# Efficient RAM and control flow in verifiable outsourced computation

Riad S. Wahby*, Srinath Setty†, Zuocheng Ren†, Andrew J. Blumberg†, and Michael Walfish*

*New York University †The University of Texas at Austin

**Abstract.** Recent work on *proof-based verifiable computation* has resulted in built systems that employ tools from complexity theory and cryptography to address a basic problem in systems security: allowing a local computer to outsource the execution of a program while providing the local computer with a guarantee of integrity and the remote computer with a guarantee of privacy. However, support for programs that use RAM and complicated control flow has been problematic. State of the art systems restrict the use of these constructs (e.g., requiring static loop bounds), incur sizable overhead on every step to support these constructs, or pay tremendous costs when the constructs are invoked.

This paper describes Buffet, a built system that solves these problems by providing inexpensive "a la carte" RAM and dynamic control flow constructs. Buffet composes an elegant prior approach to RAM with a novel adaptation of techniques from the compiler community. The result is a system that allows the programmer to express programs in an expansive subset of C (disallowing only "goto" and function pointers), can handle essentially any example in the verifiable computation literature, and achieves the best performance in the area by multiple orders of magnitude.

## 1 Introduction

How can a client outsource a computation to a server and then check that the server executed correctly?[1] And can this be done in a way that allows the server to supply private inputs and keep them confidential? Variants of this problem have been around for decades [11]; today, cloud computing is a particularly pertinent use case. Indeed, because cloud providers are large-scale, we cannot assume that execution is always correct; because they are opaque, we cannot assume that the causes of incorrect execution (corruption of data, hardware faults, malice, and more) are readily detectable. And many common cloud applications involve private server input that must remain confidential (e.g., database interactions).

Classical solutions to this problem depend on potentially undesirable assumptions or restrictions. For example, replication [28, 29, 56] assumes that replica failures are not correlated (which does not hold in homogeneous cloud platforms). Auditing [47, 59] assumes that failures follow a most-or-none distribution. Trusted hardware and attestation [62, 66–68] assumes that the hardware is not faulty. Tailored solutions exist (see [61, 71, 77] for surveys) but only for restricted classes of computations.

Over the last few years, a new solution has emerged, called *proof-based verifiable computation* [79], that gives comprehensive guarantees, makes few or no assumptions about the prover, and applies generally [15, 18, 27, 33, 37, 40, 52, 61, 69–72, 74, 75, 77]. Although the details differ among these works, all of them are based on sophisticated cryptographic and complexity-theoretic machinery: probabilistically checkable proofs (PCPs) [8, 9], efficient arguments [20, 21, 25, 40, 41, 46, 49] (including zero-knowledge variants), interactive proofs [10, 44, 45, 54, 73], etc. To be clear, it had long been known that this machinery was relevant to verifying outsourced computations [11]; the work of proof-based verifiable computation has been refining the theory and building systems around it.

Indeed, publications in this area have showcased dramatic performance and usability improvements relative to naive implementations of the theory: factors-of-a-trillion speedups; compilers; and sophisticated

[1] Checking that a given program is expressed correctly is program verification, which is a different but complementary problem.

implementations on smart phones, on GPUs, and across distributed servers. As a notable example, recent work [37], building on [40, 61], compiles zero-knowledge applications (that preserve the confidentiality of the server's private inputs) into a form that is practical for real use.

All of this work has taken place in the context of built systems that have two major components: a *front-end* translates programs into the formalism required by a cryptographic and complexity-theoretic *back-end*. In more detail, the front-end translates a program that is expressed in a high-level language into a system of equations, or set of *constraints*; a solution to these constraints corresponds to a valid execution of the computation. The back-end is a probabilistic proof protocol [43] (particularly an interactive argument [46, 49] or a non-interactive argument [20, 22, 40, 61]) by which the server (or *prover*) convinces the client (or *verifier*) that it holds a solution to the constraints.

The guiding intuition for the area is that the theoretical advantages of the back-end proof protocol should result in efficient and powerful systems: the prover can keep its solution private (when using zero-knowledge variants), and the verifier handles only a short certificate, the checks of which are in principle very efficient—far more so than performing the computation locally. However, there is overhead from the front-end, the back-end, and their interaction. This overhead manifests most prominently in setup costs incurred by the verifier and the costs paid by both verifier and prover for each input-output instance that the verifier wishes to check.

After a great deal of work, a single approach to the back-end has emerged. Although the different systems have different properties [15, 18, 22, 27, 37, 52, 70], the core probabilistically checkable encoding in all of them is now the remarkable construction of GGPR [40] (or is based on it [52]). This encoding has slashed prover costs and verifier setup costs—though neither cost is low by usual systems standards. Furthermore, there has been a real victory: the verifier's per-instance costs are genuinely inexpensive. In fact, under certain usage models [27, 31, 34], the verifier's total costs (amortized setup plus incremental) can be considered practical.

The front-end has also been a locus of activity, but the situation there is far less clear. Currently, there is a tradeoff between programmability and costs [79, Fig. 2][18, §5.4], specifically the verifier's setup costs and prover costs, which are largely driven by the number of constraints that represent a computation. This tradeoff is clear from the two major front-end approaches.

One approach is TinyRAM [18], which is currently the state of the art in an elegant line of work [14, 15, 22]. Here, the constraints represent the unrolled execution of a general-purpose CPU. One of the inputs to the constraints is a program expressed in this CPU's assembly language; this input functions as software that controls the flow through the unrolled CPU execution. In TinyRAM, the representation of RAM operations (load and store) uses a clever technique based on permutation networks (§2.3).

A principal advantage of TinyRAM is that the programmer can use standard C (to produce the assembly program); this is the best programmability in the verifiable computation literature. Furthermore, TinyRAM allows the verifier's setup work to be reused across different computations. The principal disadvantage is cost. For a computation that takes $t$ program steps, the constraints include $t$ copies of the CPU's fetch-decode-execute loop; that is, every program step incurs the cost (in number of constraints) of the CPU's fetch-decode-execute logic. On top of that, each of those $t$ steps brings additional constraints to verify RAM operations.

The other front-end approach is to require the programmer to write in a subset of C that is carefully restricted to allow a line-by-line translation from the program to constraints; for each line of code, the resulting constraints contain designated logic to verify that line. The state of the art here is embodied in Pantry [27], which builds on, and includes the functionality of, its predecessors: Pinocchio [61] and Zaatar [70].[2] Often, the representation that arises from this line-by-line translation is very concise; for

[2]There is recent work at the forefront of performance that handles set operations efficiently [52], using the same line-by-line compilation approach. There is also a cousin of this approach represented by a different line of work [33, 74, 75, 77]. But these

example, adding two variables costs only one constraint. An important exception is RAM operations: each load or store translates into multiple invocations of a cryptographic hash function, each of which is transformed into constraints. Although this technique is costly, the overhead is far less (for all but the smallest memories) than in earlier works in this line [61, 70, 72].

Pantry's advantages are roughly the inverse of TinyRAM's. Depending on the computation, Pantry can handle executions of comparatively long lengths. Also, it pays for RAM operations only when they are used. On the other hand, the price of those RAM operations, in number of constraints, is very high—far higher than TinyRAM's per-operation cost [27, §8.1][18, §5]. Furthermore, the subset of C that is exposed to the programmer lacks key constructs, most notably data dependent control flow.

This analogy is inexact, but if a Pantry constraint representation is like an ASIC, then TinyRAM is like a CPU that is controlled with software. Unfortunately, in the context of verifiable computation, both the cost of TinyRAM's generality and the restriction on Pantry's programmability present severe obstacles to practicality. This state of affairs raises a natural question, which is the subject of this paper: *Can we achieve excellent programmability (that is, present the programmer with a language that is very close to standard C) together with an efficient translation into constraints?* Toward that end, this paper makes the following contributions:

1. We design and build a new system, called *Buffet*, that answers the above question in the affirmative. Buffet incorporates the following technical innovations:
   - Buffet composes TinyRAM's RAM abstraction with the line-by-line compilation approach of Pantry, resulting in a Pantry-TinyRAM hybrid approach to RAM (§3).
   - Buffet achieves nearly the expressiveness of TinyRAM without generalized CPU fetch-execute-decode (§4), by adapting a loop flattening technique from the compilers literature. Buffet supports all of C except `goto` and function pointers.
2. We develop a conceptual framework for understanding Pantry and TinyRAM as points on the same design spectrum, thereby providing a unified description of the state of the art verifiable computation approaches (§3.3, §4.3). The resulting perspective directly enabled the design of Buffet.
3. We carry out a rigorous three-way performance comparison, based on implementations of Buffet, TinyRAM, and Pantry (§5). Besides experimentally evaluating Buffet, this study carefully compares Pantry and TinyRAM, which is the first detailed comparison of these approaches.

The result is the best of both worlds: Buffet has the best performance in the literature (orders of magnitude better than TinyRAM and Pantry) and supports almost all of standard C.

There are some disadvantages to Buffet, compared to TinyRAM. As a consequence of the model, Buffet has worse amortization behavior than TinyRAM, in terms of what computations the setup cost can be reused over. Moreover, Buffet does not provide a machine abstraction, which could hinder higher-level programmability. However, as discussed in Section 7, we believe that both issues are more pronounced in principle than they will be in practice. Also, Buffet's back-end implementation is not as efficient as the highly optimized implementation of [18]. However, that back-end can be plugged directly into Buffet's front-end; this is work in progress.

The most significant limitation of Buffet is that it is subject to the problems that are endemic to this research area: in every system released so far, the prover overhead and setup costs are still too high to be considered truly practical. Nevertheless, we regard Buffet as substantial progress: we believe that it is close to optimal, at least until the next breakthrough on the back-end occurs.

works are targeted to particular classes of computations so fall outside of our focus. (See Section 6.)

## 2 Background

This section presents the general framework in which Pantry [27] and TinyRAM [15, 18] operate, and then gives details on each of them. Parts of this description are influenced by prior work [18, 61, 70, 77, 79]; most notably, there are textual debts to Pantry [27]. Our description is tailored to the problem of verifying outsourced deterministic computations [39, 44]. However, Buffet itself and many of the prior systems (including TinyRAM, Pantry, and Pinocchio [61]) handle a more general problem—a *zero-knowledge proof of knowledge* [20, 40]—in which the prover can supply inputs to the computation and keep them private (for example, a private database for which the verifier knows a digest [18, 27, 61]).

### 2.1 Overview and framework

Existing systems (TinyRAM, Pantry, etc.) enable the following. A client, or verifier $\mathcal{V}$, sends a program $\Psi$, expressed in a high-level language, to a server, or prover $\mathcal{P}$. $\mathcal{V}$ sends input $x$ to $\mathcal{P}$ and receives output $y$, which is supposed to be $\Psi(x)$. $\mathcal{V}$ also receives a short certificate that it can efficiently and probabilistically *check* to determine whether $y$ is in fact $\Psi(x)$.

There are no assumptions about whether and how $\mathcal{P}$ malfunctions, though there is an assumed computational bound on $\mathcal{P}$. The guarantees are probabilistic, over $\mathcal{V}$'s random choices. A *VC Completeness* property states that if $y = \Psi(x)$, then a correct $\mathcal{P}$ makes $\mathcal{V}$ accept $y$ with probability 1. A *VC Soundness* property says that if $y \neq \Psi(x)$, then $\mathcal{V}$'s checks pass with less than $\epsilon$ probability, where $\epsilon$ is very small.

The existing systems work in three steps:

1. **Compile, produce constraints.** $\mathcal{V}$ and $\mathcal{P}$ compile the program into a system of equations over a set of variables, including $x$ and $y$. The equations have a solution if and only if $y = \Psi(x)$.
2. **Solve.** $\mathcal{P}$ identifies a solution.
3. **Argue.** $\mathcal{P}$ convinces $\mathcal{V}$ that it has indeed identified a solution, which establishes for $\mathcal{V}$ that $y = \Psi(x)$.

This paper's focus is the front-end (steps 1 and 2); the Pantry and TinyRAM instantiations of this component are described in Sections 2.2 and 2.3, respectively.

Given our focus on the front-end, this paper fixes a common back-end (step 3) for all systems under investigation. We are able to standardize this way because Buffet, Pantry, and TinyRAM (and many prior systems for verifiable computation) are modular: their front-ends can work with each other's back-ends. The common back-end will be Pinocchio's instantiation of GGPR [40, 61],[3] which we summarize below; details and formal definitions appear elsewhere [18, 22, 40, 61, 70].

For our purposes, GGPR is a *zero-knowledge SNARK (Succinct Non-interactive Argument of Knowledge) with preprocessing* [20, 40], which is to say that it is a protocol with the following structure and properties. There are two parties, a verifier and prover; the input to the protocol is a set of equations (or *constraints*)[4] $\mathcal{C}$, to which the prover purportedly holds a solution (or *satisfying assignment*), $z$. In the verifiable computation context, the constraints and solution are generated by steps 1 and 2 above. In a separate setup phase, the verifier, or some entity that the verifier trusts, follows a randomized algorithm to generate, and encode, a query. Online, for each new $(x, y)$ pair, the prover responds to the encoded query with a certificate; the verifier checks the certificate, and accepts or rejects it. GGPR has the following properties:

- *Completeness*: If there is a satisfying assignment to $\mathcal{C}$, a correct prover causes the verifier's checks to accept. This property contributes to VC Completeness.
- *Proof of knowledge*: If the prover does not have access to a satisfying assignment $z$, then—except with

[3] An alternative would be Zaatar's back-end [70], which we have tested and run with our Pantry, TinyRAM, and Buffet front-end implementations.

[4] Throughout this paper, we will refer to the back-end as working with "constraints". One can also view the back-end as working with "arithmetic circuits with non-deterministic inputs" [18, 61]. This is the same formalism; only the names are different.

very small probability—the prover's purported certificate causes the verifier to reject. This property contributes to VC Soundness [27, Apdx. A].

- *Zero-knowledge*: The protocol provides no information to the verifier—beyond what the verifier can deduce itself—about the values in $z$. As a result, the protocol keeps private any input supplied by the prover, *provided that input cannot be easily guessed*. (As with prior work [15, 18, 61], our evaluated examples (§5) do not have separate prover input. However, Buffet supports the property, and example applications of it are evaluated elsewhere [27, 31, 37].)
- *Efficiency*: We detail costs in Section 2.4. For now, we note that the verifier's check is fast and the prover's response is short. The principal costs are in the verifier's setup work and in the prover's work to generate the certificate.

## 2.2 Pantry

**Step 1: Compile, produce constraints.** The programmer expresses a computation $\Psi$ in a subset of C. This subset [27, 61, 70] contains loops (with static bounds), functions, structs, typedefs, preprocessor definitions, if-else statements, explicit type conversion, and standard integer and bitwise operations. In addition, Pantry includes a RAM abstraction.

Using a compiler [26, 55, 61, 70, 72], $\mathcal{V}$ and $\mathcal{P}$ transform $\Psi$ into a set of constraints $\mathcal{C}$ over $(X, Y, Z)$, where $X$ and $Y$ are vectors of variables that represent the inputs and outputs; we call the variables in $Z$ intermediate variables. Let $\mathcal{C}(X{=}x, Y{=}y)$ mean $\mathcal{C}$ with $X$ bound to $x$ ($\mathcal{V}$'s requested input) and $Y$ bound to $y$ (the purported output). Note that $\mathcal{C}(X{=}x, Y{=}y)$ is a set of constraints over $Z$; these are the constraints that step 3 (§2.1) works over.

For a given computation $\Psi$, the corresponding set of constraints $\mathcal{C}$ is constructed so that for any $x$ and $y$, we have: $y = \Psi(x)$ if and only if $\mathcal{C}(X{=}x, Y{=}y)$ is satisfiable (by some $Z{=}z$). A basic example [26, 27] is the computation add-1, whose corresponding constraints are $\mathcal{C} = \{Z - X = 0,\ Z + 1 - Y = 0\}$: for all pairs $(x, y)$, there is a $Z{=}z$ that satisfies $\mathcal{C}(X{=}x, Y{=}y)$ if and only if $y = x + 1$.

Some technical points: The domain of all variables is a large finite field, $\mathbb{F}_p$ (the integers mod a prime $p$); $p$ typically has at least 256 bits. Also, each constraint has degree 2 and is of a particular form, described elsewhere [61, 70]. Constraint variables are represented by upper-case letters ($X, Y, Z, \ldots$); concrete values taken by those variables are represented by lower-case letters ($x, y, z, \ldots$).

**Compilation process.** Given a program, the compiler unrolls loops (each iteration gets its own variables) and converts the code to static single assignment (details are described in [26]). The compiler then transforms each line into one or more constraints. Arithmetic operations compile concisely. For example, the line of code `z3=z1+z2;` compiles to $Z_3 = Z_1 + Z_2$. As in all of the works that use large finite fields to represent computations [15, 26, 61, 70, 72], inequality comparisons and bitwise operations cost $\approx w$ constraints, where $w$ is the bit width of the variables in question.

Conditional branches include constraints for both branches. As an example, Figure 1 illustrates a simple `if-else` statement and the corresponding constraints.

**RAM.** Pantry includes primitives for verifiable remote state, called GetBlock and PutBlock. Each of these primitives compiles into constraints that represent the operation of a collision-resistant hash function, $H(\cdot)$. One way to use GetBlock is for $\mathcal{V}$ to supply as part of the input to $\Psi$ a hash (or digest) $d$ of a remote input $b$ that $\Psi$ is supposed to work over (though $\mathcal{V}$ does not know $b$). Then, satisfying the constraints that represent GetBlock requires $\mathcal{P}$ to set the variables $B$ so that $H(B) = d$.[5]

Applying well-known techniques [23, 38, 53, 57], Pantry uses GetBlock and PutBlock to create a RAM abstraction. Specifically, each Load and Store call compiles into multiple GetBlock and PutBlock

[5]With GetBlock, $y = \Psi(x)$ is no longer logically equivalent to the satisfiability of $\mathcal{C}(X{=}x, Y{=}y)$. However, collision-resistance together with a suitable application of proof of knowledge (§2.1) implies VC Soundness, i.e., the verifier rejects wrong outputs with high probability [27, Apdx. A].

```
if (Z1 == 1) {
    Z2 = 10;
} else if (Z1 == 2) {
    Z2 = 20;
} else {
    Z2 = 100;
}
```

(a) Source.

```
// if Z1 == 1, Z2 = 10
M0 (0 = Z1 - 1)
M0 (0 = Z2 - 10)
(1 - M0) (0 = M2 ( Z1 - 1 ) - 1)
// else if Z1 != 1
(1 - M0) (
    // if Z1 == 2, Z2 = 20
    M1 (0 = Z1 - 2)
    M1 (0 = Z2 - 20)
    (1 - M1) (0 = M3 ( Z1 - 2 ) - 1)
    // else if Z1 != 2, Z2 = 100
    (1 - M1) (0 = Z2 - 100) )
```

(b) Constraints.

FIGURE 1—A conditional statement and corresponding constraints, under Pantry. Degree-3 constraints are unexpanded.

calls—and hence multiple invocations of $H(\cdot)$.

**Step 2: Solve.** To produce a satisfying assignment, $\mathcal{P}$ proceeds constraint-by-constraint. In cases when the solution is not immediate, a constraint has a compiler-produced *annotation* that tells $\mathcal{P}$ how to solve it. As an example, if $Z_1$ and $Z_2$ are already determined, then the solution to $Z_3 = Z_2 + Z_1$ is immediate. But in the constraints that correspond to the `if-else` statement of Figure 1, the annotations tell $\mathcal{P}$ how to set `M0`–`M3`. Similarly, to satisfy the constraints that represent GetBlock (and in response to a PutBlock), the annotations instruct $\mathcal{P}$ to interact with a backing store. We refer to such annotations and actions as being exogenous to the constraint formalism (the theoretical term is "non-deterministic input").

### 2.3 TinyRAM

As in Pantry, TinyRAM's constraints are over the finite field $\mathbb{F}_p$, and the constraints have input variables $X$, output variables $Y$, and intermediate variables $Z$.

**Step 1: Compile, produce constraints.** The programmer expresses a computation $\Psi$ in standard C, and then runs a compiler to transform $\Psi$ to an assembly program in a MIPS-like instruction set (for the TinyRAM CPU); we notate this program text $x_\Psi$. The programmer must statically bound $t$, the number of machine steps required to execute $x_\Psi$ on the TinyRAM CPU. The constraints themselves are produced by $\mathcal{V}$ and $\mathcal{P}$ in a separate, offline step that is parameterized by $t$. The constraints decompose into three subsets, described below.

**CPU execution.** The first set of constraints, $\mathcal{C}_{\text{cpu}}$, represents the TinyRAM CPU's execution, for $t$ steps, purportedly starting with memory that contains $x_\Psi$ and $x$ and producing output $y$ (this will be enforced below). The constraints have $t$ repeated blocks; each has variables for the CPU's state (registers, flag, program counter, and instruction to be executed) and represents one fetch-decode-execute cycle, the logic for which is shown in Figure 2.

Any assignment (satisfying or otherwise) to $\mathcal{C}_{\text{cpu}}$ corresponds to a purported *execution-ordered transcript* of the TinyRAM CPU: a list of its state at each step in the execution. In any *satisfying* assignment to $\mathcal{C}_{\text{cpu}}$, the variable settings correspond to the correct operation of the CPU, under the assumption that the results of `LOAD` operations are correct; that is, $\mathcal{C}_{\text{cpu}}$ leaves `LOAD` target variables unconstrained. These variables are restricted by the next two sets of constraints.

**Memory operations.** Define an *address-ordered transcript* as a sort of the execution-ordered transcript by memory address, with ties broken by execution order. Observe that in an address-ordered transcript, each `LOAD` is preceded either by its corresponding `STORE` or by another `LOAD` from the same address. Thus, one can establish the correctness of an address-ordered transcript by inspecting the consistency of sequential entries (in this paper, we use *consistency* to mean that loading from a memory cell returns the most recently stored value to that cell). Leveraging these observations, the second and third sets of

```
ProcessorState states[t]

state[0].pc = state[0].flag = 0
state[0].regs[0] = ... = state[0].regs[NUM_REGS-1] = 0

for S in [0, t-1):

   state[S].instruction = LOAD(state[S].pc)

   decode(state[S].instruction,
          &opcode, &target, &arg1, &arg2)
   next_flag = state[S].flag

   for i in [0, NUM_REGS):
      if (i != target):
         state[S+1].regs[i] = state[S].regs[i]

   switch (opcode):
      case OP_ADD:
         state[S+1].regs[target] = arg1 + arg2
         next_flag = (arg1 + arg2) > REGISTER_MAX
         break

      case OP_CJMP:
         if (state[S].flag)
            state[S+1].pc = arg1
         break

      case OP_LOAD:
         state[S+1].regs[target] = LOAD(arg1)
         break

      ...

   state[S+1].flag = next_flag

   if (opcode != OP_CJMP && opcode != OP_CNJMP
                         && opcode != OP_JMP):
      state[S+1].pc = state[S].pc + 1

state[t-1].instruction = LOAD(state[t-1].pc)

decode(state[t-1].instruction,
       &opcode, &target, &arg1, &arg2)
assert opcode == OP_ANSWER

return arg1  // expands to Y = arg1
```

FIGURE 2—Pseudocode for $\mathcal{C}_{\text{cpu}}$, the constraints that represent TinyRAM's CPU execution. In the constraints, the `for` loop is unrolled: the constraints contain $t$ repeated blocks, one for each iteration.

constraints include variables that represent an address-ordered transcript; these constraints are satisfiable if and only if the purported address-ordered transcript is a sort of the execution-ordered transcript that is pairwise consistent.

In more detail: the constraints are divided into groups that are satisfiable if and only if (a) the purported address-ordered transcript is at least a permutation (but not necessarily a sort) of the execution-ordered transcript; and (b) this permutation is indeed sorted and pairwise consistent.

The constraints in group (a), $\mathcal{C}_{\text{perm}}$, represent the logic of a permutation network [13, 19]. The inputs to this network are variables from the execution-ordered transcript, specifically two tuples (timestamp, op code, address, data) per machine cycle. One tuple represents the instruction fetch; the other, whatever the instruction requested (`LOAD`, `STORE`, or no RAM operation). $\mathcal{C}_{\text{perm}}$ also has variables that represent switch settings of the permutation network. By construction, $\mathcal{C}_{\text{perm}}$ is satisfiable if and only if its outputs are assigned to a permutation of its inputs. Note that although we have referred to "inputs" and "outputs," all variables are intermediate; the prover must obtain values (in its assignment $z$) for all of them.

The constraints in group (b), $\mathcal{C}_{\text{ck-sort}}$, work over the output variables in $\mathcal{C}_{\text{perm}}$. They are satisfiable if and only if the assigned values respect the pairwise relation establishing ordering and consistency.

**Putting the pieces together.** Where do the inputs and outputs $(x_\Psi, x, y)$ appear? A TinyRAM execution begins with a "boot" phase that stores $x_\Psi$ into the beginning of memory and $x$ into a well-known memory location that $\Psi$ expects. Concretely, the memory transcript that feeds into $\mathcal{C}_{\text{perm}}$ includes tuples for $x_\Psi$ and $x$; for example, $(j,$ `STORE`$, j, x_\Psi[j])$, $j \in \{0, \ldots, |x_\Psi|\}$, where $|x_\Psi|$ is the length of the program text. Notice that the relevant values are assigned by the *verifier* (that is, they are not part of the assignment $z$) and thus tether the execution to the verifier's request. For the output $y$, our description assumes that the output of $\Psi$ is a single machine word that is returned at the end of the execution.[6] Concretely, the final constraint in $\mathcal{C}_{\text{cpu}}$ is $y - Z^* = 0$, where $Z^*$ here is the constraint variable that represents the final setting of the register `arg1` (Fig. 2).

[6] A more general way to handle outputs is to supply $y$ as auxiliary input [14, 18], and to write $\Psi$ so that, after computing its output, it accepts iff that output equals $y$. This alternative is supported by our TinyRAM implementation and matches the original description of TinyRAM, which is geared to decision problems.

| | |
|---|---|
| certificate length | 288 bytes |
| $\mathcal{V}$ setup | $(\|Z\| + \|\mathcal{C}\|) \cdot 230\ \mu$s |
| $\mathcal{V}$ per-run | $6\text{ ms} + (\|x\| + \|y\|) \cdot 0.35\ \mu$s |
| $\mathcal{P}$ per-run | $(\|Z\| + \|\mathcal{C}\|) \cdot 240\ \mu\text{s} + (\|\mathcal{C}\| \log^2 \|\mathcal{C}\|) \cdot 0.6\ \mu$s |

$x, y$: input and output

FIGURE 3—End-to-end costs of TinyRAM [18] and Pantry [27] when applied to constraints $\mathcal{C}$, assuming the Pinocchio back-end [40, 61] (source is [27, Fig. 3, Apdx. E]). The cost of steps 1–2 are reflected in the number of variables ($|Z|$) and constraints ($|\mathcal{C}|$). The cost of step 3 is reflected in the overall structure and in the constants (assuming $|\mathcal{C}|$ is a power of 2). Buffet has the same cost model. TinyRAM's setup costs amortize better than Pantry's (see text).

To recap, any satisfying assignment to $\mathcal{C}_{\text{cpu}}$ corresponds to an execution-ordered transcript that (1) correctly represents non-RAM operations (ALU, control flow, etc.), and (2) ends with the purported output, $y$. For $\mathcal{C}_{\text{perm}}$ and $\mathcal{C}_{\text{ck-sort}}$ to be satisfiable, the values `LOAD`ed in the execution-ordered transcript must be correct and, in particular, consistent with program text $x_\Psi$ and program input $x$. Thus, the three sets of constraints as a whole are satisfiable if and only if $y$ is the correct output of the TinyRAM machine, given program $\Psi$ and input $x$.

**Step 2: Solve.** To produce the satisfying assignment $z$ to the constraint variables, the prover, given $x_\Psi$ and $x$, runs a routine on its native CPU that simulates the TinyRAM machine. This routine produces an execution-ordered transcript, yielding a satisfying assignment to the variables of $\mathcal{C}_{\text{cpu}}$.[7] This routine further selects the switch settings and determines the assignment to the memory-ordered transcript variables, in $\mathcal{C}_{\text{perm}}$.

## 2.4 Costs and amortization

The end-to-end costs of TinyRAM and Pantry (and Buffet) are summarized in Figure 3. There are several things to note here. First, the principal costs are $\mathcal{P}$'s work for each protocol run and $\mathcal{V}$'s setup work. (Strictly speaking, the setup work might be done by an entity other than $\mathcal{V}$, especially in the public verifier variant of GGPR [40]. However, that entity still has to be trusted by $\mathcal{V}$, so as a shorthand, we charge $\mathcal{V}$.) Second, these costs scale with the number of constraints ($|\mathcal{C}|$) and variables ($|Z|$); in fact, $|Z|$ generally scales linearly with $|\mathcal{C}|$ [27]. Thus, there will be an impetus, in the sections ahead, to translate program structures into economical constraint representations.

A final point is that the setup costs are incurred once for each new set of constraints. As a result, the two systems have different amortization regimes. Under Pantry, setup costs amortize over all instances (input-output pairs) of a given computation $\Psi$. In TinyRAM, by contrast, all computations of a given length use the same set of constraints, so the amortization behavior is potentially much better. (TinyRAM's constraints are sometimes said to be *universal*, but as we discuss in Section 7, in practice one constraint set would not be enough.)

To be relevant, Pantry and TinyRAM need to operate in one of two regimes. The first is when the protocol will be run multiple times on the same set of constraints (for Pantry, this means the same $\Psi$; for TinyRAM, it means different $\Psi$ that all have approximately the same value of $t$); the number of times must be high enough that the amortized overhead of the system drops below the naive approach to verification, namely running the computation at the verifier. The second regime is when the computation is not otherwise feasible (because the inputs are remote or private or both); in this case, we are less concerned with the amortized overhead of the system, but the setup costs must still be tolerable. Example computations and analysis are given in [27].

[7] Thus, $\mathcal{C}_{\text{cpu}}$ can be equivalently understood as validating the state transitions in a transcript that is non-deterministically supplied by $\mathcal{P}$; this view is the one presented in [15, 18].

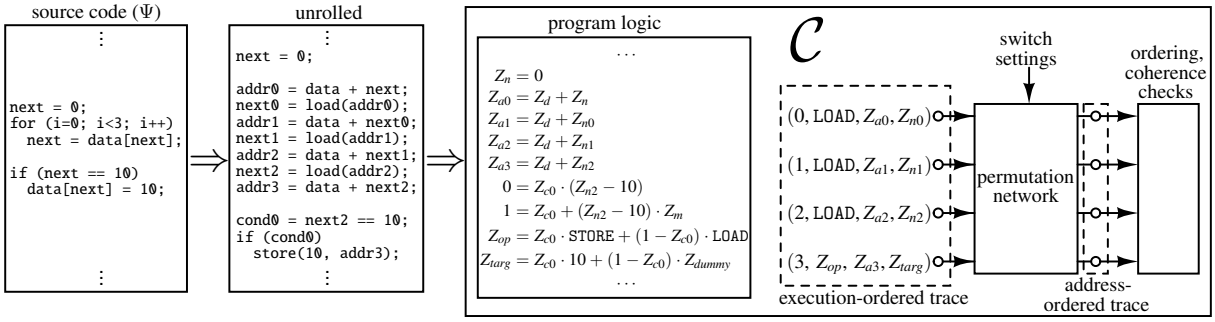


FIGURE 4—Buffet's instantiation of step 1 in the framework of §2.1. Buffet's compiler translates from an expansive subset of C to an intermediate representation, and then to three sets of constraints. The first set captures program logic, and results from applying Pantry's line-by-line compilation approach [27]. The other sets adapt the permutation network and memory consistency checks of TinyRAM [15, 18].

## 3 Representing RAM operations efficiently

As noted in the Introduction and elaborated in Section 2.4, end-to-end protocol costs are largely driven by $|\mathcal{C}|$, the size of the constraint representation. Under this metric, both TinyRAM and Pantry pay a steep price to expose a RAM abstraction to programmers. While Pantry pays only when memory is used, the cost per operation is exorbitant. TinyRAM, by contrast, pays much less to check memory but pays this cost on every operation—regardless of whether the operation interacts with memory. Concretely, a load or store in Pantry compiles to tens or hundreds of thousands of constraints [27, §8.1]; in TinyRAM, every program step compiles to one to two thousand constraints, of which several hundred are memory checking constraints (§3.1).

In light of the above, we ask, *Can Buffet pay for RAM only when the operations are used (as in Pantry), and furthermore can Buffet pay less per operation than either system?*

We find that the answer is yes: Buffet's approach is to graft TinyRAM's permutation networks into Pantry's constraints for non-RAM operations. Described that way, Buffet's approach might sound straightforward; instantiating it was not. The fundamental issue is that TinyRAM's techniques are tied to its execution model; indeed, the possibility of lifting these techniques from their context had been regarded as an "intriguing open question" [18, §5.4]. Nevertheless, with careful attention to detail, we have been able to make the hybrid approach work. The result is orders-of-magnitude savings versus TinyRAM and Pantry, for computations that interact with RAM.

Below, Section 3.1 describes Buffet's approach and the savings that it brings. Section 3.2 describes how Buffet exploits a class of compiler optimization opportunities. Section 3.3 discusses the fundamental source of Buffet's savings, versus TinyRAM and Pantry.

### 3.1 Hybridizing Pantry and TinyRAM

Under Buffet, the programmer of Ψ interacts with RAM using standard C pointers and arrays; the compiler transforms each pointer dereference or array interaction to `val=load(addr)` or `store(data,addr)` in the intermediate unrolled program.

Given Ψ, the Buffet compiler produces three sets of constraints, as depicted in Figure 4. The first set is the same as Pantry's traditional constraints, except that they do not restrict the return values of `load` (however, the *parameters* to a `load` or `store` are restricted by usual program logic, as expressed by constraints "upstream" of the given operation). As in TinyRAM, the second set of constraints represents a permutation network; the inputs to this permutation network are the return values and parameters of all of the `load` and `store` operations. And as in TinyRAM, the third set of constraints is satisfiable if and only if the permutation network's output is pairwise sorted and consistent—which ensures that a satisfying assignment to all of the constraints respects both program logic and memory correctness.

| system | per-op contribution to $\lvert\mathcal{C}\rvert$: one RAM op. | per-op contribution to $\lvert\mathcal{C}\rvert$: one non-RAM op. | total # of constraints ($\lvert\mathcal{C}\rvert$) |
|---|---|---|---|
| Pinocchio [61], Zaatar [70] | $3 \cdot 2^r$ | $c_{\text{avg}}$ | $(3 \cdot 2^r) \cdot k' \ + \ c_{\text{avg}} \cdot (t' - k')$ |
| Pantry [27] | $4700 \cdot r$ | $c_{\text{avg}}$ | $(4700 \cdot r) \cdot k' \ + \ c_{\text{avg}} \cdot (t' - k')$ |
| TinyRAM [18] | $2 \cdot c_{\text{mem}} + c_{\text{cpu}}$ | $2 \cdot c_{\text{mem}} + c_{\text{cpu}}$ | $(2 \cdot c_{\text{mem}} + c_{\text{cpu}}) \cdot t$ |
| Buffet | $c'_{\text{mem}}$ | $c_{\text{avg}}$ | $c'_{\text{mem}} \cdot k' \ + \ c_{\text{avg}} \cdot (t' - k')$ |

$t$: number of steps to execute $\Psi$ on TinyRAM CPU (§2.3)
$r$: log of memory size (for example, $r = 32$)
$c_{\text{cpu}} = 1114$: number of constraints for one TinyRAM CPU cycle
$c_{\text{mem}} = 67 + 4 \cdot \log 2t + 9r$: per-tuple constraint cost in TinyRAM
$t'$ ($\approx t$): number of program steps to execute $\Psi$ in P/Z/P
$k'$ ($\leq t'$): number of memory operations in $\Psi$ in P/Z/P
$c_{\text{avg}}$ ($\approx 22$): avg. non-RAM constraints per step for $\Psi$ in P/Z/P
$c'_{\text{mem}} = 21 + 10 \cdot \log k' + 2r$: per-tuple constraint cost in Buffet

FIGURE 5—Per-operation and total constraint costs, for a given computation $\Psi$. Buffet improves on the others both qualitatively and quantitatively: its RAM verification costs scale with the number of memory operations ($k'$) rather than the number of program steps ($t, t'$), and the scaling factor ($c'_{\text{mem}}$) is much lower than in TinyRAM ($2 \cdot c_{\text{mem}} + c_{\text{cpu}}$) and Pantry ($4700 \cdot r$). "P/Z/P" stands for "Pinocchio/Zaatar/Pantry execution model". In Zaatar and Pinocchio (Pantry's baselines), dynamically-addressed loads and stores translate to inefficient switch/case statements [18, 27]. The Pantry row reports the cost of a load; a store costs $2 \cdot 4700 \cdot r$. The TinyRAM model is from [18, §5.1] and is for 16 registers of 32 bits; different TinyRAM configurations result in small differences. The text explains $c_{\text{avg}}$ and the relationship between $t$ and $t'$.

**Refinements and savings.** The preceding picture is based on TinyRAM's technique for RAM (§2.3) but with some important refinements. The cost savings from these refinements are summarized in Figure 5.

First, Buffet sheds the $t$ repeated copies of the simulated CPU's fetch-decode-execute logic, saving $c_{\text{cpu}}$ constraints on every operation. Of course, Buffet must still pay to execute each operation; Figure 5 summarizes this cost using $c_{\text{avg}}$, which is a term that captures the average cost in constraints of a non-RAM operation in the Pantry model (shared by its base systems, Pinocchio [61] and Zaatar [70]). $c_{\text{avg}}$ is computation-dependent, but we can impose reasonable bounds, since the cost of non-RAM operations ranges from 1 (for arithmetic operations) to 34 (for operations on 32-bit values that require separating numbers into their bits, such as inequality comparisons and bitwise operations). We obtained $c_{\text{avg}} = 22$ in the figure by conservatively assuming that all non-RAM operations occur with uniform frequency.

Second, Buffet's permutation network works over what we call a *trace*: a set of tuples, one for each of the $k'$ operations that specifically interacts with memory. By contrast, recall that in TinyRAM (§2.3), the input to the permutation network is a *transcript*: two tuples for each of the $t$ execution steps. One of these tuples is for the instruction fetch; the other tuple corresponds to executing a load or store instruction. Of course, the fetched instruction might not be a load or store! Yet the constraints cannot "know" in advance what will happen on a given step, and thus each step includes logic for all possible instructions. These distinctions—trace versus transcript, one versus two tuples—are reflected in Figure 5, specifically the $2 \cdot c_{\text{mem}} \cdot t$ contribution in the TinyRAM row and the $c'_{\text{mem}} \cdot k'$ term in the Buffet row.

A critique of the preceding analysis is that $t$ and $t'$ are different kinds of quantities. For one thing, $t$ counts TinyRAM's assembly steps, whereas $t'$ counts operations in Buffet's intermediate unrolled representation. However, this objection is not fundamental, as the operations in the latter roughly correspond to those in the former; furthermore, there is a basis for comparison, in that both models ultimately "run" a program that was expressed in the same source (§5). A more serious issue is that $t$ counts only the program steps actually taken whereas $t'$ includes operations in branches not taken. However, even this distinction does not affect the analysis much, unless conditionals in $\Psi$ contain many possible paths. Even if we take $t' = 10 \cdot t$ (an extremely pessimistic choice, since this implies that conditional statements in the program entail 10 possible branches on average), Buffet's costs are nearly an order of magnitude below TinyRAM's.

**Details.** Below, we describe how Buffet handles loads and stores, in terms of steps 1 and 2 in the framework of §2.1. As a backdrop, we note that in step 1, the Buffet compiler maintains a monotonically increasing counter, *mem-ts*, that tracks memory operations. In step 2, the Buffet prover maintains a simulated RAM inside its own address space.

*Loads.* For step 1, when the compiler encounters `Zval=load(Zaddr)`, it creates constraints that "wire" the tuple $(\textit{mem-ts}, \texttt{LOAD}, Z_{\text{addr}}, Z_{\text{val}})$ into the permutation network. The compiler also inserts an annotation for step 2, which tells $\mathcal{P}$ to set $Z_{\text{val}}$ by loading address $Z_{\text{addr}}$ from its simulated RAM.

*Stores.* When executed unconditionally (meaning outside of an if-then or if-then-else block), stores are similar to loads. However, inside of a conditional block, a store operation creates complexity in both steps 1 and 2.

Concerning step 1, the problem is as follows. If a branch that contains a store is taken during execution, then the variables of that store operation must enter the permutation network. If, however, the branch is not taken, then the store operation must not be part of the execution-ordered trace. Meanwhile, Buffet's compiler does not know whether a given branch will be taken; furthermore, it must decide *statically* what enters the permutation network. Buffet resolves this issue by "dynamically casting" the store to a dummy load at run time, if the branch is not taken. Specifically, when the compiler encounters `store(Zdata, Zaddr)` inside a conditional block, it wires the following tuple into the permutation network:

$$\begin{aligned}&(\textit{mem-ts},\\ &Z_{\text{cond}} \cdot \texttt{STORE} + (1 - Z_{\text{cond}}) \cdot \texttt{LOAD},\\ &Z_{\text{addr}},\\ &Z_{\text{cond}} \cdot Z_{\text{data}} + (1 - Z_{\text{cond}}) \cdot Z_{\text{dummy}}),\end{aligned}$$

where $Z_{\text{cond}}$ captures the conditions that surround the store operation. If $Z_{\text{cond}}$=0 at run-time, then observe that $\mathcal{P}$ is obliged to treat this tuple (more precisely, the constraints to which this tuple expands) as a dummy load rather than a store.

Concerning step 2, recall that during Pantry's solving phase, there is no longer an explicit notion of control flow, conditionality, etc. Pantry's prover simply walks a list of constraints and annotations, solving each one, as instructed by annotations (§2.2). The difficulty in our present context is that, if a store operation is in an untaken branch, $\mathcal{P}$ should not actually apply the update to its simulated RAM—if $\mathcal{P}$ did so, future loads would return incorrect values, so the permutation network constraints would not be satisfied, causing $\mathcal{V}$ to reject. To address this issue, the Buffet compiler creates an annotation that instructs $\mathcal{P}$ to apply the store operation to its simulated RAM only if $\mathcal{P}$ also sets $Z_{\text{cond}}$=1.[8]

### 3.2 Optimizations

Consistent with Buffet's goal of paying for RAM operations only when necessary, its compiler eliminates loads and stores where possible; the result is fewer constraints and hence better performance. Of course, the compilers of TinyRAM and Pantry could apply similar analysis, but the overall effect on their performance would be muted, as we explain in Section 3.3.

Buffet applies two types of optimizations. First, when the address associated with a RAM operation (that is, the source of a load or the destination of a store) can be resolved at compile time, the compiler can

[8]The reason that Pantry does not face the issues just described is somewhat subtle. Recall from Section 2.2 that Pantry's RAM is implemented on a content-addressable block store, which maps digests $d$ to blocks $B$, where $H(B) = d$. When RAM is built this way [23, 38, 53], each configuration of memory has its own digest. Furthermore, each load and store takes a digest as an argument, and each store returns a new digest [27, §5.1]. Thus, if Pantry applied a store to its simulated RAM, but the store happened in an untaken branch, the result would be only to add harmless (digest, block) entries in the block store: they would not overwrite other entries (because digests are functionally unique), and they would be unreferenced by the downstream program logic (because the branch is not taken).

sometimes eliminate the RAM operation. To do so, the compiler maintains a table that maps addresses to intermediate variables. When the compiler encounters `store(Zdata,Zaddr)` where $Z_{\text{addr}}$ is statically determined, the compiler in effect delays the memory write. Specifically, it produces no corresponding constraints; it simply adds a new entry in the table mapping the value of $Z_{\text{addr}}$ to $Z_{\text{data}}$. When the compiler encounters `Zval=load(Zaddr)` where $Z_{\text{addr}}$ is statically determined, it consults the table. If there is a mapping between $Z_{\text{addr}}$ and an intermediate variable $Z_{\text{upstream}}$, the compiler produces a constraint that assigns $Z_{\text{val}} = Z_{\text{upstream}}$ (rather than wiring a new tuple into the permutation network).

When the compiler encounters a load or store whose address $A$ cannot be fully resolved at compile time, it must apply the delayed writes for any memory that could be referenced by $A$. Specifically, for each entry $(a_i, Z_i)$ in the delayed writes table, the compiler uses pointer aliasing analysis [60] to determine whether $A$ and $a_i$ could possibly reference the same memory. As an example, suppose that the compiler's analysis indicates that $A$ refers to an element of an array `data1`, that $a_i$ points into the array `data2`, and that `data1` and `data2` do not overlap; then the compiler knows that $A$ and $a_i$ cannot reference the same memory.[9] If the compiler *cannot* make a determination such as this, it produces constraints that store the value $Z_i$ to address $a_i$, and removes $(a_i, Z_i)$ from the table. After walking the table, the compiler produces constraints for the operation at address $A$.

The second type of optimization is classical load and store elimination [60]. If the `addr` parameters of two RAM operations are bound to the same intermediate variable, the compiler determines that the operations share the same address. In such cases, the Buffet compiler applies three reductions:

R1. For two load operations from the same address with no intervening store, the compiler replaces the return value of the second load with the return value of the first, and eliminates the second load.

R2. For two store operations to the same address with no intervening load, the second store obviates the first.

R3. For a store immediately followed by a load targeting the same address, the compiler eliminates the load and instead refers to the data just stored.

For an example that uses these optimizations, consider the following pseudocode:

```
out[offset] = 0
for i in [0, 10):
   out[offset] += input[i]
```

Though this code seems to access `input[i]` 10 times and `out[offset]` 21 times, Buffet reduces this to only a single store operation as follows:

1. The compiler unrolls the loop into a sequence of load, store, and arithmetic operations.
2. Since `i` is known for each iteration, the compiler can statically determine `input[i]`. Applying the first optimization, it replaces the corresponding load operations with references to intermediate variables by consulting its table of address-intermediate variable pairs.
3. The remaining RAM operations form an alternating sequence of stores and loads, all referring to `out[offset]`. Applying the second type of optimization (reduction R3), the compiler assigns the result of each load to the data of the preceding store, eliminating all of the load operations.
4. The remaining RAM operations are all stores to the same address. Applying reduction R2, the compiler eliminates all but the final store.

### 3.3 Discussion

What is the fundamental reason that Buffet can pay for memory only when it is used, whereas TinyRAM has to incur the cost on every operation (§3.1)? And why is load-store elimination of far more benefit to

[9] This analysis relies on the program's respecting the declared bounds of arrays. Accesses outside array bounds result in undefined behavior: the program may be incorrectly compiled to constraints. This limitation is consistent with the C standard [1, §J.2].

Buffet than TinyRAM (§3.2)?

These questions have the same answer: the different abstraction barriers in the two systems. In Buffet, the C compiler exploits knowledge of the program text to optimize the constraints that it produces: by wiring only RAM operations into the permutation network (§3.1) and by optimizing out unneeded constraints (§3.2).

In TinyRAM, the C compiler also has full knowledge of the program, but the compilation step produces TinyRAM assembly (the $\Psi \rightarrow x_\Psi$ step in Section 2.3). Meanwhile, this assembly program is an *input* to the constraints, and cannot affect them: in the TinyRAM model, constraints do not change on a per-computation basis [18]. Beyond this, recall that each step of the unrolled TinyRAM execution contains the logic needed to execute any possible assembly instruction (§2.3, Fig. 2); since any step might be a load or store, every step in the execution must be wired into the permutation network. Therefore, while the TinyRAM compiler could apply the optimizations in Section 3.2, the result would only be to reduce program text length $|x_\Psi|$, and potentially the maximum number of execution steps $t$. There is no sense in which the compiler could selectively eliminate expensive operations, as each program step induces the same cost.

Pantry, in contrast, *could* apply the optimizations of Section 3.2 to reduce the number of expensive operations. However, the ultimate efficacy would be limited by the extreme cost of the remaining RAM operations. In practice, Pantry is limited to at most several tens of RAM operations before its constraint set becomes intractable (§5.4); even the most aggressive optimization cannot overcome this limitation.

## 4 Efficient data dependent control flow

Using the work of the preceding section, Buffet produces concise constraints for straight line computations (because it inherits Pantry's line-by-line compilation), but the subset of C supported so far does not include a key programming construct: data dependent control flow. TinyRAM lets the programmer use all of C (due to the underlying abstraction of a general-purpose CPU); however, as discussed in the previous sections, TinyRAM's abstraction brings significant overhead.

The challenge is again for Buffet to provide the best of both worlds. Buffet's high-level solution is a source-to-source translation that adapts techniques from the compilers literature, and exploits aspects of the constraint idiom. Specifically, the Buffet compiler accepts programs written in a nearly complete subset of C (only `goto` and function pointers are excluded) and applies a *flattening* transformation to produce a C program that is less concise but has no data-dependent control structures; the compiler then translates the modified source efficiently into constraints. This approach works because in our context, there is no cost to making the intermediate source verbose—the constraint formalism unrolls computations anyway.

### 4.1 Motivating example

Consider the program of Figure 6a,which takes as input a run-length encoded string and returns as output the corresponding decoded string. (Consistent with the language supported by the Buffet compiler, all of our examples in this section refer to C code; however, for visual clarity, they are depicted in a Python-like pseudocode.) Pantry cannot compile this program (§2.2), since the number of iterations in the inner loop is determined at runtime by each `length` value.

The programmer might naively try to make the program suitable for Pantry by upper-bounding both loops separately: the inner one has a maximum of `LENGTH` iterations (because no character's `length` can exceed the output size), as does the outer one (if each `length` is 1). With these observations, and adding some `if` guards, the programmer could create a program that Pantry would compile—but at a quadratic cost, specifically $\texttt{LENGTH}^2$ unrolled iterations.

An alternative transformation is shown in Figure 6b. Observe that in this version, there is only one

```
i = j = 0
while j < LENGTH:
   char = input[i++]
   length = input[i++]
   do:
      output[j++] = char
      length--
   while length > 0
```

(a) Original.

```
i = j = length = 0
while j < LENGTH:
   if length > 0:
      output[j++] = char
      length--
   else:
      char = input[i++]
      length = input[i++]
      output[j++] = char
      length--
```

(b) Flattened.

FIGURE 6—Motivating example: a simple RLE decoder. `input` is a sequence of `char`, `length` pairs corresponding to `length` sequential `char`s in `output`; the output is known to be `LENGTH` bytes. The original pseudocode has a data dependent inner loop, while the flattened version does not.

```
while j < MAX1:
    <BODY 1>
    // data dependent bound
    limit = get_limit(j)
    for i in [0, limit):
        <BODY 2>
    <BODY 3>
```

(a) Original.

```
state = dummy = 0
while dummy < MAXITERS:
   if state == 0:
      if j < MAX1:
         <BODY 1>
         limit = get_limit(j)
         i = 0
         state = 1
      else:
         state = 3

   if state == 1:
      if i < limit:
         <BODY 2>
         i++
      else:
         state = 2

   if state == 2:
      <BODY 3>
      state = 0

   dummy++
```

(b) Flattened.

FIGURE 7—A more general loop flattening example. The original and flattened pseudocode have equivalent control flow.

loop, and its bounds are no longer data dependent. Since every path through the loop writes one character to output, the number of iterations is bounded by `LENGTH`. Intuitively, the `length` variable records the state of the outer loop: when `length` is nonzero, the program outputs a character; otherwise, it reads in more data.

## 4.2 Automatic loop flattening

Having manually transformed the example of Figure 6, we now consider how this approach can be automated. As we note above, the flattened code tracks whether the program is currently executing the outer or inner loop. Further, note that the former state always transitions to the latter, while the latter state self transitions until some condition is met.

More generally, every loop nest can be represented as a sequence of transitions among a set of states. As an example, consider the code of Figure 7a, and the equivalent flattened loop of Figure 7b. Observe that when `state` transitions from 0 to 1 in the flattened code, this corresponds in the original code to onset of the inner loop; similarly, the transition from 1 to 2 corresponds to the inner loop's exit. Thus, the

```
while j < MAX1:
   <BODY 1>
   // data dependent bound
   limit = get_limit(j)
   for i in [0, limit):
      // data dependent break
      if condition(i, j):
         break
      <BODY 2>
   <BODY 3>
```

(a) Original.

```
state = dummy = 0
while dummy < MAXITERS:
   if state == 0:
      if j < MAX1:
         <BODY 1>
         limit = get_limit(j)
         i = 0
         state = 1
      else:
         state = 3

   if state == 1:
      if i < limit:
         if condition(i, j):
            state = 2
         else:
            <BODY 2>
            i++
      else:
         state = 2

   if state == 2:
      <BODY 3>
      state = 0

   dummy++
```

(b) Flattened.

FIGURE 8—Flattening a loop containing `break` statements. The flattened pseudocode emulates the control flow of `break`.

variable `state` tracks whether the inner or the outer loop would have been executing; on each iteration through the flattened loop, the `if` guards ensure that only the corresponding code is executed. If `j` reaches the original outer loop's bound before `dummy` reaches `MAXITERS` (for example, because of data dependent logic in `<BODY 2>`), `state` transitions to a value that causes implicit self-transitions for the remainder of execution, corresponding to termination of the outer loop.

Finally, note that the `MAXITERS` bound on `dummy` cannot be automatically determined for general data-dependent code; in Buffet, the compiler requires the programmer to specify this bound. While this might seem restrictive, TinyRAM has a corresponding requirement (the parameter $t$ of Section 2.3). We discuss this similarity further in Section 4.3.

Transformations for `while` and `do` are similar to the example just given. All of these are inspired by similar, but not identical, transformations in the context of parallelizing compilers [42, 48, 50, 64, 80] (see Section 6).

**`break` and `continue`.** Using the flattening transformation, we extend Buffet to compile `break` and `continue`, as illustrated in Figure 8. The flattened code of Figure 8b achieves the desired control flow as follows: (1) the `break` statement is replaced by an assignment updating `state`; and (2) `<BODY 2>` is `if` guarded such that it is not executed after a break.

The `continue` case is similar: a `continue` before `<BODY 2>` should transfer control to the beginning of state 1 after incrementing the inner loop counter. This is achieved by (1) replacing the `continue` statement with an assignment updating `i`; and (2) `if` guarding `<BODY 2>`.

**Generalizing the transformation.** The flattening transformation for a single nested loop generalizes directly to deeper nesting and sequential inner loops. In fact, the Buffet compiler flattens arbitrary loop nests, with `break` and `continue`.

The key observation is that each loop comprises one or more states, with state transitions determined by the loop conditionals. When the compiler reaches a loop to be flattened, it constructs a control flow

graph in which the vertices correspond to segments of code inside which control flow is unconditional. The edges of this graph correspond to control flow decisions connecting these code segments; the compiler determines these decisions by analyzing the loop body and conditionals. For example, when the compiler encounters a `break` statement, it (1) splits the enclosing vertex into two vertices, corresponding to code segments before and after the `break` statement, and (2) adds two new edges to the graph, one that connects from the pre-`break` vertex to the post-`break` vertex (no `break` executed), and the other connecting the pre-`break` vertex to the vertex containing the next statement after loop execution ends (`break` executed).

After the compiler has assembled this control flow graph, it emits corresponding C code. This code comprises a statically bounded `while` loop containing a sequence of states and transitions as in the examples above. The states are code sequences corresponding to the vertices of the control flow graph, with `if` guards that test the value of a `state` variable. Transitions, which correspond to the graph edges, are expressed as assignments that update the `state` variable.

**The programmer's interface.** Buffet supports all C control flow constructs except for `goto` and function pointers. The programmer annotates any looping construct that should be flattened by applying a C++11-style attribute, `buffet::fsm`. This attribute takes one argument, a bound on the number of iterations in the flattened loop (corresponding to `MAXITERS` in Figures 7b and 8b). The programmer must therefore determine the longest run time of each loop to be flattened, similar to how the TinyRAM programmer must choose $t$ (§2.3).

### 4.3 Discussion

With regard to control flow, the three systems (Pantry, TinyRAM, and Buffet) can be seen as points on the same design spectrum, with different tradeoffs. We first cover their similarities and then their differences.

All three systems require static bounds on execution length. Pantry requires the programmer to impose static bounds on all loops, nested or otherwise (§2.2, Step 1; §4.1); TinyRAM requires the programmer to set the parameter $t$ to statically bound the processor's fetch-decode-execute loop (§2.3); and and for each flattened loop in Buffet, the programmer must provide a static bound (§4.2).

In addition, the three systems handle conditionality in similar ways. For each `if` statement in the original computation, Pantry includes constraints to represent both branches (§2.2, Fig. 1). In TinyRAM, each processor step corresponds to a dedicated set of constraints that implement a large switch statement (§2.3, Fig. 2), with separate constraints for every instruction type. Buffet has aspects of TinyRAM and Pantry: each iteration of a flattened loop compiles to a dedicated set of constraints implementing a switch statement, with separate constraints for every case within the switch. Note that Buffet and TinyRAM support data-dependent control flow using essentially the same mechanism: each iteration of a flattened loop (in the case of Buffet) or of a fetch-decode-execute step (in the case of TinyRAM) is *a state machine transition*, where the choice of the next state is dynamically determined.

The source of these correspondences is the common execution substrate, namely the underlying constraint formalism. Indeed, the reason that each step of a computation in any of the systems requires a dedicated set of constraints (including separate constraints for each logical alternative in that step) is that constraints project time and conditionality onto space. That is, constraints are equivalent to (Boolean or arithmetic) acyclic circuits, where the flow of the computation through the circuit is analogous to the passage of time.[10]

One distinction between Buffet and TinyRAM is that the former transforms sections of the program into a state machine, whereas the latter transforms the CPU on which the program runs. Buffet's approach is consistent with the goal of paying only for what is needed, first, because Buffet's compiler tailors the transition function to the loop, and second, because the Buffet compiler applies the finite state

[10]The structure of the TinyRAM constraints brings this view into sharp relief: each cycle of the TinyRAM CPU—in a sense, its fundamental unit of time—corresponds directly to a set of constraints in $\mathcal{C}_{\text{cpu}}$ (§2.3).

machine transformation only as directed by the programmer. Buffet thus pays lower overhead than TinyRAM in almost all cases.[11] Furthermore, Buffet pays that overhead only when necessary: straight line subcomputations remain as efficient as in Pantry.

Another apparent distinction between the systems concerns programmability. TinyRAM elegantly supports all of C (as noted throughout). In fact, it also, at least in principle, supports *any* high-level programming language that can be compiled to MIPS-like assembly instructions—which is most languages. Indeed, this programmability was the motivation for TinyRAM's virtual CPU abstraction [14, §1.1]. Buffet, by contrast, does not expose a machine interface; hence, it has no concept of a software-controlled program counter, and thus does not easily support language features that involve choosing arbitrary control flow at run time. In the context of C, this means that Buffet does not support function pointers. (Buffet also lacks `goto` support, as noted earlier, but this lacuna is not fundamental, and we are addressing it in ongoing work.) However, we conjecture that further application of static analysis techniques will broaden the range of high-level constructs that can be efficiently implemented in Buffet's execution model. We discuss this point further in Section 7.

## 5 Implementation and empirical evaluation

In this section, we experimentally evaluate Buffet. We compare it to its predecessors, TinyRAM and Pantry, and in the process we also compare the latter two. Specifically, we answer the following questions:

1. How does the performance of Pantry and TinyRAM compare on programs with (a) straight line computations and (b) random memory access?
2. What improvement does Buffet's memory abstraction (§3) offer over previous systems?
3. What improvement does Buffet's finite state machine (FSM) transformation (§4) offer over previous systems?

We base this evaluation on implementations of Buffet, Pantry, and TinyRAM, running on several benchmarks.

Our principal focus is on the various front-ends. As noted earlier (§2.4, Fig. 3), the costs imposed by the front-end appear in *the number of constraints that the back-end works over*. To provide context, we will also report end-to-end costs, although these depend upon both front-end and back-end performance.

The back-end that we standardize on in this paper is the Pinocchio implementation [61] of the GGPR encoding [40], as discussed earlier (§2.1). Recently, a highly optimized implementation of this protocol was released [4]. Future work is to integrate this back-end into our implementation; this will somewhat improve absolute performance (end-to-end costs), by factors of roughly 5, but it will not change the ratios among the systems, which is the focus of our comparison.

The summary of the comparison is as follows. For straight-line computations, Buffet matches Pantry's performance; both outperform TinyRAM by 2–4 orders of magnitude. For RAM operations, Buffet improves on TinyRAM's performance by 1–2 orders of magnitude, and on Pantry's by 2–3 orders of magnitude. For data dependent looping, Buffet again exceeds TinyRAM's performance by 1–2 orders of magnitude; this improvement is driven by Buffet's FSM transformation.

### 5.1 Implementation

Our Buffet implementation is built on the Pantry codebase [2]. We extended the compiler to provide support for RAM operations using C syntax. (Pantry's compiler required arrays and pointers to be statically

[11] In principle, when compiling flattened loops with extremely deep nesting and complex conditionals, Buffet's tailored transition function could incur greater overhead than TinyRAM's CPU. However, we believe that this is not a problem in practice, as such behavior does not seem to occur other than in degenerate cases. Further, we observe that all programs can be compiled with overhead at most equal to TinyRAM's through a hybrid system: the compiler might automatically determine which approach is less costly and produce constraints accordingly [77].

determined, and RAM operations required explicit annotation [27, §3].)

The Pantry and Buffet compilers operate in two stages. The first stage transforms programs into an intermediate set of constraints and *pseudoconstraints*, which abstract operations that require multiple constraints (for example, inequalities). In the second stage, the compiler expands pseudoconstraints and adds annotations (§2.2).

Buffet enhances the first stage by adding new pseudoconstraints corresponding to RAM operations (§3.1). It also adds first stage support for pointer aliasing and dataflow analysis, which it uses to generate and optimize the RAM pseudoconstraints (§3.2). In the second stage, Buffet adds new annotations for RAM operations.

To support the FSM transformation (§4), Buffet uses a separate C source-to-source compiler based on Clang [5]. This compiler acts as a preprocessor on the program text; its output is the input to Buffet's enhanced version of the Pantry compiler. The FSM compiler modifies Clang to add support for the `buffet::fsm`, attribute (§4.2).

The modifications to the Pantry compiler comprise 1700 lines of Java, 400 lines of Python, and 340 lines of C++. The FSM compiler comprises 1000 lines of C++.

### 5.2 Baselines and benchmarks

**Pantry.** Our Pantry evaluation uses the released codebase [2].

**TinyRAM.** No source code was available for TinyRAM (other than the recently released optimized back-end [4]), so we built an independent implementation. Ours differs from the original in several ways, described below. However, as discussed later, the two have comparable performance (§5.3).

First, our instruction set is slightly different from the published description [16], with the aim of optimizing the cost of $\mathcal{C}_{\text{cpu}}$ (§2.3) while retaining equivalent functionality. In brief, we borrow the *zero register* concept from the MIPS family [58], obviating several of TinyRAM's conditional instructions. We also shorten immediate operands such that instructions fit in one rather than two memory words, and update the immediate semantics of several operations (e.g., `SUB`) to compensate.

Second, we use a different method to generate the TinyRAM constraint set ($\mathcal{C}_{\text{cpu}}$, $\mathcal{C}_{\text{perm}}$, and $\mathcal{C}_{\text{ck-sort}}$). Whereas the original implementation uses a carefully hand optimized "circuit gadget" approach [15, §2.3.3], we implement the CPU logic, permutation network, and consistency checks (§2.3) in the Pantry subset of C, and compile this code with the Pantry compiler. To accommodate this, we made a small change to the Pantry compiler and prover implementations, adding a new primitive called `exo_compute`. This primitive instructs $\mathcal{P}$ to execute a program on a simulated TinyRAM CPU and to retain an execution ordered transcript, which $\mathcal{P}$ then uses (together with switch settings that it computes) as the satisfying assignment.

Third, to permute memory operation tuples (§2.3, §3.1), our implementation uses a Beneš network [19], whereas the most recent TinyRAM [18] uses a Waksman network [13]. The former requires a power-of-2 number of inputs; the latter does not.

Finally, our software analyzes a program's TinyRAM assembly code and automatically removes from $\mathcal{C}_{\text{cpu}}$ the logic that corresponds to unused instructions. For many programs, this results in substantial constraint savings.[12]

Our TinyRAM implementation comprises 280 lines of Pantry-C for the TinyRAM CPU and memory constraints, and 7200 lines of Java for a TinyRAM assembler, disassembler, and simulator.

The TinyRAM publications report on a compiler from standard C to TinyRAM programs ($\Psi \rightarrow x_\Psi$, §2.3). We did not implement this; rather, we programmed the benchmarks, described below, in TinyRAM assembly.

[12] This optimization can interfere with self-modifying code, since the program might generate instructions at run-time that do not appear in the program text itself.

| computation ($\Psi$) | size | type |
|---|---|---|
| Matrix multiplication | $m \times m$ | straight line |
| PAM clustering [76] | $m$ points, $d$ dimensions, $k$ medoids, $\ell$ iterations | straight line |
| Fannkuch benchmark [7] | $m$ elements, $\ell$ iterations | straight line |
| Pointer chasing | $m$ dereferences | RAM |
| Mergesort | $m$ elements | RAM |
| Boyer-Moore $\text{delta}_1$ table generation [24] | $m$ length pattern, $k$ length alphabet | RAM |
| RLE decoding | output length $m$ | data dependent |
| Knuth-Morris-Pratt string search [51] | $m$ length pattern, $k$ length string | data dependent |
| CSR Sparse matrix–vector multiplication [42] | $m \times m$ matrix, $k$ nonzero elements | data dependent |

FIGURE 9—Benchmark applications.

**Benchmarks.** Figure 9 lists our benchmark applications. We implemented each benchmark for native execution, as well as on Pantry, TinyRAM, and Buffet. Native benchmarks are programs in C and C++, compiled with the Intel C++ compiler version 14 with maximum optimizations (`-O3`) enabled.

The Pantry benchmark implementations are written in the Pantry subset of C. For memory benchmarks, the size of verifiable RAM is the minimum required for each computation, based on the input size.

The TinyRAM benchmarks are written for a TinyRAM CPU comprising 32 registers of 32 bits. Each benchmark program is written in heavily optimized, hand coded assembly. In producing the constraint set for each benchmark, we parameterize based on the exact values required for $t$, $|x|$, and $|x_\Psi|$ (§2.3); our hand optimizations to the benchmarks mean that $t$ and $|x_\Psi|$ are small, which is optimistic for TinyRAM.

The Buffet benchmark programs are written in the Buffet subset of C. For the straight line benchmarks, we use the same code as we do for Pantry. For the RAM and data dependent benchmarks, Buffet uses the code from the native implementations, except that in the data dependent benchmarks, we inserted the `buffet::fsm` attribute (§4.2, §5.1).

## 5.3 Setup

**Configuration.** As noted earlier, we standardize the back-end to be Pinocchio [61]. We run Pinocchio in designated verifier mode, as implemented in Pantry [2], and using the same cryptographic parameters [27, Apdx. D].

Our testbed is a cluster of machines, each of which runs Linux on a 16-core Intel Xeon E5-2680 with 32 GB RAM; the nodes are connected by a 56 Gb/s InfiniBand network.

**Measurement procedure.** For each system and benchmark, we execute the computation three times, averaging the result. The Pantry and Buffet compilers report $|\mathcal{C}|$, the number of constraints, and $|Z|$, the number of intermediate variables. The Pantry compiler also reports these figures when it compiles the TinyRAM constraint set. $\mathcal{V}$ and $\mathcal{P}$ each track time spent in computations via `getrusage`.

**Calibrating baselines.** Our TinyRAM implementation (§5.2) results in slightly larger values of $|\mathcal{C}|$ than are reported in [18, §5.1] for the same execution lengths. We have carefully analyzed this discrepancy. It results, first, from the fact that our implementation and the original apply different optimizations (§5.2). Second, we experiment with a TinyRAM CPU that has 32 registers of 32 bits each; by contrast, the relevant results in [18] are for a CPU with 16 such registers. We experiment with the "more powerful" CPU because it tends to reduce $t$ and hence TinyRAM's costs. However, the choice does increase $c_{\text{cpu}}$ by

| benchmark | system | size | $\lvert\mathcal{C}\rvert$ (millions) | $\mathcal{V}$ setup | $\mathcal{P}$ exec |
|---|---|---|---|---|---|
| Matrix mult. | TinyRAM | $m$=4 | 1.47 | 14.2 min | 20.5 min |
| | Pantry | $m$=128 | 2.1 | 9.6 min | 38.5 min |
| native: 760 $\mu s$ | Buffet | $m$=128 | 2.1 | 9.6 min | 38.5 min |
| PAM | TinyRAM | † | 2.94 | 28.0 min | 51.3 min |
| | Pantry | ‡ | 1.69 | 8.8 min | 21.2 min |
| native: 360 $\mu s$ | Buffet | ‡ | 1.69 | 8.8 min | 21.2 min |
| Fannkuch | TinyRAM | $m$=5, $l$=10 | 2.49 | 24.8 min | 49.2 min |
| | Pantry | $m$=13, $l$=100 | 1.18 | 11.5 min | 21.0 min |
| native: 11 $\mu s$ | Buffet | $m$=13, $l$=100 | 1.18 | 11.5 min | 21.0 min |
| Pointer chase | Pantry | $m$=16 | 0.99 | 9.1 min | 9.6 min |
| | TinyRAM | $m$=512 | 2.78 | 26.5 min | 46.5 min |
| native: 11 $\mu s$ | Buffet | $m$=4096 | 1.74 | 12.0 min | 21.5 min |
| Mergesort | Pantry | $m$=4 | 0.69 | 7.2 min | 10.1 min |
| | TinyRAM | $m$=16 | 2.4 | 23.3 min | 38.3 min |
| native: 4.8 $\mu s$ | Buffet | $m$=128 | 1.8 | 14.0 min | 21.7 min |
| Boyer-Moore | Pantry | $m$=8, $k$=16 | 1.26 | 13.4 min | 13.7 min |
| | TinyRAM | $m$=16, $k$=96 | 2.29 | 22.5 min | 40.0 min |
| native: 1.7 $\mu s$ | Buffet | $m$=128, $k$=4032 | 1.74 | 12.5 min | 21.2 min |
| K-M-P search | TinyRAM | $m$=4, $k$=48 | 2.66 | 25.5 min | 52.8 min |
| | BuffetStatic | $m$=16, $k$=192 | 1.79 | 13.3 min | 23.3 min |
| native: 1.4 $\mu s$ | Buffet | $m$=64, $k$=512 | 1.91 | 15.0 min | 24.7 min |
| RLE decode | BuffetStatic | $m$=64 | 2.03 | 15.2 min | 23.5 min |
| | TinyRAM | $m$=128 | 2.75 | 27.5 min | 49.8 min |
| native: 0.47 $\mu s$ | Buffet | $m$=1024 | 1.89 | 14.6 min | 23.2 min |
| Sparse mat–vec | TinyRAM | $m$=50, $k$=100 | 3.12 | 29.7 min | 58.8 min |
| | BuffetStatic | $m$=50, $k$=100 | 2.15 | 18.0 min | 31.3 min |
| native: 0.79 $\mu s$ | Buffet | $m$=250, $k$=500 | 1.84 | 13.8 min | 21.0 min |

†: $m$=4, $d$=2, $k$=2, $l$=2 ‡: $m$=20, $d$=128, $k$=2, $l$=5

FIGURE 10—Scaling limits of TinyRAM, Pantry, and Buffet: the problem sizes (in terms of input size and resulting number of constraints, $\lvert\mathcal{C}\rvert$) for each benchmark that each system is able to handle. $\mathcal{V}$'s setup time and $\mathcal{P}$'s execution time (depicted) largely depends on $\mathcal{C}$; $\mathcal{V}$'s verification time is not depicted because it is the same for all systems and independent of $\lvert\mathcal{C}\rvert$ (Figure 3), and no longer the principal protocol cost (§1, §2). Native execution times correspond to the largest input size. The first three benchmarks are straight line computations; the middle three are RAM benchmarks; the final three are data dependent control flow benchmarks. Computations are limited (by available testbed RAM) to a few million constraints. This corresponds to different computation sizes per system because of the different efficiency with which each system represents the execution of $\Psi$ in constraints.

15% in the worst case (note from Figure 5 that reducing $t$ and increasing $c_{\text{cpu}}$ are opposing effects). At the very worst, then, we are overstating TinyRAM's costs by 15%—but this difference is swamped by the multiple orders of magnitude that separate TinyRAM and Buffet.

Because our data dependent control flow benchmarks make heavy use of RAM, the difference in memory performance between Buffet and Pantry would obscure any change resulting from the work of Section 4. Thus, for the data dependent benchmarks (Fig. 9), we measure Buffet against a related system, *BuffetStatic*, that inherits Buffet's memory abstraction but, like Pantry, requires static loop bounds.

## 5.4 Results

### *Method of comparison*

We wish to do an apples-to-apples comparison of the three systems: an examination of their running times on the same computations, on the same input sizes. However, the maximum input size for which each

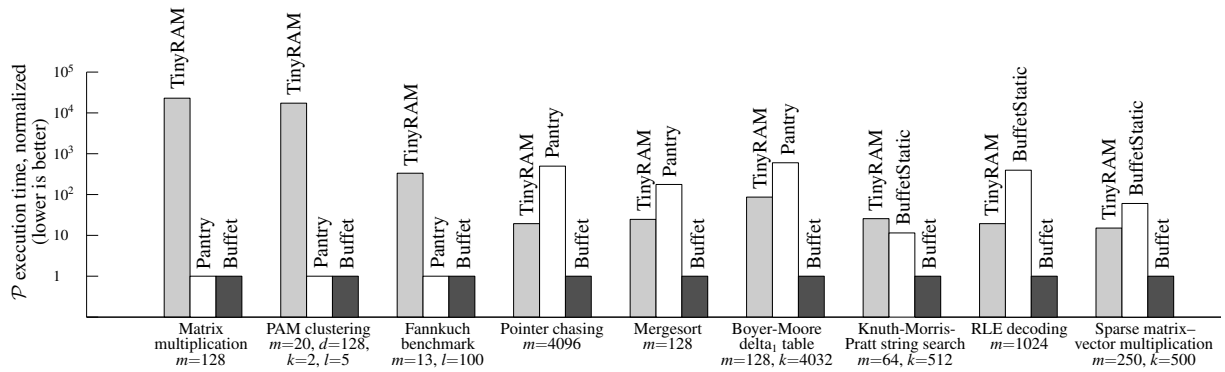


FIGURE 11—Comparative performance evaluation: $\mathcal{P}$'s execution time, normalized to Buffet. $\mathcal{V}$'s setup time is similar, and both track $|\mathcal{C}|$ (§2.4). For each benchmark, we extrapolate the performance of TinyRAM, Pantry, and/or BuffetStatic on the input size for which Buffet completed the computation (§5.4). All systems run on the same back-end (§2.1); thus, the ratio of end-to-end $\mathcal{P}$ cost is a measure of the relative front-end performance of each system.

system is able to execute a given benchmark differs. Thus, our method is as follows. First, we obtain measurements of each system by running it on the maximum input size that it can handle, in our testbed. These measurements both give us ground truth and indicate the qualitative performance of the systems. Second, we use these measurements to extrapolate the performance of the baseline systems to the input size at which Buffet executes the benchmark. Third, we perform a three-way comparison of the systems, using this extrapolated performance.

### *Ground truth and extrapolation*

Figure 10 details our measurements. The results demonstrate, first, that all computations are limited to a few million constraints in all of the systems (using our back-end and testbed). The limiting factor is testbed memory. Specifically, the "argue" step (§2.1, step 3) requires that the prover perform polynomial arithmetic based on the fast Fourier transform (FFT), and these operations require memory proportional to $|\mathcal{C}|$ [18, 32, 40, 61, 70].

Second, for each system, this constraint budget corresponds to very different computation sizes. The reason is that the systems vary widely in their *efficiency at representing computations in constraints*.

To extrapolate to larger input sizes, we do the following for TinyRAM: (1) compute the per-cycle cost, $|\mathcal{C}_{measured}|/t_{measured}$; (2) determine the number of cycles needed to execute the larger computation; and (3) account for the logarithmic increase in the per-cycle cost due to the growth of the permutation network. This yields the per-cycle constraint cost at the larger computation size and thus $|\mathcal{C}_{extrapolated}|$. We also check the measured and computed per-cycle constraint costs against the published TinyRAM figures [18, §5.1] to ensure that our model and implementation accurately represent TinyRAM's performance. We apply analogous procedures for the other baseline systems. Furthermore, we verify our extrapolation model for each baseline with a series of measurements at different computation sizes.

### *Relative performance*

Extrapolating the baselines to the input sizes at which Buffet runs the benchmarks, we now compare the relative end-to-end cost of the systems. We report $\mathcal{P}$'s execution time normalized to Buffet, as relative $\mathcal{P}$ execution time accurately captures the front-end efficiency of each system; the reason is that $\mathcal{V}$'s setup costs roughly track $\mathcal{P}$'s execution time (§2.4, Figure 3), and both end-to-end figures are driven by $|\mathcal{C}|$ (and $|Z|$, which is proportional to $|\mathcal{C}|$, as noted at the beginning of this section). Figure 11 summarizes the results.

**Pantry and TinyRAM.** In comparing Pantry and TinyRAM, we consider the straight line and RAM benchmarks of Figure 9. Because Pantry turns arithmetic operations into at most tens of constraints (§2.2), we expect its performance on straight line computations is excellent; conversely, we expect these computations to be inefficient under TinyRAM because every operation pays $c_{\text{cpu}}$ (§3.1, Fig. 5) to represent the logic of a CPU cycle (§2.3, Fig. 2). For computations involving random memory access, however, TinyRAM's efficient memory primitive should outperform Pantry's because of the latter's expensive hashing operations.

The predicted performance differences are evident in Figure 11: on straight line computations, Pantry outperforms TinyRAM by 2–4 orders of magnitude, while TinyRAM is consistently 1–2 orders of magnitude more efficient for random memory access.

**RAM performance in Buffet.** Using the same straight line and RAM benchmarks, we compare Buffet against TinyRAM and Pantry. As summarized in Figure 5, we expect Buffet to retain Pantry's performance on straight line benchmarks, and substantially outperform both systems for RAM operations.

Figure 11 confirms this hypothesis: Buffet's performance on RAM operations is 1–2 orders of magnitude better than TinyRAM's, which is itself 1–2 orders of magnitude better than Pantry's. As expected, Buffet and Pantry show identical performance on straight line programs.

**Data dependent control flow in Buffet.** The final set of benchmarks evaluates the performance of TinyRAM, BuffetStatic, and Buffet on data dependent computations. As before, we expect that Buffet's advantage on both arithmetic and RAM operations will result in better performance than TinyRAM. Similarly, we expect that while Buffet and BuffetStatic have identical *per-operation* performance, BuffetStatic will execute many more operations (§4.1).

In Figure 11, it is evident that Buffet exceeds the performance of the other systems by 1–3 orders of magnitude, consistent with expectations. Importantly, the large performance gap between BuffetStatic and Buffet on all benchmarks demonstrates the impact of Buffet's FSM transformation in enabling efficient data dependent control flow in the Pantry model.

### 5.5 Results in context

The foregoing discussion compares front-end performance; we now turn briefly to the more general question of applicability. There are two scenarios in which any of these systems can be considered applicable: (1) when $\mathcal{V}$ can save work by verifiable outsourcing, relative to executing locally; and (2) when the system is used for computations that $\mathcal{V}$ cannot perform itself.

An important concept when considering the first scenario is the *cross-over point* for $\mathcal{V}$: the number of instances of a computation (§2.4) that $\mathcal{V}$ must outsource before its total work (per-instance checking plus amortized setup) is lower than the naive approach to verification, namely executing these instances locally [27, §8.2; 70, §4]. $\mathcal{V}$'s cost to check each instance is at least 6 ms (§2.4, Fig. 3), so $\mathcal{V}$ can save work only when native computation takes longer than this. While none of the systems meets this requirement on our benchmark computations, similar computations can result in the systems breaking even in some cases. For example, $128\times128$ matrix multiplication on big integers (requiring a multi-precision native implementation) takes $\approx$100 ms [79]. Thus, Buffet and Pantry can save about 90 ms per instance by outsourcing the computation and checking its result; this savings exceeds the $\approx$10 minutes of setup time (Fig. 10) after about 7000 instances.

The second scenario involves computations that $\mathcal{V}$ cannot perform itself. For example, if it is prohibitive for $\mathcal{V}$ to handle the input (because the input is large and remote), then the naive approach to verification does not apply [27, §8.2]. Similarly, if the state is private to $\mathcal{P}$, then $\mathcal{V}$ cannot execute the computation itself, even in principle. In these cases, the systems are applicable if $\mathcal{V}$ desires verifiability and if the setup and proving costs are tolerable. Of course, what is tolerable depends on context.

## 6 Related work

As described in the introduction, there has been an explosion of work in the last few years on implemented systems for general-purpose verifiable computation based on the PCP theorem and sophisticated cryptography. The literature has grown to the point that we cannot do a complete summary here. However, there is a survey that covers the area [79], including works [44, 74, 75, 77] that are not specifically relevant here, given Buffet's focus on general-purpose computation with potentially private server inputs.

Buffet builds on Pantry [27] and TinyRAM [18]. The technical details of these systems were described in Section 2; here, we cover their significance and debts.

Pantry builds on two earlier systems: Zaatar [70] (an improvement on [71, 72]; this line of work refines the interactive argument protocol of IKO [46]) and Pinocchio [61] (an implementation of GGPR [40], which itself is described in Section 2.1). Pantry's central contribution is extending verifiable computation to allow the programmer to work with state, including remote state. Its core abstraction is a verifiable block store. This block store enables applications that are practical under certain usage regimes (MapReduce, remote databases, and private server state). Using this block store for RAM, however, is prohibitively expensive (as experiments reveal).

Our other foundation, TinyRAM [18], is the most recent in a line of work [14, 15, 22] that supports a general programming interface, including RAM and control flow constructs. The central contributions here are the decision to represent a general-purpose CPU in constraints, and the permutation network approach to verifying RAM. The latest TinyRAM system comprises a front-end, described in Section 2.3, and a highly optimized Pinocchio back-end, discussed in Section 5.

Several systems have applied and built on these foundations. For example, Trueset handles set operations efficiently, by refining GGPR and Pinocchio [52]. Like Buffet, Trueset uses the front-end strategy of line-by-line compilation, and it could be profitably integrated with Buffet. In a similar vein, Backes et al. [12] extend Pinocchio's machinery to efficiently support operations where the prover receives the input from a trusted third party and the verifier learns nothing about the data other than the result of the computation. As another example, ZØ [37] extends C#; ZØ uses Pinocchio or ZQL [36], with the selection controlled by sophisticated cost models, to compile code regions that invoke zero-knowledge features. As a final example, some works [31, 34] use Pinocchio as a back-end and, in lieu of a front-end, manually write down constraints that work in the execution model of Pantry, Zaatar, and Pinocchio; the goal is to extend the Bitcoin protocol to provide anonymity for users of the electronic currency.

Recent work [17] has combined TinyRAM with theoretical foundations for the back-end [21] that make the verifier's setup work (and the prover's memory requirement) independent of the computation length. Although this is an exciting development, so far the gains are mostly theoretical. For instance, the verifier must still do expensive setup work (proportional to the cost of representing the verifier's checking step in constraints), and the prover's costs are many orders of magnitude more than in Pantry, TinyRAM, and Buffet.

The finite state machine transformation techniques of Section 4 are based on novel adaptations of ideas from the programming language community, notably loop flattening. Several previous loop flattening approaches have been described in other contexts [42, 48, 50, 64]. Some of these techniques are intended only for regular loop nests, and none of them apply to code that uses loop control statements (`break` and `continue`). In contrast, Buffet supports both loop control statements and irregular loop nests (§4). Perhaps closest to Buffet is the work of [80] on Macah, a system that uses loop flattening in the context of a programming language and compiler for FPGAs. Like Buffet, Macah appears to be able to flatten loops that use `break` and `continue`; however, few details are given and so the precise relationship between the systems is unclear.

## 7 Summary, discussion, and future work

The experimental results (§5.4) demonstrate that Buffet achieves its goals. RAM operations are dramatically less expensive than in TinyRAM and Pantry, and they incur no overhead unless used. Data-dependent control flow is supported and is again substantially less expensive than in TinyRAM.

Nevertheless, Buffet has some limitations as compared to TinyRAM. First, Buffet's circuits are not universal, in the sense that TinyRAM uses the term. Specifically, whereas TinyRAM's constraints and setup cost are constant for all computations that satisfy a bound on execution time, Buffet requires setup for each new program and input length. However, this distinction is not likely to be pronounced in practice. Specifically, TinyRAM's constraints have three parameters $(t, |x_\Psi|, |x|)$ [18], and it is suggested that the user create constraints for exponentially increasing parameter values. Roughly speaking, this would mean $(\log_2 M)^3$ constraint sets, where $M$ is the maximum value of a parameter; for $M = 32000$ [18], TinyRAM would require setup work for thousands of constraint sets.

A second comparative limitation is expressiveness. In principle, TinyRAM supports any programming language that compiles down to machine instructions (§4.3). However, Buffet's disadvantage here is not clear cut: although Buffet's approach has implications for the programming *language* that it can support, it does not necessarily sacrifice the ability to support particular program *constructs*. In more detail, the absence of a machine abstraction means that Buffet cannot support function pointers in C (§4.3). More generally, Buffet cannot efficiently compile code that controls the abstract machine's program counter at runtime, such as objected-oriented constructs in C++ or Java, which involve indirection through a dispatch table. On the other hand, there are programming languages—those that make heavy use of static analysis—in which polymorphism and other language features do not require direct manipulation of the program counter. For example, Haskell [63] and Rust [6] implement polymorphism using a type inference system that works statically [78].[13] Another example is higher-order function calls: whole-program compilers like MLton [30] use *defunctionalization* [35, 65] to transform programs with higher-order function calls into a form with no indirect dispatching.

Based on this discussion, we conjecture that Buffet's compiler can map a rich set of higher-level programming language features to economical representations in Buffet's execution model. This conjecture implies that higher-level programmability does not require a machine abstraction, given a suitable choice of programming language. In fact, there is a broader research question here: if a computation is to be compiled to constraints (or non-deterministic circuits), what is the right combination of programming language and execution model? We leave these questions to the future.

For the present, there are two general points to take away from Buffet, corresponding to two different perspectives. One view is that Buffet has the same limitation of all systems in this research area (§6): overhead for the prover is simply too high to be useful for general applications. Thus, applicability is restricted to situations when the costs of the protocol are acceptable for one reason or another, as discussed briefly in Section 5.5 and explored at length elsewhere [27] (other examples include [31, 34, 37]). The other view is that verifiable computation as an area has tremendous potential and that—within the context of this area—Buffet strikes a sensible balance between cost and programmability.

### Acknowledgements

Josh Leners gave helpful comments on a previous draft. This work was supported by ONR grant N000141410469; NSF grants 1040083, 1048269, and 1055057; a Sloan Fellowship; and an Intel Early Career Faculty Award.

[13] Both Haskell and Rust have an exception to this: *existential types* [3] and *trait objects*, respectively, require dynamic method calls. In Haskell, however, these are a compiler-specific extension rather than a core language feature. In Rust, while trait objects are admittedly useful for generic programming, in practice they can be avoided at the cost of more verbose code.

Buffet's source code is available from http://github.com/pepper-project/.

## References


[1] *ISO/IEC 9899:2011 — C*. ISO, Dec. 2011.

[2] https://github.com/srinathtv/pantry, 2013.

[3] http://www.haskell.org/haskellwiki/Existential_type, 2013.

[4] https://github.com/scipr-lab/libsnark, 2014.

[5] http://clang.llvm.org, 2014.

[6] The Rust programming language. http://www.rust-lang.org, 2014.

[7] K. R. Anderson and D. Rettig. Performing lisp: Analysis of the Fannkuch benchmark. *ACM SIGPLAN Lisp Pointers*, VII(4), Oct. 1994.

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

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

[10] L. Babai. Trading group theory for randomness. In *STOC*, pages 421–429, May 1985.

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

[12] M. Backes, D. Fiore, and R. M. Reischuk. Nearly practical and privacy-preserving proofs on authenticated data. Cryptology ePrint Archive, Report 2014/617, Aug. 2014.

[13] B. Beauquier and E. Darrot. On arbitrary size Waksman networks and their vulnerability. *Parallel Processing Letters*, 12(3–4):287–296, 2002.

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

[15] 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*, Aug. 2013.

[16] E. Ben-Sasson, A. Chiesa, D. Genkin, E. Tromer, and M. Virza. TinyRAM architecture specification, v0.991. http://www.scipr-lab.org/system/files/TinyRAM-spec-0.991.pdf, 2013.

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

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

[19] V. Beneš. *Mathematical theory of connecting networks and telephone traffic*. Mathematics in Science and Engineering. Elsevier Science, 1965.

[20] 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*, Jan. 2012.

[21] N. Bitansky, R. Canetti, A. Chiesa, and E. Tromer. Recursive composition and bootstrapping for SNARKs and proof-carrying data. In *STOC*, June 2013.

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

[23] M. Blum, W. Evans, P. Gemmell, S. Kannan, and M. Naor. Checking the correctness of memories. In *FOCS*, Oct. 1991.

[24] R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772, Oct. 1977.

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

[26] B. Braun. Compiling computations to constraints for verified computation. UT Austin Honors thesis HR-12-10, Dec. 2012.

[27] B. Braun, A. J. Feldman, Z. Ren, S. Setty, A. J. Blumberg, and M. Walfish. Verifying computations with state. In *SOSP*, Nov. 2013. Extended version: http://eprint.iacr.org/2013/356.

[28] R. Canetti, B. Riva, and G. Rothblum. Practical delegation of computation using multiple servers. In *ACM CCS*, Oct. 2011.

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

[30] H. Cejtin, S. Jagannathan, and S. Weeks. Flow-directed closure conversion for typed languages. In *European Symposium on Programming*, 2000.

[31] A. Chiesa, C. Garman, E. Ben-Sasson, I. Miers, M. Green, M. Virza, and E. Tromer. Zerocash: Practical decentralized anonymous e-cash from Bitcoin. In *IEEE Symposium on Security and Privacy*, May 2014.

[32] J. W. Cooley and J. W. Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of Computation*, 19(90):297–301, 1965.

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

[34] G. Danezis, C. Fournet, M. Kohlweiss, and B. Parno. Pinocchio coin: Building zerocoin from a succinct pairing-based proof system. In *Workshop on Language Support for Privacy-enhancing Technologies*, Nov. 2013.
[35] O. Danvy and L. R. Nielsen. Defunctionalization at work. In *ACM International Conference on Principles and Practice of Declarative Programming*, 2001.
[36] C. Fournet, M. Kohlweiss, G. Danezis, and Z. Luo. ZQL: A compiler for privacy-preserving data processing. In *USENIX Security*, 2013.
[37] M. Fredrikson and B. Livshits. ZØ: An optimizing distributing zero-knowledge compiler. In *USENIX Security*, Aug. 2014.
[38] K. Fu, M. F. Kaashoek, and D. Mazières. Fast and secure distributed read-only file system. In *OSDI*, Oct. 2000.
[39] R. Gennaro, C. Gentry, and B. Parno. Non-interactive verifiable computing: Outsourcing computation to untrusted workers. In *CRYPTO*, Aug. 2010.
[40] R. Gennaro, C. Gentry, B. Parno, and M. Raykova. Quadratic span programs and succinct NIZKs without PCPs. In *EUROCRYPT*, 2013.
[41] C. Gentry and D. Wichs. Separating succinct non-interactive arguments from all falsifiable assumptions. In *STOC*, June 2011.
[42] A. M. Ghuloum and A. L. Fisher. Flattening and parallelizing irregular, recurrent loop nests. In *ACM PPoPP*, 1995.
[43] O. Goldreich. Probabilistic proof systems – a primer. *Foundations and Trends in Theoretical Computer Science*, 3(1):1–91, 2007.
[44] S. Goldwasser, Y. T. Kalai, and G. N. Rothblum. Delegating computation: Interactive proofs for muggles. In *STOC*, May 2008.
[45] S. Goldwasser, S. Micali, and C. Rackoff. The knowledge complexity of interactive proof systems. *SIAM J. on Comp.*, 18(1):186–208, 1989.
[46] Y. Ishai, E. Kushilevitz, and R. Ostrovsky. Efficient arguments without short PCPs. In *Conference on Computational Complexity (CCC)*, 2007.
[47] G. O. Karame, M. Strasser, and S. Čapkun. Secure remote execution of sequential computations. In *Intl. Conf. on Information and Communications Security*, 2009.
[48] A. Kejariwal, A. Nicolau, and C. D. Polychronopoulos. Enhanced loop coalescing: A compiler technique for transforming non-uniform iteration spaces. In *ISHPC05/ALPS06*, 2008.
[49] J. Kilian. A note on efficient zero-knowledge proofs and arguments (extended abstract). In *STOC*, May 1992.
[50] P. M. W. Knijnenburg. Flattening: VLIW code generation for imperfectly nested loops. In *CPC98*, 1998.
[51] D. E. Knuth, J. H. Morris, and V. R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, June 1977.
[52] A. E. Kosba, D. Papadopoulos, C. Papamanthou, M. F. Sayed, E. Shi, and N. Triandopoulos. TRUESET: Faster verifiable set computations. In *USENIX Security*, Aug. 2014.
[53] J. Li, M. N. Krohn, D. Mazières, and D. Shasha. Secure untrusted data repository (SUNDR). In *OSDI*, Dec. 2004.
[54] C. Lund, L. Fortnow, H. J. Karloff, and N. Nisan. Algebraic methods for interactive proof systems. *J. of the ACM*, 39(4):859–868, 1992.
[55] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella. Fairplay—a secure two-party computation system. In *USENIX Security*, pages 287–302, Aug. 2004.
[56] D. Malkhi and M. Reiter. Byzantine quorum systems. *Distributed Computing*, 11(4):203–213, Oct. 1998.
[57] R. C. Merkle. A digital signature based on a conventional encryption function. In *CRYPTO*, Aug. 1987.
[58] MIPS Technologies. *MIPS32™ Architecture for Programmers, Volume I*, Mar. 2001.
[59] F. Monrose, P. Wycko, and A. D. Rubin. Distributed execution with remote audit. In *NDSS*, Feb. 1999.
[60] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.
[61] B. Parno, C. Gentry, J. Howell, and M. Raykova. Pinocchio: Nearly practical verifiable computation. In *IEEE Symposium on Security and Privacy*, May 2013.
[62] B. Parno, J. M. McCune, and A. Perrig. *Bootstrapping Trust in Modern Computers*. Springer, 2011.
[63] S. Peyton Jones, L. Augustsson, D. Barton, B. Boutel, W. Burton, J. Fasel, K. Hammond, R. Hinze, P. Hudak, J. Hughes, T. Johnsson, M. Jones, J. Launchbury, E. Meijer, J. Peterson, A. Reid, C. Runciman, and P. Wadler. The Haskell 98 language report. `http://www.haskell.org/onlinereport/`, 2002.
[64] C. D. Polychronopoulos. Loop coalescing: A compiler transformation for parallel machines. In *ICPP*, Aug. 1987.
[65] J. C. Reynolds. Definitional interpreters for higher-order programming languages. In *ACM National Conference*, 1972.
[66] A.-R. Sadeghi, T. Schneider, and M. Winandy. Token-based cloud computing: secure outsourcing of data and arbitrary computations with lower latency. In *TRUST*, 2010.
[67] J. Schiffman, T. Moyer, H. Vijayakumar, T. Jaeger, and P. McDaniel. Seeding clouds with trust anchors. In *ACM Workshop on Cloud Computing Security*, Oct. 2010.
[68] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. van Doorn, and P. Khosla. Pioneer: Verifying integrity and guaranteeing execution of code on legacy platforms. In *SOSP*, Oct. 2005.
[69] S. Setty, A. J. Blumberg, and M. Walfish. Toward practical and unconditional verification of remote computations. In *HotOS*, 2011.

[70] S. Setty, B. Braun, V. Vu, A. J. Blumberg, B. Parno, and M. Walfish. Resolving the conflict between generality and plausibility in verified computation. In *EuroSys*, Apr. 2013.
[71] S. Setty, R. McPherson, A. J. Blumberg, and M. Walfish. Making argument systems for outsourced computation practical (sometimes). In *NDSS*, Feb. 2012.
[72] S. Setty, V. Vu, N. Panpalia, B. Braun, A. J. Blumberg, and M. Walfish. Taking proof-based verified computation a few steps closer to practicality. In *USENIX Security*, Aug. 2012.
[73] A. Shamir. IP = PSPACE. *J. of the ACM*, 39(4):869–877, Oct. 1992.
[74] J. Thaler. Time-optimal interactive proofs for circuit evaluation. In *CRYPTO*, Aug. 2013.
[75] J. Thaler, M. Roberts, M. Mitzenmacher, and H. Pfister. Verifiable computation with massively parallel interactive proofs. In *USENIX HotCloud Workshop*, June 2012.
[76] S. Theodoridis and K. Koutroumbas. *Pattern Recognition, Third Edition*. Academic Press, Inc., 2006.
[77] V. Vu, S. Setty, A. J. Blumberg, and M. Walfish. A hybrid architecture for interactive verifiable computation. In *IEEE Symposium on Security and Privacy*, May 2013.
[78] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *POPL*, 1989.
[79] M. Walfish and A. J. Blumberg. Verifying computations without reexecuting them: from theoretical possibility to near practicality. Technical Report TR13-165, ECCC, Nov. 2013. `http://eccc.hpi-web.de/report/2013/165/`.
[80] B. Ylvisaker, A. Carroll, S. Friedman, B. Van Essen, C. Ebeling, D. Grossman, and S. Huack. Macah: A "C-level" language for programming kernels on coprocessor accelerators. Technical report, University of Washington, Department of CSE, 2008.