CirC: Compiler infrastructure for proof systems, software verification, and more

Alex Ozdemir Fraser Brown Riad S. Wahby
aozdemir@cs.stanford.edu, mlfbrown@stanford.edu, rsw@cs.stanford.edu

***Abstract*—Cryptographic tools like proof systems, multi-party computation, and fully homomorphic encryption are usually applied to computations expressed as systems of arithmetic constraints. In practice, this means that these applications rely on compilers from high-level programming languages (like C) to such constraints. This compilation task is challenging, but not entirely new: the software verification community has a rich literature on compiling programs to logical constraints (like SAT or SMT). In this work, we show that building shared compiler infrastructure for compiling to constraint representations is *possible*, because these representations share a common abstraction: stateless, non-uniform, non-deterministic computations that we call *existentially quantified circuits*, or EQCs. Moreover, we show that this shared infrastructure is *useful*, because it allows compilers for proof systems to benefit from decades of work on constraint compilation techniques for software verification.**

To make our approach concrete we create CirC, an infrastructure for building compilers to EQCs. CirC makes it easy to compile to new EQCs: we build support for three, R1CS (commonly used for proof systems), SMT (commonly used for verification and bug-finding), and ILP (commonly used for optimization), in ≈2000 LOC. It's also easy to extend CirC to support new source languages: we build a feature-complete compiler for a cryptographic language in one week and ≈900 LOC, whereas the reference compiler for the same language took years to write, comprises ≈24000 LOC, and produces worse-performing output than our compiler. Finally, CirC enables novel applications that combine multiple EQCs. For example, we build the first pipeline that (1) automatically identifies bugs in programs, then (2) automatically constructs cryptographic proofs of the bugs' existence.

I. Introduction

Cryptographic proof systems provide short, private, and efficiently verifiable proofs that a prover knows a witness W that satisfies a relation $R(X, W)$ for public input X. However, to use these proofs, R is usually written as a *rank-1 constraint system (R1CS)*: a structured system of equations over a large finite field (§II-B). This has motivated substantial work on compiling from high-level programs to R1CS in both academia [1], [2], [3], [4], [5] and industry [6], [7], [8].

Other recent cryptosystems give rise to similar compilation problems; secure multi-party computation (MPC) and fully homomorphic encryption (FHE) support different kinds of private computation—and both require the computation be expressed as an arithmetic or boolean circuit. But compiling to constraints (equivalently, circuits) is neither new nor cryptography-specific: the programming languages and formal methods communities have a long tradition of translating programs to logical constraints (e.g., Satisfiability Modulo Theories [9] (SMT) formulas) to verify properties [10], [11], [12], [13], [14], [15], synthesize new programs [16], [17], [18], and more. Further afield, researchers also compile to integer linear programs [19], [20]: a kind of constraint system used for optimization problems.

Compilers to constraints are crucial in all of these applications, but they are hard to build—for example, Torlak and Bodik identify compilers to SMT as "the most difficult aspect of creating solver-aided tools," taking "years to develop" [14]. As a result, communities that rely on constraint compilers have poured enormous effort into building them (§II-E). Unfortunately, there has been little cross-pollination between communities, and duplicated efforts within communities. Thus, our animating question: *is it possible to create shared infrastructure for building constraint compilers that is useful across such disparate applications?* In this paper, we show that the answer is *yes*. Thus, newer applications like proof systems can take advantage of decades of insights from the compilers and verification communities.

To start, we observe that shared infrastructure is possible *in principle*, because all the constraint representations discussed above can be viewed as instances of the same abstraction: a class of non-deterministic execution substrates that we call existentially quantified circuits, or EQCs.[1] EQCs have two main features that differentiate them from CPUs (the targets of traditional compilers). First, EQCs are stateless—they contain no mutable variables, control flow, memory, or storage. Second, they admit non-determinism in the form of existentially quantified variables. For example, an R1CS instance is an EQC that is "executed" by a cryptographic proof system. Since R1CS instances are systems of equations, they are stateless and free of control flow. Moreover, cryptographic proof systems give the witness existential semantics, proving the statement: $\exists W.\ R(X, W)$.

By leveraging the EQC abstraction, we show that shared infrastructure for compiling to constraints is possible—*and useful*—in practice, for three reasons. First, the process of compiling from a high-level language to an EQC is similar, even for very different EQCs. To compile, say, C to SMT, there are well-known procedures: explore all paths through the program (unroll loops, consider all branches), guarding all state modifications by the condition under which the current path is taken [15]. That same procedure is also necessary when compiling C to boolean circuits for multi-party computation [21], or to arithmetic constraints for proof systems [22], [5]. Since this approach is largely independent of the application, sharing compilation infrastructure avoids duplicated effort. We describe our language-agnostic machinery for turning programs into circuits in Section III.

[1] Note that EQCs *do not* capture digital circuits, which are stateful and deterministic; thus, we do not consider them in this work. See Section II-E.

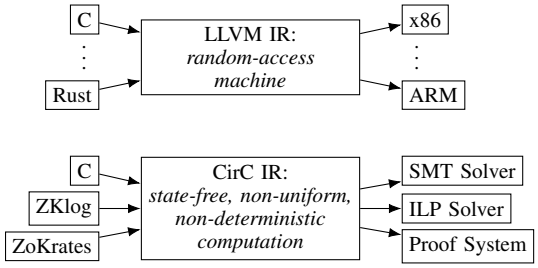

Fig. 1. LLVM uses a random-access machine abstraction to make it easy for new front-ends to target CPUs. CirC uses a non-uniform non-deterministic state-free abstraction to make it easy to target EQCs.

Second, EQCs have performance characteristics that are different from those of processors, but similar to those of other EQCs. As a result, shared EQC infrastructure can support shared optimizations, whereas reusing existing infrastructure geared towards CPUs wouldn't make sense. As one example, while CPUs support load and store instructions for memory access, simulating memory in EQCs (which are state-free) is very expensive: there are active lines of research on memory representations and related optimizations for both software verification [23], [24], [25], [26], [27], [28] and proof systems [1], [29], [30], [5], [4], [31]. We show that proof system and software verifier performance both improve under the same memory optimizations (and more) in Section VI.

Finally, shared compiler infrastructure unlocks benefits with few analogs in traditional compilers. In a traditional compiler, each target CPU supported by the compiler does essentially the same thing—it executes code. EQCs, in contrast, often have very different purposes—and shared infrastructure makes it easy to combine those purposes in ways that enable new applications. For example, verification allows users to prove that a program has some property (e.g., "contains no undefined behavior"), while proof systems allow users to prove facts to one another in spite of mutual distrust (e.g., proving "I know my password" without revealing it). Combining these functionalities, we show in Section VII that our work helps analysts find a bug (using expert hints and the SMT back-end) and then prove the existence of that bug without revealing how to trigger it, using a proof pipeline. We also demonstrate other combinations: such as optimizing R1CS using SMT.

To make these benefits concrete, we implement an infrastructure for building compilers to EQCs, which we call CirC. CirC is analogous to—and inspired by—LLVM [32], an infrastructure for compiling programs to machine code. LLVM's key abstraction is its intermediate representation (LLVM IR), which captures the computational model of CPUs: conceptually, LLVM IR is an abstraction of a random-access machine. CirC builds on a different abstraction (Fig. 1): *state-free, non-deterministic, non-uniform computation*, which captures the computational model of EQCs (§II-A). As in LLVM, language designers can add new front-ends that compile to CirC-IR, where CirC performs optimization passes; and they can add back-ends that compile from CirC-IR to a given EQC (e.g., R1CS), allowing them to "run" the resulting executable (e.g., feed the constraints to proof system).

In sum, our contributions are:

- We build CirC: an infrastructure for compiling programs to circuits (§III).
- We demonstrate CirC's extensibility. We extend CirC to the ZoKrates proof language language in a week with ≈900 lines of code, whereas the language's reference compiler was developed over years and comprises ≈24000 lines (§IV).
- We evaluate the accuracy of CirC's SMT output and the performance (size) of its R1CS output. For example, we find that for ZoKrates programs, CirC produces R1CS instances which perform slightly better than those from the language's reference compiler (§V).
- We evaluate the effect of common optimizations on the performance of verification and proof-system applications, finding that some optimizations provide substantial benefit to both (§VI).
- We demonstrate the ease with which CirC allows one to combine back-end functionalities. We use an SMT solver to optimize and analyze R1CS compilation, and use a proof system to prove knowledge of (1) a bug identified by an SMT solver and (2) a high-value input identified by an ILP solver (§VII).

Summarizing our key insights: (1) many subfields rely on the same abstraction, the EQC; (2) compiling to different EQCs uses similar steps, and EQCs have similar performance characteristics, so shared infrastructure makes sense; and (3) with shared infrastructure, different EQCs can be combined in service of new applications.

II. Background and related work

In this section, we start with a slightly more formal definition of EQCs. Then, to set the stage for our evaluation (§V–§VII), we discuss our two example use cases: cryptographic proof systems (§II-B) and automated verification (§II-C). Finally, we describe related work (§II-E).

A. Existentially quantified circuits

We refer to the broad class of non-deterministic execution substrates that this work targets as *existentially quantified circuits* (EQCs). EQCs share three key properties. First, they are *circuit-like*: they comprise sets of *wires* taking values from some domain (e.g., bits for a boolean circuit) and *constraints* that express relationships among wire values (e.g., an AND gate represents the constraint $C = A \wedge B$).

Second, EQCs are *state free*: unlike variables in a computer program or registers in a CPU, wire values in an EQC *do not change* during execution. In a boolean circuit, for example, each gate's output is determined by its inputs, which are either the outputs of other gates or input wires.

Third, EQCs have two kinds of inputs: explicit inputs, i.e., arguments supplied at the start of execution, and existentially-quantified inputs, which may take any value consistent with the explicit input values and the set of constraints. Consider the trivial EQC $\exists B.A \oplus B = 0$, where A is an explicit input and $\oplus$ is boolean XOR: when A is true, B must be true. In the language of formal logic, EQCs correspond to *quantifier-free first-order formulas*.

In complexity-theoretic terms, we say that EQCs capture *non-deterministic, non-uniform* computation [33, Ch. 6]. Their non-determinism stems from the existentially quantified inputs

whose values are, in principle, "guessed" by the execution substrate. Their non-uniformity reflects the fact that a circuit of a given size encodes a computation for a fixed-size input; thus, for a given computation, different input lengths entail distinct circuits.

B. Cryptographic proof systems

Probabilistic proof systems are powerful cryptographic tools whose applications include verifying that outsourced computations are executed correctly [2], [34], [31], implementing private cryptocurrency transactions [35], [36], and defending against hardware back-doors [37], [38]. In this section, we describe the class of probabilistic proof systems CirC targets, focusing on their computational model. Readers should consult surveys [39], [40] for additional details.

At a high level, a probabilistic proof system is a cryptographic protocol between two parties, a *prover* $\mathcal{P}$ and a *verifier* $\mathcal{V}$, whereby $\mathcal{P}$ produces a short proof that convinces $\mathcal{V}$ that $\exists w.y = \Psi(x, w)$, for Ψ a computation that takes input x and witness w and returns output y. Several lines of work [1], [2], [29], [30], [41], [42], [3], [4] instantiate end-to-end built systems. Two key features of these systems are *succinctness*—$\mathcal{P}$'s proof is small, as is $\mathcal{V}$'s work checking it—and *zero knowledge*—an accepting proof reveals nothing about the witness w other than the truth of $y = \Psi(x, w)$.

These systems comprise a *compilation stage* and a *proving stage*. The proving stage applies complexity-theoretic and cryptographic machinery to the compilation stage's output, allowing $\mathcal{P}$ to generate a proof and $\mathcal{V}$ to verify it. The compilation stage, our focus in this work, takes a source program Ψ (written, say, in C) and transforms it into a system of arithmetic constraints $\mathcal{C}$ in vectors of formal variables W, X, Y, such that $\exists w.y = \Psi(x, w) \iff \exists W.\mathcal{C}(W, X, Y)$ for $X = x$, $Y = y$. (Note that $\exists W.\mathcal{C}(W, X, Y)$ is an EQC; §II-A.) The primary figure of merit for a compiler is the size of $\mathcal{C}$: fewer constraints means less work for $\mathcal{P}$ to generate a proof, and in some proof systems it also means a shorter proof that is easier for $\mathcal{V}$ to verify.

a) The constraint formalism: The formalism used by most proof systems is called a rank-1 constraint system (R1CS). An R1CS instance $\mathcal{C}$ comprises a set of constraints over a finite field $\mathbb{F}$ (usually the integers modulo a prime p) of the form $\langle A_i, Z\rangle \cdot \langle B_i, Z\rangle = \langle C_i, Z\rangle$, where $\langle \cdot, \cdot \rangle$ is an inner product, Z is the concatenation $(W, X, Y, 1) \in \mathbb{F}^n$, and $A_i, B_i, C_i \in \mathbb{F}^n$ are constants. In other words, each constraint asserts that the product of two weighted sums of the wires in $\mathcal{C}$ equals a third weighted sum, which generalizes arithmetic circuits. $\mathcal{C}$ is *satisfied* when the values W, X, Y satisfy all constraints. To generate a proof, $\mathcal{P}$ first computes a satisfying assignment and then executes the cryptographic proving machinery.

Compiling programs from languages like C to R1CS instances is tricky. Domain differences are an immediate concern: while C has a non-trivial type system, for R1CS all computation must be encoded as arithmetic in $\mathbb{F}$, which can be awkward. For example, the assertion $x \neq 0$ has no direct encoding as a rank-1 constraint. When $\mathbb{F}$ is the integers mod p, by Fermat's little theorem this can be rewritten as $X^{p-1} = 1$, but this costs $\mathrm{O}(\log p)$ constraints; $p \approx 2^{256}$ is common for security of the proof system, so this is very costly. In this and similar cases, an important optimization is to introduce *advice* in the form of entries in the (existentially quantified) vector W. In our example, $x \neq 0$ becomes $\exists W.W \cdot X = 1$: since every nonzero element of $\mathbb{F}$ has a multiplicative inverse, this constraint is satisfiable iff $X \neq 0 \in \mathbb{F}$. For other examples see, e.g., [22], [5], [4], [31].

Beyond domain considerations lies a more insidious challenge: the fact that constraints cannot directly encode mutation, control flow, etc., so the compiler must transform input programs to eliminate these constructs. We defer discussion of this challenge to the next subsection.

C. SMT-based verification

In this section we discuss SMT and SMT-LIB, then explain how verifiers use these tools to prove properties of programs.

SMT solvers are tools that determine whether logical formulas are *unsatisfiable* (i.e., can never evaluate to true) or *satisfiable* (i.e., can evaluate to true); if satisfiable, the SMT solver provides a *satisfying assignment* to the variables in the formula. For example, given the formula $x \vee y$, an SMT solver may return a satisfying assignment of x to `true` and y to `false` (or any other valid assignment). Free variables in SMT formulas thus have existential semantics, which means that SMT formulas are EQCs (§II-A). In addition to booleans, SMT formulas can include terms from various *theories*, including bit-vectors, arrays, uninterpreted functions, real and integer arithmetic, etc; since theories are higher level than logical formulas, they make it easier for developers to use solvers.[2] The SMT-LIB [9] standard gives the semantics of each theory.

a) Compiling from high-level languages to SMT: SMT solvers are often applied in service of program correctness—everything from test case generation to bug finding to verification. Typically, a verifier translates a program and assertions about that program (e.g., `index` is within bounds of `array`) into SMT formulas (or similar). The verifier then asks the solver if the assertions make the program's formula satisfiable or not, either finding bugs or verifying their absence.

Compiling code into SMT formulas is challenging [15], [43], [14]. Since SMT (like R1CS) does not support mutable variables or branching, programs must be substantially transformed. As an example, consider this snippet of C code, and its mutation-free translation.

```
// input program (x, y, z previously defined)
if (x < 20) { x = 2; }
else        { y += z; }

// mutation-free program
x_1 = x_0 < 20    ? 2           : x_0;
y_1 = !(x_0 < 20) ? (y_0 + z_0) : y_0;
```

This snippet features variable mutations within an `if-else` statement. By introducing new versions of the variables, and guarding their new value by the condition of the statement, we can rewrite the snippet without mutations. While simple examples like this are easy enough to understand, eliminating mutation and branching quickly becomes complex. All branches must be recorded, all mutations guarded, all loops

[2]Theories have other benefits too: they allow solvers to use theory-specific algorithms for faster solving, and they allow users to specify formulas over infinite domains (e.g., the integers), which booleans cannot represent.

unrolled, and all program paths separately explored. Not only is this process complex—it also produces large formulas.

D. Integer linear programs

Mixed integer linear programs (ILPs) are a ubiquitous language for constrained optimization. An ILP comprises (1) set of real variables: $X = \{x_1, \ldots, x_n\}$, (2) a set of constraints, each having form $\sum_i c_i x_i \leq b$ (for real constants c_i, b), (3) an objective function $f(x) = \sum_i c_i x_i$ (for real constants c_i) and (4) a subset of integral variables $I \subset X$. Given an ILP, *an assignment* $v : X \to \mathbb{R}$ is *satisfying* if it respects all constraints and assigns integers to the variables in I. An ILP can be *infeasible* (there are no satisfying assignments), *unbounded* (the constraints do not entail an upper bound on f's value), or *solvable*. An ILP solver determines whether an ILP is infeasible, unbounded, or solvable; if the ILP is solvable, the solver searches for a satisfying variable assignment that achieves the maximum value of f.

E. Related work

CirC is related to and inspired by LLVM [32] and SUIF [44], but CirC targets EQCs instead of CPUs. MLIR [45] generalizes the LLVM methodology with a toolkit for constructing and manipulating interlocking IRs; in other words, MLIR is infrastructure for constructing compiler infrastructure. This work is orthogonal to CirC, which is compiler infrastructure for a specific family of non-deterministic computational models.

High-level synthesis (HLS) turns programs (say, in C) into digital circuits ([46] surveys). While digital circuits appear superficially similar to EQCs, there are two key differences: first, digital circuits do not allow existential quantification, which is very important for efficiently compiling to EQCs. Second, digital circuits are stateful; indeed, efficient use of stateful elements like flip-flops is a key focus of HLS.

Many compilers to specific EQCs exist; we discuss closely related work below. The main difference between CirC and these compilers is that CirC is infrastructure for compiling to EQCs generally, not just SMT, R1CS, etc. Generalizing existing compilers to support a wide range of EQCs would require essentially duplicating the work of building CirC.

The other difference between CirC and existing work is that most other work combines compilation with front-end–based optimization strategies. As examples: KLEE [25] combines a core constraint compilation engine with a path exploration front-end that forks at every branch; and Giraffe [38] uses program analysis to "slice" programs, then compiles a subset of these slices to proof system constraints. We don't currently support such strategies in CirC because they do not generalize to all back-ends. For example, forking compiles only a subset of program paths to constraints; this is acceptable for bug-finding, but is problematic for proof systems: some correct executions might not induce a valid proof. Supporting certain front-end–based optimization strategies is future work (§VIII).

a) Compilers for cryptography: A long line of prior work develops techniques for compiling to R1CS constraints. Ginger [22], Zaatar [47], Pantry [1], and Buffet [5] compile a subset of C and support proof-specific optimizations for rational numbers, memory, key-value stores, complex control flow, etc. Pinocchio [2] also compiles a subset of C with techniques similar to Ginger's. Geppetto [3] consumes LLVM IR and provides efficient cryptographic primitives. xJsnark [4] consumes a Java-like language and refines techniques from Buffet and Geppetto. Zinc [7] and ZoKrates [6] compile from eponymous DSLs to R1CS using existing techniques. Finally, Circom [8] is essentially a hardware description language that relies on the programmer to write constraints.[3]

Another line of work [29], [30] uses hand-crafted constraints that simulate a simple CPU, then modifies GCC to emit code for that CPU; while conceptually simple, it comes at enormous cost [5, §5]. Yet another approach, embodied by libsnark [48], ZEXE [49], and Bellman [50], uses a "macro assembler" to compose hand-crafted R1CS "gadgets."

Fairplay [21] is the earliest example of a compiler to circuits for two-party computation (2PC); FairplayMP [51] targets multi-party computation. Later works like Tasty [52] and HyCC [53] optimize by matching pieces of the source program to suitable cryptographic protocols.

CBMC-GC [54] adapts the CBMC [15] model checker to emit boolean circuits for 2PC, implicitly leveraging the similarities between compiling for model checking and for 2PC. Unlike CirC, however, CBMC-GC applies *only* to the case of boolean circuits optimized to 2PC, rather than to EQCs generally. Because of this, it cannot be used to compile to, say, arithmetic constraints for proof systems, nor does it enable the crossover applications that CirC does (§VII).

b) Compilers for SMT: Many projects compile high-level programs or parts of programs to SMT in order to prove program properties [55], [56], [57], [58], [59], [60], [61], [62], [63], [64], [65], [66], [15], [43] (e.g., using bounded model checking). For example, the CBMC verifier [56], [15] translates C programs and assertions about their correctness into SMT, unrolling loops up to a given bound. Then, it uses a solver to prove or refute verification assertions. Other program analyzers verify code written in verification-specific domain specific languages (DSLs) [67], [68], [69], [70]. For example, Alive [67] presents a DSL for peephole optimizations: programmers write optimizations in the DSL and Alive automatically verifies them. Some projects even allow users to manipulate constraints from within a higher-level language [71], [72], [73], [74].

All of the projects listed above handle their own compilation from a high-level language to SMT, but that is not always the case: there are many projects that present infrastructure for *building* verifiers [11], [14], [75], [10], [13], [76], [77]—often by handling the details of SMT compilation—and many verifiers that depend on them [78], [79], [80], [81]. Intermediate Verification Languages [76], [77], [82], [10] (IVLs), for example, decouple language details from verifier details; the Boogie IVL [10] targets SMT-like back-ends, and building a new verifier just requires translating a source language to the IVL. Taking a slightly different tack, Rosette [14] builds a virtual machine for symbolic execution, and Serval [78] uses it to lift interpreters into symbolic execution engines. CirC has a similar but broader goal: it aims to decouple source languages from *all* EQC targets, not just verifiers—and therefore must

[3]We discuss building Circom and ZoKrates compilers with CirC in Section III.

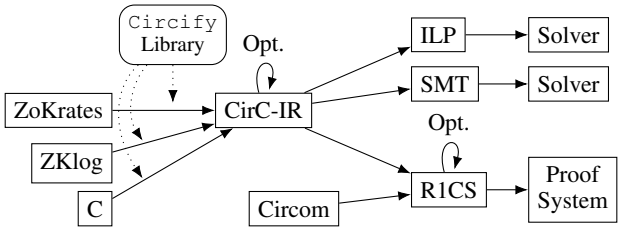

Fig. 2. CirC's architecture, with extensions (§III).

generalize lessons from these works to a more diverse range of constraint systems.

Symbolic execution (symex) tools [83], [84] combine SMT compilation of a single program path with, typically, either concrete execution or forking strategies [25], [85], [86], [87], [88], [89], [28], [90], [91], [92]; some systems use a hybrid of symex and model checking [14]. After SMT compilation, different tools proceed differently: many [25] fork execution at each conditional jump, some use a mixture of concrete and symbolic (concolic) execution [88], [28], some combine static analysis and symex [91], some combine symex and fuzzing [28], [90], and more. In each case, the symbolic tool relies on a core component that compiles programs to SMT.

c) Optimization Languages: A variety of tools translate from high-level modeling languages to ILPs, including AMPL [19] and Pyomo [20]. ILPs themselves are solved by a variety of tools including Gurobi [93] and CBC [94].

III. Design

In this section we present the high-level architecture of CirC and we describe `Circify`: a language-independent library for writing new CirC front-ends. Section IV discusses specific front-ends, back-ends, and optimizations.

A. Overview

CirC is designed to make new compilers easy to build: it should be easy for designers to add support for new input languages and new target EQCs. In CirC (as in LLVM), designers do so with new front-ends (e.g., for C) and new back-ends (e.g., for R1CS). Front-ends target CirC's IR and back-ends lower from IR to a target representation (Fig. 2).

B. CirC-IR

CirC's IR is a rich circuit language and is thus state-free and non-uniform. It's based on the SMT-LIB standard (§II-C), limited to booleans, floating-point numbers, bit-vectors, and arrays. These theories cover common primitive types in high-level languages, but one could extend the IR with other theories. In service of languages for proof systems, we extend the IR with a notion of finite fields, which underlie R1CS (§II-B).

C. `Circify`: managing state when compiling to CirC-IR

To support a new language, the first step is to build a front-end from that language to CirC-IR.

a) Building a front-end from scratch is hard: For example, consider the C program in Figure 5. This program includes variable mutations and conditional branches: two semantic phenomena that cannot be directly represented in CirC-IR because EQCs do not support mutation or branching. Other challenges include functions with early returns, loops with `break` or `continue` statements, and random-access memory, e.g., to access an array at a data-dependent index [5], [29], [56], [43]. While semantics of front-end languages differ in the details, these challenges are similar across many languages. Taking a broad view, we identify *state* as the key issue in these challenges—the state of variables, the state of memory, and even the state of the program counter. To help different language front-ends overcome challenges related to state, CirC provides `Circify`: a language-independent library for expressing state updates in CirC-IR.

b) `Circify` makes building front-ends easier: To construct a front-end, a developer essentially writes an interpreter for the source language using the `Circify` library.[4] In particular, `Circify` is responsible for managing the interpreter's execution environment: interactions with variables (e.g., declarations or mutations), functions (e.g., entry or returns), control flow (e.g., branches or breaks), lexical scopes, and even arrays (e.g., initializations, accesses), are delegated to `Circify` functions. `Circify` automatically handles the details of expressing stateful semantics in EQCs.

In the rest of this section, we describe `Circify`'s support for variables and conditional branching. This is only a small subset of `Circify`'s functionality, but it gives intuition for how `Circify` and a language-specific interpreter interact.

At a high level, `Circify` handles variable mutation by transforming the program into static single assignment form using a standard technique [95]: each time the source program assigns a new value to a variable, `Circify` creates a fresh version of that variable in CirC-IR, then constrains this version to be equal to the new value. For example, "`x = 1`" turns into "`x_`i `= 1`" for the next unused version number i.

This naive approach works for straight-line code, but handling control flow requires a technique called *guarding* [56], which works as follows: `Circify` records the conditions that must hold for any given program path to execute, and then guards all assignments on that path with those conditions. For example, consider the assignment on line 3 of Figure 5: this assignment only executes when `x = 0`. To achieve this, `Circify` uses an if-then-else (ITE) guard term, which evaluates to the new value (`y + z`) when the condition is true (i.e., when `x` is zero), and to the old value (`y`) otherwise.

Much of this guarded versioning machinery is language-independent. The only language-dependent pieces are: a definition of language values, a function that assigns values, and a function that constructs ITE terms over values. Together, these three pieces comprise `Circify`'s *language definition interface*, which `Circify`'s language-independent functionality uses to manipulate source language values. Figure 3 gives a slightly simplified definition of this interface, which comprises: (1) $\mathcal{V}$, the type of CirC-IR-embedded source language values; (2) assign, which assigns (the next version of) the variable named id to the value v and adds the corresponding

[4] This is vaguely reminiscent of Serval [78]).

- $\mathcal{V}$: a CirC-IR embedding of language values
- assign$(cs, id, v \in \mathcal{V})$
- ite$(c, t \in \mathcal{V}, f \in \mathcal{V}) \to \mathcal{V}$

Fig. 3. The `Circify` language definition interface. An interpreter must implement it to use `Circify`.

```
1 data CVal = Bool IR.Bool
2           | Uint IR.BitVec
3           | Sint IR.BitVec
4           | Struct String [(String, CVal)]
5           -- arrays, array pointers, ...
```

Fig. 4. C's $\mathcal{V}$-type, in Haskell. The `IR` module contains the CirC-IR definition.

assignment constraints to the CirC-IR constraint system cs; and (3) ite, which takes a CirC-IR boolean c and two language values t and f, and returns a language value that equals t when c is true and f otherwise.

c) An example: `Circify` for C: Figure 4 shows part our C front-end's $\mathcal{V}$-type, `CVal`, a recursive data type. Its base constructors wrap CirC-IR booleans and fixed-width integers (lines 1-3), and there is also a recursive constructor for named structures (line 4). The corresponding assign and ite definitions are straightforward when their value arguments are non-recursive: they simply emit an IR assignment or an IR ITE term for the CirC-IR value they wrap. For value arguments that contain recursive constructors, these functions recursively deconstruct the values, emitting IR assignments or IR ITE terms for each sub-value. For example, calling assign with a v that is structure results in recursive assign calls to each of the field in the structure.

After defining the `Circify` language definition interface for C, completing the front-end requires (1) writing a parser for the C source text, and (2) writing an interpreter that uses `Circify`'s functionality to translate stateful C code into CirC-IR. As an example, consider the `cond_add` C function in Figure 5. Figure 6 depicts the interpreter's calls to `Circify`, and `Circify`'s corresponding calls to the C language definition interface, as the interpreter steps through this function line-by-line. After argument declarations (not shown), the interpreter requires a few steps to interpret line 2. First, it uses the `Circify` `getVar` function to retrieve the IR representation of `x`'s current value; the result in this case is x's initial version, $\mathbf{x_0}$. Next, the interpreter builds the symbolic condition $\mathbf{x_0 = 0}$, and uses `Circify`'s `enterBranch` to indicate that subsequent calls are conditionally executed (only when $\mathbf{x_0 = 0}$) until a matching `exitBranch`.

For line 3, the interpreter first gets the values of `y` and `z`, then calls `Circify`'s `mutVar` to set `y` to $\mathbf{y_0 + z_0}$. Behind the scenes, this causes `Circify` to use the functions provided by the language definition interface. Specifically, `Circify` invokes the ite function to build a new term that is conditioned on the current path (where $\mathbf{x_0 = 0}$), then uses the assign function to bind the next version of `y` (i.e., $\mathbf{y_1}$) to that term.

Finally, the interpreter exits the branch and enters the alternative branch, i.e., the one corresponding to the $\mathbf{x \neq 0}$ condition. The rest of the interpretation process is similar: the interpreter walks the program, using `Circify` functions to handle variable interactions and branching.

```
1 uint8_t cond_add(uint8_t x, uint8_t y, uint8_t z) {
2   if (x == 0)
3     y = y + z;
4   else if (y == 1)
5     y = y + z + z;
6   return y;
7 }
```

Fig. 5. A small C program.

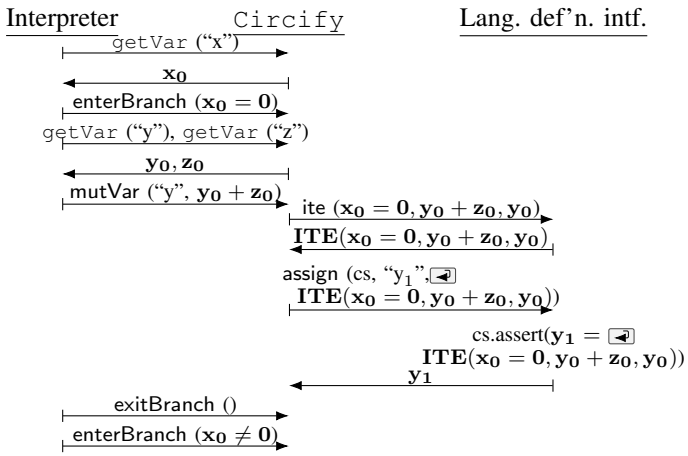

Fig. 6. Interactions between the ZoKrates front-end, `Circify`, and the front-end's language definition. Quotes denote strings, boldface denotes language values or CirC-IR terms.

d) Managing exceptional control flow: Most imperative languages include a number of constructs for *exceptional control flow*; e.g., early returns. CirC supports a single but powerful kind of exceptional control flow that we call the *breakable block*. A breakable block comprises a sequence of statements, and within the block, a `break` directs control flow to the end of the block. Early returns, loop breaks, and loop continues can all be expressed with breakable blocks. An early return sets the return value and jumps to the end of a breakable block containing the function body. A loop break jumps to the end of a breakable block containing the loop. A loop continue jumps to the end of a breakable block containing just the body of the loop. Breakable blocks can also express try-catch blocks, but they cannot simulate all control-flow (e.g., `gotos`).

We implement breakable blocks in CirC by extending the guarding system. Recall that CirC guards the actions of side-effect inducing statements on a condition c equal to the conjunction of enclosing branch conditions: $c = \bigwedge_i \mathsf{branch}_i$. To handle breakable blocks, for each breakable block i, `Circify` stores the condition under which the block has been broken out of: break_i. Then, `Circify` guards side effects within the block under $c = \bigwedge_i \mathsf{branch}_i \wedge \bigwedge_i \neg\mathsf{break}_i$, and when `Circify` encounters a break for block j, it sets break_j to $(\mathsf{break}_j \vee c)$.

e) Other functionality: While our discussion has focused on control flow and variable mutation, `Circify` solves other problems too. Specifically, `Circify` implements lexical scoping with global variables, local variables, and function scopes. It handles the allocation and access of stack arrays, and has has rudimentary support for a reference system, like in C++, or like some pointers in C (Appendix B-A). Together, `Circify`'s features provide a useful framework for

constructing front-ends for new languages.

Once a language designer has used `Circify` to attach a front-end to CirC, they can create their own optimizations over CirC-IR, or use any existing ones (§ IV-B).

D. Back-ends: from CirC-IR to circuits

To support a new EQC back-end, designers lower CirC-IR to their chosen representation. Generally speaking, writing a back-end for a new circuit representation is easier than writing a front-end, since CirC-IR is already a circuit IR. Once a developer has lowered CirC-IR to their target of choice, they can perform target-specific optimizations. This is relatively standard: many compilers support such optimizations [96], [67], and we discuss those that we implement in Section IV-C.

IV. Extensibility and implementation

As we discussed in the previous section, CirC's main goal is extensibility to new source languages and target EQCs. The compiler's design made it easy to implement five front-ends—for C, Circom, ZoKrates 0.6.1, a modified version of ZoKrates 0.7.6 that we call Z#, and a Datalog dialect—and three back-ends—to SMT, R1CS, and ILP. The rest of this section gives background on the languages and back-ends, and describes using CirC to support each of them.

A. Adding front-ends

`Circify` makes certain aspects of supporting a new language easy, because it manages the transformation of stateful programs with complex control flow into flat circuits. On the other hand, `Circify` does not assist with features that are likely to be language-specific, such as type checking and namespacing. In this section, we discuss CirC's front-ends and describe what was easy and what was hard for each.

a) C: CirC supports a subset of C that includes floats, doubles, booleans, integers, structures, stack arrays, and pointers to a statically known variable or array. Our C front-end also includes an opt-in taint-tracking system for definedness, which is useful, e.g., for detecting bugs due to undefined behavior (§VII-C). It does not support recursion or `goto`.[5] In spite of these limitations, CirC's C semantics are richer and more standards-compliant than prior work. As important examples: CirC wraps integer arithmetic modulo powers of two, as required by C11; Pequin [97] instead wraps modulo a large prime. CirC supports data-dependent array accesses and pointer offsets; PICCO [98] and CMBC-GC [56] only support constant offsets that are known at compile time.

b) ZoKrates: ZoKrates is a recent (2018) language for programming zero-knowledge proof systems. Developers write ZoKrates programs that check properties (e.g., "account balance is positive"), and the ZoKrates compiler converts those programs into equivalent R1CS. ZoKrates is a mature project: 42 contributors have authored over 3600 commits over the past several years, and there have been forty-two versions of the ZoKrates language to date. We now describe our experience implementing front-ends for two versions.

[5]Prior work [15], [5] shows how to support some of these constructs.

ZoKrates 0.6.1's types are fixed-width integers, finite field elements, booleans, field element–indexed arrays, and structures. The language supports mutable variables, conditional expressions, and statically bounded loops. Neither data-dependent control flow (e.g., conditional statements) nor loops with an input-dependent number of iterations are supported.

Our ZoKrates front-end builds on `Circify` (§III-C), which handles essentially all of the complexity (variable mutation, scoping, and function calls). The most complex remaining issue is ZoKrates's module and import system: import directives can rename imported identifiers, so name resolution depends on the current module (e.g., a structure might be defined as `S` but imported into another module as `S'`). To handle this, the front-end tracks the current module and uses this information for name resolution (≈50 LOC).

Building this compiler took one person less than one week, and the our ZoKrates front-end comprises less than 900 lines of (non-parser, non-AST) code. For comparison, the core of the ZoKrates 0.6.1 reference compiler (which also excludes parsers, ASTs, and tooling) comprises over 24000 lines and was written over the course of multiple years. Furthermore, our compiler supports a strict superset of the ZoKrates language, including (for example) if-statements. Finally, as §V will show, our compiler produces slightly better output.

ZoKrates 0.7.6 adds support for structures and functions that are generic over integer constants (used to write code that is generic over array lengths; recall that these must be known at compile time), with rudimentary inference of generic parameters. We have adopted these features in Z#, a language based on (but not fully compatible with) ZoKrates 0.7.6 that is being used in a commercial deployment of zero-knowledge proofs. Adding support for these features, a Z# AST analysis library (used for type checking and inference), and an AST-walking interpreter (used to support compile-time macros, testing, and debugging) took roughly 3 kLOC. Importantly, none of these features required changes to CirC-IR or `Circify`, giving us confidence that CirC's abstractions will easily support more advanced language features.

c) ZKlog: Datalog is a class of logic programming languages; Datalog dialects have been used to express database queries, program analyses, and more. We build a front-end for a Datalog dialect that we call ZKlog, to show that CirC can support a very different programming paradigm from that of C or ZoKrates. ZKlog programs are a collection *rules* over input variables; rules are defined in terms of *cases*, which can introduce (existentially quantified) variables and enumerate *conditions*. Figure 7 shows a ZKlog rule for the relation $\mathtt{Y} = \mathtt{X}^{\mathtt{E}}$. ZKlog supports booleans, fixed-width integers, finite field elements, and fixed-length array types; it (like other Datalog dialects) does not allow negated rule applications, so any ZKlog rule can be compiled to an EQC. Appendix C presents the abstract syntax of ZKlog.

The main challenge in supporting ZKlog is recursion: ZKlog, as a Datalog dialect, relies on recursion for iteration—there is no looping construct. Naively inlining recursive calls would cause the compiler to diverge, so we bound recursion in two ways. First, we use a programmer-specified, command-line *recursion limit* that bounds the number of times any rule can be recursively inlined. Using this construct, the programmer must correctly specify a recursion limit that's

```
1 pow(X: u16, E: u16, Y: u16) % Y = X ** E
2  if E = 0, Y = 1;
3  or E > 0, E & 1 = 0, pow(X * X, E / 2, Y);
4  or exists Z: u16,
5     E > 0, E & 1 = 1, pow(X * X, E / 2, Z), Z * X = Y.
6
```

Fig. 7. Recursive ZKlog rule for computing powers.

sufficiently large to compile a given rule into an EQC. If the recursion limit is too low, the generated EQC is *incomplete*—unsatisifiable for some inputs that it should be satisfiable for.

To alleviate the burden on the programmer, we support arbitrary recursion for *primitive recursive rules* applied to *compile-time constants*. From computability theory, a rule (more generally, a function) is primitive recursive in a formal argument x if all recursive calls are strictly decreasing in x. For example, the rule `pow` in Figure 7 is primitive recursive in `E`. However, without the condition `E > 0` on line 3, it would not be, since `E / 2` might not be less `E`.

To identify rules and arguments with primitive recursion, we allow programmers to annotate one rule argument as "`decreasing`". If the rule recurses and the annotated argument is a compile-time-constant that is strictly less than its previous value, the compiler ignores the recursion limit while compiling the rule to an EQC. We use CirC's constant-folding pass to check whether a value is a compile-time-constant that is less than in previous calls.

We can also use an SMT solver to determine whether a `decreasing` annotation is valid for *all* compile time constants. We describe this analysis in Section VII-B.

d) Circom: Circom is a hardware description language for arithmetic circuits. Our Circom compiler thus bypasses CirC-IR, and only takes advantage of the R1CS optimizations in CirC. It shows that the CirC compilation framework is modular: it's possible to attach front-ends to CirC-IR, but it's also possible to attach directly to back-ends in order to take advantage of target-specific optimizations.

B. Adding optimizations

Once a front-end has compiled its input to CirC-IR, the optimization phase begins. CirC includes a large suite of CirC-IR-based optimizations that language designers can selectively apply, depending on their input language and target circuit. Many optimizations are traditional: constant-folding, n-ary operator flattening, and inlining; these are almost always beneficial. Other optimizations are more complex and should be selectively applied. For example, CirC can replace operations over bit-vector arrays with operations over bit-vectors, via memory-checking techniques [1], [99], [29], [30], [5], [4]. This is *essential* for proof systems, which do not support arrays. When compiling to SMT—which natively supports arrays—this is unnecessary. For a more detailed example of how optimizations for one compilation pipeline may apply to others, see App. B-B.

C. Adding back-ends and target-specific optimizations

We've implemented three EQC back-ends in CirC: SMT, R1CS, and ILP.

a) SMT: Targeting SMT from CirC is trivial, since CirC-IR is based on SMT-LIB. Our SMT back-end (based on Z3's Haskell bindings [100]) supports all of CirC-IR, save finite field elements. Finite field arithmetic could be represented as modular arithmetic over bit-vectors of sufficient width, but existing SMT solvers are hopelessly in-efficient on this encoding, so we omit it. Efficiently supporting finite fields for SMT solvers is future work.

b) R1CS: Our R1CS back-end supports booleans, finite field elements, bit-vectors, and arrays checkable using memory checking techniques (§III-D).[6] Prior work [4], [1], [22], [47], [101], [102] embeds booleans and fixed-width integers in R1CS; we adapt these techniques to lower booleans and bit-vectors to R1CS. Our lowering pass also optimizes translation of certain CirC-IR terms. For example, the bit-vector term "`(c & t) | ((not c) & f)`" is better translated as a bitwise if-then-else than as two ANDs and an OR.

Following prior work on proof system compilers [4], [8], we implement one simple but powerful R1CS-specific optimization in CirC: linearity reduction. This optimization looks for linear constraints, e.g., $0 = c + \sum_i c_i x_i$. It solves for one of the variables in the constraint, then eliminates that variable from the constraint system using substitution.

c) ILP: Our ILP back-end supports booleans, bit-vectors, and arrays captured by memory checking techniques (§III-D). Given a CirC-IR circuit C that outputs an unsigned bit-vector, the back-end uses techniques from [103] and [104] to build an ILP. This ILP includes variables that encode C's inputs, and its objective is maximized when C's output is. Thus, by giving the ILP to a solver (we use CBC [94]), one can discover output-maximizing inputs for C.

D. Implementation

Our original CirC implementation comprises ≈15k lines of Haskell. The core (i.e, `Circify` and the IR definition and optimizations) is ≈5700 lines; tests and extensions are the rest. This is the implementation that we evaluate (§V). We have also re-implemented CirC in Rust (≈11k lines), which makes it easier to reason about and optimize compilation times and to bring new developers onboard. The ILP and ZKlog extensions are only in the Rust implementation.

The Rust version is the focus of future development, but we plan to release both implementations as open source.

V. Output performance and correctness

Language designers should be able to use CirC to create *correct*, *efficient* circuits. In this section, we evaluate both performance—does CirC produce circuits that perform well with respect to a given target—and correctness—can CirC accurately model input language semantics? We measure performance by comparing CirC's proof system pipelines to state-of-the-art, dedicated R1CS compilers. We evaluate correctness by running CirC's SMT pipeline on two standard verification benchmarks. Ultimately, we answer two questions. Does CirC:

- Emit R1CS outputs competitive in size with those emitted by state-of-the-art proof-system compilers? (§V-A)

[6] CirC does not currently lower floating-point arithmetic to R1CS. Although prior work supports a rudimentary floating-point representation [22], embedding IEEE 754–compatible floats in R1CS remains an open problem.

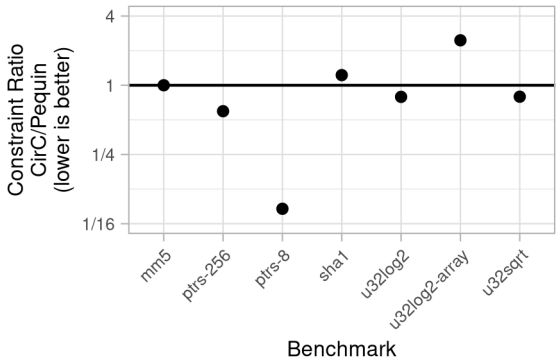

Fig. 8. Comparison between CirC and Pequin (§V-A). Relative performance depends on the benchmark, with neither compiler dominating the other.

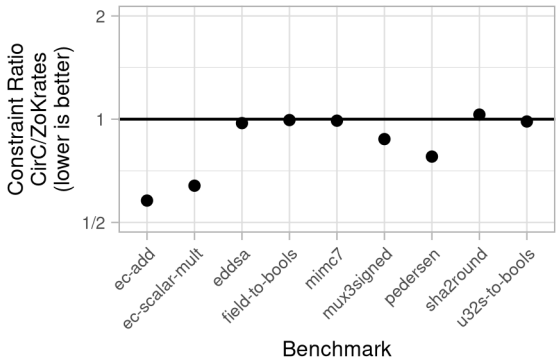

Fig. 9. Comparison between CirC and ZoKrates's reference compiler (§V-A). CirC generally produces better output.

- Emit SMT circuits that capture C program semantics with enough fidelity to find simple bugs? (§V-B)

We find that CirC's R1CS outputs perform exactly the same as the Circom compiler's, roughly the same as Pequin's, and slightly better than the ZoKrates compiler's. We also find that CirC's SMT formulas are correct on two benchmark suites from the sv-comp [105] verifier competition.

A. Performance

We consider three compilation pipelines when evaluating the performance of CirC's output: C-to-R1CS, ZoKrates-to-R1CS, and Circom-to-R1CS. We find that CirC is competitive with the state of the art in all cases, and slightly outperforms the ZoKrates compiler; our metric is the number of rank-1 constraints, which is standard (§II-B; [5], [4], [31]).

a) C-to-R1CS: Compiling C to R1CS stresses CirC's handling of boolean, bit-vector, and array (memory) constraints. On this task, we evaluate CirC against Pequin [97], a state-of-the-art compiler from C to R1CS that builds on a long line of work [22], [47], [1], [5]. We use 6 benchmarks from the Pequin software distribution covering a representative sample of control-flow patterns and primitive operations. Appendix A contains a further evaluation using the benchmarks of [5]. Pequin assumes that arithmetic never overflows; we use a modified version of CirC's R1CS machinery that matches Pequin's semantics. For each benchmark, we report the ratio between the number of constraints that CirC and Pequin produce, which is lower when CirC performs better.

Figure 8 shows the results: the compilers perform comparably. On simple arithmetic (`mm5`: matrix multiplication), they produce an identical number of constraints. On a binary-search implementation of integer square-root (`u32sqrt`), CirC has a slight edge, probably because of aggressive constant folding. On an addition- and bit manipulation–intensive hash (`sha1`), however, CirC performs slightly worse, likely due to missed inlining opportunities. CirC uses 11.9x fewer constraints for small arrays (`ptrs-8`) because CirC optimizes its memory representation for memory size and access pattern (as in xJsnark [4]), whereas Pequin uses a single memory representation that is asymptotically cheap yet concretely costly for small arrays. The benchmarks from [5] (App. A) also access small arrays, so CirC performs similarly well.

The exception is `u32log2-array`, which computes integer logarithms by decomposing the input into an (integer-typed) array of bits, then scanning that array. CirC does not yet have an optimization pass for integer-typed arrays containing only boolean values, so it treats the intermediate array as if it contains integers rather than bits, yielding much worse performance than Pequin. (This is a relatively simple optimization; adding it is future work.) When we instead evaluate a version of this function written in a more standard way (`u32log2`; Fig. 14), CirC outperforms Pequin slightly.

b) ZoKrates-to-R1CS: We evaluate CirC's ZoKrates-to-R1CS pipeline relative to the ZoKrates compiler, version 0.6.1. We evaluate against this version because CirC's ZoKrates front-end is fully compatible with it (§IV-A).

This tests how CirC performs when the source language includes R1CS-friendly features like field elements and control-flow limitations. Our benchmarks cover all major modules from the ZoKrates standard library. The modules (and benchmarks) are: utilities (`mux3`, `field-to-bools`, `u32s-to-bools`), hashes (`mimc7`, `pedersen`, `sha2round`), elliptic curve operations (`ec-scalar-mult`, `ec-add`), and signature verification (`eddsa`). In Appendix A, we evaluate on the entire standard library. As above, we report the ratio of constraint counts.

Figure 9 shows the results: CirC slightly outperforms the reference compiler. On straight-line computations with simple operations (`mimc7`, `fields-to-bools`, `u32s-to-bools`), the compilers perform similarly. When there are opportunities for common sub-expression elimination (`ec-scalar-mult`, `ec-add`), or when CirC can optimize conditional expressions (`pedersen`, `mux3signed`), CirC performs better. In one case, `sha2round`, CirC performs very slightly worse, likely due to missed inlining opportunities. Our evaluation on the whole ZoKrates standard library (App. A) corroborates these results.

c) Circom-To-R1CS: Circom [8] is effectively a hardware description language for R1CS. We support it in CirC by writing a front-end which targets R1CS directly. Thus, compiling Circom to R1CS is a test of CirC's R1CS-specific optimizations (§III). We evaluate CirC against the Circom compiler (v0.0.30) on the test suite for Circom's standard library. The compilers perform identically on all nearly all benchmarks. This is because Circom designs explicitly de-

scribe R1CS constraints, and the compilers apply the same R1CS optimizations. Appendix A contains the details of the evaluation.

B. Correctness

To evaluate the *correctness* of CirC's output, we run it on a subset of the tests from the Software Verification Competition (sv-comp). This annual competition includes many benchmarks that stress the speed and accuracy of software verifiers. By extending CirC's C front-end to support sv-comp conventions for existential inputs, assertions, and assumptions (≈40 LOC), we can run CirC on sv-comp benchmarks.

We run CirC on two benchmark categories: `signedint-egeroverflow-regression`, which tests the precision with which overflow is modeled, and `bitvector-loops`, which tests the precision with which branches, stack arrays, and basic pointers-to-stack-arrays are modeled. We choose these categories since they exercise most of CirC's support for C semantics (§III). CirC is correct in both categories.

We do *not* compare CirC's performance (e.g., the SMT solver's execution time) on sv-comp benchmarks relative to other systems. While CirC supports simple IR-level optimization passes, it does not currently include machinery for sophisticated static analysis (e.g., SMACK's static analysis for its memory representation [13]). Moreover, though CirC handles the "compilation to SMT" piece of a verifier, it is often not comparable to the *whole* verifier (e.g., CirC does not currently support a front-end forking phase like KLEE [25]). We discuss combining CirC with existing verifiers and high-performance verification strategies in Section VIII.

VI. Common performance characteristics

This section shows how SMT solvers and proof systems have similar performance characteristics, which means that optimizations for one pipeline (e.g., C-to-R1CS) can improve performance in another pipeline (e.g., C-to-SMT). This fact is not obvious at first glance. Proof system performance metrics (i.e., prover runtime) are almost entirely determined by the number of rank-1 constraints in the input circuit, while SMT solver performance metrics (i.e., solver runtime) are more difficult to understand—and are sometimes surprising [106], [107]. Nevertheless, CirC's optimization passes demonstrate performance similarities between both targets. In this section, we show that CirC's SMT-inspired constant-folding helps proof systems, too, (§VI-A), and that CirC's *oblivious array elimination* pass (§VI-B) and granular array modeling (§VI-C) help both solver and proof system back-ends.

A. Constant folding

SMT term rewriting—replacing one SMT term with an equivalent one to assist in SMT solving—is an old technique [108] used in all major solvers [109], [110], [111], [112], [113], [114]. *Constant folding* refers to a simple but important class of rewrites such as replacing $4 + 5$ with 9 or replacing the bit-vector term x<<1 with extractions and concatenations.

Constant folding helps the proof system back-end, too: the following experiment shows that it reduces the number of output constraints. We write a small (insecure!) hash function, H, which digests N 32-bit blocks into a 32-bit result. We wrap the function with an assertion that the output ends in a zero byte, encoding the predicate $\exists x.H(x)[0..8] = 0$. We use CirC to translate this predicate to R1CS and SMT, varying N and turning constant folding on and off. For R1CS, we measure the number of constraints; for SMT we measure the time that the SMT solver takes to find a satisfying x.

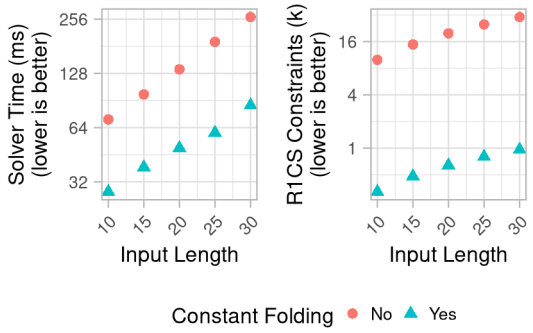

Fig. 10. Solver and proof system performance when operating on the predicate $\exists x.H(x)[0..8] = 0$ (§VI-A). Constant folding improves performance for both back-ends.

Figure 10 shows the results: constant folding reduces R1CS constraint count and SMT solver runtime. The benefit for R1CS is substantial: constant folding reduces constraint count by a factor of more than 16. The effect is smaller for SMT, likely because the SMT solver already does constant folding.[7]

B. Oblivious array elimination

In the spirit of *oblivious Turing machines*, whose head movements are input-independent, we use the term *oblivious array* to refer to an array that is accessed only at input-independent indices. CirC includes an optimization pass that identifies such arrays and replaces them with a sequence of distinct terms which are accessed independently. This optimization is essentially a strengthening of well-known scalar replacement optimizations: rather than just replacing a few array references with scalars, the entire array is eliminated.

We conduct an experiment showing that both targets respond similarly to oblivious array elimination. We write a C program that (1) declares an array of N `int`s (2) fills the array with non-deterministic inputs (3) sums all array elements and (4) asserts that sum to be non-zero. Since the array is only accessed at input-independent indices, the oblivious array elimination pass replaces it with distinct terms. We use CirC to find assertion-violating inputs using an SMT solver, and measure the solver's runtime. We also use CirC to compile the program (and assertion) to R1CS, and count the number of constraints. Figure 11 shows the results with and without the optimization, with varied N.[8] Both targets perform better when array elimination is enabled.

[7] Because the SMT solver performs a search, while the R1CS simply encodes the predicate, solver time scales exponentially with input length, while R1CS constraints scale linearly. This difference is orthogonal to the benefit of constant folding to both performance metrics.

[8] When array elimination is disabled, the R1CS target falls back to its standard array implementation, which was briefly discussed in Section V-A.

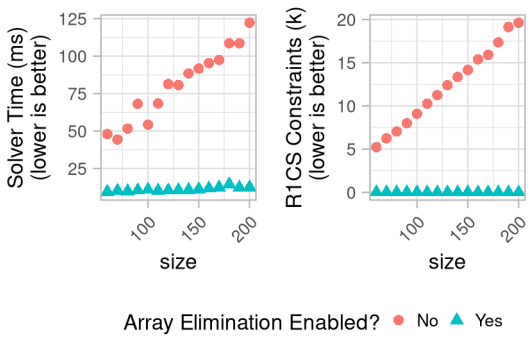

Fig. 11. Solver and proof system performance are both improved by oblivious array elimination (§VI-B).

C. Array granularity

Compiling programs that use random-access storage (henceforth, arrays) requires the compiler to model those arrays. CirC uses a *fine-grained* model: each source array is represented by its own SMT array. An alternative, *coarse-grained* model would use a single large "stack" array containing all source-level arrays. While the coarse-grained approach has some benefits [91], it is generally more expensive for SMT solvers to reason about [115], [116]. We now present an experiment demonstrating that the coarse-grained approach is also much more expensive for proof systems.

Our benchmark in this experiment is a function that takes as input an integer between 0 and $w-1$, and applies a sequence of n permutations of $\{0,1,\ldots,w-1\}$ to it. Each permutation is encoded as an array, so applying the permutation is just an array index operation (the program assumes that the input is in bounds). To vary array granularity, we apply a source transformation that fuses separate arrays into shared arrays, which simulates coarser- or finer-grained arrays (e.g., fusing all arrays simulates the "stack" approach discussed above).

Figure 12 shows our benchmark program for the case $n = 2, w = 4$; the program uses sv-comp style assumptions and assertions. The final line of the program asserts that the output i is not 0, which we use to measure performance as follows. For the SMT back-end, we ask the solver to find an assertion violation and measure how long it takes to produce a result (a violation is guaranteed to exist, because permutations are invertible). For the proof back-end, we measure the number of R1CS constraints to construct a proof that the input violates the assertion about the output.

Figure 13 shows the results for $n = 6$ and $w \in \{2,4,8\}$, varying the percentage of permutations fused into a single array. The results show that fusing permutations together—i.e., coarsening granularity—significantly reduces performance for both the SMT and proof back-ends.

VII. Crossover applications and techniques

In contrast to traditional compilers, CirC's targets serve substantially different purposes. This opens the door to applications that combine targets, and to techniques that use one target to help another. In this section, we discuss four such cross-overs: SMT-driven optimization of R1CS size, SMT-based analysis of ZKlog code, automatic detection and zero-knowledge proof of bugs, and automatic optimization and zero-knowledge proof of high value. With CirC, these cross-overs are easy to implement: each requires between 16 and ≈60 new lines of code.

```
void perm(int i) {
  __VERIFIER_assume(i >= 0 && i < 4);
  int perm0[4] = {2, 0, 1, 3}; i = perm0[i];
  int perm1[4] = {0, 1, 3, 2}; i = perm1[i];
  __VERIFIER_assert(i != 0);
}
```

Fig. 12. Array granularity benchmark: this C program applies two permutations, which are implemented as arrays, to its input (§VI-C).

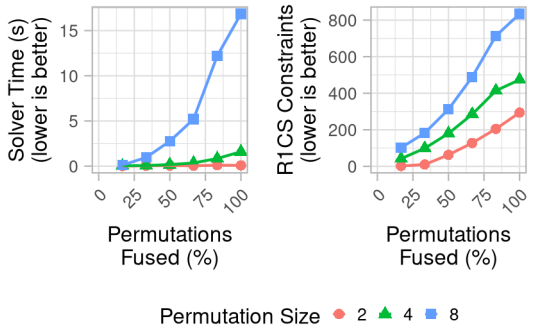

Fig. 13. Effect of array granularity on solver and proof system performance (§VI-C). Increasing x-axis corresponds to increasingly coarse-grained array representations, which increase costs for both the SMT and proof back-ends.

A. Optimizing R1CS using SMT

SMT-guided optimization is an old idea, and SMT solvers have optimized everything from code [117] to smart contracts [118] to TensorFlow graphs [119]. CirC makes it easy to apply SMT-guided optimizations to R1CS, too.

To illustrate this, we use one critical compilation task—loop unrolling—as a case study. To embed a loop like the one in Figure 14 in an EQC, the compiler must unroll it some number of times N, and in some cases emit an assertion that the bound is respected (§II). If N is too small, the resulting circuit won't handle some valid executions; if N is too large, the extra unrollings increase circuit size—and thus solving or proving time. Precisely determining N guarantees completeness while minimizing circuit size.

For this case study, we extend CirC (≈18 LOC) to use an SMT solver to determine the maximum number of iterations of a loop. Obviously this approach cannot work for all programs, but it is quite effective for some: for the `u32log2` function of Figure 14, CirC and the SMT solver determine that $N = 32$ in well under one second. Figure 15 shows the workflow for `u32log2`. Initially, the loop is unrolled once into CirC-IR which is is lowered to SMT that is satisfiable, so unrolling continues. After 33 unrollings, the corresponding SMT is now unsatisfiable, so the IR for 32 unrollings is used to finish compiling to CirC-IR. The final IR is lowered to R1CS, for use in a proof system: the intended target.

```c
#include "stdint.h"
uint32_t u32log2(uint32_t x) {
    uint32_t n_bits = 0;
    while (x != 0) {
        n_bits++;
        x >>= 1u;
    }
    return n_bits - 1;
}
```

Fig. 14. This function computes $\lfloor \log_2 \mathtt{x} \rfloor$. The SMT solver determines how many iterations to unroll (§VII-A).

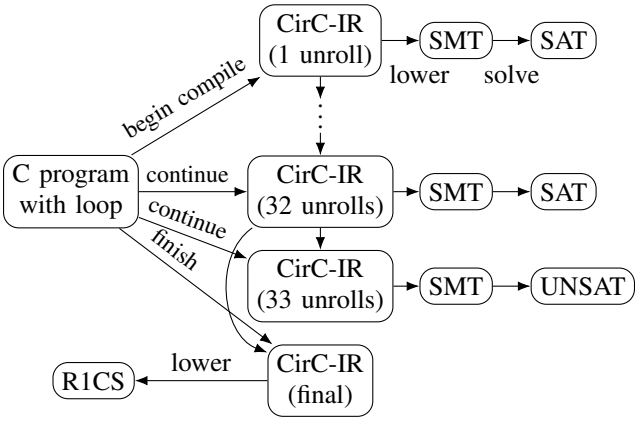

Fig. 15. Workflow for determining loop bounds with SMT while compiling to R1CS. Incrementally compiled circuit fragments are lowered to SMT for analysis. An UNSAT result causes unrolling to end.

In future work we hope to improve this technique, e.g., by using the SMT solver's incremental mode, and to use the SMT solver for more complex R1CS optimizations.

B. Analyzing ZKlog with SMT

Our ZKlog compiler does not limit recursion when a rule that is annotated as primitive recursive is applied to decreasing compile-time-constants (§IV)—but it can be difficult to tell if a rule is primitive recursive at all.[9] Consider a rule r with arguments $x_1, \ldots, x_m$ that (without loss of generality) purports to be primitive recursive in x_1. Now, consider a recursive case c for rule r that binds existential variables $x_{m+1}, \ldots, x_n$ with conditions $t_1 \wedge \cdots \wedge t_n$ such that (without loss of generality) condition t_1 is the recursive application of rule r where the first argument depends only on x_1. Thus, the case c encodes the implication:

$$[\exists x_{m+1}, \ldots, x_n.\ r(f(x_1), \ldots) \wedge t_2 \wedge \cdots \wedge t_n] \rightarrow r(x_1, \ldots, x_m)$$

For this to be a primitive recursion, it suffices to show that the other conditions in c imply $f(x_1) < x_1$. That is, that the following is unsatisfiable:

$$(f(x_1) \geq x_1) \wedge t_2 \wedge \cdots \wedge t_n \tag{1}$$

Our analysis pass checks this property by lowering it to SMT. The programmer can run the analysis on any rule that they've annotated as primitive recursive. If (1) is UNSAT for each recursive case, then the rule may be applied to any compile-time constant without introducing incompleteness.

[9]Other languages that consider primitive recursion (e.g., Gallina, the language of Coq [120]) occasionally require the programmer to prove to the compiler that a recursion is primitive.

C. Automatically finding and proving bugs

Bug bounties are a popular way for software companies to incentivize bug reporting, by offering cash rewards to reporters. But the incentives in these programs are imperfect. As examples, a company might accept a bug report but refuse to pay a bounty; or it might refuse to acknowledge a severe bug, leaving the reporter with an uncomfortable choice between remaining silent and publicly revealing the bug's details, both of which could harm innocent users. One way to address problems like this may be to prove the existence of bugs *in zero knowledge*; in fact, this application is a key element of an ongoing DARPA program [121], [122].

Prior work [123], [124], [125] constructs manual proof-of-bug pipelines, but none automatically detects bugs and then automatically proves their existence in zero knowledge: existing compilers to R1CS have no way of automatically detecting bugs, and existing SMT-based verifiers have no way of generating zero-knowledge proofs. In fact, just proving the presence of many types of bugs is beyond the reach of existing proof compilers that, like Pequin [97] (§V-A), model language semantics too imprecisely (§IV-A). Since CirC models language semantics precisely (§V-B) and can embed those semantics into both SMT and R1CS, CirC seems ideally suited to building a semi-automated proof-of-bug pipeline.

But zero-knowledge proofs-of-bug generally, and automated pipelines in particular, are still unrealistic. The main issue is that proof systems and SMT solvers both fall hopelessly short of practical applicability to production-sized codebases. Naively, this means that proving the existence of a bug in a large system might require first isolating the buggy code and then generating a proof—in the process revealing the bug's location and potentially allowing anyone to rediscover it!

Nevertheless, we see two strong reasons to hope for a path forward, with CirC as a key component. First, recent theoretical advances in efficient proofs of disjunctions [126], [127], [128], [129] promise proof systems well suited to proving statements like "there is a bug in one of the thousands of functions in this codebase"—and CirC makes it easy to marry such proof systems to SMT-based bug-finding techniques. Second, prior work shows that tools capable of detecting subtle bugs or verifying complex properties often rely on analysts' expert knowledge [78], [91]—of a class of bug, of a codebase, or even of the SMT solver itself [130]. In the case of solver-based bug finding, furnishing expert hints to the solver can dramatically increase the reach of bug-finding tools. Crucially, a zero-knowledge proof of a bug's existence does not reveal these expert hints; as a result, knowing the approximate location of a bug may not help non-experts rediscover it.

As a proof of concept, we augment CirC's C front-end with support for *assertions* and *assumptions* (≈60 LOC). Codebase owners use *assertions* to specify intended behavior. Consider a small example: the macro `mul_add_c2` (Fig. 16). It is intended to compute `c = c + 2*a*b`, where `c` is a multi-precision integer comprising three words, `c0`, `c1`, and `c2`. Figure 17 shows how a codebase owner would add an assertion of this behavior. Unfortunately, the macro mishandles integer overflow (see App. D for details). Indeed, an analyst who compiles the macro and assertion to SMT can find a violation

```
#define mul_add_c2(a,b,c0,c1,c2)        {  \
        BN_ULONG ta=(a),tb=(b),t0;          \
        BN_UMULT_LOHI(t0,t1,ta,tb);         \
        t2 = t1+t1; c2 += (t2<t1)?1:0;      \
        t1 = t0+t0; t2 += (t1<t0)?1:0;      \
        c0 += t1; t2 += (c0<t1)?1:0;        \
        c1 += t2; c2 += (c1<t2)?1:0;     }
```

Fig. 16. Incorrect carry handling in OpenSSL, responsible for CVE-2014-3570 (§VII-C). Appendix D explains the bug in detail.

```
#include "stdint.h"
int wrapper(uint64_t a, uint64_t b, uint64_t c2,
    uint64_t c1, uint64_t c0) {
  uint64_t cc2 = c2, cc1 = c1, cc0 = c0;
  mul_add_c2(a, b, c2, c1, c0);
  "SMT_assert: (= (concat c2 c1 c0) (+ (concat cc2
↪  cc1 cc0) (* [192]2 (uext 128 a) (uext 128 b))";
  return 0;
}
```

Fig. 17. Function wrapper for `mul_add_c2` with assertion of correct behavior (§VII-C). CirC's assertion language is more verbose; we simplify for brevity. `BN_ULONG`'s size depends on the architecture; we use 64 bits.

using the Z3 SMT solver in ≈700 seconds. The analyst can then produce a zero-knowledge proof that the codebase owner can verify in milliseconds (Figure 18). The above pipeline is impractical for larger programs and more complex properties, since the solving step quickly becomes intractable.

Assumptions allow the security analyst to use their expertise to make bug-finding tractable: the analyst encodes facts about the program or its potential bugs that reduce the SMT solver's search burden. As an example, consider again `mul_add_c2` (Fig. 16). Since overflow is a traditional source of bugs, and `mul_add_c2` does not explicitly handle overflow caused by its increment steps, the analyst may suspect that overflow as a possible bug source. With an (overflow) assumption that `t2 == 0 && c0 < t1` after line 6, the SMT solver can find a bug after only 2.5s. This improves on the unassisted search time by more than two orders of magnitude, but expert assumptions can also allow the solver to finish when it was unable to previously (e.g., in [131], Section 4.2).

D. Automatically finding and proving high-value inputs

CirC also supports *proofs of high-value* (which is very similar to proof-of-bug). Let $f(x) \to y$ be a function, and y_0 be a threshold output. A proof of high-value shows that one knows an x such that $\phi(x, y_0) := f(x) \geq y_0$, without revealing x. Proofs of high-value might be applied to *optimization competitions*. In an optimization competition (e.g., a Kaggle competition [132] or the ARPA-E Grid Optimization Competition [133]), competitors submit parameters which maximize an objective function, and high-performing competitors receive a reward. This setup has a similar drawback to bug bounty programs (§VIII): if competitors submit their parameters in the clear, then the competition operator could choose not to compensate them—but zero-knowledge proofs of high-value avoid the counterparty risk.

With CirC it's easy to compile an objective function (and any constraints) to an optimization format (e.g., an ILP) for finding high-value solutions. Then, it's easy to compile the same objective function to a proof-system constraint format (like R1CS) for writing zero-knowledge proofs. (Just as in the previous section, making this pipeline realistic requires using human insight to assist the ILP solver.) As a proof-of-concept, we build an automated pipeline for proofs of high-value (similar to Figure 18, but with an ILP solver in place of an SMT solver). The pipeline (≈30 LOC) takes a function f and (a) compiles the predicate $\phi(x, y) := f(x) \geq y$ to R1CS, (b) compiles f to an ILP, (c) solves the ILP to find an x^* such that $f(x^*) = y^*$, and finally, (d) writes a zero-knowledge proof of knowledge of an x^* that satisfies $\phi(x^*, y^*)$.

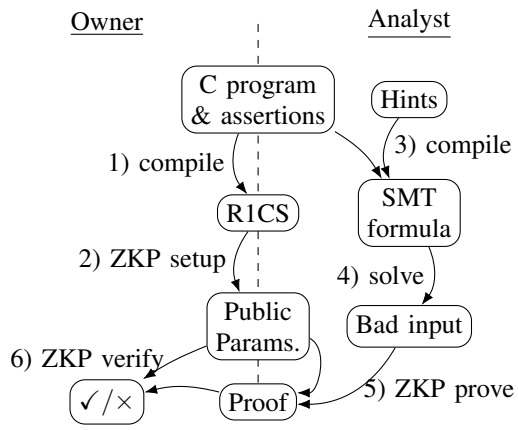

Fig. 18. Workflow for finding and proving bugs (§VII-C). Left steps are performed by the codebase owner; right steps, by the analyst. Step 2 samples public parameters needed for the zero-knowledge proof, for cryptographic reasons.

VIII. Discussion, Future Work, and Conclusion

a) Targeting other applications: CirC has applications beyond SMT, ILP, and proof systems. As one example, CirC could support *multi-party computation* (MPC), which enables mutually distrusting parties to collaboratively evaluate a function while revealing only the result [134]. MPC frameworks require the function to be expressed as a boolean [135], or arithmetic circuit [136], [137], [138], so extending CirC for MPC applications would require adding support for these. One potential issue is that MPC protocols do not support circuits with existentially quantified wires. This is likely not a problem, however, because in most (and maybe all) cases, such variables can be transformed either into private inputs supplied by one party, or into values computed from the private inputs of multiple parties. This view is implicit, for example, in the seminal work of Ishai et al. on constructing zero-knowledge proofs via MPC protocols [139].

b) Program analysis infrastructure: CirC supports IR-level optimizations, but sophisticated static analysis infrastructure—at both the language and IR level—would improve most compilation pipelines. For example, CirC could use a range analysis to shrink IR-level bit-vectors, which would make their R1CS embedding more efficient. As another example, designers could build analyses into their language front-end, e.g., to select the cryptographic protocol that gives the best efficiency on a particular program [52], [53], [38], [140]. Designing new analyses of this kind is also future work.

Beyond static analyses, there is potential to leverage more powerful SMT-based analyses in CirC. Section VII shows that CirC's SMT target can be used to search for loop bounds.

One could also imagine searching for loop invariants, tighter range bounds, aliasing relationships, and more. The program verification literature is full of SMT-based analyses which may be useful for optimizing the size of emitted circuits.

c) Combining CirC with existing verifiers: It might also be interesting to combine CirC with modern verification machinery. For example, CirC could benefit from SMACK's [13] front-end–based optimizations, while Boogie front-ends [10] could benefit from targeting cryptographic applications.

d) Conclusion: In this work, we show how CirC makes it easy to compile new source languages, support new EQC targets, and write optimizations that apply to multiple pipelines: all of these can be done with very little code, and all yield high-quality compiler output. Moreover, with CirC it's easy to combine different EQC compilation pipelines to support novel applications., e.g., automatically finding bugs and proving their existence. In short: shared infrastructure for constraint compilers *is* both possible and useful.

Acknowledgments

The authors thank Sebastian Angel, Clark Barrett, Dan Boneh, Patrick Cousot, Dawson Engler, Ranjit Jhala, Soren Lerner, Andres Nötzli, Deian Stefan, Michael Walfish, and Thomas Wies for helpful conversations. This work was supported in part by the NSF, the ONR, the Stanford Center for Blockchain Research, and the Simons Foundation. It was also supported by DARPA under Agreement HR00112020022. The views in this paper are the authors' and do not necessarily represent the views of the United States Government or DARPA.

References

[1] B. Braun, A. J. Feldman, Z. Ren, S. Setty, A. J. Blumberg, and M. Walfish, "Verifying computations with state," in *SOSP*, Nov. 2013. Extended version: http://eprint.iacr.org/2013/356.

[2] B. Parno, J. Howell, C. Gentry, and M. Raykova, "Pinocchio: Nearly practical verifiable computation," in *IEEE S&P*, May 2013.

[3] C. Costello, C. Fournet, J. Howell, M. Kohlweiss, B. Kreuter, M. Naehrig, B. Parno, and S. Zahur, "Geppetto: Versatile verifiable computation," in *IEEE S&P*, May 2015.

[4] A. E. Kosba, C. Papamanthou, and E. Shi, "xJsnark: A framework for efficient verifiable computation," in *IEEE S&P*, May 2018.

[5] R. S. Wahby, S. Setty, Z. Ren, A. J. Blumberg, and M. Walfish, "Efficient RAM and control flow in verifiable outsourced computation," in *NDSS*, Feb. 2015.

[6] "Zokrates." https://zokrates.github.io/.

[7] "Zinc." https://zinc.matterlabs.dev/.

[8] J. Baylina, "Circom." https://github.com/iden3/circom.

[9] C. Barrett, A. Stump, C. Tinelli, *et al.*, "The SMT-LIB standard: Version 2.0," in *SMT*, 2010.

[10] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino, "Boogie: A modular reusable verifier for object-oriented programs," in *FMCO*, 2005.

[11] K. R. M. Leino, "Dafny: An automatic program verifier for functional correctness," in *LPAR*, 2010.

[12] N. Bjørner and L. de Moura, "Applications of smt solvers to program verification," in *Notes for the Summer School on Formal Techniques*, 2014.

[13] Z. Rakamarić and M. Emmi, "Smack: Decoupling source language details from verifier implementations," in *CAV*, 2014.

[14] E. Torlak and R. Bodik, "A lightweight symbolic virtual machine for solver-aided host languages," 2014.

[15] E. Clarke, D. Kroening, and F. Lerda, "A tool for checking ANSI-C programs," in *TACAS*, 2004.

[16] A. Solar-Lezama, L. Tancau, R. Bodik, S. Seshia, and V. Saraswat, "Combinatorial sketching for finite programs," in *ASPLOS*, 2006.

[17] E. Kneuss, I. Kuraj, V. Kuncak, and P. Suter, "Synthesis modulo recursive functions," in *OOPSLA*, 2013.

[18] S. Srivastava, S. Gulwani, and J. S. Foster, "From program verification to program synthesis," in *POPL*, 2010.

[19] R. Fourer, D. M. Gay, and B. W. Kernighan, *AMPL: A Modeling Language for Mathematical Programming*. Boston, MA, USA: Cengage Learning, 2nd ed., 2002.

[20] W. E. Hart, C. D. Laird, J.-P. Watson, D. L. Woodruff, G. A. Hackebeil, B. L. Nicholson, J. D. Siirola, *et al.*, *Pyomo-optimization modeling in python*, vol. 67. Springer, 2017.

[21] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella, "Fairplay - secure two-party computation system," in *USENIX Security*, Aug. 2004.

[22] S. T. V. Setty, V. Vu, N. Panpalia, B. Braun, A. J. Blumberg, and M. Walfish, "Taking proof-based verified computation a few steps closer to practicality," in *USENIX Security*, Aug. 2012. Extended version: https://ia.cr/2012/598.

[23] A. Gurfinkel and J. A. Navas, "A context-sensitive memory model for verification of c/c++ programs," in *SAS*, 2017.

[24] C. Sinz, S. Falke, and F. Merz, "A precise memory model for low-level bounded model checking," in *SSV*, 2010.

[25] C. Cadar, D. Dunbar, D. R. Engler, *et al.*, "KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs," in *OSDI*, 2008.

[26] K. R. M. Leino, "This is boogie 2." https://www.microsoft.com/en-us/research/wp-content/uploads/2016/12/krml178.pdf, 2008.

[27] E. Cohen, M. Moskal, S. Tobies, and W. Schulte, "A precise yet efficient memory model for c," in *SSV*, Oct. 2009.

[28] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, "QSYM: A practical concolic execution engine tailored for hybrid fuzzing," in *USENIX Security*, 2018.

[29] E. Ben-Sasson, A. Chiesa, D. Genkin, E. Tromer, and M. Virza, "SNARKs for C: Verifying program executions succinctly and in zero knowledge," in *CRYPTO*, Aug. 2013.

[30] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza, "Succinct non-interactive zero knowledge for a von Neumann architecture," in *USENIX Security*, Aug. 2014.

[31] A. Ozdemir, R. S. Wahby, B. Whitehat, and D. Boneh, "Scaling verifiable computation using efficient set accumulators," in *USENIX Security*, Aug. 2020.

[32] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis and transformation," in *CGO*, 2004.

[33] S. Arora and B. Barak, *Computational Complexity: A Modern Approach*. Cambridge, UK: Cambridge University Press, 2009.

[34] Y. Zhang, D. Genkin, J. Katz, D. Papadopoulos, and C. Papamanthou, "vSQL: Verifying arbitrary SQL queries over dynamic outsourced databases," in *IEEE S&P*, 2017.

[35] E. Ben-Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza, "Zerocash: Decentralized anonymous payments from Bitcoin," in *IEEE S&P*, May 2014.

[36] G. Danezis, C. Fournet, M. Kohlweiss, and B. Parno, "Pinocchio coin: Building zerocoin from a succinct pairing-based proof system," in *PetSHOP*, 2013.

[37] R. S. Wahby, M. Howald, S. Garg, abhi shelat, and M. Walfish, "Verifiable ASICs," in *IEEE S&P*, 2016.

[38] R. S. Wahby, Y. Ji, A. J. Blumberg, abhi shelat, J. Thaler, M. Walfish, and T. Wies, "Full accounting for verifiable outsourcing," in *ACM CCS*, 2017.

[39] M. Walfish and A. J. Blumberg, "Verifying computations without reexecuting them: from theoretical possibility to near practicality," *Commun. ACM*, vol. 58, pp. 74–84, Feb. 2015.

[40] J. Thaler, "Proofs, arguments, and zero-knowledge." http://people.cs.georgetown.edu/jthaler/ProofsArgsAndZK.pdf.

[41] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza, "Scalable zero knowledge via cycles of elliptic curves," in *CRYPTO*, Aug. 2014.

[42] M. Fredrikson and B. Livshits, "Zø: An optimizing distributing zero-knowledge compiler," in *USENIX Security*, Aug. 2014.

[43] Y. Xie and A. Aiken, "Saturn: A scalable framework for error detection using boolean satisfiability," 2007.

[44] M. W. Hall, J. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, E. Bugnion, and M. S. Lam, "Maximizing multiprocessor performance with the SUIF compiler," *IEEE Computer*, vol. 29, no. 12, pp. 84–89, 1996.

[45] C. Lattner, J. A. Pienaar, M. Amini, U. Bondhugula, R. Riddle, A. Cohen, T. Shpeisman, A. Davis, N. Vasilache, and O. Zinenko, "MLIR: A compiler infrastructure for the end of moore's law." https://arxiv.org/abs/2002.11054.

[46] L. Daoud, D. Zydek, and H. Selvaraj, "A survey of high level synthesis languages, tools, and compilers for reconfigurable high performance computing," in *ICSS*, Apr. 2013.

[47] 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 *EuroSys*, Apr. 2013.

[48] "libsnark." https://github.com/scipr-lab/libsnark.

[49] S. Bowe, A. Chiesa, M. Green, I. Miers, P. Mishra, and H. Wu, "ZEXE: Enabling decentralized private computation," in *IEEE S&P*, May 2020.

[50] "Bellman circuit library, community edition." https://github.com/matter-labs/bellman.

[51] A. Ben-David, N. Nisan, and B. Pinkas, "FairplayMP: a system for secure multi-party computation," in *ACM CCS*, Oct. 2008.

[52] W. Henecka, S. Kögl, A.-R. Sadeghi, T. Schneider, and I. Wehrenberg, "TASTY: tool for automating secure two-party computations," in *ACM CCS*, Oct. 2010.

[53] N. Büscher, D. Demmler, S. Katzenbeisser, D. Kretzmer, and T. Schneider, "HyCC: Compilation of hybrid protocols for practical secure computation," in *ACM CCS*, Oct. 2018.

[54] A. Holzer, M. Franz, S. Katzenbeisser, and H. Veith, "Secure two-party computations in ANSI C," in *ACM CCS*, Oct. 2012.

[55] L. Cordeiro, B. Fischer, and J. Marques-Silva, "SMT-based bounded model checking for embedded ANSI-C software," *IEEE Trans. Software Engineering*, vol. 38, pp. 957–974, July 2011.

[56] D. Kroening and M. Tautschnig, "Cbmc–c bounded model checker," in *TACAS*, 2014.

[57] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski, "Frama-c," in *SEFM*, 2012.

[58] F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski, "Frama-c: A software analysis perspective," *Formal Aspects of Computing*, vol. 27, pp. 573–609, Jan. 2015.

[59] T. Ball, V. Levin, and S. K. Rajamani, "A decade of software model checking with slam," *Communications of the ACM*, vol. 54, pp. 68–76, July 2011.

[60] T. Ball, E. Bounimova, V. Levin, R. Kumar, and J. Lichtenberg, "The static driver verifier research platform," in *CAV*, 2010.

[61] K. R. M. Leino, P. Müller, and J. Smans, "Verification of concurrent programs with chalice," in *FOSAD*, 2009.

[62] B. Jacobs and F. Piessens, "The verifast program verifier," Tech. Rep. CW-520, Dept. of Computer Science, KU Leuven, Aug. 2008.

[63] N. Vazou, *Liquid Haskell: Haskell as a theorem prover*. PhD thesis, UC San Diego, 2016.

[64] N. Vazou, E. L. Seidel, R. Jhala, D. Vytiniotis, and S. Peyton-Jones, "Refinement types for haskell," in *ICFP*, 2014.

[65] N. Swamy, C. Hriţcu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P.-Y. Strub, M. Kohlweiss, *et al.*, "Dependent types and multi-monadic effects in f*," in *POPL*, 2016.

[66] L. Cordeiro, P. Kesseli, D. Kroening, P. Schrammel, and M. Trtik, "Jbmc: A bounded model checking tool for verifying java bytecode," in *CAV*, 2018.

[67] N. P. Lopes, D. Menendez, S. Nagarakatte, and J. Regehr, "Provably correct peephole optimizations with Alive," in *PLDI*, 2015.

[68] D. Menendez, S. Nagarakatte, and A. Gupta, "Alive-fp: Automated verification of floating point based peephole optimizations in llvm," in *SAS*, 2016.

[69] A. Nötzli and F. Brown, "Lifejacket: verifying precise floating-point optimizations in llvm," in *SOAP*, 2016.

[70] K. v. Gleissenthall, R. G. Kıcı, A. Bakst, D. Stefan, and R. Jhala, "Pretend synchrony: synchronous verification of asynchronous distributed programs," in *POPL*, 2019.

[71] R. Uhler and N. Dave, "Smten with satisfiability-based search," in *OOPSLA*, 2014.

[72] A. Köksal, V. Kuncak, and P. Suter, "Constraints as control," in *POPL*, 2012.

[73] L. Moura, "Z3py guide: Z3 api in python." https://ericpony.github.io/z3py-tutorial/guide-examples.htm.

[74] L. Erkök, "Sbv: Smt based verification in haskell." https://hackage.haskell.org/package/sbv.

[75] E. Torlak and R. Bodik, "Growing solver-aided languages with rosette," in *Onward!*, 2013.

[76] J.-C. Filliâtre and A. Paskevich, "Why3—where programs meet provers," in *ESOP*, 2013.

[77] P. Müller, M. Schwerhoff, and A. J. Summers, "Viper: A verification infrastructure for permission-based reasoning," in *International Conference on Verification, Model Checking, and Abstract Interpretation*, 2016.

[78] L. Nelson, J. Bornholt, R. Gu, A. Baumann, E. Torlak, and X. Wang, "Scaling symbolic evaluation for automated verification of systems code with serval," in *SOSP*, 2019.

[79] Z. Nehai and F. Bobot, "Deductive proof of ethereum smart contracts using why3." arXiv:1904.11281, 2019.

[80] M. Baranowski, S. He, and Z. Rakamarić, "Verifying rust programs with SMACK," in *ATVA*, 2018.

[81] Á. Hajdu and D. Jovanović, "solc-verify: A modular verifier for solidity smart contracts," in *VSTTE*, 2019.

[82] N. Swamy, J. Weinberger, C. Schlesinger, J. Chen, and B. Livshits, "Verifying higher-order programs with the dijkstra monad," in *PLDI*, 2013.

[83] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Computing Surveys*, vol. 51, no. 3, pp. 1–39, 2018.

[84] C. Cadar and K. Sen, "Symbolic execution for software testing: Three decades later," *Commun. ACM*, vol. 56, no. 2, pp. 82–90, 2013.

[85] D. A. Ramos and D. Engler, "Under-constrained symbolic execution: Correctness checking for real code," in *USENIX Security*, 2015.

[86] P. Godefroid, M. Y. Levin, and D. Molnar, "Sage: Whitebox fuzzing for security testing," *Queue*, vol. 10, no. 1, pp. 20–27, 2012.

[87] P. Godefroid, N. Klarlund, and K. Sen, "Dart: Directed automated random testing," in *PLDI*, 2005.

[88] K. Sen, D. Marinov, and G. Agha, "Cute: a concolic unit testing engine for c," 2005.

[89] V. Chipounov, V. Kuznetsov, and G. Candea, "S2E: A platform for in-vivo multi-path analysis of software systems," in *ASPLOS*, 2011.

[90] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution," in *NDSS*, 2016.

[91] F. Brown, D. Stefan, and D. Engler, "Sys: A static/symbolic tool for finding good bugs in good (browser) code," in *USENIX Security*, 2020.

[92] S. Poeplau and A. Francillon, "Symbolic execution with symcc: Don't interpret, compile!," in *USENIX Security*, 2020.

[93] "Gurobi." https://www.gurobi.com/.

[94] J. Forrest and R. Lougee-Heimer, "Cbc user guide," in *Emerging theory, methods, and applications*, pp. 257–277, INFORMS, 2005.

[95] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers, principles, techniques*. Addison Wesley, 1986.

[96] E. Mullen, D. Zuniga, Z. Tatlock, and D. Grossman, "Verified peephole optimizations for compcert," in *PLDI*, 2016.

[97] "Pequin: A system for verifying outsourced computations and applying SNARKs." https://github.com/pepper-project/pequin.

[98] Y. Zhang, A. Steele, and M. Blanton, "PICCO: A general-purpose compiler for private distributed computation," in *ACM CCS*, Nov. 2013.

[99] M. Blum, W. S. Evans, P. Gemmell, S. Kannan, and M. Naor, "Checking the correctness of memories," in *FOCS*, 1991.

[100] L. Abal and U. PLSysSec, "Haskell z3 bindings." https://github.com/PLSysSec/haskell-z3.

[101] J. Bootle, A. Cerulli, J. Groth, S. Jakobsen, and M. Maller, "Arya: Nearly linear-time zero-knowledge proofs for correct program execution," in *ASIACRYPT*, 2018.

[102] B. Braun, "Compiling computations to constraints for verified computation." UT Austin Honors Thesis HR-12-10, Dec. 2012.

[103] R. Brinkmann and R. Drechsler, "Rtl-datapath verification using integer linear programming," in *ASP-DAC/VLSI*, 2002.

[104] Z. Zeng, P. Kalla, and M. Ciesielski, "Lpsat: A unified approach to rtl satisfiability," in *Proceedings Design, Automation and Test in Europe. Conference and Exhibition 2001*, pp. 398–402, IEEE, 2001.

[105] D. Beyer, "Automatic verification of C and Java programs: SV-COMP 2019," in *TACAS*, 2019.

[106] M. Chang, "Performance issue on qf_nira formula. cvc4 issue 5354." https://github.com/CVC4/CVC4/issues/5354.

[107] N. Becker, P. Müller, and A. J. Summers, "The axiom profiler: understanding and debugging SMT quantifier instantiations," in *TACAS*, 2019.

[108] F. Baader and T. Nipkow, *Term rewriting and all that*. Cambridge, UK: Cambridge University Press, 1999.

[109] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli, "CVC4," in *CAV*, 2011.

[110] A. Niemetz and M. Preiner, "Bitwuzla at the smt-comp 2020." arXiv:2006.01621, 2020.

[111] A. Niemetz, M. Preiner, C. Wolf, and A. Biere, "Btor2 , btormc and boolector 3.0," in *CAV*, 2018.

[112] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *TACAS*, 2008.

[113] B. Dutertre, "Yices 2.2," in *CAV*, 2014.

[114] R. Bruttomesso, A. Cimatti, A. Franzén, A. Griggio, and R. Sebastiani, "The mathsat 4 smt solver," in *CAV*, 2008.

[115] W. Wang, C. Barrett, and T. Wies, "Partitioned memory models for program analysis," in *International Conference on Verification, Model Checking, and Abstract Interpretation*, 2017.

[116] R. M. Burstall, "Some techniques for proving correctness of programs which alter data structures," *Machine intelligence*, vol. 7, no. 23-50, p. 3, 1972.
[117] R. Sasnauskas, Y. Chen, P. Collingbourne, J. Ketema, G. Lup, J. Taneja, and J. Regehr, "Souper: A synthesizing superoptimizer." arXiv:1711.04422, 2017.
[118] E. Albert, P. Gordillo, A. Rubio, and M. A. Schett, "Synthesis of superoptimized smart contracts using max-smt," in *CAV*, 2020.
[119] Z. Jia, O. Padon, J. Thomas, T. Warszawski, M. Zaharia, and A. Aiken, "TASO: optimizing deep learning computation with automatic generation of graph substitutions," in *SOSP*, 2019.
[120] Y. Bertot and P. Castéran, *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer Science & Business Media, 2013.
[121] J. Baron, "Securing information for encrypted verification and evaluation." https://web.archive.org/web/20200221151433/https://www.darpa.mil/attachments/SIEVEProposersDaySlidesv4.pdf, 2019. DARPA SIEVE Program Proposers Day Slides.
[122] "Researchers demonstrate potential for zero-knowledge proofs in vulnerability disclosure." https://www.darpa.mil/news-events/2021-04-22, 2021. DARPA Press Release.
[123] F. Tramèr, F. Zhang, H. Lin, J.-P. Hubaux, A. Juels, and E. Shi, "Sealed-glass proofs: Using transparent enclaves to prove and sell knowledge," in *IEEE S&P*, 2017.
[124] L. Breidenbach, P. Daian, F. Tramèr, and A. Juels, "The hydra framework for principled, automated bug bounties," *IEEE Security & Privacy Magazine*, vol. 17, pp. 53–61, July 2019.
[125] L. Breidenbach, P. Daian, F. Tramèr, and A. Juels, "Enter the Hydra: Towards principled bug bounties and exploit-resistant smart contracts," in *USENIX Security*, 2018.
[126] D. Heath and V. Kolesnikov, "Stacked garbling with o(b log b) computation," 2021.
[127] D. Heath and V. Kolesnikov, "Stacked garbling for disjunctive zero-knowledge proofs.," 2020.
[128] C. Baum, A. J. Malozemoff, M. B. Rosen, and P. Scholl, "Mac'n'cheese: Zero-knowledge proofs for boolean and arithmetic circuits with nested disjunctions," in *CRYPTO*, 2021.
[129] D. Heath, Y. Yang, D. Devecsery, and V. Kolesnikov, "Zero knowledge for everything and everyone: fast ZK processor with cached ORAM for ANSI c programs," in *IEEE S&P*, 2021.
[130] K. R. M. Leino and C. Pit-Claudel, "Trigger selection strategies to stabilize program verifiers," in *CAV*, Springer, 2016.
[131] M. Andrysco, A. Nötzli, F. Brown, R. Jhala, and D. Stefan, "Towards verified, constant-time floating point operations," in *ACM CCS*, 2018.
[132] Kaggle, "Compeitions." https://www.kaggle.com/docs/competitions.
[133] ARPA-E, "Grid optimization competition." https://gocompetition.energy.gov/.
[134] Y. Lindell, "Secure multiparty computation (MPC)." https://eprint.iacr.org/2020/300. To appear in Commun. ACM.
[135] A. C. Yao, "Protocols for secure computations," in *FOCS*, 1982.
[136] O. Goldreich, S. Micali, and A. Wigderson, "How to play any mental game or A completeness theorem for protocols with honest majority," in *STOC*, 1987.
[137] M. Ben-Or, S. Goldwasser, and A. Wigderson, "Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract)," in *STOC*, 1988.
[138] D. Beaver, "Efficient multiparty protocols using circuit randomization," in *CRYPTO*, 1991.
[139] Y. Ishai, E. Kushilevitz, R. Ostrovsky, and A. Sahai, "Zero-knowledge from secure multiparty computation," in *STOC*, 2007.
[140] V. Vu, S. Setty, A. J. Blumberg, and M. Walfish, "A hybrid architecture for interactive verifiable computation," in *IEEE S&P*, 2013.
[141] J.-C. Filliâtre and S. Conchon, "Type-safe modular hash-consing," in *Proceedings of the 2006 Workshop on ML*, 2006.
[142] A. P. Ershov, "On programming of arithmetic operations," *Communications of the ACM*, 1958.

APPENDIX A
FULL BENCHMARKS

In this section, we compare CirC's Haskell implementation against the ZoKrates, Circom, and Pequin compilers on a full set of benchmarks. We use the same testbed described in Section V-A.

File	CirC		ZoKrates	
	Constraints	Time (s)	Constraints	Time (s)
ecc/babyjubjubParams	10	0.08	10	0.01
ecc/edwardsAdd	11	0.07	19	0.01
ecc/edwardsCompress	763	3.66	766	0.13
ecc/edwardsNegate	2	0.06	2	0.02
ecc/edwardsOnCurve	6	0.12	9	0.01
ecc/edwardsOrderCheck	40	0.09	60	0.03
ecc/edwardsScalarMult	6394	6.04	9994	0.46
ecc/proofOfOwnership	6350	7.92	9989	0.85
mimc7/mimc7R10	40	0.48	42	0.02
mimc7/mimc7R20	80	0.84	82	0.03
mimc7/mimc7R50	200	1.31	202	0.19
mimc7/mimc7R90	360	1.96	362	0.88
mimcSponge/mimcFeistel	661	7.29	662	11.10
mimcSponge/mimcSponge	2632	46.46	2643	89.39
pedersen/512bit	3062	9.38	3940	1.90
sha256/1024bitPadded	84237	193.11	79575	18.13
sha256/1024bit	57098	122.31	56674	8.82
sha256/1536bit	85791	182.48	87345	16.38
sha256/256bitPadded	27547	61.72	26587	3.80
sha256/512bitPacked	55544	141.51	50398	10.70
sha256/512bitPadded	55544	125.13	48904	8.74
sha256/512bit	28405	60.45	27470	3.18
sha256/shaRound	28949	56.42	28070	2.49
utils/256bitsDirectionHelper	529	0.10	1105	0.02
signatures/verifyEddsa	96765	205.78	99359	22.20
casts/1024to256array	2048	7.80	2048	0.19
casts/bool_128_to_u32_4	132	0.37	132	0.01
casts/bool_256_to_u32_8	264	0.65	264	0.05
casts/u32_4_to_bool_128	256	0.42	260	0.01
casts/u32_8_to_bool_256	512	0.64	520	0.02
multiplexer/lookup1bit	2	0.06	4	0.02
multiplexer/lookup2bit	5	0.08	6	0.01
multiplexer/lookup3bitSigned	7	0.07	8	0.02
bool/nonStrictUnpack256	508	1.49	511	0.05
bool/pack128	128	0.42	129	0.05
bool/pack256	256	1.10	257	0.06
bool/unpack128	382	1.30	635	0.02
u32/nonStrictUnpack256	254	2.05	263	0.11
u32/pack128	128	0.54	133	0.05
u32/pack256	256	0.96	265	0.09
u32/unpack128	254	1.55	511	0.08

Fig. 19. CirC vs. the ZoKrates reference compiler on the ZoKrates standard library

A. ZoKrates-to-R1cs

Our benchmark set is every main circuit in the ZoKrates standard library (v0.6.1), save circuits that only define constants (e.g. hash parameters). We report compilation time and constraint count for CirC and the ZoKrates reference compiler (v0.6.1).

Figure 19 shows the results. Generally, CirC produces slightly fewer constraints. For the elliptic curve cryptography module ("ecc"), CirC produces many fewer constraints, but it produces slightly more for bit-intensive hashing ("sha256"). In almost all cases, the ZoKrates compiler is substantially faster. The most likely cause is the term representation in CirC's Haskell implementation. Terms are recursive Haskell GADTs, and maps from them are either (a) hash maps based on a hash that traverses the full term or (b) tree maps based on the term's full string representation. CirC's Rust implementation represents terms with hash-consing [141], [142]; this yields far better performance.

B. Circom-to-R1cs

We evaluate on every test circuit in the test suite of the Circom standard library (v0.20). We omit circuits that the Circom compiler (v0.30) could not compile (either because the test was invalid or our test bed did not have enough memory).

File	CirC		Circom	
	Constraints	Time (s)	Constraints	Time (s)
aliascheck_test	261	0.10	261	0.30
babyadd_tester	6	0.08	6	0.30
babycheck_test	3	0.09	3	0.27
babypbk_test	776	0.42	776	4.96
binsub_test	49	0.07	49	0.22
constants_test	1	0.08	1	0.13
eddsamimc_test	5712	0.83	5712	19.05
eddsaposeidon_test	4208	1.07	4208	18.44
edwards2montgomery	2	0.08	2	0.15
escalarmulany_test	2554	0.36	2554	1.18
escalarmulfix_test	776	0.44	776	5.00
greatereqthan	65	0.11	65	0.18
greaterthan	65	0.10	65	0.17
isequal	2	0.08	2	0.17
iszero	2	0.07	2	0.18
lesseqthan	65	0.07	65	0.17
lessthan	65	0.09	65	0.17
mimc_sponge_hash_test	2640	0.24	2640	0.71
mimc_sponge_test	660	0.14	660	0.31
mimc_test	364	0.08	364	0.20
montgomery2edwards	2	0.06	2	0.18
montgomeryadd	3	0.05	3	0.13
montgomerydouble	4	0.07	4	0.13
mux1_1	1	0.07	2	0.15
mux2_1	3	0.07	3	0.20
mux3_1	5	0.07	5	0.21
mux4_1	9	0.07	9	0.22
pedersen2_test	701	0.40	701	3.91
sha256_2_test	30134	53.77	30134	80.88
sign_test	262	0.13	262	0.33
smtprocessor10_test	7895	5.07	7895	158.39
smtverifier10_test	4783	2.72	4783	50.12
sum_test	97	0.07	97	0.20

Fig. 20. CirC vs. the Circom compiler on the Circom test suite

Figure 20 shows the results. The compilers produce identical constraint counts in nearly every case. The runtime of CirC is typically better than the reference compiler, which is written in JavaScript.

C. C-to-R1cs

We evaluate on all C-language benchmarks from [5]. We obtain these benchmarks from the Pequin source distribution (https://github.com/pepper-project/pequin). In [5], these benchmarks are parameterized; we use the default parameters from the source distribution.

We apply two transformations to the benchmarks before evaluation. First, since CirC's C front-end does not support Pequin's (bespoke) I/O conventions for multi-dimensional input arrays, we flatten the array inputs to the matrix-multiplication benchmark. Note that CirC's C front-end does support standard C-language multi-dimensional arrays. Second, since CirC's C front-end does not evaluate C constant expressions before type-checking, we evaluate any constant expressions in array lengths.

Additionally, for benchmarks containing Buffet's loop-flattening directives, we apply Buffet's C-to-C flattening pass. This this is a source-to-source transformation, so CirC can benefit from it, just as Pequin can.

Figure 21 shows the results. Generally, CirC produces many fewer constraints. This is because these benchmarks—similar to ptrs-8 from our main evaluation—are bottlenecked on accesses to small arrays. Recall (§V-A) that CirC compiles small arrays more efficiently than Pequin.

Since small array accesses are the bottleneck, we think that this benchmark set portrays CirC in an unfairly favorable light. Nonetheless, we include these benchmarks for consistency with [5], and refer to reader to Section V-A for a fairer (in our view) comparison.

Benchmark	CirC		Pequin	
	Constraints	Time (s)	Constraints	Time (s)
sparse_matvec	5806	1.47	23466	7.63
mm	27000	577.97	27001	17.44
rle_decode	4560	1.10	9847	6.23
mergesort	10400	3.32	19781	5.45
kmp_search	75650	23.13	163664	30.56
ptrchase	168	0.11	1993	1.43
boyer_moore	2016	0.91	5612	2.06

Fig. 21. CirC vs. the Pequin compiler on C-language benchmarks from [5]

APPENDIX B
CIRC DESIGN AND IMPLEMENTATION DETAILS

This appendix provides more information on `Circify` and the optimizations we've implemented in CirC.

A. Additional `Circify` features

a) Stack arrays: `Circify` manages stack arrays of bit-vectors. One can use `Circify` to allocate stack arrays of any static size; allocation returns a concrete *allocation identifier*, `id`. Future loads and stores use `id` to indicate which array to access. Load operations take a concrete `id`, and a symbolic offset. Store operations take a concrete `id`, a symbolic offset, and a symbolic value. The effect of a store operation is guarded on the current path condition. Using this interface, front-ends can easily support accesses to dynamic locations within statically-known arrays.

b) References: `Circify` supports fixed references to (potentially out-of-scope) variables, similar to C++. References are modeled as a kind of *location*, just like a variable. References can be created to any location, and `Circify` allows new values to be written to references. `Circify`'s reference system is useful for capturing C pointers with fixed referents: e.g., pointers used as output arguments.

B. Optimization details

Often, optimization motivated by one pipeline benefit others too.

As one example, while implementing our ZoKrates-to-R1CS compiler, we added a bit-shifting optimization in the constant folding pass. This optimization recognizes terms of the form $\texttt{x} \gg \texttt{k}$ (for bit-vector `x` and constant `k`) and replaces them with a rearrangement of the bits of `x`. The latter ultimately requires substantially fewer R1CS constraints than a variable offset bit-shift.

Since this optimization is implemented over CirC-IR, it can immediately benefit other pipelines too. For example, this optimization was also quite beneficial in the C-to-R1CS pipeline.

Optimizations can also benefit different *targets*. In Section VI, we present experiments demonstrating that some optimizations benefit both SMT and R1CS targets, despite the apparent difference of these targets' execution semantics.

$n \in \mathbb{N}$ $\quad i \in$ *identifiers* $\quad \ell \in$ *constant literals*

$\oplus \in$ *binary ops.* $\quad \ominus \in$ *unary ops.*

$\tau ::= \texttt{u}n \mid \texttt{bool} \mid \texttt{field} \mid \tau[n]$	*types*
$q ::= \texttt{public} \mid \epsilon$	*qual. types*
$d ::= i : q\ \tau$	*declarations*
$t ::= \ell \mid i \mid t \oplus t \mid\ \ominus t \mid t[t] \mid [\vec{t}]$	*terms*
$a ::= t \mid i(\vec{t})$	*atoms*
$c ::= \exists \vec{d}.\ \vec{a}$	*cases*
$r ::= i(\vec{d})$:- $\vec{c}$	*rules*

Fig. 22. The abstract syntax of our Datalog dialect. For a term class represented by x, $\vec{x}$ denotes lists of such terms.

APPENDIX C
ZKLOG ABSTRACT SYNTAX

In Figure 22, we give the abstract syntax of ZKlog: our Datalog-based input language. A *type* is a boolean, field element, fixed-width unsigned integer, or a fixed-size array. A *rule* holds for some input variables if any *case* holds. Each case can existentially quantify variables, and holds if a conjunction of *atoms* holds for some assignment to the variables. Each atom is a *term* or a rule applied to some terms. A term can be a literal, variable, indexed array, or an operator applied to other terms. A *program* is a collection of rules, including a distinguished *entry* rule, e.g., a rule called `main`.

Types can be qualified as *public*. For the purpose of a zero knowledge proof, inputs to the entry rule are public if so qualified and private otherwise. All variables quantified by cases are private.

APPENDIX D
OPENSSL BUG DETAILS

Figure 23 illustrates the bug in the OpenSSL macro `mul_add_c2` (Figure 16). Double and add operations can overflow by a single bit; these overflows are handled by conditional increments to $\texttt{c}_2$ and $\texttt{t}_1$. However, the conditional increments themselves can also overflow. In some cases, this does not introduce a bug. Overflows in $\texttt{c}_2$ can be soundly ignored, since this macro implements triple-word arithmetic. The first conditional increment to $\texttt{t}_1$ cannot overflow, because $\texttt{t}_1$ must be even before the increment. However, if the second condition increment to $\texttt{t}_1$ (circled with a dashed red line) overflows, $\texttt{c}_2$ should be, but is not, incremented again.

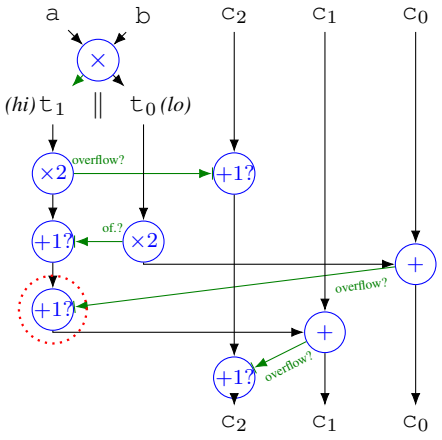

Fig. 23. Dataflow for the code in Figure 16. Conditional increments handle overflow. If the operation circled in red overflows, $\texttt{c}_2$ should be (but is not) incremented again.