Efficient Representation of Numerical Optimization Problems for SNARKs (extended version)

Sebastian Angel[⋆†] Andrew J. Blumberg[‡] Eleftherios Ioannidis[⋆] Jess Woods[⋆]
[⋆]*University of Pennsylvania* [†]*Microsoft Research* [‡]*Columbia University*

Abstract. This paper introduces Otti, a general-purpose compiler for (zk)SNARKs that provides support for numerical optimization problems. Otti produces efficient arithmetizations of programs that contain optimization problems including linear programming (LP), semi-definite programming (SDP), and a broad class of stochastic gradient descent (SGD) instances. Numerical optimization is a fundamental algorithmic building block: applications include scheduling and resource allocation tasks, approximations to NP-hard problems, and training of neural networks. Otti takes as input arbitrary programs written in a subset of C that contain optimization problems specified via an easy-to-use API. Otti then automatically produces rank-1 constraint satisfiability (R1CS) instances that express a succinct transformation of those programs. Correct execution of the transformed program implies the optimality of the solution to the original optimization problem. Our evaluation on real benchmarks shows that Otti, instantiated with the Spartan proof system, can prove the optimality of solutions in zero-knowledge in as little as 100 ms—over 4 orders of magnitude faster than existing approaches.

1 Introduction

Optimization problems are pervasive in science, government, business, and academia. Convex optimization in the form of linear programming (LP) and semidefinite programming (SDP) is widespread, and the rise of deep learning has put particular emphasis on stochastic gradient descent (SGD). Example applications include resource allocation problems, approximation of NP-hard problems, training of machine learning models, and others. Efficiently producing solutions to these problems is the subject of intensive study. However, there is much less focus on providing transparency about the nature of the solution process or the quality of the resulting answer. Today, a *solver* (e.g., a government agency like FEMA) publishes the purported optimal solution to an optimization problem (e.g., the optimal allocation of medications to shelters) and asks *clients* (parties interested or affected by the result) to trust the solution. While clients could in principle rederive the optimal solution on their own, in many applications the inputs to the optimization problem are sensitive and cannot be shared (for example, due to security clearances, personal identifiable information, or business secrets). This complicates accountability and transparency.

Existing systems for zero-knowledge succinct non-interactive arguments of knowledge (zkSNARKs) [14–16, 18, 19, 26–28, 38, 40, 48, 54, 63, 65, 70, 73, 75] offer an attractive way to bring accountability and transparency into an otherwise opaque process. Assuming that the inputs can be made available in the form of cryptographic commitments by some means, the solver can generate a cryptographic proof that convinces clients that the solution is optimal without revealing the sensitive inputs. Concrete examples of where this situation might be applicable include:

- A school commits to secret admissions criteria. Students submit application materials (test scores, etc.) and the school then proves, in zero-knowledge, that admissions were the optimal solution to the optimization problem.
- A hospital commits to a donor list. The hospital then proves in zero-knowledge to a prospective donor, who is a partner of someone in need of an organ (but not a match), that if they donate their kidney their partner can receive a compatible organ (this is called a kidney chain [12]).
- An automobile company commits to a data set. The company proves to regulators safety properties about the data in zero-knowledge (that it has adequate sample diversity, light conditions, etc.). Then, the automobile company can prove (again in zero-knowledge) that the ML model incorporated in a car was trained with the committed data.

The above examples are predicated on existing zkSNARKs being able to describe facts about optimization problems. Unfortunately, existing mechanisms for expressing optimization problems in a format amenable for zkSNARKs (e.g., arithmetic or boolean circuits, rank-1 constraint satisfiability systems) result in prohibitive costs. As a concrete example, representing an LP instance with 3 variables and 3 linear equations using a state-of-the-art compiler for SNARKs [52] results in over 1 million multiplication gates. Worse yet, we are unable to compile much larger instances due to the radical blow up in the number of constraints and the high memory burden this places on the compiler. This massive expressivity cost comes from a few sources: (1) the need to support random memory accesses; (2) the need for fixed point or double precision to approximate real numbers; (3) the need to upper bound the number of loop iterations required to find an optimal solution for any possible inputs; and (4) the need to express the complex logic of the solver.

To address these issues, we introduce Otti, a compiler for zkSNARKs that takes as input programs written in C that include optimization instances and outputs rank-1 constraint satisfiability instances (R1CS) for these programs that are succinct and efficient to prove and verify. Otti works as follows:

Exploit nondeterministic checkers. Otti uses the observation that for a prover to convince a verifier that it knows the output of some program, the prover does not actually need to *run* the program at all, much less prove that it ran the program correctly. Instead, the prover just needs to prove that the output of the program is *correct*. For example, the prover does not need to prove that a list was sorted with Quicksort, but simply that the list is in sorted order. Therefore, in such examples the prover can prove to the verifier that a purported output of the program is correct—this is equivalent to running the program and obtaining the solution. A *nondeterministic checker* is a special program that does this: the nondeterministic checker is derived from the original program and takes the solution to the original program as a nondeterministic input (how the prover gets this solution is irrelevant). The prover then confirms that the solution is correct (i.e., passes the checker) and proves this fact to the verifier, via a zkSNARK.

Build nondeterministic checkers from certificates. Otti uses the certificates of optimality which are available for many optimization problems [47] to build nondeterministic checkers. Validating these certificates is equivalent to verifying that the solution to the optimization problem is optimal. Crucially, the nondeterministic checkers produced by this approach are radically more efficient than the original programs: they reduce (and often eliminate) the need for random memory accesses, eliminate the need to upper bound loop iterations, and avoid representing the logic of the solver itself.

Probabilistic certificates of optimality. Many important classes of optimization problems (e.g., instances where SGD is used) lack deterministic certificates of optimality. In these cases, Otti introduces the notion of *probabilistic certificates of optimality* (PCO). PCOs have the same benefits as standard certificates but have a small soundness error. We show how to apply PCOs to some instances of SGD to construct efficient nondeterministic checkers. Our approach here is guided by a new conjecture that asserts that whenever there is a rapid convergence result for SGD, one can extract a PCO and therefore also a nondeterministic checker.

Automatic generation of checkers. Otti takes as input C programs extended with an API for optimization problems (e.g., Otti's API for LP is inspired by Google's Glop linear solver [6]). Otti then automatically extracts and compiles the nondeterministic checker for the optimization program defined with this API. This allows developers to be oblivious to the notions of certificates of optimality, since their theory can be quite complex for problems beyond LP.

Leverage numerical optimization solvers. Otti leverages existing fast numerical solvers (e.g., lpsolve [7]) to find the solution to the underlying optimization problems in the provided C programs (and our API) and supply these solutions to the nondeterministic checkers that Otti generates.

We have implemented a prototype of Otti on top of the CirC SNARK compiler [52] and have compiled a variety of real-world benchmarks typically used within the optimization community to measure the performance of commercial solvers. Otti can generate proofs for the R1CS corresponding to these benchmarks using the Spartan zkSNARK [63] in times ranging from 100 ms to 19 sec (for thousands to around seven million constraints). On average, proof generation for LP and SDP is 30–40$\times$ more expensive than finding the solutions themselves using existing solvers. For SGD, proof generation is on average 80$\times$ more expensive, and up to two orders of magnitude more expensive for the worst performing dataset. This constitutes a significant improvement over prior work which produces R1CS statements that take over 4 orders of magnitude longer to prove for LP statements and cannot compile any of the SDP or SGD problems.

2 Background

In this section we briefly review *zero-knowledge succinct non-interactive argument of knowledge* (zkSNARKs), give background on how computations in zkSNARKs are typically represented, and then discuss the notion of nondeterministic checkers. Note that this paper does not introduce a new zkSNARK, nor does it make any changes to existing ones. Instead, Otti's contributions are on representing numerical optimization problems in a way that is more efficient for *existing* zkSNARKs. We therefore only discuss the properties of zkSNARKs rather than the details of how they work.

2.1 zkSNARKs

A zkSNARK is a cryptographic protocol that allows a prover to convince a verifier that it has knowledge of a satisfying witness to an NP statement – without revealing any information beyond what is implied by the validity of the statement. A common choice for zkSNARKs is to target the NP complete problem of *rank-1 constraint satisfiability* (R1CS) since any nondeterministic random access machine running in a fixed number of times steps can be transformed into an R1CS instance. We discuss R1CS in the next section, but informally, zkSNARKs have the following properties:

1. **Succinct:** The size of the proof and its verification should be sublinear (ideally polylog) in the size of the statement.
2. **Non-interactive:** No interaction is required between the prover and verifier besides the transferring of any verifier inputs and the computation's output and proof.
3. **Argument of knowledge:** The prover must convince the verifier that it knows a witness that satisfies the R1CS instance. This argument is complete and sound.
    - *Completeness:* An honest prover who knows a witness that satisfies the R1CS instance can always generate a proof that convinces the verifier of this fact.
    - *Computational Soundness:* A malicious prover cannot fool the verifier into accepting an invalid proof, except with negligible probability.

4. **Zero-knowledge:** The proof reveals no information to the verifier beyond the fact that the prover knows a witness that satisfies the R1CS instance.

2.2 Rank-1 constraint satisfiability

An R1CS instance is a tuple $(\mathbb{F}, A, B, C, io, m)$, where $\mathbb{F}$ is a finite field, io is the public input and output of the instance, $A, B, C \in \mathbb{F}^{m\times m}$ are square matrices, and $m \geq |io| + 1$. This instance is satisfiable if and only if there exists a witness $w \in \mathbb{F}^{m-|io|-1}$ that makes up a solution vector $z = (io, 1, w)$ such that $(A \cdot \vec{z}) \circ (B \cdot \vec{z}) = (C \cdot \vec{z})$, where $\cdot$ is the matrix-vector product and $\circ$ is the Hadamard product. The entry of z fixed at 1 enables the encoding of constants.

Since matrix entries can be used to encode both addition and multiplication gates over $\mathbb{F}$, R1CS generalizes arithmetic circuit satisfiability. As we show in the next section, one can "compile" a program written in a high level language like C into R1CS, such that the R1CS instance is satisfiable if and only if the output of the R1CS instance (part of io) is the result of correctly evaluating the program on the public inputs.

2.3 Compiling programs to R1CS

Given a program written in a high-level language like C, how does one convert it to R1CS? There are many "frontend" arithmetizing compilers [16, 17, 24, 28, 34, 42–44, 52, 64, 67, 69, 72] written to handle this conversion and even optimize the generated satisfiability instance (i.e., minimize the number of constraints). Over time, these compilers have added support for an increasing number of programming language features like control flow, random-access memory (RAM), bounded loops, algebraic datatypes, and more. Additionally, some compilers can optimize R1CS representations using classical compilation techniques like constant folding or loop flattening and new methods like range proofs.

Let's take for example the following C program:

```c
int foo(int a) {
    int prod = 1;
    int i;
    for(i = 0; i < 3; i++) {
        prod *= a;
    }
    prod += i;
    int r = 30 / prod;
    return r;
}
```

This program, `foo`, takes an integer a as input. A compiler may start by unrolling the bounded loop into a sequence of assignments. This operation introduces versioning for variables, denoted by a subscript, a form otherwise known as single-static assignment (SSA) [59]. SSA allows the expression of mutable computations into immutable equations between versions of variables. The result may look like:

$$\begin{aligned}
&prod_0 = 1,\ i_0 = 0\\
&prod_1 = prod_0 \times a,\ i_1 = 1\\
&prod_2 = prod_1 \times a,\ i_2 = 2\\
&prod_3 = prod_2 \times a,\ i_3 = 3\\
&prod_4 = prod_3 + i_3\\
&r = 30/prod_4
\end{aligned}$$

For the sake of simplicity we will omit the transformation between the C `int` type and the elements in R1CS which are in the finite field $\mathbb{F}_p$. In this particular example we will treat them the same. The values of $prod_0, i_0, i_1, i_2, i_3$ are actually constants, so R1CS compilers can use standard techniques like constant propagation and algebraic identities to eliminate unnecessary constraints. The resulting constraints are:

$$\begin{aligned}
&prod_2 = a \times a\\
&prod_3 = prod_2 \times a\\
&prod_4 = prod_3 + 3\\
&r = 30/prod_4
\end{aligned}$$

With the exception of $r = 30/prod_4$, these equations can be represented in the form of matrices $(A \cdot \vec{z}) \circ (B \cdot \vec{z}) = (C \cdot \vec{z})$, as we discuss in Section 2.5. To make $r = 30/prod_4$ fit our desired form, we can leverage Fermat's little theorem. Since $x^{p-2} \cdot x \equiv x^{p-1} \equiv 1 \pmod{p}$, we can give the expression:

$$\begin{aligned}
&inv = (prod_4)^{p-2}\\
&r = 30 \times inv
\end{aligned}$$

This allows us to represent inv in $\log(p)$ R1CS constraints. However, there is a cheaper way to express the inverse if one leverages the non-determinism supported by R1CS.

2.4 The benefits of non-determinism

We review the notion of a *nondeterministic checker*, which we will employ as a drop-in replacement for a computation that is not efficiently represented in R1CS. Rather than expressing a computation directly, we imagine we are given the result of the computation and merely check that this result is correct. This transformation makes sense when checking a solution is more efficient (concretely) than computing it.

Some basic nondeterministic checkers appear in existing SNARK compilers under the term *exogenous computations*. They are used for expressing things like bit decomposition (crucial for performing bitwise operations) [66], matrix multiplication [71, 76], RAM and remote storage via hash functions [24], among other generic and simple constructs. In Otti, we go a step further and take advantage of the full expressivity and power of nondeterministic checkers by extracting the important properties of optimization programs and checking these properties. To fully characterize nondeterministic checkers, we give a formal definition below.

Nondeterministic checkers. A *partial function* from set X to set Y is a function from a subset of X to Y. A *total predicate* is a total function with a codomain of $\{0,1\}$. The relation between a computation $\mathcal{C}$ and a nondeterministic check $\mathcal{V}$ can be represented as a partial function $\mathcal{C}(X) \mapsto Y$ and a total predicate $\mathcal{V} : X \times Y \to \{0,1\}$, such that $\forall x \in X$ and $y \in Y$, $\mathcal{C}(x) = y \Leftrightarrow \mathcal{V}(x,y) = 1$. Conversely, both termination and non-termination with $\mathcal{C}(x) \neq y$ correspond to $\mathcal{V}(x,y) = 0$. The equivalence above implies the existence of a trivial $\mathcal{V}$ for any $\mathcal{C}$ which simply recomputes $\mathcal{C}$:

$$\mathcal{V}(x,y) := \begin{cases} 1 & \text{if } \mathcal{C}(x) = y, \\ 0 & \text{otherwise} \end{cases}$$

Unlike the trivial $\mathcal{V}$, for many computations it is actually possible to get a considerable improvement in resource use between computing $\mathcal{C}$ versus $\mathcal{V}$, either in asymptotic terms or in absolute terms. We demonstrate the benefits of nondeterminism with the example from the previous section. We define $\mathcal{C}(x) = 30/x$ and its nondeterministic check as:

$$\mathcal{V}(x,y) := \begin{cases} 1 & \text{if } x \times y = 30, \\ 0 & \text{otherwise} \end{cases}$$

We can prove that $\mathcal{C}(x) = y \Leftrightarrow \mathcal{V}(x,y) = 1$ always holds, including the case where $x = 0$, since $\mathcal{V}(0,y)$ can never be 1, as that would imply $\exists y$ such that $0 \times y = 30$, a contradiction.

We use this nondeterministic check to replace the equation $r = 30/prod_4$ from the last example with a free variable r and the equation $r \times prod_4 = 30$. This single multiplication is one R1CS constraint, a considerable improvement from the $\log(p)$ constraints generated by Fermat's little theorem.

2.5 Matrix representation

We now discuss how to turn our equations into the matrix format described in Section 2.2, and which serves as the input to many zkSNARKs. We first reformat the equations, so each corresponds to one row of the matrices:

$$\begin{aligned} &a \times a = prod_2 \\ &prod_2 \times a = prod_3 \\ &r \times (prod_3 + 3) = 30 \end{aligned}$$

We can see that $prod_4 = prod_3 + 3$ gets "wrapped into" a multiplication constraint. For this reason, addition and multiplication by a constant are basically free in R1CS. Multiplications of two variables are a single constraint.

The solution vector $\vec{z}$ will be of the form $(io, 1, w) = (a, r, 1, prod_2, prod_3)$, where a is the public input, 1 is used to encode constants, and the tuple $(prod_2, prod_3, r)$ is the witness. We create the corresponding R1CS matrices A, B, C:

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{pmatrix} \cdot \vec{z} \circ \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 3 & 0 & 1 \end{pmatrix} \cdot \vec{z} = \begin{pmatrix} 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 30 & 0 & 0 \end{pmatrix} \cdot \vec{z}$$

A vector $\vec{z} = (3, 1, 1, 9, 27)$ satisfies this R1CS instance.

3 Numerical optimization problems

Otti's focus is on producing efficient R1CS for *numerical optimization problems*. Optimization problems aim to minimize or maximize an objective function $f : \mathbb{R}^n \to \mathbb{R}$ by choosing the best available inputs according to some set of constraints, $\{g_i\}$ and $\{h_i\}$. We are concerned primarily with convex optimization problems, where the objective function is a convex function and the feasible region of inputs is convex. Optimization problems are described by a standard form:

$$\begin{aligned} &\text{maximize } f(x) \\ &\text{subject to } g_i(x) \leq 0 \\ &\qquad\qquad h_i(x) = 0 \end{aligned}$$

Standard convex optimization problems that have efficient solvers include problems where the objective and constraints fit the frameworks of linear programming (LP) and semidefinite programming (SDP). More generally, a wide class of convex problems can be efficiently solved using variants of gradient descent; this is a generic framework that requires smoothness hypotheses on the objective function but does not make assumptions about the form of f, $\{g_i\}$, and $\{h_i\}$. Gradient descent is very general and does not in fact require convexity, only enough smoothness in the objective function to calculate and numerically approximate gradient vectors.

3.1 Applications

Optimization problems are ubiquitous and there are many applications critical to business, government, and academia. Some real-world examples that Otti can handle are as follows.

- Product mix: Optimize the mix of different types of transportation (e.g., bus, plane) to minimize travel time to some destination. This can be phrased as an LP problem.
- Stocks or marketing: Determine the allocation of money to stocks or ad campaigns to maximize return or clicks over a 2 year period. This can be phrased as an LP problem.
- Scheduling: Find the optimal schedule to run tasks in a real-time system subject to a variety of time and space constraints. This can be phrased as an LP problem.
- Matrix completion: Suppose a 2D picture is given that has a lot of missing pixels. A technique known as matrix completion can be used to find values for these pixels that minimizes an important metric (nuclear norm). This can be phrased as an SDP problem.
- Circuit manufacturing: Find the minimum amount of area needed in a resistor-capacitor (RC) circuit to support a given signal propagation delay. Similarly, find the minimum power dissipation of an RC circuit subject to a given propagation delay. These can be phrased as SDP problems.
- Machine learning: Gradient descent is widely used to train the parameters for machine learning models, including neural networks, SVMs, and logistic regression.

3.2 Challenges

Expressing existing optimization problems in R1CS is difficult to do efficiently owing to the demanding features required by optimization solvers. As a concrete example, consider the Simplex algorithm for solving LP instances, which is by far the simplest among the solvers we surveyed. Below we highlight the major sources of complexity and overhead, which materialize in some form in all existing optimization solvers.

Loops. Simplex uses unbounded while loops; exiting out of some loops is dependent on a variable known only at runtime. For example: `while (lowest >= 0);` where `lowest` is either a public input or a value provided by the prover exogenously (§2.4). To compile such data-dependent loops into R1CS, a compiler must choose some large upper bound and compile that many iterations, regardless of how many are actually needed for a given instance.

RAM. Simplex performs random memory accesses. This commonly occurs with arrays—in Simplex, the statement `if (tableau[pivot_row][pivot_col] > 0);` has the variables `pivot_row` and `pivot_col`, which are not known until runtime. A compiler must therefore have a way to express random access memory. Prior compilers do this with the use of Merkle hash trees [24], hash sets [64], accumulators [53], or sorting networks [16, 69]. In all cases, these technique increase the size of the R1CS instance by orders of magnitude over computations that perform no random accesses.

Real numbers. Simplex uses real numbers. Since these numbers must be represented as field elements in R1CS, an appropriate encoding must be represented as well. This means extra constraints for handling arithmetic operations, boolean operations, and casting. This can make a single variable assignment like `double a = b * c;` into hundreds of constraints.

Missing features. SNARK compilers support only a subset of the functionality of traditional languages. Meanwhile, production solvers use vectorized instructions, GPU extensions, and external libraries. A developer who wishes to use SNARK compilers is forced to write their code in the accepted subset of the language, and lose the efficiency of existing solvers.

4 Overview of Otti

Otti responds to the aforementioned challenges with the following idea: instead of compiling an unoptimized solver that lacks many features, Otti continues to use a state-of-the-art optimization solver and instead compiles into R1CS a nondeterministic checker that confirms the optimality of the solution produced by the solver. Crucially, we show how to automatically derive this checker, and how to avoid RAM accesses and unbounded loops so that it is efficient.

In the next sections we discuss Otti's components. Figure 1 gives an overview of the high-level workflow. We start with a numerical optimization instance that represents a real-world problem. Otti asks developers to write the optimization problem using a simple API. For example, Otti's LP API is inspired by Google's glop [6]. In addition, we have implemented parsers for the most common file formats used to specify these problems: for LP, Otti can parse MPS files [3]; for SDP, Otti can parse SDPA files [36]; for SGD, Otti can parse PMLB files [51, 58]. Unlike the first two, SGD is so general that there is no standard file format available but Otti could easily be extended to support other formats. A key benefit of supporting these file formats is that Otti can run existing benchmarks without modification.

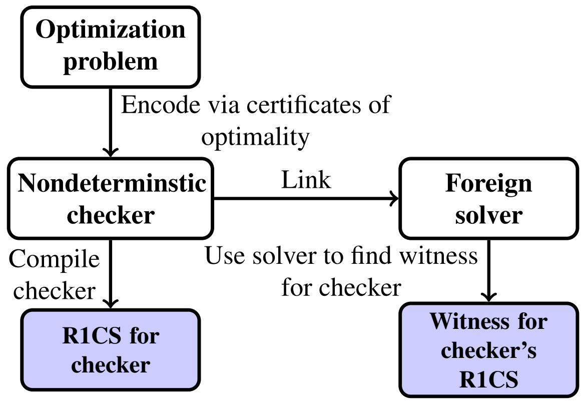
Optimization problem
Encode via certificates of optimality
Nondeterminstic checker
Link
Foreign solver
Compile checker
Use solver to find witness for checker
R1CS for checker
Witness for checker's R1CS

FIGURE 1—High-level workflow of Otti.

Once the optimization problem has been parsed or specified in C with our API, Otti exploits the notion of *certificates of optimality and infeasibility* (detailed in Section 5) to automatically construct a nondeterministic checker, which verifies the optimality of a purported solution to the problem instance.

Otti then compiles the nondeterministic checker to R1CS using a heavily modified version of the CirC SNARK compiler [52]. Otti also compiles the original optimization instance (given in the C code) to a format that is compatible with an existing numerical solver. For example, for LP, Otti compiles the original optimization instance so that it can be consumed by `lpsolve`. In short, Otti produces 2 outputs: (i) R1CS of the nondeterministic checker for the optimization problem; (ii) a fully instantiated solver that the prover can use to solve the problem (once all inputs are known) and obtain the witness for the R1CS instance. Note that if the program has code besides the optimization instance (e.g., code that generates inputs to or consumes the outputs of the optimization instance), Otti generates R1CS for those operations as well and uses an SMT solver to find the satisfying assignment for those constraints (similarly to CirC).

At the end of compilation and solving, Otti outputs a *zkInterface* [10] file (which is nowadays a standard data format for R1CS and witnesses). Otti can then use any of the (many) zkSNARK implementations that currently support this data format to produce commitments to the witness and the corresponding proofs. Otti uses Spartan [63], which is open source, has a fast prover, and does not need a trusted setup; but Otti can also use any of the others.

5 Optimization certificates

We begin by explaining the certificates for the LP and SDP convex optimization frameworks. These problems can be transformed from the *primal* formulation to the *dual* formulation, which is also a convex optimization problem.

We describe the primal problem's optimal solution by vector $\mathbf{x}$, the dual problem's optimal solution by vector $\mathbf{y}$, and define the notion of *weak duality* as: $f_D(\mathbf{y}) \geq f_P(\mathbf{x})$. Here, f_P and f_D are the objective functions for the primal and dual problems, which produce a single maximum or minimum, respectively. (Note that one can also formulate the primal as a min problem and the dual as a max problem and then invert the duality inequality.) The difference between the optimal dual and primal solutions, $f_D(\mathbf{y}) - f_P(\mathbf{x})$, is referred to as the *duality gap*. *Strong duality* occurs when the duality gap is 0.

Strong duality is a powerful condition because it provides a *certificate of optimality*: given a solution to the primal and a solution to the dual, we can check to see if the duality gap is 0. If so, then the solution is optimal [47].

For SGD problems, we cannot use duality to generate certificates. Instead, in Section 5.3 we introduce an entirely new type of certificate that exploits the fact that in many cases of interest there are bounds on the behavior of the stochastic estimates of the gradient of the objective function near the optimal point. By leveraging these bounds, we can craft *probabilistic certificates of optimality*: if the certificate check passes, then the solution is optimal with high probability.

Preview: How Otti exploits Certificates of Optimality. Otti's insight is to leverage certificates of optimality to avoid representing the solver's logic while still allowing the prover to prove that the solution to an optimization problem is optimal. This is done as follows. For LP and SDP, the prover finds the primal and dual solutions to the optimization problem using existing numerical solvers. The prover then proves that the primal solution is a feasible solution to the primal problem (that the point falls inside the feasible region), that the dual solution is a feasible solution to the dual problem, and that the duality gap is zero. For SGD, the prover finds the optimal point z using an existing solver and then proves that at z the gradient estimates $\nabla f_i(z)$ (or a representative sampling of them) have some property (e.g., are close to zero).

Generating proofs of these facts is significantly cheaper than proving the execution of the solver itself because they: (1) do not require RAM, (2) do not require loops, so there is no need to upper bound loop bounds, and (3) do not require as many expensive arithmetic operations over real values. In other words, they illustrate the power of nondeterministic checkers (§2.4), whereby verifying a key property of the solution ($\mathcal{V}$) is cheaper and yet equally as convincing to a verifier as running the computation itself ($\mathcal{C}$).

Below we discuss the theory behind this approach in more detail and give examples with code snippets in Section 6.

5.1 Linear Programming

Linear programming (LP) is a class of convex optimization problems where the objective and constraint functions are linear. LP problems have a primal form:

$$\begin{aligned} \text{find a vector } \mathbf{x} \text{ that maximizes } & \mathbf{c}^T\mathbf{x} \\ \text{subject to } & A\mathbf{x} \leq \mathbf{b} \\ & \mathbf{x} \geq 0 \end{aligned}$$

The corresponding dual is:

$$\begin{aligned} \text{find a vector } \mathbf{y} \text{ that minimizes } & \mathbf{b}^T\mathbf{y} \\ \text{subject to } & A^T\mathbf{y} \geq \mathbf{c} \\ & \mathbf{y} \geq 0 \end{aligned}$$

Here $\mathbf{x}$ and $\mathbf{y}$ are the solution vectors, and the vectors $\mathbf{c}$ and $\mathbf{b}$ can be thought of as objective functions for the primal and dual problems, respectively; A is a matrix that describes the constraints for feasible solutions.

The certificate of optimality for LP is as follows:

1. Primal feasibility	$A\mathbf{x} \leq \mathbf{b}$
	$\mathbf{x} \geq 0$
2. Dual feasibility	$A^T\mathbf{y} \geq \mathbf{c}$
	$\mathbf{y} \geq 0$
3. Strong duality	$\mathbf{b}^T\mathbf{y} = \mathbf{c}^T\mathbf{x}$

There are many other representations, as it is easy to convert between minimization and maximization problems by multiplying the objective function by -1, or transforming inequality constraints to equality constraints with slack variables. In any case, the duality theorems still apply.

What does Otti do? Given an instance of an LP problem (specified in an MPS file or using our API), Otti automatically derives the corresponding dual problem. Then, Otti derives the checks for the certificate of optimality (primal feasibility, dual correctness, strong duality). Finally, Otti compiles these checks into R1CS. In our implementation, the prover uses `lpsolve` to find the optimal solution to the primal ($\mathbf{x}$) and dual ($\mathbf{y}$) formulations, and then assigns $\mathbf{x}$ and $\mathbf{y}$ to nondeterministic variables in the R1CS instance, proving its satisfiability. By the strong duality theorem, satisfiability of these checks implies that $\mathbf{x}$ is the optimal solution.

5.2 Semidefinite Programming

LP is only sufficient for problems where the objective function is linear and all of the constraints can be specified as linear equalities or inequalities. When graphed visually, the constraints of an LP program produce a feasible region in the shape of a convex polytope; that is, the sides are flat. Semidefinite programming (SDP) can be used to solve a more general class of convex optimization problems. In fact, every

LP problem can be formulated as an SDP problem, although this would be inefficient to do in practice. SDP can also accommodate non-linear problems. The feasible region may be described by the intersection of a convex cone and an affine space (rather than only a polytope).

SDP problems can be written in standard form:

$$\begin{aligned}&\text{find a matrix } X \text{ that maximizes } C \bullet X\\ &\qquad\text{subject to } \forall i = 1,...,m,\ A_i \bullet X = b_i\\ &\qquad X \succeq 0\end{aligned}$$

X is the solution n-by-n matrix, and the C and A_i's are constant symmetric matrices with the same dimensions. We use "$X \succeq 0$" to denote that X must be a symmetric and positive semidefinite matrix. We use "$C \bullet X$" to denote $\sum_{i=1}^{n}\sum_{j=1}^{n} C_{ij} \cdot X_{ij}$. In this way, C can be thought of as the objective function, while each of the A_i's represents one of m constraints. The b_i's are real scalars that make up a vector $\mathbf{b}$, of length m. The corresponding dual problem is:

$$\begin{aligned}&\text{find a vector } \mathbf{y} \text{ and matrix } S \text{ that minimizes } \mathbf{b}^T\mathbf{y}\\ &\qquad\text{subject to } \sum_{i=1}^{m} y_i A_i + S = C\\ &\qquad S \succeq 0\end{aligned}$$

Here, the y_i's are variable real numbers that make up a vector $\mathbf{y}$, of length m. S is another variable $n \times n$ matrix that is required to be symmetric and positive semidefinite.

The certificate looks as follows:

1. Primal feasibility of X $\qquad \forall i,\ A_i \bullet X = b_i$, $\quad X \succeq 0$
2. Dual feasibility of $(\mathbf{y}, S)$ $\qquad \sum_{i=1}^{m} y_i A_i + S = C$, $\quad S \succeq 0$
3. Strong duality $\qquad C \bullet X = \mathbf{b}^T\mathbf{y}$ or $S \bullet X = 0$

Unlike LP, we cannot assert that the primal or dual problem will always obtain their optima or that there will be no duality gap. Hence, for there to be a valid certificate of optimality for SDP, we require that the instance be *strictly feasibile*: a common optimal value exists if both the primal and dual problems have feasible solutions inside the semidefinite cone [35].

What does Otti do? Otti expects an SDP problem instance specified in the SDPA format or using our API. Otti's derivation of the check is more involved than in LP. We first describe what Otti checks, then how an SDP instance is solved, and finally what happens when the instance is not strictly feasible.

Otti starts by deriving the nondeterministic checker for the certificate of optimality given above. One tricky part of the check is making sure that X and S are positive semidefinite. We use this fact to help us devise an efficient check: a matrix X is positive semidefinite if and only if it factors as LL^T, where L is a real lower triangular matrix, with non-negative diagonal entries. This factoring is known as the *Cholesky decomposition* [49]. Otti creates nondeterministic variables for the lower triangular matrix part of both decompositions, X_L and S_L, and the prover supplies these values exogenously. The nondeterministic checker that Otti derives then confirms that these matrices are lower triangular and indeed make up a Cholesky decomposition of each matrix (e.g., $X_L \cdot X_L^T = X$). Non-negative diagonal entries of X_L and S_L are used to confirm that $X \succeq 0$ and $S \succeq 0$.

Solving SDP instances. To solve SDP, Otti uses the `CSDP` [20] library, which implements the interior point algorithm [41]. This does not require the separate derivation of the dual problem by Otti, as `CSDP` derives that on its own. A drawback of SDP is that it is usually not possible to solve an SDP problem exactly. But `CSDP` terminates with a small duality gap (near 0) and its termination criteria is configurable. In our implementation, we configure `CSDP` so that the duality gap is equal to the precision level of our fixed-point implementation, and therefore, effectively 0 in that representation.

In order to satisfy strict feasibility, Otti requires the prover to specify an initial positive definite feasible (though not optimal) solution X_0. (A positive definite matrix is similar to a positive semidefinite matrix, except the diagonal entries of its Cholesky decomposition must be completely positive, rather than non-negative.) This can be leveraged by Otti (during solving) to obtain a feasible starting solution to the primal and dual problems [35]. There are many ways the prover can find such a point: it may be obvious from the problem instance, or it may be found with the short-step path-following method, the infeasible-interior-point method (where you start with any point), or by relaxing/modifying the SDP instance in some way so that a feasible point is more obvious [56]. As an example of an easy-to-derive initial point, imagine that the optimization problem aims to minimize the amount of time it takes to ship computer parts across the nation, given constraints on what different suppliers can produce in a given unit of time, where they can ship, and how fast they can ship their products. The initial point could be that one supplier produces and ships all products, and other suppliers produce and ship none. This is *not* the optimal solution (since it does not minimize the amount of time it takes to ship parts across the nation), but it may be a feasible one.

What happens if the instance is not strictly feasible? What can a prover do in the case that an SDP instance is not strictly feasible to convince the verifier of this fact? A straightforward option is for the prover to simply tell the verifier that the solution is not solvable (with no proof). After all, this would mean that a malicious prover can, at worst, deny service by claiming an instance is not solvable, but cannot violate soundness by convincing the verifier that a suboptimal solution is indeed optimal.

If the above is not acceptable, Otti also has a mechanism for the prover to generate a *proof of infeasibility* for the SDP instance, as a proof of the statement

$$(\exists i, A_i \bullet X_0 \neq b_i) \vee X_0 \not\succ 0$$

We use $\succ$ to denote positive definiteness. The above check proves infeasibility with respect to a particular starting point X_0 [35]. This means that a solver will not be able to derive an appropriate feasible solution to the primal and dual problems from X_0, and therefore, strict feasibility is not satisfied. Of course, since the prover is the one who generates X_0, a malicious prover could once again deny service by passing a bad X_0. A workaround is for the verifier to specify a set of initial values for X_0 as part of the public inputs, and for the prover to prove that the solution is infeasible with respect to all of them. How the verifier gets these points is application-specific; this is easy in the context of verifiable outsourced computation but more difficult in the zero-knowledge case. We find that for several applications we surveyed, knowing the general SDP constraints—even without necessarily knowing all of the entries in all matrices—could allow a verifier to craft various starting points.

Most of the infeasibility check is similar to the check for optimality. One of the conditions for infeasibility is that $X_0 \not\succ 0$. A single nonpositive eigenvalue is enough to confirm this condition. The prover calculates this eigenvalue and corresponding eigenvector and assigns them to nondeterministic variables in X_L and S_L. Otti then checks that this eigenvalue/eigenvector pair is valid for the provided X_0.

Finally, when the SDP instance is a small part of a larger program (rather than being the entire program), Otti allows the developer to specify (and the prover to prove) the disjunction of the above two cases: either the solution is optimal, in which case use the optimal value in the rest of the program, or the instance is infeasible in which case use some default value—without revealing to the verifier which branch was taken.

5.3 Stochastic gradient descent

For problems that do not necessarily fit the framework for either linear or semidefinite programming, a general-purpose approach is to use *gradient descent*. As long as the loss function f is adequately smooth, one can search for local optima by following a path determined by the gradient ∇f, which is defined over the entire data set. Specifically, one starts at a point x_0, sets $x_1 = x_0 - \epsilon \nabla f(x_0)$ for stepsize ϵ, and repeats. This is a very general procedure, but there is no guarantee that for a given loss function the gradient path will lead to a global optimum. Nonetheless, both theoretical work and empirical validation have shown that for examples of interest it is possible to find satisfactory optima.

It is often prohibitive to compute the gradient of the loss function f, so a standard approach is to use *stochastic gradient descent* (SGD). SGD iteratively takes steps in the direction of estimates ∇f_i that are based on a randomly selected sample i of the data set. These estimates are much cheaper to compute and are assumed to be an unbiased approximation of the gradient of the loss function. That is:

$$E(\nabla f_i) = \nabla f$$

Although it is not obvious that such a procedure should converge, a number of recent results show that stochastic gradient descent rapidly converges, especially when used in connection with adaptive stepsize algorithms [22, 23, 31, 50, 68, 74].

We show that the hypotheses for these convergence results provide sufficient control on the loss function to derive certificates for Otti to use. For example, Vaswani-Bach-Schmidt [68] study *growth conditions* on f that guarantee rapid convergence for SGD, even in non-convex settings. Notably, the *strong growth condition*, which is satisfied by *perceptrons* (a type of linear classifiers) and also often by overparametrized neural networks, stipulates that:

$$E(||\nabla f_i||^2) \leq E(||\nabla f||^2).$$

In this case, a local optimum for the loss function f is a stationary point for ∇f and thus is a stationary point for all of the f_i. So a deterministic certificate that a point is optimal can be obtained by checking that $||\nabla f_i|| = 0$ for each i.

To reduce costs further, one can construct a *probabilistic certificate* by asking the prover to commit to the purported solution, and then prove that $||\nabla f_i|| = 0$ for a random fraction of the samples. This relaxation allows a malicious prover to violate soundness with probability that depends on the number of samples tested; even then, when the prover violates soundness, the purported solution is suboptimal but must still be close to the optimal solution since the stochastic gradients are unbiased estimators.

Although the strong growth condition is a fairly stringent hypothesis, we believe that essentially any situation in which SGD converges fast enough to be useful in practice will give rise to a probabilistic certificate. For example, other convergence hypotheses for SGD (e.g., see [74]) are explicitly probabilistic: convergence for SGD can be shown to be rapid if with high probability:

$$||\nabla f_i(z)||^2 \geq \alpha_z ||z - x^*||^2$$

for a suitable constant α_z where x^* is the optimal x-value. In these cases, Otti can extract a probabilistic certificate in which there is soundness error that depends on the constant α_z. More generally, we put forth the following conjecture:

Conjecture 1. From the hypotheses of any theorem that guarantees rapid convergence of stochastic gradient descent one can extract a probabilistic certificate of optimality.

That is, whenever we expect SGD to converge quickly, Otti can produce a certificate. One justification for this conjecture is that any assumption that implies SGD converges fast means that at the optimal point, most of the time, the stochastic gradients cannot take you too far away. In future work, we plan to prove the above conjecture. In this paper, we focus on the strong growth condition in the situation when the loss function can be written as a sum

$$f(x) = \sum_{i=1}^{n} f_i(x),$$

where f_i is a smooth function. A wide variety of ML algorithms have this form, where f_i encodes the contribution to the loss function for a particular training example. The stochastic gradients are then simply ∇f_i.

What does Otti do? As in the LP and SDP cases, the prover in Otti compiles a certificate of optimality for a given SGD instance to R1CS. Under the strong growth condition on the loss function f, the certificate for a purported optimal point z looks as follows:

$$\forall i, \nabla f_i(z) = 0.$$

As discussed above, the strong growth condition guarantees that if z is in fact a local optimum of f, i.e., that $\nabla f(z) = 0$, then the certificate is valid. On the other hand, the condition that the stochastic gradients $\nabla f_i(z)$ are unbiased estimators of $\nabla f(z)$ implies that if z is not a local optimum and so $\nabla f(z) \neq 0$, then it cannot be the case that a certificate exists.

The size of the certificate is proportional to the number of data points used. However, the check of the certificate is substantially more efficient than actually performing stochastic gradient descent: for a linear classifier example with 10^5 data points, convergence requires roughly 50 descent iterations each of which loops through the subgradients for all of the points. In contrast, our certificate requires a single pass. In larger examples and when the stepsize is not tuned properly, the number of such iterations can easily be in the thousands.

Concrete loss functions. The procedure we have described above is very general, but in our experiments we focus on the particular case of loss-functions associated to perceptrons (linear classifiers). Here Otti's optimality check for SGD takes advantage of the property of the "hinge" loss function

$$f(x) = \sum_i \max(0, 1 - y_i \cdot \langle w, x_i \rangle)$$

and the "square hinge" loss function

$$f(x) = \sum_i \max(0, 1 - y_i \cdot \langle w, x_i \rangle)^2$$

where $\langle \cdot, \cdot \rangle$ denotes inner product. These functions satisfy the strong growth condition for separable data. These functions have subgradients that are most easily expressed piecewise, with one part in the $y_i \cdot \langle w, x_i \rangle < 1$ case and one which is always 0 when $y_i \cdot \langle w, x_i \rangle >= 1$. The second defines a region where the gradient of the loss function can be 0 simultaneously for all data-points; in other words w is optimal

```
#include "fxpt.h"
int main() {

  fp64 X0 = __exist(),
       X1 = __exist();

  __LP_maximize(
    3.0 * X0 + 4.0 * X1, // objective
    X0 + 2.0 * X1 <= 14.0,
    3.0 * X0 - 1.0 * X1 >= 0.0,
    X0 - 1.0 * X1 <= 2.0,
    X0 >= 0.0,
    X1 >= 0.0
  );

}
```

FIGURE 2—Provided code for primal formulation of LP instance. LP variables can either be constants, variables computed previously in the program, or nondeterministic variables provided exogenously by the prover for values previously committed.

when $\forall i, 1 - y_i \cdot \langle w, x_i \rangle = 0$. This is precisely the optimality check we encode in Otti's proof of SGD training.

6 Otti's transformations

This section gives an example of how Otti takes an optimization problem, transforms it, and then compiles it into R1CS. We use a toy LP example for simplicity, but a similar process exists for SDP and SGD, which we elaborate on in our extended tech report [13, Appendix A].

Below is the LP problem in its primal form.

$$\begin{aligned} &\text{find } \mathbf{x} \text{ that maximizes } \begin{pmatrix} 3 & 4 \end{pmatrix} \cdot \mathbf{x} \\ &\qquad \text{subject to } \begin{pmatrix} 1 & 2 \\ -3 & 1 \\ 1 & -1 \end{pmatrix} \cdot \mathbf{x} \leq \begin{pmatrix} 14 \\ 0 \\ 2 \end{pmatrix} \\ &\qquad \mathbf{x} \geq 0 \end{aligned}$$

This is what the developer formulates after it converts some real-world problem into an optimization problem. The developer then writes down this formulation using Otti's C API for LP or uses the MPS file format.

The resulting C program is given in Figure 2. It includes `fxpt.h` which is our fixed point library. The `__exist(·)` intrinsic tells Otti that this is a nondeterministic variable that should be provided by the prover and is not known at compile time or by the verifier. The `__LP_maximize(·)` intrinsic tells Otti that this is an LP problem on nondetermnistic variables $X0$ and $X1$. What Otti does next is fully automated.

First, Otti computes the dual formulation of the problem:

$$\begin{aligned} &\text{find } \mathbf{y} \text{ that minimizes } \begin{pmatrix} 14 & 0 & 2 \end{pmatrix} \cdot \mathbf{y} \\ &\qquad \text{subject to } \begin{pmatrix} 1 & -3 & 1 \\ 2 & 1 & -1 \end{pmatrix} \cdot \mathbf{y} \geq \begin{pmatrix} 3 \\ 4 \end{pmatrix} \\ &\qquad \mathbf{y} \geq 0 \end{aligned}$$

```
fp64 Y0 = __exist(),
     Y1 = __exist(),
     Y2 = __exist();

__LP_minimize(
  14.0 * Y0 + 2.0 * Y2, // objective
  Y0 + 3.0 * Y1 + Y2 >= 3.0,
  2.0 * Y0 - 1.0 * Y1 - 1.0 * Y2 >= 4.0,
  Y0 >= 0.0,
  Y1 <= 0.0,
  Y2 >= 0.0
);
```

FIGURE 3—Automatically generated code for dual formulation corresponding to the LP instance given in Figure 2.

```
int check = __check(
    // primal is satisfied
    X0 + 2.0 * X1 <= 14.0,
    3.0 * X0 - 1.0 * X1 >= 0.0,
    X0 - 1.0 * X1 <= 2.0,
    X0 >= 0.0,
    X1 >= 0.0,

    // dual is satisfied
    Y0 + 3.0 * Y1 + Y2 >= 3.0,
    2.0 * Y0 - 1.0 * Y1 - 1.0 * Y2 >= 4.0,
    Y0 >= 0.0,
    Y1 <= 0.0,
    Y2 >= 0.0,

    // strong duality holds
    3.0 * X0 + 4.0 * X1 == 14.0 * Y0 + 2.0 * Y2
);
```

FIGURE 4—Automatically generated nondeterministic checker for LP instance given in Figure 2.

Specifically, Otti generates the C code snippet given in Figure 3. Then, Otti generates the condition for strong duality. In mathematical terms, it is the following assertion:

$$\text{Strong Duality} \qquad \begin{pmatrix}14 & 0 & 2\end{pmatrix} \cdot \mathbf{y} = \begin{pmatrix}3 & 4\end{pmatrix} \cdot \mathbf{x}$$

Given the primal, the dual, and the strong duality condition, Otti generates a nondeterministic checker that verifies the certificate of optimality for this problem instance. Recall that the certificate of optimality asserts that: (1) the primal solution is satisfiable, (2) the dual solution is satisfiable, and (3) strong duality holds. Figure 4 gives the C code for the nondeterministic checker generated by Otti.

The LP nondeterministic checker defined over nondeterministic variables $X0, X1, Y0, Y1, Y2$ is the code that Otti actually compiles to R1CS. Everything else is not compiled to R1CS. Instead, the primal and dual code shown earlier is used by Otti to *find* the values for these nondeterministic inputs by invoking `lpsolve` with the corresponding parameters. This helps the prover generate the witness it needs. Minimization problems for LP are computed similarly.

7 Implementation

Otti is built on top of CirC [52], which is a compiler that can translate a subset of C (and other high-level languages) to existentially quantified circuits. CirC supports a few different output formats, including the one we use, R1CS. CirC uses a typed version of SMT-LIB [8] as its intermediate representation and the Z3 SMT solver [30] for evaluating terms of the program, error checking and optimizations, such as equation elimination. CirC is more than a compiler; it is also a solver that a zkSNARK prover can call to obtain the witness for a R1CS instance. CirC takes in inputs to the program and evaluates the compiled R1CS with those inputs using Z3. Otti uses CirC as a compiler extended in a variety of ways, and also as a solver extended with LP, SDP, and SGD solvers.

For output and proofs, Otti produces files in the *zkInterface* binary format [10], which is a recent standard for specifying R1CS in a portable manner. Any zkSNARK system that supports zkInterface, including libsnark [2], bulletproofs [25], and bellman [1], can process Otti proofs. We modified the open source implementation of the Spartan zkSNARK library [9] to support zkInterface files as input and to support arbitrary R1CS instances (the original implementation supported only instances with power-of-two-sized matrices). Spartan does not need a trusted setup and, at the time of implementation, had the fastest open-sourced zkSNARK prover.

Rational numbers. CirC does not have support for any sort of rational number representation beyond integers. We add support for rational numbers in the form of a fixed-point number representation. This type has 32 bits of precision before and after the point. We write our fixed-point as a typedef in C: `typedef double fp64;`. This type definition needs to be declared and used in place of `double` or `float` inside of any user program that deals with rational numbers. Users can simply include `fxpt.h`, a header file which contains this type definition and the function `F_EQUAL`, which compares the equality of two fixed-points within a certain epsilon. This epsilon can also be set within this file, if desired.

When Otti compiles C programs it treats fixed-point types as a double that is cast as a new FixedPoint type. The double behavior is true to the IEEE 754 standard. When being cast, the number is truncated to fit into the 32 bit precision after the point, as other C types are. For all other operations, this FixedPoint type is treated as a large integer; the number x is stored as $2^{32} \cdot x$. During multiplication and division, FixedPoint types are cast up to a 96-bit number to avoid overflow. Users should be aware that precision is inevitably lost after repeated multiplications and divisions, as in any FixedPoint format.

Versions of CirC and how Otti uses each. CirC's original implementation [5] was in Haskell; Otti extends this implementation with support for zkInterface and rational numbers as explained above, and incorporates solvers for LP and SDP. A drawback of this Haskell implementation is that it requires significant memory and time for compiling C programs into R1CS, and Otti inherits these inefficiencies. More recently [11], CirC has been ported to Rust and is significantly more resource efficient. We extend this Rust version to support our SGD certificates (we were unable to compile these certificates with the Haskell version since it exhausted our test machine's memory resources), and are in the process of porting LP and SDP to this new version as well.

8 Evaluation

This section answers our animating question: is the use of nondeterministic checkers and numerical optimization certificates of optimality and infeasibility an effective technique at reducing the cost of expressing these computations in R1CS. Our results suggest this is indeed the case.

Test suites. For LP, we use a subset of the netlib LP/data model library [4, 37]. These benchmarks are inspired by real-world applications on resource allocation, finance, and government. They are standard in evaluating the correctness and performance of LP solvers. Figure 5 lists these benchmarks, the number of variables and linear constraints, the resulting number of R1CS produced by Otti, and the size of proofs produced by Spartan.

For SDP, we use SDPLIB [21], which is a set of problems for benchmarking existing SDP solvers. These problems come from applications like truss topology design, control systems engineering, and combinatorial optimization problems. We choose initial points for the feasible problems the same way the `CSDP` solver does [20]. We evaluate infeasible instances in our extended tech report [13, Appendix B]. Figure 8 lists these benchmarks, the number of variables, the size of the semidefinite matrices, the resulting number of R1CS produced by Otti, and the size of proofs produced by Spartan.

For SGD, we use binary classification problems from the Penn ML Benchmarks (PMLB) dataset [51, 58]. This set of benchmarks is curated specifically for evaluating supervised ML algorithms, and comes from a variety of applications. The datasets we use either have binary classes and are linearly separable, or are datasets that have multiple classes but can be projected onto two linearly separable classes. Figure 12 lists these benchmarks, the number of datapoints and features in them, the resulting number of R1CS produced by Otti, and the size of proofs produced by Spartan. Unless otherwise stated, we check the stochastic estimate of *all* data points (rather than a fraction of them), so there is no additional soundness error.

Experimental setup. We perform all of our measurements on a server with 40 Intel Xeon E5-2660 v3 CPUs (2.60GHz) and 200 GB DDR4 memory. Due to the extremely slow completion time of using existing compilers and proof systems on our problem instances, it was impossible to run a baseline for even small tests from the benchmark suites we chose. The largest instance which we could run with prior work has 5 LP variables and for that, Otti is 5 orders of magnitude faster. Our baselines are instead existing LP, SDP, and SGD solvers which provide no proofs. Our results are therefore overhead over unverifiable solvers, rather than speedup over prior work.

Dataset	Equations	Variables	R1CS constraints	Proof size (KB)
afiro	28	32	36,811	19.82
sc50a	51	48	54,066	19.82
sc50b	51	48	55,085	19.82
adlittle	57	97	180,747	29.33
sc105	106	103	113,282	20.51
scagr7	130	140	229,061	29.33
israel	175	142	511,156	47.02
agg	489	163	1,069,523	47.71
sc205	206	203	220,520	29.33
brandy	221	229	815,356	47.02
beaconfd	174	262	1,149,169	47.71
agg2	517	302	1,887,762	47.71
agg3	517	302	1,891,690	47.71
lotfi	154	308	326,102	30.01
scorpion	389	358	731,137	47.02
sctap1	301	408	414,101	47.71
scfxm1	331	457	965,504	47.02
bandm	306	472	1,093,340	47.02
scagr25	472	500	823,136	47.71
degen2	445	534	626,407	47.71
scsd1	78	760	1,034,359	47.02
fffff800	525	854	1,479,725	47.71
scfxm2	661	914	1,932,500	47.02
scrs8	491	1,169	1,601,971	47.71
bnl1	644	1,175	2,324,544	81.10
scsd6	148	1,350	1,845,814	47.71
modszk1	688	1,620	1,805,821	47.71
scsd8	398	2,750	3,607,188	81.10

FIGURE 5—Number of LP equations, variables, R1CS constraints, and proof sizes generated for the LP benchmarks.

Our experimental procedure follows a 4-step process: (1) Compile: we compile each benchmark using Otti to get the corresponding R1CS. (2) Solve: we supply the public and the (prover's) private inputs to Otti, which engages Otti's numerical solver, and which produces the satisfying assignment to the R1CS instance. (3) Export: we export the R1CS instance, inputs, and witness in zkinterface's binary format (§7). (4) Prove and verify: the zkinterface files are then consumed by Spartan. Finally, we measure the number of constraints for the resulting R1CS, the size of the zkSNARK proof, and the execution time of the prover and verifier.

Prover and Verifier runtime. We measure the running time of the following components and aggregate those for each benchmark to get the end-to-end runtime. We exclude compilation time, as this is done once for each problem instance.

- *Solver runtime:* time for the numerical solver to determine the optimal solution to the optimization problem.
- *Prover runtime:* time it takes the Spartan prover to generate a zkSNARK proof given an R1CS instance and witness.
- *Verifier runtime:* time it takes the Spartan verifier to check the proof for a given R1CS instance.

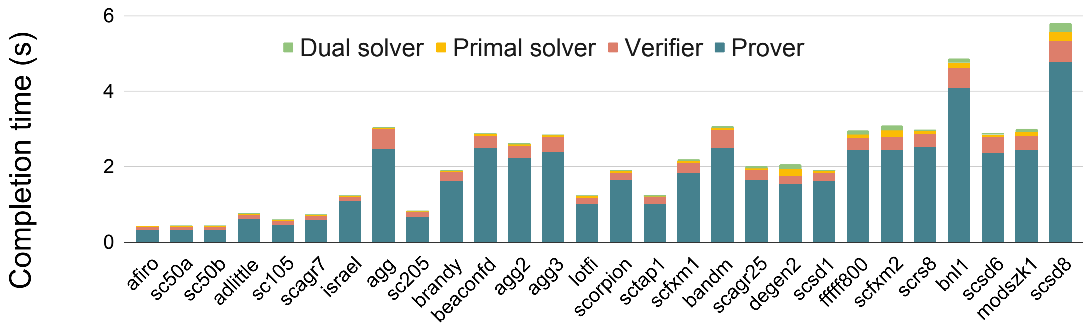

FIGURE 6—Runtime attribution of LP benchmarks in Otti.

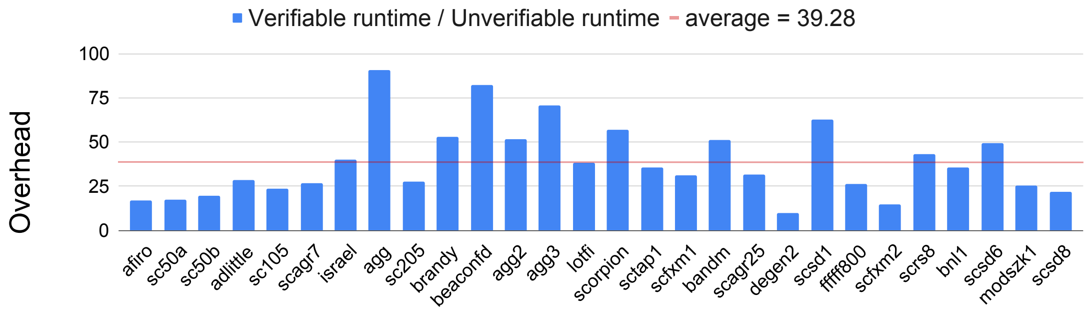

FIGURE 7—Overhead of Otti over baseline for LP benchmarks.

8.1 Linear programming

In this section we evaluate the effectiveness of Otti's automated certificates of optimality for LP problems in order to produce smaller R1CS instances. We use `lp_solve` v5.5 to solve the LP problems (both for the baseline and in Otti).

R1CS and proof sizes. Figure 5 reports the name of the different benchmarks we consider, the number of LP variables and linear equations, the size of the R1CS instance produced by Otti, and the proof sizes produced by Spartan for the different LP instances, which are all well below 100 KB. We find that the number of R1CS constraints cannot be predicted solely from the number of variables or equations; it also depends on the complexity of each equation, as some can be defined over tens or hundreds of variables, while others defined over a single variable.

Otti compiles instances with thousands of LP variables and hundreds of equations. For comparison, CirC is unable to handle more than 5 LP variables and 4 equations on our test machine due to the excessive amount of memory required. Even then, CirC produces more R1CS constraints for that tiny instance than Otti does for the largest instance in Figure 5.

Runtime. Figure 6 shows that Otti can solve and generate proofs for optimization problems within hundreds of milliseconds for small problems like `afiro` (which is one of the 13 benchmark problems from the Systems Optimization Laboratory at Stanford) to a few seconds for large problems like `modszk1` (which is a multi-sector economic planning model). As we expect, the cost is dominated by Spartan's prover generating a proof given the R1CS instance and witness—solving the primal and dual optimization problems to get the witness is only a fraction of the cost (it is the same as the baseline). Verifying the proof is relatively cheap (in all cases less than 1 second), but not constant, since Spartan's verifier performs $O(\sqrt{n})$ operations, where n is the size of the R1CS instance.

We now turn our attention to the relative overhead of Otti's prover over a baseline that generates no proofs and need not solve both the primal and dual problems. Figure 7 depicts the performance of Otti's prover normalized by said baseline. In all cases, we find that Otti introduces significant costs; Otti's prover is on average 40× more expensive than the baseline. Nevertheless, this constitutes a significant achievement, given that for proof systems with succinct proofs and no trusted setup, the applications that prior works evaluate are smaller examples with overheads of at least 3 orders of magnitude. For example, Hyrax [70] and Libra [73] evaluate the multiplication of two matrices, and observe 3 to 4 orders of magnitude overhead for the prover compared to native execution (that generates no proof). Similarly for the generation of Merkle tree proofs. Aurora [15] and Fractal [27] operate on synthetic R1CS instances but also report overheads of 3 orders of magnitude. In contrast, Otti operates on real benchmarks used by the optimization community with modest overhead.

Dataset	Equations	Matrix size	R1CS constraints	Proof size (KB)
truss1	6	13	3,007,933	79.20
hinf1	13	14	4,703,942	79.88
hinf2	13	16	6,536,398	79.88
hinf3	13	16	6,536,398	79.88
hinf4	13	16	6,536,398	79.88
hinf5	13	16	6,536,398	79.88
hinf6	13	16	6,536,398	79.88
hinf7	13	16	6,536,398	79.88
hinf8	13	16	6,536,398	79.88
hinf9	13	16	6,536,398	79.88
control1	21	15	6,968,254	79.88

FIGURE 8—Number of equations, size (n) of the SDP $n \times n$ matrix variables, number of R1CS constraints, and proof sizes generated for SDP benchmarks.

8.2 Semidefinite programming

Similarly to the LP description above, we measure the effectiveness of Otti's automated certificates of optimality, but this time for SDP problems. We use CSDP v6.2.0 to solve SDP problems in Otti and as a baseline. We talk about feasible problems in this section; for a discussion of infeasible instances, see our tech report [13, Appendix B].

Instance sizes. Figure 8 gives the results of compiling the SDP benchmarks with Otti. The number of R1CS constraints is significantly larger than in LP for small SDP instances due to (1) the higher complexity of the SDP nondeterministic checker; (2) the fact that Otti compiles both the feasible and infeasible branches (if one does not need such disjunction proofs, the costs are lower than what we report); and (3) each SDP "variable" is actually an $n \times n$ matrix, making the instance size somewhat deceiving. For example, an SDP matrix of size 16 actually has 256 variables per matrix. When this number grows above 300, our current prototype is unable to compile SDP checks; we are working on porting our prototype to CirC's new Rust codebase [11] which is more efficient. Nevertheless, this constitutes a significant improvement over prior work since we are unable to compile any of the SDP problems with any existing open sourced compiler.

Runtime. Otti uses CSDP to measure the solver runtimes for feasible SDP instances. The nature of its interior point algorithm means that the primal and dual problems are solved in tandem and cannot be divided into two separate solving times, as in LP. We also use CSDP's initsoln method to find a good heuristic starting point.

Figure 9 shows the results. Otti takes more time on SDP problems than LP, spending a few seconds to solve and generate each proof. However, the overhead relative to the baseline, given in Figure 10, is similar to the LP experiments: Otti's prover is on average 30× more expensive than the baseline. In any case, generating proofs dominates the runtime, while verification is cheap—less than a second. The set of hinf problems (from control systems engineering) demonstrates that while equivalent instance size generally causes similar runtime, this is not always the case. For example, hinf2's proving time is 3 seconds longer than other problems of the same size. This is due to Spartan's implementation currently using a variable-time multiscalar multiplication function (from curve25519-dalek), and hence time can vary depending on the instance parameters and witness. Switching to a constant-time implementation of multiscalar multiplication would avoid this behavior (a potential side channel).

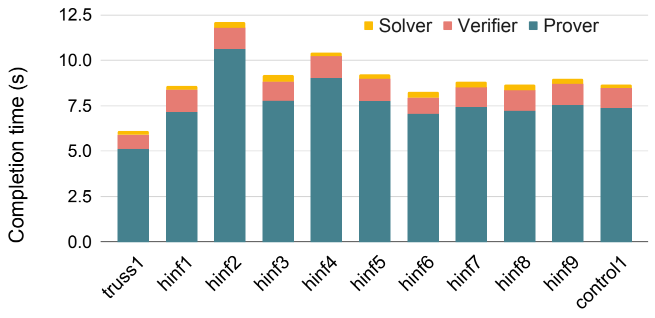

FIGURE 9—Runtime attribution of SDP benchmarks in Otti.

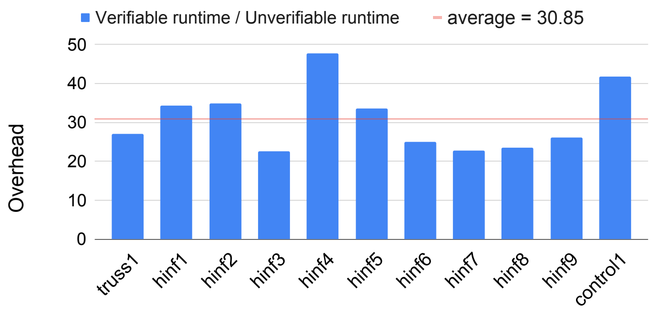

FIGURE 10—Overhead of Otti over baseline for SDP benchmarks.

8.3 Stochastic Gradient Descent

Finally we evaluate the effectiveness of Otti on SGD, with the training of a linear binary classifier (i.e., a perceptron). We generate certificates of optimality as described in Section 5.3. We use scikit-learn [55], to train the SGD classifier and get the optimal fitted plane; the associated certificate is then checked by Otti.

Instance sizes. Figure 12 reports the name of the different classification datasets, their sizes, the resulting number of R1CS constraints produced by Otti, and the size of proofs. In contrast to the LP and SDP datasets, since the certificate involves a gradient condition for every one of the input points, the resulting R1CS instances are significantly larger. The largest dataset we test is the clean2 dataset, which consists of 6,598 datapoints and 168 features. Otti compiles a nondeterministic checker for the corresponding certificate of op-

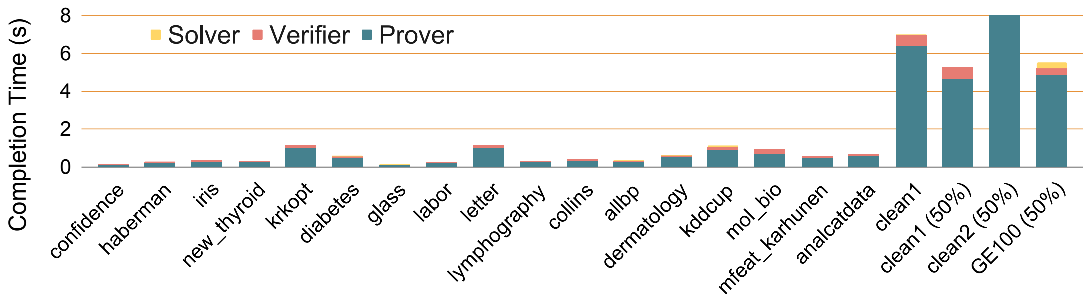

FIGURE 11—Runtime attribution of Perceptron training with SGD in Otti. The last three benchmarks use probabilistic certificates that check the stochastic gradient of half of the data points. The `clean2` results are 0.48 sec for solving, 1.34 sec for verifying, and 18.23 sec for proving.

Dataset	**Data points**	**Features**	**R1CS constraints**	**Proof size (KB)**
confidence*	72	3	13,027	14.08
haberman	306	3	60,237	19.36
iris*	150	4	4,730	11.47
new_thyroid*	215	5	25,810	14.75
krkopt*	28056	6	399,555	29.31
diabetes	768	8	212,501	28.64
glass*	205	9	7,571	11.47
labor	57	16	22,763	14.75
letter*	20000	16	374,655	29.31
lymphography*	148	18	31,823	14.75
collins*	485	23	31,733	14.75
allbp*	3772	29	103,451	20.03
dermatology*	366	34	55,877	19.36
kddcup*	494020	41	198,840	28.64
molecular_biology_promoters	106	57	41,343	19.36
mfeat_karhunen*	2000	64	162,352	28.64
analcatdata_authorship*	841	70	231,455	28.64
clean1	476	168	3,473,740	79.20
clean1†	476	168	2,262,837	79.20
clean2†	6598	168	6,773,944	79.88
GE1000†	1600	1000	571,558	45.92

FIGURE 12—Number of data points and features (for each data point), as well as the number of R1CS constraints generated by Otti and proof sizes generated by Spartan for the PMLB datasets. We use * to denote datasets with more than 2 classes that we projected onto 2 classes. We use † to denote datasets for which Otti generates a nondeterministic checker that only checks the stochastic gradient of half of the points (i.e., a probabilistic certificate). *GE1000* is short for *GAMETES_Epistasis_2_Way_1000atts_0.4H_EDM_1*.

timality, and the result is 9,448,632 R1CS constraints. We are unable to run Spartan to prove that the prover knows a witness for this nondeterministic checker since it pushes past the memory limit of our machine. Instead, we choose to compile a probabilistic certificate of optimality for this dataset by having the verifier check half of the data points instead of all. For `clean2`, this results in 6,773,944 R1CS constraints, for which we are able to generate and verify proofs.

Using a probabilistic certificate introduces additional soundness error. For example, if a dataset consists of n input data points, and a malicious prover produces a suboptimal solution for which the stochastic gradient of all but ℓ of those data points is 0, then the verifier, checking k of the n data points uniformly at random, can detect the prover's misbehavior with probability $1 - \binom{n-\ell}{k} / \binom{n}{k}$.

While this detection probability can be low when ℓ is small, it also means that the solution should be close to optimal; analyzing the exact distance is something that we plan to formalize in future work. Furthermore, other SGD convergence assumptions (e.g., [74]) are probabilistic and we believe that deriving certificates from these assumptions do not require the verifier to check a condition on all data points.

Runtime. Figure 11 shows that Otti can solve and generate proofs for training a binary classifier on real datasets, with reasonable performance. As we expect, the cost is dominated by Spartan's proof generation given the large R1CS instances and witnesses. Training on the dataset is particularly fast and can take advantage of hardware acceleration. Verifying the proof is relatively cheap, on the order of seconds.

Now, let us consider the relative overhead of Otti's prover over a baseline that generates no proofs and only implements the training. We find that Otti has variable overhead, ranging from 1 to 2 orders of magnitude. Figure 13 depicts the performance of Otti's prover normalized by said baseline. Despite this overhead, we believe this is the first work to generate zkSNARK proofs of training a classifier on a real dataset.

8.4 Otti's impact on solving, proving, or verifying time

Otti's goal is to produce an R1CS instance for a nondeterministic checker whose correct execution implies the optimality of the underlying optimization instance. As a result, Otti does not impact the performance of the underlying solvers in any way. Otti impacts the performance of zkSNARKs only in the sense that it produces the R1CS instance, but it uses the zkSNARK as a black box. Indeed our contribution is to make the R1CS instance as small as possible. Note that if Otti uses a different solver or zkSNARK, Otti would inherit their properties. For example, if we use the Groth16 [40] implementation in `libsnark` [2] instead of Spartan, the proof sizes would be constant but their generation would be slower and one would need to accommodate a trusted setup.

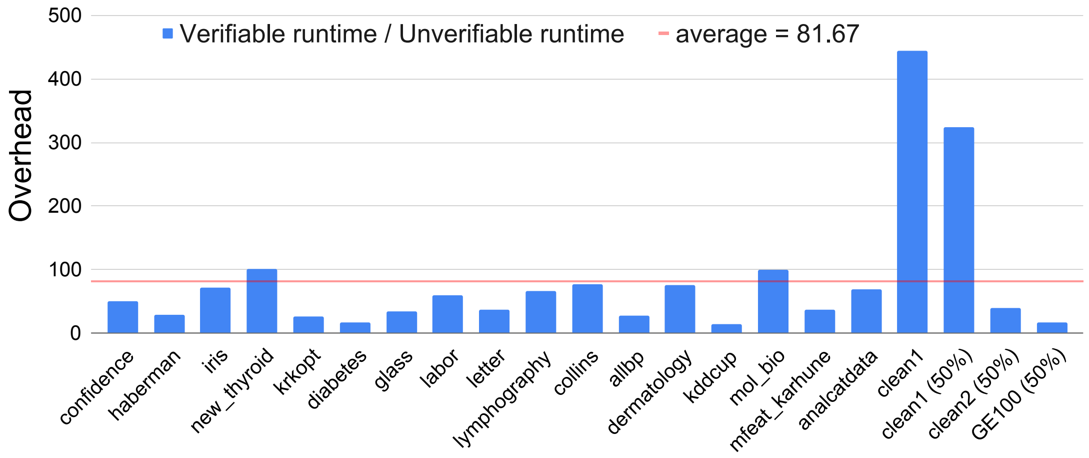

FIGURE 13—Overhead of Otti's prover (proof and solve) for training and proving a perceptron using SGD versus a baseline that lacks proofs.

9 Related work

The idea of using strong duality to verify the output of LP programs has also been explored in the past [29, 39]. For example, Goemans et al. [39] define doubly-efficient pseudo-deterministic proofs, where a polynomial time prover solves a problem and aids a polynomial time verifier in verifying that the solution is correct. One of their examples includes LP programs. Unlike Otti, their results are purely theoretical; they prove that an efficient pseudo-deterministic interactive proof exists for LP. Analogously to our LP certificate, they use strong duality to verify that they find the optimal solution.

Hoogh et al. [29] similarly use LP certificates of duality to achieve verifiable LP, but they do so in the context of semi-honest MPC, and their evaluation tests smaller instances. For example, their largest instance has around 200 variables and equations and takes over 20 hours to complete. In contrast, Otti uses LP certificates to produce small zkSNARK instances, runs on real problems, and achieves practical performance: an LP instance with 398 equations and 2,750 variables takes under 6 seconds.

To our knowledge, Otti is the first system to leverage certificates for SDP to generate faster proofs, as well as automatically extracting these certificates from standard problem formulations and file formats. Finally, Otti introduces deterministic and probabilistic certificates for SGD under the strong growth assumption and posits the existence of many more under convergence hypotheses.

Our use of the SGD certificate to prove the training of a perceptron is related to the larger area of using zero-knowledge proofs in ML. Most work in this area produces proofs of inference [32, 46, 71], or uses proofs to establish that inputs in federated learning fall in an acceptable range [60–62]. Proofs of training (which is what Otti's SGD evaluation tackles), are harder to generate and verify. DIZK [72] requires tens of machines to prove the training of a small and simple linear regression model. VeriML [76], on the other hand, generates proofs for a wider class of models but only proves that a handful of randomly-selected training iterations (rather than the full training) are correct; this limits the guarantees it provides. To our knowledge, Otti is the only system to be able to prove the full training of a realistic ML model on real datasets.

10 Discussion

Otti is the first compiler for zkSNARKs that supports optimization problems of a realistic size. The overhead of Otti, while still high, is in many cases sufficiently small so as to be practical for actual usage. Moreover, the evaluation comparison is a stringent one—we are comparing to native solvers running highly optimized floating point algorithms.

Nonetheless, there are many avenues for improvement. For one thing, it would be useful to expand the universe of optimization problems that Otti can support. For instance, our SDP protocol assumes the conditions that guarantee strong duality hold. Instead, we could draw inspiration from works that introduce certificates for convex optimization problems from Farkas's lemma and theorems of alternatives [45, 47, 57]. As another example, the cases of SGD we consider require strong hypotheses such as the strong growth condition. While the strong growth condition applies to examples of interest, it is also too stringent to accommodate other natural examples. Notably, classification problems that are not linearly separable do not satisfy it. Further work to understand the weakest possible hypotheses that still result in usable probabilistic certificates is needed.

More generally, Otti's approach of aggressively using nondeterministic checkers to reduce proof size will be of broad applicability in the design of efficient compilers for zkSNARKS. Nondeterministic checkers have been used before: prior compilers [24, 28, 42–44, 64, 65, 67] apply them primarily for avoiding certain arithmetic operations (as in the example of Section 2.3), transformation between representations (e.g., boolean to arithmetic), expressing or transferring state across proof instances [24, 28, 33, 42], or expressing threads and concurrency [64]. However, one of the lessons of Otti is that these techniques should be more widely used.

Finally, a pressing issue when using nondeterministic checkers is the question of correctness. We assumed that a solution was optimal if it passed the nondeterministic checker. However, how does one know whether the nondeterministic checker generated by Otti is itself correct (i.e., bug free)? Developing a formal methods framework for proving the correctness of the program transformations is a crucial missing piece in this ecosystem (along with formally verified compilers, though some preliminary efforts exist [34]).

11 Conclusion

This paper introduces Otti, a compiler for zkSNARKs that supports optimization problems. Otti allows the programmer to simply specify the problem and derives a proof that a particular output is optimal. The key idea behind Otti is the use of nondeterministic checkers and certificates of optimality to verify the purported optimal point rather than verifying the correct execution of the optimization algorithm. This results in a radical reduction in the size of the R1CS encodings produced. Our experimental evaluation confirms that Otti provides the first zkSNARK proofs that are practical for optimization and ML training problems with real datasets.

Acknowledgments

We thank the anonymous reviewers and our shepherd, Alin Tomescu, for helpful comments that improved our work. We also thank Fraser Brown, Patrick Cousot, Justin Thaler, Riad Wahby, and Mike Walfish for helpful conversations. We thank Alex Ozdemir for help with CirC, and Srinath Setty for help with Spartan. This work was funded in part by NSF grants CNS-2045861, CNS-2107147, CNS-2124184, DARPA contract HR0011-17-C0047, AFOSR grant FA9550-18-1-0415, and a JP Morgan Chase & Co Faculty Award. Any views expressed herein are solely those of the authors listed.

References

[1] bellman. `https://github.com/zkcrypto/bellman`.

[2] libsnark: a c++ library for zksnark proofs. `https://github.com/scipr-lab/libsnark`.

[3] MPS file format. `http://plato.asu.edu/cplex_mps.pdf`.

[4] LP/data index. `https://ampl.com/netlib/lp/data/`, 2013.

[5] CirC: Compiler to circuit representations. `https://github.com/circify/compiler`, 2021.

[6] The glop linear solver. `https://developers.google.com/optimization/lp/glop`, 2021.

[7] lpsolve: Mixed integer linear programming (MILP) solver. `http://lpsolve.sourceforge.net/5.5`, 2021.

[8] SMT-LIB: The satisfiability modulo theories library. `http://smtlib.cs.uiowa.edu`, 2021.

[9] Spartan: High-speed zksnarks without trusted setup. `https://github.com/microsoft/spartan/`, 2021.

[10] zkinterface, a standard tool for zero-knowledge interoperability. `https://github.com/QED-it/zkinterface/`, 2021.

[11] CirC: The Circuit Compiler. `https://github.com/circify/circ`, 2022.

[12] R. Anderson, I. Ashlagi, D. Gamarnik, and A. E. Roth. Finding long chains in kidney exchange using the traveling salesman problem. *Proceedings of the National Academy of Sciences of the United States of America*, 112(3), 2015.

[13] S. Angel, A. J. Blumberg, E. Ioannidis, and J. Woods. Efficient representation of numerical optimization problems for snarks (extended version). Cryptology ePrint Archive, Report 2021/1436.

[14] E. Ben-Sasson, A. Chiesa, D. Genkin, E. Tromer, and M. Virza. Snarks for c: Verifying program executions succinctly and in zero knowledge. In *Proceedings of the International Cryptology Conference (CRYPTO)*, 2013.

[15] E. Ben-Sasson, A. Chiesa, M. Riabzev, N. Spooner, M. Virza, and N. P. Ward. Aurora: Transparent succinct arguments for R1CS. In *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, 2019.

[16] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza. Succinct non-interactive zero knowledge for a von neumann architecture. In *Proceedings of the USENIX Security Symposium*, 2014.

[17] J. Bootle, A. Cerulli, J. Groth, S. K. Jakobsen, and M. Maller. Arya: Nearly linear-time zero-knowledge proofs for correct program execution. In *International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT)*, 2018.

[18] J. Bootle, A. Chiesa, and J. Groth. Linear-time arguments with sub-linear verification from tensor codes. In *Proceedings of the Theory of Cryptography Conference (TCC)*, 2020.

[19] J. Bootle, A. Chiesa, and S. Liu. Zero-knowledge succinct arguments with a linear-time prover. Cryptology ePrint Archive, Report 2020/1527, 2020.

[20] B. Borchers. CSDP, a C library for semidefinite programming. *Optimization methods and Software*, 11(1-4), 1999.

[21] B. Borchers. SDPLIB 1.2, a library of semidefinite programming test problems. *Optimization Methods and Software*, 11(1-4), 1999.

[22] L. Bottou. On-line learning and stochastic approximations. In *In On-line Learning in Neural Networks*, pages 9–42. Cambridge University Press, 1998.

[23] L. Bottou. Stochastic gradient descent tricks. In *Neural Networks: Tricks of the Trade - Second Edition*, volume 7700. 2012.

[24] B. Braun, A. J. Feldman, Z. Ren, S. Setty, A. J. Blumberg, and M. Walfish. Verifying computations with state. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2013.

[25] B. Bünz, J. Bootle, D. Boneh, A. Poelstra, P. Wuille, and G. Maxwell. Bulletproofs: Short proofs for confidential transactions and more. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2018.

[26] B. Bünz, B. Fisch, and A. Szepieniec. Transparent SNARKs

from DARK compilers. In *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, 2020.

[27] A. Chiesa, D. Ojha, and N. Spooner. Fractal: Post-quantum and transparent recursive proofs from holography. In *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, 2020.

[28] C. Costello, C. Fournet, J. Howell, M. Kohlweiss, B. Kreuter, M. Naehrig, B. Parno, and S. Zahur. Geppetto: Versatile verifiable computation. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2015.

[29] S. de Hoogh, B. Schoenmakers, and M. Veeningen. Certificate validation in secure computation and its use in verifiable linear programming. In *Proceedings of the International Conference on Cryptology in Africa*, 2016.

[30] L. De Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2008.

[31] J. Duchi, E. Hazan, and Y. Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12, 2011.

[32] B. Feng, L. Qin, Z. Zhang, Y. Ding, and S. Chu. Zen: An optimizing compiler for verifiable, zero-knowledge neural network inferences. *Cryptology ePrint Archive*, 2021.

[33] D. Fiore, C. Fournet, E. Ghosh, M. Kohlweiss, O. Ohrimenko, and B. Parno. Hash first, argue later: Adaptive verifiable computations on outsourced data. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2016.

[34] C. Fournet, C. Keller, and V. Laporte. A certified compiler for verifiable computing. In *Proceedings of the IEEE Computer Security Foundations Symposium*, 2016.

[35] R. M. Freund. Introduction to semidefinite programming (SDP). *Massachusetts Institute of Technology*, 2004.

[36] K. Fujisawa, M. Kojima, K. Nakata, and M. Yamashita. SDPA (semidefinite programming algorithm) user's manual—version 6.2. 0. *Department of Mathematical and Computing Sciences, Tokyo Institute of Technology. Research Reports on Mathematical and Computing Sciences Series B: Operations Research*, 2002.

[37] D. M. Gay. Electronic mail distribution of linear programming test problems. *Mathematical Programming Society COAL Newsletter*, 13, 1985.

[38] R. Gennaro, C. Gentry, B. Parno, and M. Raykova. Quadratic span programs and succinct NIZKs without PCPs. In *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, 2013.

[39] M. Goemans, S. Goldwasser, and D. Holden. Doubly-efficient pseudo-deterministic proofs. *arXiv preprint arXiv:1910.00994*, 2019.

[40] J. Groth. On the size of pairing-based non-interactive arguments. In *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, 2016.

[41] C. Helmberg, F. Rendl, R. J. Vanderbei, and H. Wolkowicz. An interior-point method for semidefinite programming. *SIAM Journal on optimization*, 6(2), 1996.

[42] A. Kosba, A. Miller, E. Shi, Z. Wen, and C. Papamanthou. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2016.

[43] A. Kosba, C. Papamanthou, and E. Shi. xjsnark: A framework for efficient verifiable computation. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2018.

[44] A. Kosba, Z. Zhao, A. Miller, Y. Qian, T.-H. Chan, C. Papamanthou, R. Pass, s. abhi, and E. Shi. C∅ c∅: a framework for building composable zero-knowledge proofs. *Cryptology ePrint Archive, Report 2015/1093*, 2015.

[45] M. Liu and G. Pataki. Exact duality in semidefinite programming based on elementary reformulations. *SIAM Journal on Optimization*, 25(3), 2015.

[46] T. Liu, X. Xie, and Y. Zhang. ZkCNN: Zero knowledge proofs for convolutional neural network predictions and accuracy. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2021.

[47] Z.-Q. Luo and W. Yu. An introduction to convex optimization for communications and signal processing. *IEEE Journal on selected areas in communications*, 24(8), 2006.

[48] M. Maller, S. Bowe, M. Kohlweiss, and S. Meiklejohn. Sonic: Zero-knowledge snarks from linear-size universal and updatable structured reference strings. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2019.

[49] C. B. Moler and G. W. Stewart. On the householder-fox algorithm for decomposing a projection. *Journal of Computational Physics*, 28(1), 1978.

[50] Y. E. Nesterov. Smooth minimization of non-smooth functions. *Mathematical Programming*, 103(1), 2005.

[51] R. S. Olson, W. La Cava, P. Orzechowski, R. J. Urbanowicz, and J. H. Moore. PMLB: a large benchmark suite for machine learning evaluation and comparison. *BioData Mining*, 10(36), Dec 2017.

[52] A. Ozdemir, F. Brown, and R. S. Wahby. Unifying compilers for SNARKs, SMT, and more. Cryptology ePrint Archive, Report 2020/1586, 2020.

[53] A. Ozdemir, R. S. Wahby, B. Whitehat, and D. Boneh. Scaling verifiable computation using efficient set accumulators. In *Proceedings of the USENIX Security Symposium*, 2020.

[54] B. Parno, J. Howell, C. Gentry, and M. Raykova. Pinocchio: Nearly practical verifiable computation. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2013.

[55] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, et al. Scikit-learn: Machine learning in python. *Journal of machine learning research*, 12(85), 2011.

[56] F. A. Potra and S. J. Wright. Interior-point methods. *Journal of computational and applied mathematics*, 124(1-2), 2000.

[57] M. V. Ramana. An exact duality theory for semidefinite programming and its complexity implications. *Mathematical Programming*, 77(1), 1997.

[58] J. D. Romano, T. T. Le, W. La Cava, J. T. Gregg, D. J. Goldberg, P. Chakraborty, N. L. Ray, D. Himmelstein, W. Fu, and J. H. Moore. PMLB v1.0: an open-source dataset collection for benchmarking machine learning methods. *Bioinformatics*, 38(3):878–880, 10 2021.

[59] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 1988.

[60] E. Roth, K. Newatia, Y. Ma, K. Zhong, S. Angel, and A. Haeberlen. Mycelium: Large-scale distributed graph queries with differential privacy. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2021.

[61] E. Roth, D. Noble, B. H. Falk, and A. Haeberlen. Honeycrisp: Large-scale differentially private aggregation without a trusted core. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2019.

[62] E. Roth, H. Zhang, A. Haeberlen, and B. C. Pierce. Orchard: Differentially private analytics at scale. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020.

[63] S. Setty. Spartan: Efficient and general-purpose zksnarks without trusted setup. In *Proceedings of the International Cryptology Conference (CRYPTO)*, 2020.

[64] S. Setty, S. Angel, T. Gupta, and J. Lee. Proving the correct execution of concurrent services in zero-knowledge. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[65] S. Setty, B. Braun, V. Vu, A. J. Blumberg, B. Parno, and M. Walfish. Resolving the conflict between generality and plausibility in verified computation. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, 2013.

[66] 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 *Proceedings of the USENIX Security Symposium*, 2012.

[67] G. Stewart, S. Merten, and L. Leland. Snårkl: Somewhat practical, pretty much declarative verifiable computing in haskell. In *Proceedings of the International Symposium on Practical Aspects of Declarative Languages*, 2018.

[68] S. Vaswani, F. Bach, and M. Schmidt. Fast and faster convergence of sgd for over-parameterized models and an accelerated perceptron. In *The 22nd International Conference on Artificial Intelligence and Statistics*, pages 1195–1204, 2019.

[69] R. S. Wahby, S. T. Setty, Z. Ren, A. J. Blumberg, and M. Walfish. Efficient ram and control flow in verifiable outsourced computation. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2015.

[70] R. S. Wahby, I. Tzialla, abhi shelat, J. Thaler, and M. Walfish. Doubly-efficient zkSNARKs without trusted setup. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2018.

[71] C. Weng, K. Yang, X. Xie, J. Katz, and X. Wang. Mystique: Efficient conversions for zero-knowledge proofs with applications to machine learning. In *Proceedings of the USENIX Security Symposium*, 2021.

[72] H. Wu, W. Zheng, A. Chiesa, R. A. Popa, and I. Stoica. Dizk: A distributed zero knowledge proof system. In *Proceedings of the USENIX Security Symposium*, 2018.

[73] T. Xie, J. Zhang, Y. Zhang, C. Papamanthou, and D. Song. Libra: Succinct zero-knowledge proofs with optimal prover computation. In *Proceedings of the International Cryptology Conference (CRYPTO)*, 2019.

[74] Y. Xie, X. Wu, and R. Ward. Linear convergence of adaptive stochastic gradient descent. In *International Conference on Artificial Intelligence and Statistics*, pages 1475–1485, 2020.

[75] J. Zhang, T. Xie, Y. Zhang, and D. Song. Transparent polynomial delegation and its applications to zero knowledge proof. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2020.

[76] L. Zhao, Q. Wang, C. Wang, Q. Li, C. Shen, and B. Feng. Veriml: Enabling integrity assurances and fair payments for machine learning as a service. *IEEE Transactions on Parallel and Distributed Systems*, 2021.

Appendix A SDP and SGD transformations

A.1 Semidefinite programming

We now discuss Otti's operation on SDP problems. Below is an example given in its primal form.

$$\begin{aligned}
\text{minimize} \quad & \begin{pmatrix} -0.9915 & 0.6539 & -0.6403 \\ 0.6539 & 0.9379 & -0.1421 \\ -0.6403 & -0.1421 & 0.3080 \end{pmatrix} \bullet X \\
\text{subject to} \quad & \begin{pmatrix} 0.3518 & -0.3833 & 0.3136 \\ -0.3834 & 0.8570 & -0.5891 \\ 0.3136 & -0.5891 & 0.9714 \end{pmatrix} \bullet X = 3.9889 \\
& \begin{pmatrix} -0.4945 & -0.6208 & 0.2038 \\ -0.6208 & 0.2158 & -0.0972 \\ 0.2038 & -0.0972 & -0.4218 \end{pmatrix} \bullet X = -1.0823 \\
& X \succeq 0 \\
\text{initial } X_0 = & \begin{pmatrix} 1.7803 & -0.0036 & 0.7281 \\ -0.0036 & 1.2237 & -0.0536 \\ 0.7281 & -0.0536 & 1.8439 \end{pmatrix}
\end{aligned}$$

We use the `__SDP(·)` intrinsic to indicate that this is an SDP problem and take input. We only need the primal formulation in the SDP instance, as the solver we use solves the dual problem at the same time.

The developer may get an SDP problem from an SDPA file, or may generate it themselves. The developer should then use Otti to generate the C code for the primal problem, as seen in Figure 14, and the `check_sdp` function, as seen in Figure 15. This check will be different for SDP instances for different size matrices (n) and different numbers of optimality constraints (m).

The parameters to the checker include whether the instance is feasible (a boolean), the original parts of the problem (C, A, b), the primal and dual solutions (X_0 or X, y), and the supporting witness variables (X_L, S_L) described in Section 5.2. As in LP, this nondeterministic checker is the only code Otti actually compiles to R1CS.

A.2 Gradient Descent

We now give an example of a SGD problem formulation.

$$\begin{aligned}
\text{classification} \quad & \begin{pmatrix} -1 & -1 & 1 & -1 & 1 \end{pmatrix} \\
\text{dataset} \quad & \begin{pmatrix} 157.0 & 1.0 & 0.69 \\ 268.0 & 10.0 & 2.40 \\ 209.0 & 2.0 & 1.10 \\ 134.0 & 0.1 & 0.10 \\ 21.0 & 0.1 & 0.10 \end{pmatrix}
\end{aligned}$$

The above are a few data points from the PMLB lupus dataset [51, 58]. Otti can take as input any file from the PMLB dataset (including the Lupus one described above) and generate the required C code and nondeterministic checker. Otti can of course be extended to parse any binary classification dataset the user would like. The generated Otti input file and nondeterministic checker for the above small example is given Figure 16.

```
#include "fxpt.h"
int main() {
  fp64 X0 = __exist(),
       ...
       X8 = __exist(),
       Y0 = __exist(),
       Y1 = __exist(),
       XL0 = __exist(),
       ...
       XL8 = __exist(),
       SL0 = __exist(),
       ...
       SL8 = __exist();
  int feasible = __exist();
  __SDP(
    3, 2, // n, m
    // C
    -0.9915, 0.6539, -0.6403,
    0.6539, 0.9379, -0.1421,
    -0.6403, -0.1421, 0.3080,
    // X_0
    1.7803, -0.0036, 0.7281,
    -0.0036, 1.2237, -0.0536,
    0.7281, -0.0536, 1.8439
    // A_0
    0.3518, -0.3833, 0.3136,
    -0.3834, 0.8570, -0.5891,
    0.3136, -0.5891, 0.9714,
    // A_1
    -0.4945, -0.6208, 0.2038,
    -0.6208, 0.2158, -0.0972,
    0.2038, -0.0972, -0.4218,
    3.9889, -1.0823 // b
  );
  return __check(check_sdp( <parameter list:
      feasible, C, A, b, X, y, XL, SL> ));
}
```

FIGURE 14—Automatically generated code for SDP instance.

Appendix B SDP infeasibility evaluation

We can detect proofs of infeasibility for SDP instances, as discussed in Section 5.2. Recall that Otti uses disjunction proofs (either the prover found the optimal solution *or* the instance with provided starting points was infeasible), so the R1CS representation (list in figure 8) remains the same whether or not the instance is satisfiable since both sides of the disjunction must be included anyway.

In addition to our "feasible" evaluation in the main paper, we evaluate each problem instance on the opposite side of the disjunction. To simulate infeasible instances, we pick a bad X_0 at random from outside the feasible region. In Figure 17, we show the SDP benchmarks for these infeasible instances (next to our feasible data, for comparison). We denote a problem instance with a feasible starting point by *"<name>"*, and it's infeasible starting point counterpart by *"<name>*"*. The difference in timings are mostly attributed to two factors.

```
int check_sdp( <parameter list> ){
  int solved = feasible;
  fp64 dot_b0 = (a0_0*x0)+(a0_1*x1)+...+(a0_8*x8);
  fp64 dot_b1 = (a1_0*x0)+(a1_1*x1)+...+(a1_8*x8);
  if (feasible) {
    // satisfied constraints
    solved = solved && F_EQUAL(dot_b0,b0);
    solved = solved && F_EQUAL(dot_b1,b1);
    // S
    fp64 s0 = c0-((a0_0*y0)+(a1_0*y1));
    ...
    fp64 s8 = c8-((a0_8*y0)+(a1_8*y1));
    fp64 gap = (s0*x0) + (s1*x1) + ... + (s8*x8);
    // strong duality
    solved = solved && (F_EQUAL(gap,0.0));
    // lower triangular for XL
    solved = solved && (F_EQUAL(xl1,0.0));
    solved = solved && (F_EQUAL(xl2,0.0));
    solved = solved && (F_EQUAL(xl5,0.0));
    // lower triangular for SL
    solved = solved && (F_EQUAL(sl1,0.0));
    solved = solved && (F_EQUAL(sl2,0.0));
    solved = solved && (F_EQUAL(sl5,0.0));
    // Cholesky decomposition for X (XL*XL^T=X)
    fp64 xr0 = xl0; // XR = XL^T
    ...
    fp64 xr8 = xl8;
    fp64 xm0 = (xl0*xr0)+(xl1*xr3)+(xl2*xr6);
    ...
    fp64 xm8 = (xl6*xr2)+(xl7*xr5)+(xl8*xr8);
    solved = solved && (F_EQUAL(x0,xm0));
    ...
    solved = solved && (F_EQUAL(x8,xm8));
    // Cholesky decomposition for S
    fp64 sr0 = sl0;
    ...
    solved = solved && (F_EQUAL(s8,sm8));
  } else {
    // unsatisfied constraints
    solved = solved || !F_EQUAL(dot_b0,b0);
    solved = solved || !F_EQUAL(dot_b1,b1);
    // X_0 not positive definite
    fp64 l0_0 = (sl0*xl0);
    fp64 l0_1 = (sl0*xl3);
    fp64 l0_2 = (sl0*xl6);
    fp64 r0_0 = (x0*xl0)+(x1*xl3)+(x2*xl6);
    fp64 r0_1 = (x3*xl0)+(x4*xl3)+(x5*xl6);
    fp64 r0_2 = (x6*xl0)+(x7*xl3)+(x8*xl6);
    solved = solved || (F_EQUAL(l0_0,r0_0) &&
        F_EQUAL(l0_1,r0_1) && F_EQUAL(l0_2,r0_2)
        && (sl0<0.01));
  }
  return solved;
}
```

FIGURE 15—Automatically generated check for SDP function for size $n = 3$ and $m = 2$.

```
#include "fxpt.h"

int grad_check(
    int int w0, int w1, int w2,
    fp64 x0, fp64 x1, fp64 x2,
    int y) {
    return y*(w0 * x0 + w1 * x1 + w2 * x2) >= 1;
}

int main() {
    int W0, W1, W2 = __exist();
    __SGD_train(2, 5,
      157.0, 1.0, 0.69, 268.0, 10.0,
      2.40, 209.0, 2.0, 1.10, 134.0,
      0.1, 0.1, 21.0, 0.1, 0.1,
      1, -1, 1, -1, 1);
    return __check(
        // Check point 1
        grad_check(W0,W1,W2,157.0,1.0,0.69,1),
        // Check point 2
        grad_check(W0,W1,W2,268.0,10.0,2.40,-1),
        // Check point 3
        grad_check(W0,W1,W2,209.0,2.0,1.10,1),
        // Check point 4
        grad_check(W0,W1,W2,134.0,0.1,0.1,-1),
        // Check point 5
        grad_check(W0,W1,W2,21.0,0.1,0.1,1));
);
```

FIGURE 16—Automatically generated nondeterministic checker for SGD instance

First, the solving time for infeasible instances is near 0. This is because before Otti runs the solver, it performs a quick check that X_0 is feasible. If this check doesn't pass (as in the case of our infeasible instances), then an infeasibility proof is produced and CSDP never runs. Second, Spartan's multi scalar multiplication not being constant time and different witnesses yield different proving times.

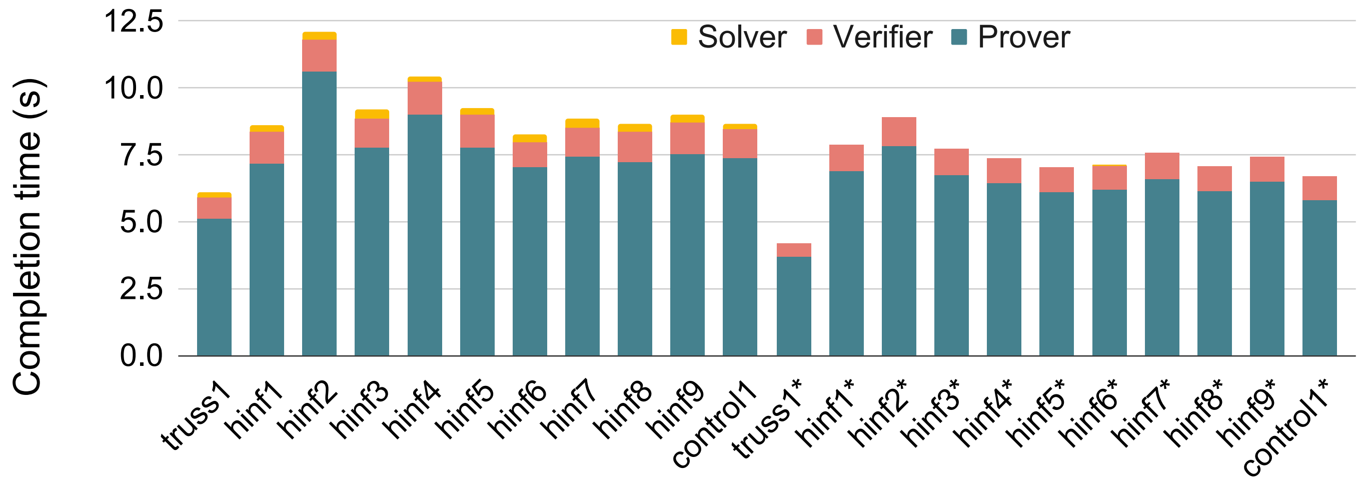

FIGURE 17—Runtime attribution of SDP benchmarks in Otti. Labels with an asterisk (*) represent infeasible instances.