vRAM: Faster Verifiable RAM With Program-Independent Preprocessing

Yupeng Zhang[*], Daniel Genkin[†,*], Jonathan Katz[*], Dimitrios Papadopoulos[‡,*] and Charalampos Papamanthou[*]
[*]University of Maryland [†]University of Pennsylvania [‡]Hong Kong University of Science and Technology
Email: {zhangyp,cpap}@umd.edu, danielg3@cis.upenn.edu, jkatz@cs.umd.edu, dipapado@cse.ust.hk

***Abstract*—We study the problem of verifiable computation (VC) for RAM programs, where a computationally weak verifier outsources the execution of a program to a powerful (but untrusted) prover. Existing efficient implementations of VC protocols require an expensive preprocessing phase that binds the parties to a single circuit. (While there are schemes that avoid preprocessing entirely, their performance remains significantly worse than constructions with preprocessing.) Thus, a prover and verifier are forced to choose between two approaches: (1) Allow verification of arbitrary RAM programs, at the expense of efficiency, by preprocessing a universal circuit which can handle all possible instructions during each CPU cycle; or (2) Sacrifice expressiveness by preprocessing an efficient circuit which is tailored to the verification of a single specific RAM program.**

We present *vRAM*, a VC system for RAM programs that avoids both the above drawbacks by having a preprocessing phase that is entirely *circuit-independent* (other than an upper bound on the circuit size). During the proving phase, once the program to be verified and its inputs are chosen, the circuit-independence of our construction allows the parties to use a smaller circuit tailored to verifying the specific program on the chosen inputs, i.e., without needing to encode all possible instructions in each cycle. Moreover, our construction is the first with asymptotically *optimal* prover overhead; i.e., the work of the prover is a constant multiplicative factor of the time to execute the program.

Our experimental evaluation demonstrates that vRAM reduces the prover's memory consumption by 55–110× and its running time by 9–30× compared to existing schemes with universal preprocessing. This allows us to scale to RAM computations with more than 2 million CPU cycles, a 65× improvement compared to the state of the art. Finally, vRAM has performance comparable to (and sometimes better than) the best existing scheme with program-specific preprocessing despite the fact that the latter can deploy program-specific optimizations (and has to pay a separate preprocessing cost for every new program).

I. Introduction

Protocols for *verifiable computation (VC)* allow a computationally weak verifier to outsource the execution of a program to a powerful but untrusted prover (e.g., a cloud provider) while being assured that the result was computed correctly. Somewhat more formally, a verifier $\mathcal{V}$ and prover $\mathcal{P}$ agree on a function f and an input x. The prover then sends a result y to the verifier, together with a proof that $y = f(x)$. There is a long line of work constructing VC protocols for arbitrary computations, the most prominent of which rely on succinct non-interactive arguments of knowledge (SNARKs) [12], [25]. This has resulted in several implemented systems; see Section I-C for an overview. While VC protocols without preprocessing have been recently implemented [7], efficient VC implementations still rely on a preprocessing phase during which a trusted party (possibly the verifier) generates a set of public parameters corresponding to a specific circuit for the function f. Furthermore, this preprocessing phase is orders of magnitude slower than evaluating f itself.

Verifying RAM Programs. While circuits can model arbitrary programs, most real-world computations are expressed in terms of random-access memory (RAM) machines. This is true both in terms of most programmers' mental model of computation, as well as in terms of the execution of assembly code on general-purpose computers. However, since most constructions of VC protocols work on computations expressed as arithmetic circuits, verification of a RAM program P is usually done by verifying the correct evaluation of an arithmetic circuit C_P that corresponds to the next-instruction function of the RAM program while checking consistency of memory, etc. As stated above, most VC implementations require the circuit to be fixed ahead of time, during a trusted preprocessing phase. Due to this, previous works for verifying RAM programs can be roughly divided into two main categories.

1) **Program-Specific Preprocessing.** If the program P to be verified is known ahead of time, it is possible to tailor the circuit C_P so as to verify P as efficiently as possible. While this tailoring is beneficial to the protocol's overall performance, it comes at the expense of usability since C_P cannot be used to verify another program P'. Examples of this approach are Pantry [17] and Buffet [50].
2) **Universal Preprocessing.** In case the RAM program to be verified is not known ahead of time, it is possible to construct a universal circuit C_{RAM} which is capable of verifying any RAM program that runs for at most T steps. Examples of this approach include [9], [11].

Both these approaches have significant drawbacks. In the first case, the verifier cannot change the RAM program P being verified without re-running the (expensive) preprocessing phase. This is a major drawback as the preprocessing cost can only be amortized by running the same program on different inputs.

In the second case, although the preprocessing cost can be amortized over the evaluation of different programs on different inputs, the universal preprocessing used in this approach imposes large concrete overheads during the proving phase. This results from the fact that C_{RAM} must be able to emulate all possible operations at every CPU step in order to handle arbitrary RAM programs. In contrast, the program-specific approach benefits from the fact that P is known when C_P is chosen, and so the set of possible instructions at each step is potentially much smaller.

Two notable exceptions to the above are the works of [7], [10], which do not need a preprocessing phase tied to a specific circuit. However, the concrete cost of these systems remains significantly higher than that of the preprocessing-based solutions mentioned above. (See Section V-C.)

Thus, in this paper we thus ask the following question: *Is it possible to construct a VC protocol for RAM programs which has similar (or even better) performance than what is achievable with program-specific preprocessing, but without knowing the program during the preprocessing phase?*

A. Our Results

We answer the above question in the affirmative by presenting *vRAM*, a VC protocol for RAM programs that improves the performance of previous works both concretely and asymptotically. In particular, our system achieves performance similar to (and often better than) state-of-the art systems with program-specific preprocessing, but without requiring the RAM program to be fixed during the preprocessing phase. This allows a single execution of the preprocessing phase to be used for verifying arbitrary RAM programs (running for some bounded number of steps) afterwards.

Our starting point is vSQL [52], a system for verifying SQL queries on outsourced databases. While not presented as such, vSQL can be viewed as a VC scheme that has a preprocessing phase that does not depend on a specific circuit beyond an upper bound on the circuit's input size. vRAM relies on two novel improvements to vSQL:

1) *Extending Expressiveness.* The vSQL protocol is only efficient for a specific type of circuits which correspond to SQL queries. We improve expressiveness by extending the class of circuits it can efficiently handle. We then show that the resulting protocol is an *argument of knowledge* (cf. Definition 1) with circuit-independent preprocessing.
2) *New RAM Reduction.* Exploiting circuit-independent preprocessing, we devise a new RAM-to-circuit reduction that reduces the concrete size of the circuit to be verified. In more detail: circuit-independent preprocessing allows the prover to construct "on the fly" during the proving phase a circuit that is optimized for a specific input. Thus, for each step of the RAM program, the produced circuit checks only the instruction that is actually executed for the given input.

A Linear Time Prover. vRAM is the first verifiable RAM protocol with asymptotically *optimal* prover overhead. In particular, for a RAM program P of size ℓ running for T steps, the prover time in vRAM is $O(T+\ell)$ (asymptotically the same as simply executing P), whereas previous approaches required time $O((\ell+T)\text{polylog}(T+\ell))$.

Experimental Evaluation. We provide an experimental evaluation of vRAM's performance as well as compare vRAM with state-of-the-art-implementations in both the program-specific [50] and universal [11] preprocessing setting (cf. Section V). When verifying RAM programs not known during the preprocessing phase, we improve the prover's running time by 9–30$\times$ as compared to prior work [11]. On the other hand, compared to systems using program-specific preprocessing [50] vRAM achieves very similar prover performance; in fact, in some cases our prover is faster despite the fact that systems with program-specific preprocessing can deploy program-specific optimizations during the preprocessing phase.

We also show that vRAM is much better in terms of memory consumption, which is currently the main bottleneck for running large instances of verifiable computation. vRAM achieves an improvement of 55–110$\times$ in terms of memory consumption compared to [11], which allows us to prove computations involving more than 2 million CPU cycles with 256GB memory (65$\times$ more than [11]). The improvements achieved by vRAM come at the cost of increased verifier's running time and proof size, however these still remain well within the capabilities of modern machines. In Section V-B we discuss the practical limitations of our approach and provide estimations for instances where VC can be applicable.

Architecture Independence. Another advantage of vRAM's circuit independent preprocessing is that it can use information obtained *after* executing the computation to optimize the RAM architecture to be used for its verification. Any parameter of the architecture (number of registers, register width, instruction set, etc.) can be tweaked so as to reduce the size of produced circuit to be verified. Finally, our construction naturally supports both arithmetic circuits and RAM programs with a *single* preprocessing phase, allowing the parties to selectively choose the optimal representation for a particular program. Thus, if the computation has a "nice" arithmetic circuit representation, one may even avoid RAM architecture entirely. These features can result in further performance improvements, as we demonstrate in Section V-D.

B. Overview of Our Techniques

As mentioned earlier, our starting point is vSQL [52] which can be shown to be a VC scheme that efficiently handles circuits that mostly consist of parallel copies of a single sub-circuit. While circuits that are constructed from SQL queries typically have this structure, this is not the case for circuits constructed via our RAM-to-circuit encoding, since each program step can perform a different instruction, thus resulting in a different sub-circuit.

Before describing our results and addressing this issue, we briefly review vSQL. At a high level, vSQL combines the interactive proof of [19], [45], [46] (these are based on [28], and we refer to all of these variants as the *CMT protocol* in the paper) with an extractable verifiable polynomial delegation (VPD) protocol [39]. The CMT protocol can be used to verify the correct evaluation of a circuit C on an input x, assuming that in the final step the verifier can evaluate a specific polynomial p_x that depends only on x (and not on C). The latter is done using a VPD protocol, which is the only part of the construction that requires preprocessing. The VPD protocol is extractable, which guarantees that the prover "knows" an input that makes the circuit evaluate to the specified output; this can be used to support NP computations that use auxiliary input (provided by the prover).

Improving the CMT Protocol. Athough the original CMT protocol [19] can handle arbitrary arithmetic circuits, it is especially efficient for highly regular circuits and in particular circuits that consist of parallel copies of identical sub-circuits [45]. In Section IV-A, we show how to modify the CMT protocol to efficiently handle circuits that consist of (non-interconnected) parallel copies of *different* sub-circuits. At a high level, we achieve this by refactoring the recursive equation used for the CMT protocol, adding an additional variable that corresponds to the positionof a sub-circuit in the larger circuit. In this way, we can efficiently handle varying wiring patterns across different sub-circuits. This improvement is crucial for improving the concrete efficiency of our VC system for RAM programs (since each program step may perform a different instruction, resulting in entirely different sub-circuits), and may be of independent interest since it expands the type of computations that are efficiently supported by the CMT protocol.

Improving the VPD Protocol. We also present a more efficient version of the VPD protocol used by vSQL [52]. We do this by augmenting the selectively secure scheme of Papamanthou et al. [39] to make it both adaptively secure and extractable (see Section IV-B), by including additional terms in the proof. As compared to the VPD scheme used in vSQL, this reduces the prover time for multilinear polynomials (i.e., multivariate polynomials of degree 1 in each variable) from quasi-linear to linear in the number of monomials, and improves concrete efficiency by 2–4$\times$.

New RAM Reduction. Previous RAM-to-circuit reductions rely on a circuit C_{RAM} that can handle any possible instruction at any given CPU step. The circuit C_{RAM} is composed of T copies of a smaller circuit C_{exe} that can verify all possible instructions (where the i-th copy of C_{exe} verifies the i-th RAM step, and T is a bound on the total number of steps). As mentioned above, this approach "wastes resources" as eventually only one instruction will be executed at each step.

In existing constructions that handle arbitrary programs this waste is unavoidable since C_{RAM} must be fixed during the preprocessing phase, before it is known which instruction will occur at each step. However, since our argument system has circuit-independent preprocessing, we can generate the circuit C_{RAM} during the proving phase, *after* the prover executes program P on input x. This allows us to replace C_{RAM} with a circuit C_P which is constructed *on-the-fly* by the prover and is *optimized for the execution of P on x*. In particular, we "customize" the i-th copy of C_{exe} to only contain the gates needed to verify the specific instruction executed during the i-th step of P on input x. While this significantly reduces the size of the produced circuit, it raises a subtle issue: C_P no longer has a succinct representation and, in the worst case, can only be described by giving the sequence of T instructions. Applying an argument system with circuit-independent preprocessing to such a circuit results in having the verifier's overhead be $\Omega(T)$ (since he must, at the very least, hold a description of the circuit) which is as large as evaluating P. In Section III, we show how this can be avoided via a new reduction in which the different copies of C_{exe} are not arranged by their order of execution, but are instead sorted by instruction type. The result is that C_P can be described by simply listing the multiplicity of each instruction. Finally, since our protocol has public-coin verification, we can make it non-interactive in the random oracle model using the Fiat-Shamir heuristic [22].

C. Related Work

Verifiable computation was formalized in [24], [41], but research on constructing interactive protocols for verifying general-purpose computations began much earlier with the works of Kilian [33] and Micali [38]. While those works have good asymptotic performance, and follow-up works further optimized those approaches (e.g., [6], [32]), subsequent implementations revealed that the concrete costs of those approaches are prohibitively high for the prover [44].

SNARKs. The next big breakthrough in general-purpose verifiable computation came with the work of Gennaro et al. [25] (building upon earlier work by Groth [30]), which introduced quadratic arithmetic programs (QAPs) and showed that they can be used to capture the correct evaluation of an arithmetic program. QAPs have since been the de-facto tool for constructing efficient succinct arguments of knowledge (SNARKs) [12], [14] that can be used to verify arbitrary NP computations. This has led to a long line of research providing both highly-optimized systems [40], [20], [47], [42], [9], [43], [35], [18], [23] and significant protocol refinements [36], [10], [21], [29]. Our solution shares intuition with some of these works, e.g., [20] also uses the technique of verifying heterogeneous sub-computations, whereas [47] produces an arithmetic circuit adapted for the given computation. Even though all of these works use different technical approaches, one theme remains common: while the verifier's performance is generally excellent, the concrete overhead for the prover (in terms of running time, memory consumption, etc.) remains prohibitive. We refer to [51] for a detailed survey.

Verifiable RAM Computation. A series of works [9], [10], [11], [17], [50], [7] consider the problem of verifying RAM computations by reducing the verification of a RAM program to the verification of a circuit. In Section V, we compare the performance of our system with the most efficient prior work in this direction [11], [50].

II. Preliminaries

Throughout this paper we use standard notation for arithmetic circuits and multilinear extensions of polynomials (see Appendix A). To simplify notation, we implicitly assume that all field operations take constant time. Thus, whenever we report asymptotic complexities we omit a factor that is polylogarithmic in the field/group size.

Bilinear Pairings and Cryptographic Assumptions. We denote the generation of the bilinear map parameters by $\mathsf{bp} = (p, \mathbb{G}, \mathbb{G}_T, e, g) \leftarrow \mathsf{BilGen}(1^{\lambda})$, where λ is the security parameter, $\mathbb{G}, \mathbb{G}_T$ are two groups of order p (with p a λ-bit prime), $g \in \mathbb{G}$ is a generator, and $e : \mathbb{G} \times \mathbb{G} \rightarrow \mathbb{G}_T$ is a bilinear map. The security of our constructions relies on the q-strong bilinear Diffie-Hellman assumption [15] and a modified version of the q-power knowledge of exponent assumption [30], [52] (presented formally in Appendix B).

A. Argument Systems and Interactive proofs

Argument Systems. An argument system for an NP relation R is a protocol that allows a computationally bounded prover $\mathcal{P}$ to convince a verifier $\mathcal{V}$ holding input x that "$\exists w$ such that $(x;w) \in R$." Here, we focus on arguments of knowledge, i.e., if the prover convinces the verifier then it must know w. We adopt the definition of [25] which includes a parameter-generation phase executed by a trusted party.

Definition 1. *Let R be an NP relation. A tuple of algorithms $(\mathcal{G},\mathcal{P},\mathcal{V})$ is an* argument *for R if the following holds.*

- ***Completeness.*** *For every $(x;w) \in R$ and $(\mathsf{pk},\mathsf{vk})$ output by $\mathcal{G}(1^\lambda)$ it holds that $\langle \mathcal{P}(\mathsf{pk},w),\mathcal{V}(\mathsf{vk})\rangle(x) = 1$.*
- ***Knowledge Soundness.*** *For any* PPT *prover $\mathcal{P}^*$ there exists a* PPT *extractor $\mathcal{E}$ which runs on the same randomness as $\mathcal{P}^*$ such that for any x we have $\Pr[(\mathsf{pk},\mathsf{vk}) \leftarrow \mathcal{G}(1^\lambda); w \leftarrow \mathcal{E}(\mathsf{pk},x) : \langle \mathcal{P}^*(\mathsf{pk}),\mathcal{V}(\mathsf{vk})\rangle(x) = 1 \wedge (x,w) \notin R] \leq \mathsf{neg}(\lambda)$.*

We say that $(\mathcal{G},\mathcal{P},\mathcal{V})$ is a **succinct** *argument system if the running time of $\mathcal{V}$ is $\mathsf{poly}(\lambda,|x|,\log|w|)$.*

Interactive Proofs. An interactive proof [27] is a protocol that allows a prover $\mathcal{P}$ to convince a verifier $\mathcal{V}$ that $f(x) = y$ where f,x,y are known to both parties. Here soundness is required even for an unbounded cheating prover.

Definition 2. *Let λ be a statistical soundness parameter. A pair of algorithms $(\mathcal{P},\mathcal{V})$ is an interactive proof for a function f with soundness $\varepsilon(\lambda)$ if:*

- ***Completeness.*** *For any f,x,y such that $f(x) = y$ it holds that $\Pr[\langle \mathcal{P},\mathcal{V}\rangle(f,x,y) = 1] = 1$.*
- ***Soundness.*** *For any f,x,y such that $f(x) \neq y$ and any prover $\mathcal{P}^*$ it holds that $\Pr[\langle \mathcal{P}^*,\mathcal{V}\rangle(f,x,y) = 1] \leq \varepsilon(\lambda)$.*

B. The CMT Protocol

High-Level Overview. Cormode et al. [19] presented an efficient interactive proof (the CMT protocol) for arithmetic circuits. At a high level, the protocol proceeds as follows. Let C be a depth-d layered arithmetic circuit over a field $\mathbb{F}$. The protocol starts by having the CMT prover $\mathcal{P}_{cmt}$ claim that the output wires have value y. Next, the CMT protocol processes C one layer at a time, from layer 0 (the output gates) to layer d (the input gates). During the ith round, $\mathcal{P}_{cmt}$ reduces a claim about the values of C's wires at layer i to a claim about the values of C's wires in layer $i+1$. The protocol terminates after d rounds with a claim about the wire values at the input layer. Since the input x is known to the CMT verifier $\mathcal{V}_{cmt}$, it can directly check $\mathcal{P}_{cmt}$'s claim. If the check succeeds, $\mathcal{V}_{cmt}$ accepts y as the output of $C(x)$.

Notation. Before presenting a formal description of the CMT protocol, we establish some additional notation. We denote the number of gates in the ith layer of C by S_i and we set $s_i = \lceil \log S_i \rceil$ (so s_i bits suffice to identify each gate is the ith layer). The evaluation of C on an input x assigns (in the natural way) a value from $\mathbb{F}$ to each gate in C based on its output wire. For each layer i of C, define the function $V_i : \{0,1\}^{s_i} \to \mathbb{F}$ that takes as input a gate $g \in \{0,1\}^{s_i}$ and outputs its value. Note that the values returned by V_d correspond to the values of the input layer of C, i.e., x. Finally, for each layer i we define functions add_i, mult_i that we call C's *wiring predicates*. The function $\mathsf{add}_i : \{0,1\}^{s_{i-1}+2s_i} \to \{0,1\}$ takes as input a gate g_1 from layer $i-1$ and two gates g_2, g_3 from layer i and outputs 1 iff g_1 is an addition gate whose input wires are connected to g_2 and g_3. The function mult_i is defined similarly for multiplication gates. Notice that the value of a gate g at layer $i < d$ can be computed as a function of the values of the gates at layer $i+1$, i.e., $V_i(g) = \sum_{u,v\in\{0,1\}^{s_{i+1}}} (\mathsf{add}_{i+1}(g,u,v) \cdot (V_{i+1}(u) + V_{i+1}(v)) + \mathsf{mult}_{i+1}(g,u,v) \cdot (V_{i+1}(u) \cdot V_{i+1}(v)))$.

Protocol Details. One way for $\mathcal{V}_{cmt}$ to check correctness of the values at layer i is to check that $V_i(g)$ outputs the correct value of the g-th gate for every gate g in that layer. Since $V_i(\cdot)$ is a summation of other values, this can be done using the sum-check protocol from Appendix C. However, the soundness guarantee of the sum-check protocol depends on the size of the underlying field. If C is defined over a small field (e.g., if C is a boolean circuit) we replace V_i with its multilinear extension $\widetilde{V}_i$ defined over a larger field $\mathbb{F}$ via

$$\begin{aligned}\tilde{V}_i(z) &= \sum_{g\in\{0,1\}^{s_i},\ u,v\in\{0,1\}^{s_{i+1}}} f_{i,z}(g,u,v) \qquad (1)\\ &\overset{\text{def}}{=} \sum_{g\in\{0,1\}^{s_i},\ u,v\in\{0,1\}^{s_{i+1}}} \tilde{\beta}_i(z,g) \cdot \Big(\tilde{\mathsf{add}}_{i+1}(g,u,v) \cdot \\ &\quad (\tilde{V}_{i+1}(u) + \tilde{V}_{i+1}(v)) + \tilde{\mathsf{mult}}_{i+1}(g,u,v) \cdot (\tilde{V}_{i+1}(u) \cdot \tilde{V}_{i+1}(v)) \Big)\end{aligned}$$

where $\tilde{\mathsf{add}}_i$ (resp., $\tilde{\mathsf{mult}}_i$) is the multilinear extension of add_i (resp., mult_i) and $\tilde{\beta}_i$ is the multilinear extension of the function that takes s_i-bit inputs z,g and outputs 1 iff $a = b$.[1]

Assume for simplicity that C has a single output wire. The CMT protocol begins with $\mathcal{P}_{cmt}$ claiming $y = \widetilde{V}_0(0)$. Then $\mathcal{P}_{cmt}$ and $\mathcal{V}_{cmt}$ execute the sum-check protocol, which results in $\mathcal{V}_{cmt}$'s needing to check that $\widetilde{V}_0(0) = \sum_{\substack{g\in\{0,1\}^{s_0}\\ u,v\in\{0,1\}^{s_1}}} f_{0,0}(g,u,v)$. In turn, this requires the verifier to evaluate $\widetilde{V}_1$ on two random points $q_1, q_2 \in \mathbb{F}^{s_1}$. Since the verifier does not have the correct gate values for layer 1, it asks $\mathcal{P}_{cmt}$ to provide $a_1 = \widetilde{V}_1(q_1)$ and $a_2 = \widetilde{V}_1(q_2)$. We have thus reduced the claim about the value of the gate in layer 0 to the validity of two claims about the gates in layer 1. Finally, in Appendix D, we describe the way to condense these two claims into a single claim about the gates in layer 1. Proceeding in this way layer by layer, the prover and verifier end with a claim about the value of $\widetilde{V}_d$, which can be checked directly by the verifier who has access to the input x. In Appendix D we give a full description of the CMT protocol, and formally state its security and asymptotic performance guarantees.

C. A Canonical RAM Architecture

In this section we establish notation for a random-access machine supporting some instruction-set architecture.

[1] Although using $\tilde{\beta}$ is not strictly necessary [46], we use it since it improves efficiency when C is composed of many parallel copies of a smaller circuit C'.

S and S^{**}	t, pc, $r_1,\ldots,r_K$, O, flag, auxiliary[2]
I and I^{**}	line number, opcode, i,j (source registers), k (target register)
A	a, t, O, b (denoting memory load or store)

TABLE I
VALUES IN A STATE AND AN INSTRUCTION.

Hardware. We focus on RAM machine computations, where the machine is parametrized by the number of registers K and the register width (word size) W. The CPU state consists of a W-bit program counter (pc) and K general-purpose, W-bit registers $r_1,\ldots,r_K$. Each instruction operates over two operands (registers) and stores its result in a third register, to which we shall refer as the destination register. The machine's memory is a randomly accessible array of 2^W bytes. We also assume two read-only unidirectional tapes containing W-bit words. The first tape is used for the program input x, and the second tape may potentially be used for auxiliary input aux.

Program Execution. A program is a sequence of instructions, where each instruction has two operands (which are either register numbers or constants) and stores its result in a third register called the destination register. A random-access machine starts executing a program with all registers, its memory, and the program counter initialized to 0. At each step, the instruction pointed by the pc is executed. By default, every instruction increments the pc by one (i.e., pointing to the next instruction), but an instruction (e.g., jump) can also modify the pc directly to facilitate arbitrary control flow. The machine's inputs are the above-mentioned tapes, accessible via special read instructions, as well as the initial contents of its memory. The machine outputs either accept or reject. We say program P accepts input (x,aux) if the machine running program P with the specified input terminates with output accept.

Machine State and Instruction Encoding. We define the notion of machine *state* as the values of the machine's registers $pc, r_1,\cdots,r_K$ at any point during the program execution. Let $S_1,\cdots,S_T$ be a list of the machines states during the execution of some program P. We augment each state S_i to also include i in it, referring to i as S_i's *step number* as well as to include an additional field O_i, referring to it as the instruction's *output field*. An *instruction I* contains information about what operation the machine should execute (e.g., addition, multiplication, etc.), the two source registers r_i, r_j as well as the target register r_k. For a specific program (which is a sequence of instructions) $P = P_1,\cdots,P_\ell$, we augment every instruction P_i to include its location i (line number) within P. The detailed values in a state and an instruction used in our implementation is shown in Table I. We take as our set of available instructions from those used by TinyRAM [9], [11]. This is an ideal starting point for our implementation as the universal circuit for the TinyRAM CPU can be described by a relatively small arithmetic circuit.

Execution Traces. The trace $tr = (S_1,I_1,S_2,I_2,\ldots,I_{T-1},S_T)$ of a program P on inputs x,aux is a sequence of CPU states and instructions, where S_1 is the initial state and each S_i is produced by executing instruction I_{i-1} on S_{i-1}. A trace tr is *valid* for a program P on input x if there is an aux such that $P(x,\mathsf{aux})$ has trace tr. Similarly, a trace tr of a program P on input x is accepting if there exists aux such that tr is valid and we say that P accepts input (x,aux).

A Universal NP Relation for RAM Programs. The following *NP* relation $RAM_{\ell,n,T}$ captures accepting RAM programs:

Definition 3. *For $\ell,n,T \in \mathbb{N}$, relation $RAM_{\ell,n,T}$ consists of tuples $(P,x;\mathsf{aux})$ such that: (i) P is a program with $\leq \ell$ instructions, (ii) x is an input of $\leq n$ words, and (iii) $P(x,\mathsf{aux})$ accepts in $\leq T$ steps.*

D. Previous Reductions from RAM to Circuit Satisfiability

Before describing our improvements, in this section we present previous approaches for constructing a circuit that can verify the execution of RAM programs. More specifically, given a time bound T, [11] constructs a circuit C such that for any RAM program P, $\exists w : C(P,x;w) = 1$ if and only if $\exists\mathsf{aux}$ such that $P(x;\mathsf{aux})$ accepts. Throughout this paper, unless otherwise noted, we do not distinguish between the program and the input data, and we let ℓ be a bound on both the program length and the input size.

The circuit C takes as input a program P and a witness w that contains a trace $tr = (S_1,I_1,S_2,I_2\cdots,I_{T-1},S_T)$ and aux. C then outputs 1 only if S_1 is the initial state, S_T is an accepting state, and the following hold at every step i in tr:

1) **Correct Instruction Execution.** State S_{i+1} is obtained from S_i after executing instruction I_i.
2) **Correct Instruction Fetches.** I_i is the instruction in P pointed to by the program counter (pc) in S_i. If $i = 1$ we require that $pc = 0$.
3) **Correct Memory Accesses.** If I_i is a load instruction accessing address a then the value loaded is v, where v is the last value written to address a by some previous instruction (and $v = 0$ if I_i is the first load from a.)

In order to verify the above three conditions, the circuit C is constructed from three sub-circuits C_{exe}, C_{mem}, and C_{route} (cf. Figure 1(left)), which we explain below.

Ensuring Correct Instruction Execution. To ensure (1), every triple S_i,I_i,S_{i+1} is given as input to a circuit C_{exe} which performs the following two checks. (a) Check that the value O_i in S_i is correctly computed by executing I_i.[3] In case I_i is a memory load instruction, C_{exe} optimistically assumes that the loaded value O_i is correct (this will be tested separately when checking memory accesses). (b) Check that O_i is equal to r_j of S_{i+1} (or pc_{i+1} in case of jump), all other registers of S_{i+1} are the same as S_i, S_i's step number is indeed i and pc_i is equal to the line number of I_i in P (as encoded in I_i).

Ensuring Correct Instruction Fetches. To ensure (2), C must check that the instruction I_i is fetched from the location in the program P pointed by pc_i in state S_i (i.e., that I_i is the pc_i-th instruction in P). In [11], this is achieved by storing P in memory and then loading instructions before they are executed. Formally, a booting sequence $B_1,\ldots,B_\ell$ is

[2]Auxiliary includes data from the prover for efficient implementation purposes, i.e., bit-decomposition of the values for computation modulo 2^{32} and bits denoting whether an instruction is jump, memory store or load.

[3]If I_i is a memory instruction, O_i is the loaded or stored value; if I_i is a jump instruction, O_i is the jump destination.

prepended to the trace tr, with B_i storing the i-th instruction of P in memory at address i. This results in a new trace $tr = (B_1, \cdots, B_\ell, S_1, I_1, S_2, I_2 \cdots, I_{T-1}, S_T)$ of length $2T + \ell$. Each $I_i \in tr$ is then viewed as two operations: One is a load operation fetching an instruction from the memory address pointed by its line number, and the other is I_i itself. In this way, the correctness of instruction fetches is reduced to checking the consistency of the memory stores and loads performed by Bs and Is, which we describe next.

Ensuring Correct Memory Accesses. To ensure (3), Ben-Sasson et al. [11] include in w an additional trace $tr^* = (A_1, \cdots, A_{2T+\ell})$, which is a permuted version of tr where: (a) all the states in which a memory access is performed are sorted by the memory address a being accessed (with ties broken by their step number in tr), and (2) non-memory instructions are pushed to the end of tr. Notice that B_i and I_i are also sorted, using the addresses i and the line number respectively. For two adjacent entries $A_i, A_{i+1} \in tr^*$ with outputs O_i, O_{i+1}, step numbers t_i, t_{i+1} and accessing addresses a_i, a_{i+1}, respectively, the circuit C_{mem} checks the following:[4]

- If $a_i = a_{i+1}$ then $t_i < t_{i+1}$. If A_{i+1} is a load instruction, the loaded value O_{i+1} is the same as the value O_i stored or loaded by A_i.
- If $a_i \neq a_{i+1}$ then $a_{i+1} > a_i$, and if A_{i+1} is a load instruction then $O_{i+1} = 0$.

Checking Consistency Between tr and tr^*. Finally, C must ensure that tr^* is a copy of tr that contains exactly the same states and instructions, just sorted by their accessed addresses. Note that the fact that tr^* is sorted correctly has already been checked by C_{mem}. Hence, it remains to ensure that a state appears in tr^* if and only if it appears in tr. This can be done by checking that there exists a permutation π such that $\pi(tr^*) = tr$. To that end, C contains a sub-circuit C_{route} which implements a $O(T \log T)$ switching network that routes every entry in tr to its matching entry in tr^*. The control bits used for the switching network (which specifies the permutation π) are provided by the prover and included in w.

Overall Complexity. For a program of size ℓ running for T steps, the above reduction yields a circuit C of size $T \cdot |C_{exe}| + (2T + \ell) \cdot |C_{mem}| + |C_{route}|$. Since C_{exe}, C_{mem} are fixed for a given architecture (i.e., they are independent of T, ℓ), and C_{route} can be implemented using $O((T + \ell) \cdot \log(T + \ell))$ gates, we have $|C| = O((T + \ell) \cdot \log(T + \ell))$.

III. Our Interactive Argument for RAM programs

In this section, we present our argument for verifying the correct execution of RAM programs. Similar to previous approaches [9], [10], [11], [17], [50], [7], our argument for RAM programs will use as a "back-end" an argument for verifying the correct evaluation of arithmetic circuits. We thus must somehow reduce the task of verifying RAM computation to the task of verifying the correct evaluation of arithmetic circuits. One candidate for such a reduction is

[4]In case A_i corresponds to B_j or I_j, the value O_i loaded is the encoding of the instruction, i.e. the concatenation of the machine operation code and the source and destination registers.

the construction of [11] (described in Section II-D) which reduces the verification of RAM programs of T steps to the task of verifying the correct evaluation of an arithmetic circuit fo size $O(T \log T)$. However, unlike [11], in this work we rely on an efficient argument for arithmetic circuits (explained in detail in Section IV) which is both interactive and has a circuit-independent preprocessing phase. As we show in this section, it is possible to leverage these two properties in order to achieve a "tighter" reduction than the reduction of [11], resulting in a more efficient argument for RAM programs. More specifically, having a circuit-independent preprocessing phase allows us to produce a concretely smaller circuit where at each step the prover only proves the correct execution of the instruction that is actually executed by the RAM program on its specific inputs, as opposed to proving the correctness of a circuit evaluating all possible instructions. Next, the interactivity property allows us to replace the routing network used in [11] for checking trace consistency with an efficient interactive protocol for randomized polynomial identity testing. This reduces the prover's complexity from $O((T + \ell) \log(T + \ell))$ to $O(T + \ell)$ as well as improves the prover's concrete efficiency.

Our final circuit construction is shown in Figure 1(right). As in Section II-D, we must check correctness of (1) instruction execution, (2) instruction fetches, and (3) memory accesses. Next we describe our implementation of these checks.

A. Ensuring Correct Instruction Execution

Let $tr = (S_1, I_1, S_2, I_2 \cdots, I_{T-1}, S_T)$. Recall that in the reduction described in Section II-D, the correct execution of tr's instructions is checked via a universal C_{exe} which performs two sets of tests on every triple $S_i, I_i, S_{i+1} \in tr$. The first test (a) checks the correctness of O_i (i.e., that performing I_i on S_i results in O_i) while the second test (b) checks that the values from S_i are consistently propagated to S_{i+1} (including correct pc_i update and ordering of steps). Notice that while the second test is relatively simple and identical for all triples, the majority of C_{exe}'s gates are actually required for performing the first test. This is since this part of C_{exe} is often implemented by a composition of smaller circuits each of which can check the execution of a specific instruction, together with a multiplexer that specifies which instruction should be checked at this step. In order to optimize the size of C_{exe}, while maintaining the succinct representation of the result circuit C, we split C_{exe} into two sub-circuits which perform these two checks independently. For the second check we will the same circuit for all triples, whereas for the first one we we will use a circuit that can only verify the logic of the particular instruction I_i. Below, we describe in detail how these circuits are implemented.

Ensuring Correct Propagation of Values. We define a circuit C_{time} that takes as input a triple S_i, I_i, S_{i+1}, and verifies that the value of the destination register in S_{i+1} is equal to O_i, all other registers in S_{i+1} remain unchanged, and pc_{i+1} was updated appropriately. Similar to Section II-D, C_{time} also checks that S_i's step number is indeed i and that pc_i is equal to

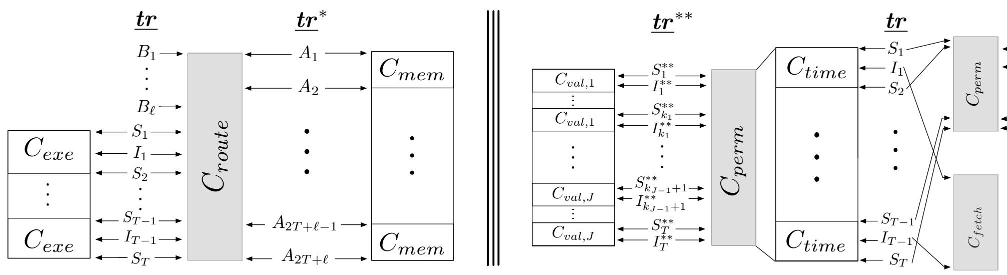

Fig. 1. Circuits for the reductions from RAM programs to circuits from Section II-D (left) and Section III (right). Circuits C_{fetch} and C_{perm} receive additional input from the verifier as described in Sections III-B and III-D, respectively.

the alleged location of I_i in P (as encoded in I_i by the prover). However, *unlike* Section II-D, we stress that C_{time} *does not verify* that O_i is the correct output after executing I_i.

Verifying Instruction Execution. Let J be the number of instruction types supported by the RAM architecture. We include in the witness w an additional trace tr^{**} that is the result of sorting the pairs $(S_i, I_i) \in tr$ by the instruction type of I_i. Define a circuit $C_{val,j}$ which takes as input a pair $(S_i^{**}, I_i^{**}) \in tr^{**}$ and checks that S_i^{**} is a valid state for the instruction I_i^{**} of type j (i.e., O_i is correctly computed by executing I_i^{**} on S_i^{**}). In this way, $C_{val,j}$ is specialized to a specific instruction type. Moreover, since tr^{**} is sorted by instruction type, the copies of $C_{val,j}$ will also appear in C sorted by j. In this way, C can be succinctly described by $(k_1, \ldots, k_J)$, where k_j (for $j = 1, \ldots, J$) denotes the number of times instruction type j appears in trace tr when program P is executed on input x (where $\sum_j k_j = T$).

B. Verifying Instruction Fetches

As described above, [11] ensures program consistency by first storing the program to memory during the machine's booting phase. Next, each instruction is sequentially loaded from memory for execution. These operations are treated the same as regular memory stores and loads, and are checked by $T+\ell$ copies of C_{mem}. Here, we explain how the correctness of these operations can be checked more efficiently assuming instructions in the program are fixed and known to the verifier (i.e., if we assume that P does not contain self-modifying code, similar to [9]).

Unlike the reduction of Section II-D, note that the trace tr does not include a boot sequence. Instead, we observe that for each triple S_i, I_i, S_{i+1}, the circuit C_{time} already checks that pc_i is equal to the line number of I_i in P (as encoded in I_i by the prover). All that remains is to verify that I_i is the instruction in P with the same line number. Equivalently, let $\{P_1, \cdots, P_\ell\}$ be the set of instructions in P where each P_i is augmented to also contain its line number within P (as defined in Section II-D). Then we only need to check that the sequence $\{I_1, \cdots, I_{T-1}\}$ is a multiset of $\{P_1, \cdots, P_\ell\}$ (the multiplicity of some P_i may be 0 to account for non-executed instructions). To that end, we add a circuit C_{fetch} that validates this multiset relation and leverages the interactive property or our scheme from Section IV. The circuit takes the sequence $I_1, \cdots, I_{T-1}$ from tr and a random value r (provided by the verifier) as input. C_{fetch} outputs the evaluation of its characteristic polynomial at point r, i.e., $\prod_{i=1}^{T-1}(I_i - r)$. The verifier also receives from the prover the multiplicity k_j of P_j in $\{P_1, \cdots, P_\ell\}$. Thus, he can compute himself the value $\prod_{j=1}^{\ell}(P_j - r)^{k_j} \overset{\text{def}}{=} \prod_{i=1}^{T-1}(I_i - r)$ and test whether it corresponds to the value output by the circuit. By the Schwartz-Zippel lemma, the probability the verifier accepts if the two polynomials are not the same (i.e., $\{I_1, \cdots, I_{T-1}\}$ is not a multiset of $\{P_1, \cdots, P_\ell\}$) is negligible. We stress that this is only secure if we ensure that the prover commits to the entire witness (including $I_1, \cdots, I_{T-1}$) before seeing r, as is the case in our construction in Section IV. In this way, we have replaced $T+\ell$ copies of C_{mem} with a smaller circuit C_{fetch} evaluating the characteristic polynomial at a random value which leads to concrete efficiency improvement.

C. Ensuring Memory Accesses

Similar to Section II-D, in order to verify memory accesses (ensuring (3)) we include in w a trace $tr^* = (A_1, \cdots, A_T)$ sorted by the memory address being accessed (again with ties broken by step number and non-memory instructions located at the end of tr^*). Since the correctness of instruction fetches is already ensured (as described above), we only sort the states S_i in tr, and the length of tr^* now becomes T. For every two adjacent entries $A_i, A_{i+1} \in tr^*$ with outputs O_i, O_{i+1}, step numbers t_i, t_{i+1} and accessing addresses a_i, a_{i+1}, respectively, the circuit C_{mem} checks the same two conditions as in Section II-D. Finally, note that the number of instruction that actually perform memory operations may be smaller than T, but we still include T copies of C_{mem} in C to account for the worst case. In Appendix E, we show how this can be further improved to only include αT copies of C_{mem}, where $0 \le \alpha \le 1$ is the ratio of memory operations in the trace.

D. Checking Consistency Between tr, tr^* and tr^{**}

Finally, it remains to check that tr^* and tr^{**} are indeed permutations of tr. Previous works [8], [9], [11] achieve this task by using routing networks, yielding a circuit of size $O((T+\ell)\log(T+\ell)$, for a T-step RAM program of size ℓ, and correspondingly increasing the prover's asymptotic running time from linear to quasilinear. Using routing networks to achieve this would yield a circuit of size $O((T+\ell)\log(T+\ell)$, for a T-step RAM program of size ℓ, which would correspondingly increase the prover's asymptotic running time from linear to quasilinear. Following the approach of [52], we leverage the interactive nature of our argument in order to avoid the use of routing networks, replacing them with a simple interactive protocol that is similar to the one used above for verifying instruction fetches. The result is that our prover's running time is only $O(T+\ell)$, i.e., asymptotically the same as simply evaluating the program.

More specifically, assume the prover holds lists $x_1,\ldots,x_m$ and $x'_1,\ldots,x'_m$ and wants to convince the verifier that they are a permutation of each other. Consider a circuit C_{perm} that takes $x_1,\ldots,x_m$ and $x'_1,\ldots,x'_m$ (provided by the prover) and a random point r (provided by the verifier) and outputs the result of $\prod_{i=1}^{m}(x_i-r)-\prod_{i=1}^{m}(x'_i-r)$. If the two lists are permutations of each other the output is always zero, otherwise by the Schwartz-Zippel lemma it is zero with negligible probability.[5] Finally, evaluating this polynomial requires $O(m)$ gates. For our argument, we use two executions of this interactive protocol, one for the pair tr,tr^* and one for tr,tr^{**}, in a way that ensures that C outputs zero only if C_{perm} outputs zero both times. From the above analysis, each of these circuits consists of $O(T+\ell)$ gates. We stress that it is crucial to have the prover commit to the two lists ahead of time, in particular before seeing r, for security purposes. This is enforced by our argument as $\mathcal{P}$ commits to the entire witness w in the first step of the protocol (cf. Construction 2, Evaluation Phase, Step 1).

We are now ready to state the following result. We defer a proof to the full version due to space limitations.

Theorem 1. *Let ℓ be a program length parameter, T be a time bound and let n be an input bound. Assuming that Construction 1 is an extractable verifiable polynomial delegation protocol, then combining the results of Section III with Construction 2 we obtain an argument system for the relation $RAM_{\ell,n,T}$ (as per Definition 3). Moreover, as the sizes of C_{time}, C_{val} and C_{mem} are constants which are independent of n,T,ℓ, the running time of $\mathcal{P}$ is $O(n+T+\ell)$ and that of $\mathcal{V}$ is $O(n+\ell+poylog(T))$. This yields a* succinct *argument with* polylog$(n+\ell+T)$ *rounds of interaction.*

IV. An Improved Argument for Arithmetic Circuits

In this section, we present our modifications to the (implicit) argument of vSQL [52]. First, we introduce a modified version of the CMT protocol that can efficiently handle circuits consisting of parallel copies of *different* sub-circuits (which is the format of our circuit, from Section III above). We then present a VPD scheme with improved efficiency and show that combining the two yields an argument of knowledge with circuit-independent preprocessing.

A. Improving The Expressibility of the CMT Protocol

Following [52], we can verify the execution of a RAM program by applying the CMT protocol to the RAM-verification circuit C described in Section III. Recall that C contains T copies of C_{mem} and C_{time}, and k_j copies of $C_{val,j}$ where $\sum_j k_j = T$. Applying the CMT protocol described in Section II-B and Appendix D (Theorem 5) to C would thus result in a prover complexity of $O(|C|\log|C|)$. In this section, we show how to modify the CMT protocol to efficiently handle circuits that consist of multiple (different) sub-circuits. When applied to our circuit C, this results in a prover time of $O(|C|\log\max\{|C_{mem}|,|C_{time}|,|C_{var}|\})$. As the sizes of C_{mem}, C_{time}, and C_{var} are constants which only depend on the specific RAM architecture, we obtain an asymptotically optimal prover running time of $O(|C|)$ which is $O(T+\ell)$.

Let C be a depth-d, size-n, layered arithmetic circuit consisting of B independent ("parallel") sub-circuits $C_1,\cdots,C_B$, each of depth at most d' and size at most n', where the outputs of $C_1,\cdots,C_n$ are fed into an aggregation circuit D of depth-d'' and size n''. In this section, we show how to modify the CMT protocol so as to prove statements about the output of C in time which is linear in the size of C. Our modified protocol proceeds as follows. We start by following the standard CMT protocol for the d'' layers of sub-circuit D. Next, for the remaining $d-d''=d'$ layers, we modify things in a similar way to [45] and [46]. Let S_i now denote the maximum number of gates in layer i across $C_1,\cdots,C_B$, and let $s_i=\lceil\log S_i\rceil$. We let V_i again be a function mapping a gate at level i to its value, but we now specify a gate g by a pair g_1,g_2, where $g_2\in[B]$ indicates the sub-circuit in which g lies and $g_1\in[S_i]$ is the index of g (at level i) within that sub-circuit. The prover and verifier then run a CMT-like protocol, but using the equation $V_i(g_1,g_2)=\sum_{u_1,v_1\in\{0,1\}^{s_{i+1}}}(\mathsf{add}_{i+1}(g_1,u_1,v_1,g_2)\cdot(V_{i+1}(u_1,g_2)+V_{i+1}(v_1,g_2))+\mathsf{mult}_{i+1}(g_1,u_1,v_1,g_2)\cdot(V_{i+1}(u_1,g_2)\cdot V_{i+1}(v_1,g_2)))$.

The equation above still recursively defines V_i in terms of V_{i+1}, but takes advantage of the fact that there is no interconnection between the different sub-circuits. This has the effect of reducing the number of variables in add_{i+1} and mult_{i+1} from $2s_{i+1}+s_i+3\lceil\log B\rceil$ to $2s_{i+1}+s_i+\lceil\log B\rceil$. Next, we define the multilinear extension of $V_i(g_1,g_2)$.

$$\tilde{V}_i(z_1,z_2)=\sum_{u_1,v_1\in\{0,1\}^{s_{i+1}},g_2\in\{0,1\}^{\log\lceil B\rceil}} f_{i,z_1,z_2}(u_1,v_1,g_2) \qquad (2)$$

$$\overset{\text{def}}{=}\sum_{u_1,v_1\in\{0,1\}^{s_{i+1}},g_2\in\{0,1\}^{\lceil\log B\rceil}}\tilde{\beta}_i(z_2,g_2)\cdot\Big(\tilde{\mathsf{add}}_{i+1}(z_1,u_1,v_1,g_2)\cdot(\tilde{V}_{i+1}(u_1,g_2)+\tilde{V}_{i+1}(v_1,g_2))+\tilde{\mathsf{mult}}_{i+1}(z_1,u_1,v_1,g_2)\cdot(\tilde{V}_{i+1}(u_1,g_2)\cdot\tilde{V}_{i+1}(v_1,g_2))\Big).$$

The only difference between equation 2 and the equation used for data-parallel circuits with same sub-circuits in [45], [46] is that $\tilde{\mathsf{add}}_{i+1}$ and $\tilde{\mathsf{mult}}_{i+1}$ take an extra variable g_2, which denotes that the gates and wiring patterns can be different in each sub-circuit. We further observe that running the same algorithm for the sumcheck protocol as in [45], [46] on equation 2 results in the same complexity on the prover, which is $O(BS_i\log S_{i+1})$. In this way, we extend the class of the circuit efficiently supported by the CMT protocol in [45],

[5] As a state (e.g., A in tr^*) contains multiple values such as O and t and we want to ensure they are permuted together, we pack the values before the check (e.g., for W-bit values (a,b,c), we set $x=a\times2^{2W}+b\times2^W+c$). If the result of a single pack overflows the field, we pack the values multiple times with respect to the first value. In our implementation, we use a 254-bit prime field, which allows packing of 7 32-bit numbers. We also use the same technique to ensure that S_i^{**} and I_i^{**} in tr^{**} are permuted together.

Definition 4. *Let $\mathbb{F}$ be a finite field, $\mathcal{F}$ a family of ℓ-variate polynomials over $\mathbb{F}$, and d a variable-degree parameter.* $(\mathsf{KeyGen}, \mathsf{Commit}, \mathsf{Evaluate}, \mathsf{Ver})$ *constitute an* extractable VPD scheme *for $\mathcal{F}$ if:*

- ***Perfect Completeness.*** *For any polynomial $f \in \mathcal{F}$ it holds that*
$\Pr\Big[(\mathsf{pp},\mathsf{vp}) \leftarrow \mathsf{KeyGen}(1^\lambda,\ell,d); \mathsf{com} \leftarrow \mathsf{Commit}(f,\mathsf{pp}); (y,\pi) \leftarrow \mathsf{Evaluate}(f,t,\mathsf{pp}) : \mathsf{Ver}(\mathsf{com},t,y,\pi,\mathsf{vp}) = \texttt{acc} \wedge y = f(t)\Big] = 1.$
- ***Soundness.*** *For any* PPT *adversary $\mathcal{A}$ the following probability is negligible:*
$\Pr\Big[(\mathsf{pp},\mathsf{vp}) \leftarrow \mathsf{KeyGen}(1^\lambda,\ell,d); (f^*,t^*,y^*,\pi^*) \leftarrow \mathcal{A}(1^\lambda,\mathsf{pp}); \mathsf{com} \leftarrow \mathsf{Commit}(f^*,\mathsf{pp}) : \mathsf{Ver}(\mathsf{com},t^*,y^*,\pi^*,\mathsf{vp}) = \texttt{acc} \wedge y^* \neq f^*(t^*)\Big].$
- ***Extractability.*** *For any* PPT *adversary $\mathcal{A}$ there exists a polynomial-time algorithm $\mathcal{E}$ with access to $\mathcal{A}'s$ random tape such that for all benign auxiliary inputs $z \in \{0,1\}^{poly(\lambda)}$ the following probability is negligible:*
$\Pr\Big[(\mathsf{pp},\mathsf{vp}) \leftarrow \mathsf{KeyGen}(1^\lambda,\ell,d); \mathsf{com}^* \leftarrow \mathcal{A}(1^\lambda,\mathsf{pp},z); f' \leftarrow \mathcal{E}(1^\lambda,\mathsf{pp},z) : \mathsf{CheckCom}(\mathsf{com}^*,\mathsf{vp}) = \texttt{acc} \wedge \mathsf{com}^* \neq \mathsf{Commit}(f',\mathsf{pp})\Big].$

Construction 1 (Verifiable Polynomial Delegation). *Let $\mathbb{F}$ be a prime-order field, and ℓ,d variable and degree parameters such that $O(\binom{\ell(d+1)}{\ell d})$ is poly(λ). Consider the following protocol for the family $\mathcal{F}$ of ℓ-variate polynomials of variable-degree d over $\mathbb{F}$.*

1) $\mathsf{KeyGen}(1^\lambda,\ell,d)$*: Select uniform $\alpha, s_1,\dots,s_\ell \in \mathbb{F}$, run $\mathsf{bp} \leftarrow \mathsf{BilGen}(1^\lambda)$ and compute $\mathbb{P} = \{g^{\prod_{i\in W} s_i}, \ g^{\alpha\cdot\prod_{i\in W} s_i}\}_{W\in\mathcal{W}_{\ell,d}}$. The public parameters are $\mathsf{pp} = (\mathsf{bp},\mathbb{P},g^\alpha)$, and the verifier parameters are $\mathsf{vp} = (\mathsf{bp},g^{s_1},\cdots,g^{s_\ell},g^\alpha)$. For every $f \in \mathcal{F}$ we denote by $\mathsf{pp}_f \subseteq \mathsf{pp}$ the minimal subset of the public parameters* pp *required to invoke* Commit *and* Evaluate *on f.*
2) $\mathsf{Commit}(f,\mathsf{pp}_f)$*: If $f \notin \mathcal{F}$ output null. Else, compute $c_1 = g^{f(s_i,\dots,s_\ell)}$ and $c_2 = g^{\alpha\cdot f(s_i,\dots,s_\ell)}$, and output the commitment $\mathsf{com} = (c_1,c_2)$.*
3) $\mathsf{CheckCom}(\mathsf{com},\mathsf{vp})$*: Check whether* com *is well-formed, i.e., output* accept *if $e(c_1,g^\alpha) = e(c_2,g)$ and* reject *otherwise.*
4) $\mathsf{Evaluate}(f,t,\mathsf{pp}_f)$*: On input $t = (t_1,\dots,t_\ell)$, compute $y = f(t)$. Next, using Lemma 1 compute the polynomials $q_i(x_i,\dots,x_\ell)$ for $i = 1,\dots,\ell$, such that $f(x_1,\dots,x_\ell) - f(t_1,\dots,t_\ell) = \sum_{i=1}^{\ell}(x_i - t_i)\cdot q_i(x_i,\dots,x_\ell)$. Output y and the proof $\pi := \{g^{q_i(s_1,\dots,s_\ell)}, g^{\alpha q_i(s_1,\dots,s_\ell)}\}_{i=1}^{\ell}$.*
5) $\mathsf{Ver}(\mathsf{com},y,t,\pi,\mathsf{vp})$*: Parse the proof π as $(\pi_1,\pi_1'\dots,\pi_\ell,\pi_\ell')$. If $e(c_1/g^y,g) \overset{?}{=} \prod_{i=1}^{\ell} e(g^{s_i - t_i},\pi_i)$ and $e(c_1,g^\alpha) = e(c_2,g)$ and $e(\pi_i,g^\alpha) = e(\pi_i',g)$ for $1 \le i \le \ell$ output* accept *otherwise output* reject.

[46] without any overhead on the prover time.[6] We analyze the complexity of the sum-check protocol from equation 2 in Appendix F. We present the following result.

Theorem 2. *Let $C : \mathbb{F}^n \to \mathbb{F}$ be a depth-d layered arithmetic circuit consisting of B parallel sub-circuits $C_1,\dots,C_B$ connected to an "aggregation" circuit D such that $|D| = O(|C|/\log|C|)$, and let $S = \max_j\{\mathsf{width}(C_j)\}$. Executing the CMT protocol from Construction 3 using Equation 2 and the above described modifications to the sum-check protocol, yields an interactive proof for C with soundness $O(d\cdot\mathsf{width}(C)/|\mathbb{F}|)$. Moreover, $\mathcal{P}$'s running time is $O(|C|\log S)$ and the protocol uses $O(d\log(\mathsf{width}(C)))$ rounds of interaction. If $\tilde{\mathsf{add}}_i$ and $\tilde{\mathsf{mult}}_i$ are computable in time $O(\mathsf{polylog}(\mathsf{width}(C)))$ for all the layers of C, then the running time of the verifier $\mathcal{V}$ is $O(n + d\cdot\mathsf{polylog}(\mathsf{width}(C)))$.*

B. A VPD Scheme with Linear Prover Time

In the last step of the CMT protocol, the verifier $\mathcal{V}_{cmt}$ evaluates a polynomial $\widetilde{V}_d$ on a random point r_d. Since the number of terms in $\widetilde{V}_d$ is equal to the number of input gates of C, this makes the verifier's work linear not only in the size of the input x but also the length of the witness w. In vSQL [52], this is avoided by using a VPD scheme that allows $\mathcal{P}$ to provide $\widetilde{V}_d(r_d)$ to $\mathcal{V}$ together with a succinct proof of its validity. (See Definition 4 for the definition of a VPD scheme. Our definition adapts that of [52] by introducing an additional algorithm CheckCom that checks if a commitment is well-formed.) Here, we improve the VPD scheme of [52] and present a new scheme with the same verifier complexity, but with linear prover in the number of terms of $\widetilde{V}_d$ (as opposed to quasi-linear).

As our starting point we use the selectively secure VPD scheme of Papamanthou et al. [39]. Unfortunately, selective security means that the parameters used for the VPD protocol are computed as a function of the specific point r_d on which the VPD will be executed. This is insufficient for our application since VPD's parameters will be generated once during the preprocessing phase which happens *before* the CMT protocol To overcome this limitation, we modify this scheme to require the prover to provide additional "extractability" terms as part of the evaluation proof. Our modified VPD scheme is given as Construction 1. We define the *variable degree* of a multivariate polynomial f be the maximum degree of f in any of its variables, and use $\mathcal{W}_{\ell,d}$ to denote the collection of all multisets of $\{1,\dots,\ell\}$ for which the multiplicity of any element is at most d. We formally state the security and asymptotic performance guarantees of the scheme in Appendix G.

C. Putting it All Together

Finally, we present our argument system with circuit-independent preprocessing. Our construction combines the modified CMT protocol from Section IV-A with the VPD scheme presented in Section IV-B. We refer to the prover and verifier of the CMT protocol as $(\mathcal{P}_{cmt},\mathcal{V}_{cmt})$, respectively, and to the algorithms of the VPD scheme as $(\mathsf{KeyGen}, \mathsf{Commit}, \mathsf{Evaluate}, \mathsf{Ver})$. We construct an argument system $(\mathcal{G},\mathcal{P},\mathcal{V})$ for the satisfiability of arithmetic circuits over finite fields, where the preprocessing done by $\mathcal{G}$ depends on a bound on the size of the circuit, the size of its input, and the field over which it is defined, but not the circuit itself.

Let $\mathcal{V}_{cmt}^{1+2}$ be the restriction of the CMT verifier from Construction 3 which performs Steps 1 and 2 of $\mathcal{V}_{cmt}$ and outputs (r_d,a_d) without performing Step 3. Construction 2 is a formal description of our argument system. Consider the following theorem.

Theorem 3. *If Construction 1 is am extractable VPD scheme,*

[6]The complexity of the CMT protocol for circuits composed of identical sub-circuits has recently been improved to $O(BS_i + S_i\log S_i)$ in [49]. Generalizing the technique for different sub-circuits is left as a future work.

Construction 2. *Let $\mathbb{F}$ be a prime-order field with $|\mathbb{F}|$ exponential in λ, and let n,t be input size and circuit size parameters. For simplicity of exposition we assume that n is a power of 2. Consider the algorithms $\mathcal{G},\mathcal{P},\mathcal{V}$ described below.*
Preprocessing Phase. *$\mathcal{G}(1^\lambda,n,t)$ runs $(\mathsf{pp},\mathsf{vp}) \leftarrow \mathsf{KeyGen}(1^\lambda,n,1)$. The proving key pk is set to be pp and the verification key vk is set to be vp.*
Evaluation Phase. *Let $C : \mathbb{F}^{n_x+n_w} \to \mathbb{F}$ be a depth-d arithmetic circuit with at most t gates such that $n_x+n_w \le n$. Moreover, let $x \in \mathbb{F}^{n_x}$ and $w \in \mathbb{F}^{n_w}$ be such that $C(x;w)=1$. Assume that $n_w/n_x = 2^m-1$ for some $m \in \mathbb{N}$. Consider the following protocol between $\mathcal{P}$ and $\mathcal{V}$.*

1) *$\mathcal{P}$ first commits to the multilinear extension $\widetilde{V}_d$ of the input layer of $C(x;w)$. That is, $\mathcal{P}$ runs $c \leftarrow \mathsf{Commit}(\widetilde{V}_d,\mathsf{pp})$ and sends c to $\mathcal{V}$. Upon receving c, $\mathcal{V}$ runs $\mathsf{CheckCom}(c,\mathsf{vp})$. If the output is reject, $\mathcal{V}$ rejects.*
2) *$\mathcal{V}$ computes the multilinear extension $\tilde{x}$ of the input x, generates a random point $r \in (\mathbb{F}^{\log(n_x)} \times 0^{\log(n_w)})$ and sends r to $\mathcal{P}$. $\mathcal{P}$ executes $(a,\pi) \leftarrow \mathsf{Evaluate}(\widetilde{V}_d,r,\mathsf{pp})$ and sends (a,π) to $\mathcal{V}$. $\mathcal{V}$ executes $\mathsf{Ver}(c,a,r,\pi,\mathsf{vp})$. In case Ver outputs reject or $a \neq \tilde{x}(r)$, $\mathcal{V}$ rejects.*
3) *$\mathcal{V}$ runs $\mathcal{V}_{cmt}^{1+2}$ and $\mathcal{P}$ runs $\mathcal{P}_{cmt}$ to verify $C(x;w)=1$. If $\mathcal{V}_{cmt}^{1+2}$ rejects at any point, $\mathcal{V}$ rejects. Otherwise, let r_d,a_d be the final values returned by $\mathcal{V}_{cmt}^{1+2}$. At this point, $\mathcal{V}$ must verify that $\tilde{V}_d(r_d)=a_d$.*
4) *$\mathcal{V}$ sends r_d to $\mathcal{P}$. Upon receiving r_d, $\mathcal{P}$ executes $\mathsf{Evaluate}(\tilde{V}_d,r_d,\mathsf{pp})$ and obtains (a'_d,π') which he sends to $\mathcal{V}$.*
5) *$\mathcal{V}$ upon receiving (a'_d,π') executes $\mathsf{Ver}(c,a'_d,r_d,\pi',\mathsf{vp})$. In case Ver outputs reject or $a'_d \neq a_d$, $\mathcal{V}$ rejects. Otherwise, $\mathcal{V}$ accepts.*

then Construction 2 is an argument system for arithmetic circuits. When used for a depth-d, layered circuit C consisting of B parallel sub-circuits $C_1,\ldots,C_B$ whose outputs feed into a circuit D with $|D| \le |C|/\log|C|$, the running time of $\mathcal{P}$ is $O(|C|\cdot\log\max_j\{\mathsf{width}(C_j)\})$ and the protocol has $O(d\log(\mathsf{width}(C)))$ rounds. If C has input length n and is log-space uniform then the running time of $\mathcal{V}$ is $O(n+d\cdot\mathrm{poylog}(|C|))$. Finally, if d is polylog$(|C|)$*, the above construction is a* succinct *argument.*

V. Experimental Evaluation

Software and Hardware. We implemented our constructions (including the RAM reduction, circuit generator, CMT protocol, and VPD protocol) in C++. We use the GMP library [3] for field arithmetic and OpenSSL's [5] SHA-256 implementation for hashing. For the bilinear pairing we use the ate-paring library [1] on a 254-bit elliptic curve.

We run our experiments on an Amazon EC2 m4.2xlarge machine having 32 GB of RAM and an Intel Xeon E5-2686v4 CPU with eight 2.3 GHz virtual cores. Our implementations are *not* parallelized and only use a single CPU core.

A. Comparison with vnTinyRAM and Buffet

In this section, we compare the performance of our system to existing systems for verifiable RAM. We compare to Buffet [50], a verifiable RAM system with program-specific prepossessing (where the parameters generated by the trusted preprocessing can only be used to verify one specific program on different inputs) and vnTinyRAM [11], a universal verifiable RAM system (where the parameters generated by the trusted preprocessing can be used to verify any program up to some bound on the number of steps). We also measure the performance of our system against naive unverified execution of the RAM program. Finally, in Section V-C we also discuss comparisons to other verifiable RAM systems.

Benchmark. As a benchmark, we evaluate the RAM programs from [50] (see Table II). Following that work, we benchmark our system using programs of three types.

1) **Circuit Friendly.** The function computed by these programs has a very efficient circuit representation. We use *matrix multiplication* as an example.
2) **Fixed Memory Access and Instruction Patterns.** These programs do not exploit the full generality of RAM machines, i.e., their memory-access patterns and control flow do not depend on the program's inputs. This allows for a tighter RAM-to-circuit reduction since it can be determined ahead of time which instruction will be executed at each time step. Thus, the produced circuit only needs to handle a specific instruction per cycle. We use *pointer chasing* and *merge sort* as examples of such programs.
3) **Input-dependent Memory Access and Instruction Patterns.** Such RAM programs use the full generality of RAM machines since they have input-dependent control flow and memory-access patterns. In particular, the circuit generated by the RAM reduction must be able to handle multiple possible instructions at every step. We use *KMP string matching* [34] and *CSR sparse matrix-vector multiplication* [26] as examples of such programs.

Benchmark	Input Size	# of Cycles	Native
1: Matrix Mult.	n=215	96M	42ms
2: Pointer Chasing	$n=16634$	50K	22μs
3: Merge Sort	$n=512$	65K	28μs
4: KMP Search	$n=2900, k=256$	30K	13μs
5: Sparse Mat-Vec Mult.	$n=1150, k=2300$	27K	12μs

TABLE II
Benchmarks in our experiments. We report the input size, the number of CPU cycles and the native running time on verifier for the instances we used in Table III.

Buffet Evaluation Methods. Buffet's front-end takes a RAM program and outputs a circuit that verifies its execution and its back-end uses a circuit-based VC system based on Pinocchio [40]. We evaluate Buffet using the released code [2].

vnTinyRAM Evaluation Methodology. We evaluate vnTinyRAM [11] using the code at [4]. As the code that takes a TinyRAM program and outputs the traces for vnTinyRAM is not available, we are unable to produce vnTinyRAM traces corresponding to the execution of any benchmark RAM program. Instead, we estimate the cost of vnTinyRAM by running the prover on traces of appropriate length resulting from execution random machine instructions. Since the performance of vnTinyRAM only depends on the total number of CPU steps and not on the instruction being executed at each step, this estimate is accurate.[7]

[7]A version of vnTinyRAM that removes unnecessary instructions in each step after running the particular program to be verified was released by Wahby et al. [50]. However, since the prover in this program-specific version is unable to handle arbitrary RAM programs, it is not appropriate for our comparison.

	Setup Time (min)			Prover Time (min)			|C| (Millions of gates)				Verification Time (ms)		
	TinyRAM	Buffet	vRAM	TinyRAM	Buffet	vRAM	TinyRAM	Buffet	vRAM (mult/total)		TinyRAM	Buffet	vRAM
#1	460000*	16.6	38.7	290000*	14.4	0.65	240000*	9.9	9.9	19.8	422*	401	26
#2	310*	20.0		150*	11.2	17.3	125*	8.6	38.5	150.8	56*	69	93
#3		16.1		200*	9.6	21.2	164*	7.9	36.2	148.3	9*	8	91
#4		22.9		90*	12.6	9.2	75*	10.5	18.2	72.4	15*	20	84
#5		20.8		82*	11.8	10.2	68*	9.4	18.1	74.3	20*	15	85

TABLE III
COMPARISON OF THE PERFORMANCE OF vRAM VERSUS BUFFET AND VNTINYRAM (* DENOTES SIMULATION DUE TO MEMORY EXHAUSTION).

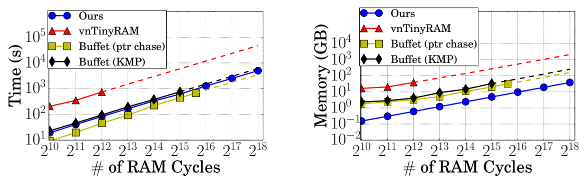

Fig. 2. Prover time (left) and memory consumption (right) of our construction vs vnTinyRAM and Buffet for various number of CPU steps.

Using a Different Back-End for vnTinyRAM and Buffet. Both Buffet and vnTinyRAM can be re-factored to use the more recent construction of [31] as their back-end. This would result in an approximate improvement of 30% in their setup, prover time and public key size as well as 50% improvement in their proof and verification key sizes. This would also improve verification time by 3×, as per the benchmarks of [4].

vRAM Evaluation Methodology. For vRAM, we implemented our own TinyRAM simulator to output the program traces used by our prover and verifier backend. We then adapted the assembly code for the programs in the Buffet benchmark, and ran them in our TinyRAM simulator to obtain execution traces, which we provided to prover-verifier backend. In order to measure the cost of our system vs. naive unverified execution, we estimate the execution time of random instructions on a single-threaded 2.3 GHz CPU core.

Experimental Results. The results of the comparison are summarized in Tables II and IV as well as in Figure 2. We executed each program on the largest input size reported in [50]. Table II summarizes their input size, number of CPU cycles and the native running time if executed on the verifier locally. As vnTinyRAM cannot handle such large parameters, we estimate its cost by extrapolation, assuming linear growth. This yields a conservative estimate since the overhead of vnTinyRAM's prover grows quasilinearly (rather than linearly) with the number of RAM instructions. We report setup time, prover and verifier time, proof size and the size of the circuit verifying the RAM program. In Figure 2, we show the prover time and memory consumption of the three systems versus the number of CPU steps. In vRAM, these are mainly determined by the number of CPU steps executed by the benchmark, rather then the specific choice of instructions executed in these steps. Consequently, we show the performance of pointer chasing as a representative example, with other programs behaving similarly. Since Buffet optimizes the circuit generated based on a particular benchmark program, we report two cases: one is pointer chasing, which is a fixed-RAM program, and the other is string search, which is a data dependent RAM program.

Comparison with vnTinyRAM. Both our system and vnTinyRAM can verify the execution of arbitrary programs with a single setup. As shown in Table III and Figure 2 (left), for all benchmarks except matrix multiplication, our system achieves an approximate 8× improvement in setup time and 9× improvement in prover time compared to vnTinyRAM. Note that vnTinyRAM is unable to exploit the fact that matrix multiplication is circuit-friendly, leading to large circuit size, setup, prover and verifier times. Since our system uses a preprocessing phase that only depends on the input size and is otherwise agnostic to the program representation, for circuit-friendly benchmarks we are able to directly use the program's circuit representation and thereby obtain an improvement of more than 4 orders of magnitude for setup time and 5 orders of magnitude for proving time compared to vnTinyRAM.[8]

The speedup obtained by vRAM is due to (1) the better RAM-to-circuit reduction from Section III; and (2) the faster argument system from Section IV. To isolate the effect of (1), in Table III we report the number of gates in the circuits produced by our reduction. Note that unlike vnTinyRAM and Buffet, in vRAM all types of gates (numbers reported in the last column) contribute to the prover time, instead of multiplication gates only.[9] Thus, to facilitate the comparison between vnTinyRAM's circuit reduction and our circuit reduction, we also report the number of multiplication gates in the table. As shown in Table III, the number of multiplication gates in our system is 3.3–4.5× less than in vnTinyRAM. Regarding (2), the performance of our argument system is demonstrated in more detail in Appendix H, where we show that the per-gate cost of our system is lower than that of QAP-based systems.

Comparison with Buffet. The main advantage of our system compared to Buffet is that it can support arbitrary programs with a single setup. As shown in Table III, the setup time for our system is 38.7 minutes for any program that runs for up to 65K CPU steps . Although the setup time of Buffet for the indicated programs is lower, an independent setup would have to be run for each different program to be verified (and the set of programs being verified must be known at the time setup is run). Moreover, we note that Buffet's setup time would likely be larger than ours if used for a program running for 65K CPU steps (which none of the benchmarks do).

Overall, the prover time of our system is comparable to that of Buffet. On one hand, for programs with fixed memory access and instruction patterns (such as pointer chasing and merge sort) Buffet can perform numerous optimizations, since the instruction to be executed in each CPU step is

[8]Note that in order to support all the benchmarks in Table III, vnTinyRAM only needs to execute a single preprocessing phase which is as large as the largest instance, i.e. matrix multiplication. However, for fair comparison, we report a separate setup time for the 4 RAM-friendly programs and compare the performance of our system to this number.

[9]Both vnTinyRAM and Buffet use the notion of quadratic constraints with each constraint verifying that the product of the outputs of two unbounded fan-in gates equals to the output of a third unbounded fan-in add gate.

	#1	#2	#3	#4	#5
Proof Size (KB)	4	256	255	236	235
Memory Usage (GB)	3.6	7.6	7.7	3.8	3.8

TABLE IV
PROOF SIZE AND MEMORY USAGE OF vRAM.

pre-determined. This allows Buffet to highly customize the resulting circuit. Nonetheless, our system is still only around $2\times$ slower than Buffet while avoiding program-dependent preprocessing. On the other hand, for programs with input-dependent memory and instruction patterns (such as KMP string search and sparse matrix-vector multiplication), our system actually *outperforms* Buffet, despite the fact that the latter can optimize the circuit during preprocessing. Moreover, as mentioned in [50, Section 4.3], if a program has deep nesting of data dependent loops or complex conditions (e.g., a state machine), the compiler of Buffet may have to incur a significantly higher overhead, since the amount of applicable optimizations will be limited. However, the performance of our construction is not adversely affected by such programs therefore our speedup compared to Buffet can be higher.

Finally, we note that when the program is circuit-friendly, e.g., matrix multiplication, Buffet can also represent the computation using a circuit. In this case, the circuit is exactly the same in both systems, and the prover time of our system is $22\times$ faster than Buffet, since our argument system outperforms Buffet's Pinocchio-based argument [40].

Memory Consumption. Another advantage of our system is that it uses much less memory in order to prove the same statement. As shown in Figure 2 (right), the memory consumption of our system is 55–110$\times$ less than vnTinyRAM, yielding a two orders of magnitude improvement. The memory consumption is also $4-8\times$ less than Buffet. In particular, on a desktop machine with 32GB of RAM, we can execute 2^{18} CPU steps, while vnTinyRAM can only reach 2^{12} steps, and Buffet can reach $2^{15}-2^{16}$ steps. We also report the memory consumption for the benchmarks we run in Table IV. The improvement is largely due to our reliance on the CMT protocol which imposes a minimal memory overhead for the non-input part of the circuit. In fact, although the circuit size is much larger than the input size, the memory usage of our VPD protocol and the CMT protocol are on the same order. In addition, in the VPD protocol, the memory is mainly used for storing the public key, thus the usage is roughly the same in the setup and the evaluate phase of VPD.

Verification Time and Proof Size. We next compare the verification time and communication cost of our system with vnTinyRAM and Buffet, both of which outperform our system. In particular, the verification time is 9–56ms for vnTinyRAM and 8–35ms for Buffet (except matrix multiplication). Also, vnTinyRAM and Buffet inherit a proof size of 288 Bytes from QAP-based SNARKS. For comparison, the verification time and the overall communication cost for our construction varies on different sizes of circuits. As shown in Table III and IV, the verification time is 84–93ms and the communication is 235–256KB for different programs. However, we believe that these are very modest quantities for any modern machine.

Proving 2 Million Instructions. To demonstrate the ability of our construction to handle the task of verifying programs that run for large amounts of CPU steps, we also ran our system on an Amazon EC2 m4.16xlarge machine featuring 256GB of RAM and an Intel Xeon E5-2676v3 CPU with 64 virtual cores running at 2.4GHz. Using this machine, we executed our system for programs consisting of 2^{21} instructions. The reported prover's time is 51000s, the memory consumption grows to 252 GB and the total number of gates in the circuit is 4.8 billion. While these numbers are concretely large, we stress that, to the best of our knowledge, this is by far the largest reported successfully performed instance of verifiable RAM computation. In particular, this instance is about $65\times$ larger than the largest instance reported in [11] (which was achieved by using a 256GB solid state drive as additional memory space). Finally, the reported verification time was less than 105ms and the total communication cost was 336.5KB.

B. Practical Limitations of vRAM

The obvious reason for the verifier to delegate computations to a prover is to save on resources such as time or memory consumption. For this to make sense, it must be the case that the resources required to verify a program are fewer than naively executing it. Assuming the verifier runs on a 1GHz machine computing 10^9 instructions per second, for QAP-based systems such as vnTinyRAM and Buffet which offer extremely efficient verification, the verifier's break-even point for saving computational power is delegating programs larger than 10 million TinyRAM instructions. As the verifier's performance in our construction are $4-10\times$ worse, the break even point for vRAM is about 135 million instructions.

However, we argue that a naive computation of the break even point is an oversimplified performance metric which hides important practical considerations. First, it assumes that the verifier's computational resources are of the same cost as the prover's recourses. An example where this is not the case is where the verifier is manufactured using old-but-trusted hardware, compared to a newer but untrusted prover (e.g., see the setting of [48], [49]). Second, in the case of zero-knowledge SNARKs, the verifier is *unable* to perform the computation by itself, as it involves the prover's private data. Thus, the verifier is forced to use the (slower) SNARK in order to validate the computation's correctness. While vRAM does not support zero-knowledge, recent follow up work [53] shows a zero-knowledge variant of the verifiable computation protocol presented in Section IV. Moreover, vRAM can also be used for delegation of data to the prover (e.g., for cloud storage, while keeping only a hash of the data locally). In this case, local execution is again impossible for the verifier, unless he is willing to download all of the data for each computation.

Finally, focusing on break-even-point metric completely hides the prover's overhead. In particular, under this metric, a VC protocol with a break-even point of a single instruction where the prover takes decades to produce a proof appears to be much more performent than current VC protocols where the proof is produced within hours and have a break even point of millions of instructions. This is especially problematic since the largest instances supported by the current generation of VC protocols are still *below* the protocol's break-even-point. As almost all computations performed by modern machines easily last hundreds of billions of cycles, we argue that it is also important to consider the ratio between the break-even point and the largest instance supported by a VC protocol (given fixed prover recourses). While having a slower verifier and a larger break-even-point, vRAM offers a much more efficient prover (both in time and more importantly in memory consumption) compared to other VC protocols. In particular, for a prover with 256GB of memory, vRAM can support computations which are about 63× away from its break-even point, compared to vnTinyRAM's 312× and Buffet's 20–40× (depending on the program to be verified).

C. *Comparison to Other RAM-based VC systems*

In this section, we briefly discuss the performance of our system compared to other RAM-based VC systems.

Pantry and SNARKs for C. Pantry [17] and SNARKs for C [9] are two VC schemes that predate Buffet and vnTinyRAM, with their performance subsumed by those systems (see [50, Figure 10] and [11, Figure 3]).

Exploiting Data Parallel Structure via Bootstrapping. Geppetto [20] is a VC system that takes a large circuit, splits it into sub-circuits, and preprocesses each sub-circuit with a SNARK separately. An additional SNARK is then applied to aggregate and verify the outputs of all sub-circuits in a "bootstrapping" step. Though verifiable RAM is not explicitly considered in [20], the system can be potentially applied to circuits checking the correctness of a RAM program, such as ones in Sections II-D and III. Due to the data parallel structure of these circuits, Geppetto can reduce the setup time asymptotically (e.g., only one setup for the sub-circuit C_{mem}, C_{time} etc.). However, it introduces a big concrete overhead for both setup and prover time because of the bootstrapping phase. For example, it requires $\sim 30{,}000$-$100{,}000$ gates to bootstrap one small sub-circuit of just 500 gates [20, Section 7.3.1].

Constant or No Preprocessing. Two alternative approaches for RAM-based VC are suggested in [10], [7] by Ben-Sasson et al. The first uses composition of elliptic curves to recursively apply a SNARK in a sequence of T fixed-size circuits, each of which validates the state of a single previous CPU step, executes the next CPU step, and outputs the new state. In this way, the resulting setup time is constant. The second constructs a RAM-based VC without any preprocessing by using PCPs. Both these systems incur a very large concrete overhead on the prover. It takes 35.5 seconds/cycle for the first system [10, Figure 1], which is about 3000× slower than ours. For the second one, it takes 0.33 seconds/cycle using 64 threads in parallel [7, Figure 1], which roughly corresponds to 21.1 seconds/cycle using single thread [7, Section 2]. This is compared to our single threaded implementation which achieves 0.015 seconds/cycle. We leave the task of achieving a speedup for our system via parallelization as future work.

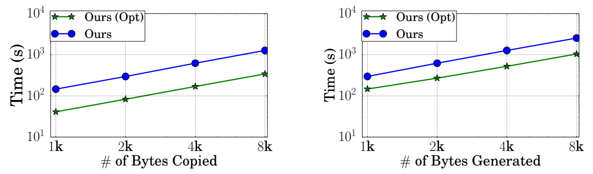

Fig. 3. Prover time for evaluating memcpy (left), RC4 (right) using vRAM with (green) and without (blue) the optimizations of Section V-D. For memcpy we vary the size of the copied memory block and for RC4 we vary the number of pseudorandom bytes generated.

D. *Just-in-Time Architecture*

Next, we use the architecture-independent preprocessing property of our scheme to improve performance for specific tasks. Common just-in-time compilation methods are used to optimize the executed code for a specific architecture. The circuit independent preprocessing feature of our construction allows us to take this approach further and modify the machine's architecture in order to better fit a specific program *after* executing it, when the program's exact behavior on its inputs is known. We illustrate this using two benchmarks from [9], [11]. We stress that since our protocol has architecture-independent preprocessing we are able to change the architecture without rerunning the preprocessing phase. In particular, the following results were achieved with a single preprocessing execution. In all cases, the verifier's runtime remained below 150ms.

Improving Performance by Adding Instructions. Figure 3(left) shows prover's time for evaluating a program which copies consecutive blocks of memory from one location to another (e.g., `memcpy`). We achieve a 3.6× improvement by introducing a memory instruction which (1) copies a byte from memory address A to memory address B and (2) increments A and B by 1 for the next loop iteration. This reduces the number of gates in the obtained circuit, thus yielding lower prover time. In this case, we did not modify any of the machine's other parameters (e.g., number of registers and register size).

Improving Performance by Changing Register Sizes. Next, Figure 3(right) shows prover's time for evaluating a RC4 pseudorandom generator on a highly specialized architecture. More specifically, we modified the machine to contain 3 8-bit registers, a 32-bit address register for memory accesses and a 32-bit program counter. Each RC4 round was implemented using 16 instructions operating over the 8-bit registers. Notice the 2.4× speedup compared to the non-optimized version, which again results from the overall reduction of necessary gates in order to generate one pseudorandom byte.

Acknowledgments

We thank the anonymous reviewers for their comments, and Stefano Tessaro for shepherding the paper. This work was supported in part by NSF awards #1514261 and #1652259, financial assistance award 70NANB15H328 from the U.S. Department of Commerce, National Institute of Standards and Technology, the 2017-2018 Rothschild Postdoctoral Fellowship, and the Defense Advanced Research Project Agency (DARPA) under Contract #FA8650-16-C-7622.

References

[1] Ate pairing. https://github.com/herumi/ate-pairing.

[2] Buffet. https://github.com/pepper-project/releases.

[3] The GNU multiple precision arithmetic library. https://gmplib.org/.

[4] libsnark. https://github.com/scipr-lab/libsnark.

[5] OpenSSL toolkit. https://www.openssl.org/.

[6] W. Aiello, S. N. Bhatt, R. Ostrovsky, and S. Rajagopalan. Fast verification of any remote procedure call: Short witness-indistinguishable one-round proofs for NP. In *ICALP 2000*.

[7] E. Ben-Sasson, I. Bentov, A. Chiesa, A. Gabizon, D. Genkin, M. Hamilis, E. Pergament, M. Riabzev, M. Silberstein, E. Tromer, and M. Virza. Computational integrity with a public random string from quasi-linear PCPs. In *Eurocrypt 2017*.

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

[9] E. Ben-Sasson, A. Chiesa, D. Genkin, E. Tromer, and M. Virza. SNARKs for C: Verifying program executions succinctly and in zero knowledge. In *Crypto 2013*.

[10] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza. Scalable zero knowledge via cycles of elliptic curves. In *Crypto 2014*.

[11] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza. Succinct non-interactive zero knowledge for a von Neumann architecture. In *USENIX Security 2014*.

[12] N. Bitansky, R. Canetti, A. Chiesa, and E. Tromer. From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again. In *ITCS 2012*.

[13] N. Bitansky, R. Canetti, O. Paneth, and A. Rosen. On the existence of extractable one-way functions. In *STOC 2014*.

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

[15] D. Boneh and X. Boyen. Short signatures without random oracles. In *Eurocrypt 2004*.

[16] E. Boyle and R. Pass. Limits of extractability assumptions with distributional auxiliary input. In *Asiacrypt 2015*.

[17] B. Braun, A. J. Feldman, Z. Ren, S. T. V. Setty, A. J. Blumberg, and M. Walfish. Verifying computations with state. In *SIGOPS 2013*.

[18] A. Chiesa, E. Tromer, and M. Virza. Cluster computing in zero knowledge. In *Eurocrypt 2015*.

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

[20] C. Costello, C. Fournet, J. Howell, M. Kohlweiss, B. Kreuter, M. Naehrig, B. Parno, and S. Zahur. Geppetto: Versatile verifiable computation. In *IEEE S&P 2015*.

[21] G. Danezis, C. Fournet, J. Groth, and M. Kohlweiss. Square span programs with applications to succinct NIZK arguments. In *Asiacrypt 2014*.

[22] A. Fiat and A. Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *Crypto 1986*.

[23] D. Fiore, C. Fournet, E. Ghosh, M. Kohlweiss, O. Ohrimenko, and B. Parno. Hash first, argue later: Adaptive verifiable computations on outsourced data. In *ACM CCS 2016*.

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

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

[26] A. M. Ghuloum and A. L. Fisher. Flattening and parallelizing irregular, recurrent loop nests. In *PPOPP 1995*.

[27] S. Goldwasser, S. Micali, and C. Rackoff. The knowledge complexity of interactive proof-systems. In *STOC 1985*.

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

[29] J. Groth. On the size of pairing-based non-interactive arguments. In *Eurocrypt 2016*.

[30] J. Groth. Short pairing-based non-interactive zero-knowledge arguments. In *Asiacrypt 2010*.

[31] J. Groth. On the size of pairing-based non-interactive arguments. In *Eurocrypt 2016*, pages 305–326, 2016.

[32] Y. Ishai, E. Kushilevitz, and R. Ostrovsky. Efficient arguments without short PCPs. In *CCC 2007*.

[33] J. Kilian. A note on efficient zero-knowledge proofs and arguments. In *STOC 1992*.

[34] D. E. Knuth, J. H. Morris, Jr, and V. R. Pratt. Fast pattern matching in strings. *SIAM J. Computing*, 6(2):323–350, 1977.

[35] A. E. Kosba, D. Papadopoulos, C. Papamanthou, M. F. Sayed, E. Shi, and N. Triandopoulos. TRUESET: Faster verifiable set computations. In *USENIX Security 2014*.

[36] H. Lipmaa. Succinct non-interactive zero knowledge arguments from span programs and linear error-correcting codes. In *Asiacrypt 2013*.

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

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

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

[40] B. Parno, J. Howell, C. Gentry, and M. Raykova. Pinocchio: Nearly practical verifiable computation. In *IEEE S&P 2013*.

[41] B. Parno, M. Raykova, and V. Vaikuntanathan. How to delegate and verify in public: Verifiable computation from attribute-based encryption. In *TCC 2012*.

[42] 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 2013*.

[43] S. T. V. 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*.

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

[45] J. Thaler. Time-optimal interactive proofs for circuit evaluation. In *Crypto 2013*.

[46] J. Thaler. A note on the GKR protocol, 2015. Available at http://people.cs.georgetown.edu/jthaler/GKRNote.pdf.

[47] V. Vu, S. Setty, A. J. Blumberg, and M. Walfish. A hybrid architecture for interactive verifiable computation. In *IEEE S&P 2013*.

[48] R. S. Wahby, M. Howald, S. Garg, A. Shelat, and M. Walfish. Verifiable asics. In *IEEE SP 2016*, 2016.

[49] R. S. Wahby, Y. Ji, A. J. Blumberg, A. Shelat, J. Thaler, M. Walfish, and T. Wies. Full accounting for verifiable outsourcing. In *ACM CCS 2017*.

[50] R. S. Wahby, S. T. Setty, Z. Ren, A. J. Blumberg, and M. Walfish. Efficient RAM and control flow in verifiable outsourced computation. In *NDSS*, 2015.

[51] M. Walfish and A. J. Blumberg. Verifying computations without reexecuting them. *Comm. ACM*, 58(2):74–84, 2015.

[52] Y. Zhang, D. Genkin, J. Katz, D. Papadopoulos, and C. Papamanthou. vSQL: Verifying arbitrary SQL queries over dynamic outsourced databases. In *IEEE S&P 2017*.

[53] Y. Zhang, D. Genkin, J. Katz, D. Papadopoulos, and C. Papamanthou. A zero-knowledge version of vsql. Cryptology ePrint Archive, Report 2017/1146, 2017.

Appendix A
Arithmetic Circuits and Multilinear Extensions

An *arithmetic circuit* C is a directed acyclic graph whose vertices are called *gates* and whose edges are called *wires*. Every in-degree 0 gate in C is labeled by a variable from a set of variables $X = \{x_1, \cdots, x_n\}$ and is referred to as an input gate. All other gates in C have in-degree 2, are labeled by elements from $\{+, \times\}$ and referred to as addition and multiplication gates, respectively. Every gate of out-degree 0 is called an output gate. In the following, we focus only on *layered* circuits and we assume that the output gates are ordered. We say that a circuit is layered if it can be divided into disjoint sets $L_1, \cdots, L_k$ such that every gate of g belongs to some set L_i and all the wires of C connect gates in two consecutive layers (i.e., between L_j and L_{j+1} for some j). We write $C : \mathbb{F}^n \rightarrow \mathbb{F}^k$ to indicate that C is an arithmetic circuit with n inputs and k outputs evaluated (as defined in a natural way) over a field $\mathbb{F}$. We denote by $|C|$ the number of gates in the circuit C, by $\mathsf{width}_i(C)$ the number of gates in the i-the layer of C and by $\mathsf{width}(C)$ the maximum width of C, i.e., $\mathsf{width}(C) = \max_i\{\mathsf{width}_i(C)\}$.

Polynomial Decomposition. We use the following lemma when proving properties of our VPD protocol.

Lemma 1 ([39]). *Let $f : \mathbb{F}^\ell \to \mathbb{F}$ be a polynomial of variable degree d. For all $t \in \mathbb{F}^\ell$ there exist efficiently computable polynomials $q_1, \dots, q_\ell$ such that: $f(x) - f(t) = \sum_{i=1}^{\ell}(x_i - t_i)q_i(x)$ where t_i is the ith element of t.*

Multilinear Extensions. For any function $V : \{0,1\}^\ell \to \mathbb{F}$ we define the multilinear extension, $\widetilde{V} : \mathbb{F}^\ell \to \mathbb{F}$, of V as follows: $\widetilde{V}(x_1, \cdots, x_\ell) = \sum_{b\in\{0,1\}^\ell} \prod_{i=1}^{\ell} \mathcal{X}_{b_i}(x_i)V(b)$ where b_i is the i-th bit of b, $\mathcal{X}_1(x_i) = x_i$ and $\mathcal{X}_0(x_i) = 1 - x_i$. Note that $\widetilde{V}$ is the unique polynomial that has degree at most 1 in each of its variables that satisfies $\widetilde{V}(x) = V(x)$ for all $x \in \{0,1\}^\ell$.

Multilinear Extensions of Arrays. An array $A = (a_0, \cdots, a_{n-1})$ where $a_i \in \mathbb{F}$ can be viewed as a function $A : \{0,1\}^{\log n} \to \mathbb{F}$ such that $A(i) = a_i$ for all $0 \le i \le n-1$. In the sequel, we abuse the terminology of multilinear extensions, by defining (in the natural way) a multilinear extension $\tilde{A}$ of an array A. A useful property of multilinear extensions of arrays is the ability to efficiently combine them. That is, given 2^m arrays $A_1, \cdots, A_{2^m}$ of equal length n, the multilinear extensions of the array corresponding to their concatenation $A = A_1||\cdots||A_{2^m}$ can be evaluated on a point $x = (x_1, \cdots, x_{m+\log n})$ as

$$\tilde{A}(x_1, \cdots, x_{m+logn}) = \sum_{i=0}^{2^m-1} \prod_{j=1}^{m} \mathcal{X}_{i_j}(x_j)\tilde{A}_i(x_{m+1}, \cdots, x_{m+\log n})$$

where i_j is the j-th bit of i and $\mathcal{X}_{i_j}(x_j)$ is defined above.

Appendix B
Cryptographic Assumptions

Our constructions make use of the following assumptions.

Assumption 1 ([15] (q-Strong Diffie-Hellman)). *For any* PPT *adversary $\mathcal{A}$, the following probability is negligible:*

$$\Pr\left[\begin{array}{c} \mathsf{bp} \leftarrow \mathsf{BilGen}(1^\lambda); \\ s \xleftarrow{R} \mathbb{Z}_p^*; \\ \sigma = (\mathsf{bp}, g^s, \dots, g^{s^q}) \end{array} : (x, e(g,g)^{\frac{1}{s+x}}) \leftarrow \mathcal{A}(1^\lambda, \sigma)\right].$$

The following is a direct generalization of Groth's q-PKE assumption [30] for multivariate polynomials.

Assumption 2 (((d,ℓ)-Power Knowledge of Exponent [52]). *For any* PPT *adversary $\mathcal{A}$ there is a polynomial-time algorithm $\mathcal{E}$ (running on the same random tape) such that for all benign auxiliary inputs $z \in \{0,1\}^{poly(\lambda)}$ the following probability is negligible:*

$$\Pr\left[\begin{array}{c} \mathsf{bp} \leftarrow \mathsf{BilGen}(1^\lambda); \\ s_1, \dots, s_\ell, \alpha \xleftarrow{R} \mathbb{Z}_p^*, s_0 = 1; \\ \sigma_1 = \{g^{\prod_{i\in W} s_i}\}_{W\in\mathcal{W}_{\ell,d}}; \\ \sigma_2 = \{g^{\alpha\cdot\prod_{i\in W} s_i}\}_{W\in\mathcal{W}_{\ell,d}}; \\ \sigma = (\mathsf{bp}, \sigma_1, \sigma_2, g^\alpha); \\ \mathbb{G}\times\mathbb{G} \ni (h, \tilde{h}) \leftarrow \mathcal{A}(1^\lambda, \sigma, z); \\ (a_0, \dots, a_{|\mathcal{W}_{\ell,d}|}) \leftarrow \mathcal{E}(1^\lambda, \sigma, z) \end{array} : \begin{array}{c} e(h, g^\alpha) = e(\tilde{h}, g) \\ \wedge \\ \prod_{W\in\mathcal{W}_{\ell,d}} g^{a_W \prod_{i\in W} s_i} \\ \neq h \end{array}\right].$$

In the above, we assume that z comes from a benign distribution (similar to [20], [29], [23]), in order to avoid the negative results of [16], [13]. Concretely, our proofs hold assuming the auxiliary input necessary for extraction comes from a benign distribution.

To simplify the exposition, we assume symmetric (Type I) pairings. However, since asymmetric pairings are more efficient in practice, our implementations use a version of our constructions based on asymmetric pairings; our assumptions can be re-stated for that setting in a straightforward manner.

Appendix C
The Sum-Check Protocol

Introduced in [37], the sum-check protocol allows a prover $\mathcal{P}$ to convince a verifier $\mathcal{V}$ that

$$H = \sum_{b_1\in\{0,1\}} \sum_{b_2\in\{0,1\}} \cdots \sum_{b_\ell\in\{0,1\}} g(b_1, b_2, \cdots, b_\ell)$$

where $g(x_1, \cdots, x_\ell)$ is an ℓ-variate polynomial over some finite field $\mathbb{F}$. While the direct computation of H will require $\mathcal{V}$ to evaluate g at least 2^ℓ times, $\mathcal{V}$'s work can be made polynomial in ℓ using the sum-check protocol which we now describe. Indeed, the protocol proceeds in ℓ rounds. During the first round, $\mathcal{P}$ sends $\mathcal{V}$ the following univariate polynomial $g_1(x) = \sum_{b_2,\cdots,b_\ell\in\{0,1\}} g(x, b_2, \cdots, b_\ell)$. Next, $\mathcal{V}$ checks that the degree of x in g_1 is at most the degree of x_1 in g and that $H = g_1(0) + g_1(1)$, rejecting if any of these checks fails. In case both checks pass, $\mathcal{V}$ sends $\mathcal{P}$ a uniform challenge r_1. During the i-th round of the protocol, $\mathcal{P}$ sends the polynomial $g_i(x) = \sum_{b_{i+1},\cdots,b_\ell\in\{0,1\}} g(r_1, \cdots, r_{i-1}, x, b_{i+1}, \cdots, b_\ell)$. $\mathcal{V}$ then checks that $g_{i-1}(r_{i-1}) = g_i(0) + g_i(1)$, rejecting otherwise. In case the check passes, $\mathcal{V}$ sends a uniform r_i to $\mathcal{P}$, to be used in the next round. At the final round, $\mathcal{V}$ accepts only if $g(r_1, \cdots, r_\ell) = g_\ell(r_\ell)$. Define the degree of each monomial in g as the sum of the powers of its variables. The total degree of g is defined as the maximal degree of any of its monomial.

Theorem 4 ([37]). *For any ℓ-variate, total-degree-d polynomial g over finite field $\mathbb{F}$, the above-described sum-check protocol is an interactive proof for the (no-input) function $\sum_{b_1,\cdots,b_\ell\in\{0,1\}} g(b_1, \cdots, b_\ell)$ with soundness $d\cdot\ell/|\mathbb{F}|$. Moreover, $\mathcal{V}$ performs poly(ℓ) arithmetic operations over $\mathbb{F}$ and one evaluation of g on a random point r.*

Remark 1. *When g is a multilinear polynomial (the degree of each variable is at most 1, and the total degree is ℓ), the running time of $\mathcal{P}$ in round i of the sum-check protocol is $\min\{O(m), O(2^{\ell-i})\}$, where m is the total number of distinct monomials in g [19], [45], [47].*

Appendix D
Formal Description of the CMT Protocol

In this section, we describe the final part of the CMT protocol which condenses to a single evolution per circuit layer, we present the formal description of the CMT protocol (in Construction 3), and we state the corresponding theorem.

Condensing to a Single Claim Per Layer. Let $\gamma : \mathbb{F} \to \mathbb{F}^{s_1}$ be the unique line defined by $\gamma(0) = q_1$ and $\gamma(1) = q_2$. The

Construction 3 (**CMT protocol**)**.** *Let $C : \mathbb{F}^n \to \mathbb{F}$ be a depth-d layered arithmetic circuit over a finite field $\mathbb{F}$. Let $x \in \mathbb{F}^n$ be inputs of C such that $C(x) = 1$. In order for the prover $\mathcal{P}_{cmt}$ to convince the verifier $\mathcal{V}_{cmt}$ that $C(x) = 1$, the protocol proceeds as follows.*

1) *Both parties set $a_0 = 1$ and $r_0 = 0$. The protocol process as follows.*
2) *For $i = 1, \ldots, d$, the protocol proceeds as follows.*
   a) *$\mathcal{P}_{cmt}$ and $\mathcal{V}_{cmt}$ run the sum-check protocol for value a_{i-1} and polynomial $f_{i-1,r_{i-1}}$ as per Equation (1). In the last step of the sum-check protocol, $\mathcal{V}_{sc}$ is supposed to evaluate $f_{i-1,r_{i-1}}$ at a random point ρ_i. $\mathcal{P}_{sc}$ then provides values (a_1, a_2) for which it claims that $a_1 = \tilde{V}_i(q_1)$ and that $a_2 = \tilde{V}_i(q_2)$ where q_1, q_2 are the last $2s_i$ elements of ρ_i.*
   b) *Let $\gamma : \mathbb{F} \to \mathbb{F}^{s_i}$ be the line defined by $\gamma(0) = q_1$ and $\gamma(1) = q_2$. $\mathcal{P}_{cmt}$ sends the degree-s_i polynomial $h_i(x) = \tilde{V}_i(\gamma(x))$. Next, $\mathcal{V}$ verifies that $h(0) = a_1$ and $h(1) = a_2$. In case both check pass, $\mathcal{V}_{cmt}$ chooses uniformly at random $r'_i \in \mathbb{F}$, sets $a_i = h(r'_i)$, $r_i = \gamma(r'_i)$ and sends (r_i, a_i) to $\mathcal{P}_{cmt}$.*
3) *$\mathcal{V}$ accepts if $a_d = \tilde{V}_d(r_d)$, where $\tilde{V}_d$ is the multilinear extension of the polynomial representing the input x.*

CMT prover $\mathcal{P}_{cmt}$ sends a degree-s_1 polynomial h claimed to be $\widetilde{V}_1(\gamma(\cdot))$ (i.e., the restriction of $\widetilde{V}_1$ to the line γ). The CMT verifier $\mathcal{V}_{cmt}$ then checks that $h(0) = a_1$ and that $h(1) = a_2$. In case both checks pass, $\mathcal{V}_{cmt}$ picks a random point r_1 and initiates a *single* execution of the sum-check protocol in order to verify that $h(r_1) = \widetilde{V}_1(\gamma(r_1))$. Thus, this condensing procedure reduces the total number of invocations of the sum-check protocol was from $O(2^d)$ to $O(d)$.

So far, we have assumed that the circuit C has only a single output value $y \in \{0, 1\}$. Larger outputs can be handled [47] by having the initial claim made by the prover to be stated directly about the multilinear extension of the claimed circuit output. Formally, consider the following theorem regarding the CMT protocol presented in Construction 3.

Theorem 5 ([28], [19], [47], [45])**.** *Let $C : \mathbb{F}^n \to \mathbb{F}^k$ be a depth-d layered arithmetic circuit over a finite field $\mathbb{F}$. The protocol presented in Construction 3 is an interactive proof for the function computed by C with soundness $O(d \cdot \log S/|\mathbb{F}|)$, where S is the maximal number of gates per circuit layer. Moreover, $\mathcal{P}$'s running time is $O(|C| \log S)$ and the protocol uses $O(d \log S)$ rounds of interaction. Finally, if $\tilde{\mathsf{add}}_i$ and $\tilde{\mathsf{mult}}_i$ are computable in time $O(\mathrm{polylog} S)$ for all the layers of C, then the running time of the verifier $\mathcal{V}$ is $O(n + k + d \cdot \mathrm{polylog} S)$.*

The following remark is particularly useful in case the circuit C being evaluated has a highly regular repetitive structure.

Remark 2 ([45])**.** *If C can be expressed as a composition of (i) parallel copies of a layered circuit C' whose maximum number of gates at any layer is S', and (ii) a subsequent "aggregation" layered circuit C'' of size $O(|C|/\log|C|)$, the running time of $\mathcal{P}$ is reduced to $O(|C| \log |S'|)$.*

Appendix E
Further Reducing the Cost of Memory Checking

Even after our RAM optimizations from Section III, the circuit C contains T copies of C_{mem}. In practice however, it is almost certain that not every cycle will perform a memory access. E.g., even for a program that consists of a single for loop that simply loads a memory location per repetition, the total percentage of memory accesses is 25% (one instruction for the memory load, plus three for counter increase, loop bound check, and jump). Motivated by this, we exploit the circuit-independent pre-processing of our argument to modify C so that it only contains αT copies of C_{mem} where α is the percentage of general memory accesses over the total steps.

In order to achieve this, we split the witness to two separate parts. The first contains tr and tr^{**} sorted by time and instruction type, and the second contains tr^*. Recall that after the optimization in Section III, tr^* only contains $A_{T+\ell}, \cdots, A_{2T+\ell}$, which is a permutation of $S_1, \cdots, S_T$ sorted by accessed memory addresses. Then, by our design, if I_i is not a memory load/store instruction, we set the accessed memory address of S_i as 0 and all the values in S_i as 0s before sorting. In this way, the first $(1-\alpha)T$ states in $A_{T+\ell}, \cdots, A_{2T+\ell}$ are all zeros (assuming the real memory address starts from 1) and there is no need to check anything for these states, as they are not memory operations. Because of this layout, now the prover only includes $A_{T+\ell+\alpha T}, \cdots, A_{2T+\ell}$ in tr^*, and tells the verifier the number of non-memory operations. With these information, it is sufficient to validate the new tr^* is a permutation of non-zero states in tr using CMT on circuit C', and the technique is described in [52] for handling circuits that receive inputs at different levels. With this optimization, we manage to reduce the number of C_{mem} further from T to αT, which is a significant improvement in practice. However, the verifier now needs to run two VPD instances (once for each part of the witness). See [52] for a more detailed explanation.

Appendix F
Complexity of the Modified CMT

We now analyze the complexity of the sum-check protocol of equation 2 in Section IV-A. For the first $2s_{i+1}$ rounds, there are at most BS_i monomials per round, as there are at most BS_i gates in the i-th layer of the circuit and the number of non-zero monomials in $\tilde{\mathsf{add}}_{i+1}$ and $\tilde{\mathsf{mult}}_{i+1}$ is bounded by the number of gates. By Remark 1, this takes $O(BS_i)$ arithmetic operations per round, so the complexity for these rounds is $O(BS_i \log S_{i+1})$. For the remaining rounds, by Remark 1, $\mathcal{P}$'s running time is $O(2^{\lceil \log B \rceil - j})$ in round $2s_{i+1} + j$ ($j = 1, \ldots, \lceil \log B \rceil$) and the complexity is $O(B)$. Thus, the complexity is dominated by the first part, i.e., $O(BS_i \log S_{i+1})$.

Appendix G
Analysis of our New VPD Scheme

Theorem 6. *Under Assumptions 1 and 2, Construction 1 is an extractable VPD scheme. For a variable-degree-d ℓ-variate polynomial $f \in \mathcal{F}$ containing m monomials, algorithm* KeyGen *runs in time $O(\binom{\ell(d+1)}{\ell d})$,* Commit *in time $O(m)$,* Evaluate *in time $O(\ell d m)$,* Ver *in time $O(\ell)$ and* CheckCom *in time $O(1)$. If $d = 1$,* Evaluate *runs in time $O(2^{\ell})$. The commitment produced* *by* Commit *consists of $O(1)$ group elements, and the proof produced by* Evaluate *consists of $O(\ell)$ elements of $\mathbb{G}$.*

Proof. The completeness requirement immediately follows from the construction of $(\mathsf{KeyGen}, \mathsf{Commit}, \mathsf{Evaluate}, \mathsf{Ver})$.

We now prove the extractability property. Let $\mathcal{A}$ be a PPT adversary that on input $(1^\lambda, \mathsf{pp})$, where $(\mathsf{pp}, \mathsf{vp})$ is the output of $\mathsf{KeyGen}(1^\lambda, \ell, d)$, outputs commitment com^* such that $\mathsf{CheckCom}(\mathsf{com}^*, \mathsf{vp})$ accepts. This implies that $e(c_1, g^\alpha) = e(c_2, g)$ where $\mathsf{com}^* \overset{\text{def}}{=} (c_1, c_2)$. By Assumption 2, there exists PPT extractor $\mathcal{E}'$ for $\mathcal{A}$ such that upon the same input as $\mathcal{A}$, and with access to the same random tape, outputs $a_0, \dots, a_{|\mathcal{W}_{\ell,d}|} \in \mathbb{F}$ such that $\prod_{W \in \mathcal{W}_{\ell,d}} g^{a_W \prod_{i \in W} s_i} = c_1$, except with negligible probability. Note that, the coefficients $(a_0, \dots, a_{|\mathcal{W}_{\ell,d}|})$ can be encoded as a variable-degree-d, ℓ-variate polynomial that has a_i as its monomial coefficients. We now build extractor $\mathcal{E}$:

1) Upon input $(1^\lambda, \mathsf{pp})$, $\mathcal{E}$ runs $\mathcal{E}'$ on the same input.
2) $\mathcal{E}$ tries to parse the output of $\mathcal{E}'$ as $a_0, \dots, a_{|\mathcal{W}_{\ell,d}|} \in \mathbb{F}$ and aborts if this fails.
3) $\mathcal{E}$ outputs f', where $f' \in \mathcal{F}$ is the polynomial with coefficients $a_0, \dots, a_{|\mathcal{W}_{\ell,d}|}$.

Note that $\mathcal{E}$ is PPT as $\mathcal{E}'$ is PPT and it only performs polynomially many operations in $\mathbb{F}$. It remains to argue that f' is a valid pre-image of Commit except with negligible probability. Observe that, if $\mathcal{E}$ does not abort, it follows from the construction of Commit that $\mathsf{Commit}(f', \mathsf{pp}) = \mathsf{com}$, where com is the output commitment of $\mathcal{A}$. By assumption 2, the probability that the output $\mathcal{E}'$ is not a valid set of coefficients is negligible which concludes the proof.

Next, we prove the soundness property. Let $\mathcal{A}$ be a PPT adversary that wins the soundness game with non-negligible probability. For $i = 1, \dots, \ell$ we define adversary $\mathcal{A}_i$ that receives the same input as $\mathcal{A}$ and executes the same code, but outputs only $(\pi_i, \pi_i') \in \pi^*$ (where π^* is the proof output by $\mathcal{A}$). Moreover, since $\mathcal{A}$ is PPT, all these adversaries are also PPT. Thus, for $i = 1, \dots, \ell$, from Assumption 2 there exists PPT $\mathcal{E}_i$ (running on the same random tape as $\mathcal{A}_i$) which on input $(1^\lambda, \mathsf{pp})$ outputs $a_{0,i}, \dots, a_{|\mathcal{W}_{\ell,d}|,i} \in \mathbb{F}$ such that the following holds: If $e(\pi_i, g^\alpha) = e(\pi_i', g)$ then $\prod_{W \in \mathcal{W}_{\ell,d}} g^{a_{W,i} \prod_{j \in W} s_j} \neq \pi_i$, except with negligible probability. Note that, the coefficients $(a_{0,i}, \dots, a_{|\mathcal{W}_{\ell,d}|,i})$ for $i = 1, \dots, \ell$ can always be encoded as a variable-degree-d, ℓ-variate polynomial which we denote by $q_i'(\mathbf{x})$ for undefined variable $\mathbf{x} = (x_1, \dots, x_\ell)$.

We construct an adversary $\mathcal{B}$ that breaks Assumption 1. On input $(1^\lambda, p, \mathbb{G}, \mathbb{G}_T, e, g, g^s, g^{s^2}, \dots, g^{s^{\ell \cdot d}})$, $\mathcal{B}$ does the following:

Parameter Generation. $\mathcal{B}$ implicitly sets $s_1 = s$ and for $i = 1, \dots, \ell$ he chooses $r_i \in \mathbb{F}$ uniformly at random and sets (also implicitly) $s_i = s \cdot r_i$. Then he chooses uniformly at random a value $\alpha \in \mathbb{F}$. Next $\mathcal{B}$ needs to generate the terms in $\mathbb{P} = \{g^{\prod_{i \in W} s_i},\ g^{\alpha \cdot \prod_{i \in W} s_i}\}_{W \in \mathcal{W}_{\ell,d}}$. Since the exponent of each term is a product of at most $\ell \cdot d$ factors where each factor is one of the values $s_i = s \cdot r_i$, it can be written as a polynomial in s with degree at most $\ell \cdot d$. Therefore, $\mathcal{B}$ can compute these terms from the values $g, g^s, g^{s^2}, \dots, g^{s^{\ell \cdot d}}$ and α. Finally, $\mathcal{B}$ runs $\mathcal{A}$ on input $(1^\lambda, \mathsf{pp})$, where $\mathsf{pp} = (p, \mathbb{G}, \mathbb{G}_T, e, g, g^\alpha, \mathbb{P})$.

Query Evaluation. Upon receiving (f^*, t^*, y^*, π^*) from $\mathcal{A}$, $\mathcal{B}$ first runs $\mathsf{Commit}(f^*, \mathsf{pp})$ to receive $\mathsf{com} \overset{\text{def}}{=} (c_1, c_2)$ and then runs $\mathsf{Ver}(\mathsf{com}, t^*, y^*, \pi^*, \mathsf{vp})$ where $\mathsf{vp} = (1^\lambda, p, \mathbb{G}, \mathbb{G}_T, e, g, g^s, g^{s^2}, \dots, g^{s^{\ell \cdot d}}, g^\alpha)$. If Ver rejects, $\mathcal{B}$ aborts, else he runs extractors $\mathcal{E}_1, \dots, \mathcal{E}_\ell$ (defined above) on the same input as $\mathcal{A}$ and receives polynomials $q_1', \dots, q_\ell'$. If for the output of any of the $\mathcal{E}_i$ it holds that $\prod_{W \in \mathcal{W}_{\ell,d}} g^{a_{W,i} \prod_{j \in W} s_j} \neq \pi_i$, $\mathcal{B}$ aborts. Otherwise, let $\delta = y^* - f^*(t^*)$ and let $Q(\mathbf{x})$ be the polynomial over $\mathbb{F}$ defined as $Q(\mathbf{x}) \overset{\text{def}}{=} f^*(\mathbf{x}) - f^*(t^*) - \sum_{i=1}^{\ell} (x_i - t_i) q_i'(\mathbf{x})$ where $t^* \overset{\text{def}}{=} (t_1, \dots, t_\ell)$. $\mathcal{B}$ picks $\tau \in \mathbb{F}$ uniformly at random. If $g^\tau = g^{-s}$, he sets $\tau \leftarrow \tau + 1$. He then computes polynomial $Q'(x) \overset{\text{def}}{=} Q(\mathbf{x})/(\tau + x_1)$ and finally outputs $(\tau, e(g,g)^{\delta^{-1} \cdot Q'(s_1, \dots, s_\ell)})$ as a challenge tuple for Assumption 1.

Since $s_1 = s, s_2 = r_2 \cdot s, \dots, s_\ell = r_\ell \cdot s$, we have $Q'(s_1, \dots, s_\ell) = Q''(s)$ where Q'' is an efficiently computable univariate polynomial of degree $\ell \cdot d$ hence $e(g,g)^{-\delta \cdot Q'(s_1, \dots, s_\ell)}$ is computable from $(1^\lambda, p, \mathbb{G}, \mathbb{G}_T, e, g, g^s, g^{s^2}, \dots, g^{s^{\ell \cdot d}})$. $\mathcal{B}$ is clearly PPT since all of $\mathcal{E}_i$ are PPT and he performs polynomially many operations in $\mathbb{F}, \mathbb{G}, \mathbb{G}_T$. Next, we analyze the success probability of $\mathcal{B}$. Recall that, by assumption $\mathcal{A}$ succeeds in violating soundness with probability ε. We observe that, *conditioned on not aborting*, $\mathcal{B}$'s output is always a valid tuple for breaking Assumption 1. Let us argue why this is true. Since verification succeeds, it holds that $e(c_1/g^{y^*}, g) = \prod_{i=1}^{\ell} e(g^{s_i - t_i}, \pi_i)$; since extraction succeeds, this can be replaced with

$$e(g,g)^{f^*(s_1, \dots, s_\ell) - \delta - f^*(t^*)} = \prod_{i=1}^{\ell} e(g^{s_i - t_i}, g^{q_i'(s_1, \dots, s_\ell)})$$

$$e(g,g)^\delta = e(g,g)^{f^*(s_1, \dots, s_\ell) - f^*(t^*)} \prod_{i=1}^{\ell} e(g^{s_i - t_i}, g^{-q_i'(s_1, \dots, s_\ell)})$$

$$e(g,g)^\delta = e(g,g)^{f^*(s_1, \dots, s_\ell) - f^*(t^*) - \sum_{i=1}^{\ell} (s_i - t_i) q_i'(s_1, \dots, s_\ell)}.$$

By the definition of Q' it follows that

$$e(g,g)^\delta = e(g,g)^{Q(s_1, \dots, s_\ell)}$$

$$e(g,g)^{\frac{\delta}{\tau + s_1}} = e(g,g)^{\frac{Q(s_1, \dots, s_\ell)}{\tau + s_1}} = e(g,g)^{Q'(s_1, \dots, s_\ell)}$$

$$e(g,g)^{\frac{1}{\tau + s_1}} = e(g,g)^{\delta^{-1} \cdot Q'(s_1, \dots, s_\ell)}.$$

Thus, the final piece in order to conclude the proof is to bound the probability that $\mathcal{B}$ aborts. Note that, conditioned on $\mathcal{A}$ winning, $\mathcal{B}$ will only abort if extraction fails which can only happen with negligible probability $\mathsf{neg}(\lambda)$. This holds since, if verification succeeds it must be that $e(\pi_i', g) = e(\pi_i, g^\alpha)$ for $i = 1, \dots, \ell$ and in this case, by Assumption 2, extraction for any of $\mathcal{E}_1, \dots, \mathcal{E}_\ell$ fails with negligible probability. Since ℓ is polynomial in λ it follows that the probability any of them fails (which by a union bound is at most equal to the sum of each individual failure probability) is also negligible. Finally, let us argue that the polynomial division $Q(\mathbf{x})/(\tau + x_1)$ is always possible. Recall, that for polynomials defined over finite fields division is always possible assuming that the divident's degree is at least as large as that of the divisor's. Moreover, the degree of the quotient is at most that of the divident's and that of the remainder is strictly smaller than that of the divisor. Let us assume for contradiction that $Q(\mathbf{x})$ is a constant polynomial. Since, $e(g,g)^{\delta} = e(g,g)^{Q(s_1,\ldots,s_{\ell+1})}$ and $e(g,g)$ is a generator or $\mathbb{G}_T$, it must be that $Q(\mathbf{x}) \overset{\text{def}}{=} \delta$ therefore we can write

$$-\delta = \sum_{i=1}^{\ell}(x_i - t_i)q_i'(\mathbf{x}) - f^*(\mathbf{x}) + f^*(t^*)$$

$$f^*(\mathbf{x}) - \delta - f^*(t^*) = \sum_{i=1}^{\ell}(x_i - t_i)q_i'(\mathbf{x})$$

$$f^*(\mathbf{x}) - y^* = \sum_{i=1}^{\ell}(x_i - t_i)q_i'(\mathbf{x})$$

From the above relation it follows that t^* is a root of the polynomial $f' \overset{\text{def}}{=} f^*(\mathbf{x}) - y^*$, i.e., $f'(t^*) = 0$ which implies that $f^*(t_1,\ldots,t_\ell) = y^*$. Thus, in this case, y^* is the correct evaluation of f^* on t^*, i.e., $\delta = 0$ and $\mathcal{A}$ did not cheat. In all other cases, the polynomial division is possible.

From the above analysis it follows that the probability that $\mathcal{B}$ succeeds is at least $(1 - \mathsf{neg}(\lambda))\varepsilon$. By assumption, ε is the non-negligible probability that $\mathcal{A}$ wins the soundness game, therefore $\mathcal{B}$'s success probability is also non-negligible. This contradicts Assumption 1 and our proof is complete.

Asymptotic Analysis. The claims for the general polynomial case follow directly from the analysis of [39]. For $d = 1$, i.e., for multi-linear polynomials, we prove the tighter bound for the runtime of $\mathsf{Evaluate}$ below.

Recall that during $\mathsf{Evaluate}$ the prover computes polynomials $q_i(x_i,\ldots,x_\ell)$ for $i = 1,\ldots,\ell$, such that $f(x_1,\ldots,x_\ell) = \sum_{i=1}^{\ell}(x_i - t_i)\cdot q_i(x_i,\ldots,x_\ell) + f(t_1,\ldots,t_\ell)$ and proof $\pi = \{g^{q_i(s_i,\ldots,s_\ell)}, g^{\alpha q_i(s_i,\ldots,s_\ell)}\}_{i=1}^{\ell}$. We start by computing $q_1(x_1,\ldots,x_\ell)$. Since the degree of every variable is at most 1, the multi-linear polynomial f can be written as $f(x_1,\ldots,x_\ell) = g(x_2,\ldots,x_\ell) + x_1 \cdot h(x_2,\ldots,x_\ell)$, where $g(x_2,\ldots,x_\ell)$ and $h(x_2,\ldots,x_\ell)$ are multi-linear polynomials of variables $x_2,\ldots,x_\ell$. In this way, f can be decomposed as

$$\begin{aligned} f(x_1,\ldots,x_\ell) &= g(x_2,\ldots,x_\ell) + x_1 \cdot h(x_2,\ldots,x_\ell) \\ &= (g(x_2,\ldots,x_\ell) + t_1 \cdot h(x_2,\ldots,x_\ell)) + (x_1 - t_1)h(x_2,\ldots,x_\ell) \\ &= R_1(x_2,\ldots,x_\ell) + (x_1 - t_1)h(x_2,\ldots,x_\ell). \end{aligned}$$

We set $q_1(x_1,\ldots,x_\ell) = h(x_2,\ldots,x_\ell)$ (which means q_1 contains no monomial with x_1), and proceed to decompose the multi-linear polynomial $R_1(x_2,\ldots,x_\ell)$ with $\ell - 1$ variables in the same way as f to compute $q_2(x_2,\ldots,x_\ell)$. Regarding the complexity of this, note that both $g(x_2,\ldots,x_\ell)$ and $h(x_2,\ldots,x_\ell)$ contain at most $2^{\ell-1}$ monomials. Therefore, it takes $2^{\ell-1}$ additions and multiplications to compute $q_1(x_1,\ldots,x_\ell)$ and $R_1(x_2,\ldots,x_\ell)$, and $2^{\ell-1}$ exponentiations to generate $g^{q_1(s_1,\ldots,s_\ell)}$ and $g^{\alpha q_1(s_1,\ldots,s_\ell)}$ in the proof, respectively. The exact same reasoning applies for all of $q_3,\ldots,q_\ell$. At the last step after computing $q_\ell(x_\ell)$, the remaining constant term is equal to the answer $f(t_1,\ldots,t_\ell)$. In general, in the ith step, we are decomposing $R_{i-1}(x_i,\ldots,x_\ell)$ with $\ell - i + 1$ variables in the same way above to compute $q_i(x_i,\ldots,x_\ell)$ and $R_i(x_{i+1},\ldots,x_\ell)$, and the complexity is $O(2^{\ell-i})$. Thus, the total complexity of computing $q_1,\ldots,q_\ell$ is $O(2^{\ell-1}) + O(2^{\ell-2}) + \ldots = O(2^\ell)$. The polynomial evaluation in order to get the answer takes the same time. Each pair π_i, π_i' is computed with two exponentiations, thus the overall running time is $O(2^\ell)$. □

Bench -mark	VPD Time/Input	CMT Time/Gate	Buffet Time/Gate	vnTinyRAM Time/Gate
#2	11.32	5.42	77.50	72.20
#3	13.17	6.36	68.34	72.15
#4	13.23	6.18	74.97	72.33
#5	12.90	5.77	72.10	72.37

TABLE V
PER GATE (INPUT) PROVER TIME FOR OUR VPD AND CMT, BUFFET AND VNTINYRAM FOR THE LAST 4 RAM PROGRAMS IN THE BENCHMARK (SAME ORDER AND SIZE AS IN TABLE III). TIME REPORTED IN μS.

APPENDIX H
MICROBENCHMARKS

Verifiable Polynomial Delegation. Table V shows the prover time of our implementation of the VPD from Section IV-B. The prover time is about $12\mu s$ per input gate, which is about $8\times$ faster than that of [52]. This is due to (i) our improved VPD construction (amounting to around 2-4$\times$) and (ii) due to the fact that 85% of the inputs used in our RAM reduction are field elements that encode single bit values (due to bit decomposition, register indices and flags), which leads to faster exponentiation times for the VPD prover.

CMT Protocol. Next, we evaluate the performance of our CMT protocol. As can be seen in Table V, the average time required per gate for the CMT prover is about $6\mu s$, which is about $4\times$ slower than the $1.7\mu s$ number reported in [52]. This is because we implemented our new CMT protocol supporting circuits with different copies of sub-circuits in Section IV-A, while the CMT protocol for regular circuits is used in [52]. Both the per-input time for the VPD protocol and the per- gate time for the CMT protocol are much faster than the per-gate time for Buffet and vnTinyRAM.

Circuit Generator. Finally, we report the number of gates required by our reduction to verify a TinyRAM cycle. We measure this by dividing the total number of gates of the circuits produced by the experimental evaluation of Section V-A over the number of TinyRAM steps. For our tested programs this circuit contained about 2500 gates, 600 of which are multiplications (for comparison, vnTinyRAM takes roughly 2000 multiplication gates, as reported in [11]. Notice that the work of [11] only needs to report the number of multiplication gates while we must report on the total number of gates.