6

Front Ends: Turning Computer Programs Into Circuits

6.1 Introduction

In Section 4.6, we saw a very efficient interactive proof, called the GKR protocol, for verifiably outsourcing the evaluation of large arithmetic circuits, as long as the circuit is not too deep. But in the real world, people are rarely interested in evaluating giant arithmetic circuits. Rather, they typically have a computer program written in a high-level programming language like Java or Python, and want to execute the program on their data. In order for the GKR protocol to be useful in this setting, we need an efficient way to turn high-level computer programs into arithmetic circuits. We can then apply the GKR protocol (or any other interactive proof or argument system for circuit evaluation) to the resulting arithmetic circuit.

Most general purpose argument system implementations work in this two-step manner. First, a computer program is compiled into a model amenable to probabilistic checking, such as an arithmetic circuit or arithmetic circuit satisfiability instance.[1] Second, an interactive proof or

[1] Many argument systems prefer to work with models such "Rank-1 Constraint Systems" that are generalizations of arithmetic circuits. These alternative models are discussed later in this survey (see Section 8.4).

argument system is applied to check that the prover correctly evaluated the circuit. In these implementations, the program-to-circuit compiler is referred to as the *front end* and the argument system used to check correct evaluation of the circuit is called the *back end*.

Some computer programs naturally lend themselves to implementation via arithmetic circuits, particularly programs that only involve addition and multiplication of integers or elements of a finite field. For example, the following layered arithmetic circuit of fan-in two implements the standard naive $O(n^3)$ time algorithm for multiplying two $n \times n$ matrices, A and B.

Let $[n] \coloneqq \{1, \ldots, n\}$. Adjacent to the input layer of the circuit is a layer of n^3 multiplication gates, each assigned a label $(i, j, k) \in [n] \times [n] \times [n]$. Gate (i, j, k) at this layer computes the product of $A_{i,k}$ and $B_{k,j}$. Following this layer of multiplication gates lies a binary tree of addition gates of depth $\log_2(n)$. This ensures that there are n^2 output gates, and if we assign each output gate a label $(i, j) \in [n] \times [n]$, then the (i, j)'th output gate computes $\sum_{k \in [n]} A_{i,k} \cdot B_{k,j}$ as required by the definition of matrix multiplication. See Figure 6.1.

As another example, Figures 4.8–4.11 portray an arithmetic circuit implementing the same functionality as the computer program depicted in Algorithm 1 (with $n = 4$). The circuit devotes one layer of gates to squaring each input entry, and then sums up the results via a complete binary tree of addition gates of fan-in two.

Algorithm 1 Algorithm Computing the Sum of Squared Entries of Input Vector

Input: Array $a = (a_1, \ldots, a_n)$

1: $b \leftarrow 0$
2: **for** $i = 1, 2, \ldots, n$ **do**
3: $\quad b \leftarrow b + a_i^2$

Output: b

While it is fairly straightforward to turn the algorithm for naive matrix multiplication into an arithmetic circuit as above, other kinds of computer programs that perform "non-arithmetic" operations, such as

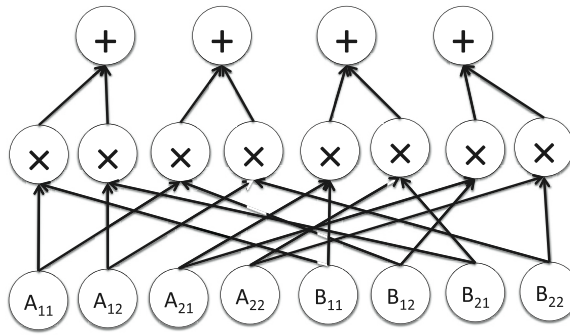

Figure 6.1: An arithmetic circuit implementing the naive matrix multiplication algorithm for 2×2 matrices.

evaluating complicated conditional statements, seem to be much more difficult to turn into small arithmetic circuits.

In Sections 6.3 and 6.4, we will see two techniques for turning arbitrary computer programs into circuits. In Section 6.5, we will see a third technique, which is far more practical, but makes use of what are sometimes called "non-deterministic circuits" and "circuits with auxiliary input". Equivalently, the third technique produces instances of the circuit *satisfiability* problem, rather than of the circuit *evaluation* problem.

We would like to make statements like "Any computer programming that halts within T time steps can be turned into a low-depth, layered, fan-in two arithmetic circuit of size at most $O(T \log T)$." In order to make statements of this form, we first have to be precise about what it means to say that a computer programs has runtime T.

6.2 Machine Code

Modern compilers are very good at efficiently turning high-level computer programs into *machine code*, which is a set of basic instructions that can each be executed in unit time on the machine's hardware. When we say that a program runs in $T(n)$ time steps, we mean that it can be compiled into a sequence of machine instructions of length at most $T(n)$. But for this statement to be precise, we have to decide

precisely what is a machine instruction. That is, we have to specify a model of the hardware on which we will think of our programs as running.

Our hardware model will be a simple *Random Access Machine* (RAM). A RAM consists of the following components.

- (Main) Memory. That is, it will contain s cells of storage, where each cell can store, say, 64 bits of data.
- A constant number (say, 8) of registers. Registers are special memory cells with which the RAM can manipulate data. That is, whereas Main Memory cells can only store data, the RAM is allowed to perform operations on data in registers, such as "add the numbers in Registers 1 and 2, and store the result in Register 3".
- A set of $\ell = O(1)$ allowed machine instructions. Typically, these instructions are of the form:
  - Write the value currently stored in a given register to a specific location in Main Memory.
  - Read the value from a specific location in Main Memory into a register.
  - Perform basic manipulations of data in registers. For example, adding, subtracting, multiplying, dividing, or comparing the values stored in two registers, and storing the result in a third register. Or doing bitwise operations on the values stored in two registers (e.g., computing the bit-wise AND of two values).
- A program counter. This is a special register that tells the machine what is the next instruction to execute.

6.3 A First Technique For Turning Programs Into Circuits [Sketch]

Our first technique for turning computer programs into circuits yields the following.[2] If a computer program runs in time $T(n)$ on a RAM with at most $s(n)$ cells of memory, then the program can be turned into a layered, fan-in 2 arithmetic circuit of depth not much more than $T(n)$ and *width* of about $s(n)$ (i.e., the number of gates at each layer of the circuit is not much more than $s(n)$).

Observe that such a transformation from programs to circuits is useless in the context of the GKR protocol, because the verifier's time complexity in the GKR protocol is at least the circuit depth, which is about $T(n)$ in this construction. In time $T(n)$, the verifier could have executed the entire program on her own, without any help from the prover. We describe this circuit-generation technique because it is conceptually important, even though it is useless in the context of the GKR protocol.

This program-to-circuit transformation makes use of the notion of a machine *configuration*. A machine configuration tells you everything about the state of a RAM at a given time. That is, it specifies the input, as well as the value of every single memory cell, register, and program counter. Observe that if a RAM has a memory of size s, then a configuration can be specified with roughly $64s$ bits (where 64 is the number of bits that can fit in one memory cell), plus some extra bits to specify the input and the values stored in the registers and program counter.

The basic idea of the transformation is to have the circuit proceed in iterations, one for each time step of the computer program. The ith iteration takes as input the configuration of the RAM after i steps of the program have been executed, and "executes one more step of the

[2] The transformation described in this section can yield either Boolean circuits (with AND, OR, or NOT gates) or arithmetic circuits (whose inputs are elements of some finite field $\mathbb{F}$ and whose gates compute addition and multiplication over the field). In fact, any transformation to Boolean circuits implies one to arithmetic circuits, since we know from Section 4.2 that any Boolean circuit can be transformed into an equivalent arithmetic circuit over any field, with at most a constant-factor blowup in size.

program". That is, it determines what the configuration of the RAM would be after the $(i+1)$'st machine instruction is executed. This is displayed pictorially in Figure 6.2.

A key point that makes this transformation work is that there is only a constant number of possible machine instructions, each of which is very simple (operating on only a constant number of registers, in a simple manner). Hence, the circuitry that maps the configuration of the machine after i steps of the program to the configuration after the $(i+1)$'st step is very simple.

Unfortunately, the circuits that are produced by this transformation have size $\tilde{\Theta}(T(n) \cdot s(n))$, meaning that, relative to running the computer program (which takes time $T(n)$), even writing down or reading the circuit is more expensive by a factor of $s(n)$.[3] The source of the inefficiency is that, for each time step of the RAM, the circuit produces an entire new machine configuration. Each configuration has size $\tilde{\Theta}(s)$, as a configuration must specify the state of the RAM's memory at that time step. Conceptually, while each step of the program only alters a constant number of memory cells in each time step, the circuit does not "know in advance" which memory cells will be updated. Hence, the circuit has to explicitly check, for each memory cell at each time step, whether or not the memory cell should be updated. This causes the circuit to be at least $s(n)$ times bigger than the runtime $T(n)$ of the RAM that the circuit simulates. This overhead renders this program-to-circuit transformation impractical.

[3] A second issue is that the circuit is very deep, i.e., depth at least $T(n)$. Because the GKR protocol's (Section 4.6) communication cost grows linearly with circuit depth, applying the GKR protocol to this circuit leads to communication cost at least $T(n)$, which is trivial (in the time required to read the prover's message, the verifier could afford to execute M on its own). This issue will be addressed in Section 6.4 below, which reduces the circuit depth to polynomial in the *space* usage rather than *runtime* of the RAM. However, this comes at the cost of increasing the circuit size from $\text{poly}(T, s)$ to $2^{\Theta(s)}$. Note that argument systems covered later in this monograph do not have communication cost growing linearly with circuit depth, and hence applying these arguments to deep circuits such as those described in this section does yield non-trivial protocols.

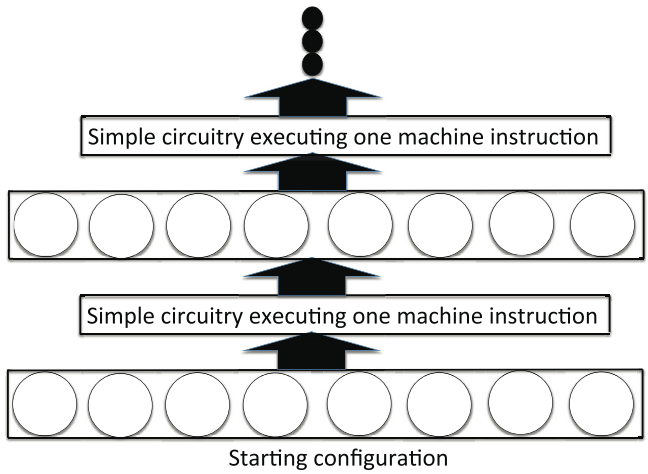

Figure 6.2: A caricature of a technique for turning any program running in time $T(n)$ and space $s(n)$ into a circuit of depth not much more than $T(n)$ and width not much more than $s(n)$.

6.4 Turning Small-Space Programs Into Shallow Circuits

The circuits that come out of the program-to-circuit transformation of Section 6.3 are useless in the context of the GKR protocol because the circuits have depth at least T. When applying the GKR protocol to such a deep circuit, the runtime of the verifier is at least T. It would be just as fast for the verifier to simply run the computer program on its own, without bothering with a prover.

A second technique for turning computer programs generates shallower circuits as long as the computer program doesn't use much space. Specifically, it is capable of taking any program that runs in time T and space s, and turning it into a circuit of depth roughly $s \cdot \log T$ and size $2^{\Theta(s)}$.

Section 4.5.5 explained how we can determine whether a Turing Machine M outputs 1 on input x in less than T time steps by determining whether there is a directed path of length at most T in M's configuration graph from the starting configuration to the accepting configuration. While that section discussed Turing Machines, the same result also applies to Random Access Machines, because both Turing Machine

or Random Access Machine using s bits of space have at most $2^{O(s)}$ configurations.

To solve this directed-path problem, it suffices to compute a single entry of the T'th power of A, where A be the adjacency matrix of M's configuration graph; that is, $A_{i,j} = 1$ if configuration i has a directed edge to configuration j in M's configuration graph, and $A_{i,j} = 0$ otherwise. This is because the (i, j)'th entry of the T'th power of A equals the number of directed paths of length T from node i to node j in the configuration graph of M. Hence, in order to determine whether there is a directed path of length T from the starting configuration of M to the accepting configuration, it is enough for the circuit to repeatedly square the adjacency matrix $\log_2 T$ times. We have seen in Section 6.1 that there is a circuit of size $O(N^3)$ and depth $O(\log N)$ for multiplying two $N \times N$ matrices. Since the configuration graph of M on input x is an $2^{\Theta(s)} \times 2^{\Theta(s)}$ matrix, the circuit that squares the adjacency matrix of the configuration graph of M $O(\log T)$ times has depth $O(\log(2^{\Theta(s)}) \cdot \log T) = O(s \log T)$, and size $2^{\Theta(s)}$.[4]

Hence, one can obtain an IP for determining the output of the RAM M by applying the GKR protocol to this circuit. However, the IP of Section 4.5.5 solves the same problem in a more direct and efficient manner.

6.5 Turning Computer Programs Into Circuit Satisfiability Instances

6.5.1 The Circuit Satisfiability Problem

In Sections 6.3 and 6.4, we saw two methods for turning computer programs into arithmetic circuits. The first method was undesirable for two reasons. First, it yielded circuits of very large depth. So large, in fact, that applying the GKR protocol to the resulting circuits led to a verifier runtime that was as bad as just having the verifier run the

[4] To be more precise, the circuit takes as input x, and first computes the adjacency matrix A of the configuration graph of M on input x. Each entry of A is a simple function of x. Then the circuit repeatedly squares A to compute the $T(n)$'th power of A and the outputs the (i, j)'th entry, where i indexes the starting configuration and j the ending configuration.

entire program without any help from a prover. Second, if the computer program ran in time T and space s, the circuit had size at least $T \cdot s$, and we'd really prefer to have circuits with close to T gates.

In this section, we are going to address both of these issues. However, to do so, we are going to have to shift from talking about circuit *evaluation* to talking about circuit *satisfiability*.

Recall that in the arithmetic circuit evaluation problem, the input specifies an arithmetic circuit $\mathcal{C}$, input x, and output(s) y, and the goal is to determine whether $\mathcal{C}(x) = y$. In the arithmetic circuit *satisfiability* problem (circuit-SAT for short), the circuit $\mathcal{C}$ takes *two inputs*, x and w. The first input x is public and fixed, i.e., known to both the prover and verifier. The second input w is often called the *witness*, or sometimes the *non-deterministic input* or *auxiliary input*. Given the first input x and output(s) y, the goal is to determine whether *there exists* a w such that $\mathcal{C}(x, w) = y$.

Preview: Succinct Arguments for Circuit Satisfiability, and Outline for the Remainder of the Section. After this section, the remainder of the monograph is devoted to developing succinct arguments, especially so-called *SNARKs*, for circuit satisfiability and generalizations thereof. These protocols will enable the untrusted prover $\mathcal{P}$ to prove that it *knows* a witness w such that $\mathcal{C}(x, w) = y$. Ideally, the proof size and verification time of the SNARK will be far smaller than they are in the naive proof system, in which $\mathcal{P}$ sends the witness w to $\mathcal{V}$ and $\mathcal{V}$ evaluates $\mathcal{C}(x, w)$ on its own. And ideally, if $\mathcal{P}$ already knows the witness—and hence does not have to spend any time to find it—$\mathcal{P}$ will run in time close to that required just to evaluate $\mathcal{C}$ on input (x, w).

In applications, $\mathcal{P}$ will typically already know w. For example, Section 1 discussed an application in which Alice chooses a random password w, publishes a cryptographic hash $y = h(w)$, and later wants to prove to Bob that she knows a pre-image of y under h. The witness in this application is w. Effectively, Alice herself generated the statement she wishes to prove, and hence she knows the witness without needing to spend massive compute power to compute it "from scratch", which in this example would entail inverting the hash function h at y.

Such applications entail a major shift in thinking, compared to the interactive proofs for circuit *evaluation* already covered. No longer is $\mathcal{P}$ claiming to have applied a specific circuit $\mathcal{C}$ or run a specific RAM M on a public input x that is known to both verifier and prover. Rather, $\mathcal{P}$ is claiming it knows *some* witness w (not known to the verifier) such that applying $\mathcal{C}$ to (x, w), or running M on (x, w), yields output y.

But as we will see, arguments for circuit satisfiability are useful *even when* $\mathcal{P}$ is only claiming to have run a specific RAM M on a public input x. In this case, the witness w is "used" merely to enable more efficient transformations of the machine M into an "equivalent circuit" $\mathcal{C}$. By this, we mean that the output of the RAM M on input x equals y if and only if there exists a w such that $\mathcal{C}(x, w) = y$.

In more detail, the remainder of the section explains that any computer program running in time T can be efficiently transformed into an equivalent instance $(\mathcal{C}, x, y)$ of arithmetic circuit satisfiability, where the circuit $\mathcal{C}$ has size close to T, and depth close to $\log T$. The output of the program on input x equals y if and only if there exists a w such that $\mathcal{C}(x, w) = y$. Moreover, any party (such as a prover) that actually runs the program on input x can easily construct a w satisfying $\mathcal{C}(x, w) = y$.

Why Circuit-SAT Instances Are Expressive. Intuitively, circuit satisfiability instances should be "more expressive" than circuit evaluation instances, for the same reason that *checking* a proof of a claim tends to be easier then discovering the proof in the first place. In the coming sections, the claim at hand is "running a designated computer program on input x yields output y". Conceptually, the witness w in the circuit satisfiability instances that we construct in the remainder of this section will represent a traditional, static *proof* of the claim, and the circuit $\mathcal{C}$ will simply check that the proof is valid. Unsurprisingly, we will see that *checking validity of a proof* can be done by much smaller circuits than circuit evaluation instances that determine the veracity of the claim "from scratch".

To make the above intuition more concrete, here is a specific, albeit somewhat contrived, example of the power of circuit satisfiability instances. Imagine a straightline program in which all inputs are elements

of some finite field $\mathbb{F}$, and all operations are addition, multiplication, and division (by division a/b, we mean multiplying a by the multiplicative inverse of b in $\mathbb{F}$). Suppose one wishes to turn this straightline program into an equivalent arithmetic circuit *evaluation* instance $\mathcal{C}$. Since the gates of $\mathcal{C}$ can only compute addition and multiplication operations (not division), $\mathcal{C}$ would need to replace every division operation a/b with an explicit computation of the multiplicative inverse b^{-1} of b, where the computation of b^{-1} is only able to invoke addition and multiplication operations. This is expensive, leading to huge circuits. In contrast, to turn the straightline program into an equivalent circuit *satisfiability* instance, we can demand that the witness w contain a field element e for every division operation a/b where e should be set to b^{-1}. The circuit can "check" that $e = b^{-1}$ by adding an output gate that computes $e \cdot b - 1$. This output gate will equal 0 if and only if $e = b^{-1}$. In this manner, each division operation in the straightline program translates into only $O(1)$ additional gates and witness elements in the circuit satisfiability instance.

One may initially be worried that this techniques introduces a "checker" output gate for each division operation in the straightline program, and that consequently, if there are many division operations the prover will have to send a very long message to the verifier in order to inform the verifier of the claimed output vector y of $\mathcal{C}$. However, since any "correct" witness w causes these "checker" gates to evaluate to 0, their claimed values are implicitly 0. This means that the size of the prover's message specifying the claimed output vector y is independent of the number of "checker" output gates in $\mathcal{C}$.

6.5.2 Preview: How Do Succinct Arguments for Circuit Satisfiability Operate?

One way we could design an efficient IP for the claim that there exists a w such that $\mathcal{C}(x, w) = y$ is to have the prover send w to the verifier, and run the GKR protocol to efficiently check that $\mathcal{C}(x, w) = y$. This would be enough to convince the verifier that indeed the program outputs y on input (x, w). This approach works well if the witness w is small. But in the computer-program-to-circuit-satisfiability transformation that

we're about to see, the witness w will be very large, of size roughly T, the runtime of the computer program. So even asking the verifier to read the claimed witness w is as expensive as asking the verifier to simply run the program herself without the help of a prover.

Fortunately, we will see in Section 7.3 that we can combine the GKR protocol with a cryptographic primitive called a *polynomial commitment scheme* to obtain an argument system that avoids having the prover send the entire witness w to the verifier. The high-level idea is as follows (see Section 7.3 for details).

In the IP for circuit satisfiability described two paragraphs above, it was essential that the prover sent the witness w at the start of the protocol, so that the prover was not able to base the choice of w on the random coin tosses of the verifier within the circuit evaluation IP to confirm that $\mathcal{C}(x, w) = y$. Put another way, the point of sending w at the start of the protocol was that it *bound* the prover to a specific choice of w *before* the prover knew anything about the verifier's random coin tosses in the subsequent IP for circuit evaluation.

We can mimic the above, without the prover having to send w in full, using cryptographic commitment schemes. These are cryptographic protocols that have two phases: a commit phase, and a reveal phase. In a sense made precise momentarily, the commit phase binds the prover to a witness w without requiring the prover to send w in full. In the reveal phase, the verifier asks the prover to reveal certain entries of w. The required binding property is that, unless the prover can solve some computational task that is assumed to be intractable, then after executing the commit phase, there must be some fixed string w such that the prover is forced to answer all possible reveal-phase queries in a manner consistent with w. Put another way, the prover is not able to choose w in a manner that depends on the questions asked by the verifier in the reveal phase.

This means that to obtain a succinct argument for circuit satisfiability, one can first have the prover run the commit phase of a cryptographic commitment scheme to bind itself to the witness w, then run an IP or argument for circuit evaluation to establish that $\mathcal{C}(x, w) = y$, and over the course of the protocol the verifier can force the prover as necessary

to reveal any information about w that the verifier needs to know to perform its checks.

If the GKR protocol is used as the circuit evaluation protocol, what information does the verifier need to know about w to perform its checks? Recall that in order to run the GKR protocol on circuit $\mathcal{C}$ with input $u = (x, w)$, the only information about the input that the verifier needs to know is the evaluation of the multilinear extension $\tilde{u}$ of u at a random point. Moreover, this evaluation is only needed by the verifier at the very end of the protocol.

We explain in Section 7 that in order to quickly evaluate $\tilde{u}$ at any point, it is enough for the verifier to know the evaluation of the multilinear extension $\tilde{w}$ of w at a related point.

Hence, the cryptographic commitment scheme should bind the prover to the multilinear polynomial $\tilde{w}$, in the sense that in the reveal phase of the commitment scheme, the verifier can ask the prover to tell her $\tilde{w}(r)$ for any desired input r to w. The prover will respond with $\tilde{w}(r)$ and a small amount of "authentication information" that the verifier insists be included to enforce binding. The required binding property roughly ensures that when the verifier asks the prover to reveal $\tilde{w}(r)$, the prover will not be able to "change" its answer in a manner that depends on r.

To summarize the resulting argument system, after the prover commits to the multilinear polynomial $\tilde{w}$, the parties run the GKR protocol to check that $\mathcal{C}(x, w) = y$. The verifier can happily run this protocol even though it does not know w, until the very end when the verifier has to evaluate $\tilde{u}$ at a single point. This requires the verifier to learn $\tilde{w}(r)$ for some point r. The verifier learns $\tilde{w}(r)$ from the prover, by having the prover decommit to $\tilde{w}$ at input r.

All told, this approach (combined with the Fiat-Shamir transformation, Section 5.2) will lead to a non-interactive argument system of knowledge for circuit satisfiability, i.e., for the claim that the prover knows a witness w such that $\mathcal{C}(x, w) = y$. If a sufficiently efficient polynomial commitment scheme is used, the argument system is nearly optimal in the sense that the verifier runs in linear time in the size of the input x, and the prover runs in time close to the size of $\mathcal{C}$.

6.5.3 The Transformation From Computer Programs To Arithmetic Circuit Satisfiability

Before describing the transformation, it is helpful to consider why the circuit generated in Method 1 of Section 6 (see Section 6.3) had at least $T \cdot s$ gates, which is significantly larger than T if s is large. The answer is that that circuit consisted of T "stages" where the ith stage computed each bit of the machine's configuration—which includes the entire contents of its main memory—after i machine instructions had been executed.

But each machine instruction affects the value of only $O(1)$ registers and memory cells, so between any two stages, almost all bits of the configuration remain unchanged. This means that almost all of the gates and wires in the circuit are simply devoted to copying bits from the configuration after i steps to the configuration after step $i+1$. This is highly wasteful, and in order to obtain a circuit of size close to T, rather than $T \cdot s$, we will need to cut out all of this redundancy.

To describe the main idea in the transformation, it is helpful to introduce the notion of the *transcript* (sometimes also called a *trace*) of a Random Access Machine M's execution on input x. Roughly speaking, the transcript describes just the *changes* to M's configuration at each step of its execution. That is, for each step i that M takes, the transcript lists just the value of each register and the program counter at the end of step i. Since M has only $O(1)$ registers, the transcript can be specified using $O(T)$ words, where a *word* refers to a value that can be stored in a single register or memory cell.

The basic idea is that the transformation from RAM execution to circuit satisfiability produces a circuit satisfiability instance $(\mathcal{C}, x, y)$, where x is the input to M, y is the claimed output of M, and the witness w is supposed to be the transcript of M's execution of input x. The circuit $\mathcal{C}$ will simply check that w is indeed the transcript of M's execution on input x, and if this check passes, then $\mathcal{C}$ outputs the same value as M does according to the ending configuration in the transcript. If the check fails, $\mathcal{C}$ outputs a special rejection symbol.

A schematic of $\mathcal{C}$ is depicted in Figure 6.3.

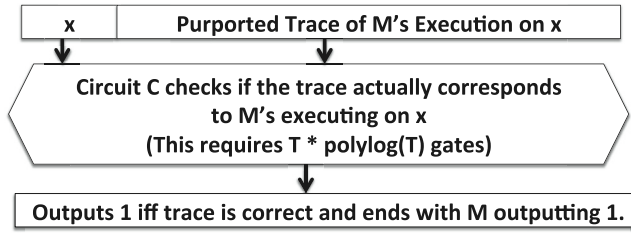

Figure 6.3: Sketch of the transformation From RAM execution on input x to an instance of circuit satisfiability.

6.5.4 Details of the Transformation

The circuit $\mathcal{C}$ takes an entire transcript of the entire execution of M on x as a non-deterministic input, where a transcript consists of (timestamp, list) pairs, one for each step taken by M. Here, a list specifies the bits contained in the current program counter and the values of all of M's registers. If a read or write operation occurs at the associated timestep of M's execution, the list also includes the memory location accessed and the value returned if a read operation occurs or written to memory if a write operation occurs. Timesteps at which no read or write operation occurs contain a special memory address indicating as much.

The circuit then checks that the transcript is valid. Details of the validity check follow.

Conceptually, one can think of a RAM running in time T as comprised of two independent pieces:

- The maintenance of its Main Memory, meaning correctly implementing all memory reads and writes. Each memory read should return the most recent value written to that memory cell.

- Assuming memory is maintained correctly, execute each of the T steps of the program. Each step of the program is simple, as it only affects the machine's registers, of which there are only a constant number, and performs at most one read or write operation to Main Memory.

Following the above conceptual partition of a RAM's operation, checking the validity of a purported transcript amounts to checking that it satisfies the following two properties.

- Memory consistency: whenever a value is read from a memory location, check that the value that the transcript claims is returned is equal to the last value written to that location.
- Time consistency: *assuming that memory consistency holds*, check that for each timestep $i \in \{1, \ldots, T-1\}$, the claimed state of the machine at time $i+1$ correctly follows from the machine's claimed state at time i.

The circuit checks time-consistency by representing the transition function of the RAM as a small sub-circuit. We provide some details of this representation in Section 6.5.4.1. It then applies this sub-circuit to each entry i of the transcript and checks that the output is equal to entry $i+1$ of the transcript. That is, for each time step i in $1, \ldots, T-1$, the circuit will have an output gate that will equal 0 if and only if entry $i+1$ of the transcript equals that application of the transition function to entry i of the transcript.

The circuit checks memory consistency by reordering the transcript entries based on the memory location read from or written to, with ties broken by time. That is, for each memory location, the read and write operations for that location appear in increasing order of timestamp. We refer to this reordering of the transcript as *memory ordering*. Transcript entries that do not perform either a memory read or write operation can be grouped together in any order and placed at the end of the reordered transcript–these entries will be ignored by the part of the circuit that checks memory consistency.

Given the memory-ordered transcript, it is straightforward for the circuit to check that every memory read from a given location returns the last value written to that location. For any two adjacent entries in the memory-ordered transcript, the circuit checks whether the associated memory locations are equal, and whether the latter entry contains a read operation. If so, it checks that the value returned by the read operation equals the value read or written in the preceding operation.

The sorting step is the most conceptually involved part of the construction of $\mathcal{C}$, and is discussed in Section 6.5.4.2. Note that all of the at most T time-consistency checks and memory-consistency checks can be done in parallel. As we will see, sorting can also be done in logarithmic depth. All together, this ensures that $\mathcal{C}$ has polylogarithmic depth.

6.5.4.1 Representing Transition Functions of RAMs as Small Arithmetic Circuits

Depending on the field over which the circuit $\mathcal{C}$ is defined, certain operations of the RAM are easy to compute inside $\mathcal{C}$ using a single gate. For example, if $\mathcal{C}$ is defined over a prime-order field $\mathbb{F}_p$ of order p, then this field naturally simulates integer addition and multiplication so long as one is guaranteed that the values arising in the computation always lie in the range $[-\lfloor p/2 \rfloor, \lfloor p/2 \rfloor]$.[5] If the values grow outside of this range, then the field, by reducing all values modulo p, will no longer simulate integer arithmetic. In contrast, fields of characteristic 2 are not able to simulate integer addition or multiplication on numbers of magnitude 2^W without spending (at least) $\Omega(W)$ gates by operating on the bit-representations of the integers. On the other hand, if $\mathcal{C}$ is defined over a field of characteristic two, then addition of two field elements is equivalent to bitwise-XOR of the binary representations of the field operations. The message here is that integer arithmetic, but not bitwise operations, are simulated very directly over fields of large prime order (up to overflow issues), whereas bitwise operations, but not integer arithmetic, are simulated very directly over fields of characteristic 2.

In general, if a Random Access Machine has word size W then any primitive instruction other than memory accesses (e.g., integer arithmetic, bitwise operations, integer comparisons, etc.) can be implemented in a circuit-satisfiability instance using $\text{poly}(W)$ many gates. This works by representing each bit of each register with a separate field element, and implementing the instruction bitwise. To give some simple examples,

[5] If operating over unsigned integers rather than signed integers, the integer values arising in the computation may lie in the range $[0, p-1]$ rather than $[-\lfloor p/2 \rfloor, \lfloor p/2 \rfloor]$.

one can compute the bitwise-AND of two values $x, y \in \{0,1\}^W$ with W multiplication gates over a large prime-order field, where the ith multiplication gate multiplies x_i by y_i. Bitwise-OR and Bitwise-XOR can be computed in a similar manner, replacing $x_i \cdot y_i$ with $x_i + y_i - x_i y_i$ and $x_i + y_i - 2x_i y_i$ respectively.

As a more complicated example, suppose the circuit is defined over a field $\mathbb{F}_p$ of large prime order. Let a and b be two field elements interpreted as integers in $\{0, 1, \ldots, p-1\}$, and suppose that one wishes to determine whether $a > b$. Let $\ell := \lceil \log_2 p \rceil$. The circuit can ensure that the witness contains 2ℓ bits $a_0, \ldots, a_{\ell-1}, b_1, \ldots, b_{\ell-1}$ representing the binary representations of a and b respectively as follows. First, to check that a_i is in $\{0,1\}$ for all $i = 0, \ldots, \ell-1$, the circuit can include an output gate computing $a_i^2 - a_i$. This gate will evaluate to 0 if and only if $a_i \in \{0,1\}$. Second, to check that $(a_0, \ldots, a_{\ell-1})$ is indeed the binary representation of $a \in \mathbb{F}_p$, the circuit can include an output gate computing $a - \sum_{i=0}^{\ell} a_i 2^i$. Assuming each $a_i \in \{0,1\}$, this output gate equals 0 if and only if $(a_0, \ldots, a_{\ell-1})$ is the binary representation of a_i. Analogous checks can be included in the circuit to ensure that $(b_0, \ldots, b_{\ell-1})$ is the binary representation of b.

Finally, the circuit can include an output gate computing an arithmetization of the Boolean circuit that checks bit-by-bit whether $a > b$. Specifically, for $j = \ell-2, \ell-3, \ldots, 1$, define

$$A_j := \prod_{j'>j} \left(a_{j'} b_{j'} + (1 - a_{j'})(1 - b_{j'})\right)$$

so that $A_j = 1$ if the $\ell - j$ high-order bits of a and b are equal. Then the following expression equals 1 if $a > b$ and 0 otherwise:

$$a_{\ell-1}(1 - b_{\ell-1}) + A_{\ell-2} a_{\ell-2}(1 - b_{\ell-2}) + \cdots + A_0 \cdot a_0 \,(1 - b_0)\,.$$

It can be checked that the above expression (which can be computed by an arithmetic circuit of depth $O(\ell) = O(\log p)$ consisting of $O(\ell)$ gates) equals 1 if $a > b$ and 0 otherwise. Indeed, if $a_{\ell-1} = 1$ and $b_{\ell-1} = 0$, then the first term evaluates to 1 and all other terms evaluate to 0, while if $a_{\ell-1} = 0$ and $b_{\ell-1} = 1$, then all terms evaluate to 0. Otherwise, if $a_{\ell-2} = 1$ and $b_{\ell-2} = 0$, then the second term evaluates to 1 and all other terms evaluate to 0, while if $a_{\ell-2} = 0$ and $b_{\ell-2} = 1$ then all terms evaluate to 0. And so on.

There has been considerable effort devoted to developing techniques to more efficiently simulate non-arithmetic operations over fields of large prime order. Section 6.6.3 sketches an important result in this direction, due to Bootle *et al.* [68].

6.5.4.2 How to Sort with a Non-Deterministic Circuit

Recall that to check that a purported transcript for RAM M satisfies memory consistency, the transcript entries must be reordered so that they are grouped by the memory cell read from or written to, with ties broken by time. Below, we describe a method based on so-called *routing networks* that enable such reordering.

The use of routing networks to check memory consistency is conceptually involved, and typically yields larger circuits than simpler alternatives discussed in Section 6.6.1, which uses Merkle trees, and Section 6.6.2, which uses fingerprinting techniques originally introduced in Section 2.1. The reader may skip this section's discussion of routing networks with no loss of continuity.

We cover routing networks both for historical context, and because the alternative transformations do not actually yield a circuit $\mathcal{C}$ such that $M(x) = y$ if and only if there exists a w satisfying $\mathcal{C}(x, w) = y$. The Merkle-trees approach yields a "computationally-sound" transformation. This means that even if $M(x) \neq y$, there will exist witnesses w such that $\mathcal{C}(x, w) = y$, but such witnesses will be computationally infeasible to find—doing so will require finding collisions in a cryptographic hash function. Meanwhile, the fingerprinting techniques use a random field element $r \in \mathbb{F}$ to check memory-consistency. Even if $M(x) \neq y$, for any *fixed* r, it will be easy to find a witness w such that $\mathcal{C}(x, w) = y$. Fingerprinting techniques are therefore only useful in settings where the prover can be forced to "choose" the witness w *before* a random r is selected. Fortunately, neither of the above issues with Merkle-hashing and fingerprinting turns out not to be an obstacle to using such techniques in SNARK design.

Routing Networks. A routing network is a graph with a designated set of T source vertices and a designated set of T sink vertices (both sets

of the same cardinality) satisfying the following property: for any perfect matching between sources and sinks (equivalently, for any desired sorting of the sources), there is a set of node-disjoint[6] paths that connects each source to the sink to which it is matched. Such a set of node-disjoint paths is called a *routing*. The specific routing network used in $\mathcal{C}$ is derived from a *De Bruijn* graph G. G consists of $\ell = O(\log T)$ layers, with T nodes at each layer. The first layer consists of the source vertices, and the last layer consists of the sinks. Each node at intermediate layers has exactly two in-neighbors and exactly two out-neighbors.

The precise definition of the De Bruijn graph G is not essential to the discussion here. What is important is that G satisfies the following two properties.

- Property 1: Given any desired sorting of the sources, a corresponding routing can be found in $O(|G|) = O(T \cdot \log T)$ time using known routing algorithms [31], [40], [180], [245].
- Property 2: The multilinear extension of the wiring predicate of G can be evaluated in polylogarithmic time. By wiring predicate of G, we mean the Boolean function (analogous to the functions add_i and mult_i in the GKR protocol) that takes as input the labels (a, b, c) of three nodes in G, and outputs 1 if and only if b and c are the in-neighbors of a in G.

  Roughly speaking, Property 2 holds because in a De Bruijn graph, the neighbors of a node with label v are obtained from v by simple bit shifts, which is a "degree-1 operation" in the following sense. The function that checks whether two binary labels are bit-shifts of each other is an AND of pairwise disjoint bit-equality checks. The direct arithmetization of such a function (replacing the AND gate with multiplication, and the bitwise equality checks with their multilinear extensions) is multilinear.

In a routing of G, each node v other than the source nodes has exactly one in-neighbor in the routing—we think of this in-neighbor as forwarding its packet to v—and each node v other than the sink nodes

[6] Two length-ℓ paths $u_1 \to u_2 \to \cdots \to u_\ell$ and $v_1 \to v_2 \to \cdots \to v_\ell$ are node-disjoint if there does not exist a pair $(i, j) \in [\ell] \times [\ell]$ such that $u_i = v_j$.

has exactly one out-neighbor in the routing. Thus, a routing in G can be specified by assigning each non-source node v a single bit b_v that specifies which of v's two in-neighbors in G is forwarding a packet to v.

To perform the sorting step, the circuit will take additional bits as non-deterministic input (i.e., as part of the witness w), called *routing bits*, which give the bit-wise specification of a routing as just described. To check memory consistency of a purported transcript, the circuit $\mathcal{C}$ sorts the (timestamp, list) pairs of the transcript into memory order by implementing the routing. This means that for each node v in G, $\mathcal{C}$ contains a "gadget" of logarithmically many gates. The gadget for v takes as input the two (timestamp, list) pairs and the routing bit b_v. Based on the routing bit, it outputs one of the two input pairs. In $\mathcal{C}$, the gadget for v is connected to the gadgets for its two in-neighbors in G. This ensures that the two inputs to v's gadget in $\mathcal{C}$ are the pairs output by v's two in-neighbors in G. One thinks of each (timestamp, list) pair of the transcript as a packet, and of v's gadget outputting a (timestamp, list) pair as v forwarding the packet it receives in the routing to the appropriate out-neighbor of v in G.

Putting Everything Together. For any RAM M running in time T, we have now sketched all of the components of a circuit $\mathcal{C}$ of size $O(T \cdot \text{polylog}(T))$ such that $M(x) = y \iff$ there exists a w satisfying $\mathcal{C}(x, w) = y$. The witness w specifies a purported transcript for M. $\mathcal{C}$ first checks the transcript for time consistency. It then uses a routing network to sort the transcript entries into memory order, meaning sorted by the memory location read from or written to, with ties broken by time. Any routing computes *some* reordering of the original transcript, and the circuit can check with $O(T \cdot \text{polylog}T)$ gates that the reordered transcript is indeed in the prescribed order—this amounts to interpreting the (memory location, timestamp) pair associated with each transcript entry as an integer, and performing one integer comparison for every adjacent pair, to confirm that they are sorted in increasing order (see Section 6.5.4.1 for details of how to implement integer comparisons in arithmetic circuits). Finally, given that the reordered transcript is in the prescribed order, the circuit can easily check that every memory read returns the last value written to that location.

Intuitively, when applying a succinct argument for circuit-satisfiability to $\mathcal{C}$, the verifier is forcing the prover not only to run the RAM M on input x, but also to produce a transcript of the execution and then *confirm* via the circuit $\mathcal{C}$ that the transcript contains no errors. Fortunately, it does not require much more work for the prover to produce the transcript and confirm its correctness then it does to run M on x in the first place.

The Wiring Predicates of $\mathcal{C}$. The circuit $\mathcal{C}$ has a very regular wiring structure, with lots of repeated structure. Specifically, its time-consistency-checking circuitry applies the same small sub-circuit (capturing the transition function of the RAM) independently to every two adjacent (timestep, list) pairs in the time-ordered transcript specified by the witness, and (after resorting the witness into memory order), the memory-consistency-checking circuitry also applies a small sub-circuit independently to adjacent (timestamp, list) pairs in the memory-ordered transcript to check that every memory read from a given location returns the last value written to that location. That is, the parts of the circuit devoted to both time-consistency and memory-consistency checking are data parallel in the sense of Section 4.6.7.

All told, it is possible to exploit this data parallel structure—and Property 2 of the routing network G above, which ensures that the sorting circuitry also has a nice, regular wiring structure—to show that (a slight modification of) the multilinear extensions $\widetilde{\text{add}}_i$ and $\widetilde{\text{mult}}_i$ of $\mathcal{C}$ can be evaluated in polylogarithmic time.

This ensures that if one applies the GKR protocol (in combination with a commitment scheme as described in Section 6.5.2) that the verifier can run in time $O(n + \text{polylog}(T))$, without ever having to explicitly enumerate over all gates of $\mathcal{C}$. Moreover, the prover can generate the entire circuit $\mathcal{C}$ and the witness w, and perform its part of the GKR protocol applied to $\mathcal{C}(x, w)$ in time $O(T \cdot \text{polylog}(T))$.

6.6 Alternative Transformations and Optimizations

The previous section gave a way to turn any RAM M with runtime T into a circuit $\mathcal{C}$ of size $\tilde{O}(T)$ such that the output of M on input

x equals y if and only if there exists a w such that $\mathcal{C}(x, w) = y$. In this section, we relax this requirement on $\mathcal{C}$ in one of two ways. First, in Section 6.6.1 we permit there to be values $y \neq M(x)$ such that there exists a w satisfying $\mathcal{C}(x, w) = y$, but we insist that if there is a polynomial-time prover capable of *finding* a w satisfying $\mathcal{C}(x, w) = y'$, then $y = M(x)$. Satisfying this requirement is still sufficient to ultimately obtain argument systems for RAM execution, by applying an argument system for circuit satisfiability to $\mathcal{C}$.[7] Second, in Section 6.6.2, we permit prover and verifier to interact while performing the transformation from M into $\mathcal{C}$—the interaction can then be removed via the Fiat-Shamir transformation.

In both of these settings, we avoid the use of routing networks in the construction of $\mathcal{C}$. This is desirable because routing networks lead to noticeable concrete and asymptotic overheads in circuit size—routing T items requires a routing network of size $\Omega(T \log T)$, which is superlinear in T.

6.6.1 Ensuring Memory Consistency via Merkle Trees

The point of using routing networks in $\mathcal{C}$ was to ensure memory consistency of the execution trace specified by the witness. An alternative technique for ensuring memory consistency is to use Merkle trees, which are covered later in this survey in Section 7.3.2.2. Roughly speaking, the idea is that $\mathcal{C}$ will insist that every memory read in the transcript is immediately followed by "authentication information" that a polynomial-time prover is only capable of producing if the value returned by the memory read is in fact the last value written to that memory location.[8,9] This leads to a circuit $\mathcal{C}$ such that the only *computationally tractable* method

[7] More precisely, the argument system must be an argument of knowledge. See Section 7.4 for details.

[8] Each write operation must also be accompanied by authentication information to enable appropriately updating the Merkle tree. We omit details for brevity.

[9] A Merkle tree is an example of a cryptographic object called an *accumulator*, which is simply a commitment to a set that furthermore supports succinct proofs of membership in the set. In this section, the relevant set is the (memory location, value) pairs comprising the RAM's memory at a given step of the RAM's execution during which a memory read occurs. In some applications, there can be concrete efficiency advantages to using accumulators other than Merkle trees [200].

of finding a satisfying assignment w for $\mathcal{C}$ is to provide an execution trace that indeed satisfies memory consistency. That is, while there will *exist* satisfying assignments w for $\mathcal{C}$ that do not satisfy memory consistency, finding such assignments w would require identifying collisions in a collision-resistant family of hash functions. This technique can lead to very large circuits if there are many memory operations, because each memory operation must be followed by a full authentication path in the Merkle tree, which consists a sequence of cryptographic hash evaluations (the number of hash evaluations is logarithmic in the size of the memory). All of these hash values must be included in the witness w, and the circuit $\mathcal{C}$ must check that the hash evaluations are computed correctly, which requires the cryptographic hash function to be repeatedly evaluated inside $\mathcal{C}$. Cryptographic hash evaluations can require many gates to implement in an arithmetic circuit. For this and related reasons, there have been significant efforts to identify collision-resistant hash functions that are "SNARK-friendly" in the sense that they can implemented inside arithmetic circuits using few gates [4], [6], [47], [138], [153], [171], [173]. For machines M that perform relatively few memory operations, Merkle trees built with such SNARK-friendly hash functions can be a cost-effective technique for checking memory consistency.

6.6.2 Ensuring Memory Consistency via Fingerprinting

Another technique for checking memory consistency is to use simple fingerprinting-based memory checking techniques (recall that we discussed Reed-Solomon fingerprinting in Section 2.1). The circuit $\mathcal{C}$ resulting from this procedure implements a *randomized* algorithm, in the following sense. In addition to public input x and witness w, $\mathcal{C}$ takes a third input $r \in \mathbb{F}$ and the guarantee is the following: for any pair x, y,

- if $M(x) = y$ then there exists a w such that for every $r \in \mathbb{F}$ such that $\mathcal{C}(x, w, r) = 1$. Moreover, such a w can be easily derived by any prover running M on input x.
- if $M(x) \neq y$, then for *every* w, the probability over a randomly chosen $r \in \mathbb{F}$ that $\mathcal{C}(x, w, r) = 1$ is at most $\tilde{O}(T)/|\mathbb{F}|$.

An important aspect of this transformation to be aware of is that, for any known r, it is easy for a cheating prover to find a w such that $\mathcal{C}(x, w, r) = y$. However, if r is chosen at random from $\mathbb{F}$ independently of w (say, because the prover commits to w and only then is public randomness used to select r), then learning that $\mathcal{C}(x, w, r) = y$ does give very high confidence that in fact $M(x) = y$. This is sufficient to combine the transformation described below with the approach described in Section 6.5.2 to obtain a succinct argument for proving that $M(x) = y$. Indeed, after the prover commits to w (or more precisely to the multilinear extension $\tilde{w}$ of w, using a polynomial commitment scheme), the verifier can then select r at random, and the prover can then run the GKR protocol to convince the verifier that $\mathcal{C}(x, w, r) = 1$. The resulting interactive protocol is public coin, so the interaction can be removed using the Fiat-Shamir transformation.

The idea of the randomized fingerprinting-based memory-consistency-checking procedure implemented within the circuit $\mathcal{C}$ is the following. As we explain shortly, by tweaking the behavior of the machine M (without increasing its runtime by more than a constant factor) it is possible to ensure the following key property holds: a time-consistent transcript for M is also memory consistent if and only if the multiset of (memory location, value) pairs written from memory equals the multiset of (memory location, value) pairs read from memory. This property turns the problem of checking memory consistency into the problem of checking whether two multisets are equal—equivalently, checking whether two lists of (memory location, value) pairs are permutations of each other—and the latter can be solved with fingerprinting techniques.

We now explain how to tweak the behavior of M so as to ensure the problem of checking memory-consistency amounts to checking equality of two multisets, and then explain how to use fingerprinting to solve this latter task.

Reducing memory-consistency-checking to multiset equality checking. Here is how to tweak M to ensure that any time-consistent transcript for M is memory-consistent if and only if the multisets of (memory location, value) pairs written vs. read from memory are identical. This technique dates to work of Blum *et al.* [59] who referred to it as an

offline memory checking procedure. At the start of the computation (time step 0), we have M initialize memory by writing an arbitrary value to each memory location, without preceding these initialization-writes with reads. After this initialization phase, suppose that we insist that every time the machine M writes a value to a memory location, it precedes the write with a read operation from the same location (the result of which is simply ignored by M), and every time M reads a value from a memory location, it follows the read with a write operation (writing the same value that was just read). Moreover, let us insist that every time a value is written to memory, M includes in the value the current timestamp. Finally, just before M terminates, it makes a linear reading scan over every memory location. Unlike all other memory reads by M, the reads during this scan are *not* followed with a matching write operation. M also halts and outputs "reject" if a read ever returns a timestamp greater than current timestamp.

With these modifications, if M does not output "reject" then the set of (memory location, value) pairs returned by all the read operations equals the set of (memory location, value) pairs written during all the write operations if and only if every write operation returns the value that was last written to that location. Clearly these tweaks only increase M's runtime by a constant factor, as the tweak turns each read operation and each write operation of M into both a read and a write operation.

Multiset equality checking (a.k.a. permutation checking) via fingerprinting. Recall that in Section 2.1 we gave a probabilistic procedure called Reed-Solomon fingerprinting for determining whether two vectors a and b are equal entry-by-entry: a is interpreted as specifying the coefficients of a polynomial $p_a(x) = \sum_{i=1}^{n} a_i x^i$ over field $\mathbb{F}$, and similarly for b, and the equality-checking procedure picks a random $r \in \mathbb{F}$ and checks whether $p_a(r) = p_b(r)$. The guarantee of this procedure is that if $a = b$ entry-by-entry, then the equality holds for every possible choice of r, while if a and b disagree in even a single entry i (i.e., $a_i \neq b_i$), then with probability at least $1 - n/|\mathbb{F}|$ over the random choice of r, the equality fails to hold.

To perform memory-checking, we do not want to check equality of vectors, but rather of multisets, and this requires us to tweak to the fingerprinting procedure from Section 2.1. That is, the above reduction from memory-consistency-checking to multiset equality checking produced two lists of (memory location, value) pairs, and we need to determine whether the two lists specify the same *set* of pairs, i.e., whether they are permutations of each other. This is different than determining whether the lists agree entry-by-entry.

To this end, let us interpret each (memory location, value) pair as a field element, via any arbitrary injection of (memory location, value) pairs to $\mathbb{F}$. This does require the field size $|\mathbb{F}|$ to be at least as large as the number of possible (memory location, value) pairs. For example, if the memory has size, say, 2^{64}, and values consist of 64 bits, it is sufficient for $|\mathbb{F}|$ to be at least 2^{128}. Under this interpretation, we can think of our task as follows. We are given two length-m lists of field elements $a = (a_1, \ldots, a_m)$ and $b = (b_1, \ldots, b_m)$, where m is the number of read and write operations performed by the machine M. We want to determine whether the lists a and b are permutations of each other, i.e., whether for every possible field element $z \in \mathbb{F}$, the number of times z appears in list a equals the number of times that z appears in list b.

Here is a randomized algorithm that accomplishes this task. Interpret a as a polynomial p_a whose roots are $a_1, \ldots, a_m$ (with multiplicity), i.e., define

$$p_a(x) := \prod_{i=1}^{m} (a_i - x),$$

and similarly

$$p_b(x) := \prod_{i=1}^{m} (b_i - x).$$

Now evaluate both p_a and p_b at the same randomly chosen input $r \in \mathbb{F}$, and output 1 if and only if the evaluations are equal. Clearly p_a and p_b are the same polynomial if and only if a and b are permutations of each other. Hence, this randomized algorithm satisfies:

- if a and b are permutations of each other then this algorithm outputs 1 with probability 1.

- if a and b are not permutations of each other, then this algorithm outputs 1 with probability at most $m/|\mathbb{F}|$. This is because p_a and p_b are distinct polynomials of degree at most m and hence can agree at at most m inputs (Fact 2.1).

We can think of $p_a(r)$ and $p_b(r)$ as fingerprints of the lists a and b that captures "frequency information" about a and b (i.e., how many times each field element z appears in the two lists), but deliberately ignores the order in which A and B are presented. A key aspect of this fingerprinting procedure is that it lends itself to highly efficient implementation within arithmetic circuits. That is, given as input lists A and B of field elements, along with a field element $r \in \mathbb{F}$, an arithmetic circuit can easily evaluate $p_a(r)$ and $p_b(r)$. For example, computing $p_a(r)$ amounts to subtracting r from each input $a_i \in a$, and then computing the product of the results via a binary tree of multiplication gates. This requires only $O(m)$ gates and logarithmic depth. Hence, this randomized algorithm for permutation checking can be efficiently implemented within the arithmetic circuit $\mathcal{C}$.

Historical Notes and Optimizations. Techniques for memory-consistency-checking closely related to those described above were given in [259] and also exploited in subsequent work [172]. Specifically, [259] checks memory consistency of an execution trace for a RAM within a circuit by exploiting permutation-invariant fingerprinting to check that claimed time-ordered and memory-ordered descriptions of the execution trace are permutations of each other. While the fingerprints can be computed within the circuit with $O(T)$ gates, this does not reduce total circuit size or prover runtime below $O(T \log T)$.[10] This holds for two reasons. First, to compute a satisfying assignment for the circuit constructed in [259], the prover must sort the transcript based on memory location, and this takes $O(T \log T)$ time. Second, there is still a need for the circuit to implement comparison operations on timestamps associated with each memory operation, and [259] uses $\Theta(\log T)$

[10] [259] asserts a prover running in time $O(T)$, but this assertion hides a factor that is linear in the word length of the RAM. [259] considers this to be a constant such as 32 or 64, but in general this word length must be $\Omega(\log T)$ to write down timestamps and index into memory, if the memory has size $\Omega(T)$.

many gates to implement each comparison operation bit-wise inside the circuit-satisfiability instance (see the final paragraph of Section 6.5.4.1).

Both sources of overhead just described were addressed in two works [68], [226]. Setty [226] observes that (as described above in this section), the need for the prover to sort the transcript based on memory location can be avoided by modifying the RAM as per the offline memory checking technique of Blum *et al.* [59]. This does not in general avoid the need to perform comparison operations on timestamps inside the circuit, because the modified RAM constructed by Blum *et al.* [59] requires checking that the timestamp returned by every read operation is smaller than the timestamp at which the read operation occurs. However, there are contexts in which such comparison operations are not necessary (see, e.g., Section 16.2), and this implies $O(T)$-sized circuits in such contexts.[11] Even outside such contexts, work of Bootle *et al.* [68] (which we sketch in Section 6.6.3 below) give a technique for reducing the *amortized* gate-complexity of performing many integer comparison operations inside a circuit over a field prime order. Specifically, they shows how to perform $O(T)$ comparison operations on integers of magnitude $\text{poly}(T)$ using $O(T)$ gates in arithmetic circuits over any prime-order field $\mathbb{F}_p$ of size at least T.[12] In summary, both sources of "superlinearity" in the size of the memory-consistency-checking circuit and prover runtime can be removed using the techniques of [68], [226], reducing both circuit size and prover runtime to $O(T)$.

Setty [226] and Campanelli *et al.* [87] observe that this fingerprinting procedure can be verified efficiently using optimized variants of succinct arguments derived from the GKR protocol [237], [244], because $p_A(r)$ can be computed via a small, low-depth arithmetic circuit with a regular wiring pattern, that simply subtracts r from each input and multiplies the results via a binary tree of multiplication gates. This ensures that

[11] Specifically, if the memory access pattern of the RAM is independent of the input, then the use of timestamps and the need to perform comparisons on them can be eliminated using a pre-processing phase requiring time $O(T)$. See Section 16.2 for details.

[12] The techniques of [68] build on permutation-invariant fingerprinting, and hence are interactive.

the circuit-satisfiability instances resulting from the transformation above can be efficiently verified via such arguments.

Additional Applications of Fingerprinting-based Permutation Checking. The above fingerprinting procedure for checking whether two vectors are permutations of each other has a long history in algorithms and verifiable computing and has been rediscovered many times. It was introduced by Lipton [183] as a hash function that is invariant to permutations of the input, and later applied in the context of interactive and non-interactive proofs with small-space streaming verifiers [92], [184], [220].

Permutation-invariant fingerprinting techniques were also applied to give zero-knowledge arguments that two encrypted vectors are permutations of each other [24], [140], [144], [198]. Such zero-knowledge arguments are also called shuffle arguments, and are directly applicable to construct an anonymous routing primitive called a mix network, a concept introduced by Chaum [94]. The ideas in these works were in turn built upon to yield SNARKs for circuit satisfiability with proofs that consist of a constant number of field or group elements [66], [123], [187]. Roughly speaking, these works use variants of permutation checking to ensure that a purported circuit transcript assigns consistant values to all output wires of each gate, i.e., to confirm that the transcript respects the wiring pattern of the circuit. Other uses of permutation-invariant fingerprinting in the context of zero-knowledge proofs were given in [229].[13]

Additional Discussion. We remark that there are other permutation-invariant fingerprinting algorithms that do *not* lend themselves to efficient implementation within arithmetic circuits, and hence are not useful for transforming an instance of RAM execution to an instance

[13] [229], like earlier work [101], uses a collision-resistant permutation-invariant hash function to check multiset equality, rather than the simple (non-collision-resistant) permutation-invariant fingerprinting function described in this section. Such hash functions are secure against polynomial time cheating provers even when the prover knows the hash function being used in the permutation-checking procedure and can choose the inputs to the procedure.

of arithmetic circuit satisfiability. An instructive example is as follows. Let $\mathbb{F}$ be a field of prime order, and suppose that it is known that all entries of the lists a and b are positive integers with magnitude at most B, where $B \ll |\mathbb{F}|$. Then we can define the polynomial $q_a(x)$ over $\mathbb{F}$ via

$$q_a(x) := \sum_{i=1}^{m} x^{a_i},$$

and similarly

$$q_b(x) := \sum_{i=1}^{m} x^{b_i}.$$

Clearly q_a and q_b are polynomials of degree at most B, and they satisfy properties analogous to p_a and p_b, namely:

- if a and b are permutations of each other then $q_a(r) = q_b(r)$ with probability 1 over a random choice $r \in \mathbb{F}$.
- if a and b are not permutations of each other, then $q_a(r) = q_b(r)$ with probability at most $B/|\mathbb{F}| \ll 1$. This is because q_a and q_b are distinct polynomials of degree at most B and hence can agree at at most B inputs (Fact 2.1).

However, given as input the entries of a and b, interpreted as field elements in $\mathbb{F}$, an arithmetic circuit cannot efficiently evaluate $q_a(r)$ or $q_b(r)$, as this would require raising r to the power of input entries, which is not a low-degree operation.

6.6.3 Efficiently Representing Non-Arithmetic Operations Over Large Prime-Order Fields

Recall from Section 6.5.4.1 that when operating over a field of large prime order p, it is convenient to interpret field elements as integers in $[0, p-1]$ or $[-\lfloor p/2 \rfloor, \lfloor p/2 \rfloor]$, as then integer addition and multiplication corresponds directly to field addition and multiplication, up to overflow issues. This means (again, ignoring overflow issues) integer addition and multiplication operations can be implemented with a *single* gate in the corresponding circuit satisfiability instance.

Non-arithmetic operations on integer values are more challenging to implement inside an arithmetic circuit. Section 6.5.4.1 described a

straightforward approach, which broke field elements into their binary representation, and computed the non-arithmetic operations by operating over these bits. The reason that this bit-decomposition approach is expensive is that it transforms an integer (which for a Random Access Machine M is a primitive data type, consuming just one machine register) into at least $\log_2 p$ field elements, and hence at least $\log_2 p$ gates. In practice, $\log_2 p$ might be roughly 128 or 256, which is a very large constant. In theory, since we would like to be able to represent timestamps via a single field element, we typically think of $\log_2 p$ as at least $\Omega(\log T)$, and hence superconstant. From either perspective, turning a single machine operation such as integer comparison into (at least) 256 gates is painfully expensive.

Ideally, we would like to replace the $\Omega(\log p)$ cost of the bit-decomposition approach to implementing these operations inside a circuit with a constant independent of p. Bootle *et al.* [68] develop techniques for achieving this in an amortized sense. That is, they showed how to simulate non-arithmetic operations over integers (e.g., integer comparisons, range queries, bit-wise operations, etc.) by arithmetic circuit-satisfiability instances working over a field of large prime order. Before providing details, here is the rough idea. The bit-decomposition approach represents integers in base-b for $b = 2$, and this means that logarithmically many field elements are required to represent a single integer. The convenient feature about using base-2 was that it was easy to check that a list of field elements represented a valid base-2 representation; in particular, that every field element in the list was either 0 or 1. This is because the low-degree expression $x \mapsto x^2 - x$ equals 0 if and only if x is in $\{0, 1\}$. Instead, Bootle *et al.* represent integers $y \in [0, 2^W]$ in a far larger base, namely base $b = 2^{W/c}$ for some specified integer constant $c > 1$. This has the benefit that y is represented via only c field elements, rather than W field elements. However, working over such a large base b means that there is no longer a degree-2 polynomial $q(x)$ that evaluates to 0 if and only if x is in $\{0, 1, \ldots, b-1\}$—the lowest-degree polynomial q with this property has degree b. Bootle *et al.* work around this issue by turning the task of checking whether a field element x is in the set $\{0, 1, \ldots, b-1\}$ into a *table lookup*, and then giving an efficient procedure for performing

such lookups inside an arithmetic circuit satisfiability instance. That is, conceptually, they have the circuit initialize a table containing the values $\{0, 1, \ldots, b-1\}$, and then have the witness include a proof that all values appearing in the base-b decomposition of any integer y arising in the computation reside in the table. As we will see, the number of gates required to initialize the table and specify and check the requisite lookup proof is roughly $\tilde{O}(b)$, so a key point is that c will be chosen to be a large enough constant so that b is smaller than the runtime T of the Random Access Machine M whose execution the circuit is simulating. This ensures constant *amortized* cost of all the $O(T)$ decomposition operations that the circuit has to perform. Details follow.

Let 2^W be a bound on the magnitude of integers involved in each non-arithmetic operation (assume 2^W is significantly smaller than the size of the prime order field over which the circuits we generate will be defined), and let T be an upper bound on the number of operations to be simulated. In the context of Section 6.6.2, T is a bound on the runtime of the Random Access Machine, and W is the word-size. This is because, if a register of the RAM contains W bits, then the RAM is incapable of representing integers larger than 2^W without resorting to approximate arithmetic. In this context, one would need to choose W at least as large as $\log_2 T$ to ensure that a timestamp can be stored in one machine word.

As sketched above, Bootle *et al.* effectively reduces each non-arithmetic operation to a lookup into a table of size $2^{W/c}$, where $c \geq 1$ is any integer parameter. For example, if $W = \ell \log_2 T$ for some constant $\ell \geq 1$, then setting $c = \ell/4$ ensures that the lookup table has size at most $T^{1/4}$. The lookup table is initialized to contain a certain set of $2^{W/c}$ pre-determined values (i.e., the values are independent of the input to the computation). In the technique of [68], the length of the witness w of $\mathcal{C}$ grows linearly in c. This is because, in order to keep the table to size $2^{W/c}$, each W-bit word of memory is represented via c field elements. That is, each W-bit word is broken into c blocks of length W/c, ensuring that each block can only take on $2^{W/c}$ possible values. This means that if a transcript for a time-T computation consists of, say, $k \cdot T$ words of memory—because each time step of the transcript

requires specifying k register values—the transcript will be represented by $k \cdot c \cdot T$ field elements in the witness for $\mathcal{C}$.

Before describing the reduction of Bootle *et al.* from non-arithmetic operations to lookups in a pre-determined table, we explain how to efficiently verify a long sequence of lookup operations.

Checking many lookup operations efficiently. Bootle *et al.* develop a technique for checking that many values all reside in the lookup table. The technique builds on the permutation-invariant fingerprinting function of Section 6.6.2. Specifically, to show that a sequence of values $\{f_1, \ldots, f_N\}$ only contains elements from a lookup table containing values $\{s_1, \ldots, s_B\}$ where $B \leq 2^{W/c}$ is the size of the lookup table, it is enough to show that there are non-negative integers $e_1, \ldots, e_B$ such that the polynomials $h(X) \coloneqq \prod_{i=1}^{N}(X - f_i)$ and $q(X) \coloneqq \prod_{i=1}^{B}(X - s_i)^{e_i}$ are the same polynomial. To establish this, the witness will specify the bit-representation of the exponents $e_1, \ldots, e_B$ (each $e_i \in \{0,1\}^{\log_2 N}$), and the circuit confirms that $h(r) = q(r)$ for an $r \in \mathbb{F}_p$ randomly chosen by the verifier after the prover commits to the witness. As usual, Fact 2.1 implies that if this check passes then up to soundness error N/p, h and q are the same polynomial. A crucial fact that enables the circuit to efficiently implement this check is that $q(r)$ can be computed by an arithmetic circuit using $O(B \log(N))$ gates, as

$$\prod_{i=1}^{B} \prod_{j=1}^{\log_2 N} (r - s_i)^{2^j \cdot e_{i,j}}.$$

In summary, this lookup table technique permits Bootle *et al.* to implement a sequence of $O(N)$ non-arithmetic operations inside an arithmetic circuit-satisfiability instance using just $O(N + B \log N)$ gates. So long as $N = O(T)$ and $B = 2^{W/c} \leq N/\log N$, this is $O(T)$ operations in total.

Gabizon and Williamson [122] describe a variant transformation they call *plookup* that reduces the number of gates to $O(N)$, which is an improvement by a logarithmic factor if $B = \Theta(N)$. To give an idea of how *plookup* works, we sketch a simplified variant due to Cairo that works under two assumptions: first, that each value s_i appearing in the lookup table appears at least once in the sequence $\{f_1, \ldots, f_N\}$, and

second that that the elements $\{s_1, \ldots, s_B\}$ cover a contiguous interval such as $\{1, 2, \ldots, B\}$, i.e., $s_i = s_1 + i - 1$ for all $i = 1, \ldots, B$.

Under these assumptions, the witness can simply consist of a sequence $\{w_1, \ldots, w_N\}$ of field elements claimed to equal $\{f_1, \ldots, f_N\}$ in sorted order, i.e., such that:

- $\{w_1, \ldots, w_N\}$ is a permutation of $\{f_1, \ldots, f_N\}$.
- When $w_1, \ldots, w_N$ are interpreted as integers,

$$s_1 = w_1 \leq w_2 \leq \ldots \leq w_N = s_B. \tag{6.1}$$

The circuit can apply permutation-invariant fingerprinting to confirm (with overwhelming probability) that the first bullet point above holds. To confirm that Equation (6.1) holds, the circuit checks that the following equalities hold:

- $w_1 = s_1$.
- $w_N = s_B$.
- For each $i = 2, \ldots, N$, $(w_i - w_{i-1}) \cdot (w_i - (w_{i-1} + 1)) = 0$.

Here, the constraints captured in the final bullet point ensure that as i ranges from 1 up to N, the w_i values start at s_1 and proceed to s_B in a non-decreasing manner. Under the two assumptions made above, this is equivalent to checking that for each $i > 1$, either $w_i = w_{i-1}$ or $w_i = w_{i-1} + 1$, which is exactly what is captured by the quadratic constraint in the final bullet point.

Reducing non-arithmetic operations to lookups. To give a sense of the main ideas of the reduction of [68], we sketch the reduction in the context of two specific non-arithmetic operations: range proofs and integer comparisons.

For simplicity, let us assume that $c = 2$. To confirm that a field element v is in the range $[0, 2^W]$, one can have the witness specify v's unique representation as a pair of field elements (a, b) such that $v = 2^{W/2} \cdot a + b$ and $a, b \in \{0, \ldots, 2^{W/2} - 1\}$. The circuit then just checks that indeed $v = 2^{W/2} \cdot a + b$ and that a and b both reside in a lookup

table of size $2^{W/2}$ initialized to store all field elements y between 0 and $2^{W/2} - 1$.

As another example, doing an integer comparison reduces to a range proof. Indeed, to prove that $a > c$ when a and c are guaranteed to be in $[0, 2^W]$, it is enough to show that the difference $a - c$ is positive, which is a range proof described above, albeit under the weaker guarantee that the input $v = a - c$ to the range proof is in $[-2^W, 2^W]$ rather than $[0, 2^W]$.

6.6.4 CPU-Like vs. ASIC-Like Program-to-Circuit Transformations

This section described frontends that produce circuit-satisfiability instances that essentially execute step-by-step some simple CPU. The idea is that frontend designers will specify a set of "primitive operations" (also known as an *instruction set*) analogous to assembly instructions for real computer processors. Developers who want to use the frontend will either write "witness-checking programs" directly in the assembly language or else in some higher-level language, and have their programs automatically compiled into assembly code and then transformed into an equivalent circuit-satisfiability instance by the front-end.

At the time of writing, several prominent projects are taking this CPU-oriented approach to front-end design. For example, StarkWare's Cairo [128] is a very limited assembly language in which assembly instructions roughly permit addition and multiplication over a finite field, function calls, and reads and writes to an immutable (i.e., write-once) memory. The Cairo CPU is a von Neumann architecture, meaning that the circuits produced by the frontend essentially take a Cairo program as public input and "run" the program on the witness. The Cairo language is Turing Complete—despite its limited instruction set, it can simulate more standard architectures, although doing so may be expensive. Another example project is called RISC-Zero.[14] which targets a CPU called the so-called *RISC-V architecture*,[15] an open-source architecture with a rich software ecosystem that is growing in popularity.

[14] https://github.com/risc0/risc0.
[15] https://riscv.org/.

For sufficiently simple instruction sets, the front-end techniques described in this section produce circuits over fields of large prime order, with $O(T)$ gates, where T is the runtime of the CPU whose execution we wish to verify. This is clearly optimal up to a constant factor. Moreover, it is possible to ensure that the wiring of the circuit is sufficiently regular that the verifier in argument systems derived from (for example) the GKR protocol can run in time polylogarithmic in T, i.e., the verifier need not materialize the entire circuit itself. However, these transformations can still be expensive in practice.

"CPU emulator" projects such as RISC-Zero and Cairo produce a single circuit that can handle all programs in the associated assembly language. Alternative approaches are "ASIC-like," producing different circuits for different programs [79], [243], [259]. This ASIC-like approach can yield smaller circuits for some programs, especially when the assembly instruction that the program executes at each timestep does not depend on the program's input. For example, it can potentially avoid frontend overhead entirely for straight-line programs such as naive matrix multiplication (Figure 6.1). But the ASIC approach may be limited; for example, at the time of writing, it's not known how to use it to support loops without predetermined iteration bounds. It seems likely that additional progress will be made to improve the generality of ASIC-like approaches, as well as the efficiency of CPU emulator approaches.

6.7 Exercises

Exercise 6.1. Describe a layered arithmetic circuit of fan-in three that takes as input a matrix $A \in \{0,1\}^{n \times n}$, interprets A as the adjacency matrix of a graph G, and outputs the number of triangles in G. You may assume that n is a power of three.

Exercise 6.2. Describe a layered arithmetic circuit of fan-in two that, given as input an $n \times n$ matrix A with entries from some field $\mathbb{F}$, computes $\sum_{i,j,k,\ell \in \{1,\ldots,n\}} A_{i,j} \cdot A_{k,\ell}$. The smaller your circuit is, the better.

Exercise 6.3. Fix an integer $k > 0$. Assume that k is a power 2 and let $p > k$ be a large prime number. Describe an arithmetic circuit of

fan-in 2 that takes as input n elements of the field $\mathbb{F}_p$, $a_1, a_2, \ldots, a_n$, and outputs the n field elements $a_1^k, a_2^k, \ldots, a_n^k$.

What is the verifier's asymptotic runtime when the GKR protocol is applied to this circuit (express your answer in terms of k and n)? Would the verifier be interested in using this protocol if n is very small (say, if $n = 1$)? What if n is very large?

Exercise 6.4. Let $p > 2$ be prime. Draw an arithmetic circuit $\mathcal{C}$ over $\mathbb{F}_p$ that takes as input one field element $b \in \mathbb{F}_p$ and evaluates to 0 if and only if $b \in \{0, 1\}$.

Exercise 6.5. Let $p = 11$. Draw an arithmetic circuit $\mathcal{C}$ over $\mathbb{F}_p$ that takes as input one field element a followed by four field elements b_0, b_1, b_2, b_3, and such that all output gates of $\mathcal{C}$ evaluate to 0 if and only if (b_0, b_1, b_2, b_3) is the binary representation of a. That is, $b_i \in \{0, 1\}$ for $i = 1, \ldots, 4$, and $a = \sum_{i=0}^{3} b_i \cdot 2^i$.

Exercise 6.6. Let $p = 11$. Let $x = (a, b)$ consist of two elements of the field $\mathbb{F}_p$. Draw an arithmetic circuit satisfiability instance that is equivalent to the conditional $a \geq b$. That is, interpreting a and b as integers in $\{0, 1, \ldots, p-1\}$, the following two properties should hold:

- $a \geq b \implies$ there exists a witness w such that evaluating $\mathcal{C}$ on input (x, w) produces the all-zeros output.
- $a < b \implies$ there does not exist a witness w such that evaluating $\mathcal{C}$ on input (x, w) produces the all-zeros output.

Additional Exercises. The interested reader can find a sequence of additional exercises on front ends at https://www.pepper-project.org/tutorials/t3-biu-mw.pdf. These exercises discuss transforming computer programs into equivalent R1CS-satisfiability instances, a generalization of arithmetic circuit-satisfiability that we discuss further in Section 8.4.