2013 IEEE Symposium on Security and Privacy

A hybrid architecture for interactive verifiable computation

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

***Abstract*—We consider *interactive, proof-based verifiable computation*: how can a client machine specify a computation to a server, receive an answer, and then engage the server in an interactive protocol that convinces the client that the answer is correct, with less work for the client than executing the computation in the first place? Complexity theory and cryptography offer solutions in principle, but if implemented naively, they are ludicrously expensive. Recently, however, several strands of work have refined this theory and implemented the resulting protocols in actual systems. This work is promising but suffers from one of two problems: either it relies on expensive cryptography, or else it applies to a restricted class of computations. Worse, it is not always clear which protocol will perform better for a given problem.**

We describe a system that (a) extends optimized refinements of the non-cryptographic protocols to a much broader class of computations, (b) uses static analysis to fail over to the cryptographic ones when the non-cryptographic ones would be more expensive, and (c) incorporates this core into a built system that includes a compiler for a high-level language, a distributed server, and GPU acceleration. Experimental results indicate that our system performs better and applies more widely than the best in the literature.

1 Introduction

We are interested in verifiable computation:[1] how can a client outsource computation to a more powerful but potentially unreliable machine (or machines) and receive assurance that the computation was performed correctly? This problem has received renewed attention lately due to the rise of cloud computing and other third-party computing models [1, 3]: even if the third-party performer is not malicious, the client still has concerns about the integrity of data and computations.

The systems community has a long tradition of work on protocols that solve this problem but make strong assumptions about the usage model or focus on a restricted class of computations. For instance, solutions assume that failures are uncorrelated [3, 14, 34], that there is a chain of trust establishing faith in the computation [40, 42], or that the computation produces intermediate results amenable to checking [35].

It has also long been known that in principle, deep results in complexity theory [5–7, 24, 32, 48] and cryptography [13, 19, 27–29] yield protocols that are general-purpose and assume nothing about potential server misbehavior other than standard cryptographic hypotheses. These protocols involve a client, or *verifier*, that makes queries to a server, or *prover*. The details differ among these protocols, but broadly, the prover commits to a proof that the computation was executed correctly (usually, this proof is an execution trace of the computation, in some model of computation). Then, the verifier applies a randomized algorithm to ask unpredictable questions of the prover, and applies efficient tests to the responses. The protocols are constructed so that if the prover is honest, its responses pass the tests, while if the prover computed incorrectly, the verifier's tests exhibit an inconsistency, with high probability.

Despite the theoretical promise of these protocols, there was little attention to implementation, as performance was believed to be infeasibly poor. For instance, verifiable computation makes sense only if the cost of the protocol is less than the cost to the client of simply executing the computation locally—and it was not clear that this is true for many of the protocols in the literature. In fact, it turns out that naive implementations require hundreds of trillions of CPU years to verify matrix multiplication [45].

However, in the last few years there has been interest in building systems using this theory [16, 43]. Also, new theoretical works have emerged that emphasize amenability to potential implementation [9, 10, 20]. Moreover, several research groups have produced running code and measured performance [16, 44, 45, 47, 49].

The implementations follow two major approaches. The first is an interactive protocol (without cryptography) due to Cormode et al. [16] and derived from a protocol of Goldwasser et al. [23]; we will refer to this protocol as CMT and its base as GKR. The second is an efficient argument system (that uses cryptographic commitments) that we developed in prior work [44, 45, 47], building on Ishai et al. [27]; in fact, there are two protocols, Zaatar and Ginger, that will be relevant for our purposes. Both lines of work (CMT and Zaatar-Ginger) refine the original theory to make performance plausible, if not truly practical.

Of the two, the CMT protocol would initially appear to have greater long-term potential, as it does not rely on cryptography (which measurements indicate is a substantial fraction of the cost [44, 45, 47]). Moreover, Zaatar and Ginger work in the batching model, which means that they assume that the user is verifying many instances of the same computation, on different inputs. In contrast, CMT does not require batching.

However, on closer examination, the comparison is not so clear-cut. First, measured performance results [45, §7] are equivocal: CMT does not appear to do much better than the cryptographic protocols and sometimes does worse. Second, CMT is substantially less general-purpose than one would like—it works only for computations that can be expressed by highly regular circuits. In fact, these restrictions are severe enough that it is not clear how to produce compliant circuits automatically. Third, batching is actually a very reasonable

[1] Our use of the term *verifiable computation* is broader than its formalization in [19], which emphasized non-interactivity. See Section 8 for discussion.

1081-6011/13 $26.00 © 2013 IEEE
DOI 10.1109/SP.2013.48

model for cloud computation; for instance, it matches data parallel computations (e.g., the Map portion of MapReduce computations).

The purpose of this paper is to reduce the restrictions on CMT and then leverage both approaches:

1. We develop *CMT-slim*, a refinement of CMT (in the non-batched model) that eliminates a source of overhead and improves performance (Section 3).
2. We develop *CMT-batching*, which substantially extends the reach of CMT (and CMT-slim) to more general computations in the batching model (Section 4).
3. We build a system, *Allspice* (Section 5). Allspice takes the input program and employs a static analyzer to determine which protocol to use. On top of CMT-slim, we apply novel optimizations and produce a robust implementation (the first system based on CMT and GKR that we are aware of). For Zaatar-Ginger, we use existing infrastructure, which supports GPU acceleration and parallel hardware [47, §5]. Our combined system compiles a high-level language (with standard features like conditionals, loops, floating-point data types, and so forth) to executables that implement the client and server.
4. We perform detailed analysis and measurements to understand in depth the comparisons among the different protocols; these relationships are subtle, and we expect that this work will be useful for understanding when to deploy different approaches (Section 6).

The end result of all of this work is a built system for general-purpose interactive verifiable computation that, to our knowledge, has the best performance and applicability in the literature: (1) Allspice (via CMT-slim) improves over CMT by a factor of 2 or 3; (2) Allspice (via CMT-batching) applies a non-cryptographic (and hence cheaper) protocol to a much wider class than CMT, in some cases far outperforming the cryptographic protocols (Zaatar and Ginger); for instance, the break-even batch size for matrix multiplication under CMT-batching is orders of magnitude smaller than under the next-best protocol, Zaatar; and (3) Allspice (via Zaatar) applies to general-purpose computations.

Serious research hurdles remain. The computational models that we use, circuits and constraints, are not well-suited to representing general-purpose programs, notably loops. The floating-point representation is simulated in finite fields and so has poor performance compared to native operations and does not handle rounding.

Nonetheless, we are encouraged by the results that we report here, as they represent continued progress towards a system that is truly practical. The current limitations are consistent with certain cloud computing scenarios: batching is consistent with data-parallel setups; a restriction on loops is precisely the discipline imposed by MapReduce; and an expensive prover is less of a problem when there is an abundance of CPU cycles.

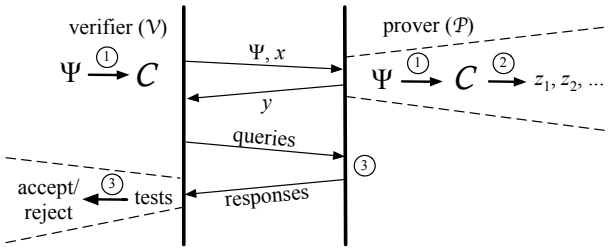


Figure 1—Proof-based interactive verifiable computation. Step ①: $\mathcal{V}$ and $\mathcal{P}$ represent a computation Ψ as an arithmetic circuit or set of constraints (denoted $\mathcal{C}$), and $\mathcal{P}$ returns an answer y in response to input x. Step ②: $\mathcal{P}$ identifies a satisfying assignment $z = (z_1, \ldots, z_n)$; the existence of z implies that $y = \Psi(x)$. Step ③: $\mathcal{V}$ queries $\mathcal{P}$, and the responses prove to $\mathcal{V}$ that a satisfying assignment exists.

2 Approach, tools, and background

This section describes the building blocks of Allspice. We first state the problem that we are trying to solve and give the high-level contours of the solution (§2.1). Next, we describe the CMT protocol (§2.2); this description is intended to be self-contained (but concise) and to give context for the refinements that we describe. Finally, we describe Zaatar and Ginger at a high level (§2.3), as necessary context.

2.1 Problem statement and framework

The following description of our problem statement and framework borrows heavily from our prior treatments [44, 47].

Problem statement. We wish to build a system in which one machine, a *verifier* $\mathcal{V}$, specifies a computation Ψ and input x to a *prover* $\mathcal{P}$. The prover $\mathcal{P}$ computes an output y and returns it to $\mathcal{V}$. If $y = \Psi(x)$, then a correct $\mathcal{P}$ should be able to convince $\mathcal{V}$ of y's correctness; otherwise, $\mathcal{V}$ should reject y with high probability. This protocol should be cheaper for $\mathcal{V}$ than executing Ψ locally, though we are willing to work in an amortized cost model (i.e., $\mathcal{V}$ saves work only when outsourcing a number of computations); this is consistent with some cloud computing scenarios. We are also willing to accept some overhead for $\mathcal{P}$, as we expect this type of assurance to have a price.

Finally, the protocol can make standard cryptographic assumptions about the limits of $\mathcal{P}$'s computational power, but it should not make other assumptions about $\mathcal{P}$; for instance, we cannot assume that $\mathcal{P}$ follows the protocol.

Framework. The solutions that we present below (§2.2–§2.3) consist of three steps, depicted in Figure 1.

First, $\mathcal{V}$ and $\mathcal{P}$ represent the computation Ψ as either an arithmetic circuit (a generalization of Boolean circuits) or constraints (a system of equations); the subsections below give more detail on these models, but broadly, the circuit and constraints are constructed so that they have a *satisfying assignment* if and only if Ψ executed correctly. In the circuit context, a satisfying assignment is a labeling of the circuit

wires for which: the input is x, the labels correspond to the correct circuit execution on input x, and the output is 0. In the constraint context, a satisfying assignment is a solution to the constraints.

Second, $\mathcal{P}$ computes $\Psi(x)$. In so doing, $\mathcal{P}$ identifies a satisfying assignment.

Third, using interactive proofs (IPs) [7, 24, 32, 48] or probabilistically checkable proofs (PCPs) [5, 6] (more accurately, using machinery [23, 27–29, 49] derived from this theory), $\mathcal{P}$ proves that it is holding a satisfying assignment. Of course, the statement "there is a satisfying assignment to this circuit (or these constraints)" admits a simple classical proof: the satisfying assignment itself, z. The trouble is that for $\mathcal{V}$ to read and check all of z would require executing the circuit, or checking the constraints, which would be as much work as carrying out Ψ.

Surprisingly, the theory implies that, by following a randomized algorithm and querying a prover $\mathcal{P}$, $\mathcal{V}$ can "check" z efficiently—without having to receive it! The properties offered by the theory are as follows; for the sake of simplicity, we state these properties loosely:

- **Completeness.** If $y = \Psi(x)$ (in which case a satisfying assignment to the circuit or constraints exists), then if $\mathcal{P}$ follows the protocol, $\Pr\{\mathcal{V} \text{ accepts}\} = 1$, where the probability is over $\mathcal{V}$'s random choices.
- **Soundness.** If $y \neq \Psi(x)$, (in which case no satisfying assignment exists and any purported proof must be spurious), then $\Pr\{\mathcal{V} \text{ accepts}\} < \epsilon$, where ϵ is small, and the probability is over $\mathcal{V}$'s random choices.

The nature of the soundness guarantee depends on context. Under the machinery described next, the soundness guarantee is unconditional: even a computationally unlimited $\mathcal{P}$ cannot fool $\mathcal{V}$ with probability greater than ϵ. Under the machinery presented in Section 2.3, the soundness guarantee is predicated on the semantic security of a homomorphic cryptosystem (in this case, ElGamal [17]) and thus ultimately on a standard cryptographic hardness assumption.

2.2 The CMT protocol

This section describes the CMT protocol [16, 49], which refines the GKR protocol [23]. Full details of GKR are in Rothblum [41, §3.2–3.3]. Our description below is influenced by these sources, though our notation sometimes diverges.

We consider computations Ψ that can have multiple inputs and outputs, so we represent the input x and output y as vectors. This protocol represents computations as arithmetic circuits, meaning a network of add and multiply gates, each with two input "wires" and one output "wire", where each wire can take values in a large finite field $\mathbb{F}$. CMT requires that $\mathbb{F} = \mathbb{F}_p$ (the integers mod a prime p). The CMT protocol requires that the circuit (a) is layered;[2] and (b) has a regular structure. We will discuss the implications of these requirements in Section 4 and for now take them as givens.

The protocol numbers the circuit layers from 0 to d (for depth) and presumes a single output gate at layer 0; the input gates are at level d. To represent the computation $y = \Psi(x)$, the circuit includes $-y$ in its input, computes $\Psi(x)$ and adds it to $-y$, and returns the sum. Thus, we say that the computation was executed correctly (meaning $\mathcal{P}$ claimed the correct y) if the circuit evaluates to 0 when given input $x' = (x, -y)$.

At a high level, the protocol works as follows. Based on the structure of the circuit (which $\mathcal{V}$ is assumed to know), the protocol, with $\mathcal{P}$'s help, proves to $\mathcal{V}$ that the output gate evaluates to 0 if and only if the gates at layer 1 satisfy a particular algebraic statement. While $\mathcal{V}$ could in principle verify this statement (by executing the circuit from layer d right up through layer 1 and then checking the statement explicitly), that would be too expensive. Instead, the protocol, again relying on $\mathcal{P}$'s help and $\mathcal{V}$'s knowledge of the circuit structure, proves to $\mathcal{V}$ that the algebraic statement about layer 1 holds if and only if another algebraic statement, this time about layer 2, holds. Again, this statement is too expensive for $\mathcal{V}$ to check explicitly, so the protocol continues until $\mathcal{V}$ has in hand a particular algebraic statement about the inputs x'. At that point, $\mathcal{V}$ checks the statement with no assistance from $\mathcal{P}$.

The protocol provides two guarantees. (1) If the prover computes the circuit correctly and participates in the protocol correctly, then $\mathcal{V}$ accepts. (2) If the prover computes the circuit incorrectly (i.e., the circuit's true output is not zero), then with high probability, the algebraic statements that the protocol claims to be equivalent are indeed equivalent—and hence all false. Thus, when $\mathcal{V}$ checks the final statement, $\mathcal{V}$ observes the falsehood, and rejects.

Details. We number the gates at each layer from 0, write them in binary, and assume (for ease of exposition) that every layer has the same number of gates, G. Let $b = \lceil \log_2 G \rceil$. Define a set of *evaluator functions* $V_0, \ldots, V_d$, one for each layer: $V_i \colon \mathbb{F}_2^b \to \mathbb{F}$. ($\mathbb{F}_2$ is the field consisting of $\{0, 1\}$.) Each V_i maps the gates at layer i to their values; if $\ell \geq G$, then $V_i(\ell) = 0$. Observe that each V_i implicitly depends on the input x', and that $V_d(j)$ returns the jth input element itself.

Motivation and straw man. At a high level, we want to reduce a claim that $V_0(0) = 0$ (i.e., that the circuit computed correctly) to a claim about V_1 and in turn to a claim about V_2 and so on, until $\mathcal{P}$ makes a claim about the input layer that $\mathcal{V}$ can check directly (since it supplies the input). Thus, we are motivated to write V_0 in terms of $V_1(\cdot)$ and more generally $V_{i-1}(\cdot)$ in terms of $V_i(\cdot)$. Define two functions, called the *wiring predicates* by Cormode et al. [16]: $\text{add}_i(g_1, g_2, g_3)$ returns 1 if gate g_1 at layer $i-1$ is the sum of the outputs of gates g_2, g_3 at layer i; it returns 0 otherwise. $\text{mult}_i(g_1, g_2, g_3)$ is analogous but returns 1 when g_1 at layer $i-1$ is the prod-

[2] Loosely, this means that the gates can be partitioned into sets (called layers), where the only wires in the circuit connect adjacent layers (versus skipping over them).

uct of the outputs of gates g_2, g_3 at layer i. Now, we can write $V_0(0) = \sum_{j_1,j_2 \in \{0,1\}^b} \text{add}_1(0,j_1,j_2) \cdot (V_1(j_1) + V_1(j_2)) + \text{mult}_1(0,j_1,j_2) \cdot V_1(j_1) \cdot V_1(j_2)$. More generally,

$$V_{i-1}(g) = \sum_{j_1,j_2 \in \{0,1\}^b} \text{add}_i(g,j_1,j_2) \cdot (V_i(j_1) + V_i(j_2)) + \text{mult}_i(g,j_1,j_2) \cdot V_i(j_1) \cdot V_i(j_2).$$

The form above (that $V_{i-1}(g)$ is a collection of sums) motivates the use of *sum-check protocols* [32, 48]; see also [4, §8.3.1] and [41, §3.2.3]. Such protocols are interactive, and consist of a verifier $\mathcal{V}_{\text{SC}}$ and a prover $\mathcal{P}_{\text{SC}}$. $\mathcal{V}_{\text{SC}}$ begins with some value K and a polynomial $f\colon \mathbb{F}^m \to \mathbb{F}$ of degree δ in each of its m variables. $\mathcal{V}_{\text{SC}}$ asks $\mathcal{P}_{\text{SC}}$ to prove that K equals the sum of the evaluations of f over all bit combinations:

$$K = \sum_{t_1 \in \{0,1\}} \sum_{t_2 \in \{0,1\}} \cdots \sum_{t_m \in \{0,1\}} f(t_1, t_2, \ldots, t_m). \qquad (1)$$

As usual, if the statement is true, $\mathcal{P}_{\text{SC}}$ can convince $\mathcal{V}_{\text{SC}}$; otherwise, $\mathcal{V}_{\text{SC}}$ is highly unlikely to be convinced. This process is vastly more efficient for $\mathcal{V}_{\text{SC}}$ than computing the sum itself.

A naive straw man would be to attempt to run one instance of the sum-check protocol for each $V_{i-1}(g)$, that is, each gate. However, this would be more work for the verifier (to say nothing of the prover) than simply executing the circuit. The technical details below get around this problem and permit the protocol to work with only d instances of the sum-check protocol. Though we do not have space to motivate all of the details, we nonetheless include them, to communicate the structure and mechanics of the protocol.

The protocol. Because sum-check protocols work with polynomials, the protocol *extends* the functions V_i to polynomials $\tilde{V}_i$. Define a *multilinear extension* $\tilde{g}$ of a function $g\colon \mathbb{F}_2^m \to \mathbb{F}$ as follows: $\tilde{g}$ agrees with g everywhere that g is defined, but $\tilde{g}$ is defined over $\mathbb{F}^m$ (instead of $\mathbb{F}_2^m$), and $\tilde{g}$ is a polynomial of degree at most one in each of the m variables.

The signature of $\tilde{V}_i$, then, is $\tilde{V}_i\colon \mathbb{F}^b \to \mathbb{F}$. Notice that the polynomial $\tilde{V}_d(\cdot)$ is the extension of the function $V_d(j) = x'_j$, where x'_j is the jth input. For $i = 1, \ldots, d$, GKR show [23, 41] that $\tilde{V}_{i-1}(\cdot)$ can be written as

$$\begin{aligned}\tilde{V}_{i-1}(q) &= \sum_{g,j_1,j_2 \in \mathbb{F}_2^b} \tilde{E}(q,g) \cdot \Big(\tilde{\text{add}}_i(g,j_1,j_2) \cdot \big(\tilde{V}_i(j_1) + \tilde{V}_i(j_2)\big) \\ &\qquad + \tilde{\text{mult}}_i(g,j_1,j_2) \cdot \tilde{V}_i(j_1) \cdot \tilde{V}_i(j_2) \Big) \\ &= \sum_{g,j_1,j_2 \in \mathbb{F}_2^b} f_q(g,j_1,j_2), \qquad (2)\end{aligned}$$

which is in a form suitable for the sum-check protocol (see equation (1)), since $f_q\colon \mathbb{F}^{3b} \to \mathbb{F}$ is a polynomial, defined as $f_q(u_1,u_2,u_3) = \tilde{E}(q,u_1) \cdot (\tilde{\text{add}}_i(u_1,u_2,u_3) \cdot (\tilde{V}_i(u_2) + \tilde{V}_i(u_3)) + \tilde{\text{mult}}_i(u_1,u_2,u_3) \cdot \tilde{V}_i(u_2) \cdot \tilde{V}_i(u_3))$. $\tilde{E}$ is a polynomial defined in Appendix A.1, and $\tilde{\text{add}}_i$ and $\tilde{\text{mult}}_i$ are multilinear extensions of add_i and mult_i.

```
1:  function VerifyCmtGkr(Circuit c)
2:      q_0 ← 0⃗ ∈ F^b
3:      a_0 ← 0
4:      d ← c.depth
5:      for i = 1, ..., d do
6:          // reduce a_{i-1} ?= Ṽ_{i-1}(q_{i-1}) to v_1 ?= Ṽ_i(w_1), v_2 ?= Ṽ_i(w_2)
7:          (v_1, v_2, w_1, w_2) ← SumCheck(i, q_{i-1}, a_{i-1})
8:
9:          // reduce v_1 ?= Ṽ_i(w_1), v_2 ?= Ṽ_i(w_2) to a_i ?= Ṽ_i(q_i)
10:
11:         h_0, h_1, ..., h_b ← GetFromProver()
12:         // above, correct prover returns b + 1 field elements,
13:         // specifically H(0), H(1), ..., H(b) ∈ F, where
14:         // H = Ṽ_i ∘ γ, for γ : F → F^b, γ(t) = (w_2 − w_1)t + w_1
15:         if h_0 ≠ v_1 or h_1 ≠ v_2 then
16:             return reject
17:         τ ←R F
18:         q_i ← γ(τ)
19:         a_i ← H*(τ) // H* is poly. interpolation of h_0, ..., h_b
20:         SendToProver(τ)
21:     if a_d = Ṽ_d(q_d) then    // requires computing Ṽ_d(q_d)
22:         return accept
23:     return reject
```

Figure 2—Pseudocode for $\mathcal{V}$ in the GKR and CMT protocol [16, 23, 41]. The protocol proceeds in layers, reducing the claim that $a_{i-1} = \tilde{V}_{i-1}(q_{i-1})$ to a claim that $a_i = \tilde{V}_i(q_i)$. SumCheck is defined in Appendix A. See Rothblum [41] for explanation of all details.

Pseudocode for the protocol is in Figure 2. $\mathcal{V}$ wants to be convinced that $\tilde{V}_0(0) = 0$; this equality is tantamount to establishing that the circuit evaluates to 0 and hence that Ψ was carried out correctly. To this end, the protocol starts with $q_0 = 0$ and $a_0 = 0$, and in each iteration $i = 1, \ldots, d$, the protocol produces some $a_i \in \mathbb{F}$ and $q_i \in \mathbb{F}^b$ and a statement: "you should believe that $a_{i-1} = \tilde{V}_{i-1}(q_{i-1})$ if and only if you can establish that $a_i = \tilde{V}_i(q_i)$" (lines 7–19). The guarantees of each iteration are: (1) If the iteration begins with a true statement, i.e., if $a_{i-1} = \tilde{V}_{i-1}(q_{i-1})$, then if $\mathcal{P}$ behaves correctly, the protocol produces another true statement, i.e., for the a_i and q_i that it returns, $a_i = \tilde{V}_i(q_i)$. (2) If the iteration begins with a false statement, i.e., if $a_{i-1} \neq \tilde{V}_{i-1}(q_{i-1})$, then the protocol produces another false statement with high probability, i.e., for the a_i and q_i that it returns, $a_i \neq \tilde{V}_i(q_i)$.

The process ends with telling $\mathcal{V}$ " ... if and only if you can establish that $a_d = \tilde{V}_d(q_d)$," at which point $\mathcal{V}$ performs that check directly (line 21). By composing the iterations together, we see that if the protocol begins with a true statement, a correct $\mathcal{P}$ makes $\mathcal{V}$ accept. On the other hand, if the protocol begins with a false statement—an incorrect claim that the circuit evaluates to 0—then with high probability the protocol produces an a_d such that $a_d \neq \tilde{V}_d(q_d)$; as a result, $\mathcal{V}$ rejects.

Guarantees and costs. The protocol described above satisfies completeness and soundness (§2.1), with $\epsilon \leq 7 \cdot b \cdot d/|\mathbb{F}|$; see Rothblum [41, §3.3.3] for a derivation. Since $|\mathbb{F}|$ is astronomical (§6.1), this error can be neglected in practice.

$\mathcal{V}$ incurs one of its principal costs inside the sum-check protocol: in each iteration, $\mathcal{V}$ must compute $\tilde{\text{add}}_i(w_0, w_1, w_2)$ and $\tilde{\text{mult}}_i(w_0, w_1, w_2)$, for random $w_0, w_1, w_2 \in \mathbb{F}^b$ (see Appendix A.1). Thus, for the protocol to be efficient for $\mathcal{V}$, the circuit must admit an efficient computation of $\tilde{\text{add}}_i$ and $\tilde{\text{mult}}_i$, a restriction that we return to in Section 4. Assuming (as CMT does) that computing $\tilde{\text{add}}_i$ and $\tilde{\text{mult}}_i$ is $O(\text{polylog}(G))$, the verifier's running time is $O(|x'| \log |x'| + d \cdot \text{polylog}(G)) = O((|x| + |y|) \cdot \log(|x| + |y|) + d \cdot \text{polylog}(G))$.

Regarding $\mathcal{P}$, the innovations of Cormode et al. [16] make the sum-check protocol efficient for $\mathcal{P}$, resulting in overall running time for $\mathcal{P}$ of just $O(d \cdot G \cdot \log G)$: not much more than executing the $d \cdot G$ gates of the computation.

2.3 Zaatar and Ginger

This section gives an overview of Zaatar [44] and Ginger [47]. We will abstract most details, as our intent is only to describe the structure of these protocols for context.

The computational model. These protocols represent computations as *constraints*, meaning a system of equations, over a finite field $\mathbb{F}$, with one or more distinguished input variables (which we denote loosely as X) and one or more distinguished output variables (denoted loosely as Y). These protocols require constraints to be total degree 2, meaning that the summands in each constraint are either variables or products of two variables. A set of constraints is *satisfiable* if there is a setting of the variables that makes all of the equations hold. We write $\mathcal{C}(X{=}x, Y{=}y)$ to mean the resulting equations when X and Y are set to respective values x and y. We say that a set of constraints $\mathcal{C}$ is *equivalent* to a computation Ψ if: for all x and y, we have $\mathcal{C}(X{=}x, Y{=}y)$ is satisfiable if and only if $y = \Psi(x)$. As an example, the computation multiply-by-5 is equivalent to $\{5 \cdot Z - Y = 0,\ Z - X = 0\}$.

Ginger shows how to transform standard program constructs (logical operations, inequality comparisons, if/else, etc.) into this "assembly language" of degree-2 constraints; see [47][46, Apdx. C] for details. Most of the resulting constraints are very concise; an exception is inequalities, which require $\approx \log_2 |\mathbb{F}|$ constraints per operation (intuitively, the inequality constraints separate a number into its bits and glue it back together for the rest of the computation). Another limitation of the constraint formalism is it is necessary to unroll loops, requiring bounds to be known at compile time.

Zaatar and Ginger incorporate compilers that perform this transformation automatically (the two protocols require slightly different forms). The compilers are derived from the Fairplay compiler [33] and take as input programs expressed in SFDL, the language of Fairplay. SFDL is a high-level language that supports conditional control flow, logical expressions, and inequality comparisons; Ginger extends the language with a primitive floating-point data type.

In the rest of the section, we will take as a given the equivalent constraints to Ψ, which we denote $\mathcal{C}$.

The verification machinery. Given a set of constraints $\mathcal{C}$, input x, and output y, Zaatar and Ginger start with a probabilistically checkable proof (PCP) that a satisfying assignment to $\mathcal{C}(X{=}x, Y{=}y)$ exists. However, the "textbook" PCPs that would admit a straightforward implementation (see [36, §7.8] and [4, §11.5]) are too big to be sent over the network to $\mathcal{V}$. Instead, Zaatar and Ginger use an efficient argument protocol [27–29]; the idea is for $\mathcal{P}$ to *cryptographically commit* to the contents of the PCP, after which $\mathcal{V}$ queries $\mathcal{P}$, ensuring that the responses are consistent with the commitment. This is where the computational assumptions on $\mathcal{P}$ enter: provided $\mathcal{P}$ cannot break the commitment primitive, then $\mathcal{P}$ is forced to behave like an oracle, which allows $\mathcal{V}$ to proceed as though the (enormous) PCP were available for inspection.

How can $\mathcal{P}$ commit to an enormous PCP? Ishai et al. [27] demonstrate that if the PCP is a linear function π, then the server can commit to the contents of the proof without materializing the entire thing. This primitive leverages additively (not fully) homomorphic encryption (for instance, Paillier [37] and ElGamal [17]), as follows. Loosely, $\mathcal{V}$ homomorphically encrypts a secret, r, and sends $\mathsf{Enc}(r)$ to $\mathcal{P}$; using the linearity of π and the homomorphic property of $\mathsf{Enc}(\cdot)$, $\mathcal{P}$ returns $\mathsf{Enc}(\pi(r))$, which $\mathcal{V}$ decrypts. Later, in response to $\mathcal{V}$'s PCP queries $q_1, \ldots, q_\mu$, $\mathcal{V}$ ensures that the responses $\pi(q_1), \ldots, \pi(q_\mu)$ are consistent with $\pi(r)$. Instead of materializing π (which is an exponentially-sized table), $\mathcal{P}$ can concisely represent the proof as some vector u.

Ginger (and its predecessor Pepper [45]) apply a series of refinements to make the above approach vastly more efficient than the original theory. By default, Ginger uses the classical linear PCP of Arora et al. [5], in which the vector u is quadratically larger than the number of variables in the satisfying assignment. Thus, making Ginger's performance plausible usually requires hand-tailoring the contents of u and the query protocol.

Zaatar, however, makes the above approach plausible in general, using a novel linear PCP based on QAPs [20]. Compared to Ginger, Zaatar is more concise asymptotically—the vector u is now *linear* in the size of the computation—though there is some additional overhead. For most computations, Zaatar is far more efficient than Ginger; when it is less efficient, the difference is bounded [44, §4]. However, for both protocols, the cryptographic operations are still a significant expense.

Amortization and costs. To amortize the query generation cost, $\mathcal{V}$ works over multiple instances of a given computation Ψ at once. Specifically, $\mathcal{V}$ has in hand parallel inputs $x^{(1)}, \ldots, x^{(\beta)}$. $\mathcal{V}$ also has values $y^{(1)}, \ldots, y^{(\beta)}$ that purport to be $\Psi(x^{(1)}), \ldots, \Psi(x^{(\beta)})$. Then, $\mathcal{V}$ extracts a cryptographic commitment to proofs $\pi^{(1)}, \ldots, \pi^{(\beta)}$ and submits the same queries to these β proofs.

Zaatar's costs are as follows. If the constraint set $\mathcal{C}$ has $|Z|$ variables, $|\mathcal{C}|$ constraints, and K additive terms, then $\mathcal{V}$'s query generation work has running time $O(|Z| + |\mathcal{C}| + K)$ for

the batch.[3] In amortized terms, $\mathcal{V}$'s per-instance running time is $O(|x| + |y| + (|Z| + |\mathcal{C}| + K)/\beta)$. The running time of $\mathcal{P}$ is $O(|Z| + |\mathcal{C}| \cdot \log^2 |\mathcal{C}|)$. See [44] for a detailed analysis of the constants and a Zaatar-Ginger comparison.

3 Refining CMT: CMT-slim

CMT verifies single-output circuits C; the verification procedure establishes that on input x, $C(x) = 0$ (§2.2). Thus, to apply CMT to checking whether $y = \Psi(x)$, we need a circuit that (a) takes input $x' = (x, -y)$ and (b) includes machinery to squeeze the output to one bit. This approach incurs overhead, as we explain in the first part of this section.

The second part of this section presents CMT-slim, our refined protocol. CMT-slim substantially changes the circuit structure; this in turn requires a slight change in the verification procedure. The combined result is a significant cost reduction.

Details of CMT's circuit structure. Under CMT, circuits have a particular structure; they are composed of three parts:

1. *The compute circuit.* This part computes Ψ.
2. *The output propagation circuit.* This part is conceptually next to the compute circuit, making the circuit wider.
3. *The comparison circuit.* This part sits atop the prior two, making the circuit taller; its purpose is to return 0 if and only if the purported and correct outputs are equal.

We now detail each of these parts. The compute circuit takes as input Ψ's inputs $x = (x_1, x_2, \ldots, x_{|X|})$ and computes the correct output $\hat{y} = (\hat{y}_1, \hat{y}_2, \ldots, \hat{y}_{|Y|})$. The output propagation circuit takes as input the additive inverses of the purported outputs $y = (y_1, \ldots, y_{|Y|})$, as well as 0, and propagates $-y$ to the layer where y appears.

The purpose of the comparison circuit is to return 0 if and only if $y_i = \hat{y}_i$ for all $i \in [1, |Y|]$. This part begins with a layer that computes $\delta_i = y_i - \hat{y}_i$, for $i \in [1, |Y|]$. If $|Y| = 1$, we are done: the circuit returns δ_1, which equals 0 if and only if $y_1 = \hat{y}_1$. However, if there are multiple outputs, the circuit cannot simply sum the δ_i, since that sum could incorrectly overflow to 0. Instead, the circuit leverages Fermat's Little Theorem (FLT) to turn each δ_i into a 0-1 value: because we are working over a finite field $\mathbb{F}_p$ for some large prime p, δ_i^{p-1} evaluates to 1 if δ_i is non-zero and 0 otherwise. Thus, the remainder of the circuit computes $S = \sum_{i=1}^{|Y|} \delta_i^{p-1}$, which does not overflow. In particular, $S \in \{0, 1, \ldots, |Y|\}$, and $S = 0$ if and only if $y_i = \hat{y}_i$ for all $i \in [1, |Y|]$, as desired.

Overhead. As stated in Section 2.2, a circuit with depth d and a maximum of G gates at each layer induces a running time of $O(d \cdot \text{polylog}(G))$ for $\mathcal{V}$ and $O(d \cdot G \cdot \log G)$ for $\mathcal{P}$, where the polylog(G) term is the cost of evaluating the circuit's wiring predicates: $\tilde{\text{add}}$ and $\tilde{\text{mult}}$. The comparison circuit requires $\lceil \log_2 |\mathbb{F}| \rceil$ layers to compute δ_i^{p-1} (via repeated squaring),[4] each layer requires $2|Y|$ gates, and its wiring predicates require $O(\log_2 |Y|)$ to compute (see [16, Apdx. A.4.1] for details). In total, then, the comparison circuit adds $O(\log_2 |\mathbb{F}| \cdot \log_2 |Y|)$ running time for $\mathcal{V}$ and $O(\log_2 |\mathbb{F}| \cdot |Y| \cdot \log_2 |Y|)$ for $\mathcal{P}$. This overhead can be significant, as observed empirically in Section 6.2 and in [47, §7].

Details of the refinements. Our principal modification to CMT is to shed two of the three circuit components described above, so the protocol works with the compute circuit directly. This modification requires us to change the verification algorithm slightly: it is now initialized based on the low-degree extension of the purported output y, rather than 0. Specifically, we change lines 2–3 in Figure 2 to initialize q_0 to a random vector in $\mathbb{F}^{\lceil \log_2 |Y| \rceil}$, and set $a_0 \leftarrow \tilde{V}_y(q_0)$. Here, $\tilde{V}_y$ is the multilinear extension of V_y, where $V_y(j)$ returns the jth purported output, y_j. As shown in Appendix A.2, this modification preserves completeness, and soundness error increases only minorly: from ϵ (which is CMT's soundness error, quantified in Section 2.2) to $\epsilon' = \epsilon + \lceil \log_2 |Y| \rceil / |\mathbb{F}|$.

Although these modifications are straightforward, they provide substantial benefits, as depicted in Figure 3 (the figure depicts other cost savings that result from implementation optimizations that we describe in Section 5).

4 Extending CMT's reach: CMT-batching

CMT (and hence CMT-slim) is promising: it has the potential to be extremely efficient for $\mathcal{V}$ (as it requires no cryptographic operations and little network cost), and its mechanics are relatively straightforward. However, it has three limitations that conflict with general-purposeness (see also Section 2.2):

1. *It represents computations as arithmetic circuits.* Arithmetic circuits cannot, to our knowledge, represent computations with inequality comparisons (`<`, `<=`, `>`, `>=`) and not-equal (`!=`) tests, except by degenerating to far more verbose Boolean circuits.
2. *These circuits must be regular, in the sense that* $\tilde{\text{add}}_i$ *and* $\tilde{\text{mult}}_i$ *should be efficiently computable.* This limits computations to those where the programmer can represent the structure of the computation with a concise mathematical expression.
3. *These circuits must be efficiently parallel: "wide" but not "tall".* Of course, not all computations are this way.

The rest of the section describes refinements that partially ease the limitations above and considers when those refinements apply. First, Section 4.1 describes our modified circuits, which take auxiliary inputs (or *advice*), supplied by the prover. These circuits-with-advice extend CMT's computational model to a larger class of computations; in fact we can automatically compile from the high-level language in the Zaatar-Ginger framework (§2.3) to a concise circuit-with-

[3] We are not explicitly charging for the cost of working in $\mathbb{F}$.

[4] This ignores some additional layers for computing S. However, these layers induce few costs and can be neglected; see [16, §3.2] for details.

protocol	$\mathcal{V}$'s running time	$\mathcal{P}$'s running time
CMT	$f \cdot (\lvert X\rvert + \lvert Y\rvert) \cdot \log_2(\lvert X\rvert + \lvert Y\rvert) + d_\Psi \cdot I(G_{\Psi,\text{prop}}) + 5 \cdot f \cdot d_{\text{eq}} \cdot \log_2 G_{\text{eq}}$	$24 \cdot f \cdot (d_\Psi \cdot G_{\Psi,\text{prop}} \cdot \log_2 G_{\Psi,\text{prop}} + d_{\text{eq}} \cdot G_{\text{eq}} \cdot \log_2 G_{\text{eq}})$
CMT-slim	$f \cdot (\lvert X\rvert + \lvert Y\rvert) \cdot 4 + d_\Psi \cdot I(G_\Psi)$	$24 \cdot f \cdot d_\Psi \cdot G_\Psi \cdot \log_2 G_\Psi$

$|X|$: number of elements (from $\mathbb{F}$) in input to Ψ
$|Y|$: number of elements (from $\mathbb{F}$) in output of Ψ
G_Ψ: width of (max. # of gates in) each layer of compute circuit
$G_{\Psi,\text{prop}}$ ($= G_\Psi + |Y|$): width of each layer of compute+propagate circuit
G_{eq} ($= 2|Y|$): width of (# gates in) comparison circuit
d_{eq} ($= \lceil \log_2 |\mathbb{F}| \rceil$): depth of (# layers in) comparison circuit
d_Ψ: depth of (# layers in) circuit to compute Ψ
f: cost of field multiplication
$I(G)$: Cost to check $\mathcal{P}$'s responses ($f \cdot \log_2^2 G + 27 \cdot f \cdot \log_2 G$; see Fig. 2, line 19 and Fig. 11, line 14), and compute $\tilde{\text{add}}$ and $\tilde{\text{mult}}$ (Fig. 11, line 24), assuming width G.

Figure 3—Running times of $\mathcal{V}$ and $\mathcal{P}$ in CMT and CMT-slim, when verifying a computation Ψ. The changes include replacing the factor $\log_2(|X| + |Y|)$ with 4, owing to an implementation optimization that is described in Section 5.1. The remaining improvements result from shedding the propagation circuit (reflected in the drop from $G_{\Psi,\text{prop}}$ to G_Ψ) and the comparison circuit (reflected in dropping the d_{eq} and G_{eq} terms). The figure leaves unspecified the cost of computing $\tilde{\text{add}}$ and $\tilde{\text{mult}}$, but CMT and GKR require circuits for which this is efficient.

advice. While this refinement (partially) addresses the first limitation above, it exacerbates the second one, of regularity. However, Section 4.2 describes how we can tolerate irregularity and non-parallelism, using batching in the Zaatar-Ginger style (§2.3); we call the resulting protocol *CMT-batching*.[5]

Finally, Section 4.3 analyzes the applicability of CMT-batching. The high-level finding is that CMT-batching is most useful when there are not many comparisons or not-equal operations, as a fraction of the total number of operations. In particular, if the number of comparisons scales with the computation's running time (as is the case for sorting, searching, etc.), then CMT-batching does not save work for $\mathcal{V}$.

4.1 Beyond arithmetic circuits

Our goal in this subsection is to transform computations expressed in a high-level language to arithmetic circuits. (For now, we are not trying to achieve efficiency for $\mathcal{V}$ relative to local execution.) A natural attempt would be to use the compiler of Zaatar-Ginger (§2.3). However, its "assembly language" is constraints, not circuits. What is involved in transforming to arithmetic circuits? An analysis of the compiler indicates that most of the programming constructs (if/else, etc.) have a natural representation in constraints or circuits.

The problematic constructs are the not-equal operation (`!=`) and inequality comparisons (`<`, `<=`, `>`, `>=`): no arithmetic circuit that we are aware of directly and efficiently computes these.[6] The constraint representation of these constructs sidesteps this issue by including auxiliary variables that $\mathcal{P}$ sets [47, §4][46, Apdx. C].

The idea of our first refinement, then, is to have $\mathcal{P}$ do the required computations "out-of-band" and to supply the outcome to $\mathcal{V}$. This requires reformulating the circuit to (a) accept as input (advice) the outcome of all inequality and not-equal operations and (b) *check* that these inputs are correct.

[5] CMT [16] and GKR [23] suggest circumventing the protocol limitations by moving some of the work offline. However, this does not change the total work for $\mathcal{V}$; it simply changes when the cost is incurred. By contrast, our approach can save work for $\mathcal{V}$, and we analyze when that happens.

[6] One could use FLT to compute not-equal, but as noted in Section 3, that approach has undesirable overhead.

Details. Let $(X^{(!=)}, X^{(<)}, M^{(!=)}, M^{(<)})$ denote a vector of advice supplied by $\mathcal{P}$ to $\mathcal{V}$. Here, $X^{(!=)}$ denotes the claimed 0-1 outcome of all not-equal checks; $X^{(<)}$ denotes the claimed 0-1 outcome of all inequality comparisons; and $M^{(!=)}$ and $M^{(<)}$ are, loosely speaking, witnesses that those claimed outcomes are indeed the correct ones (more specifically, $M^{(!=)}$ and $M^{(<)}$ are the auxiliary variables in the Zaatar-Ginger constraints). The following circuit structure is created by our compiler; it consists of two parts:

- *The compute-advised circuit.* This part computes Ψ, taking as advice the claimed outcome of all not-equal and inequality operations. Specifically, the inputs here are $(X, X^{(!=)}, X^{(<)})$, and the output is Y, where X and Y are the inputs to, and outputs of, Ψ. This part assumes that its advice is correct.
- *The check-advice circuit.* This part checks that $X^{(!=)}$ and $X^{(<)}$ are correct. It takes as input $(X^{(!=)}, X^{(<)}, M^{(!=)}, M^{(<)})$ and returns 0 if and only if $X^{(!=)}$ and $X^{(<)}$ are correct.

The check-advice circuit essentially recapitulates the Zaatar-Ginger constraints for not-equal and inequality comparisons, taking as input the values of all auxiliary variables. For example, say that the ith not-equal operation in Ψ is

```
Z1 != Z2.
```

Then, the compute-advised circuit includes wires for Z_1 and Z_2. Labeling the ith elements of the prover-supplied $X^{(!=)}$ and $M^{(!=)}$ as $X_i^{(!=)}$ and $M_i^{(!=)}$, the check-advice circuit computes:

$$(1 - X_i^{(!=)}) \cdot (Z_1 - Z_2) \quad \text{and} \quad (Z_1 - Z_2) \cdot M_i^{(!=)} - X_i^{(!=)}.$$

By inspection, the output of both of these operations is 0 if and only if $X_i^{(!=)}$ and $M_i^{(!=)}$ are set correctly by $\mathcal{P}$; in particular, if $Z_1 \neq Z_2$, then the correct setting of $M_i^{(!=)}$ is the multiplicative inverse of Z_1 and Z_2.

The approach for inequality comparisons is similar: we have an operation in the circuit for each Zaatar-Ginger constraint. There are $\lfloor \log_2(\mathbb{F}) \rfloor + 7$ such constraints (see [46, Apdx. C] for details).

Thus, the total cost of the check-advice circuit is as follows. We add G_{ck} gates to each layer and to the output, where

	$\mathcal{V}$'s running time	$\mathcal{P}$'s running time
baseline	$f \cdot d_\Psi \cdot G_\Psi$	N/A
CMT-batching	$f \cdot d_\Psi \cdot G_{\Psi,\text{ck}} \cdot 11/\beta + 4 \cdot f \cdot (\lvert X\rvert + \lvert Y\rvert + 2 \cdot G_\text{ck}) + d_\Psi \cdot \text{CheckResp}(G_{\Psi,\text{ck}})$	$24 \cdot f \cdot d_\Psi \cdot G_{\Psi,\text{ck}} \cdot \log_2 G_{\Psi,\text{ck}}$

d_Ψ, G_Ψ	depth, width of compute-advised circuit
$G_\text{ck} = 2 \cdot \lvert X^{(!=)}\rvert + (\lfloor \log_2(\mathbb{F}) \rfloor + 7) \cdot \lvert X^{(<)}\rvert$	width of check-advice circuit
$G_{\Psi,\text{ck}} = G_\Psi + G_\text{ck}$	width of full circuit, under CMT-batching
$\text{CheckResp}(G) = f \cdot (\log_2^2 G + 27 \cdot \log_2 G)$	per-layer cost to check $\mathcal{P}$'s responses (Fig. 2, line 19 and Fig. 11, line 14), assuming width G

Figure 4—Running times of $\mathcal{V}$ and $\mathcal{P}$ in CMT-batching when verifying a computation Ψ, compared to the baseline of evaluating the computation. Some of the terminology and notation is borrowed from Figure 3. The estimate of the baseline presumes that local inequality and not-equal operations are free, which is a reasonable approximation. The computation is compiled into a compute-advised and a check-advice circuit, which together include inputs for prover-supplied advice (of length G_ck). Because the protocol applies to arbitrary layered circuits, computing $\tilde{\text{add}}$ and $\tilde{\text{mult}}$ may be expensive, but it amortizes over the batch (see the first summand in $\mathcal{V}$'s costs). By contrast, the approach to advice does not necessarily scale; see text (§4.3).

$G_\text{ck} = 2 \cdot \lvert X^{(!=)}\rvert + (\lfloor \log_2(\mathbb{F}) \rfloor + 7) \cdot \lvert X^{(<)}\rvert$. We say "each layer" because the outputs of the individual checks are not necessarily at the output layer, necessitating dummy gates to propagate these outputs to the top (this is similar to the output preservation circuit that we shed; see Section 3).

4.2 Amortization to sidestep regularity

Although the refinement above (partially) addresses the first limitation, it has made the regularity issue worse: it seems difficult to build a circuit automatically *and* to obtain an efficiently computable $\tilde{\text{add}}$ and $\tilde{\text{mult}}$ for each layer. Moreover, even if the compiler could produce a regular compute-advised circuit, the check-advice circuit is unlikely to be regular.

Our solution is to give up on regularity. Instead, we will pay for the irregularity and then amortize this cost over multiple instances of the computation, similar to the way that Zaatar and Ginger amortize their query generation cost. Specifically, the cost of computing $\tilde{\text{add}}$ and $\tilde{\text{mult}}$ (see Sections 2.2 and 3) is now permitted to be much larger. This approach also addresses the third limitation: the amortization means that $\mathcal{V}$ saves work for a much larger class of circuits, allowing for verification of circuits with smaller layers (fewer gates) and larger depth (more layers) than under CMT-slim.

We call the resulting protocol *CMT-batching*. It consists of β parallel instances of Figure 2, resulting in the following structure:

1. $\mathcal{V}$ and $\mathcal{P}$ exchange inputs $x^{(1)}, \ldots, x^{(\beta)}$ and outputs $y^{(1)}, \ldots, y^{(\beta)}$.
2. $\mathcal{V}$ randomly generates q_0 and computes $a_0^{(k)} = \tilde{V}_{y^{(k)}}(q_0)$ for all instances $k \in \{1, \ldots, \beta\}$.
3. In each iteration, $\mathcal{V}$ uses the same randomness (for example, τ in Figure 2, line 17) for all instances.
4. $\mathcal{P}$ responds separately for each instance, and $\mathcal{V}$ checks each instance independently. Importantly, this means $\mathcal{V}$ can compute $\tilde{\text{add}}_i$ and $\tilde{\text{mult}}_i$ *once for all instances*.
5. At the end, $\mathcal{V}$ checks whether $a_d^{(k)} = \tilde{V}_d^{(k)}(q_d)$ for all k; throughout, if $\mathcal{V}$ rejects any instance, it rejects them all.

CMT-batching satisfies completeness. Its soundness property is that each instance, viewed individually, has the same soundness error as CMT-slim (namely $\epsilon' = 7 \cdot b \cdot d/\lvert\mathbb{F}\rvert + \lceil \log_2 \lvert Y\rvert \rceil / \lvert\mathbb{F}\rvert$; see Section 3), a claim that is proved in Appendix A.2.

4.3 Applicability

CMT-batching mitigates the three limitations, compared to the base protocol. We say "mitigates" instead of "eliminates" because $\mathcal{V}$ does not always save work versus executing the computation locally. We now analyze CMT-batching, focusing on when it does save work for $\mathcal{V}$.

Figure 4 depicts the costs of CMT-batching, comparing it to local execution. There are two significant costs for $\mathcal{V}$. The first is computing $\tilde{\text{add}}_i$ and $\tilde{\text{mult}}_i$. This costs $11 \cdot f \cdot d \cdot (G_\Psi + G_\text{ck})$ because the circuit is not regular, so we must compute from the general expression for $\tilde{\text{add}}_i$ and $\tilde{\text{mult}}_i$ (given in Appendix A). While this task can be as or more costly than executing all $d \cdot G_\Psi$ gates in the computation, it amortizes over β instances.

The second source of costs is processing the advice. These costs correspond to work done to verify a circuit component that does not execute "real work"; worse, these costs do not amortize, so they could make the protocol more expensive for $\mathcal{V}$ than executing locally. Notice the term G_ck. The key comparison is between $8 \cdot f \cdot G_\text{ck}$ (the work done by $\mathcal{V}$ to process the advice) and $f \cdot d \cdot G_\Psi$ (the cost of local execution). If $G_\text{ck} < d \cdot G_\Psi/8$ (ignoring some constants), then $\mathcal{V}$ can gain from outsourcing, and the smaller G_ck, the smaller the batch size at which $\mathcal{V}$ breaks even.

Thus, CMT-batching works well—in the sense of saving $\mathcal{V}$ work versus computing locally—when G_ck is small. Such values of G_ck correspond to computations with few not-equal operations and inequality comparisons. In particular, the number of such operations should scale sub-linearly in the running time of the computation, and the actual ceiling could be very small indeed, depending on the constants.

5 Implementation details

Allspice uses the Ginger implementation from [46]. Allspice optimizes the Zaatar prover by constructing the proof vector u (§2.3) using the fast Fourier transform (FFT), a suggestion

CMT-slim removes an additive overhead that scales with the output size of the computation. For $m \times m$ matrix multiplication with m=128 and compared to CMT, the CMT-slim verifier is 2× less expensive, the CMT-slim prover is 1.7× less expensive, and the network costs reduce by about 30%.	§6.2
CMT-batching enhances the types of computations that can be efficiently outsourced, relative to CMT-slim. For instance, CMT-batching applies to root finding by bisection, which has control flow and inequalities while CMT-slim does not.	§6.3
CMT-batching has far better performance compared to the cryptographic sub-protocols in Allspice (Ginger and Zaatar) for quasi-straightline computations. For these computations, the per-instance cost to the prover in CMT-batching can be cheaper than Ginger by 1–2 orders of magnitude and is comparable to Zaatar. The break-even batch sizes (and network costs) under CMT-batching are 1–4 orders of magnitude smaller relative to Ginger and 1–2 orders of magnitude smaller relative to Zaatar.	§6.3

Figure 5—Summary of main evaluation results.

of Gennaro et al. [20]. As a result, $\mathcal{P}$'s per-instance running time drops from $O(|Z|+|\mathcal{C}|\cdot\log^2|\mathcal{C}|)$ to $O(|Z|+|\mathcal{C}|\cdot\log|C|)$.

Below, we describe the CMT portion of Allspice and the static analyzer that decides which protocol to use.

5.1 CMT and CMT-slim

Using the CMT implementation [16] as a reference, we implemented the base CMT protocol as a C++ verifier and prover; they run as separate processes that communicate via Open MPI [2]. Our implementation is 5533 lines of base code, with roughly 615 lines required for each new computation (recall that CMT circuits are manually constructed). In the process, we made a number of optimizations that have a substantial impact on performance (see Section 6.2).

The most significant optimization applies to the prover in the sum-check protocol, saving approximately 15-20% of the prover's work; Appendix A.1 gives the details.

We also reduce the cost of computing $\tilde{V}_d(q_d)$, from $O(|X|\log|X|)$ to $O(|X|)$. (This applies to Figure 2, line 21.) We use memoization, as follows.[7] There are $|X|$ gates in the input layer. Let $n = \lceil\log_2(|X|)\rceil$. As in the base protocol [16, Apdx. A.2.2], define $\chi_0(\tau) = 1-\tau$ and $\chi_1(\tau) = \tau$. Then $\tilde{V}_d$ can be written as follows:

$$\tilde{V}_d(\tau_1,\ldots,\tau_n) = \sum_{(t_1,\ldots,t_n)\in\mathbb{F}_2^n}\left(\prod_{i=1}^{n}\chi_{t_i}(\tau_i)\right)\cdot V_d(t_1,\ldots,t_n),$$

where V_d is defined in Section 2.2. Naive evaluation of this sum requires $2^n\cdot(n+1)$ multiplications. However, notice that if we have the value of $\prod_{i=1}^{k-1}\chi_{t_i}(\tau_i)$ for all $(t_1,\ldots,t_{k-1})\in\mathbb{F}_2^{k-1}$, then computing $\prod_{i=1}^{k}\chi_{t_i}(\tau_i)$ for all $(t_1,\ldots,t_k)\in\mathbb{F}_2^k$ requires 2^k multiplications. Thus, computing $\prod_{i=1}^{n}\chi_{t_i}(\tau_i)$ for all $(t_1,\ldots,t_n)\in\mathbb{F}_2^n$ takes $\sum_{i=1}^{n}2^i < 2\cdot 2^n$ multiplications. After we have computed all of these values, computing $\tilde{V}_d$ requires only an additional 2^n multiplications, bringing us to $3\cdot 2^n$ multiplications. Since $|X|\le 2^n < 2\cdot|X|$, the total cost to compute $\tilde{V}_d$ is between $3\cdot|X|$ and $6\cdot|X|$ multiplications. By skipping the computation of $\prod_{i=1}^{n}\chi_{t_i}(\tau_i)$ in the cases where $V_d(t_1,\ldots,t_n)=0$ (e.g., when $(t_1,\ldots,t_n)$ is not a valid input gate label), we can bring the upper bound down to $4\cdot|X|$.

[7] This optimization does not apply in the streaming context, which was the original focus of CMT [16].

5.2 CMT-batching

Using the Zaatar compiler [44], we implemented a backend compiler that targets the CMT-batching protocol, as described in Section 4. Specifically, our compiler takes as input a set of constraints generated by the Zaatar compiler and converts them to a circuit of the required form. This compiler contains 2381 lines of C++ code.

5.3 Compiler and static analyzer

Using static analysis, Allspice selects the optimal protocol among CMT-batching, Ginger, and Zaatar. The analysis tool is a 300-line Python script that takes as input a program expressed in SFDL (§2.3), the size of the program's inputs, microbenchmark information about the costs of various low-level operations (e.g., encryption and decryption), and various cost models (developed in [44, 47] and Section 4). It can be configured to optimize for the prover's overhead or for the verifier's costs (resulting in lower break-even batch sizes).

The tool first runs the Ginger compiler to get Ginger constraints, the Zaatar compiler to get Zaatar constraints, and the CMT-batching compiler to get an arithmetic circuit (see §5.2 above). Then, it parameterizes the cost models with the microbenchmarks and applies the resulting quantified model to the compiler's output in order to predict the costs to the verifier and prover under each sub-protocol at the requested input size. Finally, the tool chooses the most favorable sub-protocol for the consideration in question (e.g., break-even batch size), and outputs executables for the verifier and the prover in the chosen protocol.

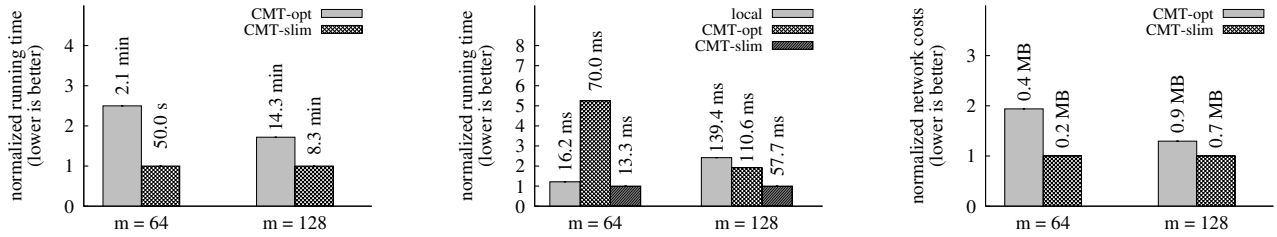
6 Experimental evaluation and comparison

We experiment with the base CMT protocol and CMT-slim to evaluate our refinements to CMT. Then, we experimentally evaluate Allspice with a set of benchmark problems to understand the regimes of applicability of each of the sub-protocols (namely Ginger, Zaatar, and CMT-batching). Figure 5 summarizes the results.

6.1 Method and setup

Our evaluation uses the following benchmark computations:

- $m \times m$ matrix multiplication,
- evaluation of a degree-2 polynomial in m variables,



(a) Per-instance running time of the prover (b) Per-instance running time of the verifier. (c) Per-instance network costs.

Figure 6—Per-instance running times and network costs of CMT-opt and CMT-slim, for matrix multiplication at m=64 and m=128. CMT-slim improves the CPU and network costs versus CMT-opt. The bar heights are normalized to CMT-slim.

- the longest common subsequence problem with two strings of length m,
- clustering a set of m data points, where each data point has d dimensions, using PAM clustering,
- finding the roots of a degree-2 polynomial in m variables using bisection,
- Floyd-Warshall all-pairs shortest paths in a graph with m vertexes, and
- Hamming distance between a pair of strings of length m.

The inputs to our computations are 32-bit integers except in polynomial evaluation, root finding by bisection, and all-pairs shortest paths. For polynomial evaluation and all-pairs shortest paths, we use floating-point rationals with 32-bit numerators and 32-bit denominators; for root finding by bisection, we use floating-point rationals with 32-bit numerators and 5-bit denominators (using the representation of this data type from [47]). We use a finite field with a prime modulus of size 128 bits for the integer computations and size 220 bits for the floating-point computations.

For Ginger [47] and Zaatar [44], we use ElGamal encryption [17] with key size of 1024 bits. For a pseudorandom generator, we use the ChaCha stream cipher [11].

We will report the prover's CPU costs, network costs, and *break-even batch sizes*: the minimum number of instances of a computation at which the total CPU cost to the verifier is less than the CPU cost to execute those instances locally. Our baseline for local computation is the GMP library. We compute the break-even batch size for a computation by running one instance locally (regarding all costs as variable) and one instance under verification (separating its costs into fixed and variable); we then solve for the point β at which β instances of the latter are cheaper than β instances of the former.

Our experiments run on two machines in the same local area network. Each has an Intel Xeon processor E5540 2.53 GHz with 48GB of RAM.[8] We use `getrusage` to measure the CPU time of the prover and the verifier; we also record the count of the application-level bytes transferred between the verifier and the prover, reporting this count as the network costs. We report the mean of results from three experiments (standard deviations are less than 10% of the means).

An exception to the previous paragraph is that it is sometimes infeasible to run with Ginger; in those cases (we will note them), we estimate costs using a model from [46], parameterized with microbenchmark results from [44, §5.1].

[8] Our largest experiments require approximately 40% of the available RAM.

6.2 Performance of CMT and CMT-slim

Our analysis in Section 3 (see Figure 3) implies that, relative to CMT, CMT-slim saves an additive cost. This cost is proportional to the number of outputs in the computation and the relative size of the comparison circuit that is removed.

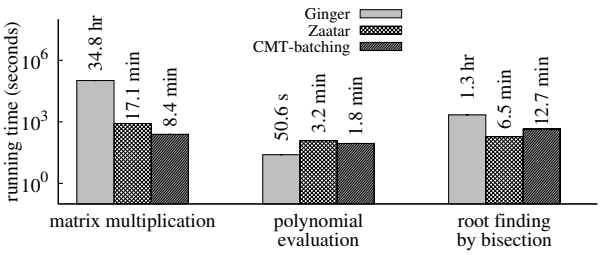
We confirm this analysis by measuring CMT-opt (an optimized version of the CMT-GMP implementation that we use in [47], which is about 30–40% faster) and CMT-slim on the example of matrix multiplication with varying input sizes (m=64, 128). We report the prover's running time, the verifier's running time, the baseline cost of doing the computation locally, and the network costs. The running times are CPU costs and ignore protocol latency.

Figure 6 depicts the results. At m=64, the CMT-slim prover is about 2× less expensive compared to CMT-opt, the verifier is about 5× less expensive, and the network costs halve. As we increase input sizes, the gains diminish; at m=128, the CMT-slim prover is 1.7× less expensive, the verifier is 2× less expensive, and the network costs drop by about 30%. This phenomenon occurs because the size of the compute circuit (§3) grows with m^3, while the size of the comparison circuit grows only with m^2; thus, asymptotically, the time spent checking the compute circuit dominates the time spent checking the comparison circuit.

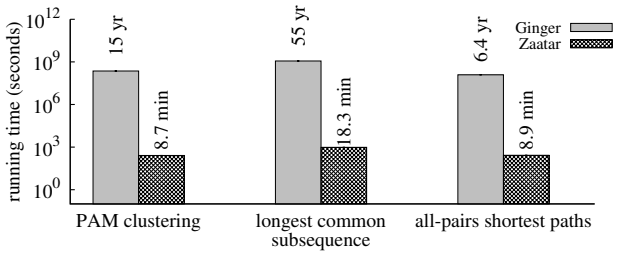
We also experimented with Hamming distance (at input sizes m=32,768 and m=65,536). Because the output is a scalar, not a vector, the comparison optimization does not save work, so CMT-opt and CMT-slim perform the same.

6.3 Comparison

We experiment with the computations listed in Section 6.1. We run matrix multiplication with m=128; polynomial evaluation with m=512; root finding by bisection with m=256, L=8; PAM clustering with m=20, d=128; longest common subsequence (LCS) with m=300; and all-pairs



(a) Per-instance running time of the prover under Allspice's sub-protocols for matrix multiplication (m=128), polynomial evaluation (m=512), and root finding by bisection (m=256, L=8). The y-axis is log-scaled.



(b) Per-instance running time of the prover under Ginger and Zaatar, for PAM clustering (m=20, d=128), longest common subsequence (m=300), and all-pairs shortest paths (m=25). The y-axis is log-scaled.

Figure 7—When applicable (i.e., when the verifier has a break-even batch size), the CMT-batching prover's performance is similar to the Zaatar prover's and significantly better than the Ginger prover's (except for the polynomial evaluation benchmark, in which the Ginger prover is faster, but the CMT-batching prover is still competitive). When CMT-batching is not applicable, Zaatar's prover dominates.

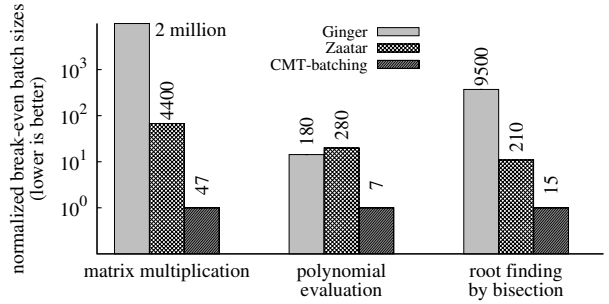


Figure 8—Break-even batch sizes for matrix multiplication (m=128), polynomial evaluation (m=512), and root finding by bisection (m=256, L=8) under the three sub-protocols. The bar heights are normalized to those of CMT-batching (an off-scale bar is truncated at the maximum height). The y-axis is log-scaled. Whenever CMT-batching enables breaking even, it has the lowest break-even batch size; otherwise, Zaatar does (this is not depicted but applies to PAM clustering, LCS, and all-pairs shortest paths).

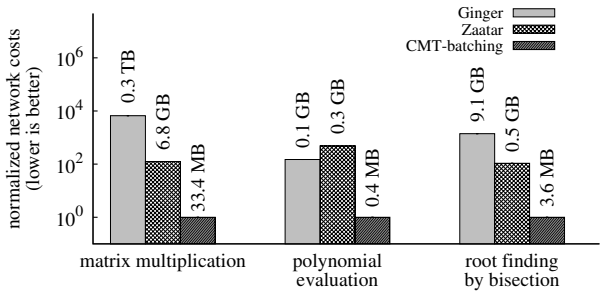


Figure 9—Total network costs of each of the sub-protocols in Allspice at the break-even batch sizes (Figure 8) for matrix multiplication (m=128), polynomial evaluation (m=512), and root finding by bisection (m=256, L=8). The bar heights are normalized to those of CMT-batching. The y-axis is log-scaled. When applicable (i.e., when the verifier has a break-even batch size), CMT-batching has the lowest network costs; otherwise, Zaatar does (this is not depicted but applies to PAM clustering, LCS, and all-pairs shortest paths).

shortest paths with m=25. We report three end-to-end performance metrics: the prover's per-instance costs, the break-even batch sizes, and network costs.

First, note that the verification protocols save work for the verifier only if the computation is at least superlinear; we therefore exclude Hamming distance here.

For all of the remaining benchmarks, the Zaatar verifier saves work at some batch size. The CMT-batching verifier, however, saves work only for some of our benchmarks (matrix multiplication, polynomial evaluation, and root finding by bisection); for the others (PAM clustering, longest common subsequence, and all-pairs shortest paths), the CMT-batching verifier does not save work, as the prover supplies a quantity of auxiliary inputs that is proportional to the running time of the computation (see Section 4).

We experiment with all three protocols when appropriate and exclude CMT-batching when it does not break even. In the case of Ginger, we estimate its costs for matrix multiplication and root finding by bisection, and measure its costs for polynomial evaluation.

For computations where CMT-batching is appropriate, Figure 7(a) depicts the per-instance running times of each sub-protocol's prover normalized to the running time of the CMT-batching prover. The prover's overhead in the CMT-batching protocol is roughly the same (within several factors) as that of the most optimal prover. For the remaining computations, Figure 7(b) depicts the per-instance running times of each prover normalized to the running time of the Zaatar prover. In these experiments, Zaatar dominates Ginger.

Figure 8 depicts the break-even batch sizes. Whenever CMT-batching is applicable, it has the best break-even batch size. Again, Zaatar dominates Ginger in all cases (this is not depicted, but see [44]).

At the break-even batch sizes, the total network costs (including the cost of transferring inputs and outputs) follow the same pattern as the break-even batch sizes; see Figure 9. When CMT-batching is applicable, it has the lowest network costs among the three sub-protocols; when it is not applicable, Zaatar does (not depicted; see [44]).

7 Discussion

Based on the estimates (which are validated by our measurements), Allspice will choose CMT-batching for matrix multiplication, polynomial evaluation, and root finding; it will choose Zaatar for everything else.

It is useful to extract the general moral here: namely, CMT-batching is superior for quasi-straight-line computations (i.e., very simple control flow), while for general-purpose computations, one has to pay for cryptography and use Zaatar.

Of course, the term "general-purpose" should be taken with a grain of salt. Although SFDL is a high-level language, the underlying constraint and circuit formalisms impose a number of unattractive limitations. Loop bounds need to be known at compile time (as loops are unrolled), both branches of conditionals are executed, and dynamic array access is expensive. Moreover, since the base protocols operate over finite fields, floating-point operations are expensive and do not implement IEEE rounding. These problems are pervasive in the area, as finite fields and circuit models of computation are standard in interactive and cryptographic protocols.

Finally, we note that the compilation process in Allspice is quite expensive; the examples we reported on took hours to compile. Although some of this is intrinsic, we believe that much of the cost is a function of an unoptimized implementation; optimization plus techniques described elsewhere [26, 30] should lead to significant improvements.

8 Other related work

We now review recent cryptographic and complexity-theoretic work on verifiable computation that we have not yet discussed. Some work targets particular classes of computations [18, 38], including matrix operations and polynomial evaluation; for a survey, see [45]. While special-purpose protocols may be more efficient for particular computations, we are interested in mechanisms that apply generally (§1). Moreover, special-purpose protocols that admit highly efficient implementations and are amenable to automatic compilation can be incorporated into our sub-protocol selection logic (§5.3).

The general-purpose protocols can be roughly classified into three groups: fully-homomorphic encryption (FHE), short PCPs, and succinct arguments. Using FHE [21] as a building block, GGP [19] formalized *verifiable computation* (VC) as non-interactive protocols ($\mathcal{P}$ sends its output and a proof to $\mathcal{V}$) that are permitted a one-time setup cost. FHE itself was woefully impractical (and unfortunately still is, despite massive improvements [22]), a characteristic inherited by GGP and other noninteractive VC protocols based on FHE [15]. In this paper, we adopt a broader notion of verifiable computation that includes interactive protocols, consistent with informal motivation in the complexity theory literature (e.g., [8]).

Indeed, complexity theory has long used "checking computations" as a motivation for PCPs, and recent theoretical work aims to make PCPs and related constructs efficient enough to be used in practice [9, 10]. However, it is not clear from the analysis how this work will actually perform, and to date there are no experimental results, so it is not yet possible to perform a meaningful comparison.

	Zaatar	Pinocchio [20, 39]
$\mathcal{P}$'s overhead	$(2e + 500f) \cdot \lvert\mathcal{C}\rvert$	$8e \cdot \lvert\mathcal{C}\rvert$
$\mathcal{V}$'s per-instance costs	$e + 3f \cdot (\lvert x\rvert + \lvert y\rvert)$	$12e + e \cdot (\lvert x\rvert + \lvert y\rvert)$
Setup costs	$(2e + 500f) \cdot \lvert\mathcal{C}\rvert$	$8e \cdot \lvert\mathcal{C}\rvert$
Setup amortizes over	batch	all instances of Ψ
Setup flexibility	$\mathcal{V}$ must incur	anyone can incur
Non-interactivity	no	yes
Zero-knowledge	no	yes
Public verifiability	no	yes
Assumptions	DDH (and R.O. to save network costs)	q-PKE, q-PDH, q-SDH
Soundness error	$< 1/2^{20}$	$< 1/2^{128}$

Figure 10—Comparison of Zaatar and Pinocchio [20, 39] for a computation Ψ with $\lvert\mathcal{C}\rvert$ constraints. The comparison is meant to be illustrative not definitive: e stands in for any ciphertext operation in the respective protocol (e.g., exponentiation in a bilinear group, scalar multiplication in an ElGamal group), and f stands in for any plaintext operation (e.g., field multiplication). In the protocols that we have examined, e is several orders of magnitude larger than f. Broadly, Pinocchio has higher cryptographic overhead, in return for more properties.

The final strand of general-purpose work is succinct *non*-interactive arguments [25, 31]. A major advance in this area was recently achieved by GGPR [20], who give an efficient circuit encoding (QAPs) that induces roughly *linear* overhead for the prover and couple it with sophisticated cryptography. Although GGPR does not explore the connection between QAPs and PCPs, the relationship is explored in [12] and [44]. Zaatar exploits this connection by incorporating QAPs (as noted in Section 2.3) but not GGPR's cryptography.

Thus, a natural question is how Allspice (via Zaatar) compares to GGPR. We now compare Zaatar to a recent refinement and implementation of GGPR, called Pinocchio [39]. To simplify a bit, whereas most Zaatar queries are formulated and responded to in the clear, Pinocchio uses carefully wrought cryptography to allow a similar query procedure to take place over *encrypted* values. On the one hand, this brings some expense (small constant factors), in terms of the prover's costs and the verifier's per-instance costs (although heroic optimizations have resulted in surprisingly small overhead for the cryptography in Pinocchio). On the other hand, the cryptography buys public verifiability, zero-knowledge proofs, and unlimited query reuse. This latter property allows Pinocchio to amortize its per-computation setup costs over all instances of the computation; by contrast, Zaatar amortizes its per-computation costs only over a batch. We give more details in Figure 10.

9 Conclusion

This paper describes Allspice, a system for verifiable computation. Allspice incorporates substantial improvements to CMT [16, 24, 41]. Optimizations to the basic CMT protocol coupled with an extension to a broader class of computations (in the batched model) result in a version of CMT that applies to many realistic computations and requires break-even batch sizes several orders of magnitude smaller than other measured implementations of verifiable computation.

The batched version of CMT does not require expensive cryptography. Furthermore, the batched model and the class of computations to which CMT-batching apply are very plausible in the cloud computing scenario—the MapReduce idiom in effect forces programmers to express computations in a form that is well-suited for this protocol!

Further, using the careful cost analysis that accompanied the optimization of CMT, we were able to build a static analyzer that allows Allspice to select the best protocol among the implemented protocols in the literature (Ginger, Zaatar, and CMT). The resulting system brings us a step closer to practical and usable system for verifiable computation in the data-parallel cloud computing model.

A CMT: Details and refinements

This appendix provides further details of CMT and proves the correctness of our refinements. We also provide general expressions for $\tilde{\text{add}}_i$ and $\tilde{\text{mult}}_i$, for use in CMT-batching (§4).

A.1 Original CMT

In this section, we fill in details of the CMT protocol that were left out of Section 2.2. We give the verifier's sum-check pseudocode and provide the missing definition of $\tilde{E}$.

The Sum-Check Verifier. Recall that in each round of CMT, $\mathcal{V}$ uses a sum-check protocol to verify that $a_{i-1} = \tilde{V}_{i-1}(q_{i-1})$. Figure 11 depicts the verifier's pseudocode for doing so. There is one major difference between this protocol and a standard sum-check protocol [4, §8.3.1]. A standard sum-check protocol requires $\mathcal{V}$ to evaluate $f_{q_{i-1}}$ at a single point (w_0, w_1, w_2), randomly chosen from $\mathbb{F}^{3b}$. However, that would require evaluating $\tilde{V}_i(w_1)$ and $\tilde{V}_i(w_2)$, which $\mathcal{V}$ cannot do because it does not know the values of the gates at layer i. Instead, $\mathcal{V}$ "cheats" and asks $\mathcal{P}$ for the values of $\tilde{V}_i(w_1)$ and $\tilde{V}_i(w_2)$. This "hint"—together with $\mathcal{V}$'s computation of $\tilde{\text{add}}(w_0, w_1, w_2)$, $\tilde{\text{mult}}(w_0, w_1, w_2)$, and $\tilde{E}(q_{i-1}, w_0)$—is enough for $\mathcal{V}$ to compute $f_{q_{i-1}}(w_0, w_1, w_2) = \tilde{E}(q_{i-1}, w_0) \cdot (\tilde{\text{add}}_i(w_0, w_1, w_2) \cdot (\tilde{V}_i(w_1) + \tilde{V}_i(w_2)) + \tilde{\text{mult}}_i(w_0, w_1, w_2) \cdot \tilde{V}_i(w_1) \cdot \tilde{V}_i(w_2))$.

We defined $\tilde{\text{add}}_i$ and $\tilde{\text{mult}}_i$ in Section 2.2; we now define $\tilde{E}$. The polynomial $\tilde{E}\colon \mathbb{F}^{2b} \to \mathbb{F}$ is the multilinear extension of the function $E\colon \mathbb{F}_2^{2b} \to \mathbb{F}$, where

$$E(q, g) = \begin{cases} 1, & \text{if } q = g \\ 0, & \text{otherwise} \end{cases}$$

```
1:  function SumCheck(i, q_{i-1}, a_{i-1})
2:      r ←R 𝔽^{3b}
3:      SendToProver("layer i")
4:      e ← a_{i-1}
5:      for j = 1, 2, ..., 3b do
6:          F_j(·) ← GetFromProver()
7:          // P returns F_j(·) as {F_j(0), F_j(1), F_j(2)}, which
8:          // is sufficient to reconstruct F_j because F_j is degree-2
9:
10:         if F_j(0) + F_j(1) ≠ e then
11:             return reject
12:
13:         SendToProver(r_j)
14:         e = F_j(r_j)
15:
16:     // notation
17:     w_0 ← (r_1, ..., r_b)
18:     w_1 ← (r_{b+1}, ..., r_{2b})
19:     w_2 ← (r_{2b+1}, ..., r_{3b})
20:
21:     // P is supposed to set v_1 = Ṽ_i(w_1) and v_2 = Ṽ_i(w_2)
22:     v_1, v_2 ← GetFromProver()
23:     a' ← Ẽ(q_{i-1}, w_0)(add̃(w_0, w_1, w_2) · (v_1 + v_2)+
24:                          mult̃(w_0, w_1, w_2) · v_1 · v_2)
25:     if a' ≠ e then
26:         return reject
27:     return (v_1, v_2, w_1, w_2)
```

Figure 11—$\mathcal{V}$'s side of the sum-check protocol in CMT [16] and GKR [23, 41], for $\tilde{V}_{i-1}(q_{i-1}) = \sum_{g,j_1,j_2 \in \mathbb{F}_2^b} f_{q_{i-1}}(g, j_1, j_2)$; see equation (2) in Section 2.2. $b = \lceil \log_2 G \rceil$ is the number of bits needed to represent a gate label. The inputs are i (the current layer number), a_{i-1}, and q_{i-1}; the protocol returns (v_1, v_2, w_1, w_2). The protocol's guarantee is that, with high probability, $a_{i-1} = \tilde{V}_{i-1}(q_{i-1})$ if and only if $v_1 = \tilde{V}_i(w_1)$ and $v_2 = \tilde{V}_i(w_2)$.

We can write a closed-form expression for $\tilde{E}$:

$$\tilde{E}(q_1, \ldots, q_b, g_1, \ldots, g_b) = \prod_{j=1}^{b} ((1 - q_j) \cdot (1 - g_j) + q_j \cdot g_j).$$

To see that this is the expression for $\tilde{E}$, note that all of the $2b$ variables in this expression have degree 1. Also, assume $q, g \in \mathbb{F}_2^b$. Then the expression equals 1 if and only if $q_j = g_j$ for all $j \in \{1, \ldots, b\}$ (i.e., if and only if $q = g$); otherwise, the expression equals 0. Since the expression is of appropriate degree, and since it agrees with E on E's domain, it must equal $\tilde{E}$.

The Sum-Check Prover. In the sum-check protocol, $\mathcal{P}$'s fundamental job is to compute evaluations of $F_j(\cdot)$ at points in $\{0, 1, 2\}$ (Figure 11, line 6), and to do so efficiently. Cormode et al. [16, Apdx. A.2] describe how to make $\mathcal{P}$ run in time $O(G \cdot \log G)$.

We further improve their scheme. Notice that in iteration j of the sum-check protocol, $\mathcal{P}$ must return $(F_j(0), F_j(1), F_j(2))$. However, $\mathcal{P}$ knows that $F_j(0) + F_j(1) = F_{j-1}(r_{j-1})$. Thus, $\mathcal{P}$ computes $F_j(1)$ and $F_j(2)$; it also com-

putes $F_{j-1}(r_{j-1})$ by interpolating a degree-2 polynomial from the values $F_{j-1}(0), F_{j-1}(1), F_{j-1}(2)$, which it has computed in iteration $j-1$. Finally, $\mathcal{P}$ computes $F_j(0) = F_{j-1}(r_{j-1}) - F_j(1)$. This process, including the interpolation, is much more efficient than computing $F_j(0)$ directly.

A.2 Correctness of the refinements

In this section, we prove the correctness (soundness and completeness) of CMT-slim (§3) and CMT-batching (§4).

Lemma A.1. CMT-slim is complete and has soundness error $\epsilon + b_0/|\mathbb{F}|$, where $b_0 = \lceil \log_2 |Y| \rceil$ and ϵ is the soundness error of CMT (which is quantified in Section 2.2).

Proof. Let $\tilde{V}_0(\cdot)$ represent the extension of V_0 for the circuit, if the circuit were computed correctly.

Completeness: If $y = C(x)$, then $\tilde{V}_y = \tilde{V}_0$ (by definition). Thus, $a_0 = \tilde{V}_y(q_0) = \tilde{V}_0(q_0)$ for all $q_0 \in \mathbb{F}^{b_0}$. Since $a_0 = \tilde{V}_0(q_0)$ is a true statement, $\mathcal{P}$ need only follow the protocol to ensure that $\mathcal{V}$ accepts. The protocol thus satisfies (perfect) completeness.

Soundness: If $y \neq C(x)$, then $V_y \neq V_0$ (they differ on at least one point in their domains). Thus, $\tilde{V}_y$ and $\tilde{V}_0$ are not the same polynomial, and in fact $\tilde{V}_y(q) = \tilde{V}_0(q)$ for at most a $b_0/|\mathbb{F}|$ fraction of the values of $q \in \mathbb{F}^{b_0}$. (This is because $\tilde{V}_0$ and $\tilde{V}_y$ are polynomials in b_0 variables, where each variable has degree at most 1 in any term; thus, we can apply the Schwartz-Zippel lemma.) If $\mathcal{V}$ chooses a q_0 for which $\tilde{V}_y(q_0) = \tilde{V}_0(q_0)$, then the protocol incorrectly accepts. If $\mathcal{V}$ chooses a q_0 for which $\tilde{V}_y(q_0) \neq \tilde{V}_0(q_0)$, then the protocol is attempting to prove a false statement, namely that $a_0 = \tilde{V}_0(q_0)$. By the soundness of CMT, this succeeds with probability $\leq \epsilon$. A standard union bound establishes the claimed error. □

The completeness of CMT-batching follows from CMT-slim's completeness: if the prover returns the correct answers for all instances in the batch, the verifier (which can be seen as β instances of the CMT-slim verifier that all make the same random choices) will always accept. We now prove the soundness of CMT-batching.

Lemma A.2. Fix a circuit C, batch size β, inputs $x^{(1)}, \ldots, x^{(\beta)}$, and purported outputs $y^{(1)}, \ldots, y^{(\beta)}$. For any instance i, if $y^{(i)} \neq C(x^{(i)})$, then the probability of the verifier accepting that instance is $\leq \epsilon'$, where ϵ' is the soundness of CMT-slim.

Proof. We prove this lemma by reducing CMT-batching to CMT-slim.

Suppose toward a contradiction that CMT-batching has soundness error $> \epsilon'$. This means that there exists some circuit C, some set of inputs $x^{(1)}, \ldots, x^{(\beta)}$, and a prover $\mathcal{P}^*$ for CMT-batching such that, when given $x^{(1)}, \ldots, x^{(\beta)}$ as inputs, $\mathcal{P}^*$ returns $y^{(i)} \neq C(x^{(i)})$, yet the verifier $\mathcal{V}'$ still accepts with probability $> \epsilon'$. Assume WLOG that $i = 1$.

We construct a CMT-slim prover, $\mathcal{P}$, for the circuit C as follows:

1. At the beginning, $\mathcal{P}$ receives x from $\mathcal{V}$, the CMT-slim verifier. It sends $(x, x^{(2)}, \ldots, x^{(\beta)})$ to $\mathcal{P}^*$, getting back $(y, y^{(2)}, \ldots, y^{(\beta)})$. It sends y to $\mathcal{V}$.
2. In each round, $\mathcal{P}$ receives a message from $\mathcal{V}$. It forwards the message to $\mathcal{P}^*$ and receives β responses. It sends the response from $\mathcal{P}^*$ for instance 1 to $\mathcal{V}$.

We want to show that when $y \neq C(x)$, $\mathcal{P}$ can get $\mathcal{V}$ to accept with probability $> \epsilon'$ (thus contradicting the soundness of CMT-slim).

Recall that, while the CMT-batching verifier $\mathcal{V}'$ shares the same randomness between instances, it otherwise treats all instances independently when verifying. Thus, we can think of $\mathcal{V}'$ as a collection of β CMT-slim verifiers that all generate the same randomness. $\mathcal{V}'$ accepts for iteration i if and only if the ith CMT-slim verifier accepts. Since $\mathcal{P}^*$ can get the 1st CMT-slim verifier in $\mathcal{V}'$ to accept with probability more than $> \epsilon'$ (by our assumption), $\mathcal{P}$ can get $\mathcal{V}$ to accept with the same probability, and we get the desired contradiction. □

A.3 General expressions for $\tilde{\text{mult}}$ and $\tilde{\text{add}}$

This section describes general expressions for $\tilde{\text{mult}}$ and $\tilde{\text{add}}$ that can be applied to every layer.

Let $S_{\text{add}_i} = \{(g_1, \ldots, g_{3b}) \in \mathbb{F}_2^{3b} \mid \text{add}_i(g_1, \ldots, g_{3b}) = 1\}$ and similarly for S_{mult_i}. Furthermore, let $\chi_0(\tau) = 1 - \tau$ and $\chi_1(\tau) = \tau$. The general expressions for $\tilde{\text{add}}_i$ and $\tilde{\text{mult}}_i$ are

$$\tilde{\text{add}}_i(\tau_1, \ldots, \tau_{3b}) = \sum_{(g_1,\ldots,g_{3b}) \in S_{\text{add}_i}} \prod_{j=1}^{3b} \chi_{g_j}(\tau_j), \qquad \text{and}$$

$$\tilde{\text{mult}}_i(\tau_1, \ldots, \tau_{3b}) = \sum_{(g_1,\ldots,g_{3b}) \in S_{\text{mult}_i}} \prod_{j=1}^{3b} \chi_{g_j}(\tau_j).$$

Note that if $(\tau_1, \ldots, \tau_{3b}) \in \mathbb{F}_2^{3b}$, then $\prod_{j=1}^{3b} \chi_{g_j}(\tau_j) = 1$ if $(\tau_1, \ldots, \tau_{3b}) = (g_1, \ldots, g_{3b})$ and equals 0 otherwise. Thus, $\tilde{\text{add}}_i(\tau_1, \ldots, \tau_{3b}) = 1$ if $(\tau_1, \ldots, \tau_{3b}) \in S_{\text{add}_i}$ and equals 0 otherwise (similarly for $\tilde{\text{mult}}_i$ and S_{mult_i}). These expressions agree with add_i and mult_i wherever add_i and mult_i are defined, and they are polynomials of appropriate degree. Thus, they must equal $\tilde{\text{add}}_i$ and $\tilde{\text{mult}}_i$.

Acknowledgments

Productive conversations with Justin Thaler, and careful reading by Justin and Bryan Parno improved our thinking and this draft. We thank Zuocheng Ren for helpful suggestions and the anonymous reviewers for their helpful comments. The research was supported by AFOSR grant FA9550-10-1-0073, NSF grants 1055057 and 1040083, a Sloan Fellowship, and an Intel Early Career Faculty Award.

References

[1] Berkeley Open Infrastructure for Network Computing (BOINC).

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

[3] 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.

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

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

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

[7] L. Babai. Trading group theory for randomness. In *STOC*, 1985.

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

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

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

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

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

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

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

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

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

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

[18] D. Fiore and R. Gennaro. Publicly verifiable delegation of large polynomials and matrix computations, with applications. In *ACM CCS*, 2012.

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

[20] R. Gennaro, C. Gentry, B. Parno, and M. Raykova. Quadratic span programs and succinct NIZKs without PCPs. In *EUROCRYPT*, May 2013.

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

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

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

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

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

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

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

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

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

[30] B. Kreuter, abhi shelat, and C. hao Shen. Billion-gate secure computation with malicious adversaries. In *USENIX Security*, Aug. 2012.

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

[32] C. Lund, L. Fortnow, H. J. Karloff, , and N. Nisan. Algebraic methods for interactive proof systems. *J. of the ACM*, 39(4):859–868, 1992.

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

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

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

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

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

[38] C. Papamanthou, E. Shi, and R. Tamassia. Signatures of correct computation. In *IACR TCC*, Mar. 2013.

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

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

[41] G. N. Rothblum. *Delegating Computation Reliably: Paradigms and Constructions*. PhD thesis, Massachusetts Institute of Technology, 2009.

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

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

[44] S. Setty, B. Braun, V. Vu, A. J. Blumberg, B. Parno, and M. Walfish. Resolving the conflict between generality and plausibility in verified computation. In *EuroSys*, Apr. 2013.

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

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

[47] S. Setty, V. Vu, N. Panpalia, B. Braun, A. J. Blumberg, and M. Walfish. Taking proof-based verified computation a few steps closer to practicality. In *USENIX Security*, 2012.

[48] A. Shamir. IP = PSPACE. *J. of the ACM*, 39(4):869–877, 1992.

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