[3] Our implementation supports both preconfiguring computations and uploading binaries online for P to execute.

3.3 Arithmetic circuits and concise gates

To address the concern about the encoding of the computation, PEPPER uses *arithmetic circuits*, instead of Boolean circuits. In a traditional arithmetic circuit, the input and output wires take values from a large set (e.g., a finite field or the integers). This extension is a natural one, as the PCP machinery is already expressed as arithmetic versions of Boolean circuits. However, we observe that the machinery also works with what we call *concise gates*, each of which encapsulates a function of many inputs (e.g., a dot product between two large vectors). Note that a gate here does not represent a low-level hardware element but rather a modular piece of the computation that enters the verification algorithm as an algebraic constraint.

This simple refinement is critical to practicality. First, it is vastly more compact to represent, say, multiplication with a single gate than as a Boolean circuit. Beyond that, for certain computations (e.g., parallelizable numerical ones, such as matrix multiplication), the circuit model imposes no overhead; that is, the "circuit" is the same as a C++ program, so the only overhead comes from proving and verifying (as demonstrated in §4.3). However, this model has known limitations; for example, comparison operations require relatively large circuits. Future work is to address this problem, perhaps using tailored PCP encodings (§3.6).

Details and an example. Using arithmetic circuits requires only minor modifications to the PCP scheme described in Section 2.1. Here, V produces a set of constraints (there, V

	op	naive impl. of [45]	batching (§3.5)	arith. circ. (§3.3)	new commit (§3.4)	new PCPs (§3.6)
PCP encoding size ($\|\pi\|$)		10^9m^6	10^9m^6	$4m^4$	$4m^4$	m^3
V's per-instance work (compare to local, naive computation: $f \cdot m^3$)						
Issue commit queries	$e+2c$	$10^9m^6 \cdot \mu'$	$10^9m^6 \cdot \mu'/\beta$	$4m^4 \cdot \mu'/\beta$	$4m^4/\beta$	m^3/β
Process commit responses	d	μ'	μ'	μ'	1	1
Issue PCP queries	c	$10^9m^6 \cdot \mu'$	$10^9m^6 \cdot \mu'/\beta$	$4m^4 \cdot \mu'/\beta$	$4m^4 \cdot \mu/\beta$	$m^3 \cdot \mu/\beta$
Process PCP responses	f	$(2\mu' + 96m^2 \cdot \rho)$	$(2\mu' + 96m^2 \cdot \rho)$	$(2\mu' + 3m^2 \cdot \rho)$	$(2\mu + 3m^2 \cdot \rho)$	$(2\mu + 3m^2 \cdot \rho)$
P's per-instance work (compare to local, naive computation: $f \cdot m^3$)						
Issue commit responses	h	$10^9m^6 \cdot \mu'$	$10^9m^6 \cdot \mu'$	$4m^4 \cdot \mu'$	$4m^4$	m^3
Issue PCP responses	f	$10^9m^6 \cdot \mu'$	$10^9m^6 \cdot \mu'$	$4m^4 \cdot \mu'$	$4m^4 \cdot \mu$	$m^3 \cdot \mu$

$\ell = 14$: # of PCP queries per repetition (Appendix A)
$\rho = 70$: # of repetitions to drive error low (Appendix A)
$\mu = \ell \cdot \rho \approx 1000$: # of PCP queries per instance
$\mu' \approx 3\mu$: # of PCP queries pre-commit-refinement (§3.4)
β: # of computation instances batch verified (§3.2, §3.5)
$|\pi|$: # of components in matrix associated to linear function π (§2.1)
e: cost of homomorphic encryption of a single field element (Fig. 3, step 1)
d: cost of homomorphic decryption of a single field element (Fig. 3, step 3)
f: cost of field multiplication (Fig. 3, steps 4–6)
h: cost of ciphertext addition plus multiplication (Fig. 3, step 2)
c: cost of generating a pseudorandom # between 0 and a 192-bit prime (§4.1)

FIGURE 2—High-order costs under our refinements, cumulatively applied, compared to naive local execution and a naive implementation of [45], for our running example of $m \times m$ matrix multiplication. Rows for V and P contain operation counts, except for the "op" field, which includes a parameter denoting the cost of the operations in that row. Section 4.5 quantifies d, e, f, h and c; Section 4.6 quantifies β. The "PCP" rows include the consistency queries and checks (Fig. 3, steps 4–6).

transforms a Boolean circuit into a set of constraints) over s variables from a finite field $\mathbb{F}$ (there, over binary variables) that can be satisfied if and only if y is the correct output of $\Psi(x)$ (there, if and only if the circuit is satisfiable); V then combines those constraints to form a polynomial over s values in the field $\mathbb{F}$ (there, in the field $GF(2)$). One way to look at this use of arithmetic circuits is that it represents the computation as a set of constraints directly.

To illustrate the above, we use the example of $m \times m$ matrix multiplication. We choose this example because it is both a good initial test (it is efficiently encodable as an arithmetic circuit) and a core primitive in many applications: image processing (e.g., filtering, rotation, scaling), signal processing (e.g., Kalman filtering), data mining, etc.

In this example computation, let A, B, C be $m \times m$ matrices over a finite field $\mathbb{F}$, with subscripts denoting entries, so $A = (A_{1,1}, \ldots, A_{m,m}) \in \mathbb{F}^{m^2}$ (for $\mathbb{F}$ sufficiently large we can represent negative numbers and integer arithmetic; see §4.1). The verifier V sends A and B to the prover P, who returns C; V wants to check that $A \cdot B = C$. Matrix C equals $A \cdot B$ if and only if the following constraints over variables $Z = (Z^a_{1,1}, \ldots, Z^a_{m,m}, Z^b_{1,1}, \ldots, Z^b_{m,m}) \in \mathbb{F}^{2m^2}$ can be satisfied:

$$Z^a_{i,j} - A_{i,j} = 0, \text{ for } i,j \in [m]\ ; \quad Z^b_{i,j} - B_{i,j} = 0, \text{ for } i,j \in [m]\ ;$$

$$C_{i,j} - \sum_{k=1}^{m} Z^a_{i,k} \cdot Z^b_{k,j} = 0, \text{ for } i,j \in [m].$$

Note that the third type of constraint, which captures the dot product, is an example of a concise gate.

V is interested in whether the above constraints can be met for some setting $Z = z$ (if so, the output of the computation is correct; if not, it is not). Thus, V proceeds as in Section 2.1 and Appendix A. V constructs a polynomial $P(Z)$ by combining the constraints: $P(Z) = \sum_{i,j} v^a_{i,j} \cdot (Z^a_{i,j} - A_{i,j}) + \sum_{i,j} v^b_{i,j} \cdot (Z^b_{i,j} - B_{i,j}) + \sum_{i,j} v^c_{i,j} \cdot (C_{i,j} - \sum_{k=1}^{m} Z^a_{i,k} \cdot Z^b_{k,j})$, where V chooses the variables $\{v\}$ randomly from $\mathbb{F}$. As before, V regards the prover P as holding linear proof oracles $\pi = (\pi^{(1)}, \pi^{(2)})$, where $\pi^{(1)}(\cdot) = \langle \cdot, z \rangle$ and $\pi^{(2)}(\cdot) = \langle \cdot, z \otimes z \rangle$ for some $z \in \mathbb{F}^{2m^2}$. And as before, V issues linearity test queries, quadratic correction test queries, and circuit test queries (the randomly chosen $\{v\}$ feed into this latter test), repeating the tests ρ times. We address V's cost of constructing $P(Z)$ and issuing queries in Section 3.5.

The completeness and soundness of the above scheme follows from the completeness and soundness of the base protocol. Thus, if $C = A \cdot B$ (more generally, if the claimed output y equals $\Psi(x)$), then V can be convinced of that fact; if the output is not correct, P has no more than $\epsilon = \kappa^\rho$ probability of passing verification.

Savings. Moving from a Boolean to a non-concise arithmetic circuit saves, for a fixed m, an estimated four orders of magnitude in the number of circuit "wires" (the "wires" being $\{Z\}$) and thus eight orders of magnitude in the query size and the prover's work (which are quadratic in the number of wires). The dot product gates decrease these quantities by another factor of m^2 (since they reduce the number of "wires" from $m^3 + 2m^2$ to $2m^2$). In Figure 2, the arithmetic circuit column reflects these two reductions, the first being reflected in the elimination of the 10^9 factor and the second in the move from the m^6 to the m^4 term.

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 3—PEPPER's commitment protocol. V decommits queries $q_1, \ldots, q_\mu$ for the price of a single commitment query, $\mathsf{Enc}(r)$. This protocol strengthens one by Ishai et al. [45] and makes only minor changes to their notation and phrasing. The intent is that the μ queries be the PCP queries and that n be the size of the proof encoding ($s^2 + s$, until §3.6). The protocol assumes an additive homomorphic encryption scheme but can be modified to work with a multiplicative homomorphic scheme (such as ElGamal [32]); see Appendix E.

3.4 Strengthening linear commitment

The commitment protocol in the base scheme of Ishai et al. [45] relies on an additive homomorphic encryption operation.[4] If executed once, this operation is reasonably efficient (hundreds of microseconds; see Section 4.5); however, the number of times that the base scheme invokes it is proportional to at least the *square* of the input size *times* μ, the number of PCP queries (roughly 1000). For the example of $m \times m$ matrix multiplication with $m = 1000$, the base scheme would thus require at least $(1000)^4 \cdot 1000 \cdot 100$ μs: over 3000 years, and that's after the concise representation given by the previous refinement!

While we would be thrilled to eliminate homomorphic encryptions, we think that doing so is unlikely to work in this context. Instead, in this section we modify the commitment protocol to perform three orders of magnitude fewer encryptions; Appendix B proves the soundness of this modification by reducing its security to the semantic security of the homomorphic encryption scheme. Moreover, our reduction is more direct than in the base scheme, which translates into further cost reductions.

Details. In the base scheme [45], each PCP query by V (meaning each of the μ queries, as described in §2.1, §3.3, and Appendix A) requires V and P to run a separate instance of commitment. Thus, to check one computation Ψ, V homomorphically encrypts $\mu \approx 1000$ (see Figure 2) vectors, and P works over all of these ciphertexts. This factor of 1000 is an issue because the vectors are encrypted componentwise, and they have many components! (In the example above, they are elements of $\mathbb{F}^{s^2}$ or $\mathbb{F}^s$, where $s = 2 \cdot 10^6$.)

Figure 3 presents our modified commitment protocol. It homomorphically encrypts only one vector $r \in \mathbb{F}^{s^2+s}$, called the *commitment query*, with the encryption work amortizing over many queries $(q_1, \ldots, q_\mu)$. This new protocol leads to a more direct security reduction than in the base scheme. In their central reduction, Ishai et al. establish that commitment allows V to treat P as a *set* of non-colluding but possibly malicious oracles. In each repetition, their V must therefore issue extra queries (beyond the ℓ PCP queries) to ensure that the oracles match. With our commitment protocol, V can treat P as a *single* (possibly cheating) oracle and submit *only* the PCP queries. Stated more formally, Ishai et al. reduce linear PCP to linear MIP (multiprover interactive proof [14]) to the argument model, whereas we reduce linear PCP directly to the argument model. We prove the reduction in Appendix B.

Savings. This refinement reduces the homomorphic encryptions and other commitment-related work by three orders of magnitude, as depicted in Figure 2 by the elimination of the μ' term from the "commit" rows in the "new commit" column. As a second-order benefit, we save another factor of three (depicted in Figure 2 as a move from μ' to μ queries), as follows. The queries to establish the consistency of multiple oracles have error $(\ell - 1)/\ell = 13/14$. However, the soundness error of our base PCP protocol is $\kappa = 7/9$. Since $(13/14)^{\rho'} = (7/9)^{\rho}$ when $\rho' \approx 3\rho$, it takes roughly three times as many repetitions of the protocol to contend with this extra error. Finally, the direct reduction yields a qualitative benefit: it simplifies PEPPER.

[4] Note that we do not require fully homomorphic encryption [37]; as we discuss in Section 5, the costs of such schemes are still prohibitive.

3.5 Amortizing query costs through batching

Despite the optimizations so far, the verifier's work remains unacceptable. First, V must materialize a set of constraints that represent the computation, yet writing these down is as much work as executing the computation. Second, V must generate queries that are *larger* than the circuit. For example, for $m \times m$ matrix multiplication (§3.3), the commitment query has $4m^4 + 2m^2$ components (matching the number of components in the vector representation of the proof), in contrast to the $O(m^3)$ operations needed to execute the computation. A similar obstacle holds for many of the PCP queries. To amortize these costs, we modify the protocols to work over multiple computation instances and to verify computations in batch; we also rigorously justify these modifications. Note that the modifications do not reduce V's checking work, only V's cost to issue queries; however, this is acceptable since checking is fast.

Details. We assume that the computation Ψ (or equivalently, $\mathcal{C}$) is fixed; V and P will work over β instances of Ψ, with each instance having distinct input. We refer to β as the *batch size*. The prover P formulates β proof oracles (linear functions): $\pi_1, \ldots, \pi_\beta$. Note that the prover can stack these to create a linear function $\boldsymbol{\pi} : \mathbb{F}^{s^2+s} \to \mathbb{F}^\beta$ (one can visualize this as a matrix whose rows are $\pi_1, \ldots, \pi_\beta$).

To summarize the protocol, V now generates *one* set of commitment and PCP queries, and submits them to *all* of the oracles in the batch.[5] The prover now responds to queries q with $\boldsymbol{\pi}(q) \in \mathbb{F}^\beta$, instead of with $\pi(q) \in \mathbb{F}$. By way of comparison, the previous refinement (§3.4) encrypts a single r for a set of queries $q_1, \ldots, q_\mu$ to a proof π. This one issues a single r and a single set of queries $q_1, \ldots, q_\mu$ to multiple proofs $\pi_1, \ldots, \pi_\beta$.

Appendix C details the protocol and proof-sketches its soundness. We note that each of the β PCPs has the same soundness error (ϵ) as if it were queried individually. However, the errors for the β PCPs in a batch are correlated.

Savings. The most significant benefit is qualitative: without batching, V cannot gain from outsourcing, as the query costs are roughly the same as executing the computation. The quantitative benefit of this refinement is, as depicted in Figure 2, to reduce the per-instance cost of commitment and PCP queries by a factor of β.

3.6 Optimizing the PCP encoding

In some cases, we can address the obstacle of prover overhead by tailoring the PCP encoding. The key observations are that (1) the basic protocol *does not require a particular PCP encoding, as long as the encoding is a linear function*, and (2) some circuits permit a much more streamlined encoding. Specifically, we can tailor the prover's linear function so that it contains only terms that the prover had to compute anyway to return the result of the computation. (In essence, we are reducing the redundancy in the PCP.) The result for some computations, such as the examples given below, is protocols in which the prover's work is only a constant factor above simply executing the computation.

Details and examples. Under the linear PCPs in Sections 2.1 and 3.3, the prover computes $z \otimes z$, where z is a satisfying assignment to the circuit (or constraint set). However, for some computations V's circuit test (§2.1) interrogates only a subset of $z \otimes z$. Loosely speaking, P thus needs to compute only that subset to produce the proof oracle.

Matrix multiplication. As in Section 3.3, V again combines constraints that represent the computation, again picks random $\{v\}$ to be coefficients of those constraints, again obtains a polynomial (which we will write as $P'(Z)$), and again submits queries to the prover's oracle to get $P'(Z)$ evaluated and to test that the prover is holding an oracle of the correct form. There are two modifications to the approach presented in Section 3.3. First, in combining the constraints to construct $P'(Z)$, V does not include the degree-1 constraints (meaning $Z^a_{i,j} - A_{i,j} = 0$, etc.). Thus, V obtains $P'(Z) = \sum_{i,j} v^c_{i,j} \cdot (C_{i,j} - \sum_{k=1}^m Z^a_{i,k} \cdot Z^b_{k,j})$, which, using our notation from before, can be written $P'(Z) = \langle \gamma_2, Z \otimes Z \rangle + \gamma'_0$.

However, we will write $P'(Z)$ differently. In Sections 2.1 and 3.3, the form above motivated the definition of $\pi^{(2)}(q)$ as $\langle q, z \otimes z \rangle$, for $q \in \mathbb{F}^{s^2} = \mathbb{F}^{m^4}$. There, answering queries to $\pi^{(2)}$ required the prover to do work proportional to m^4. This is not only more than we would like to pay, given the $O(m^3)$ naive algorithm, but also more than we *have* to pay: $P'(Z)$ includes only a subset of $Z \otimes Z$. Specifically, observe that the degree-2 terms in $P'(Z)$ all have the form $Z^a_{i,k} \cdot Z^b_{k,j}$.

The above suggests the following notation: for $x, y \in \mathbb{F}^{m^2}$, $x \oslash y$ consists of $\{x_{i,k} \cdot y_{k,j}\}$ for all i, j, k. The cardinality of $x \oslash y$ is m^3. Thus, letting $Z^a = (Z^a_{1,1}, \ldots, Z^a_{m,m})$ and $Z^b = (Z^b_{1,1}, \ldots, Z^b_{m,m})$, we write:

$$P'(Z) = \left\langle \gamma'_2, Z^a \oslash Z^b \right\rangle + \gamma'_0,$$

where $\gamma'_2 \in \mathbb{F}^{m^3}$ has the same non-zero components as γ_2. As in the base protocols, V wants to evaluate $P'(z)$ since $P'(z)$ is a bellwhether that indicates whether $Z = z$ meets all constraints (here, z is the satisfying assignment that is purportedly encoded in the oracle π).

This brings us to the second modification: the PCP encoding is now $\pi = \pi^{(c)}(\cdot) \triangleq \left\langle \cdot, z^a \oslash z^b \right\rangle$, where $z^a = A$ and $z^b = B$. To evaluate $P'(z)$, then, V asks for $\pi^{(c)}(\gamma'_2)$, and V tests whether $\pi^{(c)}(\gamma'_2) + \gamma_0 = 0$. This is our new circuit test; observe that it is $O(m^3)$ for the prover to answer, and for V to formulate a query. Like the base PCP protocols, this one requires two other types of tests. The first is again a linearity test, to ensure that $\pi^{(c)}$ is (nearly) linear. The second is

[5] Early in their paper Ishai et al. briefly mention such an approach, but they do not specify it. Later in their paper [45, §6.1], in a more general context, they reuse PCP queries but not commitment queries.

again a consistency test, to ensure that $\pi^{(c)}$ indeed represents $A \oslash B$; this test is modified from the base scheme. The details of the queries and tests, and proofs of their correctness, are in Appendix D. Finally, note that $\pi^{(c)}$ is a linear function, so it fits into the commit protocol (§3.4).

Low-degree polynomial evaluation. Suppose that we wish to evaluate a degree-D polynomial A over the variables $X = \{X_1, X_2, \ldots, X_m\}$. If we let A_k denote the vector of coefficients for the degree-k terms in A, then the evaluation of A at $X = x$ can be written $A(x) = \langle A_D, \bigotimes_{i=1}^{D} x \rangle + \cdots + \langle A_2, x \otimes x \rangle + \langle A_1, x \rangle + A_0$. For $D = 2$, we can represent the evaluation of A with a single degree-2 concise gate (§3.3), where the "assignment" z is simply x itself; then, computing $\pi^{(2)}$ in the base PCP encoding requires computing $x \otimes x$. Thus, if A is dense (meaning that it has $\Omega(m^2)$ monomials), the PCP encoding adds little prover overhead over computing $A(x)$. However, for $D = 3$, a naive application of the base protocol would include only degree-2 and degree-1 constraints, requiring prover work and query size of $O(m^4)$—higher complexity than the $O(m^3)$ computation.

We can remove this overhead, if A is dense. We modify the encoding so that the prover holds not only $\pi^{(1)}$ and $\pi^{(2)}$ (as in §2.1 and §3.3) but also $\pi^{(3)} \colon \mathbb{F}^{m^3} \to \mathbb{F}$, where $\pi^{(3)}(\cdot) \triangleq \langle \cdot, z \otimes z \otimes z \rangle$. As above, the "assignment" z is simply x itself. At this point, V must add several PCP queries to establish the (near) linearity of $\pi^{(3)}$ and the consistency of $\pi^{(3)}$ with $(\pi^{(1)}, \pi^{(2)})$. For details of the queries and tests, see [58, §7.8], which describes a similar construction. An analogous approach applies for $D > 3$ when D is odd; if D is even, there are "circuits" that add no overhead.

Savings and discussion. For matrix multiplication and degree-3 polynomial evaluation, the tailored PCP encoding reduces the prover's work and the query size from $O(m^4)$ to $O(m^3)$. In our domain of interest, m is hundreds or thousands, so the savings in V's querying costs are two or three orders of magnitude. The same factor is saved in the prover's costs. In fact, if the relevant objects (the matrices or polynomials) are dense, the prover's work to create the PCP is the same complexity as executing the computation.

There is a caveat, however. Though the prover is linear, the constant is not ideal. With matrix multiplication, for example, the prover's work is roughly $(h + f \cdot \mu) \cdot m^3$ (as depicted in the prover's rows in Figure 2). Meanwhile, there is an approach that costs the prover $f \cdot m^3$, namely Freivalds's randomized algorithm for verifying matrix multiplication [58, §7.1]: this algorithm requires $O(m^2)$ time for the verifier and no overhead for the "prover". More generally, we know that many computations admit special-purpose verification protocols (see §5). However, we believe that the work of this subsection is interesting because it refines the general-purpose PCP encodings and so might work for a broad class of other circuits.

4 Implementation and evaluation

We assess our work by answering three high-level questions, which correspond to the three goals for PEPPER that we stated in the introduction: (1) Is PEPPER simple enough to be easily implementable and optimizable? (2) For what values of the batch size, β, and the input size does PEPPER's verifier gain from outsourcing? (3) At these values, how much does PEPPER cost the prover? We answer these questions in the context of three model computations: matrix multiplication, degree-2 polynomial evaluation, and degree-3 polynomial evaluation. We chose these because they are self-contained test cases and core primitives in many applications. The rest of this section describes our implementation, parameter choices, and further optimizations (§4.1); then addresses the first question (§4.2); and then answers the latter two with experiments and analysis (§4.3–§4.7).

4.1 Implementation and optimizations

Implementation. Although PEPPER requires that computations be expressed over a finite field $\mathbb{F}$, we want to simulate computations over the integers (which requires simulating signed values and unbounded magnitudes). To that end, we use the standard technique of working in $\mathbb{F}_p$ (the integers modulo p), where p is a prime far larger than the actual input domain of the computation (e.g., see [22, 70]). In particular, we require the inputs (e.g., the matrix entries) to be 32-bit quantities and choose p to be either a 128-bit prime (for matrix multiplication and degree-2 polynomial evaluation) or a 192-bit prime (for degree-3 polynomial evaluation). To implement field operations (as needed for verification and the computations themselves), PEPPER uses multiprecision arithmetic, as implemented by the GNU MP library.

The core of the verifier and prover are implemented in C++. The prover can run as a Web service, with its interfaces exposed via HTTP URLs: we run the C++ prover core and helper PHP utilities as persistent FastCGI processes [2] to which the Apache Web server delegates HTTP requests.

Currently, our implementation parallelizes only the prover's commit phase, using OpenMP [3]. Parallelizing the verifier and the rest of the prover, and distributing the prover over multiple machines, are works in progress.

We implemented both Paillier [59] and ElGamal [32] homomorphic encryption. Because the ElGamal homomorphism is multiplicative, not additive, ElGamal requires minor modifications to the commitment protocol (Figure 3); these are described in Appendix E. Where the algorithms call for random numbers, we use pseudorandom numbers, as generated by the ChaCha/8 stream cipher [19], specifically the `amd64-xmm6` variant in its default configuration.

Parameter choices. PEPPER has three security parameters, which drive its total error of $\epsilon' < \epsilon + \epsilon^c$ (as given by Theorem B.1). The first parameter is ρ, the number of PCP repe-

titions, which determines the PCP soundness error through the bound $\epsilon < \kappa^\rho$. (Per Appendix A, $\kappa = 7/9$.) We usually take $\rho = 70$, yielding $\epsilon < 2.3 \cdot 10^{-8}$ for the PCP error. The second two parameters determine ϵ^c, the commitment soundness error. They are (a) $|\mathbb{F}|$, the size of the field for the PCP and commitment machinery, and (b) the homomorphic encryption scheme's key size, which determines ϵ_S, the error from encryption. From Appendix B, we have $\epsilon^c < 2^{14} \cdot (1/|\mathbb{F}| + \epsilon_S)^{1/3}$. We take $|\mathbb{F}| \approx 2^{128}$ (we discussed field size above) and disregard ϵ_S (we discuss key size below), yielding $\epsilon^c < 2^{-28}$.

We performed all of our experiments with both encryption schemes but report results only from ElGamal because its performance is better by almost a factor of 10 (as others have also observed [35]). By default, we take the prime modulus to be 704 bits, though we experiment with other sizes. Of course, 704 bits is not industrial strength for usual cryptographic applications. However, homomorphic encryptions (Figure 3) are a bottleneck for PEPPER, so we make a considered choice to use this key size. We believe that this choice is acceptable in our context, as the key need hold up only for one run of the protocol. Our choice is based on the RSA factoring challenge.[6]

Further optimizations: HABANERO. We report results from both PEPPER, whose design we have treated rigorously, and HABANERO, a PEPPER variant that has two performance improvements. First, under HABANERO, *V precomputes* and installs on *P* the commitment query, $\mathsf{Enc}(pk, r)$. Under this modification, we use an industrial-strength key length—1280 bits—as we intend to amortize this offline step over a long time. The soundness of this modification is counterintuitive; since we don't have space for a proof, we call it a heuristic. For intuition, we note that this modification generalizes the way that a single *r* is used for the entire batch (Appendix C). The second modification is to lower ρ from 70 to 1. This allows a prover to cheat with probability as high as $\kappa = 7/9$. However, if *V* and *P* interact repeatedly, a cheating prover would be detected.

4.2 Is PEPPER simple to implement and apply?

PEPPER is built on PCP theory, which requires a lot of detail to explain. At the same time, that theory results in a system whose mechanics are surprisingly simple. The verifier doesn't do much more than generate and encrypt some pseudorandomness (Section 3.4), ask the prover to multiply some vectors with a matrix that it holds (Sections 3.3 and 3.6), and perform basic operations on the returned vectors (Appendices A and D). Likewise, the prover doesn't need to do much more than determine the assignment *z* to "gates" (which it does anyway as a result of executing the algorithm), and return dot products.

[6] In Nov. 2005 a 640-bit key was factored using "approximately 30 2.2GHz-Opteron-CPU years" and in Dec. 2009 a 768-bit key was factored using "almost 2000 2.2GHz-Opteron-CPU years" [4].

component	language	lines of code
Crypto library	C++	549
Core library	C++	323
Prover library	C++	174
Verifier library	C++	280
Data exchange module	PHP	43
Helper utilities	C++	388
Computation-independent total		**1727**
matrix multiplication	C++	394
degree-2 polynomial evaluation	C++	462
degree-3 polynomial evaluation	C++	608

FIGURE 4—PEPPER's components and their lines of code.

Our sense of PEPPER's simplicity is buttressed by three things. The first is its small code size. Figure 4 tabulates the lines of code in our implementation (using [71]); the computation-independent modules require only 1727 lines.

Second, applying PEPPER to new computations has proved easy, in part because the "circuit" model of the computation is not very far from an imperative statement of the computation in C++ (§3.3). In fact, once our software architecture was in place, a single graduate student was able to implement new computations in less than an hour per computation. While this is obviously longer than it would take to code the computation with no verification, it is not a ridiculous amount of time. Moreover, a compiler could perform this step automatically.

Third, optimizing the implementation has so far been relatively easy. For example, parallelizing the prover's commitment phase required only three compiler directives and one C++ statement; this simple change gives a substantial speedup on multi-core hardware (as quantified below).

4.3 Experimental evaluation: method and setup

In the sections ahead, we will determine the input sizes (which we loosely refer to as *m*) and batch sizes (β) at which *V* gains from outsourcing versus computing locally, and we will report CPU and network costs at these points. We first run experiments to illuminate the prototype's performance (§4.4); then use this performance plus microbenchmarks to calibrate our CPU cost model from Figure 2 (§4.5); and then use the model to estimate break-even points under PEPPER, under HABANERO, and under the refinements that lead to them (§4.6). We treat network costs in §4.7.

Our baseline is local computation, which we assume uses multiprecision arithmetic. While this assumption deprives the baseline of native operations and hence may be an overly optimistic comparison for PEPPER, we note that it is not totally ridiculous: multiprecision arithmetic is a standard approach for avoiding overflow and roundoff concerns when multiplying and adding many large numbers.

To measure CPU usage at prover and verifier, we use

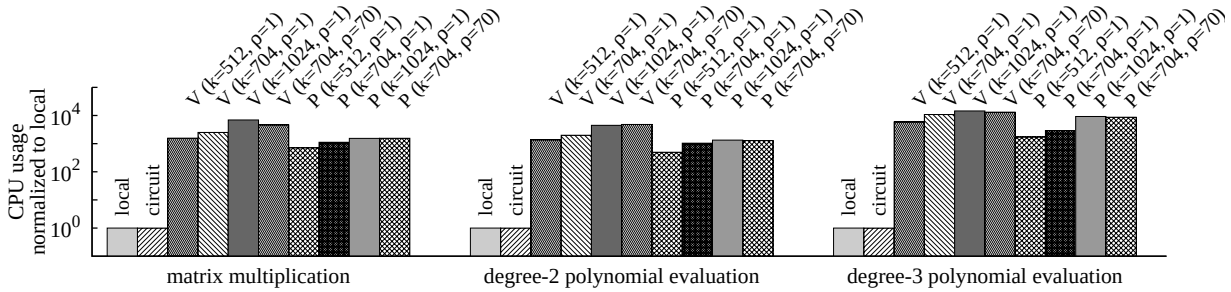

FIGURE 5—Measured CPU cost of one instance ($\beta = 1$) of PEPPER, compared to local execution, circuit execution, and under different security parameters (k is the key size, and ρ is the number of PCP repetitions), for verifier (V) and prover (P). For all computations, $m = 100$. Much of PEPPER's overhead is fixed-cost, so it amortizes under batching (see Figure 6).

`getrusage()`; to measure network costs, we record the amount of application-level data (not including network headers) exchanged between the prover and the verifier. We run our experiments on Utah Emulab's [72] d710s, which are 2.4 GHz 64-bit Intel Quad Core Xeon E5530 with 12 GB of memory; they run 64-bit Ubuntu 11.04 Linux.

4.4 What is PEPPER's end-to-end performance?

Beyond illustrating PEPPER's overhead, the experiments in this section are intended to assess the cost of the circuit representation; to indicate how the security parameters (§4.1) affect costs; and to illustrate how PEPPER's overhead can be mitigated by batching and parallelizing.

In the first set of experiments, our baseline is the CPU cost of local execution for matrix multiplication and polynomial evaluation (degrees 2 and 3), with $m = 100$ in all cases. We compare these baselines to executing the computation as a "circuit" but without verification. We also measure PEPPER, configured with smaller (512 bits) and larger (1024 bits) key sizes than our target (704 bits); these keys are not suitable for PEPPER (owing to insecurity and inefficiency, respectively), but they give a sense of what PEPPER pays for its cryptographic guarantees. Finally, we configure PEPPER to run with a single repetition ($\rho = 1$) and with multiple ones ($\rho = 70$). We do not depict HABANERO here.

We measure the CPU time at the verifier and prover consumed by PEPPER, end-to-end; this includes originating the queries, transferring them through the HTTP interface, handling responses, etc. The prover runs with one thread. For each configuration, we run 5 experiments, reporting average CPU time in seconds; standard deviations are within 5% of the means.

Figure 5 depicts the results. As shown in the figure, the circuit representation imposes no overhead. (Verification and proving are another matter—more on that soon.) The effect of PEPPER's security parameters is as follows. For matrix multiplication, PEPPER with 512-bit keys and 1024-bit keys costs, respectively, 1.7 times less and 1.9 times more than PEPPER with 704-bit keys (this holds for both prover and verifier); results for the other computations are similar. Interestingly, ρ has far less effect, since the cryptographic commitment amortizes over the ρ repetitions (§3.4).

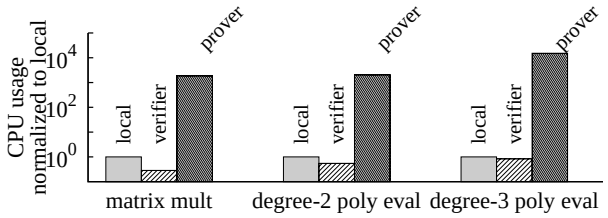

FIGURE 6—Measured CPU cost of the verifier and the prover in HABANERO, compared to local execution, for matrix multiplication ($m = 200$) under a batch size of $\beta = 100$, degree-2 polynomial evaluation ($m = 500$) under a batch size of $\beta = 200$, and degree-3 polynomial evaluation ($m = 200$) under a batch size of $\beta = 350$. At these batch sizes, the verifier gains from outsourcing.

Verification and proving clearly impose high overhead: for example, with degree-2 polynomial evaluation, the overhead over local computation is a factor of $3 \cdot 10^4$ for V and 10^4 for P. However, much of this cost is fixed and amortizes under batching (§3.5), as we demonstrate next.

Batching. We run HABANERO with multiple instances. We run 3 experiments for each computation, reporting mean CPU time (standard deviations are again within 5% of means). The prover runs with four threads; we report its cost as the sum of the CPU times taken by all four cores.

Figure 6 depicts the results. In these experiments, the verifier's cost is below the cost of local computation. (We will investigate break-even points systematically in Section 4.6.) The prover's cost is still high, but as we show next, parallelizing mitigates its latency.

Parallelizing. Here, we experiment with HABANERO, varying the number of threads (and hence cores) used. In each of the prover's two phases, we measure the wall clock time taken by the phase. We report the speedup with N cores as the ratio of total time taken (the sum of the two phases, over three trials) when one core is active to the time taken when N cores are active (again, over three trials).

	computation (Ψ)	batching (§3.5)	arith. circ. (§3.3)	new commit (§3.4)	new PCPs (§3.6)	HABANERO (§4.1)
β^* (break-even batch size)	matrix multiplication, $m = 400$	Never	$2.8 \cdot 10^{11}$	$7.1 \cdot 10^7$	$4.5 \cdot 10^4$	111
	matrix multiplication, $m = 1000$	Never	$2.0 \cdot 10^{11}$	$1.1 \cdot 10^8$	$2.7 \cdot 10^4$	111
	degree-2 polynomial eval, $m = 200$	Never	Never	$4.5 \cdot 10^4$	$4.5 \cdot 10^4$	180
	degree-2 polynomial eval, $m = 500$	Never	Never	$2.5 \cdot 10^4$	$2.5 \cdot 10^4$	118
	degree-3 polynomial eval, $m = 200$	Never	Never	$9.3 \cdot 10^6$	$4.6 \cdot 10^4$	221
	degree-3 polynomial eval, $m = 500$	$1.6 \cdot 10^{25}$	$6.0 \cdot 10^{10}$	$2.3 \cdot 10^7$	$4.6 \cdot 10^4$	220
V's computation time at β^*	matrix multiplication, $m = 400$	Never	9900 yr	2.5 yr	14 hr	2.1 min
	matrix multiplication, $m = 1000$	Never	$1.1 \cdot 10^5$ yr	61 yr	5.4 days	32 min
	degree-2 polynomial eval, $m = 200$	Never	Never	32 s	32 s	0.13 s
	degree-2 polynomial eval, $m = 500$	Never	Never	110 s	110 s	0.5 s
	degree-3 polynomial eval, $m = 200$	Never	Never	7.5 days	55 min	15 s
	degree-3 polynomial eval, $m = 500$	$5.6 \cdot 10^{17}$ yr	2100 yr	290 days	14 hr	4.0 min
P's computation time at β^*	matrix multiplication, $m = 400$	Never	$1.9 \cdot 10^{14}$ yr	$2.1 \cdot 10^7$ yr	8.1 yr	17 days
	matrix multiplication, $m = 1000$	Never	$5.4 \cdot 10^{15}$ yr	$1.3 \cdot 10^9$ yr	76 yr	260 days
	degree-2 polynomial eval, $m = 200$	Never	Never	45 hr	45 hr	24 min
	degree-2 polynomial eval, $m = 500$	Never	Never	6.2 days	6.2 days	1.6 hr
	degree-3 polynomial eval, $m = 200$	Never	Never	$7.1 \cdot 10^4$ yr	1.7 yr	5.8 days
	degree-3 polynomial eval, $m = 500$	$2.5 \cdot 10^{42}$ yr	$3.7 \cdot 10^{13}$ yr	$6.8 \cdot 10^6$ yr	27 yr	92 days

FIGURE 8—Estimated break-even points (β^*) and running times at those points under PEPPER's refinements, to two significant figures, obtained by (1) setting V's cost in Figure 2 equal to the cost of computing locally (also in Figure 2) and then (2) using the estimates of e, d, h, f, c (from Figure 7) to solve for β^*. Batching is required to make outsourcing profitable for V; without our refinements, the batch size is astronomical. Solving for m^* using more reasonable values of β also yields extremely large results. Our refinements bring these numbers into the realm of the conceivable. Both V and P can be parallelized to reduce latency; Section 4.4 describes the effect of parallelizing P.

key size	field size	e	d	h	f	c
704 bits (PEPPER)	128 bits	240 μs	95 μs	72 μs	18 ns	120 ns
704 bits (PEPPER)	192 bits	270 μs	95 μs	110 μs	45 ns	140 ns
1280 bits (HABANERO)	128 bits	660 μs	260 μs	200 μs	18 ns	120 ns
1280 bits (HABANERO)	192 bits	760 μs	260 μs	290 μs	45 ns	140 ns

FIGURE 7—Estimated parameters in our cost model (Figure 2), for the field and key sizes in our experiments.

computation (Ψ)	network costs
matrix multiplication, $m = 400$	17.1 GB
matrix multiplication, $m = 1000$	263 GB
degree-2 polynomial eval, $m = 200$	11.4 MB
degree-2 polynomial eval, $m = 500$	68.4 MB
degree-3 polynomial eval, $m = 200$	2.93 GB
degree-3 polynomial eval, $m = 500$	45.8 GB

FIGURE 9—Estimated network costs under HABANERO for the computations and break-even points (β^*) in Figure 8. These costs are significant but amortize under batching.

For $N = 2$, the speedups are 1.99 in all three cases (the ideal is 2). For $N = 4$, the prover's end-to-end latency reduces by a factor of 3.7 for matrix multiplication, a factor of 3.7 for degree-3 polynomial evaluation, and a factor of 3.1 for degree-2 polynomial evaluation (the ideal is 4). We hypothesize that the latter computation sees smaller speedup because the computation is far smaller, so fixed setup costs are comparatively larger. These results suggest that there is profit in distributing the prover over multiple machines.

4.5 Calibrating the cost models

We use a cost model for each of our computations; Figure 2 depicts the one for matrix multiplication.[7] When making predictions about HABANERO (versus PEPPER), we adjust the models to remove the cost of issuing commit queries and to set $\rho = 1$. Note that, for simplicity, the models include field multiplication (f) but not field addition. Nevertheless, for the purposes of estimating break-even points, the models are pessimistic for PEPPER. First, not counting additions underestimates the number of operations in the baseline computation. Second, the models overestimate PEPPER's operations. Specifically, the models more than compensate for the omitted field additions (which PEPPER performs in equal number to field multiplications) by double-counting field multiplications (which are more expensive).[8]

To estimate the parameters, we run a program that executes each operation at least 5000 times, using 704-bit and 1280-bit keys. Figure 7 shows the means (standard deviations are within 5% of the means). The two field sizes yield different parameter values because in the cryptographic operations, field entries appear in exponents.

How well do our models predict the end-to-end empirical results (§4.4)? The prover's costs are at most 8% above the model's prediction. The verifier's costs are at most 50% above what the model predicts. However, much of this difference can be explained by the cost of I/O operations (to files and the network); subtracting this cost from the verifier's measured performance yields an estimate that is at most 10% above the model's prediction. Finally, the base-

[7] Not depicted is an additive $2e$ term in the "Process PCP responses" row, from the modified check under ElGamal (see §4.1 and Appendix E).

[8] The reason for the double-counting is that we take ℓ, the number of PCP queries per repetition, as equal to 14, but in reality only *seven* of those 14 queries are submitted to the high-order proof (see Appendix A).

line's empirical costs range between 20% and 100% more than the model's predictions. We hypothesize that this divergence results from the model's omitting field additions (as discussed above) and from adverse caching behavior.

4.6 What are the break-even points?

Given the model's rough accuracy, it can systematically (if roughly) make predictions about break-even points—that is, about PEPPER's regime of applicability. We set V's costs in the (now parameterized) models equal to the cost of local computation, and solve for β^*, the batch size at which V gains from outsourcing under PEPPER. We do not account for the cost of serialization (this part of our implementation is unoptimized) so may slightly underestimate β^* (by a factor nearly equal to the overhead of serialization).

Figure 8 depicts the estimated break-even points. Under PEPPER (the column marked "new PCPs"), the numbers are sometimes reasonable for V. The latency from P is, ahem, somewhat larger. However, we would ask the reader to look leftward on this table to see what the numbers *used* to look like. Under HABANERO, we do not need to appeal to relativism quite as much: V gains from outsourcing at reasonable batch sizes, and while P's latency is surely high, the work can be parallelized and distributed (see §4.4). For example, if P used 128 cores, we would expect its latency to be on the order of hours or less for all but one of the rows.

4.7 What are the network costs?

The main network expense is from issuing queries (see Figure 3). To assess this cost, we estimate the sizes of the queries and responses, applying this simple model to the computations at the break-even points in Figure 8. (For HABANERO, we count only PCP queries; commitment queries are assumed to be pre-installed.) We separately validate by comparing to predictions with various experimental runs, finding that the model is within 3% of the measured traffic.

Figure 9 depicts the estimates, under HABANERO. These costs are obviously significant. However, compared to the prover's running time, the latency from network costs is not high. Moreover, PEPPER's network overhead amortizes, as the query cost stays fixed with increasing β. By contrast, some plausible baselines (for example, replicating the computation to multiple workers) have network overhead in proportion to β.

5 Related work

There has been a lot of work on verified computation, and the motivation of outsourcing to the cloud has given a new impetus to this area. In this section we try to situate our work in this landscape. We note that our work is concerned only with verifying computations, not hiding their content or input; thus, even though some of the works that we cite below achieve such hiding, our comparison will be silent about this property. Also, we do not summarize work that adopts a different threat model, such as multiple provers [10, 43] or assumptions about the difficulty of mimicking valid program counter values [67].

General-purpose approaches intended for practice. Perhaps the most basic technique, replication [6, 25, 42, 52, 56], relies on assumptions about the independence of the workers, in contrast to our goal of unconditional verification. Trusted computing [27, 62] assumes that the hardware or a hypervisor works properly. Some attestation work, though powerful, gives assurance that a remote application has desired properties but not that it is implemented correctly [26, 53, 63, 66]. Other attestation work requires a trusted base [5, 54]. Auditing and spot-checking [31, 42, 46, 50, 57] focus directly on the output, but unless the audit coverage is nearly perfect (which degenerates to executing the computation), a worker that alters a key bit (e.g., in an intermediate step) is unlikely to be detected, in contrast to the properties of PCPs (§2.1).

Special-purpose approaches from theory and practice. Recent work by Benabbas et al. [18] and Boneh and Freeman [22] outsources polynomial evaluation, which inspired our use of this computation (§3.6, §4). Ergun et al. [33] study efficient approximate property testing protocols for polynomials and related computations. Golle [40] outsources the inversion of one-way functions, and Karame et al. [47] apply this to detect cheating in a distributed sequential computation. Sion [65] and Thompson et al. [68] outsource database operations. Wang et al. outsource linear programming [69] and approximating systems of linear equations [70]. Atallah has protocols for linear algebra (see [10] and citations therein). Other work [34, 61] handles aggregate statistics computed from distributed sensors. While some of these works inspired our choice of particular computations, our focus is on general-purpose approaches.

Homomorphic encryption and secure multiparty protocols. Homomorphic encryption is a powerful building block, as it lets the prover compute over ciphertext. Groth uses homomorphic encryption to produce a zero-knowledge argument protocol [41]; unfortunately, a preliminary feasibility study that we performed indicated that this was not efficient enough for our purposes. Gentry's recent breakthrough construction of a fully homomorphic encryption scheme [37] has led to new protocols for verifiable non-interactive computation [28, 36] with input hiding. However, fully homomorphic encryption is completely infeasible on today's computers.

Closely related are secure multi-party protocols. In such protocols, parties compute an agreed-upon function of private data, revealing only the result [73]. In fact, the scheme of [36] builds such a protocol based on fully homomorphic encryption. Other general-purpose protocols for com-

putation on hidden data using homomorphic encryption include the work of [7] and the circuit application of the Boneh-Goh-Nissim cryptosystem [23]. However, in general, because of the input hiding requirements, the costs for the verifier are proportional to the size of the circuit; such protocols are not really suitable in our context. Moreover, while general-purpose compilers for such protocols exist [13, 44, 51], they are not yet practical, despite the striking improvements reported in [44].

PCPs, argument systems, and efficient interactive protocols. The potential of PCPs and efficient arguments to serve as a foundation for verified computations has long been known to theorists [8, 9, 11, 14, 20, 39, 48, 49, 55]. We borrow from this theory, guided by our goals. We turn to argument systems because advances in short (non-interactive) PCPs [15–17, 30, 60] seem too intricate to be implemented easily; e.g., they require recursive application of their own machinery. And among interactive approaches, we build upon the scheme of [45], which is simple, instead of upon an interactive protocol that is asymptotically more efficient but far more intricate [38]. We now review recent work that makes the opposite decision.

Following our position paper [64], Cormode, Mitzenmacher, and Thaler have a recent publication [29] with similar positioning. As with our system, they obtain a lot of mileage from the observation that arithmetic circuits and concise gates are compatible with the protocols and result in substantially more efficient encodings. Some of their focus is on specialized algorithms for linear problems in the streaming model. However, they also describe a general-purpose implementation, based on [38] instead of [45]. On the one hand, the protocol of [38] does not require public-key cryptography and has better asymptotic overhead than that of [45]. On the other hand, the protocol of [45] has better round complexity and does not require uniformity assumptions on the circuit families in question. A more detailed comparison using measurements on benchmark problems is work in progress.

6 Discussion, next steps, and outlook

This work began from the intuition that the protocol of Ishai et al. [45] might lead to a simple and potentially practical built system, and PEPPER appears to have fulfilled that promise: the verifier is relatively inexpensive, the prover is much better than we expected, the protocol is simple, and the tailored scheme is not so far from true practicality. A deployable system is plausibly within view.

Nonetheless, there is still much work to do to get PEPPER ready for prime time. Its protocols are still not cheap enough, particularly for the prover. Its model of computation is restrictive: we studied problems that admit concise expressions with arithmetic circuits performing multi-precision arithmetic (§3.3, §4.1), with a focus on problems that allow decomposition into identical parallel modules (§3.5) or permit tailored PCPs (§3.6). And while for arbitrary computations, one can theoretically apply PEPPER—by compiling the computation to a circuit with, say, the compiler developed as part of the Fairplay system [13, 44, 51]—our estimates indicate that this approach is unlikely to achieve acceptable performance and that a more specialized compiler is necessary.

The good news is that substantial further improvements seem possible. In the short term, there is much to be gained from engineering: a move to elliptic curve cryptography will allow us to improve the security parameters while decreasing overhead, and we plan to explore special-purpose hardware for modular exponentiation. We are working on compilers to automate the manual process described in Section 4.2; the goal is to output our non-tailored protocol for suitable computations and (when applicable) to generate the tailored PCPs that produce the most practical scheme (§3.6).

More broadly, the key challenge for this area is to expand the approach beyond computations that are naturally specified as arithmetic circuits over finite fields. For instance, a basic issue is handling floating point numerics efficiently. More generally, handling computations with interesting control flow is a central problem. Satisfactory answers to these questions will help realize the vision of making efficient arguments (and PCPs) truly practical.

Acknowledgments

Nikhil Panpalia created the parallel prover implementation. For helpful comments, we thank Lorenzo Alvisi, Dan Boneh, Allen Clement, Mike Freedman, Maxwell Krohn, Mike Lee, Josh Leners, Antonio Nicolosi, Bryan Parno, Raluca Popa, Victor Vu, Shabsi Walfish, Ed Wong, Hao Wu, and the anonymous reviewers. Emulab was helpful as always. The research was supported by AFOSR grant FA9550-10-1-0073 and by NSF grants 1055057 and 1040083.

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

References

[1] Berkeley Open Infrastructure for Network Computing (BOINC). `http://boinc.berkeley.edu`.
[2] FastCGI. `http://www.fastcgi.com/`.
[3] OpenMP. `http://openmp.org/`.
[4] The RSA Challenge Numbers. `http://www.rsa.com/rsalabs/node.asp?id=2093`.
[5] S. Alsouri, Ö. Dagdelen, and S. Katzenbeisser. Group-based attestation: Enhancing privacy and management in remote attestation. In *TRUST*, 2010.
[6] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer. SETI@home: An experiment in public-resource computing. *CACM*, 45(11):56–61, Nov. 2002.
[7] B. Applebaum, Y. Ishai, and E. Kushilevitz. From secrecy to soundness: efficient verification via secure computation. In *ICALP*, 2010.

[8] S. Arora, C. Lund, R. Motwani, M. Sudan, and M. Szegedy. Proof verification and the hardness of approximation problems. *J. of the ACM*, 45(3):501–555, May 1998.

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

[10] M. J. Atallah and K. B. Frikken. Securely outsourcing linear algebra computations. In *ASIACCS*, 2010.

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

[12] 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.

[13] A. Ben-David, N. Nisan, and B. Pinkas. FairplayMP: A system for secure multi-party computation. In *ACM CCS*, 2008.

[14] M. Ben-Or, S. Goldwasser, J. Kilian, and A. Wigderson. Multi-prover interactive proofs: how to remove intractability assumptions. In *STOC*, 1988.

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

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

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

[18] S. Benabbas, R. Gennaro, and Y. Vahlis. Verifiable delegation of computation over large datasets. In *CRYPTO*, 2011.

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

[20] M. Blum and S. Kannan. Designing programs that check their work. *J. of the ACM*, 42(1):269–291, 1995.

[21] 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.

[22] D. Boneh and D. M. Freeman. Homomorphic signatures for polynomial functions. In *EUROCRYPT*, 2011.

[23] D. Boneh, E. J. Goh, and K. Nissim. Evaluating 2-DNF formulas on ciphertexts. In *TCC*, 2005.

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

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

[26] L. Chen, R. Landfermann, H. Löhr, M. Rohe, A.-R. Sadeghi, C. Stüble, and H. Görtz. A protocol for property-based attestation. In *ACM Workshop on Scalable Trusted Computing*, 2006.

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

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

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

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

[31] W. Du, J. Jia, M. Mangal, and M. Murugesan. Uncheatable grid computing. In *IEEE Intl. Conf. on Dist. Computing Sys. (ICDCS)*, 2004.

[32] T. ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, 31:469–472, 1985.

[33] F. Ergun, R. Kumar, and R. Rubinfeld. Approximate checking of polynomials and functional equations. In *FOCS*, 1996.

[34] M. Garofalakis, J. M. Hellerstein, and P. Maniatis. Proof sketches: Verifiable in-network aggregation. In *ICDE*, 2007.

[35] S. Garriss, M. Kaminsky, M. J. Freedman, B. Karp, D. Mazières, and H. Yu. RE: Reliable Email. In *NSDI*, 2006.

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

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

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

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

[40] P. Golle and I. Mironov. Uncheatable distributed computations. In *RSA Conference*, 2001.

[41] J. Groth. Linear algebra with sub-linear zero-knowledge arguments. In *CRYPTO*, 2009.

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

[43] S. Hohenberger and A. Lysyanskaya. How to securely outsource crytographic computations. In *TCC*, 2005.

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

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

[46] S. Jha, S. Katzenbeisser, C. Schallhart, H. Veith, and S. Chenney. Semantic integrity in large-scale online simulations. *ACM Transactions on Internet Technology (TOIT)*, 10(1), Feb. 2010.

[47] G. O. Karame, M. Strasser, and S. Capkun. Secure remote execution of sequential computations. In *International Conference on Information and Communications Security (ICICS)*, 2009.

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

[49] J. Kilian. Improved efficient arguments (preliminary version). In *CRYPTO*, 1995.

[50] L. Kissner and D. Song. Verifying server computations. In *ACNS*, 2004.

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

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

[53] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig. TrustVisor: Efficient TCB reduction and attestation. In *IEEE Symposium on Security and Privacy*, 2010.

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

[55] S. Micali. Computationally sound proofs. *SIAM J. on Comp.*, 30(4):1253–1298, 2000.

[56] N. Michalakis, R. Soulé, and R. Grimm. Ensuring content integrity for untrusted peer-to-peer content distribution networks. In *NSDI*, 2007.

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

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

[59] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *EUROCRYPT*. Springer-Verlag, 1999.

[60] A. Polishchuk and D. A. Spielman. Nearly-linear size holographic proofs. In *STOC*, 1994.

[61] B. Przydatek, D. Song, and A. Perrig. SIA: Secure information aggregation in sensor networks. In *ACM Sensys*, 2003.

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

[63] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. van Doorn, and P. Khosla.

Pioneer: Verifying integrity and guaranteeing execution of code on legacy platforms. In *SOSP*, 2005.

[64] S. Setty, A. J. Blumberg, and M. Walfish. Toward practical and unconditional verification of remote computations. In *Proc. Workshop on Hot Topics in Operating Systems (HotOS)*, 2011.

[65] R. Sion. Query execution assurance for outsourced databases. In *VLDB*, 2005.

[66] F. Stumpf, A. Fuchs, S. Katzenbeisser, and C. Eckert. Improving the scalability of platform attestation. In *ACM Workshop on Scalable Trusted Computing*, 2008.

[67] S. Tang, A. R. Butt, Y. C. Hu, and S. P. Midkiff. Lightweight monitoring of the progress of remotely executing computations. In *International Workshop on Languages and Compilers for Parallel Computing*, 2005.

[68] B. Thompson, S. Haber, W. G. Horne, T. Sander, and D. Yao. Privacy-preserving computation and verification of aggregate queries on outsourced databases. In *PET*, 2009.

[69] C. Wang, K. Ren, and J. Wang. Secure and practical outsourcing of linear programming in cloud computing. In *INFOCOM*, Apr. 2011.

[70] C. Wang, K. Ren, J. Wang, and K. M. R. Urs. Harnessing the cloud for securely outsourcing large-scale systems of linear equations. In *IEEE Intl. Conf. on Dist. Computing Sys. (ICDCS)*, 2011.

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

[72] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *OSDI*, Dec. 2002.

[73] A. C.-C. Yao. How to generate and exchange secrets. In *FOCS*, 1986.

A A linear PCP

We state the queries, then the tests, then the statement of correctness of the PCP in [8, §5–6]. We use $\in_R$ to mean a uniformly random selection. The purpose of q_{10}, q_{12}, q_{14} below is *self-correction*; see [8, §5] or [58, §7.8.3] for details.

- 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 queries $q_1, \ldots, q_{14}$ 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 so, `accept`.

The $\gamma_0, \gamma_1, \gamma_2$ above are described in §2.1. The following lemmas rephrase Lemmas 6.2 and 6.3 from [8]:

Lemma A.1 (Completeness [8]). Assume V is given a satisfiable circuit $\mathcal{C}$. If π is constructed as in Section 2.1, and if V proceeds as above, then $\Pr\{V \text{ accepts } \mathcal{C}\} = 1$. The probability is over V's random choices.

Lemma A.2 (Soundness [8]). There exists a constant $\kappa < 1$ such that if some proof oracle π passes all of the tests above on $\mathcal{C}$ with probability $> \kappa$, then $\mathcal{C}$ is satisfiable.

Applying the analysis in [8], we can take $\kappa > \max\{7/9,\ 4\delta + 2/|\mathbb{F}|,\ 4\delta + 1/|\mathbb{F}|\}$ for some δ such that $3\delta - 6\delta^2 > 2/9$. This ensures soundness for all three tests. Here, δ relates to linearity testing [21]; to justify the constraint on δ and its connection to the $7/9$ floor on κ, see [12] and citations therein. Taking $\delta = \frac{1}{10}$, we can take $\kappa = 7/9 + \text{neg}$, where neg can be ignored; thus, for convenience, we assume $\kappa = 7/9$. Applying the lemma, we have that if the protocol is run $\rho = 70$ times and $\mathcal{C}$ is not satisfiable, V wrongly accepts with probability $\epsilon < \kappa^\rho < 2.3 \cdot 10^{-8}$.

B Reducing linear PCPs to arguments

This section first reduces PCPs to arguments directly, using Commit+Multidecommit (Figure 3); this will formalize Figure 1. The soundness of the reduction relies on Commit+Multidecommit actually binding the prover after the commit phase. Thus, the second (bulkier) part of the section defines a new and strengthened commitment protocol (Defn. B.1), proves that Commit+Multidecommit implements this protocol (Lemma B.1), and proves that any such protocol binds the prover in the way required by the reduction (Lemma B.2).

The reduction composes a PCP (such as the one in Appendix A) with Commit+Multidecommit. The protocol, theorem, and proof immediately below are almost entirely a syntactic substitution in [45, §4], replacing "MIP" with "PCP".

Given a linear PCP with soundness ϵ, the following is an argument system (P', V'). Assume that (P', V') get a Boolean circuit $\mathcal{C}$ and that P' has a satisfying assignment.

1. P' and V' run Commit+Multidecommit's commit phase, causing P' to commit to a function, π.
2. V' runs the PCP verifier V on $\mathcal{C}$ to obtain $\mu = \ell \cdot \rho$ queries $q_1, \ldots, q_\mu$.
3. P' and V' run the decommit phase of Commit+Multidecommit. V' uses $q_1, \ldots, q_\mu$ as the queries. V' either rejects the decommitted output, or it treats the output as $\pi(q_1), \ldots, \pi(q_\mu)$. To these outputs, V' applies the PCP verifier V. V' outputs `accept` or `reject` depending on what V would do.

Theorem B.1. Suppose (P, V) is a linear PCP with soundness ϵ. Then (P', V') described above is an argument protocol with soundness $\epsilon' \leq \epsilon + \epsilon^c$. ($\epsilon^c$ represents the error from the commitment protocol and will be filled in by Lemma B.2.)

Proof. Completeness follows from the PCP and the definition of Commit+Multidecommit. For soundness, Lemma B.2 below states that at the end of step 1 above, there is an extractor function that defines a single (possibly incorrect) or-

acle function $\tilde{\pi}$ such that, if V' didn't reject during decommit, then with all but probability ϵ^c, the answers that V' gets in step 3 are $\tilde{\pi}(q_1), \ldots, \tilde{\pi}(q_\mu)$. But (P, V) has soundness ϵ, so the probability that V' accepts a non-satisfiable $\mathcal{C}$ is bounded by $\epsilon + \epsilon^c$. □

We now strengthen the commitment primitive in Ishai et al. [45], borrowing their framework. We define a protocol: *commitment to a function with multiple decommitments* (CFMD). The sender is assumed to have a linear function, π, given by a vector, $w \in \mathbb{F}^n$; that is, $\pi(q) = \langle w, q \rangle$. In our context, w is normally $(z, z \otimes z)$. The receiver has μ queries, $q_1, q_2, \ldots, q_\mu \in \mathbb{F}^n$. For each query q_i, the receiver expects $\pi(q_i) = \langle w, q_i \rangle \in \mathbb{F}$.

Definition B.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* 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 probability of S^* winning 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.

Informally, binding means that after the sender commits, it is very likely bound to a function from queries to answers.

Lemma B.1. Commit+Multidecommit (Figure 3, Section 3.4) is a CFMD protocol with $\epsilon_B = 1/|\mathbb{F}| + \epsilon_S$, where ϵ_S comes from the semantic security of the homomorphic encryption scheme.

Proof. Correctness: for an honest sender, $b = \pi(t) = \pi(r) + \sum_{i=1}^{\mu} \pi(\alpha_i \cdot q_i) = s + \sum_{i=1}^{\mu} \alpha_i \cdot \pi(q_i) = s + \sum_{i=1}^{\mu} \alpha_i \cdot a_i$, which implies that $b = s + \sum_{i=1}^{\mu} \alpha_i \cdot a_i$, and so verification passes, with the receiver outputting $\pi(q_1), \ldots, \pi(q_\mu)$.

To show ϵ_B-binding, 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. Assume that Commit+Multidecommit does not meet ϵ_B-binding. Then there exists an efficient cheating sender S^* and an environment $\mathcal{E}$ producing $Q, \hat{Q}, i, j$ such that $q \triangleq q_i = \hat{q}_j$ and S^* can make R output $a_i \neq \hat{a}_j$ with probability $> \epsilon_B$.

We will construct an algorithm $\mathcal{A}$ that differentiates between $\alpha, \alpha' \in_R \mathbb{F}$ with probability more than $1/|\mathbb{F}| + \epsilon_S$ when given as input the following: a public key, pk; the encryption, $\mathsf{Enc}(pk, r)$, of r for a random vector $r \in \mathbb{F}^n$; $r + \alpha q$; and $r + \alpha' q$. This will contradict the semantic security of the encryption scheme. $\mathcal{A}$ has Q, q, i, j hard-wired (because $\mathcal{A}$ is working under environment $\mathcal{E}$) and works as follows:

(a) $\mathcal{A}$ gives S^* the input $(pk, \mathsf{Enc}(pk, r))$; $\mathcal{A}$ gets back e from S^* and ignores it.

(b) $\mathcal{A}$ randomly chooses $\alpha_1, \ldots, \alpha_{i-1}, \alpha_{i+1}, \ldots, \alpha_\mu$. It also randomly chooses $\hat{\alpha}_1, \ldots, \hat{\alpha}_{j-1}, \hat{\alpha}_{j+1}, \ldots, \hat{\alpha}_\mu$.

(c) $\mathcal{A}$ now leverages Q, q, i, j. $\mathcal{A}$ was given $r + \alpha q$, so it can construct $r + \alpha q + \sum_{k \in [\mu] \setminus i} \alpha_k q_k = r + Q \cdot \boldsymbol{\alpha}$, where $\boldsymbol{\alpha} = (\alpha_1, \ldots, \alpha_{i-1}, \alpha, \alpha_{i+1}, \ldots, \alpha_\mu)$. $\mathcal{A}$ gives S^* the input $(Q, r + Q \cdot \boldsymbol{\alpha})$; that is, $\mathcal{A}$ invokes S^* in the decommit phase. $\mathcal{A}$ gets back (A, b).

(d) Likewise, $\mathcal{A}$ constructs $r + \alpha' q + \sum_{k \in [\mu] \setminus j} \hat{\alpha}_k \hat{q}_k = r + \hat{Q} \cdot \hat{\boldsymbol{\alpha}}$, where $\hat{\boldsymbol{\alpha}} = (\hat{\alpha}_1, \ldots, \hat{\alpha}_{j-1}, \alpha', \hat{\alpha}_{j+1}, \ldots, \hat{\alpha}_\mu)$. $\mathcal{A}$ gives S^* $(\hat{Q}, r + \hat{Q} \cdot \hat{\boldsymbol{\alpha}})$, invoking S^* in the decommit phase again. $\mathcal{A}$ gets back $(\hat{A}, \hat{b})$.

When S^* wins (which it does with probability greater than $\epsilon_B = 1/|\mathbb{F}| + \epsilon_S$), $b = s + \boldsymbol{\alpha} \cdot A$ and $\hat{b} = s + \hat{\boldsymbol{\alpha}} \cdot \hat{A}$, but $a_i \neq \hat{a}_j$ (here, $\cdot$ represents the dot product). Now we will get two linear equations in two unknowns. The first is $\hat{b} - b = \hat{\boldsymbol{\alpha}} \cdot \hat{A} - \boldsymbol{\alpha} \cdot A$, which can be rewritten as: $\boxed{K_1 = \alpha' \hat{a}_j - \alpha a_i}$, where $\mathcal{A}$ can derive $K_1 = \hat{b} - b - \sum_{k \neq j} \hat{\alpha}_k \hat{a}_k + \sum_{k \neq i} \alpha_k a_k$. Now, let $t = r + Q \cdot \boldsymbol{\alpha}$ and let $\hat{t} = r + \hat{Q} \cdot \hat{\boldsymbol{\alpha}}$. To get the second equation, we start with $\hat{t} - t = \sum_{k \in [\mu] \setminus j} \hat{\alpha}_k \hat{q}_k - \sum_{k \in [\mu] \setminus i} \alpha_k q_k + \alpha' q - \alpha q$. This equation concerns a vector. We choose an index ℓ in the vector where q is not zero (if q is zero everywhere, then r is revealed). At that index, we have the following equation in scalars: $\boxed{K_2 = \alpha' - \alpha}$, where $\mathcal{A}$ can derive $K_2 = (\hat{t}^{(\ell)} - t^{(\ell)} - \sum_{k \neq j} \hat{\alpha}_k \hat{q}_k^{(\ell)} + \sum_{k \neq i} \alpha_k q_k^{(\ell)}) / q^{(\ell)}$. Now $\mathcal{A}$ can solve for α (since the contrary hypothesis gave $a_i \neq \hat{a}_j$). □

We now prove that after the commit phase, the prover is effectively bound to a *single* function. Our articulation again follows [45], specifically their Lemmas 3.2 and 3.6.

Lemma B.2 (Existence of an extractor function). 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. 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.

Proof. We will reuse the ideas in the proof of Lemma 3.2 in [45], but we must also ensure that q yields the same answer independent of its position and the other queries in the tuple. We begin with a definition: let $\mathsf{Ext}(v, q, i, \vec{q}) \triangleq \text{argmax}_a A_v(q, i, \vec{q}, a)$, where $A_v(q, i, \vec{q}, a)$ equals, in view $v = (v_{S^*}, v_R)$, the probability over the randomness of the decommit phase that R's ith output is a when the query tuple is $\vec{q}$; note that q is the ith component of $\vec{q}$ and is included in $\mathsf{Ext}(\cdot)$ and $A_v(\cdot)$ for convenience. In other words, $\mathsf{Ext}(\cdot)$ is the most likely a_i value to be output by R, if the full tuple of queries is $\vec{q}$ and if q appears in the ith position. Note that, after the commit phase, $\mathsf{Ext}(\cdot)$ is given deterministically.

Claim B.3. Define $\epsilon_2 = (\sqrt[3]{9/2} + 1) \cdot \sqrt[3]{\epsilon_B}$. For all $\mathcal{E}$ producing $(q, i, j, \vec{q}_1, \vec{q}_2)$, where $\vec{q}_1$'s ith component is q and $\vec{q}_2$'s jth component is also q, we have the following with probability $> 1 - \epsilon_2$ over the commit phase: either $\mathsf{Ext}(v, q, i, \vec{q}_1) = \mathsf{Ext}(v, q, j, \vec{q}_2)$, or else the probability over the decommit phase of outputting $\perp$ is greater than $1 - \epsilon_2$.

Proof. Assume otherwise. Then there is an environment $\mathcal{E}$ producing $(q, i, j, \vec{q}_1, \vec{q}_2)$ such that with probability $> \epsilon_2$ over the commit phase, $\mathsf{Ext}(v, q, i, \vec{q}_1) \neq \mathsf{Ext}(v, q, j, \vec{q}_2)$ and with probability $> \epsilon_2$ over the decommit phase, R outputs something other than $\perp$. Define $\epsilon_1 = \sqrt[3]{\epsilon_B 9/2}$. We will show below that with probability $> 1 - \epsilon_1$ over the commit phase, $\sum_{a \in \mathbb{F} \setminus \mathsf{Ext}(v,q,i,\vec{q}_1)} A_v(q, i, \vec{q}_1, a)$ is $< \epsilon_1$. Thus, with probability $> \epsilon_2 - \epsilon_1$ over the commit phase, we have (1) the probability over the decommit phase is $> \epsilon_2 - \epsilon_1$ that R outputs $\mathsf{Ext}(v, q, i, \vec{q}_1)$ in the ith position (since R outputs *something* other than $\perp$ with probability $> \epsilon_2$, yet the probability of outputting anything *other* than $\mathsf{Ext}(v, q, i, \vec{q}_1)$ is $< \epsilon_1$); and (2) likewise, the probability of outputting $\mathsf{Ext}(v, q, j, \vec{q}_2)$ in the jth position is $> \epsilon_2 - \epsilon_1$. If we now take $Q = \vec{q}_1$ and $\hat{Q} = \vec{q}_2$, we have a contradiction to the definition of CFMD because with probability $> (\epsilon_2 - \epsilon_1)^3 = \epsilon_B$ over all three phases, $a_i \neq \hat{a}_j$, which generates a contradiction because the definition of ϵ_B-Binding says that this was supposed to happen with probability $< \epsilon_B$.

We must now show that, with probability $> 1 - \epsilon_1$ over the commit phase, $\sum_{a \in \mathbb{F} \setminus \mathsf{Ext}(v,q,i,\vec{q}_1)} A_v(q, i, \vec{q}_1, a)$ is $< \epsilon_1$. If not, then with probability $> \epsilon_1$ over the commit phase, $\sum_{a \in \mathbb{F} \setminus \mathsf{Ext}(v,q,i,\vec{q}_1)} A_v(q, i, \vec{q}_1, a) \geq \epsilon_1$. Now, following Lemma 3.2 in [45], we can partition $\mathbb{F}$ into two sets T_1, T_2 such that $\sum_{a \in T_1} A_v(q, i, \vec{q}_1, a)$ and $\sum_{a \in T_2} A_v(q, i, \vec{q}_1, a)$ are each greater than $\epsilon_1/3$. (There are two cases; consider $a^* = \mathsf{Ext}(v, q, i, \vec{q}_1)$. Either $A_v(q, i, \vec{q}_1, a^*)$ is greater than $\epsilon_1/3$, or it is not. If so, then the partition is $(a^*, \mathbb{F} \setminus a^*)$. If not, then there is still a partition because the sum of the $A_v(\cdot)$ is greater than $\epsilon_1/3$.) This implies that, in the binding experiment, R outputs values from the two partitions with probability $> (2/9) \cdot (\epsilon_1)^3 = \epsilon_B$ over all three phases, which contradicts the definition of a CFMD protocol. □

Now define $\mathsf{Ext}(v, q) = \mathsf{Ext}(v, q, i^*, \vec{q}^*)$, where i^* and $\vec{q}^*$ are designated (any choice with q in the i^* position of $\vec{q}^*$ will do). The next claim says that the response to q is independent of its position and the other queries.

Claim B.4. Let $\epsilon_3 = \epsilon_2 + \epsilon_1$. For all $\vec{q}, i$, where q is in the ith position of $\vec{q}$, we have that with probability $> 1 - \epsilon_3$ over the commit phase, either R's ith output is $\mathsf{Ext}(v, q)$, or else the probability over the decommit phase of outputting $\perp$ is $> 1 - \epsilon_3$.

Proof. Assume otherwise. Then there are $\vec{q}, i$ such that with probability $> \epsilon_3$ over the commit phase, the probability of outputting something (non-$\perp$) besides $\mathsf{Ext}(v, q, i^*, \vec{q}^*)$ is $> \epsilon_3$. But with probability $> 1 - \epsilon_1$ over the commit phase, $\sum_{a \in \mathbb{F} \setminus \mathsf{Ext}(v,q,i,\vec{q})} A_v(q, i, \vec{q}, a) < \epsilon_1$ (by the partitioning argument given in the proof of Claim B.3). Thus, with probability $> \epsilon_3 - \epsilon_1$ over the commit phase, the probability of outputting something (non-$\perp$) besides $\mathsf{Ext}(v, q, i^*, \vec{q}^*)$ is $> \epsilon_3$ and $\sum_{a \in \mathbb{F} \setminus \mathsf{Ext}(v,q,i,\vec{q})} A_v(q, i, \vec{q}, a) < \epsilon_1$. Thus, with probability $> \epsilon_3 - \epsilon_1 = \epsilon_2$ over the commit phase, $\mathsf{Ext}(v, q, i^*, \vec{q}^*) \neq \mathsf{Ext}(v, q, i, \vec{q})$ and R has $> \epsilon_3 > \epsilon_2$ probability of outputting something other than $\perp$. This contradicts Claim B.3. □

To complete the proof of the lemma, define $\tilde{f}_v(q) \triangleq \mathsf{Ext}(v, q)$. Consider the probability ϵ^c, over both phases, that the output $A \neq (\tilde{f}_v(q_1), \ldots, \tilde{f}_v(q_\mu))$, i.e., that at least one of $a_i \neq \tilde{f}_v(q_i)$. By the union bound, $\epsilon^c < \sum_{i=1}^{\mu} \Pr\{a_i \neq \tilde{f}_v(q_i)\}$. By Claim B.4, $\Pr\{a_i \neq \tilde{f}_v(q_i)\} < \epsilon_3 + \epsilon_3$, since this bounds the probability that either of the two phases goes badly. Thus, $\epsilon^c < \mu \cdot 2 \cdot \epsilon_3$, as was to be shown. □

To compute ϵ^c, we ignore ϵ_S (the homomorphic encryption error); i.e., we set $\epsilon_B = 1/|\mathbb{F}|$. We then get $\epsilon^c < 2000 \cdot (7\sqrt[3]{\epsilon_B}) < 2^{14} \cdot \sqrt[3]{1/|\mathbb{F}|}$. For $|\mathbb{F}| = 2^{128}$, $\epsilon^c < 2^{-28}$.

C Batching

Under batching, V submits the same queries to β different proofs. Below, we sketch the mechanics and then proofs of correctness. First, we modify Commit+Multidecommit (Figure 3) to obtain a new protocol, called BatchedCommit+Multidecommit. The changes are as follows:

- P is regarded as holding a linear function $\boldsymbol{\pi} : \mathbb{F}^n \to \mathbb{F}^\beta$, so $\boldsymbol{\pi}(q) = (\pi_1(q), \ldots, \pi_\beta(q))$. One can visualize $\boldsymbol{\pi}$ as an $\beta \times n$ matrix, each of whose rows is an oracle, π_i. Thus, P returns vectors instead of scalars.

- Commit phase, steps 2 and 3: instead of receiving from P the scalar $\mathsf{Enc}(pk, \pi(r))$, V in fact receives a vector $\mathbf{e} = (\mathsf{Enc}(pk, \pi_1(r)), \ldots, \mathsf{Enc}(pk, \pi_\beta(r)))$ and decrypts to get $\mathbf{s} = (\pi_1(r), \ldots, \pi_\beta(r))$.
- Decommit phase, steps 5 and 6: P returns $\mathbf{a}_1, \ldots, \mathbf{a}_\mu, \mathbf{b}$, where $\mathbf{a}_i$ is supposed to equal $(\pi_1(q_i), \ldots, \pi_\beta(q_i))$ and $\mathbf{b}$ is supposed to equal $(\pi_1(t), \ldots, \pi_\beta(t))$; V checks that $\mathbf{b} = \mathbf{s} + \alpha_1\mathbf{a}_1 + \cdots + \alpha_\mu\mathbf{a}_\mu$.

Second, we modify the compilation in Appendix B as follows. P' creates β linear proof oracles, and V' and P' run BatchedCommit+Multidecommit, causing P' to commit to a linear function $\boldsymbol{\pi}\colon \mathbb{F}^n \to \mathbb{F}^\beta$. V' then submits the μ PCP queries and receives vectors $\boldsymbol{\pi}(q_1), \ldots, \boldsymbol{\pi}(q_\mu)$ in response. Then V' runs the PCP verifier on each instance separately (e.g., for the kth instance, V' looks at the kth component of each of $\boldsymbol{\pi}(q_1), \ldots, \boldsymbol{\pi}(q_\mu)$). V' thus returns a vector of β `accept` or `reject` outputs. To argue correctness, we use a theorem analogous to Theorem B.1:

Theorem C.1. Under (P', V') as described above, each of the β instances is an argument protocol with soundness $\epsilon' \leq \epsilon + \epsilon^c$. ($\epsilon^c$ is defined in Appendix B.)

Proof. (Sketch.) Nearly the same as for Theorem B.1. We need an analog of Lemma B.2, described below. □

This theorem says that if any of the β instances tries to encode a "proof" for an incorrect output, the probability that V outputs `accept` for that instance is bounded by ϵ'. This makes intuitive sense because if we fix a given instance, the probabilities should be unaffected by "extra" instances.

To formalize this intuition, we need BatchedCommit+Multidecommit to yield an extractor function for each of the β instances. To get there, we define a general protocol: *batch-CFMD*. This protocol has a binding property modified from the one in Definition B.1. In the new one, R gives stacked output $\mathbf{A} = (\mathbf{a}_1, \ldots, \mathbf{a}_\mu)$ and $\hat{\mathbf{A}} = (\hat{\mathbf{a}}_1, \ldots, \hat{\mathbf{a}}_\mu)$. The entries of $\mathbf{A}$ are denoted a_i^k; that is, $\mathbf{a}_i = (a_i^1, \ldots, a_i^\beta)$. We allow $a_i^k \in \{\mathbb{F} \cup \perp\}$ but require that $(a_1^k, \ldots, a_\mu^k) \in \mathbb{F}^\mu$ or $a_i^k = \perp$ for all $i \in [\mu]$. We now say that S^* wins if for some i, j, k, we have $q_i = \hat{q}_j$ and $a_i^k, \hat{a}_j^k \in \mathbb{F}$ but $a_i^k \neq \hat{a}_j^k$. We again say that the protocol meets ϵ_B-Binding if for all $\mathcal{E}$ and efficient S^*, S^* has less than ϵ_B probability of winning.

We can show that BatchedCommit+Multidecommit is a batch-CFMD protocol by rerunning Lemma B.1, nearly unmodified. To complete the argument, we can establish an analog of Lemma B.2. The analog replaces a single extractor function $\tilde{f}_v$ with β functions $\tilde{f}_v^1, \ldots, \tilde{f}_v^\beta$, one per instance. The analog says that, viewing each instance k separately, we have with probability $> 1 - \epsilon^c$ that either R's output for that instance is $\perp$ or else $a_i^k = \tilde{f}_v^k(q_i)$ for all $i \in [\mu]$. The proof is nearly the same as for Lemma B.2; the main difference is that $\mathsf{Ext}(\cdot)$ and $A_v(\cdot)$ receive a per-instance parameter k.

D Tailored PCP for matrix multiplication

This section expands on the tailored PCP construction for matrix multiplication, introduced in Section 3.6. Recall that V wants to verify that C equals $A \cdot B$, where $A, B, C \in \mathbb{F}^{m^2}$. Further recall that V constructs a polynomial $P'(Z)$ from the constraints in Section 3.3, but V drops the degree-1 constraints. Loosely speaking, V is justified in doing so because (a) V knows what $z_{i,j}^a$ and $z_{i,j}^b$ should be, namely the values of A and B; and (b) the resulting circuit test is still likely to catch an incorrect output C. Further recall that the modified PCP encoding is $\pi = \pi^{(c)}(\cdot) \triangleq \langle \cdot, A \oslash B \rangle$, where $x \oslash y = \{x_{i,k} \cdot y_{k,j}\}$ for all i, j, k.

After stating the queries and tests below, we give proofs of correctness and estimate costs.

- Generate linearity queries: Select $q_1, q_2 \in_R \mathbb{F}^{m^3}$. Take $q_3 \leftarrow q_1 + q_2$.
- Generate quadratic correction queries: select $q_4, q_5 \in_R \mathbb{F}^{m^2}$ and $q_7 \in_R \mathbb{F}^{m^3}$. Take $q_6 \leftarrow (q_4 \oslash q_5 + q_7)$
- Generate circuit test queries: select $q_9 \in_R \mathbb{F}^{m^3}$ and $q_8 \leftarrow \gamma_2' + q_9$. ($\gamma_2'$ is defined in Section 3.6.)
- Issue queries. Send q_1, q_2, q_3 and $q_6, \ldots, q_9$ to π, getting back $\pi(q_1), \pi(q_2), \pi(q_3), \pi(q_6), \pi(q_7), \pi(q_8), \pi(q_9)$.
- Linearity test: Check that $\pi(q_1) + \pi(q_2) = \pi(q_3)$. If not, `reject`.
- Quadratic correction test: Check that $\pi(q_6) - \pi(q_7) = \sum_{k=1}^m \langle A_{*,k}, q_{4_{*,k}} \rangle \cdot \langle B_{k,*}, q_{5_{k,*}} \rangle$, where $M_{*,k} \in \mathbb{F}^m$ denotes the kth column of matrix M, and $M_{k,*} \in \mathbb{F}^m$ denotes the kth row of matrix M. If not, `reject`.
- Circuit test: Check that $\pi(q_8) - \pi(q_9) = -\gamma_0$. ($\gamma_0'$ is defined in Section 3.6.) If so, `accept`.

Lemma D.1 (Completeness). If V is given $C = A \cdot B$, if $\pi = \pi^{(c)}$ is constructed as above, and if V proceeds as above, then $\Pr\{V \text{ accepts}\} = 1$. The probability is over V's random choices.

Proof. If $\pi = \pi^{(c)}$ is constructed as above, then the linearity test passes because $\pi^{(c)}$ is a linear function. The quadratic correction test also passes because $\pi^{(c)}(q_6) - \pi^{(c)}(q_7) = \pi^{(c)}(q_4 \oslash q_5) = \langle q_4 \oslash q_5, A \oslash B \rangle$, which indeed equals $\sum_{k=1}^m \langle A_{*,k}, q_{4_{*,k}} \rangle \cdot \langle B_{k,*}, q_{5_{k,*}} \rangle$. Finally, the circuit test passes because $\pi^{(c)}(q_8) - \pi^{(c)}(q_9) = \pi^{(c)}(\gamma_2') = \langle \gamma_2', A \oslash B \rangle$, and γ_2' and γ_0' are constructed so that if $C = A \cdot B$, then $\langle \gamma_2', A \oslash B \rangle + \gamma_0' = 0$. □

Lemma D.2 (Soundness). There exists a constant $\kappa < 1$ such that if some proof oracle π passes all of the tests above with probability $> \kappa$, then $C = A \cdot B$.

Proof. Choose δ such that $3\delta - 6\delta^2 > 2/9$, and take $\kappa > \max\{\frac{7}{9}, 2\delta + 2/|\mathbb{F}|, 2\delta + 1/|\mathbb{F}|\}$; with $\delta = 0.1$, we can take any $\kappa > 7/9$. We establish soundness by going through each of the tests in turn. (This is the same proof

flow as in [8] though the details differ for the quadratic correction test and circuit test.) First, from the given, we have that $\Pr\{\pi \text{ passes the linearity test}\} > 7/9$. This, together with our choice of δ, implies (see [12]) that $\pi^{(c)}$ is δ-close to some linear function $\phi(\cdot)$ (meaning that the fraction of inputs on which $\pi^{(c)}$ and $\phi(\cdot)$ disagree is $< \delta$). Second, we also have from the given that the probability of passing both the quadratic correction test and the linearity test is $> 2\delta + 2/|\mathbb{F}|$, so we can apply the following claim:

Claim D.3. If $\pi^{(c)}$ is δ-close to some linear function $\phi(\cdot)$ and if $\Pr\{\pi^{(c)} \text{ passes the quadratic correction test}\} > 2\delta + 2/|\mathbb{F}|$, then $\phi(\cdot)$ is exactly $\langle \cdot, A \oslash B \rangle$.

Proof. We begin with notation. Let $N_{A,B}(x,y) = \sum_{k=1}^{m} \langle A_{*,k}, x_{*,k} \rangle \cdot \langle B_{k,*}, y_{k,*} \rangle$, where $x, y \in \mathbb{F}^{m^3}$. We write the (i,k) element of q_4 and the (k,j) element of q_5 as q^4_{ik} and q^5_{kj} respectively. Let ϕ_{ijk} refer to the (i,j,k) element of the vector that corresponds to ϕ. Let $\sigma_{ijk} = \phi_{ijk} - A_{i,k}B_{k,j}$. Finally, let η be the event $\{\phi(q_4 \oslash q_5) = N_{A,B}(q_4, q_5)\}$.

Now, assume toward a contradiction that $\phi(\cdot) \neq \langle \cdot, A \oslash B \rangle$. Then $\exists i', j', k'$ such that $\sigma_{i'j'k'} \neq 0$. Let ν be the event $\{q^5_{k'j'} \neq 0\}$. We can write $\Pr\{\eta\} < \Pr\{\eta|\nu\} + \Pr\{\neg\nu\} = \Pr\{\eta|\nu\} + 1/|\mathbb{F}|$. To get an expression for $\Pr\{\eta|\nu\}$, note that if ν occurs, then $(q^5_{k'j'})^{-1}$ is defined. Also, $(\sigma_{i'j'k'})^{-1}$ is defined, by contrary assumption. Noting that η can be written $\{\sum_{i,j,k} q^4_{ik} q^5_{kj} \sigma_{ijk} = 0\}$, we have $\Pr\{\eta|\nu\} = \Pr\{q^4_{i'k'} = -(\sum_{i,j,k\setminus(i',j',k')} q^4_{ik} q^5_{kj} \sigma_{ijk}) \cdot (q^5_{k'j'})^{-1} \cdot \sigma^{-1}_{i'j'k'}\} = 1/|\mathbb{F}|$, since we can regard $q^4_{i'k'}$ as being chosen after the right-hand side of the equality. Thus, $\Pr\{\eta\} < 2/|\mathbb{F}|$. We now use the given, to derive a contradiction:

$$\begin{aligned} 2\delta + 2/|\mathbb{F}| &< \Pr\{\pi(q_6) - \pi(q_7) = N_{A,B}(q_4, q_5)\} \\ &< \Pr\{\phi(q_6) - \phi(q_7) = N_{A,B}(q_4, q_5)\} \\ &\quad + \Pr\{\pi(q_6) \neq \phi(q_6)\} + \Pr\{\pi(q_7) \neq \phi(q_7)\} \\ &< \Pr\{\phi(q_4 \oslash q_5) = N_{A,B}(q_4, q_5)\} + 2\delta, \end{aligned}$$

so $\Pr\{\eta\} > 2/|\mathbb{F}|$. Contradiction. □

Third, the probability of passing all three tests is $> 2\delta + 1/|\mathbb{F}|$ (again from the given), so we can apply the following claim to complete the lemma. □

Claim D.4. If $\pi^{(c)}(\cdot)$ is δ-close to $\phi(\cdot) = \langle \cdot, A \oslash B \rangle$ and if $\Pr\{\pi^{(c)} \text{ passes the circuit test}\} > 2\delta + 1/|\mathbb{F}|$, then $C = A \cdot B$.

Proof. Assume toward a contradiction that $C \neq A \cdot B$. Then there exists i', j' such that $C_{i'j'} \neq \sum_{k=1}^{m} A_{i',k} B_{k,j'}$, which implies that $P'(A,B) \triangleq \sum_{i,j} v^c_{i,j} \cdot (C_{i,j} - \sum_{k=1}^{m} A_{i,j} \cdot B_{i,j}) = 0$ with probability $\leq 1/|\mathbb{F}|$ (see Section 2.1). But

$$\begin{aligned} P'(A,B) &= \langle \gamma'_2, A \oslash B \rangle + \gamma'_0 \qquad \text{(by choice of } \gamma'_2, \gamma'_0) \\ &= \phi(\gamma'_2) + \gamma'_0 \qquad \text{(by definition of } \phi), \end{aligned}$$

so $\Pr\{\phi(\gamma'_2) + \gamma'_0 = 0\} \leq 1/|\mathbb{F}|$. We now use the given:

$$\begin{aligned} 2\delta + 1/|\mathbb{F}| &< \Pr\{\pi(q_8) - \pi(q_9) = -\gamma'_0\} \\ &< \Pr\{\phi(q_8) - \phi(q_9) = -\gamma'_0\} \\ &\quad + \Pr\{\pi(q_8) \neq \phi(q_8) \text{ or } \pi(q_9) \neq \phi(q_9)\} \\ &< \Pr\{\phi(\gamma'_2) = -\gamma'_0\} + 2\delta, \end{aligned}$$

so $\Pr\{\phi(\gamma'_2) + \gamma'_0 = 0\} > 1/|\mathbb{F}|$, giving a contradiction. □

We now briefly cover the costs; upper bounds are in Figure 2. The PCP encoding is m^3 work to produce. V constructs 8 queries and submits 7 of them to π; this is the same as the number of queries to $\pi^{(2)}$ in the baseline PCP (Appendix A). V requires $2m^2$ field multiplications for the quadratic correction test plus m^2 multiplications in the circuit test (to construct γ'_0), for a total of $3m^2$ per repetition. By comparison, the baseline PCP requires 2 operations for the quadratic correction test and $3m^2$ for the circuit test.

E Modifications for ElGamal

Since ElGamal encryption is multiplicatively homomorphic (rather than additively homomorphic), small modifications to Commit+Multidecommit (Figure 3) and the soundness arguments are necessary. Below, we describe these modifications and establish that the results of Appendix B still hold.

Fix the ElGamal group G, choose a generator g (known to both parties), and assume for now that $|\mathbb{F}_p| = |G|$ (we revisit this assumption below). Define the map $\psi\colon \mathbb{F}_p \to G$ by $x \mapsto g^x$. The map ψ is a group homomorphism and in fact an isomorphism; furthermore, ψ induces a ring structure on G. By composing ψ with ElGamal encryption, we get an additive homomorphism: $\mathsf{Enc}(pk, g^x)\mathsf{Enc}(pk, g^y) = \mathsf{Enc}(pk, g^{x+y})$. Of course, the verifier cannot recover $x + y$ explicitly from g^{x+y}, but this does not matter for Commit+Multidecommit. Also, given $\mathsf{Enc}(pk, g^x)$ and $a \in \mathbb{F}_p$, the properties of the ElGamal protocol imply that one can compute $\mathsf{Enc}(pk, g^{ax})$. Thus, given $(\mathsf{Enc}(pk, g^{r_1}), \ldots, \mathsf{Enc}(pk, g^{r_n}))$, one can compute $\mathsf{Enc}(pk, g^{\pi((r_1,\ldots,r_n))})$, where π is a linear function.

Therefore, we can modify Figure 3 as follows. First, during step 1, the verifier componentwise sends $\mathsf{Enc}(pk, g^r)$ rather than $\mathsf{Enc}(pk, r)$ to the prover. Next, in step 2, the prover computes $\mathsf{Enc}(pk, g^{\pi(r)})$ (without learning g^r), as described above. Then in step 3, V decrypts to get g^s. Finally, in step 6, the verifier checks that $g^b = g^{s + \alpha_1 a_1 + \cdots + \alpha_\mu a_\mu}$.

We now need to check that Lemma B.1 still holds. Correctness applies, by inspection. Binding applies because the semantic security of the encryption scheme can be formulated in terms of $\mathcal{A}(pk, \mathsf{Enc}(pk, g^r), r + \alpha q, r + \alpha' q)$ and because $g^x = g^y$ if and only if $x = y$ (since ψ is injective).

Note that when $|G| > |\mathbb{F}_p|$, the same approach works, via modular arithmetic. Specifically, although ψ is no longer a group isomorphism, it is injective. Provided that the computations never overflow, i.e., result in values in the exponent larger than $|G|$, the protocol remains correct.