

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 a client outsources a job to a third party (e.g., the cloud), how can the client check the result, without re-executing the computation? Recent work in *proof-based verifiable computation* has made significant progress on this problem by incorporating deep results from complexity theory and cryptography into built systems. However, these systems work within a stateless model: they exclude computations that interact with RAM or a disk, or for which the client does not have the full input.

This paper describes *Pantry*, a built system that overcomes these limitations. *Pantry* composes proof-based verifiable computation with untrusted storage: the client expresses its computation in terms of digests that attest to state, and verifiably outsources *that* computation. Using *Pantry*, we extend verifiability to MapReduce jobs, simple database queries, and interactions with private state. Thus, *Pantry* takes another step toward practical proof-based verifiable computation for realistic applications.

1 Introduction

This paper addresses a fundamental problem in systems security: how can a local computer verify the correctness of a remote execution? (Checking that the given program was expressed correctly is a complementary concern, studied by the field of program verification.) Our focus on execution verification is motivated by large MapReduce jobs, remote database queries, and cloud computing more generally. In these scenarios, the causes of incorrect execution include corruption of input data in storage or transit, hardware faults, platform bugs, and misconfiguration. Unfortunately, the faults, and their effects, may not be visible as such. Indeed, when a job completes, after having processed petabytes of data, how can the client be sure that the output is correct [77]?

The client could audit the output [55], but this technique fails if a problem happens outside the selected sample. The client could replicate the computation (using state machine replication [23], quorums [51], or outsourcing to two clouds [5, 22]), but this technique works only if replica faults are uncorrelated. The client could trust

the remote hardware and use attestation [60, 63], but what if the hardware is faulty? The client could use a tailored solution [8, 13, 18, 28, 39, 58, 70, 73, 75], but such solutions are not available for all applications.

Perhaps surprisingly, the client can receive a guarantee that covers the entire execution of the computation, that makes no assumptions about the performing platform (other than cryptographic hardness assumptions), and that applies generally. In *proof-based verifiable computation*, the performing computer (or *prover*) returns the results along with a *proof* that the client (or *verifier*) can efficiently and probabilistically check. If the entire computation was executed correctly, the client accepts, and if there is any error, the client rejects with high probability.

These protocols are based on deep theoretical tools: probabilistically checkable proofs (PCPs) [6, 7], interactive proofs [9, 37, 38, 48, 69], and cryptography [15, 19, 31, 32, 42, 45]. This theory provides very strong guarantees and is usually phrased as defending against an arbitrarily malicious prover. Note that maliciousness is not an accusation but rather a comprehensive *model* that includes benign malfunctions with unpredictable effects.

Recent works have aimed to realize proof-based verifiable computation in built systems [12, 24, 59, 64–67, 71, 72, 74]. On the one hand, these systems appear to approach practicality. Some of them come with compilers that allow programmers to express computations in a high-level language [12, 59, 65, 67, 74]. And the best of them achieve reasonable client performance, provided that there are many identical computations (with potentially different inputs) over which to amortize overhead—a requirement met by typical data-parallel cloud computing applications.

On the other hand, almost none of these systems admit a notion of state or storage:¹ their compilation target is *constraints*, a generalization of circuits (§2). Given this “assembly language”, the computation cannot feasibly use memory, and the client must handle all of the input and output. Besides hindering programmability, these limitations are inconsistent with remotely stored inputs (as in MapReduce jobs, queries on remote databases, etc.); for example, verifying a large MapReduce job would require the client to materialize the entire dataset.

This paper introduces *Pantry*, the first system to provide verifiable computation with state. To do so, *Pantry* marries machinery for verifying pure computations with

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author.

Copyright is held by the owner/author(s).
SOSP '13, November 3–6, 2013, Farmington, Pennsylvania, USA.
ACM 978-1-4503-2388-8/13/11
<http://dx.doi.org/10.1145/2517349.2522733>

¹The exception is concurrent work by Ben-Sasson et al. [12]; see §9.

techniques from untrusted storage [17, 29, 47, 53]. While this picture is folklore among theorists [11, 15, 32, 41], the contributions of Pantry are to work out the details and build a system, specifically:

(1) Pantry enhances state of the art systems (§2) for verifiable computation (Ginger [67], Zaatat [65], Pinocchio [59]) with a storage abstraction (§3). The programmer expresses a computation using a subset of C plus two new primitives—PutBlock and GetBlock—and the Pantry compiler produces appropriate constraints. These primitives name data blocks by a cryptographic digest, or hash, of their contents. Such blocks are used extensively in systems for untrusted storage [29, 47]; however, in Pantry, the verifier will not be fetching the blocks to check them. The key insight here is that there exist hash functions that are amenable to the constraint formalism.

(2) Using PutBlock and GetBlock, we build a verifiable MapReduce framework (§4). The programmer writes Map and Reduce functions, much as in standard MapReduce frameworks. Here, however, input and output files are named by the digests of their contents.

(3) We also use PutBlock and GetBlock (together with well-known techniques [17, 53]) to build higher-level storage abstractions: a RAM and a searchable tree (§5). We use the tree to build a database application that supports verifiable queries in a (small) subset of SQL. The notable aspects here are the placement of functionality and the result: the abstractions are exposed to the C programmer, they need not be built into the compiler, and operations on these abstractions happen verifiably even though the client does not have the state.

(4) We compose PutBlock and GetBlock with a zero-knowledge variant of Pinocchio [32, 59], to build applications in which the prover’s state is private: face matching, toll collection, etc. (§6).

The components just described have awkward usage restrictions (the database is single-writer, iteration constructs need static upper bounds, etc.), due in part to the clumsiness of the constraint formalism. Worse, the measured cost (§8) of the implementation (§7) is very high: the prover’s overhead is tremendous, and the verifier incurs a similarly high per-computation setup cost, requiring many invocations to justify this expense.

However, compared to prior systems for verifiable computation (§9), Pantry improves performance: by not handling inputs, the verifier saves CPU and network costs. This effect, together with Pantry’s enhanced expressiveness, expands the universe of applications for which verification makes sense (§10). MapReduce, for example, works over remote state, and is well-suited to amortizing the setup costs, since it entails many identical computations. And the private state applications provide functionality that does not exist otherwise or previously required intricate custom protocols. In summary, Pantry extends

proof-based verifiable computation to real applications of cloud computing (albeit at much smaller scales for now).

2 Pantry’s base: Zaatat and Pinocchio

We present Zaatat [65] and Pinocchio [59], and the underlying theory, in a unified framework. Similar frameworks appear in prior work [59, 65–67, 74], and aspects of our presentation are borrowed [65, §2][74, §2].

2.1 Overview of Zaatat and Pinocchio

A client, or *verifier* \mathcal{V} , sends a program Ψ , expressed in a high-level language, to a server, or *prover* \mathcal{P} . \mathcal{V} sends input x and receives output y , which is supposed to be $\Psi(x)$. \mathcal{V} then engages \mathcal{P} in a protocol that allows \mathcal{V} to check whether \mathcal{P} executed correctly. This protocol assumes a computational bound on \mathcal{P} (e.g., that \mathcal{P} cannot break a cryptographic primitive). However, the protocol makes no other assumptions about \mathcal{P} : its guarantees hold regardless of how or why \mathcal{P} malfunctions. These guarantees are probabilistic (over \mathcal{V} ’s random choices):

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

Given a specific computation Ψ , we call each invocation of it an *instance*. The per-instance costs for \mathcal{V} are very low. However, in order to participate in the protocol, \mathcal{V} incurs a setup cost for each Ψ , which amortizes over multiple instances, either over a batch [65] or indefinitely [59] (see Section 2.3).

2.2 Zaatat and Pinocchio in more detail

Verifiably outsourcing a computation happens in three steps, depicted in Figure 1. First, a compiler transforms the computation Ψ to an algebraic system of *constraints*. Next, \mathcal{P} produces a solution to these constraints that implies $y = \Psi(x)$. Finally, \mathcal{P} convinces \mathcal{V} that it has produced such a solution, thereby establishing that $y = \Psi(x)$. We now describe each step in detail; for the time being, we assume only one instance (§2.3 revisits).

(1) Ψ is represented as constraints. The programmer begins by expressing a computation, Ψ , in a subset of C or an equivalent high-level language (described in §2.4) and invoking a compiler [50, 59, 65, 67]. Here, we focus on the compilation target: *a set of constraints* [20, 67].

In our context, a set of constraints \mathcal{C} 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 large (e.g., 128 bits). Each constraint has total degree 2, so each summand in a constraint is either a variable or a product of two variables. Variables X and Y represent the input and output variables, respectively; for now,

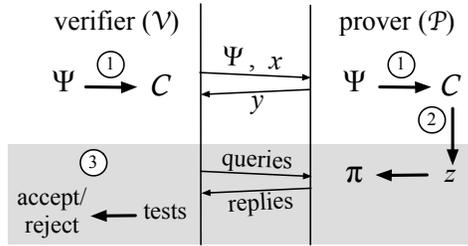


Figure 1—Verifiable outsourcing in Zaatara and Pinocchio, assuming a single instance of a computation Ψ on input x (amortization is depicted in Figure 2). Step ①: \mathcal{V} and \mathcal{P} compile Ψ from a high-level language to constraints C . Step ②: \mathcal{P} produces a satisfying assignment, z , to $C(X=x, Y=y)$. Step ③: \mathcal{P} uses complexity-theoretic and cryptographic machinery to convince \mathcal{V} that \mathcal{P} holds a satisfying assignment.

we assume one of each. Upper-case letters (X, Y, Z, \dots) represent constraint variables; their lower-case counterparts (x, y, z, \dots) represent concrete values taken by (or assigned to, or bound to) those variables.

Also, let $C(X=x)$ mean C with X bound to x (\mathcal{V} 's requested input); $C(X=x, Y=y)$ indicates that in addition Y is bound to y (the purported output). Notice that $C(X=x, Y=y)$ is a set of constraints over the variables Z . If for some z , setting $Z=z$ makes all constraints in $C(X=x, Y=y)$ hold simultaneously, then $C(X=x, Y=y)$ is said to be *satisfiable*, and z is a *satisfying assignment*.

For a given computation Ψ , a set of constraints C is said to be *equivalent* to Ψ if: for all x, y , we have $y = \Psi(x)$ if and only if $C(X=x, Y=y)$ is satisfiable. As a simple example, the constraints $C = \{Z - X = 0, Z + 1 - Y = 0\}$ are equivalent to add-1 [20]. Indeed, consider a pair (x, y) . If $y = x + 1$, then there is a satisfying assignment to $C(X=x, Y=y)$, namely $Z=x$. However, if $y \neq x + 1$, then $C(X=x, Y=y)$ is not satisfiable.

(2) \mathcal{P} computes and identifies a satisfying assignment.

\mathcal{P} “executes” $\Psi(x)$ by identifying a satisfying assignment to the equivalent constraints $C(X=x)$, and obtaining the output y in the process. To do so, \mathcal{P} runs a constraint-solving routine that takes as input a compiler-produced list of annotated constraints. This routine goes constraint-by-constraint. A common case is that a constraint introduces a variable and can be written as an assignment to that new variable (e.g., $\{\dots, Z_4 = Z_3 \cdot (Z_2 + Z_1), Z_5 = Z_4 \cdot Z_2, \dots\}$); the routine “solves” such constraints by evaluating their right-hand sides.

Some constraints require additional work of \mathcal{P} . An example is the $!=$ test (this will give some intuition for the techniques in Section 3). Consider the following snippet:

```

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

```

This compiles to the following constraints [20]:

$$C_{!=} = \left\{ \begin{array}{l} M \cdot (Z_1 - Z_2) - Z_3 = 0 \\ (1 - Z_3) \cdot (Z_1 - Z_2) = 0 \end{array} \right\}.$$

Notice that the first constraint introduces *two* new variables (M, Z_3), and thus there are multiple ways to satisfy this constraint. To choose values for these variables that also satisfy the second constraint, \mathcal{P} 's constraint-solving routine consults the constraints' annotations. The relevant annotation tells \mathcal{P} that if $Z_1 \neq Z_2$, then \mathcal{P} should set M equal to the multiplicative inverse of $Z_1 - Z_2$, which \mathcal{P} computes outside of the constraint formalism. We call this “computing exogenously” (in theoretical terms, M and Z_3 are “non-deterministic input”), and there is an analogy between the exogenous computation of M and supplying values from storage in Section 3.

(3) \mathcal{P} argues that it has a satisfying assignment.

\mathcal{P} wants to prove to \mathcal{V} that it knows a satisfying assignment to $C(X=x, Y=y)$; this would convince \mathcal{V} that the output y is correct (and moreover that the computation, expressed in constraints, was executed correctly). Of course, there is a simple proof that a satisfying assignment exists: the satisfying assignment itself. However, \mathcal{V} could check this proof only by examining all of it, which would be as much work as executing the computation.

Instead, Zaatara and Pinocchio apply the theory of PCPs [6, 7],² which implies that a classical proof—a satisfying assignment z , in this case—can be *encoded* into a long string π in a way that allows \mathcal{V} to detect the proof's validity by (a) inspecting a small number of randomly-chosen locations in π , and (b) applying efficient tests to the contents found at those locations. The details—what is in the encoding π , how \mathcal{V} selects locations to inspect, what tests \mathcal{V} applies, and why all of this works—are beyond the scope of this paper.

The protocols do not use PCPs alone: the encoded proof π is far larger than the number of steps in Ψ , so making \mathcal{V} receive π would again defeat our purpose. To get around this issue, Zaatara and Pinocchio—and their theoretical progenitors—compose PCPs with cryptography, based on assumptions that \mathcal{P} cannot break certain primitives. There are two types of protocols; our compiler produces \mathcal{V} and \mathcal{P} binaries for both.

First, Zaatara [65] instantiates an *efficient argument* [19, 42, 45, 66, 67]: \mathcal{V} extracts from \mathcal{P} a cryptographic *commitment* to π , and then \mathcal{V} *queries* \mathcal{P} , meaning that \mathcal{V} asks \mathcal{P} what values π contains at particular locations. \mathcal{V} uses PCPs to choose the locations and test the replies, and cryptography to ensure that \mathcal{P} 's replies pass \mathcal{V} 's tests only if \mathcal{P} 's replies are consistent with a proof π that a sat-

²Our description takes some expositional license: Pinocchio's explicit base is GGPR [32], which does not invoke PCPs. However, one can regard the *key* in their work as PCP queries, in encrypted form [16].

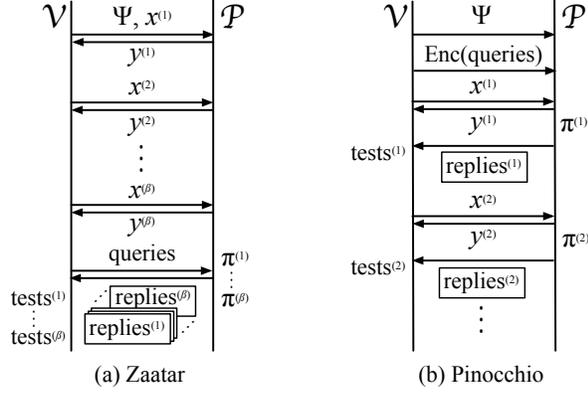


Figure 2—Amortization in Zaatat [65] and Pinocchio [59]. Superscripts denote different instances. In Zaatat, \mathcal{V} 's work to formulate queries amortizes over a batch of β instances; in Pinocchio, analogous work amortizes over all future instances of the same computation (this is better). In both protocols, the $\Psi \rightarrow \mathcal{C}$ step happens only once for each Ψ (not depicted).

ifying assignment exists. The protocol details are given in prior works [65, §2][67, §2][66].

The second variant is instantiated by Pinocchio [59] and known as a *non-interactive argument* [32, 33]: \mathcal{V} *preencrypts* queries and sends them to \mathcal{P} . As in the first variant, the queries are chosen by PCP machinery and describe locations where \mathcal{V} wants to inspect an eventual π . Here, however, \mathcal{P} replies to the queries without knowing which locations \mathcal{V} is querying. This process (hiding the queries, replying to them, testing the answers) relies on sophisticated cryptography layered atop the PCP machinery. The details are described elsewhere [16, 32, 59].

2.3 Amortization, guarantees, and costs

\mathcal{V} incurs a setup cost (to express which locations in π to query) for each computation Ψ and each input size. This cost amortizes differently in Zaatat and Pinocchio.

In Zaatat, amortization happens over a *batch*: a set of β instances of the identical computation Ψ , on different inputs (Figure 2(a)). Thus, Zaatat presumes parallelism: for $j \in \{1, \dots, \beta\}$, \mathcal{V} sends parallel inputs $x^{(j)}$, \mathcal{P} returns parallel outputs $y^{(j)}$, and \mathcal{P} formulates parallel proofs $\pi^{(j)}$ establishing that $y^{(j)} = \Psi(x^{(j)})$. The synchronization requirement is that \mathcal{V} extract commitments to all $\pi^{(j)}$ before issuing the queries (because queries are reused across the batch). Note that \mathcal{P} is an abstraction and could represent multiple machines (as in our MapReduce application in Section 4). Zaatat meets the completeness and soundness properties given earlier (§2.1), with $\epsilon < 1/10^6$ (see [65, Apdx. A.2]), and in addition provides soundness for the batch: if for any $j \in \{1, \dots, \beta\}$, $y^{(j)} \neq \Psi(x^{(j)})$, then $\Pr\{\mathcal{V} \text{ rejects the batch}\} > 1 - \epsilon$.

In Pinocchio, query formulation by \mathcal{V} and installation on \mathcal{P} happen once per Ψ , thereby amortizing over all future instances of the identical computation (Figure 2(b)).

	naive	Zaatat [65], Pinocchio [59]
\mathcal{V} , setup	0	$c_2 \cdot (Z + \mathcal{C})$
\mathcal{V} , runtime	$\beta \cdot (T(x) + c_1 y)$	$\beta \cdot (c_3 + c_4 \cdot (x + y))$
\mathcal{P} , runtime	0	$\beta \cdot (c_5 \cdot (Z + \mathcal{C}) + c_6 \cdot \mathcal{C} \cdot \log \mathcal{C})$

T : running time of computation as a function of input length.

x, y : input and output of computation.

β : number of instances over which \mathcal{V} 's setup cost amortizes
 c_1, c_2, \dots : model costs of processing input/output, cryptographic primitives, PCP queries, etc.

Figure 3—CPU costs of step (3) under Zaatat and Pinocchio, and under the naive approach: reexecute and compare. The amortization behavior is different for Zaatat and Pinocchio (see text). Also, the constants (c_2, c_3, \dots) differ: Pinocchio's c_4 is lower while for the other constants, Zaatat's values are lower. Section 8.1 discusses these constants, the magnitudes of $|Z|$ and $|\mathcal{C}|$, and the costs of step (2).

Pinocchio meets the completeness and soundness properties, with $\epsilon < 1/2^{128}$. Pinocchio also has versions that provide zero-knowledge (the prover can keep private the contents of the satisfying assignment z) and public verifiability [59]; the former provides a crucial foundation for Pantry's privacy-preserving applications (§6).

Figure 3 depicts the protocols' CPU costs for step (3). A key performance goal is that \mathcal{V} *should incur lower (amortized) CPU costs than the naive alternative: reexecuting the computation* [31].³ Performance is thus evaluated as follows [59, 65–67, 74]. (1) Are the per-instance costs for \mathcal{V} less than the running time of Ψ , when Ψ is expressed in C and compiled to machine code? (Otherwise, the performance goal cannot be met.) (2) What is the *cross-over* point, meaning the number of instances past which \mathcal{V} expends less total CPU than the naive verifier? (3) What are the overheads of \mathcal{P} , relative to normal execution?

Rough answers are as follows (see also Section 8). For question (1), the answer is “sometimes; it depends on the computation”. For (2), the cross-over points are tens of thousands or millions [65, §5.2], depending on the computation. For (3), the overheads are very high: factors of 10^4 or 10^5 are not uncommon.

To briefly compare the performance of Zaatat and Pinocchio, Pinocchio has superior amortization behavior (see above) but higher proving and setup costs (and hence higher cross-over points), by constant factors.

2.4 Expressiveness

As context for Pantry, we now describe the language features and limitations of prior work [20, 59, 65, 67].

Pre-Pantry, compilers accepted a C subset [59] (or the equivalent [50, 65, 67]) that includes functions, structs,

³One might think to compare to replicated execution (§1), but a goal of verifiable computation is to provide very strong guarantees (§2.1); replication stops working when faults are correlated.

typedefs, preprocessor definitions, if-else statements, explicit type conversion, and standard integer and bitwise operations. These compilers partially support pointers and loops: pointers and array indexes must be compile-time constants (ruling out a RAM abstraction), and likewise with the maximum number of loop iterations.

When compiled, most operations introduce only a few new variables or constraints [65, §4]. There are four exceptions. The first two are inequalities and bitwise operations; these constructs separate numbers into their bits and glue them back together [20, 59, 67], requiring $\approx \log_2 |\mathbb{F}|$ constraints and variables per operation. The other two are looping and if-else statements: loops are unrolled at compile time, and the costs of an if-else statement combine the costs of the then-block and the else-block [20].

Apart from the specifics of language constructs and costs, the pre-Pantry model of computation is severely limited, even hermetic: computations can interact with state neither as auxiliary input, nor during execution, nor as auxiliary output. Therefore, using Zaatar or Pinocchio requires \mathcal{V} to supply all inputs, receive all outputs, and eschew any notion of RAM, disk, or storage. These are the limitations addressed by Pantry.

3 Storage model and primitives in Pantry

The core of Pantry is two primitives, verifiable PutBlock and GetBlock, that extend the model above. This section describes the primitives; Sections 4–6 describe their use.

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 attempt at incorporating state into verifiable computation would be to represent load and store operations with constraints explicitly. However, doing so naively would incur horrific expense: if memory is an array of variables, then load(addr) would require a separate constraint for each possible value of addr (assuming addr is not resolvable at compile-time). This approach would also require the input state to be available to the verifier \mathcal{V} .

To overcome these problems, we want a model in which computations do not execute storage but can efficiently verify it. Given such a model, we could use constraints to represent computation (as we do now) as well as efficient *checks* of storage. But such a model is actually well-studied, in the context of untrusted storage: the state is represented by hash trees [17, 53], often accompanied by a naming scheme in which data blocks are referenced by hashes of their contents [29, 47].

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 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

$$S: \text{name} \rightarrow \text{block} \cup \perp,$$

where if $\text{block} = S(\text{name})$, then $H(\text{block}) = \text{name}$. In other words, S implements the relation H^{-1} . This naming scheme allows clients to use untrusted storage servers [29, 47]. The technique’s power is that given a name for data, 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 can compute their names and use those names as references later in the computation.

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 storage checks to \mathcal{P} ; in fact, \mathcal{P} itself could be using an untrusted block store!

We will show in later sections how to write general-purpose computations; 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 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

Pantry provides two primitives to the programmer:

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

These primitives are detailed in Figure 4. Notice that in a correct execution, $H(\text{block}) = \text{name}$. Given this relation, and given the collision-resistance of H , the programmer receives from GetBlock and PutBlock a particular storage model: S functions as write-once memory, where the addresses are in practice unique, and where an address certifies the data that it holds.

```

GetBlock (name  $n$ ):
   $block \leftarrow$  read block with name  $n$  in block store  $S$ 
  assert  $n == H(block)$ 
  return  $block$ 

PutBlock ( $block$ ):
   $n \leftarrow H(block)$ 
  store  $(n, block)$  in block store  $S$ 
  return  $n$ 

```

Figure 4—Pseudocode for verifiable storage primitives. These primitives compile to constraints that enforce the required relation between n and $block$; the constraints do not represent interactions with S explicitly.

Of course, how S is implemented is unspecified here; the choice can be different for different kinds of storage (MapReduce, RAM, etc.). And, per the definition of S , block length can vary; for example, in the MapReduce application (§4), an entire file will be one block.

To bootstrap, the client supplies one or more names as input, and it may receive one or more names as output, for use in further computations. These names are related to capabilities [40, 46]: 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.

We now describe the constraints that enforce the model. The code $b = \text{GetBlock}(n)$ 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 code $n = \text{PutBlock}(b)$ 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, \mathcal{C}_H and $\mathcal{C}_{H^{-1}}$ will usually appear inside a larger set of constraints, in which case the compiler relabels the inputs and outputs of \mathcal{C}_H and $\mathcal{C}_{H^{-1}}$ to correspond to intermediate program variables. As an 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 assign $B=b$ in a way that satisfies the constraints, \mathcal{P} must identify a concrete b , presumably from storage, such that $H(b)=x_2$.

Costs. The main cost of GetBlock and PutBlock is the set of constraints required to represent the hash function H in \mathcal{C}_H and $\mathcal{C}_{H^{-1}}$. Unfortunately, widely-used functions (e.g., SHA-1) make heavy use of bitwise operations, which do not have compact representations as constraints (§2.4). Instead, we use an *algebraic* hash function, due to Ajtai [4, 36] and based on the hardness of approximation problems in lattices. The Ajtai function multiplies its input, represented as a bit vector, by a large matrix modulo an integer. This matrix-vector multiplication can be expressed concisely in constraints because constraints naturally encode sums of products (§2.2). Indeed, Ajtai requires approximately ten times fewer constraints than SHA-1 would. Nevertheless, Ajtai uses some bitwise operations (for modular arithmetic) and hence requires a substantial number of constraints (§8.1).

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 correct 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 discard blocks inside PutBlock yet still exhibit a satisfying assignment. Such a \mathcal{P} might be caught only when executing a subsequent computation (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 [68]. Also, Pantry (like its predecessors) does not enforce *availability*: \mathcal{P} could refuse to engage, or fail to supply a satisfying assignment, even if it knows one.

What Pantry enforces 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 this reason, if \mathcal{V} 's computation 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 4).

A limitation of this model is that \mathcal{P} cannot prove to \mathcal{V} that \mathcal{V} made such an error; to the argument step (step (3) in §2.2), this case looks like the one in which \mathcal{P} refuses to provide a satisfying assignment. While that might be disconcerting, Pantry's goal is to establish that a remote execution is consistent with an expressed computation; program verification is a complementary concern (§1).

4 Verifiable MapReduce

This section describes how Pantry provides verifiability for MapReduce jobs. We begin with a brief review of the standard MapReduce model [26].

A MapReduce *job* consists of Map and Reduce functions, and input data structured as a list of key-value pairs; the output is a transformed list of key-value pairs. The programmer supplies the implementations of Map and Reduce; Map takes as input a list of key-value pairs and outputs another list of key-value pairs, and Reduce takes as input a list of values associated with a single key and outputs another list of values. The *framework* runs multiple instances of Map and Reduce as stand-alone processes, called *mappers* and *reducers*. The framework gives each mapper a chunk of the input data, *shuffles* the mappers' output, and supplies it to the reducers; each reducer's output contributes a chunk to the overall output of the job. A centralized module, which is part of the framework, drives the job (by assigning processes to machines, etc.).

Overview of MapReduce in Pantry. The verifier \mathcal{V} is a machine that invokes a MapReduce job (for instance, the desktop machine of a cloud customer). The goal of Pantry's MapReduce is to assure \mathcal{V} that its job starts from the correct input data and executes correctly from there.

The model here will be similar to the standard one outlined above, except that the input and output files will be verifiable blocks (§3): a file will be referenced by a collision-resistant hash, or *digest*, of its contents (from now on, we use “digest” and “name” interchangeably). In this model, invoking a MapReduce job requires \mathcal{V} to supply a list of digests, one for each input file; call this list x . Likewise, \mathcal{V} receives as output a list of digests, y . \mathcal{V} learns of the digests in x either from a bootstrapping step

(creating the data and keeping track of its digest, say) or as the output of a job; likewise, \mathcal{V} can use the digests in y either to download (and verify the integrity of) the actual data or to feed another job. That is, these digests are self-certifying references to the data [29, 47].

Given this model, \mathcal{V} will be guaranteed that the output digests y are correct, meaning that the actual input data (the key-value pairs whose digests are x), when transformed by \mathcal{V} 's desired Map and Reduce functions, results in output data with digests y . But providing this guarantee requires an application of the verification machinery (§2–§3), which raises a design question: what exactly is the computation to be verified, and which machine(s) implement \mathcal{P} ?

Pantry's approach is as follows (we discuss the rationale later). The verifier regards the MapReduce job as two separate batch computations (§2.3), one for the map phase and one for the reduce phase. In these computations, each mapper and reducer is an instance, with a prover. In our design, \mathcal{V} handles an intermediate digest for every (mapper, reducer) pair.

Mechanics. Pantry's MapReduce framework wraps Map and Reduce into functions `Mapper` and `Reducer`, which are depicted in Figure 5; the job is executed by multiple instances of each. For verification, Pantry's C-to-constraint compiler transforms these functions into constraints, and then each instance—playing the role of the prover—convinces \mathcal{V} that it knows a satisfying assignment to the corresponding constraints (§2.2, step (3)). Execution and verification can be decoupled, but under Zaatar, the complete execution of a phase (map or reduce) must happen before verification of that phase.

We now give more detail, beginning with some notation. Let M and R be the number of mappers and reducers, and $\mathcal{C}_{\text{Mapper}}$ and $\mathcal{C}_{\text{Reducer}}$ the constraint representations of `Mapper` and `Reducer`. Also, recall that superscripts denote instances in a batch (§2.3).

When the mappers execute, each instance $j \in \{1, \dots, M\}$ gets as its input, $x^{(j)}$, the digest of some data. The output of an instance, $map_out^{(j)}$, is a vector of R digests, one for each reducer that this mapper is “feeding”; the framework receives this output and forwards it to \mathcal{V} . Verification convinces \mathcal{V} that each mapper j knows a satisfying assignment to $\mathcal{C}_{\text{Mapper}}(X=x^{(j)}, Y=map_out^{(j)})$, which establishes for \mathcal{V} that the mapper worked over the correct data, applied Map correctly, partitioned the transformed data over the reducers correctly, and—in outputting $map_out^{(j)}$ —named the transformed data correctly. Note that $\{map_out^{(j)}\}_{j=\{1, \dots, M\}}$ are the $M \cdot R$ intermediate digests mentioned above.

The framework then supplies the inputs to the second phase, by shuffling the digests $\{map_out^{(j)}\}_{j=\{1, \dots, M\}}$ and regrouping them as $\{reduce_in^{(j)}\}_{j=\{1, \dots, R\}}$, where

```

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 5—For verifiable MapReduce, Pantry regards the depicted functions, Mapper and Reducer, as separate computations. The two functions compile to separate constraints, and \mathcal{V} verifies in two batches: one for the mappers and one for the reducers.

each $reduce_in^{(j)}$ is a vector of M digests, one for each mapper. (\mathcal{V} does this regrouping too, in order to know the reducers’ inputs.)

The framework then invokes the reducers, and the output of each reducer $j \in \{1, \dots, R\}$ is a single digest $y^{(j)}$. Verification convinces \mathcal{V} that each reducer j knows a satisfying assignment to $\mathcal{C}_{\text{Reducer}}(X=reduce_in^{(j)}, Y=y^{(j)})$. This establishes for \mathcal{V} that each reducer worked over the correct M blocks, applied Reduce to them correctly, and produced the correct output digests.

Analysis. Figure 6 compares the costs of the map phase under Pantry’s MapReduce and the naive approach of verifying a job by downloading the inputs (perhaps checking them against digests) and locally executing the computation. A similar analysis applies to the reduce phase.

Both pre-Pantry and under Pantry, the verifier can save CPU cycles compared to the naive verifier provided that the per-instance verification cost is less than the cost to execute the instance. Pre-Pantry, this condition holds only if $c_3 + c_4 \cdot (|x| + |y|) < T(|x|) + c_1|y|$, implying that using the verification machinery makes sense only if the computation is superlinear in its input size (see Figure 3). Under Pantry, however, the analogous condition holds when $c_3 + c_4 \cdot |d| \cdot (R + 1) < T_{\text{mapper}}(|ch|)$, which can hold even when the computation is *linear* in its input. If this condition holds, then the CPU cross-over point (§2.3) occurs when $M \geq \frac{c_2 \cdot (|Z_{\text{mapper}}| + |C_{\text{Mapper}}|)}{T_{\text{mapper}}(|ch|) - c_3 - c_4 \cdot |d| \cdot (R + 1)}$, per Figure 6.

Pantry also saves the verifier network costs. This happens when $M \geq \frac{c_7 \cdot (|Z_{\text{mapper}}| + |C_{\text{Mapper}}|)}{|ch| - c_8 - R \cdot |d|}$. Notice that the floor on M is proportional to the setup costs: the higher the setup costs, the more instances are needed to beat naive verification. Also, the floor moves inversely with $|ch|$: the larger the chunk size, the greater the expense incurred by the naive verifier in downloading the inputs.

We emphasize that this analysis is predicated on a baseline that is favorable to Pantry. If the baseline were instead local execution and local storage (no remote party at all), then Pantry would never save network costs. How-

	naive (local)	Pantry
CPU costs		
\mathcal{V} , setup	0	$c_2 \cdot (Z_{\text{mapper}} + C_{\text{Mapper}})$
\mathcal{V} , runtime	$M \cdot T_{\text{mapper}}(ch)$	$M \cdot (c_3 + c_4 \cdot d \cdot (R + 1))$
network costs		
setup	0	$c_7 \cdot (Z_{\text{mapper}} + C_{\text{Mapper}})$
runtime	$M \cdot ch $	$M \cdot (c_8 + d \cdot (R + 1))$
T_{mapper} : running time of a map instance		M : # of mappers
$ ch $: length of a mapper’s input		$ d $: length of a digest

Figure 6—Verification costs in Pantry’s MapReduce and naive (local) verification, for the map phase; the reduce phase is similar. The CPU costs largely follow Figure 3; the main difference is that \mathcal{V} now handles only a *digest* of the inputs. \mathcal{P} ’s costs are omitted, but the substitutions are similar.

ever, the analyzed baseline corresponds to common uses of the cloud today: MapReduce jobs execute remotely because their inputs *are* remote, so downloading and uploading ought to be recognized as a cost. Another basis for comparison is Zaatara and Pinocchio: their verifiers handle all inputs and outputs, and thus cannot ever save network costs.

Summarizing the analysis, a MapReduce application calls for Pantry if (a) verifiability is needed and (b) the computational cost of the job is high (so there is a CPU cross-over point), there is a lot of data (so there is a network cross-over point), or both.

Rationale and limitations. Our design reflects awkward aspects of the framework. For example, because of the existence of setup costs (§2.3), we chose to have \mathcal{V} handle intermediate digests. In more detail, \mathcal{V} could avoid handling intermediate digests—it could verify the job’s output digests $\{y^{(j)}\}$ directly from the input digests $\{x^{(j)}\}$ —by verifying a single batch. But each instance would have to encompass constraints for one reducer and M mappers, causing setup costs to be, undesirably, proportional to the *aggregate* mappers’ (instead of a single mapper’s) work. To further explain our choice, we

note that quadratic intermediate state is not inherently disastrous: in standard MapReduce, the framework keeps $O(M \cdot R)$ state [26].

Other limitations stem from the constraint model. For example, we eschew a general-purpose partitioning module in the mapper, as it would compile to a large number of constraints, increasing costs. Instead, the programmer must partition the output of Map into R chunks, and must similarly read from M inputs in Reduce—tasks that are hidden in standard MapReduce. Moreover, Map and Reduce face the expressiveness restrictions described earlier (§2.4); one consequence is that each mapper’s chunk size must be identical and fixed at compile time, and likewise with the reducers.

5 Verifiable data structures

This section describes Pantry’s higher-level storage abstractions: RAM, a searchable tree, and a simple database. As with MapReduce, we want to implement the abstractions as data structures in a subset of C, augmented with PutBlock and GetBlock (§3). To do so, we apply the technique of embedding in data blocks the names (or references or hashes—these concepts are equivalent here) of other blocks [17, 29, 47, 49, 53] (see also §9). In the resulting structure, the hashes are links—or pointers that authenticate what they point to. The starting hash (for instance, of the root of a tree) can authenticate any value in the structure; we review how this is done below. We can then incorporate the resulting abstractions into some larger C program, compile that program to constraints, and apply the argument step (§2.2) to those constraints.

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 implementation is in Figure 7.

The high-level idea behind this pseudocode is that the digest commits to the full state of memory [17, 53], 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 d can commit to memory, we briefly review Merkle trees [17, 53]. Every node is named by a collision-resistant hash (denoted H) of its contents. An interior node’s contents are the names (or hashes) of the node’s left and right children. Each leaf node corresponds to a memory address, and contains the value currently held at the memory address. Then, the

```

Load(address  $a$ , digest  $d$ ):
   $\ell \leftarrow \lceil \log N \rceil$ 
   $h \leftarrow d$ 
  for  $i = 1$  to  $\ell$ :
     $node \leftarrow \text{GetBlock}(h)$ 
     $x \leftarrow$   $i$ th bit of  $a$ 
    if  $x = 0$ :
       $h \leftarrow node.left$ 
    else:
       $h \leftarrow node.right$ 
   $node \leftarrow \text{GetBlock}(h)$ 
  return  $node.value$ 

Store(address  $a$ , value  $v$ , digest  $d$ ):
   $path \leftarrow \text{LoadPath}(a, d)$ 
   $\ell \leftarrow \lceil \log N \rceil$ 
   $node \leftarrow path[\ell]$ 
   $node.value \leftarrow v$ 
   $d' \leftarrow \text{PutBlock}(node)$ 
  for  $i = \ell$  to 1:
     $node \leftarrow path[i - 1]$ 
     $x \leftarrow$   $i$ th bit of  $a$ 
    if  $x = 0$ :
       $node.left \leftarrow d'$ 
    else:
       $node.right \leftarrow d'$ 
   $d' \leftarrow \text{PutBlock}(node)$ 
  return  $d'$ 

```

Figure 7—RAM operations use verifiable blocks in a Merkle tree [17, 53]. N is the number of addresses in the memory.

digest d is the hash of the root node’s contents. 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 by exhibiting a *witness-path*: the purported name of a ’s sibling, the purported 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 7 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 witness-path. Thus, using $\mathcal{C}_{\text{Load}}$ to denote the constraints to which Load compiles, $\mathcal{C}_{\text{Load}}(X=(a, d), Y=v)$ 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 7 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 C_{Store} denote the constraints that Store compiles to. To satisfy $C_{\text{Store}}(X=(a, v, d), Y=d')$, \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. We briefly describe the blowup from the constraint representation; Sections 2.2 and 4 show how this blowup feeds into the costs of \mathcal{V} and \mathcal{P} . 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).

5.2 Search tree

We now consider a searchable tree; we wish to support efficient range searches over any keys for which the less-than comparison is defined. Specifically, we wish to support the following API:

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

To implement this interface, a first cut approach would be to use the general-purpose RAM abstraction (§5.1) 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 $O(\log N)$, a search in a balanced tree of m elements would cost $O((\log N) \cdot (\log m))$. Instead, we use an alternative construction, which illustrates a strategy applicable to a wide class of data structures.

To get the per-operation cost down to $O(\log m)$, we build a *searchable* Merkle tree (this is different from the tree in §5.1). Each node in the tree contains a key, one or more values corresponding to that key, and pointers to (that is, hashes of) its children. The nodes are in sorted order, and the tree is a balanced (AVL) tree, so operations take time that is logarithmic in the number of keys stored.

A search operation (FindEquals, FindRange) descends the tree, via a series of GetBlock calls. An update operation (Insert, Remove) first descends the tree 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 searches) and the correct digests (in the case of updates).

5.3 Verifiable database queries

The data structures described above enable us to implement a simple database that supports verifiable queries.

\mathcal{V} specifies queries in a primitive SQL-like language, which supports the following non-transactional queries on single tables: SELECT (the WHERE predicates must refer to a single column), INSERT, UPDATE, DELETE, CREATE, and DROP. \mathcal{V} and \mathcal{P} convert each query into C code that invokes the APIs from Sections 3.2 and 5.2, and is then compiled into constraints.

The database itself has a simple design. Each row of every table is stored as a verifiable block, accessed through GetBlock/PutBlock (§3). These blocks are pointed to by one or more indexes, and there is a separate index for each column that the author of the computation wants to be searchable. Indexes are implemented as verifiable search trees (§5.2), and database queries are converted into a series of calls to the trees’ FindEquals, FindRange, Insert, and Remove operations.

Because this database uses verifiable data structures and the code is compiled into constraints, we get strong integrity guarantees—with little programmer effort beyond implementing the data structures and queries.

5.4 Compromises and limitations

A key compromise is that efficiency sometimes requires not using RAM and instead constructing data structures directly from verifiable pointers (§5.2, §5.3). One consequence is that the implementer of these data structures is directly exposed to the clumsiness of the constraint model (§2.4); for example, if the data structure implementation indexes into a small array at a variable offset, the code must loop through the set of possible indexes.

The constraint model imposes several other limitations. First, because traversal loops have fixed bounds, data structures have a static size (a fixed depth for trees, etc.), regardless of the number of elements that they logically contain. (However, empty cells and nodes need not consume memory or disk.) For similar reasons, the number of results returned by the search API must be fixed at compile time. Third, as every operation on a data structure is compiled into a fixed number of constraints, \mathcal{P} ’s running time to perform the operation is largely determined by the data structure’s static size.

6 Private prover state

Pantry enables applications where the prover’s state is private. For example, the prover holds photographs (e.g., of suspects), the verifier (e.g., a surveillance camera) submits a photograph, and the prover indicates if there is a match. Using Pantry, the client is assured that the response is correct, but no information about the prover’s database leaks (beyond what the output implies).

Pinocchio’s zero-knowledge (ZK) variant [32, 59] provides most of the solution. Here, step (3) of Section 2.2 persuades \mathcal{V} that \mathcal{P} has a satisfying assignment to a set of constraints (as usual), but \mathcal{P} cryptographically hides the actual satisfying assignment. Since the contents of \mathcal{P} ’s state appear in the satisfying assignment (§3), the ZK variant effectively hides \mathcal{P} ’s state—almost. The wrinkle is that, under Pantry as so far described, \mathcal{V} would begin with a cryptographic digest of \mathcal{P} ’s state (§5), and this digest itself leaks information (\mathcal{V} could conceivably guess \mathcal{P} ’s state and use a digest to check the guess).

Thus, we assume that \mathcal{V} begins with a cryptographic *commitment* [35, §4.4.1] to the prover’s state. A commitment binds the prover to its state in a way that permits verifiable queries against that state (as with the previously described digests) but also hides the state. Then, the computation to be verified takes as input a commitment (not a digest), begins by querying for values and checking that they are consistent with the commitment (as with digests), and then uses those values in the rest of the computation. To summarize, the commitment hides the prover’s beginning state from \mathcal{V} , and the ZK machinery hides the prover’s execution.

To realize this approach, we want a commitment primitive that has a reasonably efficient representation in constraints. As a compromise, we instantiate a simple scheme using HMAC-SHA256⁴ [10] (see Appendix C [21] for details). Relative to the protocol of Pedersen [61], our scheme makes a stronger cryptographic assumption but saves an order of magnitude in constraint size.

Applications. We build (§7) and evaluate (§8) several applications of the machinery described above. The first is *face matching*, which implements the example at the start of this section. This example is inspired by previous work [57], but that work provides privacy to both parties and verifiability to neither. The second is *tolling*; the prover is a car, the verifier is a toll collector, and the verifier checks the prover’s claim about what it owes for the billing period. This example is inspired by [62], which requires a custom protocol, while we require only a simple C program (§7). The third application is *regression analysis* (again inspired by prior work that requires a custom protocol [56]); the prover holds a set of patient files, the verifier is an analyst seeking to fit a model to this data, and the computation returns the best-fit parameters. The details of our applications are in Appendix D [21].

7 Implementation details

The Pantry implementation modifies the Ginger-Zaatar compiler [20, 65, 67]. The base compiler first transforms programs written in a high-level language (§2.4) into a list of assignment statements, producing a constraint

or *pseudoconstraint* for each statement. The pseudoconstraints abstract operations that require multiple constraints (inequality comparisons, bitwise operations, etc.). Next, the compiler expands the pseudoconstraints and annotates the results (§2.2). The verifier and prover each consist of computation-independent routines that take a list of annotated constraints as input. \mathcal{P} ’s routines solve the constraints and use the resulting satisfying assignment to respond to queries; \mathcal{V} ’s routine selects queries according to the argument protocol and tests the replies (§2.2).

Pantry adds several conveniences to the base compiler. Following Pinocchio [59], the Pantry compiler accepts a subset of C (§2.4). More significantly, the compiler targets the Pinocchio and the Zaatat encodings, with a unified code base. The main work here was implementing Pinocchio’s pairing-based cryptography, for which we use a public library [2, 14].

To implement GetBlock and PutBlock (§3), Pantry includes new pseudoconstraints, which expand to \mathcal{C}_{H-1} and \mathcal{C}_H , respectively. The associated annotations tell \mathcal{P} how to interact with storage \mathcal{S} (see Figure 4); we implement \mathcal{S} using the LevelDB key-value store [3].

The \mathcal{C}_{H-1} and \mathcal{C}_H constraints implement H as (a variable-length version of) the Ajtai [4, 36] hash function. Using the notation in [36], this function hashes m bits into $n \cdot \log q$ bits. Based on the analysis in [54], we set these parameters as $m=7296$, $n=64$, and $q=2^{19}$ —resulting in a digest of 1216 bits—to achieve at least 180 bits of security. To support variable-length input, we use a prefix-free variant of the Merkle-Damgård transform [44, Ch. 4.6.4] that prepends the input with its length [25].

To implement GetBlock and PutBlock, we added to the compiler pipeline 2200 lines of Java (for parsing Pantry’s subset-of-C), 2100 lines of Go and 360 lines of Python (for expanding pseudoconstraints into constraints), and 300 lines of C++ (in the prover’s constraint solving module). The MapReduce framework (§4) requires 1500 lines of C++. The verifiable data structures (§5.1–§5.2) require 400 lines in Pantry’s subset-of-C. The main component in the database application (§5.3) is a query-to-C translator, which we implement with 2000 lines of Java, on top of Cassandra’s CQL parser [1]. Our private state applications (§6) are 60 lines for face matching, 80 lines for tolling, and 143 lines for regression analysis.

8 Evaluation

Our evaluation answers two questions: (1) What are the overheads for the prover and verifier? and (2) What does the verifier gain from Pantry, versus alternatives? Given Pantry’s goals (§1–§2), these alternatives must be general-purpose and not make restrictive hypotheses about failure classes. This often means comparing to naive verifiers (§2.3). However, we would be the first to admit

⁴Ajtai is unsuitable because it is not a pseudorandom function (PRF).

computation (Ψ)	type	$O(\cdot)$
dot product of two length- m vectors	MapReduce (Z)	m
search m nucleotides for length- d substring	MapReduce (Z)	$m \cdot d$
nearest neigh. search of m length- d vectors	MapReduce (Z)	$m \cdot d$
covariance matrix for m samples of dim. d	MapReduce (Z)	$m \cdot d^2$
SELECT rows from a table with m rows	Database (P)	$\log m$
INSERT a row into a table with m rows	Database (P)	$\log m$
UPDATE a row in a table with m rows	Database (P)	$\log m$
match against m 900-bit face fingerprints	Private state (P)	m
compute toll bill for a maximum of m tolls	Private state (P)	m
fit a linear model to m -many d -dim. records	Private state (P)	$m \cdot d^2 + d^3$

Figure 8—Sample applications in our experiments. The MapReduce applications uses Zaatar (Z); the other two categories use Pinocchio (P). In the MapReduce applications (§4), Map and Reduce are roughly 60 lines, combined. The DB queries are expressed in Pantry’s query framework (§5.3, §7). The private state applications (details and code size) are described in §6 and §7.

that tailored protocols (of the kind cited in the introduction; an example is [70]) or replication are likely to far outperform Pantry.

Applications and setup. We experiment with a set of sample applications, listed in Figure 8. Additional parameters (for the cryptographic primitives in Zaatar and Pinocchio, etc.) are described in Appendix D [21].

Our experiments use a local cluster of machines, each running Linux on an Intel Xeon processor E5 2680 2.7 GHz with 32GB of RAM and a 250GB 7.5K RPM SATA disk; they are connected by a 56 Gb/s InfiniBand network. Additionally, each machine has an access to a 14PB Lustre 2.1.3 parallel file system.

8.1 Overhead and its sources

Pantry’s costs boil down to three sources of overhead:

- T1 The techniques of untrusted storage;
- T2 The constraint representation of computations; and
- T3 The argument step.

Below, we investigate each of these overheads.

We assess the cost of T1 in terms of the *number of constraints and variables* to which Pantry’s primitives compile. (We will focus on the number of constraints, $|\mathcal{C}|$, as the number of variables, $|Z|$, scales linearly in $|\mathcal{C}|$.) We use this metric because constraints are the computational model (and later, we will express actual running times in terms of constraint set size). Each constraint corresponds to a “register operation” (arithmetic, assignment, etc.), which provides an interpretation of our metric.

Figure 9 shows the number of constraints to which GetBlock and PutBlock (§3) compile, varying the size of the block. The cost is ≈ 12 constraints per byte, or 50 constraints per 32-bit word; thus, in this model, reading a number is 50 times more expensive than adding—a ratio superior to the analogous comparison between hard

operation	number of constraints ($ \mathcal{C} $)
GetBlock or PutBlock; 1KB blocks	13,000
GetBlock or PutBlock; 4KB blocks	47,000
GetBlock or PutBlock; 16KB blocks	180,000
Load (Store); 2^{20} memory cells	93,000 (190,000)
Load (Store); 2^{30} memory cells	140,000 (280,000)

Figure 9—Cost of Pantry’s storage primitives, in constraints (to the nearest 1000), for varying block size or memory size; the number of variables ($|Z|$) is similar (not shown). PutBlock is the same as GetBlock (§3.2). Store is shown in the same row as Load, and is twice as expensive (§5.1); the memory cell size here is 64 bits, and the intermediate Merkle nodes are 2432 bits. The costs scale linearly (in the block size) for GetBlock and logarithmically (in the memory size) for Load and Store.

disks and a CPU’s register operations.⁵ On the other hand, disks benefit from sequential access whereas the costs of GetBlock and PutBlock scale linearly. Moreover, constraints will translate into active CPU costs (as we will cover below), whereas real disks leverage DMA.

The preceding discussion presumes that each data item has its own name, or hash. If instead we want to give the programmer contiguously addressable random access memory (e.g., for a program’s heap), we must use the RAM abstraction (§5.1). Unfortunately, as shown in Figure 9, a verifiable Load costs 93,000 constraints to read 64 bits of memory; the ratio here is *not* close to the analogous memory-vs-register comparison. Thus, GetBlock and PutBlock are best used to implement data structures built directly from verifiable blocks (§5.2–§5.3); as indicated above, the costs are manageable if the programmer interacts with them as if they lived on disk.

Even so, storage constraints contribute heavily to the total constraint set size in our applications; the weight is clear from the two columns labeled $|\mathcal{C}|$ in Figure 10, which displays many of Pantry’s costs for our sample experiments.

This brings us to the next source of overhead: the fact that there *are* constraints (T2). Indeed, the costs of step (2) are due to the constraint representation. The final source of overhead is the argument step (T3), which—*together with T2*—determines the cost of step (3). We consider steps (2) and (3) in turn.

Constraint solving (step (2), §2.2) is a cost for \mathcal{P} . We compute the ratio of solving time (Figure 10, the “solve” column) to $|\mathcal{C}|$ for each of our sample applications. This ratio ranges from 20 to 160 μ s per constraint,⁶ where tolling has the smallest ratio and UPDATE query has the largest. The computations with the largest ratios are those with the highest proportion of GetBlock and PutBlock calls: “solving” these requires computing the Ajtai function (§3.2), which invokes many large integer arithmetic

⁵Of course, \mathcal{P} (not \mathcal{V}) also has to pay for actual execution (in step (2)).

⁶These costs are higher than necessary. Our implementation of \mathcal{P} ’s constraint-solving routine is decidedly unoptimized.

computation (Ψ)	input size	baseline	\mathcal{C} (millions)		prover (\mathcal{P})			③ verifier (\mathcal{V})	
			storage	total	② solve	③ argue	total	setup	per-instance
dot product	$m=20k$	10 ms	1.7	1.8	4.5 min	8.2 min	13 min	5.4 min	380 μs
nucleotide substr. search	$m=600k, d=4$	13 ms	1.6	4.0	4.4 min	18 min	23 min	9.9 min	390 μs
nearest neigh. search	$m=20k, d=10$	5.6 ms	0.9	1.1	2.5 min	7 min	9.5 min	4 min	380 μs
covariance matrix	$m=2.5k, d=10$	3.8 ms	0.6	0.8	1.4 min	4 min	5.4 min	2.3 min	380 μs
SELECT query	$m=2^{27}$ rows	90 μs	1.0	1.3	2.5 min	17 min	20 min	18 min	6.9 ms
INSERT query	$m=2^{20}$ rows	89 μs	2.0	2.4	6.3 min	31 min	37 min	34 min	13 ms
UPDATE query	$m=2^{20}$ rows	64 μs	2.0	2.4	6.4 min	31 min	37 min	34 min	14 ms
face matching	$m=128$	100 μs	0.2	0.7*	27 s	7.8 min	8.2 min	6.5 min	7.2 ms
tolling	$m=512$	6.7 μs	0.1	0.5*	9.8 s	7.1 min	7.3 min	5.2 min	6.2 ms
regression analysis	$m=1024, d=8$	30 μs	0.4	0.7*	50 s	8.2 min	9.1 min	7.7 min	6.2 ms

*Includes 250k constraints for commitment (§6)

Figure 10—Overheads in our sample applications at sample input sizes; for the four MapReduce applications, only the map phase is included. The input size represents a single instance. The baseline column represents the execution of a normally compiled C program. For MapReduce, the baseline is the naive verifier (§4), including a SHA-256 digest check for data integrity (§4); for the database queries, the baseline is a MySQL query; and for the private state apps, the baseline is normal execution (no verifiability). The quantity $|Z|$ is not depicted but is roughly the same as $|C|$ for each sample application. The remaining columns depict the running times (for a single instance; no amortization) of steps (2) and (3), as defined in §2.2; circled numbers refer to these steps.

operations. (Another source of overhead here is that GetBlock / PutBlock operations incur I/O costs associated with accessing the block store.)

Arguing (step (3), §2.2) induces costs for \mathcal{P} and \mathcal{V} , which are depicted for our measured applications in Figure 10 (the columns labeled ③). These costs are largely determined by $|C|$ and $|Z|$, as indicated by the models given earlier (Figures 3 and 6). In these models, the largest constants are c_2, c_3, c_5 (representing cryptographic operations), and are on the order of $100\mu s$. Note that these models are chosen for simplicity; their predictions are within a factor of two of empirical results. The primary sources of variation are the structure of the constraints (treated in prior work [65, §4]) and the relative number of bitwise constraints (small values reduce the costs of some of the cryptographic steps). A model that is more faithful (but more involved) is in Appendix E [21], which also quantifies the constants $\{c_i\}$.

The aforementioned costs can be understood by comparing to the cost of simply executing the computation (Figure 10, the “baseline” column). Both \mathcal{V} ’s setup work and \mathcal{P} ’s runtime work are orders of magnitude more than this baseline, in our sample applications. On top of these costs, the largest experiments (e.g., nucleotide substring search with $m=600k, d=4$) use roughly 75% of the available RAM in our machines (in the setup phase for \mathcal{V} and per-instance for \mathcal{P}).

8.2 All is not lost

Amidst the many appalling overheads in Figure 10, there is actually some encouraging news: the *per-instance* CPU costs for \mathcal{V} are sometimes less than local execution (compare the “per-instance” and “baseline” columns). And though it is not depicted, an analogous thing happens for network costs. Given enough instances, then, the Pantry

verifier could save resources relative to the naive verifier (§2.3). We investigate these and other benefits by taking a closer look at some of our sample applications.

MapReduce. For the MapReduce examples, we want to determine the cross-over points (§2.3, §4) for CPU and network. We will focus on the nucleotide substring search example; results for the other applications are similar.

We experiment within the limits of our testbed, and use the resulting data to extrapolate. A *work unit* will be 10 mappers (each with a chunk size of 600k nucleotides, per Figure 10) and one reducer; let N denote the total job size, in number of input nucleotides. We experiment with $N=6$ million (one work unit, 10 machines), $N=60$ million (ten work units, 100 machines), and $N=1.2$ billion (200 work units, 250 machines, each machine executing multiple workers sequentially). Across these (and smaller-scale) experiments, we observe little variation (std. deviations are within 10% of means, scaling is linear, etc.).

Figure 11 reports the extrapolated resource costs for \mathcal{V} ; the CPU (resp., network) cross-over point is 29 billion nucleotides, or 48,340 mappers (resp., 24 billion nucleotides, or 40,000 mappers). While the chunk size is tiny—reflecting overheads (§8.1)—the results are nevertheless encouraging. First, the baseline is stiff competition: it is linear-time, it runs as optimized machine code, and it uses SHA-256 (not Ajtai) for data integrity. Second, Pantry’s \mathcal{V} beats this baseline at a job size that is plausible: the human genome is roughly 3 billion nucleotides, so the cross-over point is ≈ 10 such genomes.

DB queries. This class of applications has an additional overhead: storage at the prover, for the hash trees (§5.2). Below, we assess that cost, and ask about Pantry’s ability to save resources for \mathcal{V} . What should the baseline be? In Figure 10, we present the running time of MySQL, which helps us gauge the prover’s overhead. However, for

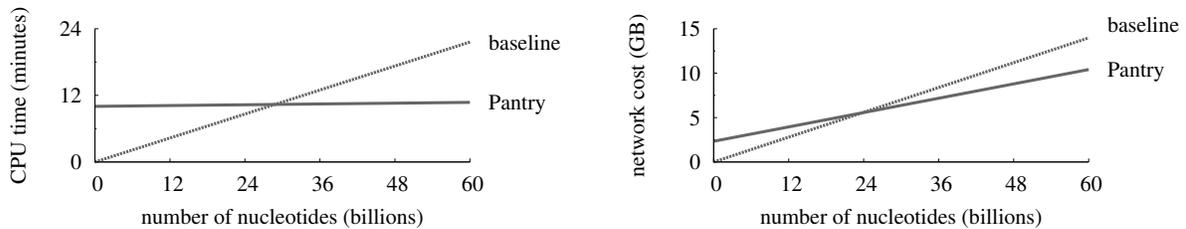


Figure 11—The verifier’s CPU and network costs (extrapolated) as a function of job size for the nucleotide substring application in Figures 8 and 10 (each mapper gets a chunk of 600k nucleotides; one reducer is allocated per ten mappers). All y-intercepts (fixed costs) and slopes (per-instance costs) are empirically determined, based on experiments that exhibit the depicted scaling with hundreds of machines. In the CPU (resp., network) graph, Pantry’s y-intercept is roughly ten minutes (resp., 2.3 GB); meanwhile, the baseline’s slope is tens of milliseconds per chunk (resp., 146.5 KB per chunk). Thus, 40,000–50,000 chunks are required for \mathcal{V} to break even, corresponding to 24–30 billion nucleotides.

	Pantry	block store (est.)
network costs		
setup, kept as storage (argue step)	430 MB	0 MB
per-instance (argue step)	288 bytes	8.3 KB
per-instance (input, output)	624 bytes	620 bytes
storage costs		
data	11.5 GB	11.5 GB
metadata (for hash tree)	262 GB	≥ 53.5 GB

Figure 12—Resource costs of a SELECT query, under Pantry and estimates for an alternative based on an untrusted block store. The table has 2^{27} rows, each holding 92 bytes in 12 columns; the query allows 5 matching rows (§5.3, §5.4).

a naive verifier to benefit from MySQL’s optimized query execution *while achieving verifiability*, it would have to download the entire database and execute the query itself.

Instead, our baseline will be reasonably network-efficient and avoid two sources of overhead in Pantry: constraints and the argument step. We assume a server that implements a hash-based block store [29, 47] (akin to the map S in §3.1) and a verifier that runs the computation *natively*; where the program calls GetBlock and PutBlock, the verifier issues an RPC to the server. Since the computation is run natively rather than in constraints, H is SHA-256 (§3.2). We have not yet built this alternative, so we estimate its network costs; we can do this since queries are highly constrained (§2.4, §5.4).

Figure 12 depicts the comparison, for a SELECT query. This table indicates, first, that our implementation needs some work: the metadata is far larger than the data (for both Pantry and the alternative) due in part to unoptimized parameter choices (number of indexes, branching factor, etc.). Second, the effect of the size of Ajtai digests (versus SHA-256) is apparent in the metadata row. Nevertheless, despite these limitations, the Pantry verifier can amortize its network costs in the setup phase (because it does not incur the network cost of handling the verifiable blocks themselves); for this computation, the network cross-over point is 55,000 instances.

Private state. For these applications, we do not ask about cross-over points because \mathcal{V} cannot naively re-

execute the computation. Instead, we just report the costs, for our sample application of tolling; costs for the others are similar. The CPU costs are in Figure 10; the storage and network resources are given below:

private state	5 KB
network (setup) and storage (ongoing)	170 MB
network (per-instance), for inputs/outputs	1 KB
network (per-instance), for argument step	288 bytes

The storage overhead here is proportional to the size of the private state; the reason is as follows. The storage overhead reflects setup costs (see above), setup costs are proportional to $|C|$ and $|Z|$ (see Figures 3 and 6), $|C|$ and $|Z|$ include terms for GetBlock’s input (§3.2), and GetBlock’s input is all of the state because there is no hash tree structure (§5). Although the constant of proportionality is high (due to the argument step), the absolute quantities are not necessarily alarming: the tolling application does not involve much state, and an overhead of several hundred megabytes could fit comfortably on a mobile phone. Moreover, the per-instance network costs are very low, owing to Pinocchio’s machinery (§2.2–§2.3).

9 Related work

Although verifiable computation has a decades-long history (see [59, 66, 74] for surveys), only recently have systems emerged that are both (a) general-purpose (i.e., not targeted to a class of functionality) and (b) rooted in powerful complexity theory and cryptography.

One line of work [24, 71, 72] refines the Muggles interactive proof protocol [37], which is purely complexity-theoretic (no cryptography). As a consequence, the resulting systems are very efficient for the verifier and prover. However, they are restricted to straight-line computations (though this limitation has been partially relaxed [74]).

Another line of work [64–67, 74] refines an efficient argument protocol (§2) due to Ishai et al. [42]. Zaatar [65] is the best-performing entry in this line; it leverages the remarkable encoding of GGPR [32] and handles general side-effect free computations [67].

Pinocchio [59] applies both GGPR’s encoding and its cryptographic constructions [32], and is the first implementation of a general-purpose non-interactive argument (it is a SNARG [33] and a SNARK [15]). It uses essentially the same computational model as Zaatat [65, 67], and for systems working within this model, Pinocchio and Zaatat have the best performance in the literature (on different axes). The two are compared in §2.2 and §2.3.

None of these three efforts handles computations over state. Pantry’s principal contribution is to extend the computational model of Pinocchio and Zaatat to do so, using ideas from untrusted storage. First, Pantry relies on Merkle trees [53] to authenticate a large untrusted memory, an idea used in theory [17] and in practice (for smart-cards [30], databases [27, 49, 52], file systems [34, 43], etc.). Second, Pantry names data blocks by their digests, and treats the digests as references for the purposes of building data structures (including Merkle trees); this idiom is due to the SFSRO [29] file system and used elsewhere (e.g., SUNDR [47]). One (rough) way to understand Pantry is that it verifiably outsources an SFSRO or SUNDR *client*. Of course, Pantry’s general approach is known [11, 15, 32, 41]. However, Pantry is the first realization of this strategy.

A fourth project, appearing in parallel with Pantry, offers a different approach to state. BCGTV [12] use a promising circuit representation from [11] (a different instantiation of steps (1) and (2) in §2.2).⁷ Using insights from [16, 32, 65], BCGTV combine their representation with a step (3) that is much like Pinocchio’s (like Pinocchio, BCGTV is a “SNARK with pre-preprocessing”). On the one hand, BCGTV achieve expressivity relative to Pantry, specifically data-dependent loops. On the other hand, they do not (at present) work with remote state (§4–§6). Furthermore, although a complete evaluation has not been done, their preliminary reported results indicate that performance is often orders of magnitude worse than Pantry. A detailed comparison is future work.

10 Discussion, limitations, and conclusion

Pantry has many limitations. A number of these stem from the clumsiness of the constraint model (§2.4), which led to various compromises described earlier (§4, §5.4, §6). A further compromise is the assumption throughout that the verifier knows the digest of the remote state; this holds when the state is read-only or when there is one client. Future work is to handle multiple writers, perhaps by outsourcing *signature* (not just hash) checks.

But the biggest limitation by far is costs—which are currently so high for the prover and the verifier’s setup

phase (§8.1) that they limit our experiments (§8.2) to scales smaller than those of real applications (to put it mildly). This issue afflicts the entire research area (§9). Indeed, key challenges are to reduce the overhead of the argument protocol (which seems possible, as the costs stem from high constants, not unfavorable asymptotics); reduce the overhead of memory operations within the constraint model (evidence exists that this can be done [11]); and go beyond, or around, the constraint model.

Nevertheless, Pantry dramatically expands the set of scenarios where verifiable computation makes sense. First, Pantry extends verifiability to computations that make indirect memory accesses (to RAM, disk, etc.). Second, because the verifier can supply digests of inputs, the per-instance CPU cost of verification can drop below the time cost to handle the actual inputs, thereby allowing the verifier to beat naive verification even when outsourcing *linear*-time computations (§4, §8.2). Third, Pantry can save network costs for the verifier versus the naive alternative (§4, §8.2). Thus, Pantry may be beneficial even if verification costs more CPU cycles than local execution—a case that defeats the goals (§2.3) of prior work [59, 65–67, 74]. Fourth, Pantry (with a major assist from Pinocchio) extends verifiability to a class of computations involving *private* remote state (§6).

The preceding paragraph describes when Pantry *could* be applicable, but we must also consider when it actually *is*. The answer depends on computation-specific factors: the cross-over points, one’s tolerance for prover overhead, and the details of the scenario. But data-parallel cloud computing (e.g., MapReduce) seems to fit the requirements: many instances of the same computation and an abundance of server CPU cycles. Moreover, a high price for the private state applications might be acceptable, since there is no naive alternative (§8.2).

In conclusion, there is a great deal of work remaining to bring verifiable computation to practice, but Pantry is a significant step toward that goal.

Acknowledgments

We first learned of the folklore approach to verifying computations with state from motivating comments by Yuval Ishai and an anonymous NDSS 2012 reviewer. Suggestions by Dan Boneh, Bryan Parno, Chris Peikert, and Shabsi Walfish substantially strengthened this work. We thank Chris and Shabsi for patient explanations. Feedback and comments from Sebastian Angel, Allen Clement, Josh Leners, David Mazières, Bryan Parno, Riad Wahby, Brent Waters, Edmond L. Wong, George Candea (our shepherd), and the anonymous reviewers improved this draft. The Texas Advanced Computing Center (TACC) at UT supplied computing resources. This work was supported by AFOSR grant FA9550-10-1-0073; NSF grants 1040672, 1055057, and 1040083; a Sloan Fellowship; and an Intel Early Career Faculty Award.

For Pantry’s source code: <http://cs.utexas.edu/pepper>

⁷Recent work takes a different approach to efficient circuit representations of various standard data structures [76]. Incorporating into our system and comparing to Pantry and BCGTV is work in progress.

References

- [1] Cassandra CQL. <http://cassandra.apache.org/doc/cql/CQL.html>.
- [2] High-speed software implementation of the optimal Ate pairing over Barreto-Naehrig curves. <https://github.com/herumi/ate-pairing>.
- [3] leveldb – a fast and lightweight key/value database library by Google. <https://code.google.com/p/leveldb/>.
- [4] M. Ajtai. Generating hard instances of lattice problems. In *ACM Symposium on the Theory of Computing (STOC)*, pages 99–108, May 1996.
- [5] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer. SETI@home: An experiment in public-resource computing. *Communications of the ACM (CACM)*, 45(11):56–61, Nov. 2002.
- [6] S. Arora, C. Lund, R. Motwani, M. Sudan, and M. Szegedy. Proof verification and the hardness of approximation problems. *Journal of the ACM*, 45(3):501–555, May 1998.
- [7] S. Arora and S. Safra. Probabilistic checking of proofs: a new characterization of NP. *Journal of the ACM*, 45(1):70–122, Jan. 1998.
- [8] M. J. Atallah and K. B. Frikken. Securely outsourcing linear algebra computations. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, pages 48–59, Apr. 2010.
- [9] L. Babai. Trading group theory for randomness. In *ACM Symposium on the Theory of Computing (STOC)*, pages 421–429, May 1985.
- [10] M. Bellare, R. Canetti, and H. Krawczyk. Keying hash functions for message authentication. In *IACR International Cryptology Conference (CRYPTO)*, pages 1–15, 1996.
- [11] E. Ben-Sasson, A. Chiesa, D. Genkin, and E. Tromer. Fast reductions from RAMs to delegatable succinct constraint satisfaction problems. In *Innovations in Theoretical Computer Science (ITCS)*, pages 401–414, Jan. 2013.
- [12] E. Ben-Sasson, A. Chiesa, D. Genkin, E. Tromer, and M. Virza. SNARKs for C: Verifying program executions succinctly and in zero knowledge. In *IACR International Cryptology Conference (CRYPTO)*, pages 90–108, Aug. 2013.
- [13] S. Benabbas, R. Gennaro, and Y. Vahlis. Verifiable delegation of computation over large datasets. In *IACR International Cryptology Conference (CRYPTO)*, pages 111–131, Aug. 2011.
- [14] J.-L. Beuchat, J. E. G. Diaz, S. Mitsunari, E. Okamoto, F. Rodriguez-Henriquez, and T. Teruya. High-speed software implementation of the optimal Ate pairing over Barreto-Naehrig curves. *Cryptology ePrint Archive*, Report 2010/354, June 2010. <http://eprint.iacr.org/>.
- [15] N. Bitansky, R. Canetti, A. Chiesa, and E. Tromer. From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again. In *Innovations in Theoretical Computer Science (ITCS)*, pages 326–349, Jan. 2012.
- [16] N. Bitansky, A. Chiesa, Y. Ishai, R. Ostrovsky, and O. Paneth. Succinct non-interactive arguments via linear interactive proofs. In *IACR Theory of Cryptography Conference (TCC)*, pages 315–333, Mar. 2013.
- [17] M. Blum, W. Evans, P. Gemmel, S. Kannan, and M. Naor. Checking the correctness of memories. In *Symposium on Foundations of Computer Science (FOCS)*, pages 90–99, Oct. 1991.
- [18] D. Boneh and D. M. Freeman. Homomorphic signatures for polynomial functions. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, pages 149–168, May 2011.
- [19] G. Brassard, D. Chaum, and C. Crépeau. Minimum disclosure proofs of knowledge. *Journal of Computer and System Sciences*, 37(2):156–189, Oct. 1988.
- [20] B. Braun. Compiling computations to constraints for verified computation. UT Austin Honors thesis HR-12-10, Dec. 2012.
- [21] B. Braun, A. J. Feldman, Z. Ren, S. Setty, A. J. Blumberg, and M. Walfish. Verifying computations with state. *Cryptology ePrint Archive*, Report 2013/356, 2013.
- [22] R. Canetti, B. Riva, and G. Rothblum. Practical delegation of computation using multiple servers. In *ACM Conference on Computer and Communications Security (CCS)*, pages 445–454, Oct. 2011.
- [23] M. Castro and B. Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS)*, 20(4):398–461, Nov. 2002.
- [24] G. Cormode, M. Mitzenmacher, and J. Thaler. Practical verified computation with streaming interactive proofs. In *Innovations in Theoretical Computer Science (ITCS)*, pages 90–112, Jan. 2012.
- [25] J.-S. Coron, Y. Dodis, C. Malinaud, and P. Puniya. Merkle-damgård revisited: how to construct a hash function. In *IACR International Cryptology Conference (CRYPTO)*, pages 430–448, Aug. 2005.
- [26] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 107–113, Dec. 2004.
- [27] P. Devanbu, M. Gertz, C. Martel, and S. G. Stubblebine. Authentic third-party data publication. In *Data and Application Security: Development and Directions*, pages 101–112. Springer, 2002.
- [28] D. Fiore and R. Gennaro. Publicly verifiable delegation of large polynomials and matrix computations, with applications. In *ACM Conference on Computer and Communications Security (CCS)*, pages 501–512, May 2012.
- [29] K. Fu, M. F. Kaashoek, and D. Mazières. Fast and secure distributed read-only file system. In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–24, Oct. 2000.
- [30] B. Gassend, G. E. Suh, D. Clarke, M. van Dijk, and S. Devadas. Caches and hash trees for efficient memory integrity verification. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 295–306, Feb. 2003.
- [31] R. Gennaro, C. Gentry, and B. Parno. Non-interactive verifiable computing: Outsourcing computation to untrusted workers. In *IACR International Cryptology Conference (CRYPTO)*, pages 465–482, Aug. 2010.
- [32] R. Gennaro, C. Gentry, B. Parno, and M. Raykova. Quadratic span programs and succinct NIZKs without PCPs. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, pages 626–645, May 2013.
- [33] C. Gentry and D. Wichs. Separating succinct non-interactive arguments from all falsifiable assumptions. In *ACM Symposium on the Theory of Computing (STOC)*, pages 99–108, June 2011.
- [34] E.-J. Goh, H. Shacham, N. Modadugu, and D. Boneh. SiRiUS: securing remote untrusted storage. In *Network and Distributed System Security Symposium (NDSS)*, pages 131–145, Feb. 2003.
- [35] O. Goldreich. *Foundations of Cryptography: II Basic Applications*. Cambridge University Press, 2004.
- [36] O. Goldreich, S. Goldwasser, and S. Halevi. Collision-free

- hashing from lattice problems. *Electronic Colloquium on Computational Complexity (ECCC)*, TR96-042:236–241, 1996.
- [37] S. Goldwasser, Y. T. Kalai, and G. N. Rothblum. Delegating computation: Interactive proofs for muggles. In *ACM Symposium on the Theory of Computing (STOC)*, pages 113–122, May 2008.
- [38] S. Goldwasser, S. Micali, and C. Rackoff. The knowledge complexity of interactive proof systems. *SIAM Journal on Computing*, 18(1):186–208, 1989.
- [39] P. Golle and I. Mironov. Uncheatable distributed computations. In *RSA Conference*, pages 425–440, Apr. 2001.
- [40] N. Hardy. The Confused Deputy: (or why capabilities might have been invented). *ACM SIGOPS Operating Systems Review*, 22(4):36–38, Oct. 1988.
- [41] Y. Ishai. Personal communication, June 2012.
- [42] Y. Ishai, E. Kushilevitz, and R. Ostrovsky. Efficient arguments without short PCPs. In *IEEE Conference on Computational Complexity (CCC)*, pages 278–291, June 2007.
- [43] M. Kallahalla, E. Riedel, R. Swaminathan, Q. Wang, and K. Fu. Plutus: scalable secure file sharing on untrusted storage. In *Conference on File and Storage Technologies (FAST)*, pages 29–42, Mar. 2003.
- [44] J. Katz and Y. Lindell. *Introduction to Modern Cryptography*. Chapman & Hall / CRC Press, 2007.
- [45] J. Kilian. A note on efficient zero-knowledge proofs and arguments (extended abstract). In *ACM Symposium on the Theory of Computing (STOC)*, pages 723–732, May 1992.
- [46] H. M. Levy. *Capability-Based Computer Systems*. Digital Press, 1984.
- [47] J. Li, M. N. Krohn, D. Mazières, and D. Shasha. Secure untrusted data repository (SUNDR). In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 121–136, Dec. 2004.
- [48] C. Lund, L. Fortnow, H. J. Karloff, and N. Nisan. Algebraic methods for interactive proof systems. *Journal of the ACM*, 39(4):859–868, 1992.
- [49] U. Maheshwari, R. Vingralek, and W. Shapiro. How to build a trusted database system on untrusted storage. In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 135–150, Oct. 2000.
- [50] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella. Fairplay—a secure two-party computation system. In *USENIX Security*, pages 287–302, Aug. 2004.
- [51] D. Malkhi and M. Reiter. Byzantine quorum systems. *Distributed Computing*, 11(4):203–213, Oct. 1998.
- [52] C. Martel, G. Nuckolls, P. Devanbu, M. Gertz, A. Kwong, and S. G. Stubblebine. A general model for authenticated data structures. *Algorithmica*, 39(1):21–41, Jan. 2004.
- [53] R. C. Merkle. A digital signature based on a conventional encryption function. In *IACR International Cryptology Conference (CRYPTO)*, pages 369–378, Aug. 1987.
- [54] D. Micciancio and O. Regev. Lattice-based cryptography. In D. J. Bernstein and J. Buchmann, editors, *Post-quantum Cryptography*, pages 147–191. Springer, 2008.
- [55] F. Monrose, P. Wycko, and A. D. Rubin. Distributed execution with remote audit. In *Network and Distributed System Security Symposium (NDSS)*, pages 103–113, Feb. 1999.
- [56] V. Nikolaenko, U. Weinsberg, S. Ioannidis, M. Joye, D. Boneh, and N. Taft. Privacy-preserving ridge regression on hundreds of millions of records. In *IEEE Symposium on Security and Privacy*, pages 334–348, May 2013.
- [57] M. Osadchy, B. Pinkas, A. Jarrous, and B. Moskovich. SCiFi—a system for secure face identification. In *IEEE Symposium on Security and Privacy*, pages 239–254, May 2010.
- [58] C. Papamanthou, E. Shi, and R. Tamassia. Signatures of correct computation. In *IACR Theory of Cryptography Conference (TCC)*, pages 222–242, Mar. 2013.
- [59] B. Parno, C. Gentry, J. Howell, and M. Raykova. Pinocchio: Nearly practical verifiable computation. In *IEEE Symposium on Security and Privacy*, pages 238–252, May 2013.
- [60] B. Parno, J. M. McCune, and A. Perrig. *Bootstrapping Trust in Modern Computers*. Springer, 2011.
- [61] T. P. Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In *IACR International Cryptology Conference (CRYPTO)*, pages 129–140, Aug. 1991.
- [62] R. A. Popa, H. Balakrishnan, and A. Blumberg. VPriv: Protecting privacy in location-based vehicular services. In *USENIX Security*, pages 335–350, Aug. 2009.
- [63] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. van Doorn, and P. Khosla. Pioneer: Verifying integrity and guaranteeing execution of code on legacy platforms. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 1–16, Oct. 2005.
- [64] S. Setty, A. J. Blumberg, and M. Walfish. Toward practical and unconditional verification of remote computations. In *Workshop on Hot Topics in Operating Systems (HotOS)*, May 2011.
- [65] 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 *European Conference on Computer Systems (EuroSys)*, pages 71–84, Apr. 2013.
- [66] S. Setty, R. McPherson, A. J. Blumberg, and M. Walfish. Making argument systems for outsourced computation practical (sometimes). In *Network and Distributed System Security Symposium (NDSS)*, Feb. 2012.
- [67] 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*, pages 253–268, Aug. 2012.
- [68] H. Shacham and B. Waters. Compact proofs of retrievability. In *ASIACRYPT*, pages 90–107, Dec. 2008.
- [69] A. Shamir. IP = PSPACE. *Journal of the ACM*, 39(4):869–877, Oct. 1992.
- [70] R. Sion. Query execution assurance for outsourced databases. In *International Conference on Very Large Databases (VLDB)*, pages 601–612, Aug. 2005.
- [71] J. Thaler. Time-optimal interactive proofs for circuit evaluation. In *IACR International Cryptology Conference (CRYPTO)*, pages 71–89, Aug. 2013.
- [72] J. Thaler, M. Roberts, M. Mitzenmacher, and H. Pfister. Verifiable computation with massively parallel interactive proofs. In *USENIX HotCloud Workshop*, June 2012.
- [73] B. Thompson, S. Haber, W. G. Horne, T. Sander, and D. Yao. Privacy-preserving computation and verification of aggregate queries on outsourced databases. In *Privacy Enhancing Technologies Symposium*, pages 185–201, Aug. 2009.
- [74] V. Vu, S. Setty, A. J. Blumberg, and M. Walfish. A hybrid architecture for interactive verifiable computation. In *IEEE Symposium on Security and Privacy*, pages 223–237, May 2013.
- [75] C. Wang, K. Ren, and J. Wang. Secure and practical outsourcing of linear programming in cloud computing. In *IEEE International Conference on Computer Communications (INFOCOM)*, pages 820–828, Apr. 2011.
- [76] S. Zahur and D. Evans. Circuit structures for improved efficiency of security and privacy tools. In *IEEE Symposium on Security and Privacy*, pages 493–507, May 2013.
- [77] L. Zhou. Personal communication, Oct. 2012.