The Dissertation Committee for Srinath T. V. Setty
certifies that this is the approved version of the following dissertation:

Toward practical argument systems for verifiable computation

Committee:

Lorenzo Alvisi, Supervisor

Michael Walfish, Co-supervisor

Andrew J. Blumberg

Bryan Parno

Vitaly Shmatikov

Brent Waters

Toward practical argument systems for verifiable computation

by

Srinath T. V. Setty, B.E.Info.Technology; M.S.Comp.Sci.

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

December 2014

To my parents

Acknowledgments

It gives me immense pleasure to write this part of my dissertation, because I can finally thank everyone who helped me get this far.

First and foremost, I consider myself extremely fortunate to have been advised by Michael Walfish. I took his secure and distributed storage systems course in the Spring of 2009. It was a real treat to attend that class as it involved reading papers and critiquing them to identify limitations and potential future work. At the end of that class, I was certain that I wanted to work in his research group. Fortunately, soon after that, I began working under his guidance on a problem in untrusted storage systems, which, at some point, took steps toward the work described in this document. Mike had deep influence on every piece of this dissertation and also on the way I think and work. His expectations on the quality of work are extremely high, but he invested significant amounts of his time to teach me skills that let me work toward meeting those expectations. To put my experience in a few words, I cannot imagine my grad school to have gone any better. Thank you Mike, for everything! I am very much indebted.

It was also through Mike that I met Andrew Blumberg. Mike and I collaborated with Andrew on the work described in this document. In addition to our numerous collaborative efforts, Andrew supported my candidacy when I was in the job market earlier this year. Thank you very much, Andrew!

I collaborated with several other students on the work described in this dissertation: Benjamin Braun, Ariel J. Feldman, Richard McPherson, Nikhil Panpalia, Zuocheng Ren, and Victor Vu. I take this opportunity to thank you all! You guys accelerated my graduation!

Several people helped and inspired me throughout my graduate program. Mike Dahlin's class on advanced operating systems got me interested in systems research. I was also lucky to have gotten an opportunity to work with Lorenzo Alvisi and Mike Dahlin when we found that a project that I was part of had synergy with their project. This is an unforgettable experience as I learned a great deal about systems research. In addition, Lorenzo and Mike helped me with internship and job applications. You have my most sincere gratitude!

Bryan Parno, Jon Howell, and John Douceur hosted me as their intern at MSR Redmond during the Summer of 2011. This gave me an opportunity to gain experience in operating systems research; I thoroughly enjoyed both the work and Microsoft's frequent events for interns. Many thanks to Bryan, Jon, and JD, for hosting me! In addition to mentoring me during my internship and serving on my dissertation committee, Bryan wrote letters of recommendation to support my candidacy at Microsoft Research; his advice on my job talk and interviews were extremely helpful.

During my first year at UT, Emmett Witchel supported and allowed me to work on a project called Airavat. This was my first experience working on a research project; I could not have asked for a better way to get started in grad school. Thank you Emmett!

Many people helped me improve my talks over the years. Marcos Aguilera, Lorenzo Alvisi, Sebastian Angel, Allen Clement, Mike Dahlin, Owen Hofmann, Jon Howell, Joshua Leners, Jinyang Li, Brad Karp, Bryan Parno, Vitaly Shmatikov, Lakshmi Subramanian, Brent Waters, Emmett Witchel, and Ed Wong, my sincere thanks to you all!

I also learned a great deal from my fellow student colleagues and friends: Sebastian Angel, Apurv Bhartia, Ram Chakravarthy, Allen Clement, Alan Dunn, Trinabh Gupta, Suman Jana, Manos Kapritsos, Anil Katti, Sangmin Lee, Michael Lee, Joshua Leners, Prince Mahajan, Gowri Parthasarathy, Don Porter, Sindhu Raghavan, Swati Rallapalli, Chris Rossbach, Indrajit Roy, Eric Rozner, Anand S., Esh VcKay, Subhashini Venugopalan, Shruthi Viswanath, Riad Wahby, and Ed Wong. Special thanks to Sebastian and his coffee machine for boosting my productivity during the job interview period.

Graduate school would have been far more stressful if not for all the help I got from the following wonderful administrative associates: Lydia Griffith, Katherine Utz, Leah Wimberly, Lindy Aleshire, Phyllis Bellon, and Sara Strandtman. They helped me with all the paperwork and administrative procedures in the university, even when I was not in town!

Finally, many thanks to my parents for believing in me and unconditionally supporting me in everything I wanted to do. I also thank my elder brother, Vinay, and my sister-in-law, Divya, for all their love, care, and affection.

Srinath T. V. Setty

The University of Texas at Austin
December 2014

Toward practical argument systems for verifiable computation

Publication No. ______________________

Srinath T. V. Setty, Ph.D.
The University of Texas at Austin, 2014

Supervisor: Lorenzo Alvisi
Co-supervisor: Michael Walfish

How can a client extract useful work from a server without trusting it to compute correctly? A modern motivation for this classic question is third party computing models in which customers outsource their computations to service providers (as in cloud computing).

In principle, deep results in complexity theory and cryptography imply that it is possible to verify that an untrusted entity executed a computation correctly. For instance, the server can employ probabilistically checkable proofs (PCPs) in conjunction with cryptographic commitments to generate a succinct proof of correct execution, which the client can efficiently check. However, these theoretical solutions are impractical: they require thousands of CPU years to verifiably execute even simple computations.

This dissertation describes the design, implementation, and experimental evaluation of a system, called Pepper, that brings this theory into the realm of plausibility. Pepper incorporates a series of algorithmic improvements and systems engineering techniques to improve performance by over 20 orders of magnitude, relative to an implementation of the theory without our refinements. These include a new probabilistically checkable proof encoding with nearly optimal asymptotics, a concise representation for computations, a more efficient cryptographic commitment primitive, and a distributed implementation of the server with GPU acceleration to reduce latency.

Additionally, Pepper extends the verification machinery to handle realistic applications of third party computing: those that interact with remote storage or state (e.g., MapReduce jobs, database queries). To do so, Pepper composes techniques from untrusted storage with the aforementioned technical machinery to verifiably offload both computations and state. Furthermore, to make it easy to use this technology, Pepper includes a compiler to automatically transform programs in a subset of C into executables that run verifiably.

One of the chief limitations of Pepper is that verifiable execution is still orders of magnitude slower than an unverifiable native execution. Nonetheless, Pepper takes powerful results from complexity theory and verifiable computation a few steps closer to practicality.

Contents

Previously published material

This dissertation describes a system called *Pepper*, which is a unified version of several systems that have appeared: Pepper [113], Ginger [115], Zaatar [112], and Pantry [45]. Furthermore, since Zaatar [112] supersedes Pepper [113] and Ginger [115] (in functionality and efficiency, for the most part), we combine these three systems under the name Zaatar.

We describe Zaatar in Chapter 4 by revising the following papers [111–113, 115]:

- S. Setty, A. J. Blumberg, M. Walfish. Toward practical and unconditional verification of remote computations, In HotOS, May 2011.
- S. Setty, R. McPherson, A. J. Blumberg, M. Walfish. Making argument systems for outsourced computation practical (sometimes), In NDSS, Feb. 2012.
- S. Setty, V. Vu, N. Panpalia, B. Braun, A. J. Blumberg, M. Walfish. Taking proof-based verified computation a few steps closer to practicality, In USENIX Security, Aug. 2012.
- S. Setty, B. Braun, V. Vu, A. J. Blumberg, B. Parno, M. Walfish. Resolving the conflict between generality and plausibility in verified computation, In ACM EuroSys, Apr. 2013.

And, we describe Pantry in Chapter 5 by revising the following paper [45]:

- B. Braun, A. J. Feldman, Z. Ren, S. Setty, A. J. Blumberg, M. Walfish. Verifying computations with state, In ACM SOSP, Nov. 2013.

Chapter 1

Introduction

How can a computer execute a program in such a way that an external entity can verify the correctness of the execution?

A modern motivation for this question is outsourcing of computations and storage to a third party, as in cloud computing [1, 5, 8]. On the one hand, cloud computing is appealing to customers. It allows anyone to rent virtually unlimited computing resources from a service provider, on a pay-per-use basis. Other benefits include high availability (as the cloud is geographically distributed and is accessible over the Internet), low cost (due to economies of scale), and resource elasticity (a customer can increase or decrease the amount of resources rented at any time). On the other hand, customers may not want to completely trust the cloud with their computations and data. First, the cloud is operated by a third party whose incentives may not be aligned with those of the customers. Second, the cloud is opaque to its users (by design). Third, cloud services use large-scale distributed systems (running on thousands or even millions of nodes), and at this scale, many things can and do go wrong (e.g., corruption of data in storage or in transit, software bugs, correlated hardware failures, misconfigurations). Indeed, when the cloud executes a computation on behalf of a client, how can the client be sure that the server computed correctly?

Variants of this problem have appeared under different names in the theory literature: checking computations [17], delegating computation [71], verifiable computation [62], etc. We group all of these notions of checking program executions under the category of *verifiable computation*.[1]

A simple solution for verifiable computation is to *replicate* computations [46, 50, 93]: execute the same computation on multiple servers, and then select an output computed by

[1]Our use of this term is broader than its definition by Gennaro et al. [62]; Chapter 3 provides details.

a majority. However, this strategy assumes that a majority of the servers computes correctly, which does not hold in the presence of correlated failures.

There are other pragmatic solutions, such as trusted hardware-based attestation [47, 105, 110] and auditing [75, 81, 99], but, like replication, they require assumptions about the failure modes of the server. Trusted hardware-based solutions assume a chain of trust rooted in the trusted hardware's manufacturer and that the trusted hardware works correctly; auditing assumes that the server either computes correctly or corrupts a large portion of its work.

In contrast to the above pragmatic approaches, deep results in complexity theory and cryptography imply solutions that do not make any assumptions about the failure modes of the server except, perhaps, cryptographic hardness assumptions. For instance, the server can employ interactive proofs (IPs) [71, 72, 90, 117], or probabilistically checkable proofs (PCPs) [14, 15, 17] coupled with cryptographic commitments (called *efficient arguments* [43, 79, 83, 84, 97]) to convince the client that it executed a computation correctly. However, these theoretical solutions are wildly impractical: they require thousands of years of CPU time to verifiably execute even simple computations.

Despite these costs, the theory presents enormous promise for verifiable computation and beyond. First, it applies to a general class of computations, and its guarantees hold regardless of the correctness of the server's hardware and software. Second, it can be applicable in scenarios far beyond verifying outsourced computation (for example, nodes in a message-passing distributed system could require each message to be accompanied by a short proof to convince its receiver that the message was generated by following a pre-defined protocol [47], a CPU could check computations it offloads to a GPU, or a base could validate a remotely deployed robot).

Several years ago, motivated by the above promise, we asked, *can we refine and incorporate strands of this theory into a built system that takes significant steps toward practicality, and then use it to verifiably outsource computations?* [111]. It was not clear whether we were going to make any progress toward this vision, since it requires reducing resource costs by a huge factor (e.g., from thousands of years to, say, a few seconds, in CPU time), which in turn calls for algorithmic improvements in an esoteric body of theory. Furthermore, in an early work on PCPs, theorists considered this exact application [17] (as noted above), *and*, despite a long line of work on establishing the theory of IPs, PCPs, and arguments—including recent efforts to improve the asymptotic efficiency of IPs [71] and PCPs [30–32]—the theory remained impractical. Thus, as expected, we faced nontrivial hurdles when we attempted to realize our vision.

Nonetheless, we have made significant progress. We have built a system, called Pepper, that brings strands of the aforementioned theory from wild impracticality into the realm of plausibility [45, 111–113, 115].

Despite dramatic improvements (20 orders of magnitude in some cases), Pepper is not yet truly practical. However, it has had impact. For instance, following our research agenda [111], there is now a thriving research area focused on building systems for verifiable computation [25–29, 39, 53, 63, 86, 104, 119, 120, 123]. We compare these works with Pepper in Chapter 2.

This dissertation describes Pepper; it builds on efficient argument protocols. We divide our contributions and results into two categories; each category corresponds to a subsystem of Pepper (as noted on page xiv).

(1) Verifiability for stateless computations (Chapter 4). The contributions in this category include a series of algorithmic refinements and systems engineering techniques to improve performance of an efficient argument protocol [79], by over 20 orders of magnitude (compared to an implementation of the protocol without our refinements). The most notable innovations in the improved efficient argument protocol are as follows.

- It includes an enhanced version of the cryptographic machinery of Ishai, Kushilevitz, and Ostrovsky (IKO) [79]. The enhanced version decreases end-to-end CPU and network costs for both the client and the server by several orders of magnitude. It is also simpler.

- It incorporates a new PCP construction based on a novel formalism to encode program executions, called *quadratic arithmetic programs (QAPs)*, due to Gennaro, Gentry, Parno, and Raykova (GGPR) [63]. The server's work is now $O(T \cdot \log T)$ instead of $O(T^2)$, where T is the number of steps in the corresponding computation.

We implement the aforementioned argument protocol in a system, called *Zaatar*. The system includes a compiler to automatically transform programs expressed in a high-level language into executables that run verifiably. Furthermore, the system includes a distributed implementation of the server and GPU acceleration for cryptographic operations to reduce latency.

(2) Verifiability for stateful computations (Chapter 5). Zaatar and related verifiable computation systems [25, 26, 39, 53, 63, 104, 120, 123] make great strides to bring the aforementioned theory toward practice. However, almost none of these systems support a notion of memory or state. The fundamental reason is that their underlying theory requires computations to be expressed as a set of *constraints* (§4.2), which is essentially a stateless model of computation. Without a notion of memory or state, these systems cannot support common uses of third party computing (e.g., MapReduce jobs over remotely stored massive data sets).

To solve this problem, we have refined the theory and built a system, called *Pantry*. It makes the following contributions.

- Pantry extends the technical machinery behind Zaatar (Chapter 4) and Pinocchio [63, 104] to support a notion of remote storage. To do so, Pantry composes techniques from untrusted storage [40, 60, 88, 96] with machinery for verifying stateless computations.
- Using the extended machinery, we build applications that require remote storage (a framework for verifiably executing MapReduce jobs, a simple database that supports a small subset of SQL) and a random access memory (RAM).
- We compose a variant of Pinocchio that supports a cryptographic property, zero-knowledge [72], with Pantry's solution to remote storage to support programs that compute over the server's private state.

We experimentally demonstrate that the client in Zaatar and Pantry can save computing resources, from outsourcing, under certain regimes. We also find that verifiable execution is still many orders of magnitude (up to 6 in our benchmarks) more expensive compared to an unverifiable execution. Nonetheless, Zaatar and Pantry can be considered nearly practical in scenarios where there are not pragmatic alternatives. Examples include (1) tasks that process a lot of remotely stored data, or compute intensive, or both, and need verifiability (§5.3), and (2) computations that work over the server's private state (§5.5).

Most importantly, Zaatar and Pantry take several significant steps toward making powerful theoretical results and verifiable computation practical. Furthermore, as we describe in Chapter 6, Zaatar, Pantry, and their contemporaries suggest that a low overhead version of PCP-based machinery could be achievable in the future and that PCPs could ultimately find their way into real systems.

Roadmap. Chapter 2 contains an overview of work related to Zaatar and Pantry. Chapter 3 describes theoretical constructs that Zaatar (Chapter 4) and Pantry (Chapter 5) build on. As a result, Chapters 4 and 5 assume familiarity with Chapter 3. Chapter 5 describes Pantry, which extends Zaatar (Chapter 4) and a related system, Pinocchio [104]; it does not assume familiarity with Zaatar as §5.1 provides an overview of Pantry's baseline systems. Chapter 6 summarizes and critiques Zaatar, Pantry, and the area of verifiable computation; thus, it assumes familiarity with Chapters 1–5.

Chapter 2

Related work

This chapter describes the landscape of solutions for verifiable outsourcing of computations and compares them with the two sub-systems of Pepper, namely Zaatar and Pantry. (Chapter 1 covers a few solutions, but this chapter intends to provide a comprehensive overview of the solution space; thus, this chapter may repeat a few details from the prior chapter.)

2.1 Approaches that make assumptions about failure modes

There are several approaches that enable verifiable computing: state machine replication [46, 50, 93], trusted hardware-based attestation [47, 105, 110], and auditing [73, 75, 81, 99]. These solutions are often pragmatic, but, unlike Zaatar and Pantry, they require stronger assumptions about the failure modes of the server.

As described in Chapter 1, state machine replication assumes that the server can be replicated such that replica failures are uncorrelated, and that a majority of the replicas always works correctly. Trusted hardware-based attestation assumes the server has a trusted hardware that always works correctly and that its manufacturer is trusted. Auditing assumes that whenever the server computes incorrectly, it corrupts a large portion of its work.

In addition to requiring the above assumptions, these pragmatic approaches do not always apply (e.g., state machine replication does not provide a solution to verifiable computing in which the server provides private inputs to the client's computation).

2.2 Approaches that apply to a restricted class of computations

Many works focus on designing protocols that enable verifying the correct execution of a specific class of computations. A well-known example is Freivalds' technique [100, §7.1] for checking the correctness of matrix multiplication. A set of works [33, 42, 58, 59, 103] design protocols for verifying polynomial evaluations. Backes et al. [18] support verifiable arithmetic computations (mean, variance, etc.) on outsourced data. Sion [118] and Thompson et al. [122] design protocols to verify the integrity of database operations. Wang et al. design protocols to verifiably outsource linear programming [124] and the problem of solving a system of linear equations [125]. Atallah et al. [16] design protocols to securely outsource linear algebra operations. There is also work to verify aggregate statistics computed from distributed sensors [61, 109]. Compared to Zaatar and Pantry, these solutions are sometimes more efficient, but, they do not apply to a general class of computations.

2.3 General solutions that are not geared toward practice

Theory of interactive proofs, PCPs, and argument protocols. There is a long line of seminal work that established the theory of interactive proofs [72, 90, 117], probabilistically checkable proofs [14, 15, 17, 56], and arguments [43, 49, 79, 83, 84, 97]. Many works have improved upon the early work on IPs and PCPs with better constructions (e.g., IPs that require only a polynomial prover, as opposed to a super-polynomial one [71], asymptotically short PCPs [30–32]).

However, unlike Zaatar and Pantry, achieving true practicality is not an explicit goal in these works. As a result, the constructions in these works do not admit an easy implementation and often have astronomically large constants. There is recent work on improving the concrete efficiency of PCPs [26], but this construction is mostly of theoretical interest at this point, since it remains far too expensive and intricate to implement [27, §1.2].

Approaches that rely on powerful cryptographic primitives. Gennaro, Gentry, and Parno [62] formalize verifiable computation (VC) as a non-interactive cryptographic primitive; then they describe a scheme for VC that composes Yao's garbled circuits [127] with Gentry's fully homomorphic encryption (FHE) [64]. Compared to Zaatar and Pantry, their protocol provides privacy for inputs and outputs of a computation. However, FHE is still far too expensive, despite performance breakthroughs [65, 66]. Thus, verifiable computation schemes based on FHE [48, 62] are orders of magnitude more expensive than Zaatar and Pantry.

Chung et al. [49] design protocols for verifiable computation that support a notion of remote state. However, unlike Pantry, their protocols rely on one or more of the following expensive building blocks: Micali's argument protocol [97] (which in turn requires asymptotically short PCPs), FHE, or private information retrieval.

2.4 Systems that share an ethos with Zaatar and Pantry

There are four other projects that have refined and implemented strands of theory similar to those behind Zaatar and Pantry. In general, there is a tradeoff between the types of computations to which these systems apply and the associated costs.

CMT, Allspice, and Thaler. Cormode, Mitzenmacher, and Thaler (CMT) [53, 120] refine and implement the interactive proofs protocol of Goldwasser, Kalai, and Rothblum [71]. Furthermore, the work of Thaler [119] includes new algorithmic techniques to make the server in the protocol of CMT essentially optimal. Compared to Zaatar, this line of work is often more efficient (e.g., the server's costs are over an order of magnitude cheaper), which is partially because it does not require expensive cryptographic operations. Additionally, their protocol is information theoretically secure. However, it requires many rounds of interaction between the client and the server (Zaatar requires only two). Furthermore, it applies only to a limited class of computations (specifically to computations that are naturally parallelizable).

Allspice [123] alleviates some of these expressiveness limitations, but it requires the client to pay a setup cost (which can be amortized by outsourcing multiple identical computations with potentially different inputs) to participate in the protocol.

GGPR, Pinocchio. Gennaro, Gentry, Parno, and Raykova (GGPR) [63] present a novel formalism to encode program executions, called quadratic arithmetic programs (QAPs). They combine QAPs with cryptographic machinery based on powerful bilinear pairings to design a protocol for verifiable computation. Zaatar observes that there is a connection between QAPs and PCPs, by designing a new PCP based on QAPs (Bitansky et al. [39] establish a similar relationship in concurrent work, in a different context); then, Zaatar composes the new PCP with the cryptographic machinery of Ishai et al. [79] to design an efficient argument protocol.

Pinocchio [104] refines the protocol of GGPR and implements it in full generality. Compared to Zaatar, Pinocchio achieves more cryptographic properties such as zero-knowledge, non-interactivity, public verifiability, and better amortization behavior, but it also pays higher cryptographic expense. An early version of Zaatar transformed programs expressed in a high-level language, called SFDL [92], into executables that run verifiably. Following Pinocchio, Zaatar supports a subset of C in addition to SFDL.

Like Zaatar, Pinocchio does not support a notion of state, but Pantry extends both Zaatar and Pinocchio to support stateful computations; most notably, Pantry leverages Pinocchio's zero-knowledge to support computations that work over the server's private state.

TinyRAM. Ben-Sasson et al. [25, 27, 29] design a novel compiler to transform programs expressed in the C programming language into circuits; these circuits represent the fetch-decode-execute loop of a simple processor. They combine this compiler with an optimized version of Pinocchio to obtain a system that can verifiably execute C programs. We refer to this full system as TinyRAM, for simplicity (note that Ben-Sasson et al. use that name to refer to the aforementioned simple processor).

Compared to Zaatar and Pantry, TinyRAM supports data-dependent looping much more naturally, but it does not include a notion of remote state. Additionally, TinyRAM's costs are often orders of magnitude worse than other systems in the area.

One of the intriguing aspects of TinyRAM is its circuit for checking memory coherence; this circuit is several orders of magnitude smaller than Pantry's Merkle-tree based circuit for checking memory coherence (§5.4.1 describes this design). However, recent work [10] shows how to use TinyRAM's techniques to replace Pantry's Merkle-tree based RAM; this reduces the cost of Pantry's RAM operations by several orders of magnitude.

More recently, Ben-Sasson et al. [28] refine the ideas of Bitansky et al. [38] and incorporate them into TinyRAM. The result is that the new implementation exchanges the client's setup costs for far higher server's costs: the client's setup cost is now independent of the length of the computation outsourced and the program itself, but the setup work is still substantial and the server's costs increase by several orders of magnitude.

TrueSet [86] incorporates an enhanced version of the QAPs of GGPR. Compared to Zaatar and Pinocchio, TrueSet has the same expressiveness, but, owing to the enhanced QAPs, it supports set operations much more efficiently. TrueSet's techniques are complementary, and they can be incorporated into Zaatar and Pantry.

Chapter 3

Problem statement and background

This chapter describes our problem statement and provides an overview of tools (probabilistically checkable proofs and argument protocols) that we build on.

3.1 Problem statement

Our goal is to implement the following protocol between two entities: a verifier, $\mathcal{V}$, and a prover, $\mathcal{P}$. First, $\mathcal{V}$ outsources the execution of a polynomial time computation, Ψ, on input, x, to $\mathcal{P}$. Second, $\mathcal{P}$ executes $\Psi(x)$, and returns an output, y. Finally, using randomness, $\mathcal{V}$ interrogates $\mathcal{P}$ to check if y is the correct output of $\Psi(x)$; if $\mathcal{P}$ convinces $\mathcal{V}$, then $\mathcal{V}$ accepts y, else it rejects. (Note that $\mathcal{P}$ is an abstraction, which could be implemented by a cluster of machines.)

In the above protocol, $\mathcal{V}$ assumes that $\mathcal{P}$ is computationally bounded, and hence $\mathcal{P}$ does not violate cryptographic hardness assumptions (e.g., $\mathcal{P}$ cannot solve discrete logarithms in polynomial time).

We call a protocol like the above a *verifiable computation protocol* if it provides the guarantees listed below. This definition is similar to the one by Gennaro et al. [62]; one difference is that we do not restrict the number of rounds of interaction between $\mathcal{V}$ and $\mathcal{P}$.

1. **Completeness.** If $y = \Psi(x)$, then a correct $\mathcal{P}$ can make $\mathcal{V}$ accept y, always.

2. **Soundness.** If $y \neq \Psi(x)$, then $\Pr\{\mathcal{V} \text{ accepts } y\} \leq \epsilon$, where the probability is over $\mathcal{V}$'s randomness, and ϵ can be made arbitrarily close to zero.

3. **Efficient delegation.** $\mathcal{V}$ saves computing resources by using $\mathcal{P}$ relative to executing Ψ,

perhaps on an *amortized* basis (e.g., when outsourcing many instances of Ψ each with potentially different inputs).

3.2 Verifiable computation from efficient argument protocols

An (obvious) way for $\mathcal{V}$ to verify if $\mathcal{P}$ computed correctly is to execute Ψ locally to obtain an output and check if that output matches $\mathcal{P}$'s output. But, under this solution, $\mathcal{V}$ does not save resources from outsourcing its work to $\mathcal{P}$. We now describe a way that uses powerful results from complexity theory and cryptography.

By definition, for every *NP* language $\mathcal{L}$, and for every problem instance $\mathcal{C}$, if $\mathcal{C} \in \mathcal{L}$, then there exists a witness, z, such that a deterministic polynomial-time algorithm can verify the membership of $\mathcal{C}$ in $\mathcal{L}$ using z. Remarkably, there also exists a proof π that convinces $\mathcal{V}$ of $\mathcal{C}$'s membership in $\mathcal{L}$ but only needs to be inspected in a *constant number* of places—yet if $\mathcal{C}$ is not in $\mathcal{L}$, then for any purported proof, the probability that $\mathcal{V}$ is wrongly convinced of $\mathcal{C}$'s membership can be arbitrarily close to zero. This remarkable statement is the rough content of the PCP theorem [14, 15].

Following [14], we take $\mathcal{L}$ to be Boolean circuit satisfiability: the question of whether the input wires of a given Boolean circuit $\mathcal{C}$ can be set to make $\mathcal{C}$ evaluate to 1.[1] It suffices to consider this problem because $\mathcal{L}$ is NP-complete; any other problem in NP can be reduced to it. Of course, a satisfying assignment z—a setting of all wires in $\mathcal{C}$ such that $\mathcal{C}$ evaluates to 1—constitutes an (obvious) proof that $\mathcal{C}$ is satisfiable: $\mathcal{V}$ could check z against every gate in $\mathcal{C}$. Note that this check requires inspecting all of z. In contrast, the PCP theorem yields a $\mathcal{V}$ that makes only a *constant number* of queries to an oracle π and satisfies:

- **Completeness.** If $\mathcal{C}$ is satisfiable, then there exists a linear function π (called a proof oracle) such that, after $\mathcal{V}$ queries π, $\Pr\{\mathcal{V}$ accepts $\mathcal{C}$ as satisfiable$\} = 1$, where the probability is over $\mathcal{V}$'s random choices.
- **Soundness.** If $\mathcal{C}$ is not satisfiable, then $\Pr\{\mathcal{V}$ accepts $\mathcal{C}$ as satisfiable$\} < \epsilon$ for *all* purported proof functions $\tilde{\pi}$. Here, ϵ is a constant that can be driven arbitrarily low.

However, there is still a problem in the context of verifying outsourced computation: $\mathcal{V}$ cannot ask $\mathcal{P}$ to send π (as π is significantly larger than the number of steps in Ψ), or query $\mathcal{P}$ for parts of π that it needs (the above guarantees hold only when $\mathcal{V}$ has oracle access

[1] A Boolean circuit is a set of interconnected gates, each with input wires and an output wire, with wires taking 0/1 values.

to an immutable π). Efficient argument protocols [43] circumvent these issues. The idea of efficient arguments is to use PCPs in an interactive protocol: $\mathcal{P}$ computes π and responds to $\mathcal{V}$'s queries. However to force $\mathcal{P}$ to behave like a fixed proof, $\mathcal{V}$ obtains a cryptographic commitment to π, and then decommits only those bits of π that it needs.

There are many constructions of PCPs and argument protocols in the theory literature. We chose to build on the argument protocol of Ishai, Kushilevitz, and Ostrovsky [79], which uses the probabilistically checkable proofs construction of Arora et al. [14, 15] (we justify our choice to build on these constructions in Section 3.2.2). We now turn to their details.

3.2.1 A PCP construction by Arora et al. [14, 15]

Arora et al. describe a PCP construction in which a correct proof, π, is a linear function over a finite field, $\mathbb{F}$. It is also presented in [31, 79], and we borrow some of our notation from these three sources. Following [79], we call such proofs *linear PCPs*. A linear function $\pi\colon \mathbb{F}^n \mapsto \mathbb{F}^b$ can be regarded as a $b \times n$ matrix M, where $\pi(q) = M \cdot q$; in the case $b = 1$, π returns a dot product with the input.

Recall that a motivation of PCPs is to avoid $\mathcal{V}$ having to check a purported assignment, z, against every gate, as that would be equivalent to reexecuting its computation. Instead, in the PCPs of Arora et al., $\mathcal{V}$ will construct a polynomial $Q(V, Z)$ that represents $\mathcal{C}$, using its randomness (see below), and π will be carefully constructed to allow evaluation of this polynomial. Suppose that there are s gates in $\mathcal{C}$. For each of the gates in $\mathcal{C}$, $\mathcal{V}$ creates a variable $Z_i \in \{0, 1\}$ that represents the output of gate i. $\mathcal{V}$ also creates s algebraic constraints, as follows. If gate i is the AND of Z_j and Z_k, then $\mathcal{V}$ adds the constraint $Z_i - Z_j \cdot Z_k = 0$; if gate i is the NOT of Z_j, then $\mathcal{V}$ adds the constraint $1 - (Z_i + Z_j) = 0$; if gate i is an input gate for the jth input, $\mathcal{V}$ adds the constraint $Z_i - in_j = 0$; and finally, for the last gate, representing the output of the circuit, we also have $Z_s - 1 = 0$. $\mathcal{V}$ then obtains the polynomial $Q(V, Z)$ by combining all of the constraints: $Q(V, Z) = \sum_{i=1}^{s} V_i \cdot Q_i(Z)$, where $Z = (Z_1, \ldots, Z_s)$, each $Q_i(Z)$ is given by a constraint (like the ones described above), and $\mathcal{V}$ chooses a value for $(V_1, \ldots, V_s)$ uniformly and independently at random from a finite field $\mathbb{F}$; we will denote those values as $v = (v_1, \ldots, v_s)$. The reason for the randomness is given immediately below.

Notice that $Q(v, z)$ detects whether z is a satisfying assignment: (1) if z is a satisfying assignment to the circuit, then it also satisfies all of the $\{Q_i(Z)\}$, yielding $Q(v, z) = 0$; but (2) if z is not a satisfying assignment to the circuit, then the randomness of the $\{v_i\}$ makes

$Q(v, z)$ unlikely to equal 0 (as illustrated in the next paragraph). Thus, the proof oracle π must encode a purported assignment z in such a way that $\mathcal{V}$ can quickly evaluate $Q(v, z)$ by making a few queries to π. To explain the encoding, let $\langle q_1, q_2 \rangle$ represent the inner (dot) product between two vectors q_1 and q_2, and $q_1 \otimes q_2$ represent the outer product $q_1 \cdot q_2^T$ (that is, all pairs of components from the two vectors). Observe that $\mathcal{V}$ can write

$$Q(v, Z) = \langle \gamma_2, Z \otimes Z \rangle + \langle \gamma_1, Z \rangle + \gamma_0.$$

The $\{\gamma_0, \gamma_1, \gamma_2\}$ are determined by the $\{Q_i(Z)\}$, the values on the input wires and output wires, and the choice of $\{v_i\}$, with $\gamma_2 \in \mathbb{F}^{s^2}$, $\gamma_1 \in \mathbb{F}^s$, and $\gamma_0 \in \mathbb{F}$. The reason that $\mathcal{V}$ can write $Q(v, Z)$ this way is that all of the $\{Q_i(Z)\}$ are degree-2 functions.

Given this representation of $Q(v, Z)$, $\mathcal{V}$ can compute $Q(v, z)$ by asking for $\langle \gamma_2, z \otimes z \rangle$ and $\langle \gamma_1, z \rangle$. This motivates the form of a correct proof, π. We write $\pi = (z, z \otimes z)$, by which we mean $\pi = (\pi^{(1)}, \pi^{(2)})$, where $\pi^{(1)}(\cdot) = \langle \cdot, z \rangle$ and $\pi^{(2)}(\cdot) = \langle \cdot, z \otimes z \rangle$. We refer to the vector $(z, z \otimes z)$ as a *proof vector* and denote it with u in this document. At this point, we have our first set of queries: $\mathcal{V}$ checks whether $\pi^{(2)}(\gamma_2) + \pi^{(1)}(\gamma_1) + \gamma_0 = 0$. If z is a satisfying assignment and π is correctly computed, the check passes. Just as important, if z' is *not* a satisfying assignment—which is always the case if $\mathcal{C}$ is not satisfiable—then $\mathcal{V}$ is not likely to be convinced. To see this, first assume that $\mathcal{V}$ is given a syntactically correct but non-satisfying $\tilde{\pi}$; that is, $\tilde{\pi} = (z', z' \otimes z')$, where z' is a non-satisfying assignment. The test above—that is, checking whether $\tilde{\pi}^{(2)}(\gamma_2) + \tilde{\pi}^{(1)}(\gamma_1) + \gamma_0 = 0$—checks whether $Q(v, z') = 0$. However, there must be at least one i' for which $Q_{i'}(z')$ is not 0, which means that the test passes if and only if $v_{i'} \cdot Q_{i'}(z') = -\sum_{i \neq i'} v_i \cdot Q_i(z')$. But the $\{v_i\}$ are conceptually chosen *after* z', so the probability of this event is upper-bounded by $1/|\mathbb{F}|$.

The above test is called the *circuit test*, and it has so far been based on an assumption: that if $\tilde{\pi}$ is invalid, it encodes *some* (non-satisfying) assignment. In other words, we have been assuming that $\tilde{\pi}^{(1)}$ and $\tilde{\pi}^{(2)}$ are linear functions that are consistent with each other. But of course a malevolently constructed oracle might not adhere to this requirement. To relax the assumption, we need two other checks. First, with *linearity tests* [22, 41], $\mathcal{V}$ makes three queries to $\pi^{(1)}$ and three to $\pi^{(2)}$, and checks the responses. If the checks pass, $\mathcal{V}$ develops a reasonable confidence that $\pi^{(1)}$ and $\pi^{(2)}$ are linear functions, which is another way of saying that $\pi^{(1)}(\cdot)$ is returning $\langle \cdot, z \rangle$ for some z and that $\pi^{(2)}(\cdot)$ is returning $\langle \cdot, u \rangle$ for some $u \in \mathbb{F}^{s^2}$. In the second test, the *quadratic correction test*, $\mathcal{V}$ makes four queries total and checks their responses; if the checks pass, $\mathcal{V}$ develops reasonable confidence that these two linear functions have the required relationship, meaning that $u = z \otimes z$. Once these tests have passed, the aforementioned precondition for the validity of circuit test holds.

In all, $\mathcal{V}$ makes $\ell = 14$ queries. The details of the queries and tests, and a formal statement of their completeness and soundness, are in [14] and Appendix A.1. Here, we just informally state that if $\mathcal{C}$ is satisfiable, then $\mathcal{V}$ will always be convinced by π, and if $\mathcal{C}$ is not satisfiable, then $\mathcal{V}$'s probability of passing the tests is upper bounded by a constant κ (for any $\tilde{\pi}$). If the scheme is repeated ρ times, for $\mu = \ell \cdot \rho$ total queries, the error probability ϵ becomes $\epsilon = \kappa^{\rho}$.

3.2.2 An argument protocol by Ishai, Kushilevitz, and Ostrovsky

Linear PCPs (like the one described above) are generally used as a building block in constructing more efficient PCPs, which are then used in an argument protocol. However, Ishai, Kushilevitz, and Ostrovsky (IKO) [79] design an argument protocol directly from linear PCPs.

IKO observe that in the PCP construction of Arora et al. [14, 15], π is a linear function (determined by z and $z \otimes z$); they develop a *commitment to a linear function* primitive. In this primitive, $\mathcal{P}$ commits to a linear function by pre-evaluating the function at a point chosen by $\mathcal{V}$ and hidden from $\mathcal{P}$; then, $\mathcal{V}$ submits one query, and the response must be consistent with the pre-evaluation. Roughly speaking, $\mathcal{V}$ can now proceed as if $\mathcal{P}$'s responses are given by an oracle π. (More accurately, $\mathcal{V}$ can proceed as if $\mathcal{P}$'s responses are given by a set of non-colluding oracles, one per PCP query.)

In more detail, $\mathcal{V}$ obtains a commitment from $\mathcal{P}$ by homomorphically encrypting a random vector r and asking $\mathcal{P}$ to compute $\mathsf{Enc}(\pi(r))$; $\mathcal{P}$ can do this without seeing r, by the linearity of π and the homomorphic properties of the encryption function (we do not need or assume fully homomorphic encryption [64]). $\mathcal{V}$ can then apply the decryption function to recover $\pi(r)$. To submit a PCP query q and obtain $\pi(q)$, $\mathcal{V}$ asks $\mathcal{P}$ for $\pi(q)$ and $\pi(r + \alpha q)$, for α randomly chosen from $\mathbb{F}$. $\mathcal{V}$ then requires that $\pi(r + \alpha q) = \pi(r) + \alpha\pi(q)$, or else $\mathcal{V}$ rejects $\pi(q)$. Figure 3.1 depicts this commitment protocol. By running parallel instances of their protocol (one for each time that $\mathcal{V}$ wants to inspect π), Ishai et al. convert any PCP protocol that uses linear functions into an *argument system* [43, 72].

Arguments are defined as follows; we borrow some of our notation and phrasing from [79], and we restrict the definition to Boolean circuits. An *argument* $(\mathcal{P}, \mathcal{V})$ *with soundness error* ϵ comprises two probabilistic polynomial time entities, $\mathcal{P}$ and $\mathcal{V}$, that take a Boolean

BasicCommit+BasicDecommit protocol of Ishai, Kushilevitz, and Ostrovsky (IKO) [79]

The protocol assumes an additive homomorphic encryption scheme $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ over a finite field, $\mathbb{F}$.

Commit phase

Input: Prover holds a vector $u \in \mathbb{F}^n$, which defines a linear function $\pi\colon \mathbb{F}^n \to \mathbb{F}$, where $\pi(q) = \langle u, q\rangle$.

1. Verifier does the following:
   - Generates public and secret keys $(pk, sk) \leftarrow \mathsf{Gen}(1^\lambda)$, where λ is a security parameter.
   - Generates vector $r \in_R \mathbb{F}^n$ and encrypts r component-wise, so $\mathsf{Enc}(pk, r) = (\mathsf{Enc}(pk, r_1), \ldots, \mathsf{Enc}(pk, r_n))$.
   - Sends $\mathsf{Enc}(pk, r)$ and pk to the prover.
2. Using the homomorphism in the encryption scheme, the prover computes $e \leftarrow \mathsf{Enc}(pk, \pi(r))$ without learning r. The prover sends e to the verifier.
3. The verifier computes $s \leftarrow \mathsf{Dec}(sk, e)$, retaining s and r.

Decommit phase

Input: the verifier holds $q \in \mathbb{F}^n$ and wants to obtain $\pi(q)$.

4. The verifier picks a secret $\alpha \in_R \mathbb{F}$ and sends to the prover (q, t), where $t = r + \alpha q \in \mathbb{F}^n$.
5. The prover returns (a, b), where $a, b \in \mathbb{F}$. If the prover behaved, then $a = \pi(q)$ and $b = \pi(t)$.
6. The verifier checks: $b \overset{?}{=} s + \alpha a$. If so, it outputs a. If not, it rejects, outputting $\perp$.

Figure 3.1: IKO's commitment protocol. V decommits an evaluation of $\mathcal{P}$'s linear function at a location, q. To decommit $\mathcal{P}$'s function at multiple locations, IKO run multiple instances of this protocol.

circuit $\mathcal{C}$ as input and meet two properties:

- **Completeness.** If $\mathcal{C}$ is satisfiable and $\mathcal{P}$ has access to the satisfying assignment z, then the interaction of $\mathcal{V}(\mathcal{C})$ and $\mathcal{P}(\mathcal{C}, z)$ always makes $\mathcal{V}(\mathcal{C})$ accept $\mathcal{C}$'s satisfiability.
- **Soundness.** If $\mathcal{C}$ is not satisfiable, then for every efficient malicious $\mathcal{P}^*$, the probability (over $\mathcal{V}$'s random choices) that the interaction of $\mathcal{V}(\mathcal{C})$ and $\mathcal{P}^*(\mathcal{C})$ makes $\mathcal{V}(\mathcal{C})$ accept $\mathcal{C}$ as satisfiable is $< \epsilon$.

Chapter 4

Zaatar: Verifying stateless computations

Probabilistically checkable proofs (PCPs) and argument protocols imply a solution for verifiable computation (as noted in Chapter 3). In more detail, PCPs and arguments enable a verifier, $\mathcal{V}$, to outsource its computation, Ψ, on its input, x, to a prover, $\mathcal{P}$, and then check if $y = \Psi(x)$ by issuing queries, using cryptography, to $\mathcal{P}$, where y is $\mathcal{P}$'s claimed output of the computation. This solution is very promising because it does not require $\mathcal{V}$ to make assumptions about the failure modes of $\mathcal{P}$ (i.e., the guarantees of the theory, mentioned in §3.2.2, hold regardless of how $\mathcal{P}$ behaves, provided it does not violate cryptographic hardness assumptions).

While promising, this solution is completely impractical for the following reasons:

- *The protocol is too complicated.* Asymptotically efficient PCP constructions [30–32] are far too intricate to implement and optimize.
- *The prover's overheads are far too enormous.* In addition to complicated constructions, state-of-the-art PCPs have astronomically large constants in their algorithms, which makes them too expensive in practice.
- *The phrasing of the computation is too primitive.* Most PCP constructions use Boolean circuits to encode computations. The Boolean circuit model can encode a general class of computations. However, Boolean circuits are usually far too verbose for most program constructs of a high-level language (e.g., a Boolean circuit that multiplies two 32-bit integers contains roughly 10,000 Boolean gates). Additionally, end-to-end costs of PCPs and argument protocols are directly proportional to the size of the representation of a computation. Furthermore, writing computations as Boolean circuits is (obviously) far too inconvenient.

- *The setup work is too high for the verifier.* Transforming Ψ into a Boolean circuit, $\mathcal{C}$, and generating queries to $\mathcal{P}$ take much more time for $\mathcal{V}$ than locally executing Ψ.

The first two obstacles above can be addressed by using the protocol of Ishai, Kushilevitz, and Ostrovsky (IKO) [79]. As described in Section 3.2.2, the key reason is that IKO design an argument protocol using linear PCPs [14, 15], which are simpler to implement. Thus, given the simplicity of the ingredients in the protocol of IKO, we chose to build on this strand of theory.

However, this choice brings two additional obstacles:

- *The prover's asymptotics are not ideal.* The prover's work is at least quadratic in the number of steps in Ψ.
- *The commitment protocol is far too expensive.* Commitment requires cryptographic operations and hence multiprecision arithmetic, and the scheme of Ishai et al. [79] invokes these operations far too much (by several orders of magnitude) to be practical.

We now turn to Zaatar, a system that refines and implements the argument protocol of IKO (for verifiably outsourcing computations), and addresses the remaining obstacles. Zaatar uses algebraic constraints over a large finite field to shrink program encoding and to broaden the space of computations (§4.2, §4.5); it reduces commitment costs by requiring fewer cryptographic operations while offering better security (§4.3); it uses batching to reduce $\mathcal{V}$'s setup costs (§4.4); and it employs a new probabilistically checkable proof encoding to achieve nearly ideal asymptotics for the prover (§4.6).

We provide an overview of Zaatar, before describing the aforementioned innovations.

4.1 Zaatar in a nutshell

Figure 4.1 depicts Zaatar. Zaatar provides the following interface to a client or a verifier, $\mathcal{V}$. $\mathcal{V}$ expresses its computation, Ψ, in a a high-level language [92], and sends it to a server or a prover, $\mathcal{P}$, along with a *batch* of different inputs, $x^{(1)}, \ldots, x^{(\beta)}$. $\mathcal{P}$ executes Ψ β times with different inputs to obtain a batch of outputs, $y^{(1)}, \ldots, y^{(\beta)}$, and returns the outputs to $\mathcal{V}$.

In Zaatar, verification is a three-step process. (We provide a summary here; the sections ahead contain details.) Step ①: $\mathcal{V}$ and $\mathcal{P}$ compile Ψ into a set of algebraic constraints over a large finite field (a generalization of arithmetic circuits). Step ②: $\mathcal{P}$ creates β linear

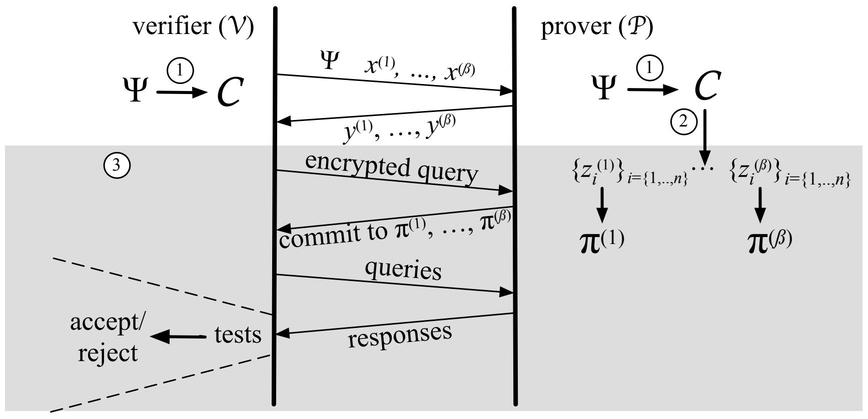
verifier ($\mathcal{V}$)
prover ($\mathcal{P}$)
$\Psi \xrightarrow{①} \mathcal{C}$
Ψ $x^{(1)}, \ldots, x^{(\beta)}$
$y^{(1)}, \ldots, y^{(\beta)}$
$\Psi \xrightarrow{①} \mathcal{C}$
②
③
$\{z_i^{(1)}\}_{i=\{1,..,n\}}$ … $\{z_i^{(\beta)}\}_{i=\{1,..,n\}}$
$\pi^{(1)}$
$\pi^{(\beta)}$
encrypted query
commit to $\pi^{(1)}, \ldots, \pi^{(\beta)}$
queries
responses
accept/ reject
tests

Figure 4.1: Zaatar in a nutshell. $\mathcal{V}$ outsources the execution of its computation, Ψ, on a batch of inputs, $x^{(1)}, \ldots, x^{(\beta)}$, to an untrusted prover, $\mathcal{P}$, which returns a batch of outputs, $y^{(1)}, \ldots, y^{(\beta)}$. We use superscripts to denote different instances of the same computation. Note that the prover could be distributed (i.e., each instance j could execute on a different machine). $\mathcal{V}$ checks if $\mathcal{P}$ computed correctly, using a three-step process. Step ①: $\mathcal{V}$ and $\mathcal{P}$ compile Ψ to a set of constraints, $\mathcal{C}$. Step ②: $\mathcal{P}$ produces satisfying assignments $z^{(1)}, \ldots, z^{(\beta)}$ to $\mathcal{C}(X{=}x^{(1)}, Y{=}y^{(1)}), \ldots, \mathcal{C}(X{=}x^{(\beta)}, Y{=}y^{(\beta)})$, respectively. Step ③: $\mathcal{V}$ and $\mathcal{P}$ engage in an interactive protocol in which $\mathcal{V}$ checks if $\mathcal{P}$ knows satisfying assignments to $\mathcal{C}(X{=}x^{(1)}, Y{=}y^{(1)}), \ldots, \mathcal{C}(X{=}x^{(\beta)}, Y{=}y^{(\beta)})$, which in turn implies that it computed correctly. $\mathcal{V}$'s queries are reused across all instances in the batch. Although computation need not happen in batch, verification cannot begin until $\mathcal{V}$ has all $y^{(j)}$. See §4.1 for additional details.

functions, $\pi^{(1)}, \ldots, \pi^{(\beta)}$, to convince $\mathcal{V}$ that $y^{(i)} = \Psi(x^{(i)})$ for $1 \le i \le \beta$. Step ③: $\mathcal{V}$ and $\mathcal{P}$ engage in an interactive protocol in which $\mathcal{V}$ checks if $\mathcal{P}$ computed correctly.

In more detail, step ③ proceeds in two phases (Figure 4.2 provides a high level overview). In the first phase, *the commit phase*, $\mathcal{V}$ obtains a commitment to $\mathcal{P}$'s linear functions by sending an encrypted query to $\mathcal{P}$. In the second phase, *the decommit phase*, $\mathcal{V}$ issues a set of PCP queries, $q_1, \ldots, q_\mu$, and $\mathcal{P}$ responds to them by returning $\pi^{(i)}(q_j)$, for $1 \le i \le \beta$ and $1 \le j \le \mu$. $\mathcal{V}$ first checks that $\mathcal{P}$'s responses are consistent with the commitment in the first phase, and then runs a set of PCP tests, once for each instance in the batch. If all the tests pass, $\mathcal{V}$ accepts $y^{(i)}$ as the correct output of $\Psi(x^{(i)})$ (for $1 \le i \le \beta$); otherwise, it rejects the entire batch. (Note that the execution of a computation need not happen in batch, but verification cannot begin until $\mathcal{V}$ has all the outputs.)

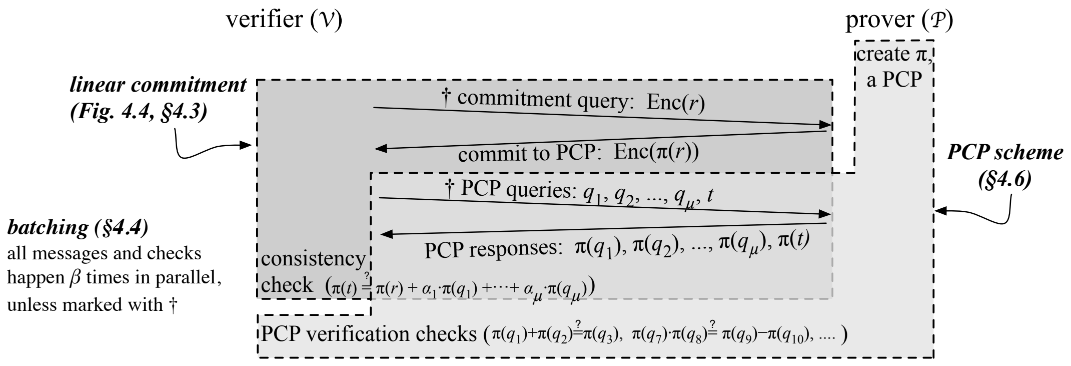
verifier ($\mathcal{V}$)
prover ($\mathcal{P}$)
create π, a PCP
linear commitment (Fig. 4.4, §4.3)
† commitment query: Enc(r)
commit to PCP: Enc(π(r))
PCP scheme (§4.6)
† PCP queries: $q_1, q_2, \ldots, q_\mu, t$
PCP responses: $\pi(q_1), \pi(q_2), \ldots, \pi(q_\mu), \pi(t)$
batching (§4.4)
all messages and checks happen β times in parallel, unless marked with †
consistency check $(\pi(t) \stackrel{?}{=} \pi(r) + \alpha_1 \cdot \pi(q_1) + \cdots + \alpha_\mu \cdot \pi(q_\mu))$
PCP verification checks $(\pi(q_1)+\pi(q_2) \stackrel{?}{=} \pi(q_3),\ \pi(q_7) \cdot \pi(q_8) \stackrel{?}{=} \pi(q_9) - \pi(q_{10}), \ldots)$

Figure 4.2: High-level depiction of step ③ in Figure 4.1. We formalize this picture and prove its soundness in Appendix A.2.

4.2 Arithmetic circuits, concise gates, and algebraic constraints

To address the concern about the encoding of the computation, we change the model of computation in the protocol of Ishai et al. [79] to be *arithmetic circuits*, instead of Boolean circuits. In a traditional arithmetic circuit, the input and output wires take values from a large set (e.g., a finite field or the integers). This extension is a natural one, as the PCP machinery is already expressed as algebraic versions of Boolean circuits (recall that $\mathcal{V}$ creates a set of algebraic constraints starting from a Boolean circuit, as noted in §3.2.1; the same process naturally extends to arithmetic circuits). However, we observe that the machinery also works with what we call *concise gates*, each of which encapsulates a function of many inputs (e.g., a dot product between two large vectors). Note that a gate here does not represent a low-level hardware element but rather a modular piece of the computation that enters the verification algorithm as an algebraic constraint.

This simple refinement is critical to practicality for many applications. First, it is vastly more compact to represent, say, multiplication of two 32-bit integers with a single gate than as a Boolean circuit. Beyond that, for certain computations (e.g., parallelizable numerical ones, such as matrix multiplication), the circuit model imposes no overhead; that is, the "circuit" is the same as a C++ program, so the only overhead comes from proving and verifying. However, this model has known limitations. For example, if a computation invokes a comparison operation, a bitwise operation, or a logical operation, arithmetic circuits degenerate to Boolean circuits.

To address the above expressiveness limitations of arithmetic circuits and of concise gates, we design algebraic constraints to encode those program constructs (instead of first representing those program constructs as arithmetic circuits or Boolean circuits and then transforming them into constraints). We now introduce some terminology and definitions.

A quadratic constraint is an equation of total degree 2 that uses additions and multiplications (e.g., $A \cdot Z_1 + Z_2 - Z_3 \cdot Z_4 = 0$). A set of constraints is *satisfiable* if the variables can be set to make all of the equations hold simultaneously; such an assignment is called a *satisfying assignment*. As a simple example, a computation that increments its input is equivalent to (in the sense of §3.2) the constraint set $\{Y = Z + 1, Z = X\}$. As we show in Section 4.5, this model of computation can concisely represent many commonly used program constructs. For now, we focus on a simple example.

Details and an example. Using algebraic constraints requires only minor modifications to the PCP scheme described in Section 3.2.1. Here, $\mathcal{V}$ produces a set of algebraic constraints (there, $\mathcal{V}$ transforms a Boolean circuit into a set of constraints) over s variables from a finite field $\mathbb{F}$ (there, over binary variables) that can be satisfied if and only if y is the correct output of $\Psi(x)$ (there, if and only if the circuit is satisfiable); $\mathcal{V}$ then combines those constraints to form a polynomial over s values in the field $\mathbb{F}$ (there, in the field $GF(2)$).

To illustrate the above, we use the example of $m \times m$ matrix multiplication. We choose this example because it is both a good initial test (it is efficiently encodable as a set of constraints) and a core primitive in many applications: image processing (e.g., filtering, rotation, scaling), signal processing (e.g., Kalman filtering), data mining, etc.

In this example computation, let A, B, C be $m \times m$ matrices over a finite field $\mathbb{F}$, with subscripts denoting entries, so $A = (A_{1,1}, \ldots, A_{m,m}) \in \mathbb{F}^{m^2}$ (for $\mathbb{F}$ sufficiently large we can represent negative numbers and integer arithmetic; see §4.5.1). The verifier $\mathcal{V}$ sends A and B to the prover $\mathcal{P}$, which returns C; $\mathcal{V}$ wants to check that $A \cdot B = C$. Matrix C equals $A \cdot B$ if and only if the following constraints over variables $Z = (Z^a_{1,1}, \ldots, Z^a_{m,m}, Z^b_{1,1}, \ldots, Z^b_{m,m}) \in \mathbb{F}^{2m^2}$ can be satisfied:

$$Z^a_{i,j} - A_{i,j} = 0, \text{ for } i,j \in [m]; \quad Z^b_{i,j} - B_{i,j} = 0, \text{ for } i,j \in [m];$$
$$C_{i,j} - \sum_{k=1}^{m} Z^a_{i,k} \cdot Z^b_{k,j} = 0, \text{ for } i,j \in [m].$$

$\mathcal{V}$ is interested in whether the above constraints can be met for some setting $Z =$

z (if so, the output of the computation is correct; if not, it is not). Thus, $\mathcal{V}$ proceeds as in Section 3.2.1. V constructs a polynomial $Q(V, Z)$ by combining the constraints: $Q(V, Z) = \sum_{i,j} v^a_{i,j} \cdot (Z^a_{i,j} - A_{i,j}) + \sum_{i,j} v^b_{i,j} \cdot (Z^b_{i,j} - B_{i,j}) + \sum_{i,j} v^c_{i,j} \cdot (C_{i,j} - \sum_{k=1}^{m} Z^a_{i,k} \cdot Z^b_{k,j})$, where $\mathcal{V}$ chooses the variables $\{v\}$ randomly from $\mathbb{F}$. As before, $\mathcal{V}$ regards the prover $\mathcal{P}$ as holding linear proof oracles $\pi = (\pi^{(1)}, \pi^{(2)})$, where $\pi^{(1)}(\cdot) = \langle \cdot, z \rangle$ and $\pi^{(2)}(\cdot) = \langle \cdot, z \otimes z \rangle$ for some $z \in \mathbb{F}^{2m^2}$. And as before, $\mathcal{V}$ issues linearity test queries, quadratic correction test queries, and circuit test queries (the randomly chosen $\{v\}$ feed into this latter test), repeating the tests ρ times.

The completeness and soundness of the above scheme follows from the completeness and soundness of the base protocol (Section 3.2.2). Thus, if $C = A \cdot B$ (more generally, if the claimed output y equals $\Psi(x)$), then $\mathcal{V}$ can be convinced of that fact; if the output is not correct, $\mathcal{P}$ has no more than $\epsilon = \kappa^\rho$ probability of passing verification.

Savings. Moving from a Boolean to a non-concise arithmetic circuit saves, for a fixed m, an estimated four orders of magnitude in the number of constraint variables (i.e., $|\{Z\}|$) and thus eight orders of magnitude in the query size and the prover's work (which are quadratic in the number of variables). The use of algebraic constraints decrease these quantities by another factor of m^2 (since they reduce the number of variables from $m^3 + 2m^2$ to $2m^2$). In Figure 4.3, the algebraic constraints column reflects these two reductions, the first being reflected in the elimination of the 10^9 factor and the second in the move from the m^6 to the m^4 term.

4.3 Strengthening linear commitment

The commitment protocol in the base scheme of Ishai et al. [79] relies on an additive homomorphic encryption operation.[1] If executed once, this operation is reasonably efficient (hundreds of microseconds; see Section 4.8); however, the number of times that the base scheme invokes it is proportional to at least the *square* of the input size *times* μ, the number of PCP queries (roughly 1000). For the example of $m \times m$ matrix multiplication with $m = 1000$, the base scheme would thus require at least $(1000)^4 \cdot 1000 \cdot 100$ μs: over 3000 years, and that's after the concise representation given by the previous refinement!

While we would be thrilled to eliminate homomorphic encryptions, we think that doing so is unlikely to work in this context. Instead, in this section we modify the com-

[1]Note that we do not require fully homomorphic encryption [64]; as we discuss in Section 2, the costs of such schemes are still prohibitive.

	op	naive impl. of [79]	batching (§4.4)	algebraic constraints (§4.2)	new commit (§4.3)	new PCPs (§4.6)
PCP encoding size ($\lvert\pi\rvert$)		$10^9 m^6$	$10^9 m^6$	$4m^4$	$4m^4$	$2m^3$
$\mathcal{V}$'s per-instance work (compare to local, naive computation: $f \cdot m^3$)						
Issue commit queries	$e + 2c$	$10^9 m^6 \cdot \mu'$	$10^9 m^6 \cdot \mu'/\beta$	$4m^4 \cdot \mu'/\beta$	$4m^4/\beta$	$2m^3/\beta$
Process commit responses	d	μ'	μ'	μ'	1	1
Issue PCP queries	c	$10^9 m^6 \cdot \mu'$	$10^9 m^6 \cdot \mu'/\beta$	$4m^4 \cdot \mu'/\beta$	$4m^4 \cdot \mu/\beta$	$2m^3 \cdot \mu/\beta$
Process PCP responses	f	$(2\mu' + 96m^2 \cdot \rho)$	$(2\mu' + 96m^2 \cdot \rho)$	$(2\mu' + 3m^2 \cdot \rho)$	$(2\mu + 3m^2 \cdot \rho)$	$(2\mu + 9m^2 \cdot \rho)$
$\mathcal{P}$'s per-instance work (compare to local, naive computation: $f \cdot m^3$)						
Issue commit responses	h	$10^9 m^6 \cdot \mu'$	$10^9 m^6 \cdot \mu'$	$4m^4 \cdot \mu'$	$4m^4$	$2m^3$
Issue PCP responses	f	$10^9 m^6 \cdot \mu'$	$10^9 m^6 \cdot \mu'$	$4m^4 \cdot \mu'$	$4m^4 \cdot \mu$	$2m^3 \cdot \mu$

$\ell = 14$: # of PCP queries per repetition (Appendix A.1)
$\rho = 70$: # of repetitions to drive error low (Appendix A.1)
$\mu = \ell \cdot \rho \approx 1000$: # of PCP queries per instance
$\mu' \approx 3\mu$: # of PCP queries pre-commit-refinement (§4.3)
β: # of computation instances batch verified (§4.1, §4.4)

$\lvert\pi\rvert$: # of components in matrix associated to linear function π (§3.2.1)
e: cost of homomorphic encryption of a single field element (Fig. 4.4, step 1)
d: cost of homomorphic decryption of a single field element (Fig. 4.4, step 3)
f: cost of field multiplication (Fig. 4.4, steps 4–6)
h: cost of ciphertext addition plus multiplication (Fig. 4.4, step 2)
c: cost of generating a pseudorandom # between 0 and a 192-bit prime (§4.7)

Figure 4.3: High-order costs under our refinements, cumulatively applied, compared to naive local execution and a naive implementation of [79], for our running example of $m \times m$ matrix multiplication. Rows for $\mathcal{V}$ and $\mathcal{P}$ contain operation counts, except for the "op" field, which includes a parameter denoting the cost of the operations in that row. Section 4.8 quantifies d, e, f, h, c, and β. The "PCP" rows include the consistency queries and checks (Fig. 4.4, steps 4–6). Figure C.2 generalizes the last column, and includes a few optimizations.

Commit+Multidecommit

The protocol assumes an additive homomorphic encryption scheme $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ over a finite field, $\mathbb{F}$.

Commit phase

Input: Prover holds a vector $u \in \mathbb{F}^n$, which defines a linear function $\pi\colon \mathbb{F}^n \to \mathbb{F}$, where $\pi(q) = \langle u, q \rangle$.

1. Verifier does the following:
   - Generates public and secret keys $(pk, sk) \leftarrow \mathsf{Gen}(1^\lambda)$, where λ is a security parameter.
   - Generates vector $r \in_R \mathbb{F}^n$ and encrypts r component-wise, so $\mathsf{Enc}(pk, r) = (\mathsf{Enc}(pk, r_1), \ldots, \mathsf{Enc}(pk, r_n))$.
   - Sends $\mathsf{Enc}(pk, r)$ and pk to the prover.
2. Using the homomorphism in the encryption scheme, the prover computes $e \leftarrow \mathsf{Enc}(pk, \pi(r))$ without learning r. The prover sends e to the verifier.
3. The verifier computes $s \leftarrow \mathsf{Dec}(sk, e)$, retaining s and r.

Decommit phase

Input: the verifier holds $q_1, \ldots, q_\mu \in \mathbb{F}^n$ and wants to obtain $\pi(q_1), \ldots, \pi(q_\mu)$.

4. The verifier picks μ secrets $\alpha_1, \ldots, \alpha_\mu \in_R \mathbb{F}$ and sends to the prover $(q_1, \ldots, q_\mu, t)$, where $t = r + \alpha_1 q_1 + \cdots + \alpha_\mu q_\mu \in \mathbb{F}^n$.
5. The prover returns $(a_1, a_2, \ldots, a_\mu, b)$, where $a_i, b \in \mathbb{F}$. If the prover behaved, then $a_i = \pi(q_i)$ for all $i \in [\mu]$, and $b = \pi(t)$.
6. The verifier checks: $b \overset{?}{=} s + \alpha_1 a_1 + \cdots + \alpha_\mu a_\mu$. If so, it outputs $(a_1, a_2, \ldots, a_\mu)$. If not, it rejects, outputting $\perp$.

Figure 4.4: Zaatar's commitment protocol. V decommits evaluations of $\mathcal{P}$'s linear function at locations, $q_1, \ldots, q_\mu$, with a single commitment phase. This is a strengthening of the commitment protocol of IKO (depicted in Figure 3.1) since in their protocol to decommit $\mathcal{P}$'s linear function at μ locations, $\mathcal{V}$ has to μ instances of the commit phase (and hence requiring $\mathcal{V}$ to perform a factor of μ more homomorphic encryptions relative to this version, and $\mu \approx 1000$). The protocol assumes an additive homomorphic encryption scheme but, with small modifications, works with a multiplicative homomorphic scheme, such as ElGamal [57] (details in Appendix A.4).

mitment protocol to perform three orders of magnitude fewer encryptions; Appendix A.2 proves the soundness of this modification by reducing its security to the semantic security of the homomorphic encryption scheme. Moreover, our reduction is more direct than in the base scheme, which translates into further cost reductions.

Details. In the base scheme [79], each PCP query by $\mathcal{V}$ (meaning each of the μ queries, as described in §3.2.1, §4.2, and Appendix A.1) requires $\mathcal{V}$ and $\mathcal{P}$ to run a separate instance of the commitment protocol in Figure 3.1. Thus, to check one computation Ψ, $\mathcal{V}$ homomorphically encrypts $\mu \approx 1000$ (see Figure 4.3) vectors, and $\mathcal{P}$ works over all of these ciphertexts. This factor of 1000 is an issue because the vectors are encrypted componentwise, and they have many components! (In the example above, they are elements of $\mathbb{F}^{s^2}$ or $\mathbb{F}^s$, where $s = 2 \cdot 10^6$.)

Figure 4.4 presents our modified commitment protocol. It homomorphically encrypts only one vector $r \in \mathbb{F}^{s^2+s}$, called the *commitment query*, with the encryption work amortizing over many queries $(q_1, \ldots, q_\mu)$. This new protocol leads to a more direct security reduction than in the base scheme. In their central reduction, Ishai et al. establish that commitment allows $\mathcal{V}$ to treat $\mathcal{P}$ as a *set* of non-colluding but possibly malicious oracles. In each repetition, their $\mathcal{V}$ must therefore issue extra queries (beyond the ℓ PCP queries) to ensure that the oracles match. With our commitment protocol, $\mathcal{V}$ can treat $\mathcal{P}$ as a *single* (possibly cheating) oracle and submit *only* the PCP queries. Stated more formally, Ishai et al. reduce linear PCP to linear MIP (multiprover interactive proof [24]) to the argument model, whereas we reduce linear PCP directly to the argument model. We prove the reduction in Appendix A.2.

Savings. This refinement reduces the homomorphic encryptions and other commitment-related work by three orders of magnitude, as depicted in Figure 4.3 by the elimination of the μ' term from the "commit" rows in the "new commit" column. As a second-order benefit, we save another factor of three (depicted in Figure 4.3 as a move from μ' to μ queries), as follows. The queries to establish the consistency of multiple oracles have error $(\ell - 1)/\ell = 13/14$. However, the soundness error of our base PCP protocol is $\kappa = 7/9$. Since $(13/14)^{\rho'} = (7/9)^{\rho}$ when $\rho' \approx 3\rho$, it takes roughly three times as many repetitions of the protocol to contend with this extra error. Finally, the direct reduction yields a qualitative benefit: it simplifies Zaatar.

4.4 Amortizing query costs through batching

Despite the optimizations so far, the verifier's work remains unacceptable. First, $\mathcal{V}$ must materialize a set of constraints that represent the computation, yet writing these down is as much work as executing the computation. Second, $\mathcal{V}$ must generate queries that are *larger* than the number of steps in Ψ. For example, for $m \times m$ matrix multiplication (§4.2), the commitment query has $4m^4 + 2m^2$ components (matching the number of components in the vector representation of the proof), in contrast to the $O(m^3)$ operations needed to execute the computation. A similar obstacle holds for many of the PCP queries. To amortize these costs, we modify the protocols to work over multiple computation instances and to verify computations in batch; we also rigorously justify these modifications. Note that the modifications do not reduce $\mathcal{V}$'s checking work, only $\mathcal{V}$'s cost to issue queries; however, this is acceptable since checking is fast.

Details. We assume that the computation Ψ (or equivalently, $\mathcal{C}$) is fixed; $\mathcal{V}$ and $\mathcal{P}$ will work over β instances of Ψ, with each instance having distinct input. We refer to β as the *batch size*. The prover $\mathcal{P}$ formulates β proof oracles (linear functions): $\pi_1, \ldots, \pi_\beta$. Note that the prover can stack these to create a linear function $\boldsymbol{\pi} : \mathbb{F}^{s^2+s} \to \mathbb{F}^\beta$ (one can visualize this as a matrix whose rows are $\pi_1, \ldots, \pi_\beta$).

To summarize the protocol, $\mathcal{V}$ now generates *one* set of commitment and PCP queries, and submits them to *all* of the oracles in the batch.[2] The prover now responds to queries q with $\boldsymbol{\pi}(q) \in \mathbb{F}^\beta$, instead of with $\pi(q) \in \mathbb{F}$. By way of comparison, the previous refinement (§4.3) encrypts a single *r* for a set of queries $q_1, \ldots, q_\mu$ to a proof π. This one issues a single *r* and a single set of queries $q_1, \ldots, q_\mu$ to multiple proofs $\pi_1, \ldots, \pi_\beta$. Appendix A.3 details the protocol and proves its soundness.

Savings. The most significant benefit is qualitative: without batching, $\mathcal{V}$ cannot gain from outsourcing, as the query costs are roughly the same as executing the computation. The quantitative benefit of this refinement is, as depicted in Figure 4.3, to reduce the per-instance cost of commitment and PCP queries by a factor of β.

4.5 Broadening the space of computations

As described in §4.2, to support a general programming model, Zaatar constructs algebraic constraints directly, instead of translating arithmetic circuits and concise gates into constraints. In particular, this section describes how Zaatar supports computations over floating-point fractional quantities and a programming model that includes inequality tests, logical expressions, conditional branching, etc. Additionally, our techniques[3] apply to the many protocols that use the constraint formalism or arithmetic circuits. Furthermore, the constraints that we design lend themselves to automatic compilation. That is, given a computation in a high level language, there is a mechanical process to generate a set of constraints such

[2]Early in their paper Ishai et al. briefly mention such an approach, but they do not specify it. Later in their paper [79, §6.1], in a more general context, they reuse PCP queries but not commitment queries.

[3]We suspect that many of the individual techniques are known. However, when the techniques combine, the material is surprisingly hard to get right, so we will delve into (excruciating) detail, consistent with our focus on built systems.

that they are equivalent to the computation (in the sense of §3.2, extended to a set of algebraic constraints), and then one could apply an argument protocol to verifiably execute the computation (as described in §3.2). In fact, we have implemented a compiler (derived from Fairplay's [92]) that transforms high-level computations first into constraints and then into verifier and prover executables.

Framework to map computations to constraints. The challenges of representing computations as constraints over finite fields include: the "true answer" to the computation may live outside of the field; sign and ordering in finite fields interact in an unintuitive fashion; and constraints are simply equations, so it is not obvious how to represent comparisons, logical expressions, and control flow. To explain Zaatar's solutions, we first present an abstract framework that illustrates how Zaatar broadens the set of computations soundly and how one can apply the approach to further computations.

Recall that in §3.2, we used $\mathcal{C}$ to refer to a Boolean circuit. We now use $\mathcal{C}$ to refer to a set of algebraic constraints. In our context, a set of constraints $\mathcal{C}$ will have a designated input variable X and output variable Y (this generalizes to multiple inputs and outputs), and we use $\mathcal{C}(X{=}x, Y{=}y)$ to denote $\mathcal{C}$ with variable X bound to x and Y bound to y.[4] We say that a set of constraints $\mathcal{C}$ is *equivalent* to a desired computation Ψ if: for all x, y, $\mathcal{C}(X{=}x, Y{=}y)$ is satisfiable (i.e., there exists a setting of unbound values for variables in $\mathcal{C}(X{=}x, Y{=}y)$ such that all constraints evaluate to 0) if and only if $y = \Psi(x)$.

To map a computation Ψ over some domain D (such as the integers, $\mathbb{Z}$, or the rationals, $\mathbb{Q}$) to equivalent constraints over a finite field, the programmer or compiler performs three steps, as illustrated and described below:

$$\Psi \text{ over } D \xrightarrow{\text{(C1)}} \Psi \text{ over } U \xrightarrow{\text{(C2)}} \theta(\Psi) \text{ over } \mathbb{F} \xrightarrow{\text{(C3)}} \mathcal{C} \text{ over } \mathbb{F}$$

C1 *Bound the computation.* Define a set $U \subset D$ and restrict the input to Ψ such that the output and intermediate values stay in U.

C2 *Represent the computation faithfully in a suitable finite field.* Choose a finite field, $\mathbb{F}$, and a map $\theta\colon U \to \mathbb{F}$ such that computing $\theta(\Psi)$ over $\theta(U) \subset \mathbb{F}$ is isomorphic to computing Ψ over U. (By "$\theta(\Psi)$", we mean Ψ with all inputs and literals mapped by θ.)

C3 *Transform the finite field version of the computation into constraints.* Write a set of con-

[4]Note that in the example mentioned in §4.2, these designated input and output variables are implicit.

```
Ψ :

if (X1 < X2)
  Y = 3
else
  Y = 4
```

$$\mathcal{C}_< = \left\{ \begin{array}{ll} B_0(1-B_0) & = 0, \\ B_1(2-B_1) & = 0, \\ \vdots & \vdots \\ B_{N-2}(2^{N-2} - B_{N-2}) & = 0, \\ \theta(X_1) - \theta(X_2) - (p - 2^{N-1}) - \sum_{i=0}^{N-2} B_i & = 0 \end{array} \right\} \qquad \mathcal{C}_\Psi = \left\{ \begin{array}{l} M\{\mathcal{C}_<\}, \\ M(Y-3) = 0, \\ (1-M)\{\mathcal{C}_{>=}\}, \\ (1-M)(Y-4) = 0 \end{array} \right\}$$

Figure 4.5: Pseudocode for our case study of Ψ, and corresponding constraints $\mathcal{C}_\Psi$. Ψ's inputs are signed integers x_1, x_2; per steps C1 and C2 (§4.5.1), we assume $x_1 - x_2 \in U \subset [-2^{N-1}, 2^{N-1})$, where $p > 2^N$. The constraints $\mathcal{C}_<$ test $x_1 < x_2$ by testing whether the bits of $\theta(x_1) - \theta(x_2)$ place it in $[p - 2^{N-1}, p)$. $M\{\mathcal{C}\}$ means multiplying all constraints in $\mathcal{C}$ by M and then reducing to degree-2.

straints over $\mathbb{F}$ that are equivalent to $\theta(\Psi)$.

4.5.1 Signed integers and floating-point rationals

We now instantiate C1 and C2 for integer and rational number computations; the next section addresses C3.

Consider $m \times m$ matrix multiplication over N-bit signed integers. For step C1, each term in the output, $\sum_{k=1}^{m} A_{ik}B_{kj}$, has m additions of $2N$-bit subterms so is contained in $[-m \cdot 2^{2N-1}, m \cdot 2^{2N-1})$; this is our set U.

For step C2, take $\mathbb{F} = \mathbb{Z}/p$ (the integers mod a prime p, to be chosen shortly) and define $\theta\colon U \to \mathbb{Z}/p$ as $\theta(u) = u \bmod p$. Observe that θ maps negative integers to $\{\frac{p+1}{2}, \frac{p+3}{2}, \ldots, p-1\}$, analogous to how processors represent negative numbers with a 1 in the most significant bit (this technique is standard [42, 125]). Of course, addition and multiplication in $\mathbb{Z}/p$ do not "know" when their operands are negative. Nevertheless, the computation over $\mathbb{Z}/p$ is isomorphic to the computation over U, provided that $|\mathbb{Z}/p| > |U|$ (as shown in Appendix B.1). Thus, for the given U, we require $p > m \cdot 2^{2N}$. Note that a larger p brings larger costs (see Figure C.2 and §4.8.1), so there is a three-way trade-off among p, m, N.

We now turn to rational numbers. For step C1, we restrict the inputs as follows: when written in lowest terms, their numerators are $(N_a + 1)$-bit signed integers, and their denominators are in $\{1, 2, 2^2, 2^3, \ldots, 2^{N_b}\}$. Note that such numbers are (primitive) floating-point numbers: they can be represented as $a \cdot 2^{-q}$, so the decimal point floats based on q. Now, for $m \times m$ matrix multiplication, the computation does not "leave" $U = \{a/b \colon |a| < 2^{N'_a}, b \in \{1, 2, 2^2, 2^3, \ldots, 2^{N'_b}\}\}$, for $N'_a = 2N_a + 2N_b + \log_2 m$ and $N'_b = 2N_b$ (shown in Appendix B.1).

For step C2, we take $\mathbb{F} = \mathbb{Q}/p$, the quotient field of $\mathbb{Z}/p$. Take $\theta(\frac{a}{b}) = (a \bmod p, b \bmod p)$. For any $U \subset \mathbb{Q}$, there is a choice of p such that the mapped computation over $\mathbb{Q}/p$ is isomorphic to the original computation over $\mathbb{Q}$ (shown in Appendix B.1). For our U above, $p > 2m \cdot 2^{2N_a+4N_b}$ suffices.

Limitations and costs. To understand the limitations of Zaatar's floating-point representation, consider the number $a \cdot 2^{-q}$, where $|a| < 2^{N_a}$ and $|q| \leq N_q$. To represent this number, the IEEE standard requires roughly $N_a + \log N_q + 1$ bits [68] while Zaatar requires $N_a + 2N_q + 1$ bits (shown in Appendix B.1). As a result, Zaatar's range is vastly more limited: with 64 bits, the IEEE standard can represent numbers on the order of 2^{1023} and 2^{-1022} (with $N_a = 53$ bits of precision) while 64 bits buys Zaatar only numbers on the order of 2^{32} and 2^{-31} (with $N_a = 32$). Moreover, unlike the IEEE standard, Zaatar does not support a division operation or rounding.

However, comparing Zaatar's floating-point representation to its *integer* representation, the extra costs are not terrible. First, the prover and verifier take an extra pass over the input and output (for implementation reasons; see Appendix B.1 for details). Second, a larger prime p is required. For example, $m \times m$ matrix multiplication with 32-bit integer inputs requires p to have at least $\log_2 m + 64$ bits; if the inputs are rationals with $N_a = N_q = 32$, then p requires $\log_2 m + 193$ bits. The end-to-end costs are about 2× those of the integers case (see Section 4.8). Of course, the actual numbers depend on the computation. (Our compiler computes suitable bounds with static analysis.)

4.5.2 General-purpose program constructs

Case study: branch on order comparison. We now illustrate C3 with a case study of a computation, Ψ, that includes a less-than test and a conditional branch; pseudocode for Ψ is in Figure 4.5. For clarity, we will restrict Ψ to signed integers; handling rational numbers requires additional mechanisms (see Appendix B.2).

How can we represent the test $x_1 < x_2$ using constraint *equations*? The solution is to use special *range constraints* that decompose a number into its bits to test whether it is in a given range; in this case, $\mathcal{C}_<$, depicted in Figure 4.5, tests whether $e = \theta(x_1) - \theta(x_2)$ is in the "negative" range of $\mathbb{Z}/p$ (see Section 4.5.1). Now, under the input restriction $x_1 - x_2 \in U$, $\mathcal{C}_<$ is satisfiable if and only if $x_1 < x_2$ (shown in Appendix B.2). Analogously, we can construct $\mathcal{C}_{>=}$ that is satisfiable if and only if $x_1 \geq x_2$.

Finally, we introduce a 0/1 variable M that encodes a choice of branch, and then arrange for M to "pull in" the constraints of that branch and "exclude" those of the other. (Note that the prover need not execute the untaken branch.) Figure 4.5 depicts the complete set of constraints, $\mathcal{C}_\Psi$; these constraints are satisfiable if and only if the prover correctly computes Ψ (shown in Appendix B.2).

Logical expressions and conditionals. Besides order comparisons and if-else, Zaatar can represent `==`, `&&`, and `||` as constraints. An interesting case is `!=`: we can represent `Z1!=Z2` with $\{M \cdot (Z_1 - Z_2) - 1 = 0\}$ because this constraint is satisfiable when $(Z_1 - Z_2)$ has a multiplicative inverse and hence is not zero. These constructs and others are detailed in Appendix B.3.

Limitations and costs. We now assess the limitations and costs of Zaatar's programming model. As noted above, Zaatar includes a compiler to automatically transform programs into a set of constraints. It compiles a subset of SFDL, the language of the Fairplay compiler [92]. Thus, our limitations are essentially those of SFDL; notably, loop bounds have to be known at compile time.

How efficient is our representation? The program constructs above mostly have concise constraint representations. Consider, for instance, `comp1==comp2`; the equivalent constraint set $\mathcal{C}$ consists of the constraints that represent `comp1`, the constraints that represent `comp2`, and an additional constraint to relate the outputs of `comp1` and `comp2`. Thus, $\mathcal{C}$ is the same size as its two components, as one would expect.

However, two classes of computations are costly. First, inequality comparisons require variables and a constraint for every bit position; see Figure 4.5. Second, the constraints for if-else and `||`, as written, seem to be degree-3; notice, for instance, the $M\{C_<\}$ in Figure 4.5. To be compatible with the core protocol, these constraints must be rewritten to be total degree 2 (as mentioned in §4.3), which carries costs. Specifically, if $\mathcal{C}$ has s variables and $|\mathcal{C}|$ constraints, an equivalent total degree 2 representation of $M\{\mathcal{C}\}$ has $s + |\mathcal{C}|$ variables and $2 \cdot |\mathcal{C}|$ constraints (shown in Appendix B.3).

4.6 A new probabilistically checkable proof encoding

As mentioned earlier in the chapter, the protocol of Ishai et al. [79] faces a severe obstacle to plausible practicality: the prover's per-instance work and the verifier's query setup work are *quadratic* in the size of the computation. We make this statement more precise below, but for now recall from Section 3.2.2 that the server's proof vector, u, in the protocol of Ishai et al. is $(z, z \otimes z)$, so $|u| = |z| + |z|^2$.

In this section, we describe how Zaatar addresses the issue. This section describes Zaatar's encoding, in which the proof vector size and the verifier's setup work are *linear* in the size of the computation.[5]

The high-level idea in Zaatar's encoding is to retain IKO's structure but to replace the linear PCP of Arora et al. [14] with a new linear PCP that is based on Quadratic Arithmetic Programs (QAPs), a formalism to encode program executions introduced by Gennaro, Gentry, Parno, and Raykova (GGPR) [63, §7–8]. However, this encoding imposes costs. Appendix C.2 weighs these costs against the benefits, finding that the linear PCP is far more favorable than the alternative.

This substitution works because (a) the linear commitment protocol requires only that the PCP is a linear function; and (b) QAPs have a linear query structure that yields a PCP, a key observation of Zaatar (Bitansky et al. [39] concurrently make a similar observation; see Chapter 2). To obtain this QAP-based PCP, we extract the essence of GGPR's construction (which is more complex because it is geared to a different regime; see Section 2). Once we do so, we inherit a prover whose proof vector is, to first approximation, the satisfying assignment itself.

Since PCPs "derive their magic" from a highly redundant encoding of the proof, it may seem surprising that we have a protocol in which the proof vector needs little redundancy. However, as Ishai et al. [79] observed, a linear PCP contains *implicit* redundancy because the actual proof is *not* the vector but rather the linear function, which is exponentially larger than the classical proof (if written out as a string, it would contain an entry for every point in its domain). For this reason, Ishai et al. speculated that avoiding redundancy in the proof vector might be possible; the construction described below resolves this conjecture.

Loosely speaking, QAPs achieve this compaction by encoding circuits (we adapt the encoding to constraints) as high-degree polynomials, in contrast to the low-degree polynomials of Section 3.2.1.

Details. The new PCP protocol takes as given a constraint set $\mathcal{C}$, over the variables (X, Y, Z), that is equivalent to a computation Ψ. (Recall that X is the set of input variables, Y is the set of output variables, Z are the unbound variables, and let $|\mathcal{C}|$ be the number of constraints in $\mathcal{C}$.) The protocol constructs two polynomials. The construction is somewhat analogous to

[5]These costs include an extra log factor applied to the number of steps in the computation. The reason is that the size of our field $\mathbb{F}$ must be larger than the number of steps in the computation, and meanwhile each entry in the proof is $\log|\mathbb{F}|$ bits. However, we neglect this factor in our description by referring to "the size of the computation", which captures the field size.

the description in Section 3.2.1; we will mention some of the parallels. The first polynomial, which we call the *divisor polynomial*, $D(t)$, is univariate (and over $\mathbb{F}$) and fixed for all computations Ψ of a given size; $\mathcal{V}$ explicitly materializes $D(t)$. The second polynomial, $P_{x,y}(t, Z)$, depends on the constraints $\mathcal{C}$, the input x, and the purported output y. We write this polynomial as $P(t, Z)$; it is analogous to the polynomial $Q(V, Z)$ in Section 3.2.1, though here $t \in \mathbb{F}$, versus $V \in \mathbb{F}^{|\mathcal{C}|}$. As with $Q(V, Z)$, neither party fully materializes $P(t, Z)$. We give the complete construction in Appendix C.1 and here state the relevant properties.

The construction ensures that given z, $P(t, z)$ can be factored as $D(t) \cdot H_{x,y,z}(t)$ for some $H_{x,y,z}(t)$ if and only if z satisfies $\mathcal{C}(X{=}x, Y{=}y)$. Meanwhile, this factoring (and hence satisfiability) can be checked *probabilistically*, as follows. Let τ be a random choice from $\mathbb{F}$. Then (1) If z satisfies $\mathcal{C}(X{=}x, Y{=}y)$, then the polynomials factor, so we have $D(\tau) \cdot H_{x,y,z}(\tau) = P(\tau, z)$, for all $\tau \in \mathbb{F}$; but (2) If z is not a satisfying assignment, then for *all* polynomials $\tilde{H}(t)$, we have $D(\tau) \cdot \tilde{H}(\tau) \neq P(\tau, z)$, except with probability $2 \cdot |\mathcal{C}|/|\mathbb{F}|$. This is because polynomials that are different are equal almost nowhere in their domains (an extreme case is two lines, which cross at most once). Since our fields are large (§4.8), the preceding probability is very small.

The query procedure and $\mathcal{P}$'s proof vector, then, are designed to allow $\mathcal{V}$ to check whether $D(t) \cdot H_{x,y,z}(t) = P(t, z)$, by checking whether this relation holds at a point τ. Specifically, they allow $\mathcal{V}$ to obtain the values $H(\tau) \in \mathbb{F}$ and $P(\tau, z) \in \mathbb{F}$, where: $\mathcal{V}$ chooses τ randomly from $\mathbb{F}$, $\mathcal{P}$ holds the polynomial $H(t)$ and the assignment z ($\mathcal{V}$ has no direct access to either), and neither party materializes $P(t, Z)$. (The analogy here is with the queries that allow $\mathcal{V}$ to obtain $Q(v, z)$, in Section 3.2.1.) If $D(\tau) \cdot H(\tau) = P(\tau, z)$, then $\mathcal{V}$ accepts and otherwise rejects. (The analogy is with the condition that $Q(v, z) = 0$, in Section 3.2.1.) This procedure probabilistically checks whether z is a satisfying assignment; it is complete and sound because of properties (1) and (2) above.

The proof vector. A correct proof vector u is (z, h), where z is a purported satisfying assignment to $\mathcal{C}(X{=}x, Y{=}y)$, and $h = (h_0, \ldots, h_{|\mathcal{C}|}) \in \mathbb{F}^{|\mathcal{C}|+1}$ are the coefficients of the polynomial $H_{x,y,z}(t)$, introduced above. As in Section 3.2.1, this proof vector u can be regarded as two linear functions, which we denote $\pi_z(\cdot) = \langle \cdot, z \rangle$ and $\pi_h(\cdot) = \langle \cdot, h \rangle$. Thus, the proof vector's length is equal to the number of variables plus the number of constraints, or $|Z| + |\mathcal{C}|$.

The queries and check. To carry out the probabilistic check described above, $\mathcal{V}$ must first ensure that $\mathcal{P}$ is holding a linear function, so $\mathcal{V}$ issues linearity queries (as in Section 3.2.1).

Next, $\mathcal{V}$ must obtain the value $H_{x,y,z}(\tau) \in \mathbb{F}$. To do so, $\mathcal{V}$ submits $q_h = (1, \tau, \tau^2, \ldots, \tau^{|\mathcal{C}|})$ to $\mathcal{P}$ and asks for $\pi_h(q_h)$, which equals $\langle q_h, h \rangle = \sum_{i=0}^{|\mathcal{C}|} h_i \cdot \tau^i = H_{x,y,z}(\tau)$.

$\mathcal{V}$ also needs the value $P(\tau, z) \in \mathbb{F}$. As shown in Appendix C.1, $P(t, Z)$ is a polynomial in t and Z, with the form

$$P(t,Z) = \left(\sum_{i=1}^{|Z|} Z_i \cdot A_i(t) + A'(t)\right) \cdot \left(\sum_{i=1}^{|Z|} Z_i \cdot B_i(t) + B'(t)\right) - \left(\sum_{i=1}^{|Z|} Z_i \cdot C_i(t) + C'(t)\right),$$

for some polynomials $\{A_i(t), B_i(t), C_i(t)\}_{i=1\ldots|Z|}$ and $\{A'(t), B'(t), C'(t)\}$. Now, observe that evaluating $P(t, Z)$ at $t{=}\tau$ yields $P(\tau, Z)$, a polynomial in Z with the form:

$$P(\tau, Z) = (\langle q_a, Z\rangle + L_a) \cdot (\langle q_b, Z\rangle + L_b) - (\langle q_c, Z\rangle + L_c)\,,$$

where $\{q_a, q_b, q_c\} \in \mathbb{F}^{|Z|}$ depend on τ, and $\{L_a, L_b, L_c\} \in \mathbb{F}$ depend on τ, x, and y. Finally we can say how $\mathcal{V}$ obtains $P(\tau, z)$: it asks $\mathcal{P}$ for $\pi_z(q_a)$, $\pi_z(q_b)$, and $\pi_z(q_c)$.

$\mathcal{V}$'s check is then the following. $\mathcal{V}$ computes $D(\tau)$ and $\{L_a, L_b, L_c\}$, and checks

$$D(\tau) \cdot \pi_h(q_h) \overset{?}{=} (\pi_z(q_a) + L_a) \cdot (\pi_z(q_b) + L_b) - (\pi_z(q_c) + L_c)\,.$$

Note that the set of $\mathcal{V}$'s protocol that we just described is not complete. In particular, for simplicity, it does not include *self-correction* (the purpose of self-correction is described in [14, §5] and [100, §7.8.3]). However, the full protocol and its analysis are in Appendix C.

4.7 Implementation

Zaatar's implementation consists of two components: (1) a library that implements an enhanced version of the argument protocol of IKO [79] (§4.2–4.6), and (2) a compiler that transforms a program in a high-level language into a set of constraints.

Zaatar's argument protocol is implemented in C++ (about 6000 lines, per [126]). Zaatar's compiler consists of two stages. The front-end compiles a subset of Fairplay's SFDL [92] to constraints. This transformation is detailed elsewhere [44], but broadly speaking, it works as follows. The front-end turns a program (even if it has conditionals and loops) into a list of assignment statements; then it produces a constraint or *pseudoconstraint* for each statement (pseudoconstraints abstract certain operations; for instance, order comparisons expand to $O(\log \mathbb{F})$ actual constraints). It is derived from Fairplay and is implemented in 5294 lines of Java, starting from Fairplay's 3886 lines (per [126]). The back-end transforms constraints into C++ code, which invokes the library code that implements Zaatar's argument protocol;

the back-end then invokes gcc to generate executables for the prover and the verifier. The back-end is 1105 lines of Python code.

When executed, the verifier and the prover run as separate processes and exchange data using Open MPI [9]. Both the verifier and the prover can offload their cryptographic operations to GPUs using CUDA [3]; in addition, the prover can be distributed over multiple machines, with each machine computing a subset of a batch (as we describe below). For encryption (see Figure 4.4), we use ElGamal [57] with 1024-bit keys; for a pseudorandom generator, we use the `amd64-xmm6` variant of the ChaCha/8 stream cipher [34].

Parallelization. Many of Zaatar's remaining costs are in the cryptographic operations in the commitment protocol (Figures 4.3 and C.2). To mitigate these costs, we distribute the prover over multiple machines, leveraging Zaatar's inherent parallelism (from batching, described in §4.4). We also implement the prover and verifier on GPUs, which raises two questions. (1) Isn't this just moving the problem? Yes, and this is good: GPUs are optimized for the types of operations that bottleneck Zaatar. (2) Why do we assume that the *verifier* has a GPU? Desktops are more likely than servers to have GPUs, and the prevalence of GPUs is increasing. Also, this setup models a future in which specialized hardware for cryptographic operations is common.

To distribute Zaatar's prover, we run multiple copies of it (one per host), each copy receiving a fraction of the batch (Section 4.4). In this configuration, the provers use the Open MPI [9] message-passing library to synchronize and exchange data.

To further reduce latency, each prover offloads work to a GPU (see also [120] for an independent study of GPU hardware in the context of [53]). We exploit three levels of parallelism here. First, the prover performs a ciphertext operation for each component in the commitment vector (§4.3); each operation is (to first approximation) separate. Second, each operation computes two independent modular exponentiations (the ciphertext of an ElGamal encryption has two elements). Third, modular exponentiation itself admits a parallel implementation (each input is a multiprecision number encoded in multiple machine words). Thus, in our GPU implementation, a group of CUDA [3] threads computes each exponentiation.

We also parallelize the verifier's encryption work during the commitment phase (§4.3), using the approach above plus an optimization: the verifier's exponentiations are fixed base [95, Chapter 14.6], letting Zaatar memoize intermediate squares. As another optimization, we implement simultaneous multiple exponentiation [95, Chapter 14.6], which accelerates the prover.[6]

We implement exponentiations for the prover and verifier with the `libgpucrypto` library of SSLShader [80], modified to support the memoization.

An optimization. We optimize the prover in Zaatar by constructing the proof vector u (§4.6) using the fast Fourier transform (FFT), a suggestion of GGPR [63]. As a result, $\mathcal{P}$'s per-instance running time drops from $O(|Z| + |\mathcal{C}| \cdot \log^2 |\mathcal{C}|)$ (Figure C.2) to $O(|Z| + |\mathcal{C}| \cdot \log |\mathcal{C}|)$.

4.8 Evaluation

In this section, our goal is to assess the effect of Zaatar's algorithmic refinements and systems engineering techniques (§4.2–4.7) on end-to-end performance. We do that by answering the following questions: (1) What is the effect of Zaatar's refinements on the costs of the prover and the verifier? (2) What are the costs of the prover and the verifier compared to simply executing a computation? and (3) What is the effect of parallelizing the prover? In addition, this section discusses the expressiveness of constraints and the limitations of Zaatar itself (§4.8.4).

Benchmark computations. To answer the questions above, we use a set of benchmark computations: (a) matrix multiplication, (b) polynomial evaluation, (c) root finding via bisection [114, Figure 18], (d) Partitioning Around Medoids (PAM) clustering [121], (e) Floyd-Warshall all-pairs shortest paths [52], and (f) the longest common subsequence (LCS) problem. Computations (a), (d), and (f) use 32-bit signed integers as inputs. Computation (c) uses *rational number* inputs with 32-bit numerators, 5-bit denominators. Computations (b) and (e) also have rational inputs, with 32-bit numerators, 32-bit denominators. We use a finite field with a prime modulus of size 128 bits for the integer computations and size 220 bits for the rational number computations. The details of rational number handling and representation are given in Appendix B.1.

Metrics and setup. We measure latency and computing cycles used by Zaatar's prover and verifier. We report the prover's costs, including the CPU time to execute the computation and to participate in the verification protocol. For the verifier, we report the *cross-over point*, β^*:

[6]Although the last optimization is well-known, we were inspired by other works that implement it [74, 89, 104].

the number of instances of a computation at which the verifier's total costs equals the cost of executing a batch locally. Note that this quantity captures only the point at which the verifier is better off verifying a batch versus executing the batch; this quantity does not take into account the prover's CPU costs or the network costs (we evaluate Zaatar's cross-over points for network costs in Section 5.7). We measure local computation using implementations built on the GMP library.[7]

For each result that we report, we run at least three experiments and take averages (the standard deviations are always within 5% of the means). We measure CPU time using `getrusage` and latency using PAPI's real time counter [11]. Our experiments use the Longhorn cluster at the Texas Advanced Computing Center (TACC). Each machine is configured identically and runs Linux on an Intel Xeon processor E5540 2.53 GHz with 48GB of RAM. Experiments with GPUs use machines with an NVIDIA Quadro FX 5800. Each GPU has 240 CUDA cores and 4GB of memory.

4.8.1 Effect of Zaatar's refinements on end-to-end costs

To understand the effect of Zaatar's algorithmic refinements on the end-to-end performance of the verifier and the prover, we consider a series of baseline systems that contain a subset of Zaatar's refinements. We cannot experiment with those baselines as they are too expensive to run on real hardware. Therefore, we predict their performance using our cost models (Figures 4.3 and C.2), which we now validate. We run microbenchmarks to quantify the model's parameters. We run a program that executes each operation in the cost model 1000 times and report the average CPU time, for two field sizes (standard deviations are within 5% of the means). The results are immediately below:[8]

field size	e	d	h	f_{lazy}	f	f_{div}	c
128 bits	65 μs	170 μs	91 μs	68 ns	210 ns	2 μs	160 ns
220 bits	88 μs	170 μs	130 μs	90 ns	320 ns	3 μs	260 ns

We use our cost model to validate our experimental results for Zaatar; we find that the empirical CPU costs are 5-15% larger than the model's predictions. Thus, we use our cost

[7]This is an optimistic baseline as the use of GMP's big number arithmetic increases the cost of local computation. However, without such a baseline, Zaatar's $\mathcal{V}$ will not beat local computation (at input sizes, for our benchmarks, that we can currently experiment with), and hence we will not be able to report β^*. Nonetheless, Pantry (Chapter 5) reduces the costs of $\mathcal{V}$ further to a point where $\mathcal{V}$ can beat an optimized native computation baseline at an input size that we can experiment with; Section 5.7.2 provides additional details.

[8]The f_{lazy} parameter is not explicitly in Figure C.2, but some of the instances of f in the figure should be read as f_{lazy}, which is the cost of a field multiplication that does not require applying "mod p".

model as a reasonable proxy for actual experimental results; we predict cross-over points, and the costs to the prover and the verifier at those cross-over points, under the baseline systems and Zaatar.

Figure 4.6 depicts the estimated cross-over points, and the costs of the prover and the verifier (under the aforementioned baseline systems and Zaatar) for matrix multiplication at various input sizes. Observe that Zaatar's refinements dramatically reduce the costs of the protocol entities and the cross-over points.

4.8.2 Cost of verifiable execution and the need for batching

We compare the prover's costs and the verifier's under Zaatar to the cost of running the computation locally. To do so, we experiment with the above benchmark computations at the following input sizes: (a) matrix multiplication with $m = 128$, (b) polynomial evaluation with $m = 512$, (c) root finding for degree-2 polynomials with m=256 variables and L=8 iterations, (d) PAM clustering with 2560 data points (m=20 samples with d=128 dimensions clustered into two groups), (e) Floyd-Warshall with m=25 nodes, and (f) LCS between two strings of length m=300.

Figure 4.7 summarizes the results. The prover in Zaatar is substantially slower than local computation, and the verifier incurs a large setup cost. However, the verifier's per-instance costs (the cost to verify an instance of a computation) is less than the cost to execute the computation locally, so as described in §4.4 and quantified in Figure 4.6, the verifier can gain from outsourcing a batch of instances of the same computation with potentially different inputs.

4.8.3 Effect of parallelization on the latency of the prover

The previous subsection established that the computational burden is still heavy for Zaatar's prover. However, the latency can be tamed. Specifically, we expect that (a) hardware acceleration (e.g., GPUs) reduces latency per-instance, and (b) distributing the prover over more machines makes the latency of a batch not much greater than the latency of a single instance. We experiment by running Zaatar under various hardware configurations (multiple machines, GPUs, etc.), measuring the latency at the verifier. Figure 4.8 depicts the results. GPU acceleration improves per-instance latency by roughly 20%, and distribution indeed achieves near-linear speedup.

	computation (Ψ)	batching (§4.4)	alg. constraints (§4.2)	new commit (§4.3)	new PCPs (§4.6)
β^* (cross-over point)	matrix multiplication, $m = 512$	Never	$3.2 \cdot 10^9$	$3.6 \cdot 10^6$	3500
	matrix multiplication, $m = 1024$	Never	$4.7 \cdot 10^9$	$5.4 \cdot 10^6$	2600
	matrix multiplication, $m = 2048$	Never	$8.2 \cdot 10^9$	$9.5 \cdot 10^6$	2300
$\mathcal{V}$'s computation time at β^*	matrix multiplication, $m = 512$	Never	3000 yr	3.3 yr	28 hr
	matrix multiplication, $m = 1024$	Never	$3.4 \cdot 10^4$ yr	40 yr	7.1 days
	matrix multiplication, $m = 2048$	Never	$4.8 \cdot 10^5$ yr	560 yr	50 days
$\mathcal{P}$'s computation time at β^*	matrix multiplication, $m = 512$	Never	$7.4 \cdot 10^{12}$ yr	$9.5 \cdot 10^6$ yr	9.0 yr
	matrix multiplication, $m = 1024$	Never	$1.7 \cdot 10^{14}$ yr	$2.3 \cdot 10^8$ yr	54 yr
	matrix multiplication, $m = 2048$	Never	$4.9 \cdot 10^{15}$ yr	$6.4 \cdot 10^9$ yr	380 yr

Figure 4.6: Cross-over points (β^*) and running times at those points under a subset of Zaatar's refinements, to two significant figures. Batching is required to make outsourcing profitable for $\mathcal{V}$; without our refinements, the batch size is astronomical. Solving for m^* using more reasonable values of β also yields extremely large results. Our refinements reduce these numbers significantly. Both $\mathcal{V}$ and $\mathcal{P}$ can be parallelized to reduce latency; Figure 4.8 describes the effect of parallelizing $\mathcal{P}$.

		$\mathcal{P}$'s costs			$\mathcal{V}$'s costs	
computation (Ψ)	local	solve	argue	e2e CPU time	setup	per-instance
matrix multiplication ($m = 128$)	139.4 ms	12.6 s	16.9 min	17.1 min	7.9 min	32.3 ms
polynomial evaluation ($m = 512$)	288.2 ms	2.1 s	3.2 min	3.2 min	1.3 min	1.7 ms
root finding by bisection ($m = 256, L = 8$)	0.8 s	6.2 s	6.3 min	6.5 min	2.7 min	2.2 ms
PAM clustering ($m = 20, d = 128$)	50.9 ms	8.5 s	8.6 min	8.7 min	4.3 min	3.2 ms
all-pairs shortest path ($m = 25$)	8.2 ms	1.4 s	8.9 min	8.9 min	4.9 min	1.4 ms
longest common subsequence ($m = 300$)	1.5 ms	13.4 s	18.0 min	18.3 min	9.2 min	1.2 ms

Figure 4.7: Per-instance cost of the Zaatar prover and Zaatar verifier compared to the baseline of local computation (executed with the GMP library [4]), under various computations. The "e2e CPU time" is decomposed into its contributions ("solve" refers to the cost of step ② and "argue" refers to the cost of step ③ in Figure 4.1). The end-to-end running time of Zaatar's prover is far more than the cost to execute the computation. However, the costs do not scale up: when batching computations, the latency of the batch is roughly equal to the latency of an instance; see Figure 4.8. The "setup" column under $\mathcal{V}$'s costs refers to the cost of query generation that Zaatar amortizes (as described in 4.4), and the per-instance column refers to cost of verifying an instance of a computation in a batch and it scales with the batch size.

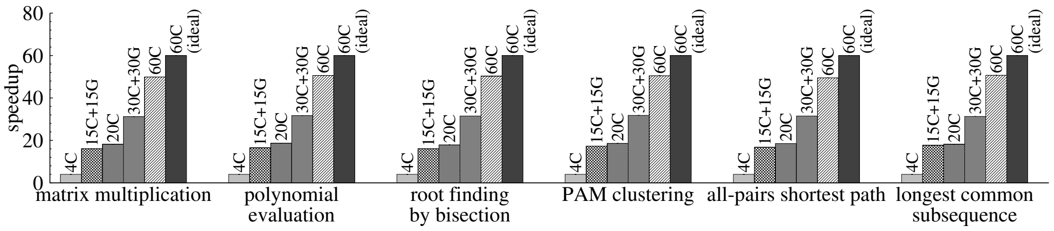

Figure 4.8: Speedups from parallelizing and distributing the prover. We run with m=100 for matrix multiplication, $m = 256$ for polynomial evaluation, $m = 25, L = 8$ for root finding by bisection, m=10, d=128 for PAM clustering, m=100 for longest common subsequence, and m=15 for all-pairs shortest paths. We use $\beta = 60$ in all cases. Configurations are denoted with bar labels; for example, 4C means 4 CPU cores, and 15C+15G means 15 CPU cores with 15 GPUs. GPU acceleration improves per-instance latency by about 20%, and Zaatar's prover achieves near-linear speedup as it gets more hardware resources.

4.8.4 Applicability of constraints and limitations of Zaatar

This section has established that (a) Zaatar produces vast performance improvements over a naive implementation of the protocol of Ishai et al. [79], (b) Zaatar's verifier wins only after batching many computations, and (c) Zaatar's prover is substantially more expensive than simply executing the computation. Points (b) and (c) are consequences of the underlying machinery, specifically that (i) the computation must be encoded as constraints and (ii) the verification machinery brings intrinsic costs. Below, we delve into (i) followed by (ii).

Concerning the constraints formalism, it is, on the one hand, expressive. Indeed, degree-2 constraints can represent any computation that terminates in polynomial time (this is implied by Pippenger and Fischer's result [107]). That is, in principle any program for which an upper-bound on running time can be established at compile time can be represented. On the other hand, the constraints formalism imposes undesirable costs. These costs vary depending on the program construct and computation. Straight-line operations (e.g., integer additions and multiplications) translate directly and efficiently into constraints. Comparisons, by contrast, require $O(\log|\mathbb{F}|)$ constraints (see Section 4.5). Worse, under natural translations of computations, indirect memory accesses (for instance, array indices that are not known at compile time) produce an excessive number of constraints.

Concerning the verification machinery (that is, even given a program already expressed in constraints), there are several limitations and overheads. First, verification requires touching each input and output, so the client saves CPU cycles only when outsourcing computations that take time superlinear in the input size. Second, the sheer size of the queries introduces a substantial setup cost for the verifier; the batched model (§4.4) addresses this cost but the verifier "breaks even" only when it has enough instances to batch. Third, the proof encoding introduces overhead for the prover. Finally, the cryptographic operations are a burden on the verifier and prover, particularly the prover.

4.9 Summary

Zaatar considerably expands the applicability of probabilistically checkable proofs (PCPs) and efficient arguments for verifiable computation. To do so, it incorporates new theoretical refinements and systems engineering techniques, which improve the performance of a strand of theory by over 20 orders of magnitude.

Despite these dramatic speedups, Zaatar has several limitations. First, verifiable execution is still expensive by multiple orders of magnitude compared to an unverifiable native execution, because the computation must be encoded as constraints and because of intrinsic costs of the protocol. Second, the verifier incurs a setup cost that needs to be amortized by outsourcing multiple identical computations.

Nevertheless, we expect assurance to have price, and indeed, there are regimes in which Zaatar's costs may not be ridiculous. As an example, consider outsourcing data-parallel computations in the cloud. This setup has (a) an abundance of cheap computing power (i.e., the prover's overheads might be tolerable), and (b) a computation structure that precisely matches the amortization requirement of Zaatar's verifier.

However, realizing the above application requires addressing a technical problem: Zaatar supports only stateless computations, owing to the constraints formalism, but data-parallel computations in the cloud often presume remote inputs (e.g., MapReduce jobs compute over vast data sets that live in the cloud). Chapter 5 describes how Pantry addresses this problem.

Chapter 5

Pantry: Verifying stateful computations

The theoretical underpinnings of Zaatar (Chapter 4) were thought to be wildly impractical several years ago, even for simple applications. But, Zaatar dramatically changes the landscape of verifiable computation: it reduces the resource costs of verifiable execution by over 20 orders of magnitude, and it significantly broadens the space of computations to which the theory applies. Additionally, as mentioned in Chapters 1 and 2, Zaatar has many contemporaries: (1) CMT [53, 120], Allspice [123], Thaler [119], (2) GGPR, Pinocchio [63, 104], and (3) TinyRAM [25, 27]. Like Zaatar, these systems appear to approach practicality: (1) several of them include compilers that allow programmers to express computations in a high-level language [27, 104, 123], and (2) 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.

However, almost none of these systems admit a notion of state or storage:[1] their compilation target is *constraints* (§4.5, §5.1). 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 chapter describes Pantry, the first verifiable computation system to support a notion of state. To do so, Pantry composes the machinery for verifying stateless computations, Zaatar (Chapter 4) and Pinocchio [104], with techniques from untrusted storage [40, 60, 88, 96]. While this picture is folklore among theorists [25, 37, 63, 78], the con-

[1]The exception is the work of Ben-Sasson et al. [27], which supports a notion of volatile state; see Chapter 2.

tributions of Pantry are to work out the details and build a system. In more detail, Pantry makes the following contributions.

(1) Pantry enhances its predecessor systems for verifiable computation, Zaatar (Chapter 4) and Pinocchio [104], with a storage abstraction (§5.2). 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 [60, 88]; 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 (§5.3). 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 [40, 96]) to build higher-level storage abstractions: a RAM and a searchable tree (§5.4). 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 [63, 104], to build applications in which the prover's state is private: face matching, toll collection, etc. (§5.5).

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 (§5.7) of the implementation (§5.6) 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 the aforementioned prior systems for verifiable computation, 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 (Chapter 6). 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 verifiable computation to real applications of cloud computing (albeit at much smaller scales for now).

5.1 Pantry's base: Zaatar and Pinocchio

Pantry extends Zaatar (Chapter 4) and Pinocchio [104] (as mentioned earlier). We now present these baseline systems and their underlying theory in a unified framework, to emphasize similarities.

5.1.1 Overview of Zaatar 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 use randomness 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 $\Pr\{V \text{ rejects}\} > 1 - \epsilon$, 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 (in Zaatar) or indefinitely (in Pinocchio [104]). Section 5.1.3 provides details of this amortization behavior.

5.1.2 Zaatar and Pinocchio in more detail

As in Section 4.1, verifiably outsourcing a computation happens in three steps, depicted in Figure 5.1. 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 (§5.1.3 revisits).

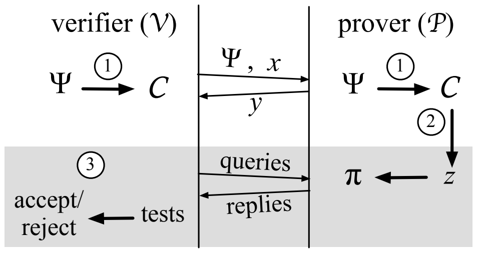

Figure 5.1: Verifiable outsourcing in Zaatar and Pinocchio, assuming a single instance of a computation Ψ on input x (amortization is depicted in Figure 5.2). Step ①: $\mathcal{V}$ and $\mathcal{P}$ compile Ψ from a high-level language to constraints $\mathcal{C}$. Step ②: $\mathcal{P}$ produces a satisfying assignment, z, to $\mathcal{C}(X{=}x, Y{=}y)$. Step ③: $\mathcal{P}$ uses complexity-theoretic and cryptographic machinery to convince $\mathcal{V}$ that $\mathcal{P}$ holds a satisfying assignment.

(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 §5.1.4) and invoking a compiler. Here, we focus on the compilation target: *a set of constraints* (Section 4.5).

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

As described in Section 4.5, recall that we use $\mathcal{C}(X{=}x)$ to mean $\mathcal{C}$ with X bound to x ($\mathcal{V}$'s requested input); $\mathcal{C}(X{=}x, Y{=}y)$ indicates that in addition Y is bound to y (the purported output). Notice that $\mathcal{C}(X{=}x, Y{=}y)$ is a set of constraints over the variables Z. If for some z, setting $Z{=}z$ makes all constraints in $\mathcal{C}(X{=}x, Y{=}y)$ hold simultaneously, then $\mathcal{C}(X{=}x, Y{=}y)$ is said to be *satisfiable*, and z is a *satisfying assignment*.

Furthermore, recall that for a given computation Ψ, a set of constraints $\mathcal{C}$ is said to be *equivalent* to Ψ if: for all x, y, we have $y = \Psi(x)$ if and only if $\mathcal{C}(X{=}x, Y{=}y)$ is satisfiable. As a simple example, the constraints $\mathcal{C}{=}\{Z - X = 0,\ Z + 1 - Y = 0\}$ are equivalent to add-1 [44]. Indeed, consider a pair (x, y). If $y = x + 1$, then there is a satisfying assignment to $\mathcal{C}(X{=}x, Y{=}y)$, namely $Z{=}x$. However, if $y \neq x + 1$, then $\mathcal{C}(X{=}x, Y{=}y)$ 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 $\mathcal{C}(X{=}x)$, 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., $\{\ldots, Z_4 = Z_3 \cdot (Z_2 + Z_1), Z_5 = Z_4 \cdot Z_2, \ldots\}$); 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 5.2). Consider the following snippet:

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

This compiles to the following constraints [44]:

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

Notice 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 5.2.

(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 $\mathcal{C}(X{=}x, Y{=}y)$; 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, Zaatar and Pinocchio apply PCPs [14, 15],[2] which, as described in Section 3.2.1, 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.

Furthermore, as described in Section 3.2, PCPs alone are not sufficient: 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, Zaatar 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 (by extending the compiler described in §4.7).

First, as described in Chapter 4, Zaatar instantiates an efficient argument [43, 79, 83]: $\mathcal{V}$ 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 satisfying assignment exists.

The second variant is instantiated by Pinocchio [104] and known as a *non-interactive argument* [63, 67]: $\mathcal{V}$ *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 [39, 63, 104].

5.1.3 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 Zaatar and Pinocchio.

In Zaatar, amortization happens over a *batch*: a set of β instances of the identical computation Ψ, on different inputs (Figure 5.2(a), §4.4). Thus, Zaatar presumes parallelism: for $j \in \{1, \ldots, \beta\}$, $\mathcal{V}$ 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 V extract commitments to all $\pi^{(j)}$ before issuing the queries (because queries are

[2]Our description takes some expositional license: Pinocchio's explicit base is GGPR [63], which does not invoke PCPs. However, one can regard the *key* in their work as PCP queries, in encoded form [39].

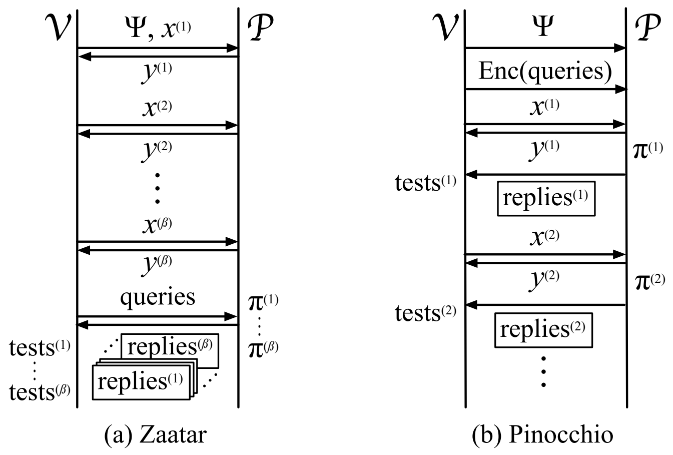

Figure 5.2: Amortization in Zaatar and Pinocchio. Superscripts denote different instances. In Zaatar, $\mathcal{V}$'s work to formulate queries amortizes over a batch of β instances; in Pinocchio, analogous work amortizes over all future instances of the same computation (this is better). In both protocols, the $\Psi \to \mathcal{C}$ step happens only once for each Ψ (not depicted).

reused across the batch). Note that $\mathcal{P}$ is an abstraction and could represent multiple machines (as in our MapReduce application in Section 5.3). Zaatar meets the completeness and soundness properties given earlier (§5.1.1), with $\epsilon < 1/10^6$ (see Appendix C.3), and in addition provides soundness for the batch: if for any $j \in \{1, \ldots, \beta\}$, $y^{(j)} \neq \Psi(x^{(j)})$, then $\Pr\{V \text{ rejects the batch}\} > 1 - \epsilon$.

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 5.2(b)). 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 [104]; the former provides a crucial foundation for Pantry's privacy-preserving applications (§5.5).

Figure 5.3 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 [62].*[3] Performance is thus evaluated as follows (Chapter 4, [104, 123]). (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

[3]One might think to compare to replicated execution (§1), but a goal of verifiable computation is to provide very strong guarantees (§5.1.1); replication stops working when faults are correlated.

	naive	**Zaatar [112], Pinocchio [104]**
$\mathcal{V}$, setup	0	$c_2 \cdot (\lvert Z\rvert + \lvert \mathcal{C}\rvert)$
$\mathcal{V}$, runtime	$\beta \cdot (T(\lvert x\rvert) + c_1\lvert y\rvert)$	$\beta \cdot (c_3 + c_4 \cdot (\lvert x\rvert + \lvert y\rvert))$
$\mathcal{P}$, runtime	0	$\beta \cdot (c_5 \cdot (\lvert Z\rvert + \lvert \mathcal{C}\rvert) + c_6 \cdot \lvert \mathcal{C}\rvert \cdot \log \lvert \mathcal{C}\rvert)$

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
$c_1, c_2, \ldots$: model costs of processing input/output, cryptographic primitives, PCP queries, etc.

Figure 5.3: CPU costs of step (3) under Zaatar and Pinocchio, and under the naive approach: reexecute and compare. The amortization behavior is different for Zaatar and Pinocchio (see text). Also, the constants $(c_2, c_3, \ldots)$ differ: Pinocchio's c_4 is lower while for the other constants, Zaatar's values are lower. Section 5.7.1 discusses these constants, the magnitudes of $|Z|$ and $|\mathcal{C}|$, and the costs of step (2). The last column in this table simplifies the Zaatar column in Figure C.2.

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 Sections 4.8 and 5.7). For question (1), the answer is "sometimes; it depends on the computation". For (2), the cross-over points are tens of thousands or millions [123, §6.3], 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 Zaatar and Pinocchio, Pinocchio has superior amortization behavior (see above) but higher proving and setup costs (and hence higher cross-over points), by constant factors.

5.1.4 Expressiveness

As context for Pantry, we now describe the language features and limitations of its baseline systems.

Pre-Pantry, compilers accepted programs in a high-level language (a C subset [104] or SFDL [92]) that includes functions, structs, 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. There are four exceptions. The first two are inequalities and bitwise operations; these constructs separate numbers into their bits and glue them back together (Section 4.5, [44, 104]), 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.

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.

5.2 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 5.3–5.5 describe their use.

To explain Pantry's approach, we note that the interface to step (3) in Section 5.1.2 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 [40, 96], often accompanied by a naming scheme in which data blocks are referenced by hashes of their contents [60, 88].

If we could efficiently represent the computation of the hash function as constraints, then we could extend the computational model in Section 5.1 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 5.1.2 to prove to $\mathcal{V}$ that $\mathcal{P}$ holds such an assignment. We now describe this approach.

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

5.2.1 Verifiable blocks: overview

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

$$S\colon name \rightarrow block \cup \bot,$$

where if $block = S(name)$, then $H(block) = name$. In other words, S implements the relation H^{-1}. This naming scheme allows clients to use untrusted storage servers [60, 88]. 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.

5.2.2 Verifiable blocks: details and costs

Pantry provides two primitives to the programmer:

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

These primitives are detailed in Figure 5.4. Notice that in a correct execution, $H(\text{block})$=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.

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 (§5.3), 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 [76, 87]: 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 = 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 = 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 (§5.1.4). Instead, we use an *algebraic* hash function, due to Ajtai [12, 70] 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 (§5.1.2). 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 (§5.7.1).

5.2.3 Guarantees and non-guarantees

Appendices D.1 and D.2 describe the formal guarantees of Pantry; here we give an informal and heuristic explanation.

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 [116]. 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 5.4).

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 §5.1.2), 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 (Chapter 1).

5.3 Verifiable MapReduce

This section describes how Pantry provides verifiability for MapReduce jobs. We begin with a brief review of the standard MapReduce model [55].

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

```
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 5.5: For verifiable MapReduce, Pantry applies the verification machinery to the depicted functions, mapper() and reducer(), which use the storage primitives from § 5.2; their execution is verified in two batches.

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 (§5.2): 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 [60, 88].

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 (§5.1–§5.2), 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 (§5.1.3), 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 5.5; 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 (§5.1.2, 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 (§5.1.3).

When the mappers execute, each instance $j \in \{1, \ldots, M\}$ 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 $\{map\_out^{(j)}\}_{j=\{1,\ldots,M\}}$ are the $M \cdot R$ intermediate digests mentioned above.

The framework then supplies the inputs to the second phase, by shuffling the digests $\{map\_out^{(j)}\}_{j=\{1,\ldots,M\}}$ and regrouping them as $\{reduce\_in^{(j)}\}_{j=\{1,\ldots,R\}}$, where 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 $j \in \{1, \ldots, R\}$ 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.

	naive (local)	Pantry
CPU costs		
$\mathcal{V}$, setup	0	$c_2 \cdot (\lvert Z_{\text{mapper}}\rvert + \lvert \mathcal{C}_{\text{Mapper}}\rvert)$
$\mathcal{V}$, runtime	$M \cdot T_{\text{mapper}}(\lvert ch\rvert)$	$M \cdot (c_3 + c_4 \cdot \lvert d\rvert \cdot (R+1))$
network costs		
setup	0	$c_7 \cdot (\lvert Z_{\text{mapper}}\rvert + \lvert \mathcal{C}_{\text{Mapper}}\rvert)$
runtime	$M \cdot \lvert ch\rvert$	$M \cdot (c_8 + \lvert d\rvert \cdot (R+1))$

T_{mapper}: running time of a map instance M: # of mappers
$|ch|$: length of a mapper's input $|d|$: length of a digest

Figure 5.6: 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 5.3; 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.

Analysis. Figure 5.6 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 5.3). 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 (§5.1.3) occurs when $M \geq \frac{c_2 \cdot (|Z_{\text{mapper}}| + |\mathcal{C}_{\text{Mapper}}|)}{T_{\text{mapper}}(|ch|) - c_3 - c_4 \cdot |d| \cdot (R+1)}$, per Figure 5.6.

Pantry also saves the verifier network costs. This happens when $M \geq \frac{c_7 \cdot (|Z_{\text{mapper}}| + |\mathcal{C}_{\text{Mapper}}|)}{|ch| - c_8 - R \cdot |d|}$. 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. However, 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 Zaatar 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 (§5.1.3), 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 [55].

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 (§5.1.4); one consequence is that each mapper's chunk size must be identical and fixed at compile time, and likewise with the reducers.

5.4 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 (§5.2). 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 [40, 60, 88, 91, 96]. 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

```
Load(address $a$, digest $d$)
    $\ell \leftarrow \lceil \log N \rceil$
    $h \leftarrow d$
    for $i = 0$ to $\ell - 2$:
        $node \leftarrow \text{GetBlock}(h)$
        $x \leftarrow i$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 $d$)
    $path \leftarrow \text{LoadPath}(a, d)$
    $\ell \leftarrow \lceil \log N \rceil$
    $node \leftarrow path[\ell - 1]$
    $node.value \leftarrow v$
    $d' \leftarrow \text{PutBlock}(node)$
    for $i = \ell - 2$ to 0:
        $node \leftarrow path[i]$
        $x \leftarrow i$th bit of $a$
        if $x = 0$:
            $node.left \leftarrow d'$
        else:
            $node.right \leftarrow d'$
        $d' \leftarrow \text{PutBlock}(node)$
    return $d'$
```

Figure 5.7: RAM operations implemented with GetBlock and PutBlock, using a Merkle tree [96]. N is the number of addresses in memory.

that program to constraints, and apply the argument step (§5.1.2) to those constraints.

5.4.1 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 5.7.

The high-level idea behind this pseudocode is that the digest commits to the full state of memory [40, 96], 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 [40, 96]. 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 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 5.7 is simply applying this idea: the verifiable blocks in Section 5.2 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 5.7 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 $\mathcal{C}_{\text{Store}}$ denote the constraints that Store compiles to. To satisfy $\mathcal{C}_{\text{Store}}(X{=}(a,v,d), Y{=}d')$, $\mathcal{P}$ must (1) exhibit a path through the tree, to a, that is consistent with d, and (2) compute a new digest that is consistent with the old path and with the memory update. Thus, the constraints enforce the guarantee that Store promises.

Costs. We briefly describe the blowup from the constraint representation; Sections 5.1.2 and 5.3 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 (§5.2.2).

5.4.2 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 (§5.4.1) 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 §5.4.1). 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).

5.4.3 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 5.2.2 and 5.4.2, 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 (§5.2). 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 (§5.4.2), 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.

5.4.4 Compromises and limitations

A key compromise is that efficiency sometimes requires not using RAM and instead constructing data structures directly from verifiable pointers (§5.4.2, §5.4.3). One consequence is that the implementer of these data structures is directly exposed to the clumsiness of the constraint model (§5.1.4); 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.

5.5 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 [63, 104] provides most of the solution. Here, step (3) of Section 5.1.2 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 (§5.2), 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 (§5.4), 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* [69, §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 instantiate 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-SHA256 (see Appendix D.3 for details). Relative to the protocol of Pedersen [106], our scheme makes a stronger cryptographic assumption but saves an order of magnitude in constraint size. Of course, this scheme uses SHA-256,[4] so it is more expensive for us than Ajtai's function (§5.2.2), but the expense is incurred only once per execution (§5.7.1).

Applications. We build (§5.6) and evaluate (§5.7) 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 [102], 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 [108], which requires a custom protocol, while we require only a simple C program (§5.6). The third application is *regression analysis* (again inspired by prior work that requires a custom protocol [101]); 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

[4] Ajtai is unsuitable because it is not a pseudorandom function (PRF) and therefore would not hide the prover's beginning state.

best-fit parameters. The details of our applications are in Appendix D.4.

5.6 Implementation details

The Pantry implementation modifies the Zaatar compiler. The base compiler first transforms programs written in a high-level language (§5.1.4) 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 (§5.1.2). 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 (§5.1.2).

Pantry adds several conveniences to the base compiler. Following Pinocchio [104], the Pantry compiler accepts a subset of C (§5.1.4). More significantly, the compiler targets the Pinocchio and the Zaatar encodings, with a unified code base. The main work here was implementing Pinocchio's pairing-based cryptography, for which we use a public library [6, 36].

To implement GetBlock and PutBlock (§5.2), 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 S (see Figure 5.4); we implement S using the LevelDB key-value store [7].

The $\mathcal{C}_{H^{-1}}$ and $\mathcal{C}_H$ constraints implement H as (a variable-length version of) the Ajtai [12, 70] hash function. Using the notation in [70], this function hashes m bits into $n \cdot \log q$ bits. Based on the analysis in [98], 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 [82, Ch. 4.6.4] that prepends the input with its length [54].

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 (§5.3) requires 1500 lines of C++. The verifiable data structures (§5.4.1–§5.4.2) require 400 lines in Pantry's subset-of-C. The main component in the database application (§5.4.3) is a query-to-C translator, which we implement with 2000 lines of Java, on top of Cassandra's CQL parser [2]. Our private state appli-

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 neighbor search of m length-d vectors	MapReduce (Z)	$m \cdot d$
covariance matrix for m samples of dimension 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-dimensional records	Private state (P)	$m \cdot d^2 + d^3$

Figure 5.8: Sample applications in our experiments. The MapReduce applications use Zaatar (Z); the other two categories use Pinocchio (P). In the MapReduce applications (§5.3), Map and Reduce are roughly 60 lines, combined. The DB queries are expressed in Pantry's query framework (§5.4.3, §5.6). The private state applications (details and code size) are described in §5.5 and §5.6.

cations (§5.5) are 60 lines for face matching, 80 lines for tolling, and 143 lines for regression analysis.

5.7 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–§5.1), these alternatives must be general-purpose and not make restrictive hypotheses about failure classes. This often means comparing to naive verifiers (§5.1.3). However, we would be the first to admit that tailored protocols (of the kind cited in Chapter 2; an example is [118]) or replication are likely to far outperform Pantry.

Applications and setup. We experiment with a set of sample applications, listed in Figure 5.8. Additional parameters (for the cryptographic primitives in Zaatar and Pinocchio, etc.) are described in Appendix D.4.

Our experiments use a cluster of machines in the Stampede cluster at the Texas Advanced Computing Center (TACC). Each machine runs 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 access to a 14PB Lustre 2.1.3

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 5.9: 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 (§5.2.2). Store is shown in the same row as Load, and is twice as expensive (§5.4.1); 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.

parallel file system.

5.7.1 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 5.9 shows the number of constraints to which GetBlock and PutBlock (§5.2) 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 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

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

			$\lvert\mathcal{C}\rvert$ (millions)		prover ($\mathcal{P}$)			③ verifier ($\mathcal{V}$)	
computation (Ψ)	**input size**	**baseline**	**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 (§5.5)

Figure 5.10: 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 (§5.3), including a SHA-256 digest check for data integrity (§5.3); 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 $\lvert Z\rvert$ is not depicted but is roughly the same as $\lvert\mathcal{C}\rvert$ 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 §5.1.2; circled numbers refer to these steps.

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 (§5.4.1). Unfortunately, as shown in Figure 5.9, 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 (§5.4.2–§5.4.3); 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 5.10, 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), §5.1.2) is a cost for $\mathcal{P}$. We compute the ratio of solving time (Figure 5.10, the "solve" column) to $|\mathcal{C}|$ for each of our sample applications. This ratio ranges from 20 to 160 μs 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 (§5.2.2), which invokes many large integer arithmetic operations. (Another source of overhead here is that GetBlock / PutBlock operations incur I/O costs associated with accessing the block store.)

Arguing (step (3), §5.1.2) 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 $|\mathcal{C}|$ and $|Z|$, as indicated by the models given earlier (Figures 5.3 and 5.6). In these models, the largest constants are c_2, c_3, c_5 (representing cryptographic operations), and are on the order of 100μs. 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 and the relative number of bitwise constraints (small values reduce the costs of some of the cryptographic steps). We quantify the constants $\{c_i\}$ in Appendix D.4.

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

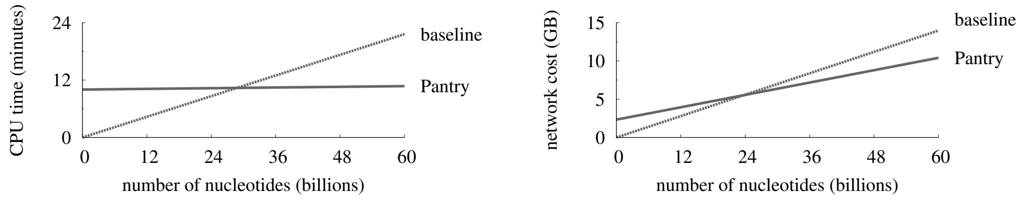

Figure 5.11: The verifier's CPU and network costs (extrapolated) as a function of job size for the nucleotide substring application in Figures 5.8 and 5.10 (each mapper gets a chunk of 600k nucleotides; one reducer is allocated per ten mappers). All y-intercepts (fixed costs) and slopes (per-instance 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.

The aforementioned costs can be understood by comparing to the cost of simply executing the computation (Figure 5.10, 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 $\mathcal{P}$).

5.7.2 All is not lost

Amidst the many appalling overheads in Figure 5.10, 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 (§5.1.3). 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 (§5.1.3, §5.3) 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 5.10) 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 5.11 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 (§5.7.1)—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 (§5.4.2). Below, we assess that cost, and ask about Pantry's ability to save resources for $\mathcal{V}$. What should the baseline be? In Figure 5.10, we present the running time of MySQL, which helps us gauge the prover's overhead. However, for 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 [60, 88] (akin to the map S in §5.2.1) 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 (§5.2.2). We have not built this alternative, so we estimate its network costs; we can do this since queries are highly constrained (§5.1.4, §5.4.4).

Figure 5.12 depicts the comparison, for a SELECT query. This table indicates, first, for the data set that we use, 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.

	Pantry	block store (estimated)
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 5.12: 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 (§5.4.3, §5.4.4).

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 5.10; 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 $|\mathcal{C}|$ and $|Z|$ (see Figures 5.3 and 5.6), $|\mathcal{C}|$ and $|Z|$ include terms for GetBlock's input (§5.2.2), and GetBlock's input is all of the state because there is no hash tree structure (§5.4). 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 (§5.1.2–§5.1.3).

5.8 Summary of Pantry

To summarize Pantry, it eliminates a major limitation of prior state-of-the-art systems by extending the machinery of its baseline systems (Zaatar and Pinocchio) with a notion of state. Furthermore, it mitigates the issue with the client's setup costs by enabling new applications (e.g., data-parallel computations over remotely stored inputs or programs that compute over the server's private state) for which the client's setup costs might be tolerable. However, Pantry retains a major limitation of its baseline systems: the prover is still too expensive (compared to simply executing a computation) to consider it practical for real. Nonetheless, Pantry expands verifiability to realistic applications of third party computing. The next chapter summarizes Pantry further along with a summary of the rest of the dissertation.

Chapter 6

Summary, limitations, and discussion

This dissertation began as an attempt to build a practical system in which a client can verifiably outsource its computations (and associated state) to an untrusted server (e.g., the cloud). A chief goal was to design a system that does not require *any* assumptions about the failure modes of the server, except, perhaps, cryptographic hardness assumptions. This led us to consider solutions based on theoretical constructs such as probabilistically checkable proofs (PCPs) and efficient arguments (it is worth noting that an early work on PCPs [17] posed the exact problem, in a theoretical context). While these solutions are applicable in principle, their costs were too large to be considered practical. Thus, a key challenge was to reduce the costs of the theory and build a usable system.

This dissertation does not completely achieve the above goal, but it reports significant progress. We describe two built systems, Zaatar and Pantry, that dramatically reduce costs and improve applicability of a strand of the aforementioned theoretical constructs (in the context of verifiable computation).

First, Zaatar slashes the costs of a PCP-based efficient argument protocol, using a combination of algorithmic improvements and systems engineering techniques (§4.3–4.6). Second, Zaatar includes a compiler to make this technology usable: it automatically transforms programs in a subset of C into executables that run verifiably (§4.7). Third, Pantry composes machinery for verifying stateless computations with techniques from untrusted storage to verifiably offload both computations and state. Fourth, using its techniques for handling remote state, Pantry extends verifiability to realistic applications of third party computing such as MapReduce jobs with remotely stored inputs, queries against remote databases, programs that compute over private state, etc. (§5.3–5.5). Fifth, Pantry drastically expands the scenarios in which the client can gain from outsourcing: (a) Since the client 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 client to beat naive verification even when outsourcing *linear*-time computations (§5.3, §5.7.2), (b) Since the client can save network costs compared to the naive alternative (§5.3, §5.7.2), Pantry may be beneficial even if verification costs more CPU cycles than local execution—a case that defeats the goals (§5.1.3) of prior work [27, 104, 123], and (c) Pantry (with a major assistance from Pinocchio) extends verifiability to a class of computations that the verifier cannot execute on its own, even in principle, as they involve *private* remote state (§5.5).

Limitations and next steps. Despite all of this progress, Zaatar and Pantry have many limitations. First, the client in Zaatar and Pantry incurs a setup cost (proportional to executing the computation once) that needs to be amortized by outsourcing multiple identical computations (§4.4). Pantry mitigates this limitation by identifying applications for which the client's setup costs may be tolerable (§5.3, §5.5), but eliminating the setup costs would be ideal. There is recent work to make this setup work independent of the computation [27], and also independent of the length of the computation [28]. However, a downside of these approaches is that the overheads for the server in these systems is several orders of magnitude more than the costs under Zaatar and Pantry. Thus, avoiding the client's setup costs *without* introducing undesirable overheads for the server remains an open problem in the area.

Second, the server's cost to verifiably execute a computation is still multiple orders of magnitude more than the cost of simply executing the computation (Figures 4.7 and 5.10). There are two factors that contribute to the high overheads of the server: (a) the requirement of the underlying theoretical constructs to represent a computation as a set of constraints (§4.5), and (b) the requirement of the argument protocol to execute several cryptographic operations for every step of the computation. Each of these factors impose up to 3 orders of magnitude overhead (in our benchmarks), which results in up to 6 orders of magnitude slow down compared to simply executing a computation. These high overheads limit our experiments (§5.7.2) to scales smaller than those of real applications (to put it mildly). Furthermore, in addition to costs it imposes, the constraints model is clumsy (§4.8, §5.1.4), which leads to various compromises described earlier (§5.3, §5.4.4, §5.5).

The aforementioned cost issues afflict the entire research area (§2.4). Key challenges are to reduce the overhead of the argument protocol (which seems possible, as the costs stem from high constants, not unfavorable asymptotics), perhaps by doing fewer cryptographic operations per step in the computation (a recent work does this for a class of computations [86]); use hardware acceleration to make the resource costs of verifiable execution comparable to those of naive replication; and go beyond, or around, the constraint model.

Discussion. Looking back to when we started a few years ago, none of the available theoretical constructs were implemented; indeed the estimated costs were too large to run even simple computations on real hardware. An immediate goal of ours was to transform this theoretical machinery to a point where we could run experiments, at least for a simple computation such as matrix multiplication. Following our early work in this area [111, 113], which articulated a research agenda to put strands of this theory to practice, several groups have produced different systems with similar motivations [25–29, 39, 45, 53, 63, 86, 104, 112, 115, 119, 120, 123].

The good news is that these systems make tremendous progress in some aspects: whereas the first systems in the area (Pepper [113], CMT [53]) supported only computations that are naturally represented as concise arithmetic circuits (e.g., matrix multiplication, polynomial evaluation), Pantry supports realistic applications of cloud computing (e.g., MapReduce jobs, database queries, etc.), and TinyRAM [27–29] even exposes a general machine abstraction. However, there is some bad news. Except for the work of Thaler [119], none of the works cited in the prior paragraph achieves better qualitative performance than Pepper and CMT do on matrix multiplication. That is, much of the work in this research area can be seen as broadening expressiveness, rather than improving performance.

Thus, as we look ahead, there is still a great deal of work remaining to bring argument systems and verifiable computation to practice—a point that has been emphasized throughout this dissertation. Nonetheless, the massive progress in the area suggests that a low overhead variant of these systems could be achievable down the road, and that such a variant could be a key tool in building secure systems.

Appendix A

Reducing linear PCPs to arguments, batching, and optimizations

A.1 A linear PCP

We state the queries, then the tests, then the statement of correctness of the PCP in [14, §5–6]. We use $\in_R$ to mean a uniformly random selection. The purpose of q_{10}, q_{12}, q_{14} below is *self-correction*; see [14, §5] or [100, §7.8.3] for details.

- Generate linearity queries: Select $q_1, q_2 \in_R \mathbb{F}^s$ and $q_4, q_5 \in_R \mathbb{F}^{s^2}$. Take $q_3 \leftarrow q_1 + q_2$ and $q_6 \leftarrow q_4 + q_5$.
- Generate quadratic correction queries: Select $q_7, q_8 \in_R \mathbb{F}^s$ and $q_{10} \in_R \mathbb{F}^{s^2}$. Take $q_9 \leftarrow (q_7 \otimes q_8 + q_{10})$.
- Generate circuit queries: Select $q_{12} \in_R \mathbb{F}^s$ and $q_{14} \in_R \mathbb{F}^{s^2}$. Take $q_{11} \leftarrow \gamma_1 + q_{12}$ and $q_{13} \leftarrow \gamma_2 + q_{14}$.
- Issue queries. Send queries $q_1, \ldots, q_{14}$ to oracle π, getting back $\pi(q_1), \ldots, \pi(q_{14})$.
- Linearity tests: Check that $\pi(q_1) + \pi(q_2) \overset{?}{=} \pi(q_3)$ and that $\pi(q_4) + \pi(q_5) \overset{?}{=} \pi(q_6)$. If not, `reject`.
- Quadratic correction test: Check that $\pi(q_7) \cdot \pi(q_8) \overset{?}{=} \pi(q_9) - \pi(q_{10})$. If not, `reject`.
- Circuit test: Check that $(\pi(q_{11}) - \pi(q_{12})) + (\pi(q_{13}) - \pi(q_{14})) \overset{?}{=} -\gamma_0$. If so, `accept`.

The $\gamma_0, \gamma_1, \gamma_2$ above are described in §3.2.1. The following lemmas rephrase Lemmas 6.2 and 6.3 from [14]:

Lemma A.1.1 (Completeness [14]). Assume $\mathcal{V}$ is given a satisfiable circuit $\mathcal{C}$. If π is constructed as in Section 3.2.1, and if $\mathcal{V}$ proceeds as above, then $\Pr\{\mathcal{V} \text{ accepts } \mathcal{C}\} = 1$. The probability is over $\mathcal{V}$'s random choices.

Lemma A.1.2 (Soundness [14]). There exists a constant $\kappa < 1$ such that if some proof oracle π passes all of the tests above on $\mathcal{C}$ with probability $> \kappa$, then $\mathcal{C}$ is satisfiable.

Applying the analysis in [14], we can take $\kappa > \max\{7/9,\ 4\delta + 2/|\mathbb{F}|,\ 4\delta + 1/|\mathbb{F}|\}$ for some δ such that $3\delta - 6\delta^2 > 2/9$. This ensures soundness for all three tests. Here, δ relates to linearity testing [41]; to justify the constraint on δ and its connection to the 7/9 floor on κ, see [22] and citations therein. Taking $\delta = \frac{1}{10}$, we can take $\kappa = 7/9 + \text{neg}$, where neg can be ignored; thus, for convenience, we assume $\kappa = 7/9$. Applying the lemma, we have that if the protocol is run $\rho = 70$ times and $\mathcal{C}$ is not satisfiable, $\mathcal{V}$ wrongly accepts with probability $\epsilon < \kappa^{\rho} < \epsilon_G$.

A.2 Reducing linear PCPs to arguments

This section first reduces PCPs to arguments directly, using Commit+Multidecommit (Figure 4.4); this will formalize Figure 4.2. The soundness of the reduction relies on Commit+Multidecommit actually binding the prover after the commit phase. Thus, the second (bulkier) part of the section defines a new and strengthened commitment protocol (Defn. A.2.1), proves that Commit+Multidecommit implements this protocol (Lemma A.2.1), and proves that any such protocol binds the prover in the way required by the reduction (Lemma A.2.2).

The reduction composes a PCP (such as the one in Appendix A.1) with Commit+Multidecommit. The protocol, theorem, and proof immediately below are almost entirely a syntactic substitution in [79, §4], replacing "MIP" with "PCP".

Given a linear PCP with soundness ϵ, the following is an argument system $(\mathcal{P}', \mathcal{V}')$. Assume that $(\mathcal{P}', \mathcal{V}')$ get a Boolean circuit $\mathcal{C}$ and that $\mathcal{P}'$ has a satisfying assignment.

1. $\mathcal{P}'$ and $\mathcal{V}'$ run Commit+Multidecommit's commit phase, causing $\mathcal{P}'$ to commit to a function, π.
2. $\mathcal{V}'$ runs the PCP verifier $\mathcal{V}$ on $\mathcal{C}$ to obtain $\mu = \ell \cdot \rho$ queries $q_1, \ldots, q_\mu$.
3. $\mathcal{P}'$ and $\mathcal{V}'$ run the decommit phase of Commit+Multidecommit. $\mathcal{V}'$ uses $q_1, \ldots, q_\mu$ as the queries. $\mathcal{V}'$ either rejects the decommitted output, or it treats the output as $\pi(q_1), \ldots, \pi(q_\mu)$. To these outputs, $\mathcal{V}'$ applies the PCP verifier $\mathcal{V}$. $\mathcal{V}'$ outputs `accept` or `reject` depending on what $\mathcal{V}$ would do.

Theorem A.2.1. Suppose $(\mathcal{P}, \mathcal{V})$ is a linear PCP with soundness ϵ. Then $(\mathcal{P}', \mathcal{V}')$ described above is an argument protocol with soundness $\epsilon' \leq \epsilon + 7.4 \cdot 10^{-12}$. ($7.4 \cdot 10^{-12}$ represents the error from the commitment protocol and will be filled in by Lemma A.2.2.)

Proof. Completeness follows from the PCP and the definition of Commit+Multidecommit. For soundness, Lemma A.2.2 below states that at the end of step 1 above, there is an extractor function that defines a single (possibly incorrect) oracle function $\tilde{\pi}$ such that, if $\mathcal{V}'$ didn't reject during decommit, then with all but probability $7.4 \cdot 10^{-12}$, the answers that $\mathcal{V}'$ gets in step 3 are $\tilde{\pi}(q_1), \ldots, \tilde{\pi}(q_\mu)$. But $(\mathcal{P}, \mathcal{V})$ has soundness ϵ, so the probability that $\mathcal{V}'$ accepts a non-satisfiable $\mathcal{C}$ is bounded by $\epsilon + 7.4 \cdot 10^{-12}$. □

We now strengthen the commitment primitive in Ishai et al. [79], borrowing their framework. We define a protocol: *commitment to a function with multiple decommitments* (CFMD). The sender is assumed to have a linear function, π, given by a vector, $w \in \mathbb{F}^n$; that is, $\pi(q) = \langle w, q \rangle$. In our context, w is normally $(z, z \otimes z)$. The receiver has μ queries, $q_1, q_2, \ldots, q_\mu \in \mathbb{F}^n$. For each query q_i, the receiver expects $\pi(q_i) = \langle w, q_i \rangle \in \mathbb{F}$.

Definition A.2.1 (Commitment to a function with multiple decommitments (CFMD)). Define a two-phase experiment between two probabilistic polynomial time actors (S, R) (a sender and receiver, which correspond to our prover and verifier) in an environment $\mathcal{E}$ that generates $\mathbb{F}$, w and $Q = (q_1, \ldots, q_\mu)$. In the first phase, the *commit phase*, S has w, and S and R interact, based on their random inputs. In the *decommit phase*, $\mathcal{E}$ gives Q to R, and S and R interact again, based on further random inputs. At the end of this second phase, R outputs $A = (a_1, \ldots, a_\mu) \in \mathbb{F}^\mu$ or $\perp$. A CFMD meets the following properties:

- **Correctness**. At the end of the decommit phase, R outputs $\pi(q_i) = \langle w, q_i \rangle$ (for all i), if S is honest.
- ϵ_B-**Binding**. Consider the following experiment. The environment $\mathcal{E}$ produces two (possibly distinct) μ-tuples of queries: $Q = (q_1, \ldots, q_\mu)$ and $\hat{Q} = (\hat{q}_1, \ldots, \hat{q}_\mu)$. R and a cheating S^* run the commit phase once and two independent instances of the decommit phase. In the two instances R presents the queries as Q and $\hat{Q}$, respectively. We say that S^* *wins* if R's outputs at the end of the respective decommit phases are $A = (a_1, \ldots, a_\mu)$ and $\hat{A} = (\hat{a}_1, \ldots, \hat{a}_\mu)$, and for some i, j, we have $q_i = \hat{q}_j$ but $a_i \neq \hat{a}_j$. We say that the protocol meets the ϵ_B-Binding property if for all $\mathcal{E}$ and for all efficient S^*, the probability of S^* winning is less than ϵ_B. The probability is taken over three sets of independent randomness: the commit phase and the two runnings of the decommit phase.

Informally, binding means that after the sender commits, it is very likely bound to a function from queries to answers.

Lemma A.2.1. Commit+Multidecommit (Figure 4.4, Section 4.3) is a CFMD protocol with $\epsilon_B = 1/|\mathbb{F}| + \epsilon_S$, where ϵ_S comes from the semantic security of the homomorphic encryption scheme.

Proof. Correctness: for an honest sender, $b = \pi(t) = \pi(r) + \sum_{i=1}^{\mu} \pi(\alpha_i \cdot q_i) = s + \sum_{i=1}^{\mu} \alpha_i \cdot \pi(q_i) = s + \sum_{i=1}^{\mu} \alpha_i \cdot a_i$, which implies that $b = s + \sum_{i=1}^{\mu} \alpha_i \cdot a_i$, and so verification passes, with the receiver outputting $\pi(q_1), \dots, \pi(q_\mu)$.

To show ϵ_B-binding, we will show that if S^* *can* systematically cheat, then an adversary $\mathcal{A}$ could use S^* to break the semantic security of the encryption scheme. Assume that Commit+Multidecommit does not meet ϵ_B-binding. Then there exists an efficient cheating sender S^* and an environment $\mathcal{E}$ producing $Q, \hat{Q}, i, j$ such that $q \triangleq q_i = \hat{q}_j$ and S^* can make R output $a_i \neq \hat{a}_j$ with probability $> \epsilon_B$.

We will construct an algorithm $\mathcal{A}$ that differentiates between $\alpha, \alpha' \in_R \mathbb{F}$ with probability more than $1/|\mathbb{F}| + \epsilon_S$ when given as input the following: a public key, pk; the encryption, $\mathsf{Enc}(pk, r)$, of r for a random vector $r \in \mathbb{F}^n$; $r + \alpha q$; and $r + \alpha' q$. This will contradict the semantic security of the encryption scheme. $\mathcal{A}$ has Q, q, i, j hard-wired (because $\mathcal{A}$ is working under environment $\mathcal{E}$) and works as follows:

(a) $\mathcal{A}$ gives S^* the input $(pk, \mathsf{Enc}(pk, r))$; $\mathcal{A}$ gets back e from S^* and ignores it.

(b) $\mathcal{A}$ randomly chooses $\alpha_1, \dots, \alpha_{i-1}, \alpha_{i+1}, \dots, \alpha_\mu$. It also randomly chooses $\hat{\alpha}_1, \dots, \hat{\alpha}_{j-1}, \hat{\alpha}_{j+1}, \dots, \hat{\alpha}_\mu$.

(c) $\mathcal{A}$ now leverages Q, q, i, j. $\mathcal{A}$ was given $r + \alpha q$, so it can construct $r + \alpha q + \sum_{k \in [\mu] \setminus i} \alpha_k q_k = r + Q \cdot \boldsymbol{\alpha}$, where $\boldsymbol{\alpha} = (\alpha_1, \dots, \alpha_{i-1}, \alpha, \alpha_{i+1}, \dots, \alpha_\mu)$. $\mathcal{A}$ gives S^* the input $(Q, r + Q \cdot \boldsymbol{\alpha})$; that is, $\mathcal{A}$ invokes S^* in the decommit phase. $\mathcal{A}$ gets back (A, b).

(d) Likewise, $\mathcal{A}$ constructs $r + \alpha' q + \sum_{k \in [\mu] \setminus j} \hat{\alpha}_k \hat{q}_k = r + \hat{Q} \cdot \hat{\boldsymbol{\alpha}}$, where $\hat{\boldsymbol{\alpha}} = (\hat{\alpha}_1, \dots, \hat{\alpha}_{j-1}, \alpha', \hat{\alpha}_{j+1}, \dots, \hat{\alpha}_\mu)$. $\mathcal{A}$ gives S^* $(\hat{Q}, r + \hat{Q} \cdot \hat{\boldsymbol{\alpha}})$, invoking S^* in the decommit phase again. $\mathcal{A}$ gets back $(\hat{A}, \hat{b})$.

When S^* wins (which it does with probability greater than $\epsilon_B = 1/|\mathbb{F}| + \epsilon_S$), $b = s + \boldsymbol{\alpha} \cdot A$ and $\hat{b} = s + \hat{\boldsymbol{\alpha}} \cdot \hat{A}$, but $a_i \neq \hat{a}_j$ (here, $\cdot$ represents the dot product). Now we will get two linear equations in two unknowns. The first is $\hat{b} - b = \hat{\boldsymbol{\alpha}} \cdot \hat{A} - \boldsymbol{\alpha} \cdot A$, which can be rewritten as: $\boxed{K_1 = \alpha' \hat{a}_j - \alpha a_i}$, where $\mathcal{A}$ can derive $K_1 = \hat{b} - b - \sum_{k \neq j} \hat{\alpha}_k \hat{a}_k + \sum_{k \neq i} \alpha_k a_k$. Now, let $t = r + Q \cdot \boldsymbol{\alpha}$ and let $\hat{t} = r + \hat{Q} \cdot \hat{\boldsymbol{\alpha}}$. To get the second equation, we start with $\hat{t} - t = \sum_{k \in [\mu] \setminus j} \hat{\alpha}_k \hat{q}_k - \sum_{k \in [\mu] \setminus i} \alpha_k q_k + \alpha' q - \alpha q$. This equation concerns a vector. We choose an index ℓ in the vector where q is not zero (if q is zero everywhere, then r is revealed). At that index, we have the following equation in scalars: $\boxed{K_2 = \alpha' - \alpha}$, where $\mathcal{A}$ can derive $K_2 = (\hat{t}^{(\ell)} - t^{(\ell)} - \sum_{k \neq j} \hat{\alpha}_k \hat{q}_k^{(\ell)} + \sum_{k \neq i} \alpha_k q_k^{(\ell)})/q^{(\ell)}$. Now $\mathcal{A}$ can solve for α (since the contrary hypothesis gave $a_i \neq \hat{a}_j$). □

We now prove that after the commit phase, the prover is effectively bound to a *single* function. Our articulation again follows [79], specifically their Lemmas 3.2 and 3.6.

Lemma A.2.2 (Existence of an extractor function). Let (S, R) be a CFMD protocol with binding error ϵ_B. Let $7.4 \cdot 10^{-12} = \mu \cdot 2 \cdot (2\sqrt[3]{9/2} + 1) \cdot \sqrt[3]{\epsilon_B}$. Let $v = (v_{S^*}, v_R)$ represent the views of S^* and R after the commit phase (v captures the randomness of the commit phase). For every efficient S^* and for every v, there exists a function $\tilde{f}_v : \mathbb{F}^n \to \mathbb{F}$ such that the following holds. For any environment $\mathcal{E}$, the output of R at the end of the decommit phase is, except with probability $7.4 \cdot 10^{-12}$, either $\perp$ or satisfies $a_i = \tilde{f}_v(q_i)$ for all $i \in [\mu]$, where $(q_1, \dots, q_\mu)$ are the decommitment queries generated by $\mathcal{E}$, and the probability is over the random inputs of S^* and R in both phases.

Proof. We will reuse the ideas in the proof of Lemma 3.2 in [79], but we must also ensure that q yields the same answer independent of its position and the other queries in the tuple. We begin with a definition: let $\mathsf{Ext}(v, q, i, \vec{q}) \triangleq \operatorname{argmax}_a A_v(q, i, \vec{q}, a)$, where $A_v(q, i, \vec{q}, a)$ equals, in view $v = (v_{S^*}, v_R)$, the probability over the randomness of the decommit phase that R's ith output is a when the query tuple is $\vec{q}$; note that q is the ith component of $\vec{q}$ and is included in $\mathsf{Ext}(\cdot)$ and $A_v(\cdot)$ for convenience. In other words, $\mathsf{Ext}(\cdot)$ is the most likely a_i value to be output by R, if the full tuple of queries is $\vec{q}$ and if q appears in the ith position. Note that, after the commit phase, $\mathsf{Ext}(\cdot)$ is given deterministically.

Claim A.2.3. Define $\epsilon_2 = (\sqrt[3]{9/2} + 1) \cdot \sqrt[3]{\epsilon_B}$. For all $\mathcal{E}$ producing $(q, i, j, \vec{q}_1, \vec{q}_2)$, where $\vec{q}_1$'s ith component is q and $\vec{q}_2$'s jth component is also q, we have the following with probability $> 1 - \epsilon_2$ over the commit phase: either $\mathsf{Ext}(v, q, i, \vec{q}_1) = \mathsf{Ext}(v, q, j, \vec{q}_2)$, or else the probability over the decommit phase of outputting $\perp$ is greater than $1 - \epsilon_2$.

Proof. Assume otherwise. Then there is an environment $\mathcal{E}$ producing $(q, i, j, \vec{q}_1, \vec{q}_2)$ such that with probability $> \epsilon_2$ over the commit phase, $\mathsf{Ext}(v, q, i, \vec{q}_1) \neq \mathsf{Ext}(v, q, j, \vec{q}_2)$ and with probability $> \epsilon_2$ over the decommit phase, R outputs something other than $\perp$. Define $\epsilon_1 = \sqrt[3]{\epsilon_B 9/2}$. We will show below that with probability $> 1 - \epsilon_1$ over the commit phase, $\sum_{a \in \mathbb{F} \setminus \mathsf{Ext}(v,q,i,\vec{q}_1)} A_v(q, i, \vec{q}_1, a)$ is $< \epsilon_1$. Thus, with probability $> \epsilon_2 - \epsilon_1$ over the commit phase, we have (1) the probability over the decommit phase is $> \epsilon_2 - \epsilon_1$ that R outputs $\mathsf{Ext}(v, q, i, \vec{q}_1)$ in the ith position (since R outputs *something* other than $\perp$ with probability $> \epsilon_2$, yet the probability of outputting anything *other* than $\mathsf{Ext}(v, q, i, \vec{q}_1)$ is $< \epsilon_1$); and (2) likewise, the probability of outputting $\mathsf{Ext}(v, q, j, \vec{q}_2)$ in the jth position is $> \epsilon_2 - \epsilon_1$. If we now take $Q = \vec{q}_1$ and $\hat{Q} = \vec{q}_2$, we have a contradiction to the definition of CFMD because with probability $> (\epsilon_2 - \epsilon_1)^3 = \epsilon_B$ over all three phases, $a_i \neq \hat{a}_j$, which generates a contradiction because the definition of ϵ_B-Binding says that this was supposed to happen with probability $< \epsilon_B$.

We must now show that, with probability $> 1 - \epsilon_1$ over the commit phase, $\sum_{a \in \mathbb{F} \setminus \mathsf{Ext}(v,q,i,\vec{q}_1)} A_v(q, i, \vec{q}_1, a)$ is $< \epsilon_1$. If not, then with probability $> \epsilon_1$ over the commit phase, $\sum_{a \in \mathbb{F} \setminus \mathsf{Ext}(v,q,i,\vec{q}_1)} A_v(q, i, \vec{q}_1, a) \geq \epsilon_1$. Now, following Lemma 3.2 in [79], we can partition $\mathbb{F}$ into two sets T_1, T_2 such that $\sum_{a \in T_1} A_v(q, i, \vec{q}_1, a)$ and $\sum_{a \in T_2} A_v(q, i, \vec{q}_1, a)$ are each greater than $\epsilon_1/3$. (There are two cases; consider $a^* = \mathsf{Ext}(v, q, i, \vec{q}_1)$. Either $A_v(q, i, \vec{q}_1, a^*)$ is greater than $\epsilon_1/3$, or it is not. If so, then the partition is $(a^*, \mathbb{F} \setminus a^*)$. If not, then there is still a partition because the sum of the $A_v(\cdot)$ is greater than $\epsilon_1/3$.) This implies that, in the binding experiment, R outputs values from the two partitions with probability $> (2/9) \cdot (\epsilon_1)^3 = \epsilon_B$ over all three phases, which contradicts the definition of a CFMD protocol. □

Now define $\mathsf{Ext}(v, q) = \mathsf{Ext}(v, q, i^*, \vec{q}^*)$, where i^* and $\vec{q}^*$ are designated (any choice with q in the i^* position of $\vec{q}^*$ will do). The next claim says that the response to q is independent of its position and the other queries.

Claim A.2.4. Let $\epsilon_3 = \epsilon_2 + \epsilon_1$. For all $\vec{q}, i$, where q is in the ith position of $\vec{q}$, we have that with probability $> 1 - \epsilon_3$ over the commit phase, either R's ith output is $\mathsf{Ext}(v, q)$, or else the probability over the decommit phase of outputting $\perp$ is $> 1 - \epsilon_3$.

Proof. Assume otherwise. Then there are $\vec{q}, i$ such that with probability $> \epsilon_3$ over the commit phase, the probability of outputting something (non-$\perp$) besides $\mathsf{Ext}(v, q, i^*, \vec{q}^*)$ is $> \epsilon_3$. But with probability $> 1 - \epsilon_1$ over the commit phase, $\sum_{a \in \mathbb{F} \setminus \mathsf{Ext}(v,q,i,\vec{q})} A_v(q, i, \vec{q}, a) < \epsilon_1$ (by the partitioning argument given in the proof of Claim A.2.3). Thus, with probability $> \epsilon_3 - \epsilon_1$ over the commit phase, the probability of outputting something (non-$\perp$) besides $\mathsf{Ext}(v, q, i^*, \vec{q}^*)$ is $> \epsilon_3$ and $\sum_{a \in \mathbb{F} \setminus \mathsf{Ext}(v,q,i,\vec{q})} A_v(q, i, \vec{q}, a) < \epsilon_1$. Thus, with probability $> \epsilon_3 - \epsilon_1 = \epsilon_2$ over the commit phase, $\mathsf{Ext}(v, q, i^*, \vec{q}^*) \neq \mathsf{Ext}(v, q, i, \vec{q})$ and R has $> \epsilon_3 > \epsilon_2$ probability of outputting something other than $\perp$. This contradicts Claim A.2.3. □

To complete the proof of the lemma, define $\tilde{f}_v(q) \triangleq \mathsf{Ext}(v, q)$. Consider the probability $7.4 \cdot 10^{-12}$, over both phases, that the output $A \neq (\tilde{f}_v(q_1), \ldots, \tilde{f}_v(q_\mu))$, i.e., that at least one of $a_i \neq \tilde{f}_v(q_i)$. By the union bound, $7.4 \cdot 10^{-12} < \sum_{i=1}^{\mu} \Pr\{a_i \neq \tilde{f}_v(q_i)\}$. By Claim A.2.4, $\Pr\{a_i \neq \tilde{f}_v(q_i)\} < \epsilon_3 + \epsilon_3$, since this bounds the probability that either of the two phases goes badly. Thus, $7.4 \cdot 10^{-12} < \mu \cdot 2 \cdot \epsilon_3$, as was to be shown. □

To compute $7.4 \cdot 10^{-12}$, we ignore ϵ_S (the homomorphic encryption error); i.e., we set $\epsilon_B = 1/|\mathbb{F}|$. We then get $7.4 \cdot 10^{-12} < 2000 \cdot (7\sqrt[3]{\epsilon_B}) < 2^{14} \cdot \sqrt[3]{1/|\mathbb{F}|}$. For $|\mathbb{F}| = 2^{128}$, $7.4 \cdot 10^{-12} < 2^{-28}$.

A.3 Batching

Under batching, $\mathcal{V}$ submits the same queries to β different proofs. Below, we sketch the mechanics and then proofs of correctness. First, we modify Commit+Multidecommit (Figure 4.4) to obtain a new protocol, called BatchedCommit+Multidecommit. The changes are as follows:

- $\mathcal{P}$ is regarded as holding a linear function $\vec{\pi}\colon \mathbb{F}^n \to \mathbb{F}^\beta$, so $\vec{\pi}(q) = (\pi_1(q), \ldots, \pi_\beta(q))$. One can visualize $\vec{\pi}$ as an $\beta \times n$ matrix, each of whose rows is an oracle, π_i. Thus, $\mathcal{P}$ returns vectors instead of scalars.
- Commit phase, steps 2 and 3: instead of receiving from $\mathcal{P}$ the scalar $\mathsf{Enc}(pk, \pi(r))$, $\mathcal{V}$ in fact receives a vector $\vec{e} = (\mathsf{Enc}(pk, \pi_1(r)), \ldots, \mathsf{Enc}(pk, \pi_\beta(r)))$ and decrypts to get $\vec{s} = (\pi_1(r), \ldots, \pi_\beta(r))$.
- Decommit phase, steps 5 and 6: $\mathcal{P}$ returns $\vec{a}_1, \ldots, \vec{a}_\mu, \vec{b}$, where $\vec{a}_i$ is supposed to equal $(\pi_1(q_i), \ldots, \pi_\beta(q_i))$ and $\vec{b}$ is supposed to equal $(\pi_1(t), \ldots, \pi_\beta(t))$; $\mathcal{V}$ checks that $\vec{b} = \vec{s} + \alpha_1\vec{a}_1 + \cdots + \alpha_\mu\vec{a}_\mu$.

Second, we modify the compilation in Appendix A.2 as follows. $\mathcal{P}'$ creates β linear proof oracles, and $\mathcal{V}'$ and $\mathcal{P}'$ run BatchedCommit+Multidecommit, causing $\mathcal{P}'$ to commit to a linear function $\vec{\pi}\colon \mathbb{F}^n \to \mathbb{F}^\beta$. $\mathcal{V}'$ then submits the μ PCP queries and receives vectors $\vec{\pi}(q_1), \ldots, \vec{\pi}(q_\mu)$ in response. Then $\mathcal{V}'$ runs the PCP verifier on each instance separately (e.g., for the kth instance, $\mathcal{V}'$ looks at the kth component of each of $\vec{\pi}(q_1), \ldots, \vec{\pi}(q_\mu)$). $\mathcal{V}'$ thus returns a vector of β `accept` or `reject` outputs. To argue correctness, we use a theorem analogous to Theorem A.2.1:

Theorem A.3.1. Under $(\mathcal{P}', \mathcal{V}')$ as described above, each of the β instances is an argument protocol with soundness $\epsilon' \leq \epsilon + 7.4 \cdot 10^{-12}$. ($7.4 \cdot 10^{-12}$ is defined in Appendix A.2.)

Proof. (Sketch.) Nearly the same as for Theorem A.2.1. We need an analog of Lemma A.2.2, described below. $\square$

This theorem says that if any of the β instances tries to encode a "proof" for an incorrect output, the probability that $\mathcal{V}$ outputs `accept` for that instance is bounded by ϵ'. This makes intuitive sense because if we fix a given instance, the probabilities should be unaffected by "extra" instances.

To formalize this intuition, we need BatchedCommit+Multidecommit to yield an extractor function for each of the β instances. To get there, we define a general protocol: *batch-CFMD*. This protocol has a binding property modified from the one in Definition A.2.1. In the new one, R gives stacked output $\vec{A} = (\vec{a}_1, \ldots, \vec{a}_\mu)$ and $\hat{\vec{A}} = (\hat{\vec{a}}_1, \ldots, \hat{\vec{a}}_\mu)$. The entries of $\vec{A}$ are denoted a_i^k; that is, $\vec{a}_i = (a_i^1, \ldots, a_i^\beta)$. We allow $a_i^k \in \{\mathbb{F} \cup \perp\}$ but require that $(a_1^k, \ldots, a_\mu^k) \in \mathbb{F}^\mu$ or $a_i^k = \perp$ for all $i \in [\mu]$. We now say that S^* wins if for some i, j, k, we have $q_i = \hat{q}_j$ and $a_i^k, \hat{a}_j^k \in \mathbb{F}$ but $a_i^k \neq \hat{a}_j^k$. We again say that the protocol meets ϵ_B-Binding if for all $\mathcal{E}$ and efficient S^*, S^* has less than ϵ_B probability of winning.

We can show that BatchedCommit+Multidecommit is a batch-CFMD protocol by re-running Lemma A.2.1, nearly unmodified. To complete the argument, we can establish an analog of Lemma A.2.2. The analog replaces a single extractor function $\tilde{f}_v$ with β functions $\tilde{f}_v^1, \ldots, \tilde{f}_v^\beta$, one per instance. The analog says that, viewing each instance k separately, we have with probability $> 1 - 7.4 \cdot 10^{-12}$ that either R's output for that instance is $\perp$ or else $a_i^k = \tilde{f}_v^k(q_i)$ for all $i \in [\mu]$. The proof is nearly the same as for Lemma A.2.2; the main difference is that $\mathsf{Ext}(\cdot)$ and $A_v(\cdot)$ receive a per-instance parameter k.

A.4 Modifications for ElGamal

Since ElGamal encryption is multiplicatively homomorphic (rather than additively homomorphic), small modifications to Commit+Multidecommit (Figure 4.4) and the soundness arguments are necessary. Below, we describe these modifications and establish that the results of Appendix A.2 still hold.

Fix the ElGamal group G, choose a generator g (known to both parties), and assume for now that $|\mathbb{F}_p| = |G|$ (we revisit this assumption below). Define the map $\psi\colon \mathbb{F}_p \to G$ by $x \mapsto g^x$. The map ψ is a group homomorphism and in fact an isomorphism; furthermore, ψ induces a ring structure on G. By composing ψ with ElGamal encryption, we get an additive homomorphism: $\mathsf{Enc}(pk, g^x)\mathsf{Enc}(pk, g^y) = \mathsf{Enc}(pk, g^{x+y})$. Of course, the verifier cannot recover $x + y$ explicitly from g^{x+y}, but this does not matter for Commit+Multidecommit. Also, given $\mathsf{Enc}(pk, g^x)$ and $a \in \mathbb{F}_p$, the properties of the ElGamal protocol imply that one can compute $\mathsf{Enc}(pk, g^{ax})$. Thus, given $(\mathsf{Enc}(pk, g^{r_1}), \ldots, \mathsf{Enc}(pk, g^{r_n}))$, one can compute $\mathsf{Enc}(pk, g^{\pi((r_1, \ldots, r_n))})$, where π is a linear function.

Therefore, we can modify Figure 4.4 as follows. First, during step 1, the verifier componentwise sends $\mathsf{Enc}(pk, g^r)$ rather than $\mathsf{Enc}(pk, r)$ to the prover. Next, in step 2, the prover computes $\mathsf{Enc}(pk, g^{\pi(r)})$ (without learning g^r), as described above. Then in step 3, V decrypts to get g^s. Finally, in step 6, the verifier checks that $g^b = g^{s+\alpha_1 a_1+\cdots+\alpha_\mu a_\mu}$.

We now need to check that Lemma A.2.1 still holds. Correctness applies, by inspection. Binding applies because the semantic security of the encryption scheme can be formulated in terms of $\mathcal{A}(pk, \mathsf{Enc}(pk, g^r), r + \alpha q, r + \alpha' q)$ and because $g^x = g^y$ if and only if $x = y$ (since ψ is injective).

Note that when $|G| > |\mathbb{F}_p|$, the same approach works, via modular arithmetic. Specifically, although ψ is no longer a group isomorphism, it is injective. Provided that the computations never overflow, i.e., result in values in the exponent larger than $|G|$, the protocol remains correct.

Appendix B

Broadening the space of computations

B.1 Signed integers, floating-point rationals

In this appendix and the two ahead, we describe how Zaatar applies to general-purpose computations. This appendix describes Zaatar's representation of signed integers and its representation of primitive floating-point quantities (this treatment expands on Section 4.5.1). The next appendix details the case study (from Section 4.5.2) of an inequality test and a conditional branch. Appendix B.3 describes other program constructs.

Our goal in these appendices is to show how to map computations to equivalent constraints over finite fields (according to the definition of equivalent given in Section 4.5). To do so, we follow the framework from Section 4.5. Recall that the three steps in that framework are: (C1) Bound the computation Ψ, which starts out over some domain D (such as $\mathbb{Z}$ or $\mathbb{Q}$), to ensure that Ψ stays within some set $U \subset D$; (C2) Establish a map between U and a finite field $\mathbb{F}$ such that computing Ψ in $\mathbb{F}$ is equivalent to computing Ψ in U. (C3) Transform the finite field version of the computation into constraints.

B.1.1 Signed integers

To illustrate step C1, consider $m \times m$ matrix multiplication over signed integers, with inputs of N bits (where the top bit is the sign): the computation does not "leave" $U = [-m \cdot 2^{2N-1}, m \cdot 2^{2N-1})$, where $U \subset \mathbb{Z}$.

For step C2, we take $\mathbb{F} = \mathbb{Z}/p$ and define θ between U and $\mathbb{Z}/p$ as follows:

$$\theta: U \to \mathbb{Z}/p$$
$$u \mapsto u \bmod p.$$

Note that:

(1) If U is an interval $[a, b]$ and $|\mathbb{Z}/p| > |U|$, then θ is 1-to-1.

(2) If $x_1, x_2 \in U$ and $x_1 + x_2 \in U$, then $\theta(x_1) + \theta(x_2) = \theta(x_1 + x_2)$.

(3) If $x_1, x_2 \in U$ and $x_1 x_2 \in U$, then $\theta(x_1)\theta(x_2) = \theta(x_1 x_2)$.

To argue property (1), take $x \bmod p = y \bmod p$, which means $x = y + p \cdot k$, for $k \in \mathbb{Z}$. If $x \neq y$ (so θ is not 1-to-1), then $|y - x| \geq p$, which implies that there must be at least $p + 1$ elements in U, since U is an interval that includes x and y. But $|U| < |\mathbb{Z}/p| = p$, a contradiction. Properties (2) and (3) follow because θ is a restriction of the usual reduction map, so it preserves addition and multiplication.

Thus, computation in $\mathbb{Z}/p$ is isomorphic to computation in U: the constraint representation uses only field operations to represent computations (see Section 4.5), and for the purposes of field operations, $\mathbb{Z}/p$ acts like U.

B.1.2 Floating-point rational numbers

Step C1

To illustrate this step, we again consider $m \times m$ matrix multiplication and this time require the input entries to be in the set $T = \{a/b : |a| \leq 2^{N_a}, b \in \{1, 2, 2^2, 2^3, \ldots, 2^{N_b}\}\}$. To bound the computation to a set U, we use the claim below.

Claim B.1.1. For the computation of matrix multiplication, with input entries restricted to T, the computation of matrix multiplication is restricted to $U = \{a/b : |a| < 2^{N'_a}, b \in \{1, 2, 2^2, 2^3, \ldots, 2^{2N_b}\}\}$, for $N'_a = 2N_a + 2N_b + \log_2 m$.

Proof. Consider an entry in the output; it is of the form $\sum_{k=1}^{m} A_{ik} B_{kj}$, where each $A_{ik} B_{kj}$ is contained in $S = \{a/b : |a| < 2^{2N_a}, b \in \{1, 2, 2^2, 2^3, \ldots, 2^{2N_b}\}\}$. Thus, we can write each output entry as $\sum_{k=1}^{m} a_k / b_k$, the sum of m numbers from the set S. Writing each b_k as 2^{e_k}, and letting $e^* = \max_k e_k$, we can write the sum as

$$\frac{\sum_k a_k 2^{e^* - e_k}}{2^{e^*}}.$$

The denominator of this sum is contained in $\{1, 2, 2^2, 2^3, \ldots, 2^{2N_b}\}$. The absolute value of each summand in the numerator, $a_k 2^{e^* - e_k}$, is no larger than $2^{2N_a+2N_b}$, and there are m summands, so the absolute value of the numerator is no larger than $m \cdot 2^{2N_a+2N_b} = 2^{2N_a+2N_b+\log_2 m}$. A fortiori, the intermediate sums are contained in U (they have fewer than m terms). $\square$

Step C2

We must identify a field $\mathbb{F}$ where the computation can be mapped; that is, we need a field that behaves something like $\mathbb{Q}$. For this purpose, we take $\mathbb{F} = \mathrm{Frac}(\mathbb{Z}/p)$, the quotient field of $\mathbb{Z}/p$, which we denote $\mathbb{Q}/p$.

This paragraph reviews the definition and properties of $\mathbb{Q}/p$ because we will need these details later. As a quotient field, $\mathbb{Q}/p$ is the set of *equivalence classes* on the set $\mathbb{Z}/p \times (\mathbb{Z}/p \setminus \{0\})$, under the equivalence relation $\sim$, where $(a, b) \sim (c, d)$ if $ad = bc \bmod p$; the field operations are $(a, b) + (c, d) = (ad + bc \bmod p, bd \bmod p)$ and $(a, b) \cdot (c, d) = (ac \bmod p, bd \bmod p)$, where a pair (x, y) represents its equivalence class. Note that although elements of $\mathbb{Q}/p$ are *represented* as having two components, each of which seems able to take p or $p - 1$ values, the cardinality of $\mathbb{Q}/p$ is only p. In fact, $\mathbb{Q}/p$ is isomorphic to $\mathbb{Z}/p$, via the map $f((a, b)) = a \cdot b^{-1}$.

We must now define a map from U to $\mathbb{Q}/p$; in doing so, we will take U to be an arbitrary subset of $\mathbb{Q}$:

$$\begin{aligned} \theta: U &\to \mathbb{Q}/p \\ \frac{a}{b} &\mapsto (a \bmod p, b \bmod p). \end{aligned}$$

Note that θ is well-defined. (This fact is standard, but for completeness, we briefly argue it. Take $q_1 = \frac{a_1}{b_1}$, $q_2 = \frac{a_1 x}{b_1 x}$. Then $\theta(q_1) = (a_1 \bmod p, b_1 \bmod p)$ and $\theta(q_2) = (a_1 x \bmod p, b_1 x \bmod p)$. But we have $(a_1 \bmod p)(b_1 x \bmod p) \equiv (b_1 \bmod p)(a_1 x \bmod p) \pmod p$, so $\theta(q_1) \sim \theta(q_2)$.)

As mentioned above, $\mathbb{Q}/p$ does not have as much "room" as one might guess. To make θ 1-to-1, we must choose p carefully. The lemma below says how to do so, but we need a definition first. Define the *s-value of a finite set* $U \subset \mathbb{Q}$ as follows. Write the *ith* element q_i of U as a_i/b_i where $a_i, b_i \in \mathbb{Z}$, $b_i > 0$, and a_i and b_i are co-prime. Then the s-value of U, written $s(U)$, is $s(U) = \max_{i,j}(|a_i| \cdot b_j)$.

Lemma B.1.2. For any $U \subset \mathbb{Q}$, if $p > 2 \cdot s(U)$, then θ is 1-to-1.[1]

Proof. Take $q_1, q_2 \in U$, where $\theta(q_1) \sim \theta(q_2)$. We need to show that $q_1 = q_2$. Write $q_1 = a_1/b_1$ and $q_2 = a_2/b_2$ in reduced form (that is, a_i and b_i are co-prime). Note that by definition of s-value and choice of p, p is greater than each of a_1, b_1, a_2, b_2, so we can write $\theta(q_1)$ as (a_1, b_1) and $\theta(q_2)$ as (a_2, b_2). Since $\theta(q_1) \sim \theta(q_2)$, we have $a_1 b_2 \equiv a_2 b_1 \pmod{p}$. But then if $a_1 b_2 \neq a_2 b_1$ (as would be implied by $q_1 \neq q_2$) we would have:

$$\begin{aligned} p &\leq |a_1 b_2 - a_2 b_1| \\ &\leq |a_1 b_2| + |a_2 b_1| \\ &\leq 2 \cdot s(U), \end{aligned}$$

making $p \leq 2 \cdot s(U)$, a contradiction. □

Representing computations over $\mathbb{Q}/p$ faithfully. Note that:
(1) If $q_1, q_2 \in U$ and $q_1 + q_2 \in U$, then $\theta(q_1) + \theta(q_2) \sim \theta(q_1 + q_2)$.
(2) If $q_1, q_2 \in U$ and $q_1 q_2 \in U$, then $\theta(q_1)\theta(q_2) \sim \theta(q_1 q_2)$.
These properties can be verified by inspection. Since θ preserves addition and multiplication, and since θ is 1-to-1 by choice of p, the computation in $\mathbb{Q}/p$ is isomorphic to the computation in $\mathbb{Q}$.

Examples. If U is defined as in Claim B.1.1, then the s-value is upper-bounded by

$$m \cdot 2^{2(N_a+N_b)} \cdot 2^{2N_b} = m \cdot 2^{2N_a+4N_b}.$$

Applying Lemma B.1.2, if we take $p > 2m \cdot 2^{2N_a+4N_b}$, then computation over $\mathbb{Q}/p$ is isomorphic to computation over U, as claimed in Section 4.5.1. As another example, consider numbers of the form $a \cdot 2^{-q}$, where $|a| < 2^{N_a}$ and $|q| \leq N_q$. Then the smallest positive number is $1/2^{N_q}$ and the largest positive number is $2^{N_a+N_q}/1$, giving an s-value of $2^{N_a+N_q} \cdot 2^{N_q}$. The prime thus requires at least $\log_2(2 \cdot 2^{N_a+N_q} \cdot 2^{N_q}) = N_a + 2N_q + 1$ bits, as claimed in Section 4.5.1.

Canonical forms and θ^{-1}

Later, it will be convenient to have defined θ^{-1} explicitly—and to have expressed this definition in terms of a particular representation of elements of $\mathbb{Q}/p$. This may seem strange because the whole concept of equivalence class is that, within a class, all representations

[1]This bound on p improves the originally published work.

are equivalent. However, our constraints for certain computations, such as less-than, will require assumptions about the representation of an element (see Appendix B.2.2). Thus, we define a canonical representation below; we focus on the case when U is of the form $\{a/b \colon |a| < 2^{N_a}, b \in \{1, 2, 2^2, 2^3, \ldots, 2^{N_b}\}\}$.

Definition B.1.1 (Canonical form in $\mathbb{Q}/p$). An element $(a, b) \in \theta(U)$ is a *canonical form* or *canonical representation* of its equivalence class if $a \in [0, 2^{N_a}] \cup [p - 2^{N_a}, p)$ and $b \in \{1, 2, 4, \ldots, 2^{N_b}\}$. Every element in $\theta(U)$ has such a representation, by definition of U and θ.

We now define θ^{-1}; let (e_a, e_b) denote a canonical form of e:

$$\begin{aligned}\theta^{-1} \colon \theta(U) &\to U \\ e &\mapsto \begin{cases} e_a/e_b, & 0 \le e_a \le 2^{N_a} \\ (e_a - p)/e_b, & p - 2^{N_a} \le e_a < p \end{cases}\end{aligned}$$

Note that when e_a is in the "upper" part of the range, θ^{-1} maps e to a negative number in $\mathbb{Q}$. Note also that the canonical form for an equivalence class may not be unique. However, the following two claims establish that this non-uniqueness is not an issue in our context.

Claim B.1.3. θ^{-1} is well-defined.

Proof. For $e \in \theta(U)$, let $e = (a, b) \sim (c, d)$, where (a, b) and (c, d) are both canonical forms. We wish to show that $\theta^{-1}((a, b)) = \theta^{-1}((c, d))$.

We have $\theta^{-1}((a, b)) \in U$ and $\theta^{-1}((c, d)) \in U$, by definition of θ^{-1} and U. Also, we have $\theta(\theta^{-1}((a, b))) \sim (a, b)$, as follows. If $a \in [0, 2^{N_a}]$, then $\theta^{-1}((a, b)) = a/b$ and $\theta(a/b) = (a, b)$. If $a \in [p - 2^{N_a}, p)$, then $\theta^{-1}((a, b)) = (a - p)/b$ and $\theta((a - p)/b) = (a - p \bmod p, b) \sim (a, b)$. Likewise, $\theta(\theta^{-1}((c, d))) \sim (c, d)$. Now, let $u_1 = \theta^{-1}((a, b))$ and $u_2 = \theta^{-1}((c, d))$. Assume toward a contradiction that $u_1 \neq u_2$; then $\theta(u_1) \not\sim \theta(u_2)$, by Lemma B.1.2. Thus $(a, b) \sim \theta(\theta^{-1}((a, b)) \not\sim \theta(\theta^{-1}((c, d))) \sim (c, d)$, a contradiction. □

Claim B.1.4. An element in $\theta(U)$ cannot have two canonical representations (a, b) and (c, d) with $a \in [0, 2^{N_a}]$ and $c \in [p - 2^{N_a}, p)$.

Proof. Take $(a, b) \sim (c, d)$ where $a \in [0, 2^{N_a}]$ and $c \in [p - 2^{N_a}, p)$ (note that $b, d > 0$). Because θ^{-1} is a function (Claim B.1.3), $\theta^{-1}((a, b)) = \theta^{-1}((c, d))$. However, $\theta^{-1}((a, b)) = a/b \ge 0$ and $\theta^{-1}((c, d)) = (c - p)/d < 0$, which is a contradiction. □

Discussion

Most of our work above presumed a restriction on U: that the denominators of its elements are powers of 2. We defined U this way because, without this restriction, we would need a much larger prime p, per Lemma B.1.2. However, this restriction is not fundamental, and our framework does not require it. On the other hand, the restriction yields primitive floating-point numbers with acceptable precision at acceptable cost (see Section 4.5.1).

Implementation detail

When working with computations over $\mathbb{Q}$, we express them over the finite field $\mathbb{Q}/p$. However, our implementation (source code, etc.) assumes that the finite field is represented as $\mathbb{Z}/p$. Fortunately, as noted above, $\mathbb{Q}/p$ is isomorphic to $\mathbb{Z}/p$ via the following map:

$$\begin{aligned} f{:}\mathbb{Q}/p &\to \mathbb{Z}/p \\ (a,b) &\mapsto ab^{-1}. \end{aligned}$$

We take advantage of this isomorphism to reuse our implementation over $\mathbb{Z}/p$. Specifically, when computing over $\mathbb{Q}/p$, V and P follow the protocol below.

Definition B.1.2 (Zaatar-Q protocol). Let Ψ be a computation over $\mathbb{Q}/p$, and let Ψ' be the same computation, expressed over $\mathbb{Z}/p$. The Zaatar-Q protocol for verifying Ψ is defined as follows:

1. $V \to P$: a vector x, over the domain $\mathbb{Q}/p$.
2. $P \to V$: $y = \Psi(x)$.
3. $P \to V$: x' and y'. P obtains x', y' (which are vectors in $\mathbb{Z}/p$) by applying f elementwise to x and y.
4. V checks that for all $(a,b) \in \{x \cup y\}$ and the corresponding element $c \in \{x' \cup y'\}$, $cb \equiv a \bmod p$. This confirms that P has applied f correctly. If the check fails, V rejects.
5. V engages P using the existing Zaatar implementation, to verify that $y' = \Psi'(x')$.

The implementation convenience carries a cost: P must compute b^{-1} for each element a/b in the input and output of Ψ (as part of computing f), and V must check that P applied f correctly. On the other hand, some cost seems unavoidable. In fact, it might cost even more if the implementation worked in $\mathbb{Q}/p$ directly: arithmetic is roughly twice as expensive using the $\mathbb{Q}/p$ representation versus the $\mathbb{Z}/p$ representation.

B.2 Case study: branch and inequalities

Below, we will give constraints for a computation that branches based on a less-than test. (This will instantiate step C3 for the case study in Section 4.5.2.) Most of the work is in representing the less-than test; we do so with *range constraints* that take apart a number and interrogate its bits.

B.2.1 Order comparisons over the integers

Preliminaries

We will assume that the programmer or compiler has applied steps C1 and C2 to bound the inputs, x_1 and x_2, and to choose $\mathbb{F}$; thus, their difference is bounded too. Specifically, we assume $x_1 - x_2 \in U \subset [-2^{N-1}, 2^{N-1})$, $\mathbb{F} = \mathbb{Z}/p$ for some $p > 2^N$, and $\theta(x) = x \bmod p$. (See Appendix B.1.1.)

With these restrictions, $x_1 < x_2$ if and only if $x_1 - x_2 \in [-2^{N-1}, 0)$, which holds if and only if $\theta(x_1) - \theta(x_2) \in [p - 2^{N-1}, p)$; the second equivalence follows because θ is 1-to-1 and preserves addition and multiplication, as shown in the previous appendix.

Step C3

To instantiate step C3, we write a set of constraints, $\mathcal{C}_<$:[2]

$$\mathcal{C}_< = \left\{ \begin{array}{l} B_0(1 - B_0) = 0, \\ B_1(2 - B_1) = 0, \\ \quad \vdots \\ B_{N-2}(2^{N-2} - B_{N-2}) = 0, \\ \\ \theta(X_1) - \theta(X_2) - (p - 2^{N-1}) - \sum_{i=0}^{N-2} B_i = 0 \end{array} \right\}$$

Lemma B.2.1. $\mathcal{C}_<$ is satisfiable if and only if $\theta(x_1) - \theta(x_2) \in [p - 2^{N-1}, p)$.

Proof. Assume $\theta(x_1) - \theta(x_2) \in [p - 2^{N-1}, p)$. Let $X_3 = \theta(x_1) - \theta(x_2) - (p - 2^{N-1})$. Observe that $X_3 \in [0, 2^{N-1})$, so X_3's binary representation has bits $z_0, z_1, \ldots, z_{N-2}$. Now, set $B_i = z_i \cdot 2^i$ for $i \in \{0, 1, \ldots, N-2\}$. This will satisfy all but the last constraint because B_i is set equal to

[2] Step C3 in the body text (§4.5) calls for "equivalent" constraints, but the definition of "equivalent" in the same section presumes a designated output variable, which the constraints for logical tests do not have. However, one can extend the definition of "equivalent" to logical tests.

either 0 or 2^i. And the last constraint is satisfied from the definition of X_3 and because we set the $\{B_i\}$ so that $\sum_{i=0}^{N-2} B_i = X_3$. For the other direction, if the constraints are satisfiable, then $\theta(x_1) - \theta(x_2) = p - 2^{N-1} + \sum_{i=0}^{N-2} B_i$, where the $\{B_i\}$ are powers of 2, or 0. This means that $\theta(x_1) - \theta(x_2) \in [p - 2^{N-1}, p)$. □

Corollary B.2.2. For x_1, x_2 as restricted above, $\mathcal{C}_<$ is satisfiable if and only if $x_1 < x_2$.

In other words, assuming the input restrictions, $\mathcal{C}_<$ is equivalent to the logical test of < over $\mathbb{Z}$.

B.2.2 Order comparisons over the rationals

When dealing with the rationals, extra preliminary work is required to apply step C3; the core reason is that each element in $\mathbb{Q}/p$ has multiple representations (recall that $\mathbb{Q}/p$ is isomorphic to $\mathbb{Z}/p$).

Preliminaries

We assume that the programmer or compiler has applied steps C1 and C2 to restrict the inputs, x_1 and x_2, so that $x_1 - x_2 \in U$, for $U = \{a/b \colon |a| < 2^{N_a}, b \in \{1, 2, 2^2, \ldots, 2^{N_b}\}\}$. Similarly, we assume that $\mathbb{F}$ is $\mathbb{Q}/p$, p is chosen according to Lemma B.1.2, and $\theta(a/b) = (a \bmod p, b \bmod p)$. (See Appendix B.1.2.)

At this point, we need the $x_1 < x_2$ test to be in a form suitable for representation in $\mathbb{Q}/p$. Observe that $x_1 < x_2$ if and only if $x_1 - x_2 \in S = \{a/b \colon -2^{N_a} \le a < 0, b \in \{1, 2, 2^2, \ldots, 2^{N_b}\}\}$, which holds if and only if $\theta(x_1) - \theta(x_2) \in \theta(S)$; as with the integers case, the second biconditional follows because θ is 1-to-1, and preserves addition and multiplication. However, we wish to represent this condition in a way that explicitly refers to the representation of $\theta(x_1) - \theta(x_2)$.

Claim B.2.3. $\theta(x_1) - \theta(x_2) \in \theta(S)$ if and only if the numerator in the canonical representation (see Definition B.1.1) of $\theta(x_1) - \theta(x_2)$ is contained in $[p - 2^{N_a}, p)$.

Proof. We will use the definition of θ^{-1} in the previous appendix. Let $e = \theta(x_1) - \theta(x_2)$. If $e \in \theta(S)$, then $\theta^{-1}(e) = a/b$, where $a \in [-2^{N_a}, 0)$ and $b \in \{1, 2, 2^2, \ldots, 2^{N_b}\}$. Thus, $\theta(a/b) = (p + a, b)$, where $p + a \in [p - 2^{N_a}, p)$, and $\theta(a/b) = \theta(\theta^{-1}(e)) \sim e$, so e has a canonical representation of the required form. On the other hand, if $e \sim (a, b)$, where $a \in [p - 2^{N_a}, p)$, then $\theta^{-1}(e) = (a - p)/b \in S$, so $e \sim \theta(\theta^{-1}(e)) \in \theta(S)$. □

Step C3

We instantiate step C3 with the following constraints $\mathcal{C}_<$:

$$\mathcal{C}_< = \left\{\begin{array}{ll} A_0((1,1) - A_0) & = (0,1), \\ A_1((2,1) - A_1) & = (0,1), \\ \vdots & \vdots \\ A_{N_a-1}((2^{N_a-1},1) - A_{N_a-1}) & = (0,1), \\ & \\ A - (p - 2^{N_a}, 1) - \sum_{i=0}^{N_a-1} A_i & = (0,1), \\ & \\ B_0((1,1) - B_0) & = (0,1), \\ B_1((1,1) - B_1) & = (0,1), \\ \vdots & \vdots \\ B_{N_b}((1,1) - B_{N_b}) & = (0,1), \\ & \\ \sum_{i=0}^{N_b} B_i - (1,1) & = (0,1), \\ & \\ B - \sum_{i=0}^{N_b} B_i \cdot (1, 2^i) & = (0,1), \\ & \\ \theta(X_1) - \theta(X_2) - A \cdot B & = (0,1) \end{array}\right\}$$

Lemma B.2.4. $\mathcal{C}_<$ is satisfiable if and only if the numerator in the canonical representation (see Definition B.1.1) of $\theta(x_1) - \theta(x_2)$ is contained in $[p - 2^{N_a}, p)$.

Proof. Assume that $X_3 = \theta(x_1) - \theta(x_2)$ has the required form (a, b). We have $k = \log_2 b \in \{0, 1, 2, \ldots, N_b\}$ and $a \in [p - 2^{N_a}, p)$. Now, take $B_k = (1,1)$ and all other $B_j = (0,1)$; this satisfies all of the B_i constraints, including $\sum_{i=0}^{N_b} B_i - (1,1) = (0,1)$, which requires that exactly one B_i be equal to $(1,1)$. For B, take $B = (1, b) = (1, 2^k)$, to satisfy $B - \sum_{i=0}^{N_b} B_i \cdot (1, 2^i) = (0,1)$.

Now, let $a' = a - (p - 2^{N_a})$. The binary representation of a' has bits $z_0, z_1, \ldots, z_{N_a-1}$. Set $A_i = (z_i, 1)(2^i, 1)$ for $i \in \{0, 1, \ldots, N_a - 1\}$. This will satisfy all of the individual A_i constraints. And, since $\sum_{i=0}^{N_a-1} A_i = (a', 1)$, we can take $A = (a, 1)$ to satisfy $A - (p - 2^{N_a}, 1) - \sum_{i=0}^{N_a-1} A_i = (0,1)$. The remaining constraint is the last one in the list. It is satisfiable because we took $B = (1, b)$ and $A = (a, 1)$, giving $X_3 - (a,1) \cdot (1,b) = (0,1)$.

For the other direction, if the constraints are satisfiable, then $X_3 = \theta(x_1) - \theta(x_2)$ can be written as $(a,1)(1,b)$, where $b \in \{1, 2, \ldots, 2^{N_b}\}$ and where $a = p - 2^{N_a} + \sum_{i=0}^{N_a-1} z_i 2^i$, for $z_i \in \{0,1\}$. This implies that $a \in [p - 2^{N_a}, p)$. $\square$

In analogy with the integers case, notice that the lemma, together with the reasoning in "Preliminaries", implies the following corollary.

Corollary B.2.5. If the input restrictions are met, then $\mathcal{C}_<$ is satisfiable if and only if $x_1 < x_2$. That is, $\mathcal{C}_<$ is equivalent to $<$ over $\mathbb{Q}$.

B.2.3 Branching

We now return to the case study in Section 4.5.2. We will abstract the domain ($\mathbb{Z}/p$ or $\mathbb{Q}/p$): when we write 0 in constraints below, it denotes the additive identity, which is $(0,1)$ in $\mathbb{Q}/p$, and when we write 1, it denotes the multiplicative identity, which is $(1,1)$ in $\mathbb{Q}/p$. Recall the computation Ψ and the constraints $\mathcal{C}_\Psi$ (Figure 4.5):

```
if (X1 < X2)
  Y = 3
else
  Y = 4
```

$$\mathcal{C}_\Psi = \left\{ \begin{array}{l} M\{\mathcal{C}_<\}, \\ M(Y-3)=0, \\ (1-M)\{\mathcal{C}_{>=}\}, \\ (1-M)(Y-4)=0 \end{array} \right\}$$

We now argue that $\mathcal{C}_\Psi$ is equivalent to Ψ. (The definition of "equivalent" is given in Section 4.5.)

Lemma B.2.6. The constraints $\mathcal{C}_\Psi(X_1 = x_1, X_2 = x_2, Y = y)$ are satisfiable if and only if $y = \Psi(x_1, x_2)$.

Proof. Assume $\mathcal{C} = \mathcal{C}_\Psi(X_1 = x_1, X_2 = x_2, Y = y)$ is satisfiable. Since $\mathcal{C}_<$ and $\mathcal{C}_{>=}$ cannot be simultaneously satisfiable (that would imply opposing logical conditions), then $M = 0$ or $1 - M = 0$. If $1 - M = 0$, then $y = 3$, since we are given that the constraint $M(Y-3) = 0$ is satisfiable when $Y = y$. Moreover, $\mathcal{C}_<$ must be satisfiable, implying that $x_1 < x_2$ (see Corollaries B.2.2 and B.2.5). On the other hand, by analogous reasoning, if $M = 0$, then $y = 4$, $\mathcal{C}_{>=}$ is satisfiable, and $x_1 \geq x_2$. Thus, we have two cases: (1) $x_1 < x_2$ and $y = 3$ or (2) $x_1 \geq x_2$ and $y = 4$. But this means that $y = \Psi(x_1, x_2)$ for all x_1, x_2 in the permitted input.

Now assume that $y = \Psi(x_1, x_2)$. If $x_1 < x_2$, then $y = 3$. Take $M = 1$ to satisfy the constraints $(1-M)\{\mathcal{C}_{>=}\} = 0$ and $(1-M)(Y-4) = 0$. Also, $M(Y-3) = 0$ is satisfied, because $y = 3$. Last, $\mathcal{C}_<$ can be satisfied, because $x_1 < x_2$. Thus, the constraints are satisfiable if $x_1 < x_2$. Similar reasoning establishes that the constraints are satisfiable if $x_1 \geq x_2$. □

We can generalize the computation Ψ. For instance, let Ψ_1, Ψ_2 be sub-computations, which we abbreviate in code as `comp1` and `comp2`. Let $\mathcal{C}_{\Psi_1}$ and $\mathcal{C}_{\Psi_2}$ denote the constraints that are equivalent to Ψ_1 and Ψ_2, and rename the distinguished output variables in $\mathcal{C}_{\Psi_1}$ and $\mathcal{C}_{\Psi_2}$ to be Y_1 and Y_2, respectively. Below, Ψ and $\mathcal{C}_\Psi$ are equivalent:

Ψ :

```
if (X1 < X2)
  Y = comp1
else
  Y = comp2
```

$$\mathcal{C}_\Psi = \left\{ \begin{array}{l} M\{\mathcal{C}_<\}, \\ M\{\mathcal{C}_{\Psi_1}\}, \\ M(Y - Y_1) = 0, \\ (1-M)\{\mathcal{C}_{>=}\}, \\ (1-M)\{\mathcal{C}_{\Psi_2}\}, \\ (1-M)(Y-Y_2) = 0 \end{array} \right\}$$

The reasoning that establishes the equivalence is very similar to the proof of Lemma B.2.6. (The differences are as follows. In the forward direction, take $M = 1$. Then, since $Y = y$ and $M(Y - Y_1)$ is satisfied, $Y_1 = y$; meanwhile, $\mathcal{C}_{\Psi_1}(Y_1 = y, X_1 = x_1, X_2 = x_2)$ must be satisfied, which implies $y = \Psi_1(x_1, x_2)$ and hence $y = \Psi(x_1, x_2)$. In the reverse direction, take $x_1 < x_2$. Then we have $y = \Psi_1(x_1, x_2)$. But this implies that when $Y_1 = y$, we can satisfy $\mathcal{C}_{\Psi_1}$, so set $Y_1 = y$. Furthermore, set $Y = y$, and we thus satisfy $M(Y - Y_1)$ and hence all constraints.)

We can generalize further. First, the logical test in the "if" can be an arbitrary test constructed from `==`, `!=`, `&&`, `||`, `>`, `>=`, `<`, `<=`; in this case, we must also construct the negation of the test (just as we need constraints that represent both $\mathcal{C}_<$ and $\mathcal{C}_{>=}$). Second, we need not capture the result of the conditional in Y; we can assign the result to an intermediate variable Z. In that case, we would replace the constraints $M(Y - Y_1) = 0$ and $(1-M)(Y-Y_2) = 0$ with $M(Z - Y_1)$ and $(1-M)(Z-Y_2)$, respectively, and of course we would need other constraints that capture the flow from Z to the ultimate output, Y.

B.3 Program constructs and costs

This appendix describes further program constructs; as with the case study, the work here corresponds to step C3 in our framework. However, in this appendix, we will not delve into as much detail as in the previous appendices; a more precise syntax and semantics is future work. Below, we describe how we map program constructs to constraints and then briefly consider the costs of doing so.

B.3.1 Program constructs

Aside from order comparisons, the computations and constraints below are independent of the domain of the computation; as in Appendix B.2.3, 0 and 1 denote the additive and multiplicative identities in the field in question.

Tests

==. Consider the fragment `(comp1) == (comp2)`, where `comp1` and `comp2` are computations Ψ_1 and Ψ_2. Renaming the output variables in Ψ_1 and Ψ_2 to be Y_1 and Y_2, respectively, we can represent the fragment with the constraint $Y_1 - Y_2 = 0$.

!=. Consider the program fragment `Z1 != Z2`. An equivalent constraint is $M \cdot (Z_1 - Z_2) - 1 = 0$, where M is a new auxiliary variable. This constraint is satisfiable if and only if $Z_1 - Z_2$ has a multiplicative inverse; that is, it is satisfiable if and only if $Z_1 - Z_2 \neq 0$, or $Z_1 \neq Z_2$. As above, we can represent `(comp1) != (comp2)`; the constraint would be $M \cdot (Y_1 - Y_2) - 1 = 0$.

<, <=, >, >=. Appendix B.2 described in detail the constraints that represent <. A similar approach applies for the other three order comparisons. For example, for `X1 <= X2` over the rationals, we want to enforce that the canonical numerator (see Definition B.1.1) of $X_1 - X_2 \in [p - 2^{N_a}, p) \cup \{0\}$. To do so, we modify $\mathcal{C}_<$ in Appendix B.2.2 as follows. First, we add a constraint $A_0'(A_0' - (1,1)) = (0,1)$. Second, we change the A constraint from

$$A - (p - 2^{N_a}, 1) - \sum_{i=0}^{N_a-1} A_i = (0,1)$$

to

$$A - (p - 2^{N_a}, 1) - \sum_{i=0}^{N_a-1} A_i - A_0' = (0,1).$$

Composing tests into expressions

To compose logical expressions, we provide `&&` and `||`. We do not provide logical negation explicitly, but our computational model includes inverses for all tests (for example, == and !=), so the programmer or compiler can use DeMorgan's laws to write the negation of any logical expression in terms of `&&` and `||`.

`||`. Consider the expression `(cond1) || (cond2)`, and let $\mathcal{C}_1$ and $\mathcal{C}_2$ be the constraints that are equivalent to `cond1` and `cond2` respectively. The expression is equivalent to the following constraints, where M_1, M_2 are new variables:

$$\mathcal{C}_{||} = \left\{ \begin{array}{l} (M_1 - 1)(M_2 - 1) = 0, \\ M_1\{\mathcal{C}_1\}, \\ M_2\{\mathcal{C}_2\} \end{array} \right\}$$

We now argue that $\mathcal{C}_{||}$ is equivalent to the original expression. If `cond1` holds, then $\mathcal{C}_1$ is satisfiable; choose $M_1 = 1$ and $M_2 = 0$ to satisfy all constraints. Note that if `cond2` also holds, then setting $M_2 = 1$ also works, but the prover might wish to avoid "executing" (i.e., finding a satisfying assignment for) $\mathcal{C}_2$. On the other hand, if $\mathcal{C}_{||}$ is satisfiable, then $M_1 = 1$ or $M_2 = 1$, or both. If $M_1 = 1$, then $\mathcal{C}_1$ is satisfied, which implies that `cond1` holds. The identical reasoning applies if $M_2 = 1$.

`&&`. To express `(cond1) && (cond2)`, the programmer simply includes $\mathcal{C}_1$ and $\mathcal{C}_2$.

Conditionals

We covered conditional branching in detail in Appendix B.2.3. Below we describe two other conditional constructs, EQUALS-ZERO and NOT-EQUALS-ZERO, that are useful as "type casts" from integers to 0-1 values.

NOT-EQUALS-ZERO. The computation Ψ is `Y = (X != 0) ? 1 : 0`, and it can be represented with the following constraints:

$$\mathcal{C}_{\text{NOT-EQUALS-ZERO}} = \left\{ \begin{array}{l} X \cdot M - Y = 0, \\ (1 - Y) \cdot X = 0 \end{array} \right\}.$$

One can verify by inspection that $\mathcal{C}_{\text{NOT-EQUALS-ZERO}}(Y = y, X = x)$ is satisfiable if and only if $y = \Psi(x)$. Note that we could implement NOT-EQUALS-ZERO by using a conditional branch (see Appendix B.2.3) together with a `!=` test (see above). However, relative to that option, the constraints above are more concise (fewer constraints, fewer variables). They are also more concise than the representation given by Cormode et al. [53]. Roughly speaking, Cormode et al. represent NOT-EQUALS-ZERO with a constraint like $X^{p-1} - Y = 0$, where p is the modulus of $\mathbb{Z}/p$ (the approach works because Fermat's Little Theorem says that for any non-zero X, $X^{p-1} \equiv 1 \pmod{p}$); this approach requires $\log p$ intermediate variables.

EQUALS-ZERO. This computation is the inverse of the previous; the constraint representation of $\mathcal{C}_{\text{EQUALS-ZERO}}$ is the same as $\mathcal{C}_{\text{NOT-EQUALS-ZERO}}$ but with Y replaced by $1 - Y$.

B.3.2 Costs

As mentioned in Section 4.5.2, there are two main costs of the constructs above. First, the order comparisons require a variable and a constraint for each bit position, Second, the constraints for conditional branching and `||` appear to be degree-3 or higher—notice the $M\{\mathcal{C}\}$ notation in these constructs—but must be reduced to be degree-2, as required by the protocol (see §3.2.1, §4.5) Below, we describe this reduction and its costs.

We will start with the degree-3 case and then generalize. Let $\mathcal{C}$ be a constraint set over variables $\{Z_1, \ldots, Z_n, M\}$, and let $\mathcal{C}$ have a degree-3 constraint, $Q(Z_1, \ldots, Z_n, M)$. Q has the form $R(M) \cdot S(Z_1, \ldots, Z_n)$, where $R(M)$ is M or $(1 - M)$; this follows because higher-degree constraints only ever emerge from multiplication by an auxiliary variable. We reduce Q by constructing a $\mathcal{C}'$ that is the same as $\mathcal{C}$ except that Q is replaced with the following two constraints, using a new variable M':

$$M' - S(Z_1, \ldots, Z_n) = 0,$$
$$R(M) \cdot M' = 0.$$

Claim B.3.1. $\mathcal{C}$ is satisfiable if and only if $\mathcal{C}'$ is satisfiable.

Proof. Abbreviate $Z = Z_1, \ldots, Z_n$. Assume $\mathcal{C}$ is satisfied by assignment $Z = z, M = m$. Use this same setting for $\mathcal{C}'$. So far, all constraints other than the two new ones are satisfied in $\mathcal{C}'$. To satisfy the two new ones, set $M' = S(z)$. This satisfies the first new constraint. It also satisfies the second new constraint because either $M' = 0$ or $M' \neq 0$, in which case $S(z) \neq 0$, which implies (because $\mathcal{C}$ is satisfied and hence Q is too) that $R(m) = 0$.

Now assume that $\mathcal{C}'$ is satisfiable with assignment $Z = z, M = m, M' = m'$. In $\mathcal{C}$, set $Z = z, M = m$. Now, in this assignment, in $\mathcal{C}'$, $R(m) = 0$ or $m' = 0$. If $R(m) = 0$, then $Q(z, m) = 0$. If $m' = 0$, then $S(z) = 0$, so $Q(z, m) = 0$ again. □

Since applying a single transformation of the kind above does not change the satisfiability of the resulting set, we can transform all of the constraints this way, to make $M\{\mathcal{C}\}$ degree-2. The costs of doing so are as follows. Each of the $|\mathcal{C}|$ constraints in $\mathcal{C}$ causes us to add another constraint and a new variable. Thus, if $\mathcal{C}$ has s variables and $|\mathcal{C}|$ constraints, then our representation of $M\{\mathcal{C}\}$ has $s + |\mathcal{C}|$ variables and $2 \cdot |\mathcal{C}|$ constraints.

The approach above generalizes to higher degrees. Higher-degree constraints emerge from nesting of branches or `||` operations. If there are k levels of nesting somewhere in the computation, then the computation's constraints have a subset of the form

$$\mathcal{C}_{\text{nested}} = M_k\{M_{k-1}\{\cdots M_1\{\mathcal{C}\}\cdots\}\}.$$

(Each of the M_i could also be $1 - M_i$.) Then there is a set of equivalent degree-2 constraints that uses in total $s + k \cdot |\mathcal{C}|$ variables and $(k+1) \cdot |\mathcal{C}|$ constraints.

The details are as follows. Consider a single constraint in $\mathcal{C}_{\text{nested}}$; it has the form $R(M_k)\cdots R(M_2)R(M_1)S(Z) = 0$, where $S(Z)$ is degree-2. Replace this constraint with the following ones, to form $\mathcal{C}'_{\text{nested}}$:

$$\begin{aligned}
&M'_0 - S(Z) = 0,\\
&M'_1 - R(M_1) \cdot M'_0 = 0,\\
&M'_2 - R(M_2) \cdot M'_1 = 0,\\
&\quad\vdots\\
&M'_{k-1} - R(M_{k-1}) \cdot M'_{k-2} = 0,\\
&R(M_k) \cdot M'_{k-1} = 0.
\end{aligned}$$

The proof that $\mathcal{C}'_{\text{nested}}$ and $\mathcal{C}_{\text{nested}}$ are equivalent is similar to the proof of Claim B.3.1 and is omitted for the sake of brevity. Observe that this construction introduces, per constraint in $\mathcal{C}$, k new variables and k new constraints, leading to the costs stated above.

Appendix C

A linear PCP protocol based on QAPs

This section describes a new linear PCP [79] protocol, which is used by Zaatar. This protocol is based on Quadratic Arithmetic Programs (QAPs), which is a formalism due to Gennaro et al. [63].

Our description will be tailored to our context. In particular, the PCP protocol will check the satisfiability of constraints that are assumed to represent a computation (§4.5). However, this generalizes to checking the satisfiability of any degree-2 constraint set. Since degree-2 constraint satisfaction is an NP-complete problem [13], the PCP that we present here generalizes to checking NP relations. The core idea is to transform a set of constraints to a set of polynomials in such a way that the constraints are satisfiable if and only if the polynomials have a particular algebraic relation.

C.1 The construction

Notation. We are given a constraint set $\mathcal{C}$ over the variables $W = (X, Y, Z)$, where X is the set of distinguished input variables, Y is the set of distinguished output variables, and Z is the set of remaining variables. Let $|\mathcal{C}|$ denote the number of constraints in $\mathcal{C}$. Also, let $n = |W|$ and $n' = |Z|$. The following indexing will be convenient: the variables in Z are labeled as $W_1, \ldots, W_{n'}$, and the variables in X and Y are labeled as $W_{n'+1}, \ldots, W_n$. We will be working over a finite field, $\mathbb{F}$.

Building blocks. The building blocks described in the next several paragraphs are borrowed from QAPs [63], though our notation and phrasing is different, and we work with constraints explicitly.

We require that each constraint is in *quadratic form* (Section C.2). That is, constraint

j has the form $p_{j,A}(W) \cdot p_{j,B}(W) = p_{j,C}(W)$, where $p_{j,A}, p_{j,B}$, and $p_{j,C}$ are degree-1 polynomials over W. Now, for variable W_i (which will be a member of X, Y, or Z), let $a_{i,j}$ be the coefficient of W_i in $p_{j,A}$. Similarly, let $b_{i,j}$ be the coefficient of W_i in $p_{j,B}$, and let $c_{i,j}$ be the coefficient of W_i in $p_{j,C}$. Finally, for constraint j, let $a_{0,j}$, $b_{0,j}$, and $c_{0,j}$ be the constant terms in $p_{j,A}, p_{j,B}$, and $p_{j,C}$.

We will construct polynomials that encode these constraints. On the way there, it will be helpful to visualize three $(n+1) \times |\mathcal{C}|$ variable-constraint matrices: A, B, C. In each of these matrices, each row represents a variable in W (as a special case, row $i = 0$ represents the constant terms), and each column represents a constraint in $\mathcal{C}$. In the A (resp., B and C) matrix, the (i, j) cell contains $a_{i,j}$ (resp., $b_{i,j}$ and $c_{i,j}$); thus, this cell is non-zero if variable i appears in constraint j in the $p_{j,A}$ (resp., $p_{j,B}$ and $p_{j,C}$) component. Observe that the matrices A, B, C encode the constraints; we will now turn these matrices into polynomials.

We construct degree-$|\mathcal{C}|$ polynomials $\{A_i(t)\}, \{B_i(t)\}, \{C_i(t)\}$, for $i \in [0..n]$, by interpolation. Take distinguished non-zero points $\sigma_1, \sigma_2, \ldots, \sigma_{|\mathcal{C}|} \in \mathbb{F}$, and for each i require that $A_i(\sigma_j) = a_{i,j}$, $B_i(\sigma_j) = b_{i,j}$, and $C_i(\sigma_j) = c_{i,j}$; at this point, there are $|\mathcal{C}|$ point-value pairs constraining each of the $A_i(t)$, $B_i(t)$, and $C_i(t)$. Finally, require $A_i(0) = B_i(0) = C_i(0) = 0$; this gets us to $|\mathcal{C}| + 1$ points and evaluations, which fully defines the polynomials $A_i(t), B_i(t)$, and $C_i(t)$, by interpolation. Moreover, we can *represent* each $A_i(t)$, $B_i(t)$, and $C_i(t)$ in terms of their evaluations at the values $\{\sigma_j\}$. That is, we can write:

$$\begin{array}{lll}
A_0(t) = (a_{0,1}, a_{0,2}, \ldots, a_{0,|\mathcal{C}|}) & B_0(t) = (b_{0,1}, b_{0,2}, \ldots, b_{0,|\mathcal{C}|}) & C_0(t) = (c_{0,1}, c_{0,2}, \ldots, c_{0,|\mathcal{C}|}) \\
A_1(t) = (a_{1,1}, a_{1,2}, \ldots, a_{1,|\mathcal{C}|}) & B_1(t) = (b_{1,1}, b_{1,2}, \ldots, b_{1,|\mathcal{C}|}) & C_1(t) = (c_{1,1}, c_{1,2}, \ldots, c_{1,|\mathcal{C}|}) \\
\quad\vdots & \quad\vdots & \quad\vdots \\
A_n(t) = (a_{n,1}, a_{n,2}, \ldots, a_{n,|\mathcal{C}|}) & B_n(t) = (b_{n,1}, b_{n,2}, \ldots, b_{n,|\mathcal{C}|}) & C_n(t) = (c_{n,1}, c_{n,2}, \ldots, c_{n,|\mathcal{C}|})
\end{array}$$

Now, construct the *divisor polynomial*, $D(t)$:

$$D(t) = \prod_{j \in [1..|\mathcal{C}|]} (t - \sigma_j).$$

Finally, encode all of the constraints in a single polynomial $P(t, W)$ over t and the constraint variables W:

$$P(t, W) = \left(\sum_{i=0}^{n} W_i \cdot A_i(t)\right) \cdot \left(\sum_{i=0}^{n} W_i \cdot B_i(t)\right) - \left(\sum_{i=0}^{n} W_i \cdot C_i(t)\right).$$

We now give some notation and conventions. We will write $P_{x,y}(t, Z)$ to mean $P(t, W)$ with $X{=}x$ and $Y{=}y$. We will take $w = (x, y, z) \in \mathbb{F}^n$ to mean an assignment to the variables (X, Y, Z); by convention, $w_0 = 1$. Thus, $P_{x,y}(t, z)$ means $P(t, W{=}w)$, for some w; we often write $P(t, W{=}w)$ as $P_w(t)$. Observe that $P_{x,y}(t, Z)$ has the following form, as claimed in Section 4.6: $P_{x,y}(t, Z) = (\sum_{i=1}^{n'} Z_i \cdot A_i(t) + A'(t)) \cdot (\sum_{i=1}^{n'} Z_i \cdot B_i(t) + B'(t)) - (\sum_{i=1}^{n'} Z_i \cdot C_i(t) + C'(t))$, where $A'(t)$ is a linear combination of $A_0(t), A_{n'+1}(t), \ldots, A_n(t)$, the coefficients given by $1, x, y$, and analogously for $B'(t)$ and $C'(t)$.

Claim C.1.1. Let $w = (x, y, z)$ be an assignment to the variables (X, Y, Z). Then $D(t)$ divides $P_w(t)$ if and only if z satisfies $\mathcal{C}(X{=}x, Y{=}y)$.

Proof. Assume $D(t)$ divides $P_w(t)$. Fix a constraint $j \in [1..|\mathcal{C}|]$. By definition of $D(t)$, the polynomial $t - \sigma_j$ is a factor of $P_w(t)$, so σ_j is a root of $P_w(t)$. Thus, $0 = P_w(\sigma_j) = \left(\sum_{i=0}^{n} w_i \cdot A_i(\sigma_j)\right) \cdot \left(\sum_{i=0}^{n} w_i \cdot B_i(\sigma_j)\right) - \left(\sum_{i=0}^{n} w_i \cdot C_i(\sigma_j)\right)$. By construction of $\{A_i(t), B_i(t), C_i(t)\}$, we have $\left(\sum_{i=1}^{n} w_i \cdot a_{i,j} + a_{0,j}\right) \cdot \left(\sum_{i=1}^{n} w_i \cdot b_{i,j} + b_{0,j}\right) = \sum_{i=1}^{n} w_i \cdot c_{i,j} + c_{0,j}$. That is, constraint j is satisfied at $w{=}(x, y, z)$, by definition of $a_{i,j}$, $b_{i,j}$, $c_{i,j}$. But we chose j arbitrarily, so z satisfies all constraints in $\mathcal{C}(X{=}x, Y{=}y)$.

For the other direction, if the z "piece" of w satisfies $\mathcal{C}(X{=}x, Y{=}y)$, then every constraint is satisfied, which implies $P_w(\sigma_j) = 0$ for all $\{\sigma_j\}$, so all $\{\sigma_j\}$ are roots of $P_w(t)$, so $P_w(t)$ can be factored into $(t - \sigma_1) \cdots (t - \sigma_{|\mathcal{C}|}) \cdot H_w(t) = D(t) \cdot H_w(t)$, for some $H_w(t)$. □

The QAP-based proof oracle. Let z be the prover's purported assignment to $\mathcal{C}(X{=}x, Y{=}y)$. A correct proof oracle is $\pi = (\pi_z, \pi_h)$, where $\pi_z(\cdot) = \langle \cdot, z \rangle$ and $\pi_h(\cdot) = \langle \cdot, h \rangle$. Here, $h = (h_0, \ldots, h_{|\mathcal{C}|}) \in \mathbb{F}^{|\mathcal{C}|+1}$ represents the coefficients of a polynomial $H(t)$; that is, $H(t) = \sum_{j=0}^{|\mathcal{C}|} h_j \cdot t^j$. In a correct proof oracle for a correct computation, $H(t)$ satisfies $D(t) \cdot H(t) = P_w(t)$.

The PCP protocol. The protocol is depicted in Figure C.1. A detail is that queries q_a, q_b, q_c, q_d are self-corrected (see [14, §5] or [100, §7.8.3]). The soundness of the protocol is at least $1 - \kappa^\rho$; Section C.3 establishes this bound and quantifies κ. We state the high-order verifier costs immediately below and give details in Section C.4.

The verifier's running time is proportional to $|Z| + |\mathcal{C}|$ and so is the randomness that it must generate; the constant of proportionality depends on the cost of field operations. These costs are almost linear in the number of steps in the computation; there is actually an extra logarithmic term (since the field size, which must be larger than the number of steps in the computation, affects the costs). However, to reflect the practical reality, we mostly disregard

A linear PCP based on QAPs [63]

The verifier $\mathcal{V}$ interacts with a proof oracle π as follows. A correct proof oracle encodes z and h, where z satisfies $\mathcal{C}(X{=}x, Y{=}y)$, and h is the coefficients of a polynomial $H_w(t)$ that satisfies $D(t) \cdot H_w(t) = P_w(t)$, for $w = (x, y, z)$.

Loop ρ times:

- Generate linearity queries: Select $q_5, q_6 \in_R \mathbb{F}^{n'}$ and $q_8, q_9 \in_R \mathbb{F}^{|\mathcal{C}|+1}$. Take $q_7 \leftarrow q_5 + q_6$ and $q_{10} \leftarrow q_8 + q_9$. Perform $\rho_{\text{lin}} - 1$ more iterations of this step.
- Generate divisibility correction queries:
  - Select $\tau \in_R \mathbb{F}$.
  - Take $q_a \leftarrow (A_1(\tau), A_2(\tau), \ldots, A_{n'}(\tau))$, and $q_1 \leftarrow (q_a + q_5)$.
  - Take $q_b \leftarrow (B_1(\tau), B_2(\tau), \ldots, B_{n'}(\tau))$, and $q_2 \leftarrow (q_b + q_5)$.
  - Take $q_c \leftarrow (C_1(\tau), C_2(\tau), \ldots, C_{n'}(\tau))$, and $q_3 \leftarrow (q_c + q_5)$.
  - Take $q_d \leftarrow (1, \tau, \tau^2, \ldots, \tau^{|\mathcal{C}|})$, and $q_4 \leftarrow (q_d + q_8)$.
- Issue queries: Send $q_1, \ldots, q_{4+6\rho_{\text{lin}}}$ to oracle π, getting back $\pi(q_1), \ldots, \pi(q_{4+6\rho_{\text{lin}}})$.
- Linearity tests: Check that $\pi(q_5) + \pi(q_6) \stackrel{?}{=} \pi(q_7)$ and that $\pi(q_8) + \pi(q_9) \stackrel{?}{=} \pi(q_{10})$, and likewise for the other $\rho_{\text{lin}} - 1$ iterations. If not, `reject`.
- Divisibility correction test: Return `reject` unless

$$D(\tau) \cdot (\pi(q_4) - \pi(q_8)) \stackrel{?}{=} \left(\pi(q_1) - \pi(q_5) + \sum_{i=n'+1}^{n} w_i \cdot A_i(\tau) + A_0(\tau)\right) \cdot \left(\pi(q_2) - \pi(q_5) + \sum_{i=n'+1}^{n} w_i \cdot B_i(\tau) + B_0(\tau)\right)$$
$$- \left(\pi(q_3) - \pi(q_5) + \sum_{i=n'+1}^{n} w_i \cdot C_i(\tau) + C_0(\tau)\right).$$

If $\mathcal{V}$ makes it here, `accept`.

Figure C.1: See the text for the definition of $D(t)$ and $P_w(t)$, and the construction of $\{A_i(t)\}$, $\{B_i(t)\}$, $\{C_i(t)\}$, and recall that x and y are labeled as $\{w_{n'+1}, \ldots, w_n\}$.

this term by referring to "the size of the computation", which captures the field size. Of course, costs that are proportional to the computation itself do not save the verifier work; thus, we amortize these costs over multiple instances of the computation (§4.4), in the context of the efficient argument system described below.

The efficient argument system. Zaatar is an efficient argument system [79, 83] that composes the PCP in Figure C.1 with an improved version of IKO's cryptographic machinery (§4.3). The soundness error of the argument system is upper-bounded by $\kappa^{\rho} + 9 \cdot \mu \cdot |\mathbb{F}|^{-1/3}$, where μ is the number of PCP queries; see the analysis in [114, Apdx A.2]. The verifier incurs additional costs from the linear commitment primitive; these costs are proportional to the computation size. The prover's costs stem from constructing the proof vector (treated in Section C.4) plus responding to queries (the costs are proportional to the size of the computation). The network costs are (a) a full query sent from $\mathcal{V}$ to $\mathcal{P}$, and (b) a random seed from which $\mathcal{V}$ and $\mathcal{P}$ derive the PCP queries pseudorandomly (see [114, Apdx A.3]).

C.2 Cost-benefit analysis

Given our goal of removing prover overhead, Zaatar's PCP is very promising. However, we need to consider its benefits against the cost of its additional requirements. This section performs an analysis, summarized in Figure C.2. Our chosen baseline for this analysis is Zaatar without the new PCP encoding (this is a predecessor of Zaatar, called Ginger [115]).

Summary of the analysis. Zaatar requires more constraints over a larger set of variables than Ginger does for the same computation; all other things being equal, this slight blowup would increase the prover's and the verifier's costs. Also, Zaatar requires additional bookkeeping from the prover (when constructing the proof encoding) and the verifier (when constructing queries). However, these two effects are dwarfed by a vast reduction in the size of the proof encoding under Zaatar. The consequence is a correspondingly vast improvement in both the prover's work and the verifier's query setup work (and hence the cross-over points, as defined in Section 4.8). Finally, while there are cases when Zaatar is worse than Ginger, they are contrived computations with a particular structure (e.g., evaluation of dense degree-2 polynomials).[1]

Below, we present the analysis. The comparison depends heavily on the number of constraints and variables in Zaatar versus the alternative, so we begin with these quantities.

Constraints in Zaatar versus Ginger. Whereas Ginger requires only that constraints are degree-2 (§3.2.1), Zaatar imposes an additional requirement. Under Zaatar, each constraint Q_j must be in the form $p_A \cdot p_B = p_C$, where p_A, p_B, and p_C are degree-1 polynomials over the variables in the constraint set. We call this *quadratic form*; the requirement stems from the way that Zaatar, via QAPs, encodes constraints in polynomials (Appendix C.1 gives detail).

[1]Also, the degenerate cases are detectable, so the compiler could simply choose to use Ginger (or [53, 120]) over Zaatar; see [123].

	Ginger [115]	Zaatar
proof vector size ($\lvert u_{\text{ginger}}\rvert$ or $\lvert u_{\text{zaatar}}\rvert$)	$\lvert Z_{\text{ginger}}\rvert + \lvert Z_{\text{ginger}}\rvert^2$	$\lvert Z_{\text{zaatar}}\rvert + \lvert \mathcal{C}_{\text{zaatar}}\rvert = 2 \cdot (\lvert Z_{\text{ginger}}\rvert + K_2)$
worst case proof vector size	$\lvert Z_{\text{ginger}}\rvert + \lvert Z_{\text{ginger}}\rvert^2$	$\lvert u_{\text{ginger}}\rvert \cdot \big(1+\delta\big)$, where δ is $2/\big(\lvert Z_{\text{ginger}}\rvert + 1\big)$
***P*'s per-instance CPU costs**		
Construct proof vector	$T + f \cdot \lvert Z_{\text{ginger}}\rvert^2$	$T + 3f \cdot \lvert \mathcal{C}_{\text{zaatar}}\rvert \cdot \log^2 \lvert \mathcal{C}_{\text{zaatar}}\rvert$
Issue responses	$(h + (\rho \cdot \ell + 1) \cdot f) \cdot \lvert u_{\text{ginger}}\rvert$	$(h + (\rho \cdot \ell' + 1) \cdot f) \cdot \lvert u_{\text{zaatar}}\rvert$
***V*'s per-instance CPU costs**		
Construct computation-specific queries	$\rho \cdot (c \cdot \lvert \mathcal{C}_{\text{ginger}}\rvert + f \cdot K)/\beta$	$\rho \cdot (c + (f_{div} + 5f) \cdot \lvert \mathcal{C}_{\text{zaatar}}\rvert + f \cdot K + 3f \cdot K_2)/\beta$
Construct computation-oblivious queries	$(e + 2c + \rho \cdot (2\rho_{\text{lin}} \cdot c + (\ell + 1) \cdot f)) \cdot \lvert u_{\text{ginger}}\rvert/\beta$	$(e + 2c + \rho \cdot (2\rho_{\text{lin}} \cdot c + \ell' \cdot f)) \cdot \lvert u_{\text{zaatar}}\rvert/\beta$
Process responses	$d + \rho \cdot (2\ell + \lvert x\rvert + \lvert y\rvert) \cdot f$	$d + \rho \cdot (\ell' + 3\lvert x\rvert + 3\lvert y\rvert) \cdot f$

$\mathcal{C}_{\text{ginger}}, \mathcal{C}_{\text{zaatar}}$: Ginger and Zaatar constraint sets for Ψ (§C.2)
$\lvert Z_{\text{ginger}}\rvert$: number of variables in $\mathcal{C}_{\text{ginger}}$ (§C.2)
$\lvert \mathcal{C}_{\text{ginger}}\rvert$ (= $\lvert Z_{\text{ginger}}\rvert$): number of constraints in $\mathcal{C}_{\text{ginger}}$ (§C.2)
K: number of additive terms in $\mathcal{C}_{\text{ginger}}$ (§C.2)
K_2 ($\leq K$): number of distinct additive degree-2 terms in $\mathcal{C}_{\text{ginger}}$ (§C.2)
ρ_{lin}, ρ: number of linearity tests, number of PCP repetitions
e, d: cost of encrypting, decrypting an element in $\mathbb{F}$ (§4.8.1)
h: cost of ciphertext add plus multiply (§4.8.1)
c: cost of pseudorandomly generating an element in $\mathbb{F}$ (§4.8.1)

T: running time of Ψ (§4.8.1)
$\lvert Z_{\text{zaatar}}\rvert$ (= $\lvert Z_{\text{ginger}}\rvert + K_2$): number of variables in $\mathcal{C}_{\text{zaatar}}$ (§C.2)
$\lvert \mathcal{C}_{\text{zaatar}}\rvert$ (= $\lvert Z_{\text{ginger}}\rvert + K_2$): number of constraints in $\mathcal{C}_{\text{zaatar}}$ (§C.2)
$\lvert x\rvert, \lvert y\rvert$: number of elements in input, output (§3.2.1)
β: batch size (number of instances) (§4.8.1)
$\ell = 3\rho_{\text{lin}} + 2$: number of (high-order) PCP queries in Ginger [114]
$\ell' = 6\rho_{\text{lin}} + 4$: number of (total) PCP queries in Zaatar (§C.1)
f: cost of multiplying in $\mathbb{F}$ (§4.8.1)
f_{div}: cost of division in $\mathbb{F}$ (§4.8.1)

Figure C.2: Costs to prover and verifier to verify a computation Ψ, under Zaatar and Ginger. Zaatar increases the number of constraints and variables, and requires additional bookkeeping for the prover and verifier, but these effects are dominated by a vast reduction in the proof encoding, so the Zaatar prover and verifier are far more efficient overall. The term K_2 is key to the comparison; this term is large only for degenerate computations (see text). The table assumes that the constraints $\mathcal{C}_{\text{ginger}}$ and $\mathcal{C}_{\text{zaatar}}$ have been generated by our compilers. *Computation-specific* queries refer to those that depend on the structure of the computation and its constraints, while *computation-oblivious* queries depend only on the length of the proof vector.

We can obtain constraints $\mathcal{C}_{\text{zaatar}}$ in quadratic form, given a set of Ginger constraints $\mathcal{C}_{\text{ginger}}$; indeed, our compiler first compiles to Ginger constraints and then performs the following transformation. For every constraint in $\mathcal{C}_{\text{ginger}}$, we retain all of the degree-1 terms and replace all degree-2 terms with a new variable. For example, if a constraint in $\mathcal{C}_{\text{ginger}}$ is $\{3 \cdot Z_1 Z_2 + 2 \cdot Z_3 Z_4 + Z_5 - Z_6 = 0\}$, then $\mathcal{C}_{\text{zaatar}}$ would replace that constraint with the following constraints, which are all in quadratic form: $\{(3 \cdot Z'_1 + 2 \cdot Z'_2 + Z_5) \cdot (1) = Z_6,\ Z_1 Z_2 = Z'_1,\ Z_3 Z_4 = Z'_2\}$.

We bound the number of variables and constraints in $\mathcal{C}_{\text{zaatar}}$ as follows. Letting $|Z_{\text{zaatar}}|$ (resp., $|Z_{\text{ginger}}|$) equal the number of variables in $\mathcal{C}_{\text{zaatar}}(X{=}x,\ Y{=}y)$ (resp., $\mathcal{C}_{\text{ginger}}(X{=}x,\ Y{=}y)$), by construction of $\mathcal{C}_{\text{zaatar}}$ we have $|Z_{\text{zaatar}}| = |Z_{\text{ginger}}| + K_2$, where K_2 is the number of distinct degree-2 terms that appear in all of $\mathcal{C}_{\text{ginger}}$. Similarly, $|\mathcal{C}_{\text{zaatar}}| = |\mathcal{C}_{\text{ginger}}| + K_2$.

We analyze the drop in proof vector size at the end of this section; see also the first two lines of Figure C.2.

The prover's work. Because of the shorter proof vector length, the prover's work to reply to queries drops, usually dramatically (see the "Issue responses" row in Figure C.2). However, the prover has an additional cost under Zaatar.

The prover must compute the coefficients of the polynomial $H_{x,y,z}(t) = P(t,z)/D(t)$ (see Section 4.6). As a starting point in this computation, the prover knows values taken by the polynomials $\{A_i(t), B_i(t), C_i(t)\}$ and $\{A'(t), B'(t), C'(t)\}$ at well-known values of t. Using operations based on the FFT (interpolation [85], polynomial multiplication [51], and polynomial division), the prover obtains the coefficients of $H_{x,y,z}(t)$ in time $\approx 3 \cdot f \cdot (|\mathcal{C}_{\text{zaatar}}| \cdot \log^2 |\mathcal{C}_{\text{zaatar}}|)$, as depicted in Figure C.2. The process is detailed in Appendix C.4.

The verifier's work. Like the prover, the verifier in Zaatar also gains from the shorter proof vector; see the "Computation-oblivious queries" line in Figure C.2. However, the Zaatar verifier incurs two additional costs, which we summarize immediately below and detail in Appendix C.4.

The first is the cost to construct the query, which is depicted in the "Computation-specific queries" line in the figure. Under Ginger, the verifier must compute γ_1 and γ_2 in order to issue a circuit query (§3.2.1); this requires generating a pseudorandom number for each constraint and then multiplying it with each term in the given constraint, yielding amortized costs of $\rho \cdot (c \cdot |\mathcal{C}_{\text{ginger}}| + f \cdot K)/\beta$ for a batch of size β. The analog under Zaatar is computing queries to z and to h, which costs for the batch $\rho \cdot (c + (f_{div} + 5f) \cdot |\mathcal{C}_{\text{zaatar}}| + f \cdot K + 3f \cdot K_2)$.

The second cost is that Zaatar's verifier requires two more operations per input and output, owing to the details of query construction (see Appendices C.1 and C.4).

Detailed analysis of $|u|$. For a given computation, Ginger's proof vector has length $|u_{\text{ginger}}| = |Z_{\text{ginger}}| + |Z_{\text{ginger}}|^2$ (per Section 3.2.1). By contrast, Zaatar's proof vector has length $|u_{\text{zaatar}}| = |Z_{\text{zaatar}}| + |\mathcal{C}_{\text{zaatar}}|$ (per Section 4.6); recalling that $|Z_{\text{zaatar}}| = |Z_{\text{ginger}}| + K_2$ and $|\mathcal{C}_{\text{zaatar}}| = |\mathcal{C}_{\text{ginger}}| + K_2$, we can write $|u_{\text{zaatar}}| = |Z_{\text{ginger}}| + |\mathcal{C}_{\text{ginger}}| + 2K_2$. However, $|\mathcal{C}_{\text{ginger}}| \approx |Z_{\text{ginger}}|$, and we will in fact take $|\mathcal{C}_{\text{ginger}}| = |Z_{\text{ginger}}|$: our compiler, when configured to output Ginger constraints, creates roughly one new variable for each constraint that it introduces.[2] Thus, we get $|u_{\text{zaatar}}| = 2 \cdot (|Z_{\text{ginger}}| + K_2)$.

To compare $|u_{\text{zaatar}}|$ to $|u_{\text{ginger}}|$, we make three points. First, $|u_{\text{zaatar}}|$ is less than $|u_{\text{ginger}}|$ as long as $K_2 < K_2^* \overset{\text{def}}{=} (|Z_{\text{ginger}}|^2 - |Z_{\text{ginger}}|)/2$. Indeed, we expect that for most computations, K_2 will be far smaller than K_2^*. Roughly speaking, this fails to occur only when the computation involves adding the product of many multiplications; the reason is that (a) K_2^* corresponds to a computation in which the average number of *distinct* degree-2 terms per Ginger constraint is $(|Z_{\text{ginger}}| - 1)/2$, and (b) our compiler produces a Ginger constraint with more than $(|Z_{\text{ginger}}| - 1)/2$ terms only when compiling a program excerpt that involves summing many terms that are degree-2 (or higher).[3] If most of the constraints have this form, it means that most of the computation involves such sums, which is a degenerate case. An example is degree-2 polynomial evaluation, for which the Ginger encoding is actually very concise [113].

Second, even in the degenerate cases, $|u_{\text{zaatar}}|$ is very close to $|u_{\text{ginger}}|$. The worst case is when K_2 is maximal, which happens when every pair of variables in Z_{ginger} appears as a degree-2 term in $\mathcal{C}_{\text{ginger}}$; that is, the maximum value of K_2 is $K_2 = |Z_{\text{ginger}}| \cdot (|Z_{\text{ginger}}| + 1)/2$. Recalling that $|u_{\text{zaatar}}| = 2 \cdot (|Z_{\text{ginger}}| + K_2)$, we get $|u_{\text{zaatar}}| \le 2 \cdot |Z_{\text{ginger}}| + |Z_{\text{ginger}}| \cdot (|Z_{\text{ginger}}| + 1) = 3 \cdot |Z_{\text{ginger}}| + |Z_{\text{ginger}}|^2$. But $|u_{\text{ginger}}| = |Z_{\text{ginger}}| + |Z_{\text{ginger}}|^2$, so $|u_{\text{zaatar}}| \le |u_{\text{ginger}}| \cdot (1 + 2/(|Z_{\text{ginger}}| + 1))$, which is indeed close to $|u_{\text{ginger}}|$.

Third, for all of the computations that we investigate and evaluate, K_2 is far smaller than K_2^* (i.e., we are nowhere close to the degenerate cases), leading to vast improvements in the length of the proof vector, and hence breakeven batch sizes.

[2]A careful analysis of the compiler indicates that the actual bound is $|\mathcal{C}_{\text{ginger}}| \le (1 + \alpha) \cdot |Z_{\text{ginger}}|$, for $\alpha = 4/(\log_2 |\mathbb{F}| + 3)$. However, our fields are large (§4.8.1), which is why the text treats α as equal to 0.

[3]The other constructs that produce constraints with degree-2 terms (< and ==) produce only an average of one or two distinct degree-2 terms per constraint and add at least twice as many new variables.

C.3 Correctness

A verifier $\mathcal{V}$ is given access to a proof oracle π, which is supposed to establish the satisfiability of $\mathcal{C}(X{=}x, Y{=}y)$.

Lemma C.3.1 (Completeness). If $\mathcal{C}(X{=}x, Y{=}y)$ is satisfiable, if $\pi = (\pi_z, \pi_h)$ is constructed as above, and if $\mathcal{V}$ proceeds according to Figure C.1, then $\Pr\{\mathcal{V} \text{ accepts}\} = 1$.

Proof. If π is constructed properly, then it is a linear function, so it passes the linearity tests. Next we consider the divisibility test. Let $w = (x, y, z)$. If $\mathcal{C}(X{=}x, Y{=}y)$ is satisfiable by z, then by Claim C.1.1, there exists $H_w(t)$ such that $D(t) \cdot H_w(t) = P_w(t)$. Also, if π is constructed properly, then $\mathcal{V}$ obtains $\pi_z(q_a)$ as $\pi_z(q_1) - \pi_z(q_5)$, where $\pi_z(q_a) = \sum_{i=1}^{n'} w_i \cdot A_i(\tau)$. Similarly, $\mathcal{V}$ obtains $\pi_z(q_b)$ as $\pi_z(q_2) - \pi_z(q_5)$, where $\pi_z(q_b) = \sum_{i=1}^{n'} w_i \cdot B_i(\tau)$. Likewise $\pi_z(q_3) - \pi_z(q_5) = \pi_z(q_c) = \sum_{i=1}^{n'} w_i \cdot C_i(\tau)$. Finally, $\mathcal{V}$ obtains $\pi_h(q_d)$ similarly, where $\pi_h(q_d) = \sum_{j=0}^{|\mathcal{C}|} h_j \cdot \tau^j = H_w(\tau)$. Thus, the divisibility test is checking the following:

$$D(\tau) \cdot H_w(\tau) \overset{?}{=} \left(\sum_{i=0}^{n} w_i \cdot A_i(\tau)\right) \cdot \left(\sum_{i=0}^{n} w_i \cdot B_i(\tau)\right) - \left(\sum_{i=0}^{n} w_i \cdot C_i(\tau)\right) = P_w(\tau), \qquad \text{(C.1)}$$

which holds, since $D(t) \cdot H_w(t) = P_w(t)$. Thus, both tests pass, so $\mathcal{V}$ accepts. $\square$

Lemma C.3.2 (Soundness). There exists a constant $\kappa < 1$ such that if $\mathcal{C}(X{=}x, Y{=}y)$ is not satisfiable and if $\mathcal{V}$ proceeds according to Figure C.1, then $\Pr\{\mathcal{V} \text{ accepts}\} < \kappa$ for *all* purported proof oracles $\tilde{\pi}$.

Proof. Assume for now that $\tilde{\pi}$ is a linear function (this restricts the proof oracle's power to cheat, and we will revisit this in a moment). Consider $\tilde{\pi}(q_a)$, $\tilde{\pi}(q_b)$, and $\tilde{\pi}(q_c)$; these equal $\langle q_a, \tilde{z}\rangle$, $\langle q_b, \tilde{z}\rangle$, and $\langle q_c, \tilde{z}\rangle$, for some $\tilde{z}$ chosen by the prover. Likewise, $\tilde{\pi}(q_d)$ equals $\widetilde{H}(\tau)$, for some polynomial $\widetilde{H}(t)$, chosen by the prover. Thus, the divisibility test is checking whether $D(\tau) \cdot \widetilde{H}(\tau) = P_{\tilde{w}}(\tau)$, for $\tilde{w} = (x, y, \tilde{z})$.

However, $\mathcal{C}(X{=}x, Y{=}y)$ is not satisfiable, so there is no $\widetilde{H}(t)$ for which $D(t) \cdot \widetilde{H}(t) = P_{\tilde{w}}(t)$ (by Claim C.1.1). Thus, Equation (C.1) holds only if τ is a root of the polynomial $Q(t) = P_{\tilde{w}}(t) - D(t) \cdot \tilde{H}(t)$. But by the Schwartz-Zippel lemma, $\Pr_\tau\{Q(\tau) = 0\} \leq 2 \cdot |\mathcal{C}|/|\mathbb{F}|$, since the degree of $Q(t)$ is bounded by $2 \cdot |\mathcal{C}|$ and τ is randomly chosen from $\mathbb{F}$.

We now address the possibility that $\tilde{\pi}$ is not a linear function. Take $\kappa > \max\{(1 - 3\delta + 6\delta^2)^{\rho_{\text{lin}}}, 6\delta + 2 \cdot |\mathcal{C}|/|\mathbb{F}|\}$, for $0 < \delta < \delta^*$, where δ^* is the lesser root of $6\delta^2 - 3\delta + 2/9 = 0$. The following claim implies the lemma: if, for some π, the tests pass with probability greater than κ, then $\mathcal{C}(X{=}x, Y{=}y)$ is satisfiable. We will now argue this claim.

If the linearity tests pass with probability greater than $(1-3\delta+6\delta^2)^{\rho_{\text{lin}}}$, then π is δ-close to linear; this follows from results of Bellare et al. [22, 23]; see the analysis in [114, Apdx. A.2]. Next, consider the divisibility correction test (DCT). Assuming the proof oracle is δ-close to linear, the probability of passing the DCT is $> \kappa \geq 6\delta + 2\cdot|\mathcal{C}|/|\mathbb{F}|$. But the probability that any of the six queries in this test $(q_1, q_2, q_3, q_4, q_5, q_8)$ "hit" π where it is not linear is upper-bounded by 6δ, by the union bound. So with probability $> 2\cdot|\mathcal{C}|/|\mathbb{F}|$, the tests pass if querying the closest linear function to π. But then $\mathcal{C}(X{=}x, Y{=}y)$ is satisfiable because, as argued above, if $\mathcal{C}(X{=}x, Y{=}y)$ is not satisfiable and $\mathcal{V}$ queries a linear function, the probability of passing the tests $\leq 2\cdot|\mathcal{C}|/|\mathbb{F}|$. □

As in [114, Apdx A.2] we choose δ to minimize cross-over points. We neglect the ratio $2\cdot|\mathcal{C}|/|\mathbb{F}|$, since $|\mathcal{C}|$ roughly captures the size of the computation and $|\mathbb{F}|$ is astronomical (e.g., $|\mathbb{F}| = 2^{192}$). We choose $\delta = 0.0294$ and $\rho_{\text{lin}} = 20$, and hence $\kappa = 0.177$ suffices. We then take $\rho = 8$ for an upper-bound on soundness error of $\kappa^{\rho} < 9.6\times10^{-7}$.

C.4 Costs in more detail

Earlier in the paper (Figure C.2 and Section C.2), we stated the costs of Zaatar. This section fleshes out some of those claims.

We repeat the observation of Gennaro et al. [63] that the polynomials $\{A_i(t)\}, \{B_i(t)\}$, and $\{C_i(t)\}$ can be represented efficiently, in terms of their evaluations at the $\{\sigma_j\}$. That is, $A_i(t)$ can be written as a list $\{(j, a_{i,j}) \mid a_{i,j} \neq 0, j \in \{1, \ldots, |\mathcal{C}|\}\}$, and similarly for $B_i(t)$ and $C_i(t)$. For convenience, we let $\sigma_0 = 0$ and $a_{i,0} = b_{i,0} = c_{i,0} = 0$.

The prover. To construct the π_h component of its proof vector (§C.1), the prover must compute the coefficients of $H_w(t)$, where $D(t)\cdot H_w(t) = P_w(t)$. Section C.2 states that the cost to do so is $3\cdot f\cdot|\mathcal{C}|\cdot\log^2|\mathcal{C}|$. We now detail the process. It is three steps (well-explained in [94, Chapter 1.7]).

Step 1 is writing $P_w(t)$ in the form $A(t)\cdot B(t) - C(t)$, for degree-$|\mathcal{C}|$ polynomials $\{A(t), B(t), C(t)\}$, to obtain the coefficients of $\{A(t), B(t), C(t)\}$. The prover constructs the set $\{(\sigma_j, \sum_i w_i \cdot a_{i,j}) \mid j \in \{0, \ldots, |\mathcal{C}|\}$, which is the evaluations of $A(t)$. The prover then uses multipoint interpolation [85, Chapter 4.6.4] to compute the coefficients of $A(t)$, in time $\approx f\cdot|\mathcal{C}|\cdot\log^2|\mathcal{C}|$. The prover does likewise for $B(t)$ and $C(t)$. Step 2 is computing the coefficients of $P_w(t)$ in time $\approx f\cdot|\mathcal{C}|\cdot\log|\mathcal{C}|$, using multiplication based on the fast Fourier transform

(FFT) [51]. Step 3 is computing the coefficients of $P_w(t)/D(t) = H_w(t)$ in time $\approx f \cdot |\mathcal{C}| \cdot \log |\mathcal{C}|$, using FFT-based polynomial division.

The verifier. We will be focused on Figure C.1. We stated $\mathcal{V}$'s query setup costs as (§C.2):

$$c + (f_{div} + 5f) \cdot |\mathcal{C}| + f \cdot K + f \cdot 3K_2.$$

We now explain these costs. Selecting τ costs c. Generating $q_d = (1, \tau, \dots, \tau^{|\mathcal{C}|})$ costs $f \cdot |\mathcal{C}|$. Most of the remaining costs are generating $(A_0(\tau), A_1(\tau), \dots, A_n(\tau))$, $(B_1(\tau), \dots, B_n(\tau))$, and $(C_1(\tau), \dots, C_n(\tau))$.

Gennaro et al. [63] observe that a Lagrange basis is useful for this purpose; we give the details here. We can write each polynomial $A_i(t)$ as follows (and analogously for $B_i(t), C_i(t)$):

$$A_i(t) = \sum_{j=0}^{|\mathcal{C}|} a_{i,j} \cdot \ell_j(t), \quad \text{where } \ell_j(t) = \prod_{\substack{0 \le k \le |\mathcal{C}| \\ k \ne j}} \frac{t - \sigma_k}{\sigma_j - \sigma_k}.$$

We can use Barycentric Lagrange interpolation [35] to write:

$$\begin{aligned} A_i(t) &= \ell(t) \cdot \sum_{j=0}^{|\mathcal{C}|} a_{i,j} \cdot \frac{v_j}{t - \sigma_j}, \text{ where} \\ \ell(t) &= (t - \sigma_0)(t - \sigma_1) \cdots (t - \sigma_{|\mathcal{C}|}), \text{ and} \\ v_j &= 1 / \prod_{\substack{0 \le k \le |\mathcal{C}| \\ k \ne j}} (\sigma_j - \sigma_k). \end{aligned}$$

We now explain the remaining costs. Computing $\ell(\tau)$ takes $|\mathcal{C}|$ multiplications; then, computing $D(\tau)$ takes one division and one multiplication, as $D(\tau) = (1/\tau) \cdot \ell(\tau)$. Computing $\{v_j\}$ can be done efficiently via a careful choice of the $\{\sigma_j\}$ (the protocol permits *any* distinct, non-zero values here): if we arrange for $\sigma_1, \dots, \sigma_{|\mathcal{C}|}$ to follow an arithmetic progression (a convenient choice is $1, 2, \dots, |\mathcal{C}|$), then computing $1/v_{j+1}$ from $1/v_j$ requires only two operations. Since one can compute $1/v_0$ using $|\mathcal{C}|$ multiplications, the total time to compute the $\{v_j\}$ is $(f_{div} + 3f) \cdot |\mathcal{C}|$ operations.

Finally, given $\{v_j\}$ and $\ell(\tau)$ and using the representation above, one can compute $\{A_i(\tau)\}$, $\{B_i(\tau)\}$, and $\{C_i(\tau)\}$ with a number of multiplications equal to the total number of non-zero $\{a_{i,j}, b_{i,j}, c_{i,j}\}$. This number is computation-dependent (see §C.1), but we can bound it in our framework. Recall that our compiler obtains Zaatar constraints by transforming Ginger constraints (§C.2). The Zaatar constraints, when written in quadratic form, induce no more than $K + 3K_2$ non-zero $\{a_{i,j}, b_{i,j}, c_{i,j}\}$, where K and K_2 are as defined in Section C.2.

In Section C.2, we stated that the verifier requires three operations per input and output. This cost comes from computing the following quantities in the divisibility correction test: $\sum_{n'+1}^{n} w_i \cdot A_i(\tau)$, $\sum_{n'+1}^{n} w_i \cdot B_i(\tau)$, and $\sum_{n'+1}^{n} w_i \cdot C_i(\tau)$.

Appendix D

Pantry's correctness, primitives, and applications

D.1 Pantry's correctness

This appendix and the next will establish Pantry's correctness. These arguments mainly draw on existing techniques and folklore; we write them down here for completeness.

We wish to establish that Pantry's verifier $\mathcal{V}$ accepts correct outputs y and rejects incorrect ones with probability similar to that of Zaatar's soundness (§5.1.1, §5.1.3). By the Completeness property of Zaatar (Appendix C) and an equivalent property in Pinocchio [63, 104], and the implementation of the prover $\mathcal{P}$ (specifically, the use of the map S), $\mathcal{V}$ can be made to accept correct outputs with certainty. The more involved step is showing that $\mathcal{V}$ rejects incorrect answers. One might think to apply the soundness property (§5.1.2), but this property is not enough: its technical guarantee is that *if no satisfying assignment exists*, then $\mathcal{V}$ is likely to reject (Appendix C, [63, 104]). Meanwhile, $\mathcal{C}(X{=}x, Y{=}y)$ could be satisfiable, even if y is incorrect in the context of steps 1 and 2 (§5.1.2). As a simple example, imagine that the computation Ψ is:

```
name = PutBlock(x);
B    = GetBlock(name);
if (B == x)
    y = 1;
else
    y = 0;
return y;
```

The correct answer here is $y{=}1$. But $y{=}0$ also results in many satisfying assignments to $\mathcal{C}_\Psi(X{=}x, Y{=}0)$; in particular, any setting of the B variables for which $H(B){=}H(x){=}name$, where $B{\neq}x$, will satisfy $\mathcal{C}_\Psi(X{=}x, Y{=}0)$. Since soundness says nothing about what $\mathcal{V}$ does when there *are* satisfying assignments, soundness cannot be used to argue that $\mathcal{V}$ will reject $y = 0$.

We need another property, called *proof of knowledge* (PoK). A formal definition is below; less formally, this property states that if $\mathcal{P}$ can make $\mathcal{V}$ accept a claimed output y with non-negligible probability, then there is an efficient algorithm that can run $\mathcal{P}$ to produce a satisfying assignment to $\mathcal{C}(X{=}x, Y{=}y)$. Even more informally, one can think of this property as stating that if $\mathcal{V}$ accepts the interaction, then $\mathcal{P}$ must have "known" an assignment.

The power of the PoK property in our context is the following. If y is an incorrect output and $\mathcal{C}(X{=}x, Y{=}y)$ is satisfiable, *the only satisfying assignments contain memory consistency violations*; meanwhile, memory consistency violations imply hash collisions, and manufacturing such collisions is presumed to be hard. Therefore, no efficient algorithm can produce satisfying assignments of this adverse form, and hence (by the italicized assertion) no efficient algorithm can produce any satisfying assignments, and hence—here is where we use the PoK property—the prover cannot systematically make the verifier accept the corresponding output. Very informally, the prover must not "know" any adverse satisfying assignments, which, by the PoK property, implies that it cannot make the verifier accept them.

In the rest of this appendix, we formally define a PoK property and use it to establish Pantry's correctness; Appendix D.2 proves that Pantry meets this property. We will restrict attention to the case that Pantry uses Zaatar; a similar analysis applies when Pantry uses Pinocchio.[1]

D.1.1 Setup and definition of proof-of-knowledge

Recall the Zaatar setup. $\mathcal{V}$ and $\mathcal{P}$ are given a set of constraints $\mathcal{C}$ (over variables X, Y, Z), input x, and output y. $\mathcal{C}(X{=}x, Y{=}y)$ is a set of constraints over variables $Z = (Z_1, \ldots, Z_{n'})$; each $Z_i \in \mathbb{F}$. $\mathcal{V}$ and (a possibly incorrect $\mathcal{P}$) interact. If, after getting the purported output y, $\mathcal{V}$ accepts, we notate that as $(\mathcal{V}, \mathcal{P})(\mathcal{C}, x, y) = 1$.

Definition D.1.1 (Proof of knowledge (PoK).). There exists a PPT *extractor algorithm E* (which is presumed to have oracle access to the prover: it can run the prover by supplying arbitrary patterns) for which the following holds. For all $\mathcal{P}$ and all polynomially-bounded

[1]Pinocchio has been shown to have a PoK property [63, §8]. This PoK property is stronger than the one that we prove for Zaatar (though Pinocchio's relies on non-falsifiable "knowledge assumptions" whereas Zaatar's relies on standard assumptions). As a consequence, the analysis in this appendix also applies to Pantry's use of Pinocchio.

$(\mathcal{C}, x, y)$, if

$$\Pr\{(\mathcal{V}, \mathcal{P})(\mathcal{C}, x, y) = 1\} > \epsilon_K$$

then

$$\Pr_s\{E_s^{\mathcal{P}}(\mathcal{C}, x, y) \to z = z_1, \ldots, z_{n'}, \text{ such that } z \text{ satisfies } \mathcal{C}(X{=}x, Y{=}y)\} > \epsilon'_K,$$

where ϵ'_K is non-negligible. The first probability is taken over the random choices of the Zaatar protocol (specifically, the coin flips of the commit phase, the decommit phase, and the choice of PCP queries). The second probability is taken over s, the random choices of the extractor algorithm E.

The next appendix proves that Zaatar has this property; for now, we take it as a given. As we will see below, the quantity ϵ_K will wind up being Pantry's actual error: it will upper-bound the probability that $\mathcal{V}$ accepts an incorrect output. Sometimes this parameter is referred to as "knowledge error", and we will be motivated to ensure that it is not much larger than the soundness error. Notice that we cannot make this parameter lower than the soundness error, since a protocol that has knowledge error of at most ϵ_K has soundness error of at most ϵ_K (that is, PoK implies soundness). This is because if no satisfying assignment exists at all, then of course the probability of producing one is zero (for all algorithms), which implies (by PoK) that $\mathcal{V}$ rejects with probability at least $1 - \epsilon_K$, which yields the soundness property.

D.1.2 $\mathcal{V}$ rejects incorrect outputs

This section considers only single executions; the next section generalizes to the case of state carried across program executions.

We will use the PoK property (Defn. D.1.1) to establish that $\mathcal{V}$ rejects semantically incorrect outputs y' with high probability. In the context of a computation Ψ, the (unique) semantically correct output y on input x is the value or vector that results from following the logic of Ψ on input x. This logic includes *program logic* and *storage consistency*. Program logic means, for example, that the result of an "add" operation should actually be the sum of the two numbers.

Storage consistency is a typical definition: "reads should see writes". In our context, this means that if the program "reads address n" (that is, executes `GetBlock` with input n), then the return value b should be the "most recently written value to address n" (that is, the program should have executed n = `PutBlock`(b), and between that call and the GetBlock, there should be no intervening invocations $n = \texttt{PutBlock}(b')$, where $b' \neq b$). If an input x would cause Ψ to issue a call $\texttt{GetBlock}(n)$ for which there was no preceding call $n = \texttt{PutBlock}(b)$, then there is no semantically correct output; in this situation, we sometimes say that the correct output is $\perp$ and that x itself as a semantically incorrect input.

Of course, the preceding notions require an ordering on operations; this order follows from program order, and induces an ordering on the constraints that the Pantry compiler produces. In more detail, recall that for a high-level program Ψ, the Pantry compiler produces constraints $\mathcal{C}$ that correspond to Ψ's program logic: the program variables in Ψ appear in $\mathcal{C}$, and the equations in $\mathcal{C}$ enforce program logic through the relations among the program variables. (The constraints $\mathcal{C}$ are said to be *equivalent* to the computation Ψ.) An assignment $w = (x, y, z)$ to $\mathcal{C}$ thus corresponds to a *transcript* for Ψ: a string consisting of the program Ψ with loops unrolled and with all variables $(X, Y, Z_1, Z_2, \ldots)$ replaced with values $(x, y, z_1, z_2, \ldots)$. In what follows, we will move back and forth between the notion of transcript τ and its corresponding assignment $w_\tau = (x, y, z)$.

A *valid transcript* is one that obeys program semantics. Specifically, in a valid transcript τ:

P1 *All operations respect program logic.* By the transcript-assignment equivalence, this property is equivalent to saying that the assignment $w_\tau = (x, y, z)$ satisfies the constraints $\mathcal{C}$.

P2 *Storage operations respect consistency.* Specifically, if $b = \texttt{GetBlock}(n)$ appears in τ, then an operation $n = \texttt{PutBlock}(b)$ appears earlier in τ (with no intervening $n = \texttt{PutBlock}(b')$, where $b' \neq b$).

Claim D.1.1. For a computation Ψ, if $y \neq \perp$ is semantically correct on input x, then there exists a valid transcript in which the input variables are set to x and the output variables are set to y. (Also, this transcript is unique in our present context.)

Proof. The transcript is an unrolled program execution. So if the program Ψ would correctly produce y from x, then we can write down all of the operations that lead from x to y. This list will respect validity (properties P1 and P2), since validity admits those transcripts (and only those transcripts) that obey the semantics.

The transcript is unique since each operation, when executed correctly, is deterministic. Note in particular that storage operations are deterministic: PutBlock operations are deterministic by construction (given an input block, PutBlock returns a digest of it), and the semantics given above specify the unique return value of a GetBlock invocation. □

Claim D.1.2. Let $\mathcal{V}$ be Pantry's verifier, operating on constraints $\mathcal{C}$ and input x. If $y \neq \perp$ is the semantically correct output, then for all provers $\mathcal{P}$ and all $y' \neq y$, $\Pr\{(\mathcal{V}, \mathcal{P})(\mathcal{C}, x, y') = 1\} \leq \epsilon_K$.

Proof. Assume otherwise. Then there exists a prover $\mathcal{P}'$ and an incorrect answer y' for which $\Pr\{(\mathcal{V}, \mathcal{P}')(\mathcal{C}, x, y') = 1\} > \epsilon_K$. By the PoK property (Defn D.1.1, Lemma D.2.1), there exists an extractor algorithm $E^{\mathcal{P}'}$ that, with probability greater than ϵ'_K, produces some assignment z' such that (x, y', z') satisfies $\mathcal{C}$; let τ' be the transcript corresponding to the assignment $w'_{\tau'} = (x, y', z')$. Also, since $y \neq \perp$ is semantically correct, Claim D.1.1 implies that there exists a valid transcript τ (while τ is unique, we will not explicitly rely on that uniqueness below). By the validity of τ, there is an assignment $w_\tau = (x, y, z)$ that satisfies $\mathcal{C}$.

Compare τ and τ'. Consider the first position in these strings where they disagree (they must disagree somewhere, for their outputs are different). We now make two claims about this point of divergence: (1) it must be a `GetBlock`(n) operation, and (2) the input to this operation must be the same in both τ and τ'.

The reason for (1) is that if τ and τ' first disagreed on a different operation (either its inputs or outputs), they would agree up until that operation, and then disagree on a deterministic operation (all operations besides GetBlock are deterministic); hence, at least one of the two transcripts would be in violation of program logic, which would mean that at least one of w_τ and $w'_{\tau'}$ would not satisfy $\mathcal{C}$, which would contradict statements above. Similarly, to establish (2), observe that the constraints are constructed so that the input to GetBlock is deterministically produced from the computation's input (x) and the computation up to that point (and τ and τ' agree up to that point).

From claims (1) and (2), the output of the GetBlock in τ (call it b) and in τ' (call it b') are different; that is, $b \neq b'$. However, w and w' are both satisfying, so τ and τ' obey property P1. From the compilation of GetBlock into $\mathcal{C}_{H^{-1}}$, and the construction of $\mathcal{C}_{H^{-1}}$, per Section 5.2, we have $n = H(b)$ and $n = H(b')$, where H is a collision-resistant hash function (CRHF). Also, because τ is valid, it obeys P2, which means that τ contains an earlier instance of $n = \text{PutBlock}(b)$, where (by P1) $H(b) = n$. But τ and τ' match through that earlier point in the transcript, which means that τ' also contains $n = \text{PutBlock}(b)$. Thus, τ' contains b and b', with $b' \neq b$ and $H(b) = H(b')$.

Therefore, an adversarial algorithm $\mathcal{A}$ can produce a collision in H as follows. $\mathcal{A}$ runs $E^{\mathcal{P}'}$ to get z' (which succeeds with $> \epsilon'_K$ probability), forms $w = (x, y', z')$, sorts w by output digests, scans to find b and b', and outputs them. This succeeds in producing a collision with probability $> \epsilon'_K$, which contradicts the assumed collision-resistance of H. □

D.1.3 Remote state

Arguing the correctness of Pantry's MapReduce (§5.3), among other applications, requires allowing state to be carried across executions. To this end, we generalize the definitions above.

We consider a model in which $\mathcal{V}$ and $\mathcal{P}$ interact sequentially: $\mathcal{V}$ supplies input x_0 and specifies Ψ_0 to $\mathcal{P}$, receiving output y_0; next, $\mathcal{V}$ supplies input x_1 and specifies Ψ_1 to $\mathcal{P}$, receiving output y_1, etc. Suppose that there are $t+1$ pairs in all: $(x_0, y_0), \ldots, (x_t, y_t)$.

We define the semantic correctness of y_i inductively. Specifically, we say that y_0 is semantically correct if it meets the earlier description (i.e., if the correct operation of Ψ_0 on input x_0 produces y_0). For y_i, where $i > 0$, we say that y_i is semantically correct if (a) all previous $\{(x_j, y_j)\}_{j=0}^{i-1}$ are semantically correct; (b) y_i respects program logic on x_i; and (c) if Ψ_i issues $\texttt{GetBlock}(n)$, then the return value should be the b in the most recent $n = \texttt{PutBlock}(b)$ call, as above; here, however, we are looking not only at the current execution but at the concatenated (valid) transcripts $\tau_0, \ldots, \tau_{i-1}$ together (these transcripts exist by the correctness of $y_0, \ldots, y_{i-1}$).

Label with $\mathcal{C}_i$ the constraints that correspond to computation Ψ_i. We now make a claim that is analogous to Claim D.1.2:

Claim D.1.3. Consider a sequence of interactions between $\mathcal{V}$ and $\mathcal{P}$ that produces pairs $(x_0, \hat{y}_0), \ldots, (x_t, \hat{y}_t)$, where for $i \in \{0, \ldots, t\}$, the semantically correct output y_i is not $\perp$. For all provers $\mathcal{P}$, and all i, if $\hat{y}_i \neq y_i$, then for some $j \leq i$, we have $\Pr\{(\mathcal{V}, \mathcal{P})(\mathcal{C}_j, x_j, \hat{y}_j) = 1\} \leq \epsilon_K$.

Proof. (Sketch.) The proof is similar to that of Claim D.1.2. Let $\hat{y}_i$ be the first semantically incorrect output in the sequence. Assume to the contrary that $\Pr\{(\mathcal{V}, \mathcal{P})(\mathcal{C}_j, x_j, \hat{y}_j) = 1\} > \epsilon_K$, for all $j \in \{0, \ldots, i\}$; by the PoK property, E can produce, with probability greater than $(\epsilon'_K)^{i+1}$, a list of assignments $\hat{z}_0, \ldots, \hat{z}_i$ (which satisfy the respective constraint sets, given the respective inputs and outputs). Let $\hat{\tau}_0, \ldots, \hat{\tau}_i$ be the corresponding transcripts, and concatenate these together to form one large aggregate transcript, $\hat{\tau}_*$. There is a valid aggregate transcript τ_* that differs from $\hat{\tau}_*$ in at least one location (because $y_i \neq \hat{y}_i$).

As in Claim D.1.2, the two transcripts must again diverge in a GetBlock operation (all other operations are deterministic; furthermore, the inputs $\{x_0, \ldots, x_i\}$ match in the two transcripts, and so do the outputs $\{\hat{y}_0, \ldots, \hat{y}_{i-1}\}$, since $\hat{y}_i$ is the first semantically incorrect output in the sequence). This implies that $\hat{\tau}_*$ contains a collision. An adversarial PPT algorithm can thus produce a collision with probability at least $(\epsilon'_K)^{i+1}/t$ (by guessing i, running i instances of the extractor E, and sorting the resulting witnesses), in contradiction to the presumed collision-resistance of H. $\square$

The preceding analysis can be extended to cover the data structures that we build using the GetBlock and PutBlock abstractions (§5.4). In the case of the verifiable RAM, this analysis is a mild variant of the arguments for online memory-checking given by Blum et al. [40]. That paper specifies a simple memory semantics (roughly, reads and writes are totally ordered and each read is matched by a preceding write), describes a Merkle tree-based on-line checking algorithm, and argues that in order to violate the memory semantics an adversary must fake some of the hash checks that validate a path through the Merkle tree. Inspection of our verifiable RAM design (Section 5.4.1, Figure 5.7) indicates that violation of the memory semantics would result in a violation of Claim A.2.

Discussion. Notice that the preceding claims are conditional on $\mathcal{V}$ supplying correct inputs (i.e., a condition for the claims is that there *are* correct outputs). In particular, if the verifier supplies a made-up digest as a reference to storage, the protocol provides no guarantees. In practice, this means that the onus is on the verifier to supply correct digests as input.

Of course, if the verifier makes up a digest, then heuristically speaking, the prover will not be able to manufacture a satisfying assignment, since that would require inverting H. In fact, if the verifier chooses a digest d by random selection of b and then setting $d \leftarrow H(b)$, then we can show that the prover cannot convince the verifier to accept with greater than the knowledge error ϵ_K (this relies on the preimage-resistance, or one-wayness, of H, which is Ajtai's function [12]). By contrast, if the verifier chooses an input digest arbitrarily (perhaps in collusion with the prover!), then we cannot apply the preceding guarantees; however, cases where the verifier chooses a "wrong" digest for which it knows that the prover knows a preimage are elaborate exercises in shooting oneself in the foot.

Finally, note that the security proof for remote state presumes that either the same verifier is participating across the sequence, or that there is a chain of trust linking them. This issue is handled somewhat better in the non-interactive "proof-carrying data" (PCD) framework [38], where an extractor can produce a complete transcript, given a certificate. On the other hand, existing PCD protocols rely on non-falsifiable hypotheses.

D.2 Zaatar and proof-of-knowledge

This appendix will establish that Zaatar meets a proof-of-knowledge (PoK) property. Recall from the prior appendix that we are motivated to ensure that the knowledge error, ϵ_K, is not much larger than Zaatar's soundness error, ϵ_{zaat}; as established elsewhere (Appendix C),

$\epsilon_{\text{zaat}} = \epsilon_{\text{pcp}} + \epsilon_c$, where ϵ_{pcp} is the soundness error of the Zaatar PCP (approximately $5 \cdot 10^{-7}$), and ϵ_c is the error from the commitment protocol (for Zaatar, $\epsilon_c \approx 6000 \cdot \sqrt[3]{1/|\mathbb{F}|}$).

Lemma D.2.1. The Zaatar argument protocol has the PoK property with $\epsilon_K = 2 \cdot \epsilon_{\text{pcp}} + \epsilon_c$, and $\epsilon'_K = \left(\epsilon_{\text{pcp}}/2\right) \cdot \left(1 - n' \cdot e^{-100}\right)$.

Proof. The proof combines techniques from Barak and Goldreich (BG02) [19] and from the soundness proof of Zaatar (Appendices A.2 and C) and IKO [79] (which is Zaatar's base). We will assume familiarity with the technical details of Zaatar and IKO, but not of BG02. At a very high level, all of these protocols consist of a commit phase (in which the verifier makes the prover commit to an oracle, which is supposed to be the PCP) and a decommit phase; in the latter phase, the verifier submits the PCP queries.

The above works prove, loosely speaking, that at the end of the commit phase of the protocol, the prover is effectively bound to a particular (possibly inefficient) function *f*, from queries to responses. We face several technical difficulties in the present context. One of them is that just because *f* exists does not mean that it is easy to make the prover *respond* to queries. We will get around this issue by first showing that if there is a $> \epsilon_K$ probability of $\mathcal{V}$ accepting, then it must be true that for almost all of the possible queries, the prover responds with non-negligible probability. Then, loosely speaking, the extraction procedure will amplify the non-negligible probability to be near-certain. This is done by pumping the prover: feeding it many different interactions. Another difficulty is that when the extractor performs this pumping, we have to be sure that values *other* than the correct one will be sufficiently infrequent that the pumping process won't get confused; we get around this by reformulating the claims that the prover is bound to a function *f*.

The proof proceeds according to the following outline:

1. We will describe an extraction procedure, leaving a number of parameters unspecified.
2. We will analyze the extraction procedure and in so doing fill in the parameters. The analysis is in several parts:
    - We will reformulate some of the analysis of the binding properties of Zaatar (Appendix A.2).
    - We will define notions [19] of queries being "strong" (or weak) and "clear" (or confounding); these notions are relative to a given commit phase. We hope that in a *useful* commit phase, the vast majority of queries are both strong and clear; furthermore, we hope that a non-negligible fraction of commit phases are useful.

- We will show that in useful commit phases, the function that the prover is bound to is a valid PCP oracle that encodes a satisfying assignment and has a soundness error identical to our usual.
- We will show that in useful commit phases, the overwhelming majority of queries are strong.
- We will show that in useful commit phases, the overwhelming majority of queries are clear.
- The above results will be used to upper-bound ϵ_K and lower-bound ϵ'_K.

D.2.1 Preliminaries

There are three sets of random coin flips in the Zaatar protocol: c represents the random coin flips that determine the commit phase, d represents the random coin flips that determine the decommit phase, and r represents the random coin flips that determine the PCP queries. Often, we will assume that the coins for the commit phase have been flipped, and we will be working within a commit phase c.

Let A_i be the prover's response to the ith query, independent of whether the decommitment succeeds; when A_i depends on all three sources of randomness, we write $A_i(c, d, r)$. A common case is that we will be interested in A_i, within some commit phase (i.e., the commit coin flips will have already been determined); in that case, A_i is a function of (d, r) and can be written $A_i(d, r)$.

Whether $\mathcal{V}$ accepts is a random variable that is a function of (c, d, r). Likewise, *whether $\mathcal{V}$ decommits* (that is, whether the decommitment succeeds) is a random variable.

Let $Q_1(r), \ldots, Q_\mu(r)$ represent the μ PCP queries generated by a particular choice of the PCP verifier's coin flips, r. The Q_i are random variables, but of course they do not depend on c or d.

Let $\mathcal{V}_{\text{pcp}}$ denote Zaatar's PCP verifier. We will refer to $\mathcal{V}_{\text{pcp}}$ as generating queries and *accepting* their replies. (This can be formalized/notated with a query generation procedure $\mathcal{Q}((\mathcal{C}, x, y), r, i)$, which, given the PCP coin flips, returns the ith query. Similarly, we can write down a decision procedure $\mathcal{D}((\mathcal{C}, x, y), r, a_1, \ldots, a_\mu)$ that returns 1 or 0. While the notation is borrowed from BG02 [19], the formalization itself is standard in the PCP literature.)

A PCP admits *reverse sampling* (as defined in BG02 [19]) if, given a PCP query q and a position i, it is possible to choose the other PCP queries according to the random coins r, but holding q in the ith position. BG02 formalize this by saying that, given q, i, there is an efficient algorithm that can randomly and uniformly sample from all r such that $\mathcal{Q}((\mathcal{C}, x, y), r, i) = q$. In our context, it will be more helpful to think of the reverse sampling property as saying that for all q, i, it is possible to efficiently sample according to the conditional distribution $\{Q_1(r), \dots, Q_\mu(r)\}_{|Q_i(r)=q}$. Zaatar's PCP has the reverse sampling property.

D.2.2 The extraction procedure

See Figure D.1 for the extractor, E.

D.2.3 Analysis of the extractor

The binding of Zaatar, revisited

Following IKO [79], Zaatar's soundness analysis contains a binding game (Defn. A.2.1); a commitment protocol is *admissible* if for all *environments* (loosely speaking, an environment encapsulates the process of producing PCP queries), the probability of the prover winning the binding game is negligible. The definition in IKO and Zaatar is quantified over all deterministic environments.

In the present work, the binding game is now played inside an environment $\mathcal{E}$ that (a) chooses a distinguished query q and the positions i and i' deterministically (as previously), and (b) chooses the other queries $\vec{q}$ and $\vec{q'}$ *randomly*, according to a distribution of $\mathcal{E}$'s choosing. The definition of "S^* wins" is the same (outputting conflicting field values and successfully decommitting), and a protocol is now *admissible* if for all environments $\mathcal{E}$, the probability of S^* winning is less than $\epsilon_B = 1/|\mathbb{F}|$, where the probability now is taken over the coins r, r' that generate the two choices of queries as well as the three phases of the binding game (commit phase, and two runs of the decommit phase).

The new definition of admissible protocol (which quantifies over probabilistic environments) is, by averaging, equivalent to the old one (which quantifies over deterministic environments); IKO also observe this equivalence [79]. To see that meeting the old definition implies meeting the new one, observe that if the protocol doesn't meet the new property in some environment $\mathcal{E}$, then there must (in $\mathcal{E}$) be an adverse $\vec{q}$ and $\vec{q'}$ for which S^*'s probability of winning the old binding game is larger than ϵ_B, contradicting the old definition.

Next, we rerun some of the analysis in Zaatar, under probabilistic environments. Define $A_c(q, i, a) = \Pr_{d,r}\{A_i(d, r) = a \mid Q_i(r) = q\}$; this quantity is with respect to a particular commit phase c, and answers the question, "given that q is in the ith position, if we reverse sample to get the other queries and flip the decommit coins, what is the probability

```
// Goal is to produce a witness $z$ that satisfies $\mathcal{C}(X{=}x, Y{=}y)$
extract($\mathcal{P}$, $\mathcal{C}$, $x$, $y$):
    flip the "commit coins", and run the commit phase.
    // for the remainder of the procedure, we will be in this commit phase.

    for $t = 1, \ldots, n'$:     // extract the $t$th witness element
        for $k = 1, \ldots, T_1$:
            choose $q_r \in_R \mathbb{F}^{n'}$
            $q_s \leftarrow q_r + e_t$
            $\sigma_1 \leftarrow$ extract_response($q_r$, $\mathcal{C}$, $x, y$)
            $\sigma_2 \leftarrow$ extract_response($q_s$, $\mathcal{C}$, $x, y$)
            $z_t^{(k)} \leftarrow \sigma_2 - \sigma_1$
        if  a majority of $\{z_t^{(1)}, \ldots, z_t^{(T_1)}\}$ equal the same value $v$:
            $z_t \leftarrow v$
        else:
            abort()
    output $z_1, \ldots, z_n$

extract_response($q$, $\mathcal{C}$, $x$, $y$):
    for $i = 1, \ldots, \mu$:
        for $j = 1, \ldots, T_2$:
            • place $q$ in position $i$, and reverse sample to get full set of queries: $q_1, \ldots, q_\mu$.
 Here, $q_i = q$.
            • run $\mathcal{P}$ in the decommit phase, flipping decommit coins randomly.
            • if decommit succeeds, save the $i$th response, labeling it $\sigma^{(i,j)}$
        if more than $(\delta/3) \cdot T_2$ of the saved $\sigma^{(i,\cdot)}$ are equal, store the value, calling it a can-
 didate.

    if  there is exactly one candidate value, $\sigma$:
        return $\sigma$
    else:
        abort()
```

Figure D.1: Definition of knowledge extractor, E. It borrows techniques from the oracle recovery procedure of Barak and Goldreich [19]. For now, δ, T_1, T_2 are parameters. e_t is the vector with a 1 in component t and 0s elsewhere.

that the ith output is a?". Define $\mathsf{Ext}(c, q, i) = \text{argmax}_{a \in \mathbb{F}} A_c(q, i, a)$. Also, define $f_c(q)$ to be $\mathsf{Ext}(c, q, i^*)$, for some distinguished i^* (for example, $i^* = 1$).

Claim D.2.2. For all $q \in \mathbb{F}^{n'}$, $i \in [\mu]$, we have:

$$\Pr_{c}\left\{\Pr_{d,r}\{\{A_i(c,d,r) \neq f_c(q)\} \text{ and decommit happens} \mid Q_i(r) = q\} < \epsilon_3\right\} > 1 - \epsilon_3,$$

where $\epsilon_3 < 6 \cdot \sqrt[3]{1/|\mathbb{F}|}$.

Proof. (Sketch.) This claim is similar to Claim A.2.4. Essentially, wherever Zaatar's proofs for Claims B.3 and B.4 talk about "the probability over the decommit phase", one should write "...over the decommit phase and choice of $Q(r)$". Also, the binding game that enforces the probabilities is of course over five (not three) sets of random coin flips. □

Claim D.2.3 (Existence of $f_c(\cdot)$ and commit error). Define $\epsilon_c = 2 \cdot \mu \cdot \epsilon_3$.

$$\Pr_{c,d,r}\left\{\text{decommit happens and } \cup_{i=1}^{\mu}\{A_i(c,d,r) \neq f_c(Q_i(r))\}\right\} < \epsilon_c,$$

Proof. Fix $i \in [\mu]$. Claim D.2.2 implies that

$$\forall q\colon \Pr_{c,d,r}\{\{A_i(c,d,r) \neq f_c(q)\} \text{ and decommit happens} \mid Q_i(r) = q\} < 2\epsilon_3.$$

By an averaging argument, we get:

$$\Pr_{c,d,r}\{\{A_i(c,d,r) \neq f_c(Q_i(r))\} \text{ and decommit happens}\} < 2\epsilon_3.$$

A union bound over the μ query positions implies the result. □

Notions of strong and clear

Definition D.2.1 (strong and weak queries). Consider the event $\{A_j(d,r) = f_c(Q_j(r))\}$; notice that whether this event holds is a function of the random coin flips (c,d,r). In commit view c, a query $q \in \mathbb{F}^{n'}$ is:

- *δ-strong* if

$$\exists i\colon \Pr_{d,r}\left\{\mathcal{V} \text{ accepts and } \cap_{j=1}^{\mu}\{A_j(d,r) = f_c(Q_j(r))\} \mid Q_i(r) = q\right\} \geq \delta.$$

- δ-*weak* if

$$\forall i \colon \Pr_{d,r}\left\{\mathcal{V} \text{ accepts and } \cap_{j=1}^{\mu} \{A_j(d,r) = f_c(Q_j(r))\} \mid Q_i(r) = q\right\} < \delta.$$

This differs slightly from the Barak-Goldreich definition, which refers to strong and weak *answers*. The motivation for this definition is that if we can show most queries are strong (which we will be able to), then `extract_response` (in Figure D.1) will produce $f_c(q)$ with non-negligible probability.

Definition D.2.2 (clear and confounding queries). In commit view c, a query $q \in \mathbb{F}^{n'}$ is:

- $\delta/10$-*clear* if

$$\forall i \colon \Pr_{d,r}\{\mathcal{V} \text{ decommits and } \{A_i(d,r) \neq f_c(q)\} \mid Q_i(r) = q\} \leq \delta/10.$$

- $\delta/10$-*confounding* if

$$\exists i \colon \Pr_{d,r}\{\mathcal{V} \text{ decommits and } \{A_i(d,r) \neq f_c(q)\} \mid Q_i(r) = q\} > \delta/10.$$

This, too, is different from the analogous Barak-Goldreich definition, since they do not talk about a specific function $f_c(\cdot)$. The motivation for this definition is that if we can show most queries are clear (which we will be able to), then `extract_response` (Figure D.1) does not have to worry that a field element other than $f_c(q)$ shows up often enough to be confounding.

When a query is both strong and clear, observe that the `extract_response` subroutine is likely to deliver a clear "signal."

Auspicious commit phases happen often enough

Define a commit phase as *auspicious* if, in that phase, $\Pr_{d,r}\{\mathcal{V} \text{ accepts and } \cap_{j=1}^{\mu} \{A_j(d,r) = f_c(Q_j(r))\}\} > (1/2) \cdot \epsilon$; an *auspicious* commit phase will not necessarily be *useful*, but auspiciousness is a precondition to usefulness (see Claim D.2.9 and the analysis that follows it).

Recall the premise of the PoK property: $\Pr_{c,d,r}\{\mathcal{V} \text{ accepts}\} > \epsilon_K$. The next claim guarantees that, when this premise holds, auspicious commit phases happen with non-negligible probability.

Claim D.2.4 (Auspicious commit phases). If $\Pr_{c,d,r}\{\mathcal{V} \text{ accepts}\} > \epsilon_K$, then

$$\Pr_c\left\{\Pr_{d,r}\{\mathcal{V} \text{ accepts and } \cap_{j=1}^{\mu}\{A_j(d,r) = f_c(Q_j(r))\}\} > (1/2)\cdot\epsilon\right\} > (1/2)\cdot\epsilon,$$

where $\epsilon \overset{\text{def}}{=} \epsilon_K - \epsilon_c$, and ϵ_c was defined in Claim D.2.3.

Proof. From Claim D.2.3,

$$\Pr_{c,d,r}\{\mathcal{V} \text{ decommits and } \cup_{j=1}^{\mu}\{A_j(c,d,r) \neq f_c(Q_j(r))\}\} < \epsilon_c.$$

But accepting implies decommitting and not the other way around, so

$$\Pr_{c,d,r}\{\mathcal{V} \text{ accepts and } \cup_{j=1}^{\mu}\{A_j(c,d,r) \neq f_c(Q_j(r))\}\} < \epsilon_c.$$

Combining the given with the inequality immediately above, we get:

$$\Pr_{c,d,r}\{\mathcal{V} \text{ accepts and } \cap_{j=1}^{\mu}\{A_j(c,d,r) = f_c(Q_j(r))\}\} > \epsilon_K - \epsilon_c = \epsilon.$$

Standard counting or averaging implies the result. □

Recall that $\mathcal{V}_{\text{pcp}}$ denotes the Zaatar PCP verifier. The next two claims state that with probability that cannot be neglected (a) $\mathcal{V}_{\text{pcp}}$ accepts (which implies that $f_c(\cdot)$ is of the right form), and (b) all queries issued by $\mathcal{V}_{\text{pcp}}$ are, in the context of the argument protocol, δ-strong.

After auspicious commit phases, $f_c(\cdot)$ is a valid PCP oracle

Claim D.2.5 ($\mathcal{V}_{\text{pcp}}$ accepts often). Assuming we are in an auspicious commit phase,

$$\Pr_r\left\{\mathcal{V}_{\text{pcp}} \text{ accepts } (f_c(Q_1(r)),\ldots,f_c(Q_\mu(r)))\right\} > (1/2)\cdot\epsilon.$$

Proof. In an auspicious commit phase

$$(1/2)\cdot\epsilon < \Pr_{d,r}\left\{\mathcal{V} \text{ accepts and } \cap_{i=1}^{\mu}\{A_i(d,r) = f_c(Q_i(r))\}\right\}.$$

But if $\mathcal{V}$ accepts on a particular set of coin flips, then $\mathcal{V}_{\text{pcp}}$ must accept the same answers, since the latter is a precondition for the former. So we can bound the expression above:

$$
\begin{aligned}
&\le \Pr_{d,r}\left\{\mathcal{V}_{\text{pcp}} \text{ accepts } (A_1(d,r),\ldots,A_\mu(d,r)) \text{ and } \cap_{i=1}^{\mu}\{A_i(d,r) = f_c(Q_i(r))\}\right\}\\
&\le \Pr_{d,r}\left\{\mathcal{V}_{\text{pcp}} \text{ accepts } (f_c(Q_1(r)),\ldots,f_c(Q_\mu(r)))\right\}\\
&= \Pr_{r}\left\{\mathcal{V}_{\text{pcp}} \text{ accepts } (f_c(Q_1(r)),\ldots,f_c(Q_\mu(r)))\right\}.
\end{aligned}
$$

The second inequality holds because the event in its LHS is a restricted case of the event in its RHS. The equality holds because its LHS is independent of the d coins. □

Take $\epsilon/2 = \epsilon_{\text{pcp}}$, where ϵ_{pcp} is Zaatar's PCP soundness error. The claim above, together with the properties of Zaatar (soundness in Lemma C.3.2, and one other: see below), implies the following:

Corollary D.2.6 ($f_c(\cdot)$ **is often a valid PCP oracle).** In auspicious commit phases, $f_c(\cdot)$ is a well-formed Zaatar PCP oracle: it is 0.0294-close to a linear function that encodes a witness z that satisfies $\mathcal{C}(X{=}x, Y{=}y)$.

This corollary relies on a property of Zaatar's PCP that is stronger than soundness: "well-formedness". As stated, this property is (a shade) stronger than *PCP proof-of-knowledge (PCP PoK)*. PCP PoK [19] says that if $\mathcal{V}_{\text{pcp}}$ accepts with greater than the soundness error, then not only is $\mathcal{C}$ satisfiable (which is what the soundness property gives) but also there is an efficient algorithm that can extract a satisfying witness, given access to the PCP oracle. As Barak and Goldreich [19] observe, many PCPs have the PCP PoK property (Zaatar does too), but there are few (if any) proofs in the literature. The reason that our well-formedness property is slightly stronger than a PCP PoK property is that it actually specifies the form of the PCP (and any PCP meeting this form can, through self-correction, yield a witness).

After auspicious commit phases, most queries are strong

Claim D.2.7. Assuming we are in an auspicious commit phase,

$$\Pr_r\{\mathcal{V}_{\text{pcp}} \text{ makes only } \delta\text{-strong queries}\} > \epsilon/4,$$

for $\delta = (1/4)\epsilon/\mu$.

Proof. Fix a query position $i \in [\mu]$:

$$\begin{aligned}
&\Pr_{d,r}\left\{\mathcal{V} \text{ accepts and } \cap_{j=1}^{\mu} \{A_j(d,r) = f_c(Q_j(r))\} \text{ and } Q_i(r) \text{ is } \delta\text{-weak}\right\} \\
&= \sum_{q: q \text{ is } \delta\text{-weak}} \Pr_{d,r}\left\{\mathcal{V} \text{ accepts and } \cap_{j=1}^{\mu} \{A_j(d,r) = f_c(Q_j(r))\} \mid Q_i(r) = q\right\} \cdot \Pr_r\{Q_i(r) = q\} \\
&= \sum_{q: q \text{ is } \delta\text{-weak}} \delta \cdot \Pr_r\{Q_i(r) = q\} < \delta
\end{aligned}$$

By the union bound over positions $1, \dots, \mu$,

$$\Pr_{d,r}\left\{\mathcal{V} \text{ accepts and } \cap_{j=1}^{\mu} \{A_j(d,r) = f_c(Q_j(r))\} \text{ and } \textit{any } Q_j(r) \text{ is } \delta\text{-weak}\right\} < \mu \cdot \delta.$$

Combining this with the definition of auspicious, we get

$$\begin{aligned}
(1/2)\epsilon - \mu \cdot \delta &< \Pr_{d,r}\left\{\mathcal{V} \text{ accepts and } \cap_{j=1}^{\mu} \{A_j(d,r) = f_c(Q_j(r))\} \text{ and all of } Q_1(r), \dots, Q_\mu(r) \text{ are } \delta\text{-strong}\right\} \\
&\le \Pr_{d,r}\{\text{all of } Q_1(r), \dots, Q_\mu(r) \text{ are } \delta\text{-strong}\} \\
&= \Pr_r\{\text{all of } Q_1(r), \dots, Q_\mu(r) \text{ are } \delta\text{-strong}\}
\end{aligned}$$

Substituting $\delta = (1/4)\epsilon/\mu$ in the lower bound gives the result. □

Corollary D.2.8 (**Most queries are strong**). Recalling that $\mathcal{V}_{\text{pcp}}$ makes μ PCP queries, if ρ' of these queries are *independently and uniformly random*, then (in auspicious commit phases) the fraction of total queries that is δ-strong is greater than $(\epsilon/4)^{1/\rho'}$.

At this point, we are ready to argue that the overwhelming majority of queries are δ-strong. Looking at Zaatar's PCP, it has $\rho' = 320$ queries that hit π_z randomly. Furthermore, we took $\epsilon = 2\epsilon_{\text{pcp}} \approx 10^{-6}$ (since ϵ_{pcp}, the soundness error of Zaatar's PCP, is $\approx 5 \cdot 10^{-7}$ from Appendix C). Thus, by Corollary D.2.8, after auspicious commit phases, the fraction of total queries that is δ-strong is greater than $(\epsilon/4)^{1/320} > 0.95$.

In most commit phases, most queries are not confounding

Recall that the notion of being δ-confounding is a function of the commit phase. We will now show that in the vast majority of commit phases, the vast majority of q are not ϵ_3-confounding (ϵ_3 is from Claim D.2.2).

Claim D.2.9 (Most queries are clear). Letting $\Pr_q$ denote a uniformly random choice of q,

$$\Pr_c\left\{\Pr_q\{q \text{ is } \epsilon_3\text{-confounding}\} < 1/20\right\} > 1 - 20\mu\epsilon_3$$

Proof. Let $G_{q,i}(c)$ denote the event in commit phase c that

$$\Pr_{d,r}\{\{A_i(d,r) \neq f_c(q)\} \text{ and decommit happens} \mid Q_i(r) = q\} > \epsilon_3.$$

Once q and i have been fixed, this expression is either true or not in commit phase c, and that is what the events G will capture. Claim D.2.2 implies:

$$\forall q, i\colon \Pr_c\left\{G_{q,i}(c)\right\} < \epsilon_3.$$

Applying a union bound over query positions, we get

$$\forall q\colon \Pr_c\left\{\cup_{i=1}^{\mu} G_{q,i}(c)\right\} < \mu\epsilon_3.$$

By definition of ϵ_3-confounding

$$\forall q\colon \Pr_c\{q \text{ is } \epsilon_3\text{-confounding}\} < \mu\epsilon_3.$$

Applying a standard averaging argument followed by a Markov bound

$$\Pr_c\left\{\Pr_q\{q \text{ is } \epsilon_3\text{-confounding}\} > 1/20\right\} < 20\mu\epsilon_3$$

The complementary probabilities and events to the ones immediately above imply the claim. □

Completing the analysis

We require

$$\epsilon_3 < \min\left\{\frac{\epsilon}{40\mu}, \frac{\epsilon}{80\mu}\right\}$$

because:

1. The first component in the min ensures that $\epsilon_3 < \delta/10$ (recall that $\delta = \epsilon/(4\mu)$). This

bound gives us a gap (between δ and $\delta/10$) that helps us pump the prover in the "inner extraction loop".

2. The second component will ensure that the fraction of useful commit phases is $> \epsilon/2 - 20\mu\epsilon_3 > \epsilon/4$, which we want, to ensure that ϵ'_K (in the definition of PoK) is non-negligible.

We must verify that the upper bound on ϵ_3 holds. Recall that $\epsilon_3 < 6 \cdot \sqrt[3]{1/|\mathbb{F}|}$ (from Claim D.2.2) and $\epsilon = 2\epsilon_{\text{pcp}} \approx 10^{-6}$ (see Corollaries D.2.6 and D.2.8); also, μ is almost exactly 1000. Fortunately, at the field size that Pantry works with (128 bits), $6 \cdot \sqrt[3]{1/|\mathbb{F}|} < \epsilon/(80\mu)$, so the bound holds.

Analyzing the steps of the extractor. By Claim D.2.4, if the PoK premise ($\Pr\{\mathcal{V} \text{ accepts}\} > \epsilon_K$) holds, the choice of commit phase in the extractor is *useful* with probability $> \epsilon/2 - (20\mu\epsilon_3) > \epsilon/4$; this is a commit phase that is both auspicious *and* bounds the fraction of ϵ_3-confounding queries, in the sense of Claim D.2.9. From now on, we assume such a useful commit phase. By Corollary D.2.6, $f_c(\cdot)$ is δ'-close to a linear function that encodes a satisfying witness $\vec{z}$, for some δ' that is $< .03$. Note that this δ' is different from the δ in some of the claims stated earlier.

Now fix t; consider iteration k. Look at query q_r in this iteration. By definition of δ'-close, we have $\Pr_{q_r}\{q_r \text{ hits } f_c(\cdot) \text{ where it is not linear}\} < .03$, where the probability is taken over the coins that generate q_r. Also, by Corollary D.2.8, $\Pr_{q_r}\{q_r \text{ is } \delta\text{-weak}\} < 1 - (\epsilon/4)^{1/\rho'} < .05$. And we have $1/20 > \Pr_{q_r}\{q_r \text{ is } \epsilon_3\text{-confounding}\} \geq \Pr_{q_r}\{q_r \text{ is } \delta/10\text{-confounding}\}$. The first inequality comes from Claim D.2.9; the second, from the bound on ϵ_3 and the definition of confounding. Therefore, the probability (over the random choice of q_r and q_s) that q_r and q_s both have the desirable properties (namely: hit $f_c(\cdot)$ where linear; strong; clear) is at least $1 - 2(.03 + .05 + .05) = 0.74$. Call an iteration k in which this event occurs "good".

Next, consider the "inner loop" (the function `extract_response`), assuming the iteration is good. We'll speak of q_r, but the same analysis applies to q_s:

- Because q_r is δ-strong, there is a query position i^* for which the i^* response is $f_c(q_r)$, with probability at least δ over the reverse sampling coins. Thus, the expected number of times decommit succeeds when $i = i^*$ is $\geq \delta \cdot T_2$. Now we apply a Chernoff bound, using this form [100, Thm. 4.2]: $\Pr\{X \leq (1-a)E[X]\} < e^{-a^2 \cdot E[X]/2}$. We take $E[X] \geq \delta \cdot T_2$ and $a \geq 2/3$. This implies that for $T_2 > 21/\delta$, the probability in iteration k that position i^* will *not* label $f_c(q_r)$ a candidate is $< (1/100)$. The probability is over the coins used for reverse sampling in the j loop of iteration i^*.

- Now fix any position $i \in [\mu]$. Call all field elements besides $f_c(q_r)$ *scrap*. We wish to upper bound the probability of the event (over the reverse sampling coins used in the j loop) that all scrap, together, is decommitted more than $(\delta/3) \cdot T_2$ times: this probability is an upper bound on the probability that any field value is actually labeled a candidate (since if the scrap together does not clear the threshold, then no element by itself does). q_r is $(\delta/10)$-clear, so the expected number of times that all scrap, together, is decommitted is $< (\delta/10) \cdot T_2$. We use this form of the Chernoff bound [100, Thm 4.3]: $\Pr\{X \geq (1+a)E[X]\} < e^{-a^2 E[X]/4}$. For $T_2 \geq 4.7/\delta > (\ln(100\mu))/(2.5 \cdot \delta)$, an upper bound on the event in question is 10^{-5}.
- Now we can union bound over all μ query positions: the probability (over the reverse sampling coins in `extract_response`) that *any* position has a scrap candidate is $< \mu \cdot 10^{-5}$. Combining this with the event that $f_c(q_r)$ is not labeled a candidate, we get that $f_c(q_r)$ is *not* returned from `extract_response` with probability upper-bounded by $2/100$. The same goes for $f_c(q_s)$.

Now, if iteration k is good, and furthermore produces $f_c(q_r)$ and $f_c(q_s)$, then $\sigma_2 - \sigma_1 = f_c(q_r+e_t) - f_c(q_r) = \vec{z} \cdot (q_r+e_t) - \vec{z} \cdot (q_r) = \vec{z} \cdot e_t = z_t$. Thus, in iteration k, the probability (over all of the randomness that the algorithm used in the iteration: choice of q_r, q_s and reverse sampling in `extract_response`) of outputting z_t is greater than $> 1 - 2(.03 + .05 + .05 + .02) = 7/10$. Now we apply another Chernoff bound, this time over the iterations k. For $T_1 > 3500$, the probability that there are fewer than $T_1/2$ instances of z_t is $< e^{-100}$.

Applying a union bound to all positions in the witness, the probability of not extracting the witness (if we're in a useful commit phase) is $< n' \cdot e^{-100}$. Also, the probability of a useful commit phase is, as stated above, greater than $\epsilon/4$; furthermore, $\epsilon = 2 \cdot \epsilon_{\text{pcp}}$ (see page 125). Therefore, the probability (over all of the extractor's many coin flips) of producing a witness is at least $\left(\epsilon_{\text{pcp}}/2\right) \cdot \left(1 - n' \cdot e^{-100}\right)$, which was what the lemma claimed. $\square$

Our analysis produced lower bounds for T_1 and T_2: $T_1 > 3500$ and $T_2 > 80$ billion. The extractor thus has an appalling concrete cost: producing *one* component of a witness requires running the verifier-prover decommit phase (including generating queries) $5 \cdot 10^{17}$ times, and that's only if the event of a useful commit phase happened, which has probability $\geq \epsilon/4 \approx 2.5 \cdot 10^{-7}$! Thus, the expected time to generate a witness is 10^{24} times the effort required to run the decommit phase. Nevertheless, the extractor runs in "polynomial time", as required. (The quotation marks are because our analysis is not asymptotic; in an asymptotic analysis, n' would grow, the error terms would depend on n', etc.)

Furthermore, the expected time to obtain a witness, though massive, is still far less than the expected time to generate a hash collision, as Pantry uses a hash function with at least 180 bits of security (§5.6). This gap is sufficient to generate the required contradictions in the proofs in Appendix D.1.

D.3 An HMAC-based commitment

In Section 5.5, we explain that in order to enable applications where the prover's state is private, we need a commitment to bind the prover to the state while hiding it from the verifier. Ordinarily, we would use a standard commitment scheme, such as Pedersen's [106], which would guarantee binding with respect to a computationally-bound prover along with information-theoretic hiding with respect to the verifier. Because Pedersen's protocol cannot be represented efficiently as constraints, we instead use a simple scheme based on HMAC-SHA256, which also provides computational binding, but hiding that is only computational. We present our scheme and prove its security here.

1. $\mathsf{Setup}(1^n) \to \mathsf{CK}$
   Setup takes a unary string of length n, a security parameter, and returns CK, a public commitment key that is used to distinguish commitments based on this construction from other MACs generated using HMAC-SHA256.

2. $\mathsf{Commit}(m, r) \to c$, where $c = \text{HMAC-SHA256}_r(CK \| m)$ and $r \leftarrow_R \{0,1\}^{512}$
   Commit takes the message m and a randomly-chosen value r as input and returns a commitment c. r can later be revealed to decommit.

3. $\mathsf{Decommit}(m', r', c) \to \begin{cases} \text{true} & \text{if } c = \text{HMAC-SHA256}_{r'}(CK \| m') \\ \text{false} & \text{otherwise} \end{cases}$
   Decommit takes the purported message m' and decommitment key r' as input and recomputes the HMAC-SHA256 to check whether it equals the received commitment c. If so, the commitment is considered validly decommitted, and it is considered invalidly decommitted otherwise.

Lemma D.3.1. The construction above, which we denote $\Pi = (\mathsf{Setup}, \mathsf{Commit}, \mathsf{Decommit})$, is a correct, computationally hiding, computationally binding commitment if (1) HMAC-SHA256 is a PRF and (2) SHA-256 is a CRHF.

Proof. A commitment scheme is correct if $\mathsf{Decommit}(m, r, \mathsf{Commit}(m, r)) = \text{true}$ for all m and r. One can see that Π is correct because $\mathsf{Decommit}(m, r, c) = \text{true}$ when $c = \text{HMAC-SHA256}_r(CK \| m)$, which is exactly what $\mathsf{Commit}(m, r)$ computes. The proofs of hiding and binding follow from Claims D.3.2 and D.3.3 respectively. $\square$

Claim D.3.2. If HMAC-SHA256 is a PRF,[2] then Π is a computationally hiding commitment.

Proof. Computational hiding is defined with respect to the following game played by a probabilistic polynomial time (PPT) adversary $\mathcal{A}$:

1. The committer runs $\mathsf{Setup}(1^n) \to \mathsf{CK}$
2. $\mathcal{A}$ picks two messages m_0 and m_1.
3. The committer chooses $b \leftarrow_R \{0, 1\}$ and $r \leftarrow_R \{0, 1\}^k$, computes $c = \mathsf{Commit}(m_b, r)$, and sends c to $\mathcal{A}$.[3]
4. $\mathcal{A}$ outputs b' and wins if $b' = b$.

Denote the probability (over the random choices of $\mathcal{A}$ and the committer) that $\mathcal{A}$ wins this game against commitment scheme Π by $\Pr\left\{\mathsf{BreakHiding}_{\mathcal{A},\Pi}(n) = 1\right\}$. We say that Π is computationally hiding if $\epsilon(n) \overset{\text{def}}{=} \Pr\left\{\mathsf{BreakHiding}_{\mathcal{A},\Pi}(n) = 1\right\} - \frac{1}{2}$ is negligible.

To see why $\epsilon(n)$ must be negligible, we consider a variant of our scheme $\widetilde{\Pi} = (\widetilde{\mathsf{Setup}}, \widetilde{\mathsf{Commit}}, \widetilde{\mathsf{Decommit}})$ where $\text{HMAC-SHA256}_r(CK \| m)$ is replaced by $f(CK \| m)$ and f is a truly random function. In that case, $\Pr\left\{\mathsf{BreakHiding}_{\mathcal{A},\widetilde{\Pi}}(n) = 1\right\} = \frac{1}{2}$ and therefore,

$$\epsilon(n) = \Pr\left\{\mathsf{BreakHiding}_{\mathcal{A},\Pi}(n) = 1\right\} - \Pr\left\{\mathsf{BreakHiding}_{\mathcal{A},\widetilde{\Pi}}(n) = 1\right\}.$$

Now, suppose that we construct a PPT algorithm $\mathcal{D}$ that attempts to distinguish between HMAC-SHA256 and f defined as follows.

1. $\mathcal{D}$ is given 1^n along with an oracle $\mathcal{O}$ that is either HMAC-SHA256_r, where $r \leftarrow_R \{0, 1\}^{512}$, or f.
2. $\mathcal{D}$ runs $\mathsf{Setup}(1^n) \to \mathsf{CK}$ and $\mathcal{A}(1^n)$. When $\mathcal{A}$ provides two messages m_0 and m_1, $\mathcal{D}$ picks $b \leftarrow_R \{0, 1\}$, and returns $c = \mathcal{O}(CK \| m_b)$ to $\mathcal{A}$.
3. When $\mathcal{A}$ outputs b', $\mathcal{D}$ returns 1 if $b' = b$ and 0 otherwise.

If $\mathcal{O}$ is HMAC-SHA256_r, then $\mathcal{A}$'s view when run as a subroutine of $\mathcal{D}$ is identical to $\mathcal{A}$'s view

[2] Bellare shows that HMAC is a PRF if the underlying compression function is a PRF [21]. We assume that SHA-256 is a PRF when its initialization vector is chosen randomly and kept secret.

[3] The value of k depends on the commitment scheme.

when playing the computational hiding game. Thus,

$$\Pr\left\{\mathcal{D}^{\text{HMAC-SHA256}_r}(1^n) = 1\right\} = \Pr\left\{\mathsf{BreakHiding}_{\mathcal{A},\Pi}(n) = 1\right\}$$

where $\Pr\left\{\mathcal{D}^{\text{HMAC-SHA256}_r}(1^n) = 1\right\}$ is taken over r and $\mathcal{D}$'s and $\mathcal{A}$'s random choices, and similarly,

$$\Pr\left\{\mathcal{D}^f(1^n) = 1\right\} = \Pr\left\{\mathsf{BreakHiding}_{\mathcal{A},\widetilde{\Pi}}(n) = 1\right\}$$

and so

$$\epsilon(n) = \Pr\left\{\mathcal{D}^{\text{HMAC-SHA256}_r}(1^n) = 1\right\} - \Pr\left\{\mathcal{D}^f(1^n) = 1\right\}.$$

If $\epsilon(n)$ were not negligible, then $\mathcal{D}$ would be able to distinguish between HMAC-SHA256 and f, violating our assumption that HMAC-SHA256 is a PRF. □

Claim D.3.3. If SHA-256 is a CRHF, then Π is a computationally binding commitment.

Proof. Computational binding is defined with respect to the following game played by a PPT adversary $\mathcal{A}$.

1. $\mathcal{A}$ runs $\mathsf{Setup}(1^n) \to \mathsf{CK}$
2. $\mathcal{A}$ picks two messages m_0 and m_1 such that $m_0 \neq m_1$ and two decommitment keys r_0 and r_1. $\mathcal{A}$ then computes $\mathsf{Commit}(m_0, r_0) \to c_0$ and $\mathsf{Commit}(m_1, r_1) \to c_1$
3. $\mathcal{A}$ outputs CK, m_0, m_1, r_0, r_1, c_0, and c_1 and wins if $c_0 = c_1$.

Let the probability (over $\mathcal{A}$'s random choices) that $\mathcal{A}$ wins this game against our scheme Π be $\Pr\left\{\mathsf{BreakBinding}_{\mathcal{A},\Pi}(n) = 1\right\}$. If this probability is negligible, then we can say that Π is computationally binding.

To see why it must be negligible, we construct a PPT algorithm $\mathcal{B}$ that uses $\mathcal{A}$ in an attempt to find a collision in SHA-256. $\mathcal{B}$ is defined as follows.

1. $\mathcal{B}$ is given 1^n and runs $\mathcal{A}(1^n)$.

2. When $\mathcal{A}$ outputs CK, m_0, m_1, r_0, r_1, c_0, and c_1, $\mathcal{B}$ constructs four messages:

$$\begin{aligned} a_0 &= (r_0 \oplus \text{ipad}) \,||\, CK \,||\, m_0 \\ b_0 &= (r_0 \oplus \text{opad}) \,||\, \text{SHA-256}(a_0) \\ a_1 &= (r_1 \oplus \text{ipad}) \,||\, CK \,||\, m_1 \\ b_1 &= (r_1 \oplus \text{opad}) \,||\, \text{SHA-256}(a_1), \end{aligned}$$

where opad is a string of 64 `0x5c` bytes and ipad is a string of 64 `0x36` bytes. If $b_0 \neq b_1$, $\mathcal{B}$ outputs $m = b_0$ and $m' = b_1$. Otherwise, $\mathcal{B}$ outputs $m = a_0$ and $m' = a_1$. $\mathcal{B}$ wins if SHA-256(m) = SHA-256(m') and $m \neq m'$.

$\mathcal{A}$'s view when run as a subroutine of $\mathcal{B}$ is identical to $\mathcal{A}$'s view when playing the computational binding game. Moreover, because HMAC-SHA256$_r(x)$ = SHA-256$((r \oplus \text{opad}) \,||\, \text{SHA-256}((r \oplus \text{ipad}) \,||\, x))$, $\mathcal{B}$ wins exactly when $\mathcal{A}$ would have won the computational binding game (i.e., when $\mathsf{Commit}(m_0, r_0) = \mathsf{Commit}(m_1, r_1)$ where $m_0 \neq m_1$). Thus,

$$\Pr\left\{\mathsf{Collision}_{\mathcal{B}}^{\text{SHA-256}}(1^n) = 1\right\} = \Pr\left\{\mathsf{BreakBinding}_{\mathcal{A},\Pi}(n) = 1\right\}$$

where $\Pr\left\{\mathsf{Collision}_{\mathcal{B}}^{\text{SHA-256}}(1^n) = 1\right\}$ is taken over $\mathcal{B}$'s (really $\mathcal{A}$'s) random choices. As a result, if the probability that $\mathcal{A}$ wins the computational binding game were non-negligible, then the probability that $\mathcal{B}$ finds a collision in SHA-256 would be as well, violating the assumption that SHA-256 is a CRHF. $\square$

D.4 Applications, parameters, and modeling

This appendix describes the configuration of our experimental evaluation (§5.7) in more detail.

D.4.1 Details of sample applications

Dot product. Computes the dot product between two integer arrays, each of length m. Each mapper gets a chunk of the input vectors and computes a partial dot product, outputting an integer. Each reducer gets as input a list of numbers, and sums it. Another reducer phase sums the sums.

Nucleotide substring search. Searches m nucleotides for length-d substring. Each mapper gets as input a chunk of DNA and the same length-d substring; if a mapper finds a match, it outputs the position of the match. Each reducer takes as input a list of locations and concatenates them.

Nearest neighbor search. The search takes as input a length-d target vector and a list of m vectors, each of length d. Each mapper gets as input a subset of the search list of m vectors and the target vector. A mapper computes the Euclidean distance between the target vector and each vector in the subset, outputting a list of distances. Each reducer takes as input a list of Euclidean distances and computes the minimum. Another reducer phases computes the minimum among these minimums.[4]

Covariance matrix. Computes the covariance matrix for m samples, each of dimension d. Each mapper gets as input a subset of the samples and computes a $d \times d$ covariance matrix, for its samples. Each reducer aggregates a set of these matrices, producing another $d \times d$ matrix. Then, a final reduce phase produces the final covariance matrix.

SELECT, INSERT, and UPDATE. These queries do as their names imply. Our database has three indices, and parameters are given in Figures 5.10 and 5.12.

Face matching. The prover stores a list of 928-bit fingerprints of faces and a threshold for each fingerprint. The verifier supplies a fingerprint of 928 bits, and the prover indicates that there is a match if and only if the Hamming distance between the input fingerprint and one of the faces in the list is below the threshold for that fingerprint. This algorithm is based on the approach of Osadchy et al. [102].

Tolling. The verifier is a toll collector, and the prover is a driver. The prover uses toll roads during a month and maintains a private database of its own toll road usage. Whenever the prover passes a tolling location, it adds a tuple to its database of the form (time, tolling_location_id, toll_amount). The verifier can randomly and unpredictably "spot check" the prover whenever it uses a tolling location by storing a tuple of the same form in a separate database; the prover cannot tell whether it has been spot checked. At the end of the month, the prover sends a commitment to its database to the verifier. The computation to be verified takes as input the prover's commitment to its database and the spot checks that the verifier collected. The computation outputs REJECT if one of the spot checks does not have a "close matching tuple" in the database (two tuples are a close match when the tolling_location_id and toll_amount match and when the difference in the times is less than a system parameter). Otherwise, the computation returns the total cost of tolls incurred by the prover in that month.

Regression analysis. The verifier is a data analyst who, for example, would like to learn a model for the effectiveness of a drug, based on a patient's background and symptoms; the prover is a clinic. The prover holds a list of patient records and sends a commitment to this data to the verifier. The computation takes as input the prover's commitment, a set of patient features to model, and a parameter $k > 0$. The computation returns a linear function obtained

[4]This computation would be better named "nearest neighbor distance search", as it returns the distance rather than the closest vector; with minor changes (and few performance effects), the computation could return the distance and the nearest vector.

by applying ridge regression [77] with regularization parameter k to all patient records in the prover's database; in the regression, the independent variables are the features, and the dependent variable is patient recovery time. That is, the linear function produced by the computation predicts a patient's recovery time, as a function of the patient's features, but does not reveal the details of any particular patient record.

D.4.2 Parameters

For the experiments that use Zaatar, we configure the field $\mathbb{F}$ (recall that $\mathbb{F} = \mathbb{F}_p$) to have a prime modulus of 128 bits. Zaatar uses ElGamal encryption (as part of step (3) in Section 5.1.2), and our experiments presume 1024-bit keys [112]. For the experiments that use Pinocchio, we configure the field to have a prime modulus of 254 bits. For Pinocchio's pairing-based cryptography, we use a BN curve that provides 128 bits of security [20].

D.4.3 Modeling

Below, we quantify the constants in the cost model in Figure 5.3. We run a set of microbenchmarks to measure the costs of the basic operations (e.g., encryption, decryption, multiplication, etc.) on our hardware platform (§5.7), and we use a detailed cost model from prior work [112] to estimate the constants. The values are as follows:

	Zaatar	Pinocchio
c_1	9 ns	9 ns
c_2	77 μs	230 μs
c_3	205 μs	6 ms
c_4	4.8 μs	0.35 μs
c_5	170 μs	243 μs
c_6	1.5 μs	$0.6 \log \lvert\mathcal{C}\rvert$ μs

Accuracy and assumptions. For applications that we use in Pantry, our end-to-end empirical results are generally within 20% of their predictions, but for the prover, the empirics are smaller than predictions of the cost model by up to a factor of 2. The primary reason for this deviation, as mentioned earlier, is that Pantry's applications include a large number of storage constraints (§5.7), and the values taken by the variables in those constraints are much smaller than the prime modulus, p, which reduces the value of c_5 for such applications.

Extensions for a more faithful model. One way to improve the accuracy of our simple cost model is to make c_5 depend on the relative number of bitwise operations and on the average number of bits in the values taken by variables in the constraints of a computation.

Bibliography

[1] Amazon Elastic Compute Cloud (EC2). `http://aws.amazon.com/ec2/`.

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

[3] CUDA (http://developer.nvidia.com/what-cuda).

[4] The GNU MP bignum library. `http://gmplib.org/`.

[5] Google Cloud Platform. `https://cloud.google.com/`.

[6] High-speed software implementation of the optimal Ate pairing over Barreto-Naehrig curves. `https://github.com/herumi/ate-pairing`.

[7] leveldb – a fast and lightweight key/value database library by Google. `https://code.google.com/p/leveldb/`.

[8] Microsoft's Cloud Platform. `https://azure.microsoft.com/`.

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

[10] Pantry's source code. `http://www.github.com/srinathtv/pantry/`.

[11] PAPI: Performance Application Programming Interface.

[12] M. Ajtai. Generating hard instances of lattice problems. In *ACM Symposium on the Theory of Computing (STOC)*, pages 99–108, May 1996.

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

[14] 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.

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

[16] 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.

[17] L. Babai, L. Fortnow, L. A. Levin, and M. Szegedy. Checking computations in polylogarithmic time. In *ACM Symposium on the Theory of Computing (STOC)*, 1991.

[18] M. Backes, D. Fiore, and R. M. Reischuk. Verifiable delegation of computation on outsourced data. In *ACM Conference on Computer and Communications Security (CCS)*, Nov. 2013.

[19] B. Barak and O. Goldreich. Universal arguments and their applications. *SIAM Journal on Computing*, 38(5):1661–1694, 2008.

[20] P. S. L. M. Barreto and M. Naehrig. Pairing-friendly elliptic curves of prime order. In *Selected Areas in Cryptography (SAC)*, 2006.

[21] M. Bellare. New proofs for NMAC and HMAC: Security without collision-resistance. In *CRYPTO*, 2006.

[22] M. Bellare, D. Coppersmith, J. Håstad, M. Kiwi, and M. Sudan. Linearity testing in characteristic two. *IEEE Transactions on Information Theory*, 42(6):1781–1795, Nov. 1996.

[23] M. Bellare, S. Goldwasser, C. Lund, and A. Russell. Efficient probabilistically checkable proofs and applications to approximations. In *ACM Symposium on the Theory of Computing (STOC)*, 1993.

[24] M. Ben-Or, S. Goldwasser, J. Kilian, and A. Wigderson. Multi-prover interactive proofs: how to remove intractability assumptions. In *ACM Symposium on the Theory of Computing (STOC)*, 1988.

[25] 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.

[26] E. Ben-Sasson, A. Chiesa, D. Genkin, and E. Tromer. On the concrete-efficiency threshold of probabilistically-checkable proofs. In *ACM Symposium on the Theory of Computing (STOC)*, June 2013.

[27] E. Ben-Sasson, A. Chiesa, D. Genkin, E. Tromer, and M. Virza. SNARKs for C: Verifying program executions succinctly and in zero knowledge. In *International Cryptology Conference (CRYPTO)*, pages 90–108, Aug. 2013.

[28] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza. Scalable zero knowledge via cycles of elliptic curves. In *International Cryptology Conference (CRYPTO)*, pages 276–294, Aug. 2014.

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

[30] E. Ben-Sasson, O. Goldreich, P. Harsha, M. Sudan, and S. Vadhan. Short PCPs verifiable in polylogarithmic time. In *IEEE Conference on Computational Complexity (CCC)*, 2005.

[31] E. Ben-Sasson, O. Goldreich, P. Harsha, M. Sudan, and S. Vadhan. Robust PCPs of proximity, shorter PCPs and applications to coding. *SIAM Journal on Computing*, 36(4):889–974, Dec. 2006.

[32] E. Ben-Sasson and M. Sudan. Short PCPs with polylog query complexity. *SIAM Journal on Computing*, 38(2):551–607, May 2008.

[33] S. Benabbas, R. Gennaro, and Y. Vahlis. Verifiable delegation of computation over large datasets. In *International Cryptology Conference (CRYPTO)*, pages 111–131, Aug. 2011.

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

[35] J.-P. Berrut and L. N. Trefethen. Barycentric Lagrange interpolation. *SIAM Review*, 46(3):501–517, 2004.

[36] 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/`.

[37] 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.

[38] N. Bitansky, R. Canetti, A. Chiesa, and E. Tromer. Recursive composition and bootstrapping for SNARKs and proof-carrying data. In *ACM Symposium on the Theory of Computing (STOC)*, pages 111–120, June 2013.

[39] N. Bitansky, A. Chiesa, Y. Ishai, R. Ostrovsky, and O. Paneth. Succinct non-interactive arguments via linear interactive proofs. In *Theory of Cryptography Conference (TCC)*, pages 315–333, Mar. 2013.

[40] M. Blum, W. Evans, P. Gemmell, S. Kannan, and M. Naor. Checking the correctness of memories. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 90–99, Oct. 1991.

[41] M. Blum, M. Luby, and R. Rubinfeld. Self-testing/correcting with applications to numerical problems. *Journal of Computer and System Sciences*, 47(3):549–595, Dec. 1993.

[42] D. Boneh and D. M. Freeman. Homomorphic signatures for polynomial functions. In *International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, pages 149–168, May 2011.

[43] G. Brassard, D. Chaum, and C. Crépeau. Minimum disclosure proofs of knowledge. *Journal of Computer and System Sciences*, 37(2):156–189, Oct. 1988.

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

[45] B. Braun, A. J. Feldman, Z. Ren, S. Setty, A. J. Blumberg, and M. Walfish. Verifying computations with state. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 341–357, Nov. 2013.

[46] M. Castro and B. Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS)*, 20(4):398–461, Nov. 2002.

[47] A. Chiesa and E. Tromer. Proof-carrying data and hearsay arguments from signature cards. In *ICS*, 2010.

[48] K.-M. Chung, Y. Kalai, and S. Vadhan. Improved delegation of computation using fully homomorphic encryption. In *International Cryptology Conference (CRYPTO)*, 2010.

[49] K.-M. Chung, Y. T. Kalai, F.-H. Liu, and R. Raz. Memory delegation. In *International Cryptology Conference (CRYPTO)*, pages 151–168, Aug. 2011.

[50] A. Clement, M. Kapritsos, S. Lee, Y. Wang, L. Alvisi, M. Dahlin, and T. Riche. UpRight cluster services. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2009.

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

[52] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press, 1990.

[53] 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.

[54] J.-S. Coron, Y. Dodis, C. Malinaud, and P. Puniya. Merkle-damgård revisited: how to construct a hash function. In *International Cryptology Conference (CRYPTO)*, pages 430–448, Aug. 2005.

[55] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 107–113, Dec. 2004.

[56] I. Dinur. The PCP theorem by gap amplification. *Journal of the ACM*, 54(3), June 2007.

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

[58] F. Ergün, R. Kumar, and R. Rubinfeld. Approximate checking of polynomials and functional equations. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, 1996.

[59] 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.

[60] K. Fu, M. F. Kaashoek, and D. Mazières. Fast and secure distributed read-only file system. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–24, Oct. 2000.

[61] M. Garofalakis, J. M. Hellerstein, and P. Maniatis. Proof sketches: Verifiable in-network aggregation. In *IEEE Conference on Data Engineering (ICDE)*, 2007.

[62] R. Gennaro, C. Gentry, and B. Parno. Non-interactive verifiable computing: Outsourcing computation to untrusted workers. In *International Cryptology Conference (CRYPTO)*, pages 465–482, Aug. 2010.

[63] R. Gennaro, C. Gentry, B. Parno, and M. Raykova. Quadratic span programs and succinct NIZKs without PCPs. In *International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, pages 626–645, May 2013.

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

[65] C. Gentry and S. Halevi. Implementing Gentry's fully-homomorphic encryption scheme. In *International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, 2011.

[66] C. Gentry, S. Halevi, and N. Smart. Homomorphic evaluation of the AES circuit. In *International Cryptology Conference (CRYPTO)*, 2012.

[67] 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.

[68] D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1):5–48, Mar. 1991.

[69] O. Goldreich. *Foundations of Cryptography: II Basic Applications*. Cambridge University Press, 2004.

[70] O. Goldreich, S. Goldwasser, and S. Halevi. Collision-free hashing from lattice problems. *Electronic Colloquium on Computational Complexity (ECCC)*, TR96-042:236–241, 1996.

[71] 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.

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

[73] P. Golle and I. Mironov. Uncheatable distributed computations. In *RSA Conference*, pages 425–440, Apr. 2001.

[74] J. Groth. Linear algebra with sub-linear zero-knowledge arguments. In *International Cryptology Conference (CRYPTO)*, 2009.

[75] A. Haeberlen, P. Kouznetsov, and P. Druschel. PeerReview: Practical accountability for distributed systems. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 175–188, Oct. 2007.

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

[77] A. E. Hoerl and R. W. Kennard. Ridge regression: Biased estimation for nonorthogonal problems. *Technometrics*, 12(1):55–67, 1970.

[78] Y. Ishai. Personal communication, June 2012.

[79] Y. Ishai, E. Kushilevitz, and R. Ostrovsky. Efficient arguments without short PCPs. In *IEEE Conference on Computational Complexity (CCC)*, pages 278–291, June 2007.

[80] K. Jang, S. Han, S. Han, S. Moon, and K. Park. SSLShader: Cheap SSL acceleration with commodity processors. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2011.

[81] G. O. Karame, M. Strasser, and S. Capkun. Secure remote execution of sequential computations. In *International Conference on Information and Communications Security (ICICS)*, 2009.

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

[83] 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.

[84] J. Kilian. Improved efficient arguments (preliminary version). In *International Cryptology Conference (CRYPTO)*, pages 311–324, Aug. 1995.

[85] D. E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming*. Addison-Wesley, third edition, 1997.

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

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

[88] J. Li, M. N. Krohn, D. Mazières, and D. Shasha. Secure untrusted data repository (SUNDR). In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 121–136, Dec. 2004.

[89] H. Lipmaa. Progression-free sets and sublinear pairing-based non-interactive zero-knowledge arguments. In *Theory of Cryptography Conference (TCC)*, 2012.

[90] 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.

[91] U. Maheshwari, R. Vingralek, and W. Shapiro. How to build a trusted database system on untrusted storage. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 135–150, Oct. 2000.

[92] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella. Fairplay—a secure two-party computation system. In *USENIX Security*, pages 287–302, Aug. 2004.

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

[94] T. Mateer. *Fast Fourier Transform algorithms with applications*. PhD thesis, Clemson University, 2008.

[95] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of applied cryptography*. CRC Press, 2001.

[96] R. C. Merkle. A digital signature based on a conventional encryption function. In *International Cryptology Conference (CRYPTO)*, pages 369–378, Aug. 1987.

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

[98] D. Micciancio and O. Regev. Lattice-based cryptography. In D. J. Bernstein and J. Buchmann, editors, *Post-quantum Cryptography*, pages 147–191. Springer, 2008.

[99] F. Monrose, P. Wycko, and A. D. Rubin. Distributed execution with remote audit. In *Network & Distributed System Security Symposium (NDSS)*, pages 103–113, Feb. 1999.

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

[101] 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 (S&P)*, pages 334–348, May 2013.

[102] M. Osadchy, B. Pinkas, A. Jarrous, and B. Moskovich. SCiFI – a system for secure face identification. In *IEEE Symposium on Security and Privacy (S&P)*, pages 239–254, May 2010.

[103] C. Papamanthou, E. Shi, and R. Tamassia. Signatures of correct computation. In *Theory of Cryptography Conference (TCC)*, pages 222–242, Mar. 2013.

[104] B. Parno, C. Gentry, J. Howell, and M. Raykova. Pinocchio: Nearly practical verifiable computation. In *IEEE Symposium on Security and Privacy (S&P)*, pages 238–252, May 2013.

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

[106] T. P. Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In *International Cryptology Conference (CRYPTO)*, pages 129–140, Aug. 1991.

[107] N. Pippenger and M. J. Fischer. Relations among complexity measures. *Journal of the ACM*, 26(2):361–381, Apr. 1979.

[108] R. A. Popa, H. Balakrishnan, and A. Blumberg. VPriv: Protecting privacy in location-based vehicular services. In *USENIX Security*, pages 335–350, Aug. 2009.

[109] B. Przydatek, D. Song, and A. Perrig. SIA: Secure information aggregation in sensor networks. In *ACM Conference on Embedded Networked Sensor Systems (Sensys)*, 2003.

[110] 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.

[111] 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.

[112] 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 *The European Conference on Computer Systems (EuroSys)*, pages 71–84, Apr. 2013.

[113] S. Setty, R. McPherson, A. J. Blumberg, and M. Walfish. Making argument systems for outsourced computation practical (sometimes). In *Network & Distributed System Security Symposium (NDSS)*, Feb. 2012.

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

[115] 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.

[116] H. Shacham and B. Waters. Compact proofs of retrievability. In *International Conference on the Theory and Application of Cryptology and Information Security (AsiaCrypt)*, pages 90–107, Dec. 2008.

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

[118] R. Sion. Query execution assurance for outsourced databases. In *International Conference on Very Large Data Bases (VLDB)*, pages 601–612, Aug. 2005.

[119] J. Thaler. Time-optimal interactive proofs for circuit evaluation. In *International Cryptology Conference (CRYPTO)*, pages 71–89, Aug. 2013.

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

[121] S. Theodoridis and K. Koutroumbas. *Pattern Recognition, Third Edition*. Academic Press, Inc., 2006.

[122] 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 (PET)*, pages 185–201, Aug. 2009.

[123] V. Vu, S. Setty, A. J. Blumberg, and M. Walfish. A hybrid architecture for interactive verifiable computation. In *IEEE Symposium on Security and Privacy (S&P)*, pages 223–237, May 2013.

[124] 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.

[125] C. Wang, K. Ren, J. Wang, and K. M. R. Urs. Harnessing the cloud for securely outsourcing large-scale systems of linear equations. In *International Conference on Distributed Computing Systems (ICDCS)*, 2011.

[126] D. A. Wheeler. SLOCCount. `http://www.dwheeler.com/sloccount/`.

[127] A. C.-C. Yao. How to generate and exchange secrets. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, 1986.