# Verifying computations with state

Benjamin Braun⋆, Ariel J. Feldman†, Zuocheng Ren⋆, Srinath Setty⋆, Andrew J. Blumberg⋆, and Michael Walfish⋆
⋆The University of Texas at Austin †University of Pennsylvania

## Abstract

When outsourcing computations to the cloud or other third-parties, a key issue for clients is the ability to verify the results. Recent work in proof-based verifiable computation, building on deep results in complexity theory and cryptography, has made significant progress on this problem. However, all existing systems require computational models that do not incorporate state. This limits these systems to simplistic programming idioms and rules out computations where the client cannot materialize all of the input (e.g., very large MapReduce instances or database queries).

This paper describes Pantry, the first built system that incorporates state. Pantry composes the machinery of proof-based verifiable computation with ideas from untrusted storage: the client expresses its computation in terms of digests that attests to state, and verifiably outsources *that* computation. Besides the boon to expressiveness, the client can gain from outsourcing even when the computation is sublinear in the input size. We describe a verifiable MapReduce application and a queriable database, among other simple applications. Although the resulting applications result in server overhead that is higher than we would like, Pantry is the first system to provide verifiability for realistic applications in a realistic programming model.

## 1 Introduction

This paper addresses a fundamental problem in systems security: how can a local computer verify the correctness of a remotely executed computation? A motivating application is verifiable MapReduce; more generally, we are interested in verifiable cloud computing, including computations that involve verifiable database queries. Verification in these examples is important, to guard against corrupted data or incorrect execution; these faults can be caused by many things (bugs, misconfiguration, operator error, malice, etc.).

One possible approach is via replication [3, 10, 36], but this assumes that replica failure is uncorrelated or at least can be bounded. Attestation [41, 42] provides another way to build protocols for this problem, but it requires a chain of trust rooted in the hardware manufacturer, which may be inconsistent with the proposed applications above. Other possible solutions are tailored to particular applications [18, 49, 51].

In principle, an attractive approach is to have the performing computer return the results along with a *proof* that the results were computed correctly. In fact, recently there has been a flurry of work on verifiable computation that builds on powerful theoretical tools—probabilistically checkable proofs [4, 5], interactive proofs [22, 23, 33, 48], and cryptographic protocols [20, 26, 29, 30]—to realize a proof-based approach in built systems [12, 40, 43–46, 50]. In these setups, the client performs a probabilistic verification of the proof; if the computation was done correctly, the client accepts, otherwise, the client rejects with high probability.

The appeal of these protocols is that they make only cryptographic assumptions and apply to general-purpose computations that can be expressed as constraints (which are generalized circuits; see Section 2). Furthermore, the best of these systems are not so far from practical, provided we amortize the overhead of verification by batching many identical computations (with potentially different inputs) together.

On the one hand, the batched model is well-suited to typical cloud computing applications on enormous data sets. On the other hand, none of these systems allows a notion of state or storage: the client must have all of the input, the computation cannot use memory and the computation must be free of side effects.[1] In particular, these limitations rule out MapReduce and would require a computation that involved a remote database to materialize the entire database at the client.

This paper introduces a system called Pantry that provides verifiable computations with state. Specifically, Pantry provides a model of verifiable computation that includes a block store accessed via put/get. Using these primitives, we build a number of applications on top of Pantry, including verifiable MapReduce and a simple searchable database that supports a subset of SQL.

Pantry's top-level innovation is to marry the emerging approach to side-effect free verifiable computation based on complexity theory with the hash-based stores used in untrusted storage [14, 17, 32]. The key insight is that there exist suitable hash functions that are well-suited to being computed verifiably. We should note that the idea that something like this can be done appears to have been folklore in the theory community for a long time; the hard work of Pantry has been to work out the

[1] Recent work [6] addresses these issues in theory, but we are not aware of an implementation.

details and build a system. Specifically, Pantry makes the following concrete contributions:

1. Pantry enhances the computational model of state of the art systems for verifiable computation with a storage primitive (§3). For the programmer, Pantry exposes the interface of PutBlock/GetBlock, which operates over content hash blocks: blocks that are named by a cryptographic digest, or hash, of their contents. Such blocks are used extensively in work where clients leverage untrusted storage servers [14, 17, 32]; in Pantry, by contrast, the client does not receive the blocks themselves, though they are accessible within the computation.
2. Based on PutBlock and GetBlock, Pantry provides a framework for verifiable MapReduce (§4). Pantry's MapReduce traffics in digests rather than filenames: the input is digests of the respective mappers' inputs, and the output is a set of digests for the results. However, our framework provides the programmer with an interface nearly identical to the standard idiom: the programmer writes C code for the mappers and reducers.
3. Pantry also provides a simple verifiable database application that supports a (small) subset of SQL (Cassandra's CQL [1]). To do this, we develop general-purpose storage abstractions on top of PutBlock and GetBlock: a simple RAM and a searchable tree (§5). These abstractions are built from Merkle tree [37] structures. We expose these abstractions to the C programmer, using our C-to-constraints compiler. The result is that queries against these data structures (range queries, etc.) are performed verifiably *even though the client does not have the input locally*.
4. We describe an extension to support computations where the remote database contains information that the server wishes to keep private (§6).

To evaluate the system (§8), we experiment with the basic primitives, we run MapReduce on a variety of realistic sample computation (e.g., nearest neighbor calculation and string search), and we run our database queries on a sample database with roughly 32K rows. Although the overhead for the prover is too high, we find a number of input regimes in which the verifier can save work.

Summarizing, Pantry is the first built system for verifiable computation that offers a computational model with state. The presence of state broadens the range of supported computations to include those that interact with RAM, those with side-effects, and those where the client does not have all the input. In particular, Pantry is the first system where the client can verify computations based on a *digest* of the input. This allows verification costs to be sublinear in the input to the computation, removing a central restriction on the regimes of applicability of prior work. One of the particularly attractive aspects of Pantry is that our main application, verifiable MapReduce, is perfectly suited to the batching amortization necessary for the client to save work (§2).

To be sure, Pantry is not quite ready for practical use. The biggest issue is that the overhead for the server remains too high, which restricts our evaluation to smaller scales than would occur in real applications. Also, the database applications are somewhat unrealistic since we do not handle multi-user access. Nonetheless, in contrast to prior verifiable computation systems, Pantry supports realistic sample computations that are representative of actual applications of cloud computing.

## 2 Background and tools

In this section we describe the framework in which we work and the machinery on which we build. This framework is similar to that of prior work [40, 44, 52] and is closest to Zaatar [44], which is an interactive protocol. Our description below is influenced by [44, §2] and [52, §2]; see these sources for further detail.

### 2.1 Overview of the base machinery

Our base framework provides the following. A client, or *verifier* $\mathcal{V}$, sends a description of a computation $\Psi$ (for instance, a program in a high-level language) to a server, or *prover* $\mathcal{P}$. $\mathcal{V}$ also sends a *batch* of different inputs $x^{(1)}, \ldots, x^{(\beta)}$ to $\mathcal{P}$, who is expected to run $\Psi$ on each of those inputs; that is, $\mathcal{P}$ returns a batch of outputs $y^{(1)}, \ldots, y^{(\beta)}$, and if $\mathcal{P}$ computed correctly, $y^{(j)} = \Psi(x^{(j)})$ for $j \in \{1, \ldots, \beta\}$. We call each invocation of $\Psi$ in the batch an *instance*. (Note that $\mathcal{P}$ is an abstraction and could represent multiple machines, as in our MapReduce application in Section 4.)

$\mathcal{V}$ then engages $\mathcal{P}$ in an interactive protocol that allows $\mathcal{V}$ to check whether the server computed correctly. This protocol assumes a standard computational bound on $\mathcal{P}$ (for instance, that $\mathcal{P}$ cannot break a cryptographic primitive). However, the protocol does not make other assumptions about $\mathcal{P}$; its guarantees hold regardless of $\mathcal{P}$'s behavior. These guarantees are probabilistic, and the probability is over $\mathcal{V}$'s random choices:

- **Completeness.** If $y^{(j)} = \Psi(x^{(j)})$ for $j \in \{1, \ldots, \beta\}$, then if $\mathcal{P}$ follows the protocol, $\Pr\{\mathcal{V} \text{ accepts}\} = 1$.
- **Soundness.** If for any $j \in \{1, \ldots, \beta\}$, $y^{(j)} \neq \Psi(x^{(j)})$, then $\Pr\{V \text{ rejects}\} > 1 - \epsilon$, where $\epsilon$ can be made small.

The details of costs are in Section 2.3. A summary is that the protocol brings substantial overhead for $\mathcal{P}$. Also, $\mathcal{V}$ must incur a setup cost per batch; given a large enough batch size (thousands or tens of thousands for certain computations; see [44, §5]), $\mathcal{V}$ gains from using the framework (versus executing the $\beta$ instances itself).

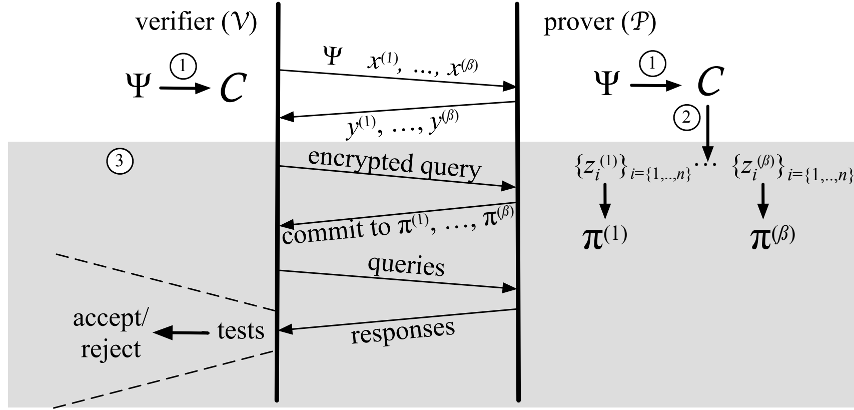


Figure 1—Base machinery for allowing a verifier $\mathcal{V}$ to verify the purported outputs of a computation $\Psi$ performed by a prover $\mathcal{P}$ on different inputs $x^{(1)}, \ldots, x^{(\beta)}$ (throughout, superscripts denote different instances of the same computation). Note that the prover could be distributed (each instance $j$ could execute on a different machine). Although computation need not happen in batch, verification cannot begin until $\mathcal{V}$ has all $y^{(j)}$. Verification is a three-step process. Step ①: $\mathcal{V}$ and $\mathcal{P}$ compile $\Psi$ to constraints $\mathcal{C}$. Step ②: $\mathcal{P}$ produces satisfying assignments $z^{(1)}, \ldots, z^{(\beta)}$ to $\mathcal{C}(X{=}x^{(1)}, Y{=}y^{(1)}), \ldots, \mathcal{C}(X{=}x^{(\beta)}, Y{=}y^{(\beta)})$, respectively. Step ③: $\mathcal{P}$ uses complexity-theoretic and cryptographic machinery to convince $\mathcal{V}$ that $\mathcal{P}$ holds satisfying assignments. $\mathcal{V}$'s querying work is reused across all instances in the batch.

## 2.2 Details of the base machinery

Verifiably outsourcing a computation is a three-step process, outlined in Figure 1. This section details these three steps, assuming one instance (so no batching). The next section revisits batching, and considers expressiveness and limitations.

**(1) $\Psi$ is represented as constraints.** The programmer begins by writing the computation, $\Psi$, in a high-level language. In our current framework, this language is a subset of C (see Section 2.3 for details). A compiler transforms a program in this language to a set of constraints, as we now detail.

In our context, a set of constraints is a system of equations in variables $(X,Y,Z)$, over a large finite field, $\mathbb{F}$; we choose $\mathbb{F} = \mathbb{F}_p$ (the integers mod a prime $p$), where $p$ is 128 or 192 bits. Each constraint has total degree 2, so each summand in a constraint is either a variable or a product of two variables. Variable $X$ represents the set of input variables, and variable $Y$ represents the set of output variables; for now, we assume that each of these sets has one element.

As notation, let $\mathcal{C}(X{=}x, Y{=}y)$ mean $\mathcal{C}$ with variable $X$ bound to $x$ and $Y$ bound to $y$; notice that $\mathcal{C}(X{=}x, Y{=}y)$ is a set of constraints over the variables $Z$. If for some $z$, setting $Z{=}z$ makes all constraints in $\mathcal{C}(X{=}x, Y{=}y)$ hold simultaneously, then $\mathcal{C}(X{=}x, Y{=}y)$ is said to be *satisfiable*, and $z$ is a *satisfying assignment*.

The compiler represents $\Psi$ as a set of constraints $\mathcal{C}$ such that: for all $x, y$, we have that $\mathcal{C}(X{=}x, Y{=}y)$ is satisfiable if and only if $y = \Psi(x)$. For example, add-1 is equivalent to the following constraints [9]:

$$\{Z - X = 0,\ Z + 1 - Y = 0\}.$$

Given $\Psi$ represented as constraints $\mathcal{C}$, given input $x$, and given purported output $y$, verification means persuading $\mathcal{V}$ that $\mathcal{P}$ holds an assignment $z$ that satisfies $\mathcal{C}(X{=}x, Y{=}y)$; the existence of such a $z$ implies $y = \Psi(x)$. This brings us to the next two steps.

**(2) $\mathcal{P}$ computes and identifies a satisfying assignment.** To identify a satisfying assignment, $\mathcal{P}$ goes constraint by constraint, "evaluating" each one to get a binding for each variable. Since most constraints introduce only one new variable, this process is straightforward (see the add-1 example above). But for some constraints, $\mathcal{P}$ must do additional work. An example is as follows; this will give some intuition for the techniques that we use in Section 3. Consider a code snippet, with a `!=` test [9]:

```
if (Z1 != Z2) {
    Z3 = 1;
} else {
    Z3 = 0;
}
```

This compiles to:

$$\mathcal{C}_{!=} = \left\{ \begin{array}{rcl} M \cdot (Z_1 - Z_2) - Z_3 & = & 0 \\ (1 - Z_3) \cdot (Z_1 - Z_2) & = & 0 \end{array} \right\}$$

Notice the auxiliary variable $M$; if $Z_1 \neq Z_2$, then $\mathcal{P}$ must set $M$ equal to the multiplicative inverse of $(Z_1 - Z_2)$, which $\mathcal{P}$ computes outside the constraint formalism. We call this "computing exogenously", and there is an analogy between computing inverses and supplying values from storage in Section 3.

The step of identifying a satisfying assignment is implemented by C++ code that is generated by our compiler; the code computes $\Psi(x)$ and performs the procedure sketched above.

**(3) $\mathcal{P}$ *argues* that it has a satisfying assignment.** $\mathcal{P}$ wants to prove that it has a satisfying assignment to $\mathcal{C}(X{=}x, Y{=}y)$; as noted above, this would convince $\mathcal{V}$ of the correctness of the purported output $y$ (in fact, it would convince $\mathcal{V}$ of something stronger, which is that the computation, as represented by constraints, was executed correctly). Of course, $\mathcal{P}$ could send the satisfying assignment, and $\mathcal{V}$ could plug it into $\mathcal{C}(X{=}x, Y{=}y)$ to check it, but that would be as much work as executing the computation.

Instead, $\mathcal{P}$ and $\mathcal{V}$ use an *efficient argument* protocol [26, 29, 30, 38]. Such a protocol is called an "argument", rather than a proof, because its soundness rests on a cryptographic assumption about the limits of $\mathcal{P}$'s computational power; in other words, a computationally unlimited $\mathcal{P}$ could convince $\mathcal{V}$ that false statements are true.

The arguments that our framework uses [26, 44, 46] compose PCPs [4, 5] with cryptographic commitments. PCPs yield a way to encode a satisfying assignment in a proof in such a way that $\mathcal{V}$ inspects only a small number of randomly-chosen locations in a purported proof $\pi$, applies simple tests to the values found at those locations, and then accepts or rejects, consistent with the completeness and soundness guarantees above. However, we do not want $\mathcal{V}$ to receive $\pi$ (it is far larger than the number of steps in the computation). Instead, $\mathcal{P}$ *cryptographically commits* to the contents of the encoded proof, after which $\mathcal{V}$ asks $\mathcal{P}$ for particular proof entries; the cryptographic guarantee is that $\mathcal{P}$'s replies are consistent with its original commitment, or else the protocol rejects. In other words, $\mathcal{P}$ acts like a fixed proof $\pi$, so $\mathcal{V}$ can apply the PCP tests to the replies of $\mathcal{P}$, and decide whether to accept or reject.

As with step 2, this step is implemented by C++ code that is generated by our compiler.

### 2.3 Costs, expressiveness, and limitations

The machinery described above is powerful; it offers the completeness and soundness guarantees from Section 2.1, with $\epsilon < 1/10^6$ (see [44, Apdx. A.2]). However, this machinery requires a particular usage model (batching), and limits expressiveness. We detail these restrictions below.

**Batched model.** The machinery is worthwhile to $\mathcal{V}$—in the sense that $\mathcal{V}$ saves work versus computing $\Psi$ itself—if $\mathcal{V}$ works in a batched model, as follows. Compilation (step (1)) happens once, and is done by $\mathcal{V}$ and $\mathcal{P}$. Computing $\Psi$ and identifying a satisfying assignment (step (2)) is a cost incurred by $\mathcal{P}$, and it happens once for each instance. The query creation cost (in step (3)) is incurred by $\mathcal{V}$, once, and then $\mathcal{V}$ submits the same queries to all proofs in the batch.

Notice that $\mathcal{P}$ is indifferent to batching, since each new proof requires computing the responses to the queries. Per-instance, $\mathcal{P}$'s work is $O(|Z| + |\mathcal{C}| \cdot \log |\mathcal{C}|)$ [52, §5], following a suggestion of [20]; here, $|Z|$ is the number of variables in $Z$, and $|\mathcal{C}|$ is the number of constraints in $\mathcal{C}$. Although this is not much more work than executing the computation in the constraint model, the asymptotic notation is hiding large constants (see Section 8).

$\mathcal{V}$'s costs are treated differently. $\mathcal{V}$ incurs a setup cost proportional to the execution cost of the computation, which then amortizes over the $\beta$ instances; this cost is $O(|Z| + |\mathcal{C}| + K)$, where $K$ is the number of additive terms in $\mathcal{C}$ [44, §4]. As with $\mathcal{P}$, the asymptotic notation hides high constants, mainly owing to the cryptographic commitment. Also, per-instance, $\mathcal{V}$'s "checking work" requires a linear pass over the input and output, plus a *constant* amount of work. Thus, in amortized terms, $\mathcal{V}$'s total per-instance work is $O(|x|+|y|+(|Z|+|\mathcal{C}|+K)/\beta)$. In other words, $\mathcal{V}$ breaks even—in the sense of saving CPU cycles from outsourcing—when $\beta$ is large enough to make the following expression (which captures the per-instance cost) less than the cost of running the computation locally: $\text{setup_cost}/\beta + \text{checking_work}$.

**Expressiveness and limitations.** Currently, our subset of C supports functions, structs, typedefs, preprocessor definitions, if-else statements, explicit type conversion, and all standard integer and bitwise operations. Additionally, the subset has partial support for pointers and loops; pointers must be compile-time constants and loops may iterate at most some fixed number of times (these limitations are similar to those of [40]).

Most operations have a concise representation as constraints (they do not add many variables or constraints). There are four exceptions. The first two are inequalities and bitwise operations; these operations separate numbers into their bits, perform the operation, and glue the number back together again [40, 46], requiring $\approx \log_2 |\mathbb{F}|$ constraints per operation. The other two are looping and if-else statements: loops are unrolled at compile time, and the costs associated with an if-else statement combine the costs of its then-block and its else-block [9].

Another limitation is that computations are hermetic: computations can interact with state neither as auxiliary input, nor during execution (there is no disk or RAM abstraction in the programming model), nor as auxiliary output. As a result, verifiability requires $\mathcal{V}$ to supply all inputs and receive all outputs. The sections ahead address this limitation.

## 3 Storage model and primitives

Pantry extends the computational model in Section 2 by adding a verifiable PutBlock/GetBlock primitive, described in this section. As an example use of this primi-

tive, we describe a verifiable MapReduce protocol (§4). We also layer higher-level storage abstractions (RAM, searchable trees) on top of this primitive, leading to verifiability for general-purpose programs (§5).

To explain Pantry's approach, we note that the "interface" to step (3) in Section 2.2 is a set of constraints and a purported satisfying assignment. Thus, a "first cut" approach to incorporating state into verifiable computation would be to explicitly represent load and store operations with constraints. However, we do not know how to represent indirection without incurring horrific expense: to our knowledge, memory would be an array of variables, and if addr is not compile-time resolvable, then "load(addr)" would require the moral equivalent of a giant case statement, resulting in a separate constraint for each possible value of addr. This approach would also require the input state to be available to the verifier $\mathcal{V}$.

To overcome these problems, we want to arrange for storage to live "outside" the constraints. Thus, we want a computational model that separates computation from state maintenance in the sense that the computation does not "execute" storage but can efficiently verify it. Given such a model, the hope is that we could use constraints to represent computation (as we do now) as well as the efficient checks of storage. But what model separates computation from state by applying efficient checks to state? This problem is actually well-studied, in the context of untrusted storage [17, 32, 37]: the state is represented by hash trees [37], and computing over that state involves trafficking in collision-resistant hashes.

If we could efficiently represent the computation of the hash function as constraints, then we could extend the computational model in Section 2 with the semantics of untrusted storage. At that point, a satisfying assignment to the constraints would imply correct computation and correct interaction with state; and at *that* point, we could use step (3) from Section 2.2 to prove to $\mathcal{V}$ that $\mathcal{P}$ holds such an assignment. We now describe this approach.

### 3.1 Verifiable blocks: overview

The lowest level of storage is a block store; it consists of variable-length blocks of data, in which the blocks are named by collision-resistant hash functions (CRHFs) of those blocks. Letting $H$ denote a CRHF, a correct block store is a map

$$S\colon \mathit{name} \to \mathit{block} \cup \perp,$$

where if $\mathit{block} = S(\mathit{name})$, then $H(\mathit{block}) = \mathit{name}$. In other words, $S$ implements the relation $H^{-1}$. This naming scheme allows clients to use untrusted storage servers [14, 17, 32, 34]. The technique's power is that given a name for data—which name a client is presumed to have obtained through trustworthy means—the client can check that the returned block is correct, in the sense of being consistent with its name. Likewise, a client that creates new blocks knows what names to give them, and can use those names as references later in the computation.

**function** GETBLOCK (name $n$)
  $\mathit{block} \leftarrow$ read block with name $n$ in block store $S$
  assert $n == H(\mathit{block})$
  **return** *block*

**function** PUTBLOCK (*block*)
  $n \leftarrow H(\mathit{block})$
  store $(n, \mathit{block})$ in block store $S$
  **return** $n$

Figure 2—"Pseudocode" for verifiable storage primitives; we use quotation marks because these primitives compile directly to constraints that enforce the required relation between $n$ and *block*.

But unlike the scenario in prior work, our $\mathcal{V}$ cannot actually check the contents of the blocks that it "retrieves" or impose the correct names of the blocks that it "stores", as the entire computation is remote. Instead, $\mathcal{V}$ represents its computations with constraints that $\mathcal{P}$ can satisfy only if $\mathcal{P}$ uses the right blocks. Another way to understand this approach is that $\mathcal{V}$ uses the verification machinery to outsource the checks themselves; in fact, because $\mathcal{V}$ outsources the storage checks to $\mathcal{P}$, $\mathcal{P}$ itself could be using an untrusted block store![2]

We will show in later sections how to write general-purpose computations in this model; for now, we illustrate the model with a simple example. Imagine that the computation takes as input the name of a block and returns the associated contents as output. The constraints themselves are set up to be satisfiable if and only if the return value hashes to the requested name. In effect, $\mathcal{P}$ is being asked to identify a preimage of $H$, which (by the collision-resistance of $H$) $\mathcal{P}$ can do only if it returns the actual block previously stored under the requested name.

### 3.2 Verifiable blocks: details and costs

We will work in a single-user model. Pantry extends the computational model in Section 2 by providing two primitives to the programmer:

```
block = GetBlock(name);
name = PutBlock(block);
```

These primitives are detailed in Figure 2. Notice that in a correct execution, $H$(block)=name. Given this relation, and given the collision-resistance of $H$, the programmer receives from GetBlock and PutBlock a particular stor-

[2] One might wonder whether $\mathcal{P}$ could also use an untrusted *CPU*, say if we were to represent the checking logic itself (step (3), §2.2) inside the verification machinery and then outsource the PCP checks themselves. This idea has been studied by theorists [4, 5]; for the sake of simplicity, we do not pursue it for now.

age model: $S$ functions as write-once memory, where the "addresses" are in practice unique, and where the addresses certify the data that lives at those addresses.

Note that this is an abstraction. First, how $S$ is actually implemented is unspecified here; the choice can be different for different kinds of storage (MapReduce, RAM, etc.). Second, the size of the blocks is unspecified. $S$ itself implements a map that is variable-length. For example, in the MapReduce application, an entire file will be a single block.

To bootstrap this certification, the client supplies one or more "addresses" (really names) as its input, and it may receive one or more names in its output, for use in further computations. These names are related to capabilities [24, 31]: with capabilities, a reference certifies to the system, by its existence, that the programmer is entitled to refer to a particular object; here, the reference itself certifies to the programmer that the system is providing the programmer with the correct object.

The preceding paragraphs explain the storage model. We now describe the constraints that enforce the model. A line of code "b = GetBlock(n)" compiles to constraints $\mathcal{C}_{H^{-1}}$, where: the input variable, $X$, represents the name; the output variable, $Y$, represents the block contents; and $\mathcal{C}_{H^{-1}}(X{=}n, Y{=}b)$ is satisfiable if and only if $b \in H^{-1}(n)$ (i.e., $H(b) = n$). The line "n = PutBlock(b)" compiles to the same constraints, except that the inputs and outputs are switched. Specifically, this line compiles to constraints $\mathcal{C}_H$, where: $X$ represents the block contents, $Y$ represents the name, and $\mathcal{C}_H(X{=}b, Y{=}n)$ is satisfiable if and only if $n = H(b)$.

Of course, in a program that invokes PutBlock and GetBlock, $\mathcal{C}_H$ and $\mathcal{C}_{H^{-1}}$ appear inside a larger set of constraints; thus, the compiler relabels the inputs and outputs of $\mathcal{C}_H$ and $\mathcal{C}_{H^{-1}}$ to correspond to intermediate program variables. As an illustrative example, consider the following computation:

```
add(int x1, name x2)
{
    block b = GetBlock(x2);
    /* assume that b is a field element */
    return b + x1;
}
```

The corresponding constraints are:

$$\mathcal{C} = \{Y - B - X_1 = 0\} \cup \mathcal{C}_{H^{-1}}(X{=}X_2, Y{=}B),$$

where the notation $X{=}X_2$ and $Y{=}B$ means that, in $\mathcal{C}_{H^{-1}}$ above, the appearances of $X$ are relabeled $X_2$ and the appearances of $Y$ are relabeled $B$. Notice that variable $B$ is unbound in $\mathcal{C}(X_1{=}x_1, X_2{=}x_2, Y{=}y)$. To satisfy those constraints, $\mathcal{P}$ must set $B{=}b$, which requires identifying a concrete $b$, from storage, such that $H(b){=}x_2$.

**Costs.** The main cost of GetBlock and PutBlock is the set of constraints required to realize the hash function $H$ in $\mathcal{C}_H$ and $\mathcal{C}_{H^{-1}}$. Ordinarily, we would adopt a widely-used function such as SHA-1 or SHA-256 for this purpose. But these functions make heavy use of bitwise operations, which do not have concise representations as constraints (§2.3). Instead, we use an *algebraic* CRHF, which we call GGH [21],[3] that is based on the hardness of well-known approximation problems in lattices; it fits our framework since GGH multiplies its input, represented as a vector, by a matrix, and meanwhile our constraints themselves are systems of equations over a finite field. Indeed, GGH requires 14× fewer constraints than SHA-1 would. Nevertheless, GGH is not devoid of bitwise operations and thus remains a relatively expensive operation (§8.2).

### 3.3 Guarantees and non-guarantees

Notice that the constraints do not capture the actual interaction with the block store $S$; the prover $\mathcal{P}$ is separately responsible for maintaining the map $S$. What ensures that $\mathcal{P}$ does so honestly? The high-level answer is the checks in the constraints plus the collision-resistance of $H$.

As an illustration, consider this code snippet:

```
n  = PutBlock(b);
b' = GetBlock(n);
```

In a reasonable (sequential) computational model, a read of a memory location should return the value written at that location; since our names act as "locations", a correction execution of the code above should have variables $b$ and $b'$ equal. But the program is compiled to constraints that include $\mathcal{C}_H$ (for PutBlock) and $\mathcal{C}_{H^{-1}}$ (for GetBlock), and these constraints could in principle be satisfied with $b' \neq b$, if $H(b') = H(b)$. However, $\mathcal{P}$ is prevented from supplying a spurious satisfying assignment because collision-resistance implies that identifying such a $b$ and $b'$ is computationally infeasible. That is, practically speaking, $\mathcal{P}$ can satisfy the constraints only if it stores the actual block and then returns it.

However, Pantry does not formally enforce *durability*: a malicious $\mathcal{P}$ could in principle discard blocks inside PutBlock yet still exhibit a satisfying assignment. Such a $\mathcal{P}$ might be caught only much later (when $\mathcal{V}$ issues a corresponding GetBlock, $\mathcal{P}$ would be unable to satisfy the constraints), and at that point, it might be too late to get the data back. For a formal guarantee of durability, one can in principle use other machinery [47]. Also, Pantry (like its predecessors) does not enforce *availability*: $\mathcal{P}$ could refuse to engage in the protocol or fail to supply a satisfying assignment, even if it "knows" how to do so.

[3] In particular, GGH here does not refer to the GGH lattice cryptosystem that has been "subject to cryptanalytic attacks" [39].

What Pantry enforces—via the completeness and soundness properties (§2.1) applied to the constraint representation of the computation—is *integrity*, meaning that purported memory values (the blocks that are used in the computation) are consistent with their names, or else the computation does not verify.

For the same reason, if $\mathcal{V}$ outsources a computation that executes "GetBlock(foo)", and foo is an erroneous name in the sense that it does not represent the hash of any block previously stored, then $\mathcal{P}$ has no way of providing a satisfying assignment. This is as it should be: the computation itself is erroneous (in this model, correct programs pass the assert in GetBlock; see Figure 2), so it is fitting that $\mathcal{P}$ has no computationally feasible way to make $\mathcal{V}$ consistently accept a purported output.

A limitation of this model is that $\mathcal{P}$ cannot prove to $\mathcal{V}$ that $\mathcal{V}$ made such an error; as far the verification machinery is concerned, this case looks like the case in which $\mathcal{P}$ simply refuses to provide a satisfying assignment. While that might be disconcerting, our goal here is to establish that remotely executed computations are consistent with the requested computation; program verification (establishing that the computation itself is expressed correctly) is a complementary concern with a vast literature.

# 4 Verifiable MapReduce

This section describes how Pantry provides verifiability in MapReduce computations.

To review the MapReduce programming model [15], the programmer provides two functions. The first is map(), whose input is a list of key-value pairs and whose output is another list of key-value pairs. The second is reduce(), whose input is a list of values associated with a single key and whose output is another list of values. As input to the computation, the programmer names a set of files in a directory; as output, the programmer receives a set of files.

**Interface.** The verifier $\mathcal{V}$ is the machine that invokes the MapReduce computation. $\mathcal{V}$ can be the *tracker* (the module that drives the computation, called the *master* by Dean and Ghemawat [15] and the *JobTracker* by Hadooop), or else a separate machine, such as a customer of the cloud. In Pantry's MapReduce, the programmer has additional responsibilities. First, partitioning is exposed to the programmer: map() partitions its output, and reduce() likewise reads from partitioned input. However, this restriction is neither onerous nor fundamental (details below). Second, the programmer must arrange for $\mathcal{V}$ to supply a list of names, or *digests* (we use the terms interchangeably from now on), one digest for each input file; likewise, $\mathcal{V}$ receives back a list of digests, one for each file produced by a reducer. Third, $\mathcal{V}$ receives intermediate digests: one for each (mapper, reducer) pair.

These intermediate digests are used during verification, which happens in two batches (§2.1). The first batch represents the map phase, with each mapper's work treated as a separate instance in the batch, and likewise for the second batch, where each reducer is a separate instance. As far as $\mathcal{V}$ is concerned, the explicit inputs and outputs are all digests, rather than the actual data; however, if $\mathcal{V}$ accepts both batches, then with high probability the cloud has produced the correct output digests. As noted in Section 3.3, this does not prove that the performers have stored the corresponding data durably, but it does establish that the digests are correct; they could be used as the input to a further computation, such as another MapReduce, another transformation of the data, or simply a computation whose purpose is to download the actual data that corresponds to the digests.

Below, we describe the mechanics, some awkwardness and limitations, and the costs of this design.

**Mechanics.** In this description, we will use the terms *explicit input* and *explicit output* to refer to the verifier-supplied inputs and verifier-received outputs. These will be *digests*, rather than data.

Our MapReduce tracker wraps the programmer-supplied map() and reduce() in functions mapper() and reducer(); see Figure 3. The tracker then instantiates mapper() and reducer() on various performing computers. Then, the verifier $\mathcal{V}$ and the performing computers transform mapper() and reducer() to constraints, and finally apply the verification machinery. Verification and execution can be decoupled, but the complete execution of the map phase must happen before verification of the map phase, and likewise for the reduce phase.

We now detail the verification step, beginning with some notation. Let $M$ and $R$ be the respective number of mappers and reducers. Also, recall that superscripts denote instances in a batch, $X$ denotes the explicit input variables (which may be a vector), and $Y$ denotes the explicit output variables or vector (§2).

As stated above, verification happens in two batches. The first batch establishes for $\mathcal{V}$ that each mapper() was executed correctly, meaning: the performer worked over the correct data, map()ed this data correctly, and partitioned the transformed data over the reducers correctly. As shown in Figure 3, each instance $j \in \{1,\ldots,M\}$ in the batch gets as its explicit input, $x^{(j)}$, the digest of a block (such as a file); the explicit output of an instance, $y^{(j)}$, is a vector with $R$ components, one for each reducer that this mapper is "feeding". Note that $\{y^{(j)}\}_{j=\{1,\ldots,M\}}$ are the $M \cdot R$ intermediate digests mentioned above.

$\mathcal{V}$ and the performing computers transform mapper() to constraints $\mathcal{C}_{\text{mapper}}$, using our C-to-constraint compiler; of course, GetBlock and PutBlock result in the constraints described in Section 3. At this point, mapper $j$,

```
DigestArray mapper(Digest X) {

    Block list_in = GetBlock(X);
    Block list_out[NUM_REDUCERS];
    Digest Y[NUM_REDUCERS];

    // invoke programmer-supplied map()
    map(list_in, &list_out);

    for (i = 0; i < NUM_REDUCERS; i++)
        Y[i] = PutBlock(list_out[i]);

    return Y;
}
```

```
Digest reducer(DigestArray X) {

    Block list_in[NUM_MAPPERS];
    Block list_out;

    for (i = 0; i < NUM_MAPPERS; i++)
        list_in[i] = GetBlock(X[i]);

    // invoke programmer-supplied reduce()
    reduce(list_in, &list_out);

    Y = PutBlock(list_out);

    return Y;
}
```

Figure 3—For verifiable MapReduce, Pantry applies the verification machinery to the depicted functions, mapper() and reducer(), which use the storage primitives from Section 3; their execution is verified in two batches.

playing the role of a prover, establishes that it has a satisfying assignment to $\mathcal{C}_{\text{mapper}}(X{=}x^{(j)}, Y{=}y^{(j)})$; for this purpose, $\mathcal{V}$ and mapper $j$ use the machinery in step (3) of Section 2.2.

$\mathcal{V}$ then shuffles the explicit outputs of the first phase to be the explicit inputs to the second phase. In notation, $\mathcal{V}$ shuffles $\{y^{(j)}\}_{j=\{1,\ldots,M\}}$ to be $\{x^{(j)}\}_{j=\{1,\ldots,R\}}$, where each $x^{(j)}$ is a vector of $M$ digests.

Verifying the reducers proceeds similarly. Verification establishes for $\mathcal{V}$ that each reducer() (see Figure 3) takes in the correct $M$ blocks and reduce()s them correctly. The explicit inputs to each reducer() are the intermediate digests just described; the explicit outputs are $\{y^{(j)}\}_{j=\{1,\ldots,R\}}$, where each $y^{(j)}$ is a single digest; and, letting $\mathcal{C}_{\text{reducer}}$ denote the constraint representation of the function reducer(), the verification machinery proves the satisfiability of $\mathcal{C}_{\text{reducer}}(X{=}x^{(j)}, Y{=}y^{(j)})$ for $j \in \{1, \ldots, R\}$.

**Limitations.** A limitation of our current implementation is that the programmer who supplies map() is responsible for partitioning the output into $R$ chunks (see Figure 3), and similarly with reduce() reading from $M$ inputs; meanwhile, exposing this kind of partitioning removes some of the benefit of the map/reduce abstraction. However, this limitation is not fundamental, as the tracker could draw the boundary differently, wrapping traditional map() and reduce() functions into mapper() and reducer(), with boilerplate code to partition.

A further limitation is the quadratic number ($M \cdot R$) of intermediate digests. We could instead have a linear number ($M + R$) of intermediate digests, using a design in which the mappers write to a global RAM, and the reducers read from it (see Section 5.1), but this requires some complexity at verification time. Also, quadratic intermediate state is not problematic per se; in traditional MapReduce, the master keeps $M \cdot R$ bytes of state [15].

Also, map() and reduce() are subject to the restrictions of any other computation in this model (§2.3); for instance, they pay for bitwise operations, and if they themselves need to use RAM, they incur the costs described in the next section.

Finally, each mapper() and reducer() compiles into constraints statically, which has two implications. First, $\mathcal{V}$ must be able to bound the size of the data that each mapper and each reducer works over. Second, even if a mapper() or reducer() is given a small amount of data, it must use the same constraints as the others, leading to waste. These limitations are fundamental; the first stems from the constraint model of computation (which unrolls loops), and the second from batching.

**Costs.** Recall that we are working in the batched model (§2.3). Thus, the fixed (query setup) cost for $\mathcal{V}$ is proportional to running one instance of mapper() and one instance of reducer(). Also, the variable (per-instance verification) costs are proportional to $M \cdot R$ because this is the number of explicit inputs and outputs that the verification machinery "sees" (in between the two phases). The total proving work is proportional to $M + R$. We quantify these costs in our experiments; for now, we note that if $M \cdot R$ is small relative to the MapReduce's input, then the amortized per-instance cost for $\mathcal{V}$ is *sublinear* in the total data processed by the MapReduce computation.

## 5 Verifiable data structures

We now describe how Pantry uses the primitives in the prior section to build higher-level abstractions: RAM, a searchable tree, and a simple database application. As with MapReduce in the prior section, we implement the higher-level abstractions in C code, using PutBlock and GetBlock. We then use these abstractions in some larger program, compile that program down to constraints, and apply the verification machinery to those constraints. This provides verifiability since (as discussed in Section 3) the prover cannot satisfy the constraints except by placing the "right" blocks (meaning ones previously stored) in the constraint variables that represent the out-

```
function LOAD(address a, digest d)
    ℓ ← ⌈log N⌉
    h ← d
    for i = 0 to ℓ − 2 do
        node ← GetBlock(h)
        x ← ith bit of a
        if x = 0 then
            h ← node.left
        else
            h ← node.right
    node ← GetBlock(h)
    return node.value
```

```
function STORE(address a, value v, digest d)
    path ← LoadPath(a, d)
    ℓ ← ⌈log N⌉
    node ← path[ℓ − 1]
    node.value ← v
    d′ ← PutBlock(node)
    for i = ℓ − 2 to 0 do
        node ← path[i]
        x ← ith bit of a
        if x = 0 then
            node.left ← d′
        else
            node.right ← d′
        d′ ← PutBlock(node)
    return d′
```

Figure 4—RAM operations implemented with verifiable blocks, using a Merkle tree [37]. $N$ is the number of addresses in memory.

put of GetBlock.

Our design problem, then, is to implement data structures in C using PutBlock and GetBlock. This problem is actually well-studied [14, 17, 32, 37]; However, we will work through the details for the sake of completeness and to highlight the sources of costs. Although this section does not introduce new individual techniques, we believe that it is a novel synthesis, as it illustrates how to provide state to programs that are represented as constraints; this has required design decisions, such as the choice of PutBlock and GetBlock as primitives, and the decision to build searchable trees directly from verifiable blocks rather than layered on top of RAM.

At the highest level, the technique is to embed in blocks the names (or references or hashes—these concepts are equivalent here) of other blocks. This creates a structure linked together by hashes, in which the "starting hash" (for instance, of the root of a tree) can be used to authenticate values in the root of the tree. The result is that it is possible to build data structures out of chains of "authenticated pointers". We give more details below.

### 5.1 Verifiable RAM

Pantry's verifiable RAM abstraction enables random access to contiguously-addressable, fixed-size memory cells. It exposes the following interface:

```
value = Load(address, digest);
new_digest = Store(address, value, digest);
```

Pseudocode for the implementations of Load and Store is in Figure 4.

The high-level idea behind this pseudocode is that the digest commits to the full state of memory [37], in a way that we explain shortly. Then, a Load guarantees that the claim "*address* contains *value*" is consistent with *digest*. For Store, the guarantee is that *new_digest* captures the same memory state that *digest* does with the exception that *address* now holds *value*.

To explain how a digest can commit to memory, we briefly rehash (no pun intended) Merkle trees [37]. Every node is named by a collision-resistant hash (which we denote $H$) of its contents. An interior node's contents are two names (themselves hashes), representing the node's left child and its right child. Each leaf node corresponds to a memory address, and contains the value currently held at the memory address. Then, the hash of the root node's contents is a digest $d$ that effectively commits to the state of memory. Indeed, if entity $A$ holds a digest $d$, and entity $B$ claims "the value at address $a$ is $v$", then $B$ could argue that claim to $A$: $B$ could exhibit the name of $a$'s sibling, the name of their parent, the name of that parent's sibling, the name of their parent, and so on, to the root. $A$ could then check that the hash relationships hold and match $d$. For $B$ to succeed in a spurious claim, it would have to identify a collision in $H$.

The pseudocode in Figure 4 is simply applying this idea: the verifiable blocks in Section 3 provide the required names-are-hashes referencing scheme, and the GetBlock invocations compile to constraints that force $\mathcal{P}$ to exhibit a path through the tree. Thus, using $\mathcal{C}_{\text{Load}}$ to denote the constraints that Load compiles to, the constraints $\mathcal{C}_{\text{Load}}(X{=}(a,d), Y{=}v)$ can be satisfied only if the digest $d$ is consistent with address $a$ holding value $v$, which is the guarantee that Load is supposed to be providing.

How does $\mathcal{P}$ identify a path through the tree? In principle, it could recompute the internal nodes on demand from the leaves. But for efficiency, our implementation caches the internal nodes to avoid recomputation.

To invoke Load or Store, the program must begin with a digest; in Pantry, $\mathcal{V}$ supplies this digest as part of the input to the computation. One way to bootstrap this is for $\mathcal{V}$ to first create a small amount of state locally, then compute the digest directly, then send the data to $\mathcal{P}$, and then use the verification machinery to track the changes in the digest. Of course, this requires that a computation's output include the new digest.

This brings us to the implementation of Store, which takes as input one digest and returns a digest of the new state. Store begins by placing in local variables the contents of the nodes along the required path (LoadPath in Figure 4 is similar to Load, and involves calls to GetBlock); this ensures continuity between the old state and the new digest. Store then updates this path by creating new verifiable blocks, starting with the block for address $a$ (which is a new verifiable block that contains a new value), to that block's parent, and so on, up to the root. Let $\mathcal{C}_{\text{Store}}$ denote the constraints that Store compiles to. To satisfy $\mathcal{C}_{\text{Store}}(X{=}(a,v,d), Y{=}d')$, $\mathcal{P}$ must (1) exhibit a path through the tree, to $a$, that is consistent with $d$, and (2) compute a new digest that is consistent with the old path and with the memory update. Thus, the constraints enforce the guarantee that Store promises.

**Costs.** Letting $N$ denote the number of memory addresses, a Load or Store compiles to $O(\log N)$ constraints and variables, with the constant mostly determined by the constraint representation of $H$ inside GetBlock and PutBlock (§3.2). As usual, $\mathcal{V}$ works in the batched model (its setup cost is proportional to one run of the program, its per-instance costs scale with the size of the input and output digests, etc.).

## 5.2 Search tree

We now consider a searchable tree that is amenable to queries; we wish to support efficient range queries over any keys for which the $<$ comparison is defined. Specifically, we wish to support the following API:

values = FindEquals(key, digest)
values = FindRange(key_start, key_end, digest)
digest = Insert(key, value, digest)
digest = Remove(key, digest)

To implement this interface, a first cut approach would be to use the general-purpose RAM abstraction defined above to build a binary tree or B-tree out of pointers (memory addresses). Unfortunately, this approach is more expensive than we would like, since every pointer access in RAM costs $O(\log N)$, leading to costs of $O((\log N) \cdot (\log m))$ for a lookup in a tree of $m$ elements.[4]

To get the per-operation cost down to $O(\log m)$, we use a special-purpose data structure. Before continuing, we should qualify our use of "special-purpose": yes, we are eschewing RAM (and again using collision-resistant hashes as names or references). But this strategy applies to a wide class of data structures.

Essentially, we use a Merkle tree with a different structure from the one above (§5.1). As with an ordinary search tree, each node here contains a key and one or more values corresponding to that key. Where an ordinary search tree would maintain pointers, our nodes contain names (or hashes) of their children. The nodes are in sorted order, and queries and updates on the tree (assuming it is balanced) take time that is logarithmic in the number of keys stored.

A lookup operation (FindEquals, FindRange) descends the tree, via a series of GetBlock calls. An update operation (Insert, Remove) first descends the tree (via GetBlock) to identify the node where the operation will be performed; then modifies that node (via PutBlock, thereby giving it a new name), and then updates the nodes along the path to the root (again via PutBlock), resulting in a new digest. As with RAM, these operations are expressed in C and compile to constraints; if $\mathcal{P}$ satisfies the resulting constraints then, unless it has identified a collision in $H$, it is returning the correct state (in the case of lookups) and the correct digests (in the case of updates).

## 5.3 Verifiable database

To demonstrate the usefulness of our verifiable data structures, we use them to implement a simple verifiable database.

$\mathcal{V}$ specifies queries in a subset of the Cassandra Query Language (CQL) [1], itself a subset of SQL. This subset supports the following non-transactional queries on single tables: SELECT (on single columns), INSERT, UPDATE, DELETE, CREATE, and DROP COLUMNFAMILY. $\mathcal{V}$ and $\mathcal{P}$ then convert each query into C using a translator derived from Cassandra's query parser; this C calls the APIs from Sections 3.2 and 5.2, and is then compiled into constraints.

The database itself has a simple design. First, each row of every table is stored as a verifiable block (and can be accessed through the GetBlock/PutBlock API). Second, these blocks are pointed to by one or more indexes; there is a separate index for each column that the author of the computation author wants to be searchable. Indexes are implemented as verifiable search trees, as described above (§5.2).

Given the above structure, query execution works as follows. A SELECT compiles into C code that invokes FindEquals or FindRange on the tree that corresponds to the column being queried; the result is to place into program variables hashes that refer to the queried blocks. Then, the C code uses these hashes to fetch the actual rows, using GetBlock. Similarly, an UPDATE compiles to code that finds references to the matching row blocks, updates the rows, stores them with PutBlock, and then uses the Insert and Remove operations to update the trees. INSERT and DELETE[5] function similarly.

[4] We assume that the tree is balanced; building a self-balancing tree is future work.

[5] Pantry does not currently support reclaiming the storage of unused verifiable blocks, however.

The above design is of course straightforward. The interesting part is that because the design uses verifiable data structures and because the C code that implements the functionality is compiled into constraints, we get strong integrity guarantees—with little programmer effort beyond implementing the data structures and queries.

### 5.4 Limitations

As we explain in Section 2.3, one of the limitations of our computational model is that all loops must have fixed bounds and must be fully unrolled at compile time. This restriction has several implications for verifiable data structures. First, because the loops that traverse them have a fixed number of iterations, data structures must have a fixed maximum size (i.e., a fixed number of cells in RAMs and hash tables, and a maximum depth for trees). Similarly, the maximum number of results returned by queries such as the search tree's FindRange operation must also have a static limit. Third, as every operation is compiled into a fixed number of constraints, $\mathcal{P}$'s running time to perform the operation and to find a satisfying assignment to the constraints is fixed regardless of the number of elements that the data structure holds.

## 6 Extensions to handle private state

Pantry's innovations to incorporate state (§3–5) apply to any verifiable computation protocol that works in the constraint model or equivalent. Pantry uses Zaatar [44] because among the general-purpose schemes for verifiable computation, its per-instance costs for $\mathcal{V}$ and $\mathcal{P}$ are lowest. However, for some applications it is advantageous to use verifiable computation protocols with an additional property, namely *zero-knowledge* [23]; that is, we require that the verifiable computation protocol not reveal information to the verifier beyond what is implied by the output of the computation.

There are a number of verifiable computation protocols that provide zero-knowledge [8, 27, 29, 40]; the Pinocchio protocol [40], is a notable example, as it is closely related to Zaatar and also aims toward practice. Since Pinocchio essentially works in the constraint model, Pantry can use Pinocchio in place of Zaatar. An implementation of this extension is work in progress.

The main virtue of this substitution is letting Pantry support computations where the prover wants to keep some of the inputs and state private but is willing to let clients learn the outputs. For instance, consider an application where a server maintains a central database consisting of pictures of suspects, and is willing to let clients (e.g., surveillance cameras) verifiably lookup if someone's picture matches an entry in the database, but the server wants to keep its central database secret.

## 7 Implementation details

Our implementation of the design in Sections 3–5 is based on the released Zaatar compiler [44], updated with 13,300 lines of additional Java to take C as input, instead of SFDL. We implement verifiable blocks (§3) by adding 2200 lines of Java, 1900 lines of Go, and 200 lines of Python to the compiler pipeline, and 300 lines of C++ to the prover. The MapReduce framework (§4) requires 1500 lines of C++ code, and the verifiable data structures (§5) require another 400 lines of C that is compiled into constraints by our compiler and 200 lines of C++ code in the prover. The DB app requires 1200 lines of Java, which we use for CQL interpretation (currently our DB application is compiled with queries statically).

The constraints for verifiable blocks implement $H$ as (a variable-length version of) the GGH [21] hash function. Using the notation in [21], GGH hashes $m$ bits into $n \cdot \log q$ bits. Based on the analysis in a survey by Micciancio and Regev [39], we set these parameters as $m$=5376, $n$=64, and $q$=4096, to achieve an equivalent of at least 114 bits of security. To support variable-length input, we use a a prefix-free variant of the Merkle-Damgård transform [28, Ch. 4.6.4], proposed by Coron et al. [13], that prepends the input with its length.

The compiler transforms programs, written in a subset of C (§2.3), into a list of assignment statements and then outputs a constraint or pseudoconstraint for each statement (the compiler is derived from Fairplay's [35] and is described elsewhere [9, 44]).

The pseudoconstraints abstract operations (inequality comparisons, bitwise operations, etc.) that require multiple constraints; a second compiler pass expands the pseudoconstraints *and* outputs C++ code that $\mathcal{P}$ executes, to identify a satisfying assignment. Building on this compiler, we add one pseudoconstraint each for GetBlock and PutBlock; in the compiler's second pass, these pseudoconstraints compile into $\mathcal{C}_{H^{-1}}$ and $\mathcal{C}_H$, respectively (with appropriate renaming; see Section 3.2). The compiler also outputs C++ code that $\mathcal{P}$ will execute, to implement the actual interaction with storage (denoted $S$ in Section 3). The current implementation of $\mathcal{P}$ uses the Kyoto Cabinet [2] persistent key-value store to realize $S$, the mapping of digests to blocks.

## 8 Experimental evaluation

We evaluate Pantry by measuring its end-to-end performance. We use a set of MapReduce computations and a set of database queries as our benchmark computations. Figure 5 summarizes our evaluation results.

### 8.1 Method and setup

We use the following benchmark computations to measure Pantry's end-to-end performance.

| | |
|---|---|
| Pantry expands the class of computations that can be efficiently outsourced; under Pantry, a client can verify computations even if it does not have the full input to those computations. | §8.1 |
| The total cost to the prover under verifiable MapReduce is several orders of magnitude higher than that of native unverifiable execution, but the client saves work as long as it outsources a sufficient number of mappers and reducers. | §8.3 |
| The cost of verifying the correct execution of remote database queries has excellent scaling behavior, since the costs grow logarithmically in the size of the database. | §8.2, §8.4 |

Figure 5—Summary of main evaluation results.

| block size | # constraints produced by GetBlock |
|---|---|
| 0.5 KB | 6,600 |
| 2.0 KB | 24,000 |
| 8.0 KB | 91,000 |

Figure 6—The number of constraints produced by the GetBlock operation; PutBlock is the same. The number scales linearly in the size of the block being stored or retrieved. The number of variables in the constraint set is roughly the same as the number of constraints (not depicted).

| | #constraints produced by | |
|---|---|---|
| number of addresses in RAM | Load | Store |
| $2^5$ | 18,000 | 35,000 |
| $2^{10}$ | 34,000 | 67,000 |
| $2^{20}$ | 66,000 | 130,000 |

Figure 7—The number of constraints in Pantry for verifiable RAM operations, with varying number of addresses, when the size of the block stored in a RAM cell is at most 125 KB. The number of constraints scales logarithmically with the number of addresses in RAM. The number of variables in the constraint set is roughly the same as the number of constraints (not depicted).

1. Searching a sequence of $k$ nucleotides in a DNA sequence of length $m$.
2. Computing the dot product between two vectors of length $m$.
3. Searching for a nearest-neighbor in a list of $m$ samples, where each sample is a $d$-dimensional vector.
4. Computing the covariance matrix for a dataset of $m$ samples with $d$ dimensions each.
5. Computing the second moment of a frequency vector of length $m$ [12].
6. Querying a database to select and update rows in a table.

We express the first five benchmark computations as MapReduce programs. For the MapReduce computations, all inputs are 32-bit integers. For the database benchmark, we use a database that is pre-populated with information about students; the database has a single table; there is one row per student in the table, and columns store information about each student (name, age, class, average, etc.).

We use the same finite field for all of the above computations; the prime modulus is 128 bits. Pantry inherits its cryptographic tools and implementation from Zaatar [44]: Pantry uses ElGamal encryption (for cryptographic commitment; see Section 2.2, step (3)) with a key of size 1024 bits, and where the protocol requires random bits, Pantry uses a pseudorandom generator, specifically the ChaCha stream cipher [7].

For experiments, we use a local cluster of machines; each machine runs Linux on an Intel Xeon processor E5540 2.53 GHz with 48GB of RAM. To measure CPU time, we use `getrusage()`.

For all of the above computations, our baseline is the cost of executing the computation, written in C. Our baseline works with digests and blocks, and performs the same checks as do computations under Pantry; the blocks are stored on a local disk, indexed by hash, in the Kyoto Cabinet store [2].

Our baseline pays for both local storage (representing the baseline of fully local computations) *and* the computational costs (hashing, etc.) to outsource storage verifiably [17, 32, 34], when in fact a natural baseline would be one or the other of these two costs. In future work, we will avoid such an optimistic baseline, by separating the two costs and comparing Pantry to two baselines.

### 8.2 Microbenchmarks

To evaluate the overhead of our storage primitives, we run microbenchmarks. For GetBlock and PutBlock, we vary the size of the block retrieved or stored, and record the number of constraints and the number of variables that Pantry's compiler generates. For Load and Store, we vary the number of cells in the RAM, and measure the number of constraints and the number of variables in Pantry's compiler's output. Figures 6 and 7 summarize our measurements. As expected, the costs of the GetBlock and PutBlock operations scale linearly in the size of the block being stored or retrieved, and the costs of the Load and Store operations scale logarithmically in the number of addresses in the RAM.

### 8.3 Performance of verifiable MapReduce

We run the benchmark computations using the following input sizes: DNA substring search where $m$=10,000 and $k$=19; dot product where $m$=10,000; nearest-neighbor

| computation (Ψ) | local | prover's costs under Pantry solve constraints | construct proof vector | crypto work | answer queries | total CPU time |
|---|---|---|---|---|---|---|
| DNA substring search | 0.2 s | 0.4 min | 0.6 min | 3.0 min | 1.6 min | 5.7 min |
| dot product computation | 1.0 s | 2.3 min | 13.5 s | 1.2 min | 1.0 min | 4.8 min |
| nearest neighbor search | 0.6 s | 1.4 min | 14.6 s | 1.2 min | 0.9 min | 3.7 min |
| covariance matrix computation | 0.2 s | 1.2 min | 11.4 s | 1.1 min | 0.7 min | 3.2 min |
| second frequency moment | 0.6 s | 1.2 min | 7.4 s | 0.6 min | 0.5 min | 2.5 min |

Figure 8—Pantry's prover's end-to-end costs (decomposed) compared to the cost of executing the computation locally for various benchmark computations. The prover's end-to-end costs are high relative to an unverifiable execution, mainly owing to the model of computation and the costs associated with the cryptographic verification machinery that Pantry uses.

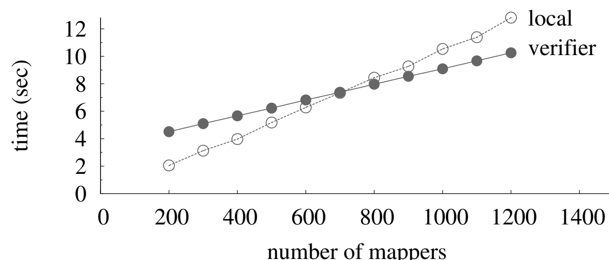


Figure 9—Total CPU time of the verifier under Pantry and the baseline for DNA substring search as the number of mappers is increased, where each mapper searches for a short nucleotide sequence in a DNA string of length 1000 nucleotides and a reducer is allocated for every 10 mappers. For experiments involving more than 650 mappers, the cost of doing the verification using Pantry is less than the cost of rerunning the computation locally.

search where $m$=10,000 and $d$=10; covariance matrix computation where $m$=1000 and $d$=10; and second frequency moment computation where $m$=10,000. We use 10 mappers and 1 reducer to execute the computation. Figure 8 summarizes the results. As expected, native execution is several orders of magnitude faster than Pantry's prover.

However, the verifier can save work by outsourcing a computations, above a certain size. To demonstrate this, we use the DNA substring search example. We vary the input size from $m$=200,000 to $m$=1,200,000 (keeping $k$=19), and give each mapper a sequence of 1000 nucleotides, resulting in $M$ (the number of mappers) varying from 200 to 1200. Figure 9 depicts the results. For experiments involving more than 650 mappers, the cost of using Pantry and verifying is less than the cost of running the computation locally. (Note that each local instance costs 10 ms because of the costs associated with computing hashes of the computation's inputs and outputs stored in a Kyoto Cabinet database.)

### 8.4 Performance of verifiable database queries

We pre-populate three databases with varying number of rows (1024, 8192, and 32,768 rows), and run three queries against them. Query 1 is "SELECT * FROM Student WHERE Average $>$ 90 LIMIT 5". Query 2 is "SELECT * FROM Student WHERE Class = 2009 LIMIT 5". Query 3 is to first perform query 1 and insert the result into another database. Figure 10 depicts our results. By design, the cost of running these queries verifiably on the server grows logarithmically in the size of the database, and our measurements confirm that scaling behavior.

### 8.5 Discussion

Our evaluation makes clear that although we can run our applications on inputs of reasonable size, the overhead for the prover remains a real problem; also, the verifier has to batch-verify several thousands to tens of thousands of instances, in our examples, to save work. Although these batch sizes are realistic in scenarios such as MapReduce and perhaps acceptable in usage models where the remote data is simply unavailable to the client, it is desirable to reduce them. Furthermore, although in principle we handle general state, realistically our computational model is not truly general purpose, insofar as efficient verifiers must use our special-purpose data structures rather than generic RAM operations.

Some of the limitations of Pantry will be ameliorated by optimizations in work we plan to do. For one thing, we can batch the query sub-computations, since all instances of GetBlock and PutBlock have the same query structure—this will save work for the verifier. We also have a design for handling dynamic loop bounds using the state machinery; this will allow a wider range of computations to be expressed efficiently.

## 9 Related work

The problem of verifiable outsourced computation is an old one, and there are many implemented systems in the literature (e.g., see [45] for partial surveys of this literature). Pantry descends from two distinct strands of research in this area: the emerging area of proof-based verifiable computation and the substantial body of work on untrusted storage.

In the last few years, there has been interesting new work on built systems for general-purpose verifiable computation using sophisticated tools from complexity theory and cryptography. The novel aspect of this work is its focus on practical performance (in the context of ac-

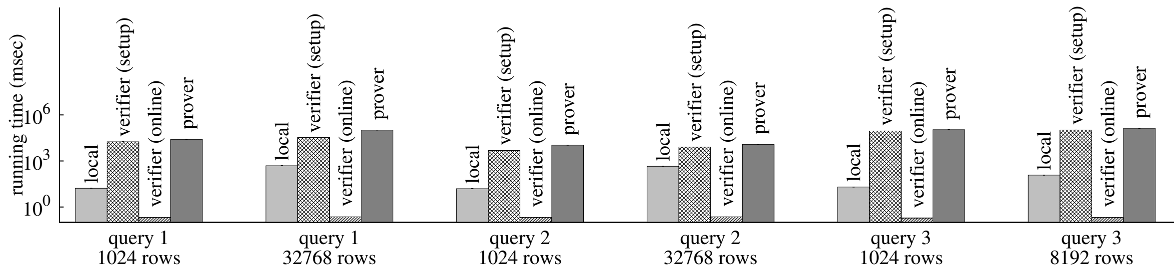


Figure 10—Cost of executing queries (stated in §8.4), locally without verification and under Pantry, under various table sizes. The cost to the verifier has two components: a *setup cost*, which can be reused across multiple instances of the same query in a batch, and an *online cost*, which scales linearly with the number of instances in a batch. Verifying remote database queries has excellent scaling behavior, since the costs grow logarithmically in the size of the database.

tual implementations); previous work in the theory community used outsourced computation as a motivational example but was primarily concerned with asymptotics. Pantry is built from one approach to such systems [43–46], which are interactive protocols based on [26]. Other work in this area includes [11, 40, 50]; we chose the foundation that we did because [11, 50] imposes certain restrictions that would make our work here difficult (although the work of [52] partially relaxes these) and [40] gives up performance to eliminate interactivity (although it has superior amortization of setup costs).

All of these efforts use essentially the same computational model; in particular, they all require the client to materialize the input and the computation to be side-effect free (§2.3). However, there are antecedents of the kinds of ideas about using state that we develop here in some of these papers and in closely related theory literature. Specifically, [20] alludes to handling state via signatures, as does [25]. Moreover, very recent work [6] spells out a similar idea in the context of a theoretical protocol for verifiable computation that works in the RAM model.

There is also a body of this work on this problem (starting from [19]) which relies on fully-homomorphic encryption (FHE) and ideas from secure multi-party protocols. However, due to the costs of FHE, none of these protocols has resulted in built systems to date.

The implementation of storage used in Pantry is squarely in the tradition of the large body of work on untrusted storage. In particular, the data representations we use to handle state are descended from the representations pioneered by [17, 37]. However, the literature on untrusted storage (including the line of work on proofs of retrievability [47] and the recent work on untrusted cloud storage [16, 34]) is aimed at solving somewhat different problems (durability, storage consistency respectively), and is not directly applicable.

## 10 Summary and conclusion

This paper presents Pantry, a solution to the problem of general-purpose verifiable computation (VC) with state; to our knowledge, it is the first built system for VC that does not pay preposterous performance for representing complicated state. Pantry combines a state of the art interactive protocol for VC [44] with hash-tree data structures. The result is a system that can represent standard programming idioms relatively efficiently and that allows the verifier to supply *digests* of inputs in place of the inputs themselves, broadening the set of computations amenable to verifiability. Notably, the ability to work with digests of the input means that Pantry's verification costs can be sublinear in the input to the computation—the first general-purpose system for verified computation for which this holds. As a demonstration, we implement a framework for verifiable MapReduce and a simple verifiable database.

Verifiable MapReduce is a particularly important computation both because of its relevance for the cloud computing scenarios in which verifiability is important as well as the fact that it is particularly amenable to the batched model in which the protocol of [44] breaks even. Our performance on the sample computations is not as good as we would like, particularly in its prover overhead, but it is close. In situations where we expect to pay something for verifiability and there is plenty of computational power (e.g., in the cloud), the protocol of Pantry might be reasonable.

## References

[1] Cassandra CQL. `http://cassandra.apache.org/doc/cql/CQL.html`.

[2] Kyoto Cabinet. `http://fallabs.com/kyotocabinet/`.

[3] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer. SETI@home: An experiment in public-resource computing. *CACM*, 45(11):56–61, Nov. 2002.

[4] S. Arora, 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.

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

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

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

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

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

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

[11] G. Cormode, M. Mitzenmacher, and J. Thaler. Practical verified computation with streaming interactive proofs. 2011. `http://arxiv.org/abs/1105.2003`.

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

[13] J.-S. Coron, Y. Dodis, C. Malinaud, and P. Puniya. Merkle-damgård revisited: how to construct a hash function. In *CRYPTO*, 2005.

[14] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *SOSP*, 2001.

[15] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. In *OSDI*, 2004.

[16] A. J. Feldman, W. P. Zeller, M. J. Freedman, and E. W. Felten. SPORC: Group collaboration using untrusted cloud resources. In *OSDI*, 2010.

[17] K. Fu, M. F. Kaashoek, and D. Mazières. Fast and secure distributed read-only file system. In *OSDI*, 2000.

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

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

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

[21] O. Goldreich, S. Goldwasser, and S. Halevi. Collision-free hashing from lattice problems. *Electronic Colloquium on Computational Complexity (ECCC)*, TR96-042, 1996.

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

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

[24] N. Hardy. The Confused Deputy: (or why capabilities might have been invented). *ACM SIGOPS Operating Systems Review*, 22(4):36–38, Oct. 1988.

[25] Y. Ishai. Personal communication, 2012.

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

[27] Y. Ishai, E. Kushilevitz, R. Ostrovsky, and A. Sahai. Zero-knowledge from secure multiparty computation. In *STOC*, 2007.

[28] J. Katz and Y. Lindell. *Introduction to Modern Cryptography*. Chapman & Hall / CRC Press, 2007.

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

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

[31] H. M. Levy. *Capability-Based Computer Systems*. Digital Press, 1984.

[32] J. Li, M. N. Krohn, D. Mazières, and D. Shasha. Secure untrusted data repository (SUNDR). In *OSDI*, 2004.

[33] 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.

[34] P. Mahajan, S. Setty, S. Lee, A. Clement, L. Alvisi, M. Dahlin, and M. Walfish. Depot: Cloud storage with minimal trust. *ACM Trans. on Comp. Sys.*, 29(4), Dec. 2011.

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

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

[37] R. C. Merkle. Digital signature based on a conventional encryption function. In *CRYPTO*, 1987.

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

[39] D. Micciancio and O. Regev. Lattice-based cryptography. In D. J. Bernstein and J. Buchmann, editors, *Post-quantum Cryptography*. Springer, 2008.

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

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

[42] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. van Doorn, and P. Khosla. Pioneer: Verifying integrity and guaranteeing execution of code on legacy platforms. In *SOSP*, 2005.

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

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

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

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

*USENIX Security*, 2012.

[47] H. Shacham and B. Waters. Compact proofs of retrievability. In *ASIACRYPT*, Dec. 2008.

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

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

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

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

[52] V. Vu, S. Setty, A. J. Blumberg, and M. Walfish. A hybrid architecture for interactive verifiable computation. In *IEEE Symposium on Security and Privacy*, 2013.