Efficient Representation of Numerical Optimization Problems for SNARKs

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 SNARKs that provides language-level 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 constraints satisfiability (R1CS) instances that express a succinct transformation of those programs whose correct execution implies the optimality of the solution to the original optimization problem. Our experimental evaluation on real numerical solver benchmarks used by commercial LP, SDP, and SGD solvers shows that Otti, instantiated with the Spartan proof system, can prove the optimality of solutions in as little as 300 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. 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 generally 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 ideal allocation of medications to shelters given a set of constraints) 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) [8–10, 12, 13, 19–21, 30, 32, 38, 44, 49, 51, 56, 58, 60] 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 trusted means, the solver can generate a cryptographic proof that convinces clients that the solution is optimal without revealing the sensitive inputs. Alternatively, the inputs could be *sunset*: they could be given as commitments today, and then decommited at a future time, either set a priori or on demand in response to a legal challenge. In either case, designing efficient zkSNARKs for optimization problems could provide the needed technical mechanisms.

Unfortunately, existing general-purpose 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 [42] 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 non-deterministic 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 *non-deterministic checker* is a special program that does exactly this: the non-deterministic checker is derived from the original program and takes the solution to the original program as a non-deterministic 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 non-deterministic checkers from certificates. Otti uses the certificates of optimality which are available for many optimization problems [37] to build non-deterministic checkers. Verifying these certificates are correct is equivalent to verifying that the solution to the optimization problem is optimal. Crucially, the non-deterministic 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 the number of loop iterations, and avoid representing the complex logic of the solver itself.

Probabilistic certificates of optimality. Many important classes of optimization problems (e.g., some instances where SGD is applied) 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 non-deterministic checkers. Our approach here is guided by a conjectural meta-theorem that asserts that whenever there is a rapid convergence result for SGD, one can extract a PCO and therefore also a non-deterministic checker.

Automatic generation of checkers. Otti takes as input C programs extended with an API for optimization problems similar to those of existing solvers (e.g., Otti's API for LP is inspired by Google's Glop linear solver [3]). Otti then automatically extracts and compiles the non-deterministic checker for the optimization program defined with this API. This allows developers to be oblivious to the notions of certificates of optimality. Since the theory behind these certificates is complicated for optimization problems beyond LP, this is essential for Otti to be usable.

Leverage numerical optimization solvers. Otti leverages existing fast numerical solvers (e.g., lpsolve [4]) to find the solution to the underlying optimization problems in the provided C programs (and our API) and supply these solutions to the non-deterministic checkers that Otti generates.

We have implemented a prototype of Otti on top of the CirC SNARK compiler [42] 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 [49] in times ranging from 200 ms to 30 mins (for thousands to around a quarter of a billion constraints). On average, proof generation for LP and SDP is 30–40$\times$ more expensive than finding the solutions themselves using existing solvers; it is two orders of magnitude more expensive for SGD. 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 non-deterministic 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 by producing a proof that reveals no 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 non-deterministic 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 for a given R1CS instance 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 has knowledge of a witness that satisfies the R1CS instance. This argument is complete and computationally 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` can fool the `Verifier` into accepting an invalid proof 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 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 [10, 11, 18, 21, 26, 34–36, 42, 50, 53, 55, 57] written to handle this conversion and even optimize the generated satisfiability instance (i.e., minimize the number of constraints). Over time, these compilers have addded 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 and circuit minimization.

Let's take for example the following C program:

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

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) [48]. SSA allows the expression of mutable computations into immutable equations between versions of variables.

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

For the sake of simplicity we will omit the transformation between the C `int` types and the finite-field numbers in R1CS. In this particular example we will treat them the same, as we can choose reasonable inputs that do not overflow for a sufficiently large prime ($\sim$300 bits) field $\mathbb{F}_p$. We can see that $prod_0, i_0, i_1, i_2, i_3$ can be treated as constants, so R1CS compilers can use standard techniques like constant propagation and algebraic identities to eliminate unnecessary constraints.

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

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:

$$
\begin{aligned}
inv_{prod4} &= x^{p-2} \\
r &= 30 * inv_{prod4}
\end{aligned}
$$

This represents inv_{prod4} in $\log(p)$ R1CS constraints. However, there is a cheaper way to express the inverse if one leverages the nondeterminism supported by R1CS.

2.4 The benefits of nondeterminism

We review the notion of a *nondeterministic check*, which we will employ as a drop-in replacement for a computation that is not efficiently represented in R1CS. Rather than expressing the computation itself, we imagine we are given this result and merely check it. This transformation makes sense when the check is efficient compared to the computation.

Nondeterministic checkers sometimes appear in existing compilers under the term *exogenous computations* and are common for expressing things like bit decomposition (crucial for performing bitwise operations) [52], RAM and remote storage via hash functions [18], among many others. Otti will make extensive use of this functionality, so we give a formal definition for non-deterministic checkers. We start with some preliminaries: 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 $\{0,1\}$.

Now, 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 $\forall x \in X,\ y \in Y,\ \mathcal{C}(x) \mapsto y \Leftrightarrow \mathcal{V}(x,y) = 1$. Conversely, both non-termination and termination with $\mathcal{C}(x) \mapsto y$ 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 $\mathcal{C}$.

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

The benefit of nondeterminism is that it is possible to get a considerable improvement in resource use between computing $\mathcal{C}$ versus $\mathcal{V}$, either in asymptotic terms or in absolute terms. Hence the trivial $\mathcal{V}$ is not practically useful.

We demonstrate the use of nondeterminism with the example from the previous section. We take $\mathcal{C}(x) = 30/x$ and

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

We can test a few numbers x, y and see that $\mathcal{C}(x) \mapsto y \Leftrightarrow \mathcal{V}(x, y) = 1$ holds for all of them, including the edge case $a = 0$. We know $\mathcal{V}(0, y)$ can never be 1, as that would imply there exists a y such that $0 * y = 30$, 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 a constraint $r * prod_4 = 30$. This single multiplication is one 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:

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

We can see that $prod_4 = prod_3 + 3$ gets "wrapped into" a multiplication constraint. In general, addition and multiplication by a constant are basically free in R1CS constraints, as they can be wrapped in like this. Multiplications of two variables are a single constraint.

The solution vector $\vec{z}$ will be of the form $(io, 1, w) = (a, 1, prod_2, prod_3, r)$, 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:

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

A vector $\vec{z} = (3, 1, 9, 27, 1)$ satisfies these constraints.

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 $f\colon \mathbb{R}^n \to \mathbb{R}$ 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:

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

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 procedures, including deep learning/ neural networks, support vector machines, 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 [18], hash sets [50], accumulators [43], or sorting networks [10, 55]. 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. All existing SNARK compilers support only a subset of the functionality of traditional languages. Meanwhile, existing implementations of solvers use vectorized instruction sets, GPU extensions, external libraries, and other efficient modifications. A developer who wishes to use SNARK compilers is currently forced to write their own implementation in the accepted subset of the language, and lose the efficiency of commercial 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 program that checks the optimality of the solution produced by the solver. Crucially, we show how to automatically derive this checker program, 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 similar to Google's glop [3]. In addition, we have implemented parsers for the most common file formats used to specify these problems: for LP, Otti can parse MPS files [1]; for SDP, Otti can parse SDPA files [28]; for SGD, Otti can parse PMLB files [41, 47]. 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.

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 non-deterministic checker. The non-deterministic checker is a C program that verifies the optimality of a purported solution to the problem instance.

Otti then compiles the non-deterministic checker to R1CS

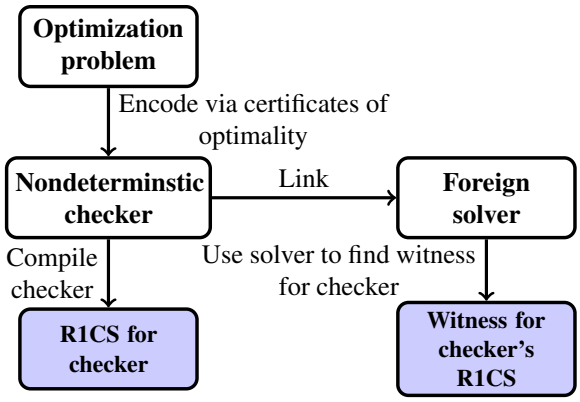


FIGURE 1—High-level workflow of Otti.

using a heavily modified version of the CirC SNARK compiler [42]. 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 non-deterministic 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 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). The R1CS and witness can then be consumed by zkSNARK proof systems that supports R1CS. We use Spartan [49].

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 (the standard form shown in Section 3) to the *dual* formulation, which is also a convex optimization problem. The dual's optimal solution, described by vector $\vec{y}$, is a lower bound on the optimal solution, described by vector $\vec{x}$, of the primal problem. This is referred to as weak duality: $f(\vec{x}) \leq f'(\vec{y})$, where f' is the bjective function for the dual problem. The difference between the optimal primal and dual solutions, $f(\vec{x})$ and $f'(\vec{y})$, 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.

For SGD optimization problems, the certificates we use are different. Instead of a duality gap, we exploit the fact that in many cases of interest there are bounds on the behavior of the stochastic estimates ∇f_i of the gradient ∇f of the objective function near the optimal point. Roughly speaking, at an

optimal point z the estimates satisfy $\nabla f_i(z) = 0$ with high probability and for other points x, $||\nabla f_i(z)|| > \kappa||x - z||$ for some constant κ 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 using a state-of-the-art solver and then proves that at this point z the gradient estimates $\nabla f_i(z)$ (or a representative sampling of them) are close to zero.

Generating proofs of these facts is significantly cheaper than proving the execution of the solver 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 as the solver.

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 standard (primal) form:

$$\begin{aligned} &\text{minimize } \vec{c}^T \cdot \vec{x} \\ &\text{subject to } A \cdot \vec{x} \geq \vec{b} \\ &\vec{x} \geq 0 \end{aligned}$$

The corresponding dual is:

$$\begin{aligned} &\text{maximize } \vec{y}^T \cdot \vec{b} \\ &\text{subject to } A^T \cdot \vec{y} \leq \vec{c} \\ &\vec{y} \geq 0 \end{aligned}$$

The certificate of optimality for an LP problem is as follows:

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

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 ($\vec{x}$) and dual ($\vec{y}$) formulations, and then assigns $\vec{x}$ and $\vec{y}$ to nondeterministic variables in the R1CS instance, proving its satisfiability. By the strong duality theorem, satisfiability of these checks implies that $\vec{x}$ 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:

$$\begin{aligned} &\text{minimize } C \bullet X \\ &\text{subject to } \forall i, A_i \bullet X = b_i \\ &X \succeq 0 \end{aligned}$$

The variables in these problems are semidefinite matrices, rather than real numbers. We use "$X \succeq 0$" to denote that X is a symmetric and positive semidefinite matrix. We use "$C \bullet X$" to denote $\sum_{i=1}^{n} \sum_{j=1}^{n} C_{ij} \cdot X_{ij}$. We say the dimensions of the matrices involved are n^2. Such a primal problem could also be viewed as a dual problem:

$$\begin{aligned} &\text{maximize } \sum_{i=1}^{m} y_i \cdot b_i \\ &\text{subject to } \sum_{i=1}^{m} y_i \cdot A_i + S = C \\ &S \succeq 0 \end{aligned}$$

The certificate looks as follows:

1. Primal feasibility of X $\quad \forall i, A_i \bullet X = b_i$, $X \succeq 0$

2. Dual feasibility of (y, S) $\quad \sum_{i=1}^{m} y_i \cdot A_i + S = C$, $S \succeq 0$

3. Strong duality $\quad C \bullet X - \sum_{i=1}^{m} y_i \cdot b_i = S \bullet X$

Unlike LP, we cannot assert that the primal or dual problem will always obtain their optima or that there will be no duality gap. We additionally require *strict feasibility*: a common optimal value exists if both the primal and dual problems have feasible solutions inside the semidefinite cone [27].

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 non-deterministic 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: if a matrix X factors as LL^T, where L is a real lower triangular matrix, with non-negative diagonal entries, then X is positive semidefinite. This factoring is known as the *Cholesky decomposition* [39]. Otti creates nondeterministic variables for the lower triangular matrix part of both decompositions, XQ and SQ, and the `Prover` supplies these values exogenously. The non-deterministic checker that Otti derives then confirms that these matrices are lower triangular and indeed make up a Cholesky decomposition of each matrix (e.g., $XQ \cdot XQ^T = X$). Non-negative diagonal entries of XQ and SQ are used to confirm that $X \succeq 0$ and $S \succeq 0$.

Solving SDP instances. To solve SDP, Otti uses the `CSDP` [14] library, which implements the interior point algorithm [33]. 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. This can be leveraged by Otti (during solving) to obtain a feasible starting solution to the primal and dual problems [27]. 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 [46]. 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 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. This proof is as follows:

Infeasibility of SDP instance $\quad (\exists i, A_i \bullet X^0 \neq b_i) \vee X^0 \not\succeq 0$

The above check proves infeasibility with respect to a particular starting point X^0 [27]. 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 $X^0 \not\succeq 0$. 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 XQ and SQ. Otti then checks that this eigenvalue/eigenvector pair is valid for the provided X^0.

Finally, 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. 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.

Many loss functions of interest can be written as:

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

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. In this context, it can be very expensive to compute the gradient of the loss, and so instead it is often more feasible to try *stochastic gradient descent*. Stochastic gradient descent (SGD) iteratively takes steps in the direction of ∇f_i; each such direction is a subgradient that provides an unbiased estimate of the actual gradient, i.e., regarding these choices as random we have:

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

This is substantially more efficient than computing the full gradient, although of course in general convergence may be slower. Notwithstanding, a number of recent results show that stochastic gradient descent rapidly converges, especially when used in connection with adaptive stepsize algorithms [16, 17, 24, 40, 54, 59].

The hypotheses for these convergence results provide sufficient control on the loss function to derive certificates. For example, Vaswani-Bach-Schmidt [54] study *growth conditions* on f that guarantee rapid convergence for SGD, even in non-convex settings. Notably, the *strong growth condition* stipulates that:

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

We use $||\cdot||$ to refer to the norm. 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 certificate that a point is optimal can be obtained simply by checking that $||\nabla f_i|| = 0$ for each i. Examples of problems that satisfy this requirement are perceptron classifiers (e.g., linear classifiers). More generally, overparametrized neural networks often satisfy this requirement.

Other hypotheses (e.g., see [59]) are 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; in these cases, Otti can extract a *probabilistic certificate of optimality* 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.

The above is synergistic, as it means that whenever we expect SGD to converge quickly, Otti can produce a certificate. In future work we plan to prove the above conjecture. In this paper, we focus on the strong growth condition.

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. 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

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

and the "square hinge" loss function

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

where $\langle \cdot, \cdot \rangle$ denotes inner product. These functions satisfy the strong growth condition for separable data. These functions

```
#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.

have subgradients that are most easily expressed piecewise, with one part in the $y_i * \langle w, x_i \rangle < 1$ case and one which is always 0 when $y_i * \langle w, x_i \rangle >= 1$. The second defines a region where the derivative of the loss-function can be 0 simultaneously for all data-points; in other words w is optimal when $\forall i, 1 - y_i * \langle w, x_i \rangle = 0$. 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 Appendix A. Below is the LP problem in its primal form.

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

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 non-deterministic 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 non-determnistic variables $X0$ and $X1$. What Otti does next is fully automated.

First, Otti computes the dual formulation of the problem.

```
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.

In mathematical notation, it is:

$$\text{find } \vec{y} \text{ that minimizes } \vec{y}^T \cdot \begin{pmatrix} 14 \\ 0 \\ 2 \end{pmatrix}$$
$$\text{subject to } \begin{pmatrix} 1 & -3 & 1 \\ 2 & 1 & -1 \end{pmatrix} \cdot \vec{y} \leq \begin{pmatrix} 3 & 4 \end{pmatrix}$$
$$\vec{y} \geq 0$$

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:

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

Given the primal, the dual, and the strong duality condition, Otti generates a non-deterministic 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 non-deterministic checker generated by Otti.

The LP non-deterministic checker defined over non-deterministic 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 non-deterministic 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 [42], which is a system written in Haskell that compiles a C subset to existentially quantified circuits. It supports a few different output formats, including the one we use, R1CS. CirC uses a typed version of SMT-LIB [5] as its intermediate representation and the Z3 [23] SMT solver 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

```
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 non-deterministic checker for LP instance given in Figure 2.

prover can call to obtain the satisfying assignment 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 [7], which is a recent standard for specifying R1CS in a portable manner. We also modified the open source implementation of the Spartan zkSNARK library [6] to support zkinterface files as input and to support arbitrary R1CS instances (the original implementation supported only instances with power-of-two-sized matrices).

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.

8 Evaluation

This section answers our animating question: is the use of non-deterministic 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 [2, 29]. 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, and the resulting R1CS produced by Otti.

For SDP, we use SDPLIB [15], 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 [14]. We evaluate infeasible instances in Appendix B. Figure 8 lists these benchmarks, the number of variables, the size of the semidefinite matrices, and the resulting R1CS produced by Otti.

For SGD, we use binary classification problems from the Penn ML Benchmarks (PMLB) dataset [41, 47]. This set of benchmarks is curated specifically for evaluating supervised ML algorithms, and comes from a variety of applications. The datasets we use have binary classes and are linearly separable. Figure 11 lists these benchmarks, the number of datapoints and features in them, and the resulting R1CS produced by Otti by leveraging non-deterministic checkers. Note that `GE1000` is short for the `GAMETES_Epistasis_2_Way_1000` dataset.

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 and SDP solvers which provide no proofs. Our results are therefore overhead over unverifiable solvers, rather than speedup over prior work.

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, and the execution time of the `Prover` and `Verifier` for each problem.

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

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

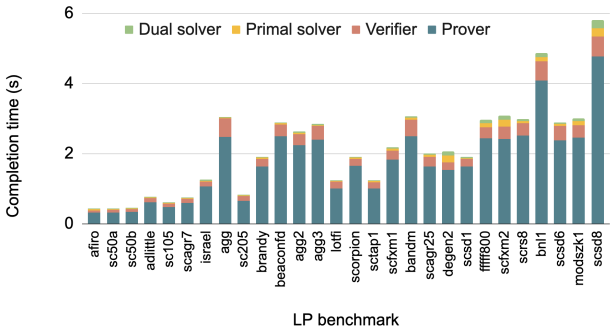


FIGURE 6—Runtime attribution of LP benchmarks in Otti.

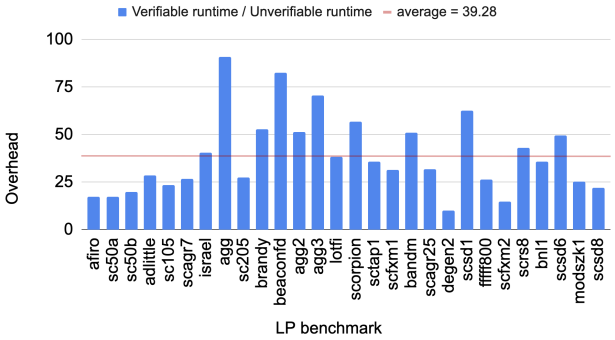


FIGURE 7—Overhead of Otti over baseline for LP benchmarks.

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.

We omit commitments and proof sizes generated by the underlying zKSNARK since they vary widely among schemes and they are orthogonal to Otti.

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).

Instance sizes. Figure 5 reports the name of the different benchmarks we consider, the number of LP variables and linear equations, and the size of the R1CS instance produced by Otti. 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 equations can be defined over tens or hundreds of variables, while others defined over a single variable. Otti is currently able to compile instances with thousands of LP variables and hundreds of equations. For comparison, CirC is unable to compile instances with more than 5 variables and 4 equations on our machine with 200 GB of RAM due to the excessive amount of memory required to compile them. 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. 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

Dataset	SDP equations	SDP matrix size	R1CS constraints
truss1	6	13	3,007,933
hinf1	13	14	4,703,942
hinf2	13	16	6,536,398
hinf3	13	16	6,536,398
hinf4	13	16	6,536,398
hinf5	13	16	6,536,398
hinf6	13	16	6,536,398
hinf7	13	16	6,536,398
hinf8	13	16	6,536,398
hinf9	13	16	6,536,398
control1	21	15	6,968,254

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

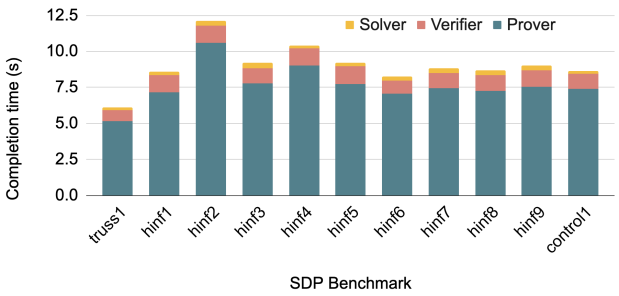


FIGURE 9—Runtime attribution of SDP benchmarks in Otti.

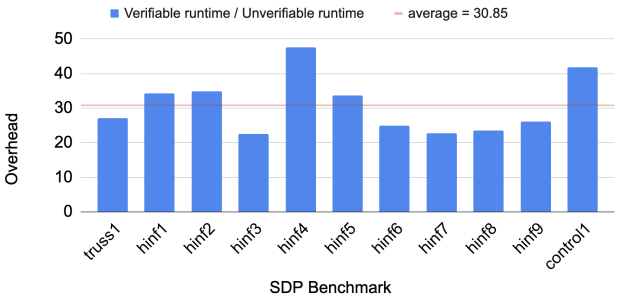


FIGURE 10—Overhead of Otti over baseline for SDP benchmarks.

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 toy examples with overheads of at least 3 orders of magnitude. For example, Hyrax [56] and Libra [58] 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 [9] and Fractal [20], 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.

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 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 non-deterministic checker; (2) the fact that Otti compiles both the feasible and infeasible branches; (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 based on CirC is unable to compile SDP checks. 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.

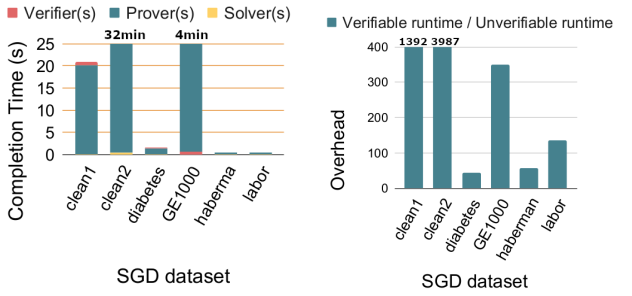
Figure 9 shows the results. Otti takes more time on SDP problems than LP, spending a few seconds to solve and generate each proof. However, the overhead relative to the baseline, given in Figure 10, is similar to the LP experiments: Otti's prover is on average 30× more expensive than the baseline. In 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).

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` [45], to train the SGD classifier and get the optimal fitted plane which is then checked by Otti's checkers on all data points.

Dataset	Data points	Features	R1CS constraints
clean1	476	168	10,605,356
clean2	6,598	168	146,819,367
diabetes	768	8	828,106
GE1000	1,600	1,000	211,918,394
haberman	306	3	135,974
labor	57	16	135,180

FIGURE 11—Number of data points, features (for each data point), and R1CS constraints generated for the PMLB datasets.



(a) Runtime

(b) Overheard relative to the baseline

FIGURE 12—Runtime cost of SGD training in Otti.

Instance sizes. Figure 11 reports the name of the different classification datasets, as well as their sizes and the resulting number of R1CS constraints produced by Otti. In contrast to the LP and SDP datasets, since Otti generates a check for every one of the input points, the resulting R1CS instances are significantly larger. One possible tradeoff is to let the `Verifier` provide a random seed, and check the subgradient at only a pseudorandom subset of the points at the cost of introducing additional soundness error. Such a relaxed protocol could be used to bound the number of misclassified points; under stronger hypotheses about the structure of the data, it might be possible to obtain better soundness bounds.

Runtime. Figure 12a 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 1–2 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 quite variable overhead, ranging from 1 to 4 orders of magnitude. Figure 12b 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 for real classifier and datasets.

9 Related Work

The idea of using strong duality to verify the output of LP programs has also been explored in the past [22, 31]. For example, Goemans et al. [31] 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. [22] 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 SNARK instances, runs on real problems from the `netlib` library, and achieves practical performance: an LP instance with 398 equations and 2,750 variables takes under 6 seconds.

Beyond the above, as far as we know Otti is the only system to leverage certificates for SDP and SGD to generate fast proofs, as well as automatically extracting these certificates from standard problem formulations and file formats.

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 and further work. For one thing, it would be extremely useful to expand the universe of optimization problems for which Otti applies. The SGD examples we considered required strong hypotheses. While the strong growth condition applies to many examples of interest, it is also too stringent to accomodate many 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 certificates is needed. We believe that a deeper understanding of our conjectured "meta-theorem" about the connection between certificates and convergence results will lead to improvements in this direction.

More generally, Otti's approach of aggressively using non-deterministic 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 [18, 21, 34–36, 50, 51, 53] 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 [18, 21, 25, 34], or expressing threads and concurrency [50]. However, one of the lessons of Otti is that these techniques should be more widely used.

Finally, a pressing issue when using non-deterministic

checkers is the question of correctness. We assumed that a solution was optimal if it passed the non-deterministic checker. However, how does one know whether the non-deterministic 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 [26]).

11 Conclusion

This paper introduces Otti, a compiler for SNARKs that natively supports a wide class of optimization problems including linear programming (LP), semidefinite programming (SDP), and many problems amenable to stochastic gradient descent (SGD). 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 non-deterministic 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 machine learning problems with real datasets.

Acknowledgments

We thank Fraser Brown, Patrick Cousot, Justin Thaler, Riad Wahby, and Mike Walfish for helpful conversations related to this work. We also thank Alex Ozdemir for his help with the CirC SNARK compiler. 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] Mps file format. `http://plato.asu.edu/cplex_mps.pdf`.

[2] Lp/data index. `https://ampl.com/netlib/lp/data/`, 2013.

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

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

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

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

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

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

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

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

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

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

[17] L. Bottou. Stochastic gradient descent tricks. In G. Montavon, G. B. Orr, and K. Müller, editors, *Neural Networks: Tricks of the Trade - Second Edition*, volume 7700 of *Lecture Notes in Computer Science*, pages 421–436. Springer, 2012.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] S. de Hoogh, B. Schoenmakers, and M. Veeningen. Certificate validation in secure computation and its use in verifiable linear programming. In *International Conference on Cryptology in Africa*, pages 265–284. Springer, 2016.

[23] 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.

[24] J. Duchi, E. Hazan, and Y. Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159, 2011.

[25] 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.

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

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

[28] 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.

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

[30] 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)*. Springer, 2013.

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

[32] 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.

[33] C. Helmberg, F. Rendl, R. J. Vanderbei, and H. Wolkowicz. An interior-point method for semidefinite programming. *SIAM Journal on optimization*, 6(2):342–361, 1996.

[34] 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.

[35] 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.

[36] A. Kosba, Z. Zhao, A. Miller, Y. Qian, H. Chan, C. PAPAMAN-THOU, R. Pass, S. ABHI, and E. SHI. C∅ c∅: a framework for building composable zero-knowledge proofs. *Cryptology ePrint Archive, Report 2015/1093*, 2015.

[37] 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.

[38] 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.

[39] C. B. Moler and G. W. Stewart. On the householder-fox algorithm for decomposing a projection. *Journal of Computational Physics*, 28(1):82–91, 1978.

[40] Y. E. Nesterov. Smooth minimization of non-smooth functions. *Math. Program.*, 103(1):127–152, 2005.

[41] 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.

[42] A. Ozdemir, F. Brown, and R. S. Wahby. Unifying compilers for snarks, smt, and more. Cryptology ePrint Archive, Report 2020/1586, 2020.

[43] 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.

[44] 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.

[45] 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(Oct):2825–2830, 2011.

[46] F. A. Potra and S. J. Wright. Interior-point methods. *Journal of computational and applied mathematics*, 124(1-2):281–302, 2000.

[47] 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. *arXiv preprint arXiv:2012.00058v2*, 2021.

[48] 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.

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

[50] 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.

[51] 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.

[52] 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.

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

[54] 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. PMLR, 2019.

[55] 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.

[56] 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.

[57] 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.

[58] 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.

[59] 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. PMLR, 2020.

[60] 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.

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.

$$\text{minimize} \begin{pmatrix} -0.9915 & 0.6539 & -0.6403 \\ 0.6539 & 0.9379 & -0.1421 \\ -0.6403 & -0.1421 & 0.3080 \end{pmatrix} \bullet X$$

$$\text{subject to} \begin{pmatrix} 0.3518 & -0.3833 & 0.3136 \\ -0.3834 & 0.8570 & -0.5891 \\ 0.3136 & -0.5891 & 0.9714 \end{pmatrix} \bullet X = 3.9889$$

$$\begin{pmatrix} -0.4945 & -0.6208 & 0.2038 \\ -0.6208 & 0.2158 & -0.0972 \\ 0.2038 & -0.0972 & -0.4218 \end{pmatrix} \bullet X = -1.0823$$

$$X \succeq 0$$

$$\text{initial } X^0 = \begin{pmatrix} 1.7803 & -0.0036 & 0.7281 \\ -0.0036 & 1.2237 & -0.0536 \\ 0.7281 & -0.0536 & 1.8439 \end{pmatrix}$$

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 the provided `sdp_meta.py` file to generate the C code for the primal problem, as seen in Figure 13, and the `check_sdp` function, as seen in Figure 14. 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 (XQ, SQ) described in Section 5.2. As in LP, this non-deterministic checker is the only code Otti actually compiles to R1CS.

A.2 Gradient Descent

We now give an example of a SGD problem formulation.

$$\text{classification} \begin{pmatrix} -1 & -1 & 1 & -1 & 1 \end{pmatrix}$$

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

The above are a few data points from the PMLB lupus dataset [41, 47]. Otti can take as input any file from the PMLB dataset (including the Lupus one described above) and generate the required C code and non-deterministic checker. Otti can of course be extended to parse any binary classification dataset the user would like. The generated Otti input file and non-deterministic checker for the above small example is given Figure 15.

```
#include "fxpt.h"
int main() {
  fp64 X0 = __exist(),
       ...
       X8 = __exist(),
       Y0 = __exist(),
       Y1 = __exist(),
       XQ0 = __exist(),
       ...
       XQ8 = __exist(),
       SQ0 = __exist(),
       ...
       SQ8 = __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, XQ, SQ> ));
}
```

FIGURE 13—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 16, 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 XQ
    solved = solved && (F_EQUAL(xq1,0.0));
    solved = solved && (F_EQUAL(xq2,0.0));
    solved = solved && (F_EQUAL(xq5,0.0));
    // lower triangular for SQ
    solved = solved && (F_EQUAL(sq1,0.0));
    solved = solved && (F_EQUAL(sq2,0.0));
    solved = solved && (F_EQUAL(sq5,0.0));
    // Cholesky decomposition for X (XQ*XQ^T=X)
    fp64 xr0 = xq0; // XR = XQ^T
    ...
    fp64 xr8 = xq8;
    fp64 xm0 = (xq0*xr0)+(xq1*xr3)+(xq2*xr6);
    ...
    fp64 xm8 = (xq6*xr2)+(xq7*xr5)+(xq8*xr8);
    solved = solved && (F_EQUAL(x0,xm0));
    ...
    solved = solved && (F_EQUAL(x8,xm8));
    // Cholesky decomposition for S
    fp64 sr0 = sq0;
    ...
    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 = (sq0*xq0);
    fp64 l0_1 = (sq0*xq3);
    fp64 l0_2 = (sq0*xq6);
    fp64 r0_0 = (x0*xq0)+(x1*xq3)+(x2*xq6);
    fp64 r0_1 = (x3*xq0)+(x4*xq3)+(x5*xq6);
    fp64 r0_2 = (x6*xq0)+(x7*xq3)+(x8*xq6);
    solved = solved || (F_EQUAL(l0_0,r0_0) &&
        F_EQUAL(l0_1,r0_1) && F_EQUAL(l0_2,r0_2)
        && (sq0<0.01));
  }
  return solved;
}
```

FIGURE 14—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 15—Automatically generated non-deterministic checker for SGD instance

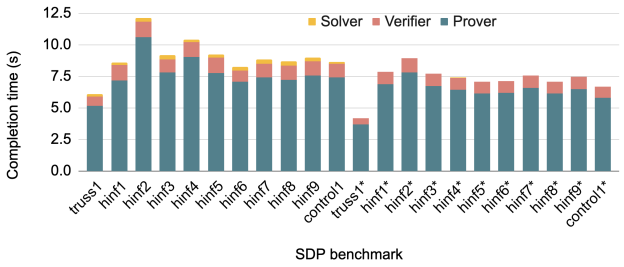


FIGURE 16—Runtime attribution of SDP benchmarks in Otti. Labels with an asterisk (*) represent infeasible instances.

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.