Abstract
The programming languages community, the cryptography community, and others rely on translating programs to high-level source languages (e.g., C) to logical constraint representations. Unfortunately, building compilers for this task is difficult and time consuming. In this work, we show that all of these communities can build upon a shared compiler infrastructure, because they all share a common abstraction: stateless, non-deterministic computations that we call existentially quantified circuits, or EQCs.

To make our approach concrete we create CirC, an infrastructure for building compilers to EQCs. CirC makes it easy to add support for new EQCs: we build support for two, one used by the PL community and one used by the cryptography community, 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 ≈700 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.

1 Introduction
The programming languages and formal methods communities have a long tradition of translating programs to logical constraints (e.g., Satisfiability Modulo Theories [14] (SMT) formulas) to verify program properties [11, 26, 43, 83, 106, 120], synthesize new programs [74, 113, 114], and more. The cryptography community, meanwhile, compiles programs to boolean circuits, arithmetic constraints, and similar representations used by probabilistic proof systems [118, 131, 138] (which give efficiently verifiable, privacy-preserving proofs) and multi-party computation [87] (which enables collaborative computation among mutually distrusting parties). Still other communities compile to integer linear programs [58], 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 call this “the most difficult aspect of creating solver-aided tools,” taking "years to develop" [120]. As a result, communities that rely on constraint compilers have poured enormous effort into building them (§2.4), with 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!

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. 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 SMT formula is an EQC that is "executed" by an SMT solver. Since SMT formulas are sets of logical relations, they are stateless and free of control flow. Moreover, SMT formulas can include existentially quantified variables, i.e., ∃x.P(x) for a predicate P. An SMT solver executes such formulas by finding an x satisfying P(x) or determining that no such x exists.

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’s a well-known procedure: 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 [43]. That same procedure is also necessary when compiling C to boolean circuits for multi-party computation [89], or to arithmetic constraints for proof systems [111, 130]. Since this approach is largely independent of the application, sharing compilation infrastructure avoids duplicated effort.

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 [39, 44, 64, 82, 112, 135] and proof systems [22, 24, 31, 76, 100, 130]. We show that proof system

Note that EQCs do not capture digital circuits, which are stateful and deterministic; thus, we do not consider them in this work. See Section 2.4.

While some SMT solvers also support universal quantifiers, purely existential formulas are already quite useful, and thus are our focus in this work. We hope to explore quantification more broadly in future work.
and software verifier performance both improve under the same memory optimizations (and more) in Section 6.

Finally, we show that shared compiler infrastructure yields benefits with few analogs in traditional compilers. In a traditional compiler, each target CPU supported by the compiler does 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 7 that our work can automatically identify a bug using a verification pipeline, then prove the existence of that bug without revealing how to trigger it, using a proof pipeline.

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 [80], 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 (§2.1). 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., SMT constraints), allowing them to “run” the resulting executable (e.g., feed the constraints to an SMT solver).

We evaluate CirC for two radically different use cases—automated verification and cryptographic proof systems—and show that it allows compiler implementors to:

- Easily support new front-end languages (§4). CirC currently supports a rich subset of C (∼2000 LOC); Circom [15], a domain-specific language for proof systems (∼1200 LOC); and ZoKrates [139], a language for embedding proofs in smart contracts (∼700 LOC). For example, our ZoKrates compiler implements the full language specification, is an order of magnitude smaller than the language’s reference compiler (24000 LOC), and was much easier to build (one person in one week vs. 36 contributors over three years).
- Easily support new EQC back-ends (§4). CirC currently supports both SMT constraints (∼400 LOC) and constraints for proofs (called R1CS [22]; ∼1600 LOC).
- Create correct, efficient EQCs (§5). For example, CirC outperforms the ZoKrates reference compiler.
- Write optimizations that help multiple targets (§6).
- Easily combine back-end functionalities (§7). We are the first to use an SMT solver to optimize R1CS (§7.2), and the first to combine SMT and R1CS to automatically find bugs and then prove their existence (§7.1).

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. We begin with background on EQCs, our use cases, and related work (§2), then illustrate the compiler’s design (§3), evaluate CirC (§4–§7), and discuss limitations and next steps (§8).

2 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 (§4–§7), we discuss our two example use cases: automated verification (§2.2) and cryptographic proof systems (§2.3). Finally, we describe related work (§2.4).

2.1 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 $C = A \land B$).

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 bitwise XOR: when $A = 1$, $B$ must take the value 1.
In complexity-theoretic terms, we say that EQCs capture non-deterministic, non-uniform computation [5, 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.

### 2.2 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 \( x \lor y \), 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 (§2.1).

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, roughly analogous to writing code in a high-level language compared to writing code in assembly. The SMT-LIB [14] standard specifies the semantics of each theory.

**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., \( \text{index is within bounds of } \text{array} \)) 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.

Translating (or compiling) source code into SMT formulas is a non-trivial task [43, 120, 133]. Since SMT does not support mutable variables, the verifier must transform high-level code to static single assignment (SSA) form. Since SMT doesn’t support control flow constructs, it must unroll all loops (up to a bound) and replace mutations inside conditional branches with if-then-else terms that apply the mutation when the corresponding branch in the source program would have been taken. For example, consider this code snippet and its SMT-compatible representation:

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

```c
// STM-compatible program
x_{1} = x_{0} < 20 ? 2 : x_{0};
y_{1} = !(x_{0} < 20) ? (y_{0} + z_{0}) : y_{0};
```

For this snippet, the verifier transforms the conditional into assignments guarded by the branch condition or its negation. \( x \) is set to \( z \) when the condition evaluates to true; otherwise, \( z \) is added to \( y \). The result uses no mutation or conditionals.

### 2.3 Cryptographic proof systems

Probabilistic proof systems are powerful cryptographic tools whose applications include verifying that outsourced computations are executed correctly [100, 102, 136], implementing private cryptocurrency transactions [21, 51], and defending against hardware back-doors [128, 129]. In this section, we describe the class of probabilistic proof systems CirC targets, focusing on their computational model. Readers should consult surveys [118, 131] for additional details.

At a high level, a probabilistic proof system is a cryptographic protocol between two parties, a prover \( P \) and a verifier \( V \), whereby \( P \) produces a short proof that convinces \( V \) that \( \exists w. y = \Psi(x, w) \), for \( \Psi \) a computation that takes input \( x \) and witness \( w \) and returns output \( y \). Several lines of work [22–24, 31, 47, 59, 76, 102] instantiate end-to-end built systems. Two key features of these systems are succinctness—\( P \)’s proof is small, as is \( V \)’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 \( P \) to generate a proof and \( V \) to verify it. The compilation stage, our focus in this work, takes a source program \( \Psi \) (written, say, in C) and transforms it into a system of arithmetic constraints \( C \) in vectors of formal variables \( W, X, Y \), such that \( \exists w, y. \Psi(x, w) \iff \exists W. C(W, X, Y) \) for \( X = x \), \( Y = y \). (Note that \( \exists W. C(W, X, Y) \) is an EQC; §2.1.) The primary figure of merit for a compiler is the size of \( C \): fewer constraints means less work for \( P \) to generate a proof, and in some cases a shorter proof that is easier for \( V \) to verify.

**The constraint formalism.** The formalism used by most proof systems is called a rank-1 constraint system (R1CS). An R1CS instance \( C \) 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 \( C \) equals a third weighted sum, which generalizes arithmetic circuits. \( C \) is satisfied when the values \( W, X, Y \) satisfy all constraints. To generate a proof, \( P \) first computes a satisfying assignment and then executes the cryptographic proving machinery.

Compiling programs from languages like C to R1CS instances is tricky: as with SMT formulas, constraints cannot
directly encode mutation, control flow, etc., so the compiler must transform the program as described in Section 2.2. What’s more, all computations must be written in terms of 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 \( \mathcal{O}(\log p) \) constraints; \( p = 2^{256} \) 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., [76, 100, 111, 130].

### 2.4 Related work

CirC is related to and inspired by LLVM [80] and SUIF [67], but CirC targets EQCs instead of CPUs. MLIR [81] aims to enhance LLVM 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 ([52] 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, proofs, 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 [39] combines a core constraint compilation engine with a path exploration front-end that forks at every branch; and Giraffe [129] uses program analysis to “slice” programs, then compiles a subset of these slices to proof system constraints. CirC could take the place of the respective compilation phases if paired with an exploration or slicing phase (§8).

#### Compilers for SMT

Many projects compile high-level programs or parts of programs to SMT in order to prove program properties [8, 9, 43, 45, 46, 48, 71, 73, 77, 84, 116, 125, 126, 133] (e.g., using bounded model checking). For example, the CBMC verifier [43, 77] 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) [88, 90, 98, 123]. For example, Alive [88] 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 [56, 75, 91, 122]. For example, the Kaplan language [75] allows users to manipulate constraints directly from within the Scala language (e.g., in order to branch on the satisfiability of a formula).

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, 57, 83, 94, 106, 119, 120]—often by handling the details of SMT compilation—and many verifiers that depend on them [10, 66, 95, 96]. Intermediate Verification Languages [11, 57, 94, 117] (IVLs), for example, decouple language details from verifier details; the Boogie IVL [11] targets SMT-like back-ends, and building a new verifier just requires translating a source language to the IVL. CirC has a similar but broader goal: it aims to decouple source languages from all EQC targets, not just verifiers—and therefore must generalize lessons from IVLs to a more diverse range of constraint systems.

Symbolic execution (symex) tools [7, 40] combine SMT compilation of a single program path with, typically, either concrete execution or forking strategies [34, 39, 42, 61, 62, 105, 107, 109, 115, 135]; some systems use a hybrid of symex and model checking [120]. After SMT compilation, different tools proceed differently: many [39] fork execution at each conditional jump, some use a mixture of concrete and symbolic (concolic) execution [109, 135], some combine static analysis and symex [34], some combine symex and fuzzing [115, 135], and more. In each case, the symbolic tool relies on a core component that compiles programs to SMT.

#### Compilers for cryptography

A long line of prior work develops techniques for compiling to R1CS constraints. Ginger [111], Zaatar [110], Pantry [31], and Buffet [130] compile a subset of C and support proof-specific optimizations for rational numbers, memory, key-value stores, complex control flow, etc. Pinocchio [102] also compiles a subset of C with techniques similar to Ginger’s. Geppetto [47] consumes LLVM IR and provides efficient cryptographic primitives. xJsnark [76] consumes a Java-like language and refines techniques from Buffet and Geppetto. Zinc [137] and ZoKrates [139] compile from eponymous DSLs to R1CS using existing techniques. Finally, Circom [15] is essentially a hardware description language that relies on the programmer to write constraints. We build Circom and ZoKrates support for CirC in Section 4.

Another line of work [22, 24] uses hand-crafted constraints that simulate a simple CPU, then modifies GCC to emit code for that CPU; while conceptually simple, this comes at enormous cost [130, §5]. Yet another approach, embodied by...
Unifying Compilers for SNARKs, SMT, and More

Proof
R1CS
Opt.

Unifying Compilers for SNARKs, SMT, and More
with stateful semantics, designers create a front-end that (e.g., for C) and new back-ends (e.g., for R1CS). Front-ends
interpreters for their language (somewhat similar to Serval [96]);
Circify IR with a notion of finite fields, which underly R1CS (§2.3).
languages geared towards proof systems (§4), we extend the
ries cover common primitive types in high-level languages,
floating-point numbers, bit-vectors, and arrays. These theo-
SMT-LIB standard (§2.2), limited to formulas over booleans,
Compiling to CirC’s IR.
To add support for a new language
the CirC compilation pipeline and how language designers
specific optimizations. In the rest of this section, we describe
an EQC, it can still
While BoolPair only includes booleans,
when the interpreter must handle control flow (e.g., branches or breaks) or the environment (e.g., variable mutations),
designers invoke Circify functions. Circify handles the
complexity of translating mutable state and control flow to IR,
using standard techniques like path guarding, versioning
variables, guarding mutations, and transforming memory to
SMT array operations (§2.2). Using Circify requires designers
to define a few features of their language; we describe
these features and walk through their instantiation for a
language with booleans and pairs (BoolPair, Fig. 3).
First, designers define an IR representation of symbolic
values for their language. In BoolPair, a value is either a
boolean or a pair of values (implemented in the Val data
type in Figure 3). Circify’s internal state stores a mapping
of variables to symbolic values. The mapping is generic over
value type, so Circify can automatically track program state
regardless of the language’s underlying value representation.
While BoolPair only includes booleans, Circify supports
any value type that can be embedded in CirC-IR.
Next, Circify requires designers to specify two functions,
ite and assign (Fig. 3). Since Circify automatically transforms
mutable variables into SSA form, it must be able to
create fresh variables and assign existing values to them; it
does this using assign. In BoolPair, assign creates a fresh
boolean variable, then asserts its equality to an existing value
(lines 7–9). For a pair, the assignment is recursive (lines 10–
12), appending an index number to the names for the first
and second values.
Circify uses ite to construct values that depend on
the current path condition. This is necessary for compiling
control flow (e.g., turning "if (x) { y = z; }" into
\texttt{y\_1 = if x\_0 then z\_0 else y\_0\}).

Figure 2. CirC’s architecture, with extensions (§3).

libsnark [86], ZEXE [29], and Bellman [18], uses a “macro assembler” to compose hand-crafted R1CS “gadgets.”
Fairplay [89] is the earliest example of a compiler to cir-
cuits for two-party computation (2PC); FairplayMP [19] tar-
gets multi-party computation. Later works like Tasty [68]
and HyCC [38] optimize by matching pieces of the source
program to suitable cryptographic protocols.
CBMC-GC [69] adapts the CBMC [43] model checker to
emit boolean circuits for 2PC, implicitly leveraging the simi-
larities 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 (§7).

3 Design
CirC’s main goal is extensibility: it should be easy for design-
ers to add support for new source languages and 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, which is similar to SMT-LIB (§2.2), and
back-ends lower from IR to a target representation. Figure 2
shows how CirC is extensible at each stage of the compila-
tion pipeline. For example, even though the hardware design
language Circom is essentially already an EQC, it can still
plug into CirC’s R1CS back-end and take advantage of target-
specific optimizations. In the rest of this section, we describe
the CirC compilation pipeline and how language designers
can use it to easily create new compilers.

Compiling to CirC’s IR. To add support for a new language
with stateful semantics, designers create a front-end that
translates their language to CirC’s IR. The IR is based on the
SMT-LIB standard (§2.2), limited to formulas over boolean,
floating-point numbers, bit-vectors, and arrays. These theo-
ries cover common primitive types in high-level languages,
but one could extend the IR with other theories. In service of
languages geared towards proof systems (§4), we extend the
IR with a notion of finite fields, which underly R1CS (§2.3).
To help designers compile from high-level languages to IR,
CirC exposes Circify, a library for managing IR-embedded
state and control flow. Designers write (essentially) inter-
preters for their language (somewhat similar to Serval [96]);

```
data Val = Bool IR.Bool
       | Pair Val Val

instance Circify.Embeddable Val where
  -- assign :: String -> Val -> Circify.Circify Val
  assign name t = case t of
    Bool b -> do var <- Circify var name IR.SortBool
                   Circify assert $ IR.Eq var b
       return $ Bool val
    Pair a b -> do a' <- assign (name ++ ".1") a
                  b' <- assign (name ++ ".2") b
       return $ Pair a' b'

  -- ite :: IR.Bool -> Val -> Val -> Circify.Circify Val
  ite c t f = case (t, f) of
    (Bool tb, Bool tf) -> return $ Bool $ IR.Ite c tb tf
    (Pair t1 t2, Pair f1 f2) ->
      liftM2 Pair (ite c t1 f1) (ite c t2 f2)
  _ -> error "Cannot ITE different term types"
```
evalStmt :: AST.Stmt -> BoolPair ()
evalStmt s = case s of
  AST.Assign var expr -> do
    e <- evalExpr expr
    Circify.ssaAssign var e
  AST.If cond true false -> do
    c <- aslBool <$> evalExpr cond
    Circify.guarded c $ Circify.scoped $ evalStmt true
    Circify.guarded (IR Not c) $ Circify.scoped $ evalStmt false
  ...
evalExpr :: AST.Expr -> BoolPair Val
evalExpr = ...
Unifying Compilers for SNARKs, SMT, and More

![Table showing line counts for various extensions of CirC.](image)

**Figure 5.** Lines of code in each CirC extension (§4). We exclude ASTs and parsers from the line count since they are orthogonal to this work and sometimes come from libraries.

competitive with—and often better than—the ZoKrates reference compiler (§5). In this section, we describe the ZoKrates language and the process of writing a CirC front-end for it.

**ZoKrates.** ZoKrates is a new (2018) language for programming cryptographic 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: 36 contributors have authored over 2400 commits over the past several years, and, for example, the financial company Deloitte has used the language to prove statements about tax obligations [65]. ZoKrates includes both an optimizing compiler targeting R1CS and tooling for embedding the resulting proofs in Ethereum smart contracts.

ZoKrates’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, but no form of data-dependent control flow (e.g., no if-statements). Finally, ZoKrates has a sophisticated module system, including support for renaming imports (e.g., from-import-as directives).

**Parsing and lexing.** To support a new language in CirC, the first step is to build (or acquire) a parser. Since there is no existing Haskell library for parsing ZoKrates, we implement our own. We define an AST (145 LOC), generate a span-tracking lexer using Alex [3] (201 LOC), and generate a span-tracking parser using Happy [60] (253 LOC). We also write a file loader which recursively loads imported files (53 LOC), using existing machinery from our Circom front-end.

**Setting up Circify.** In order to use Circify for easy compilation to CirC-IR, designers must define a value type, an assign function, and an ite function (§3). ZoKrates’s primitive types are straightforward: booleans become SMT-LIB booleans, bounded integers become bit-vectors, and field elements are directly supported by CirC-IR. Arrays become lists of values and structures become maps from structure field names to values. The assign and ite functions have straightforward recursive definitions, similar to Figure 3.

**Compiling ZoKrates to CirC IR.** The next step is compiling from the ZoKrates AST, which has stateful semantics, into CirC-IR. Circify handles the hard parts—e.g., the transformation from stateful to stateless semantics—for us; all we need is to write an interpreter. To do so, we first write functions that translate basic ZoKrates operations (e.g., binary operators or array accesses) into IR. We then use Circify-provided functions to handle control flow and assignment, as described in Section 3.

The main complexity is handling ZoKrates’s module and import system: since import directives can rename imported identifiers, function and structure names depend on the current module. For example, a structure might be called S in its defining module and be imported into another module as S’. To handle this, the interpreter modifies function and structure lookups based on the current module (≈50 LOC). The interpreter handles other small complexities, too: it special-cases built-in functions (≈40 LOC) and attaches source-code spans to errors (≈20 LOC). Finally, the interpreter must embed the ZoKrates main function in CirC-IR, marking ZoKrates’s “private” inputs as existentially quantified (§2.1) in the resulting EQC (≈20 LOC). In total, compiling from ZoKrates AST to CirC-IR requires 656 LOC.

**Optimizing.** While our compiler is now functional, recall that, for performance reasons, it should generate as few constraints as possible (§2.3). To this end, we implement additional optimizations over IR and in IR-to-R1CS lowering.

One example is constant folding for bit-shifts: while our constant folding optimization previously folded constants shifted by constants, it ignored terms (i.e., non-constants) shifted by constants. We modify our optimization pass to replace such shifts with a combination of bit-vector extractions and concatenations. This is a well-known SMT rewrite [49], and it also improves the generated R1CS.

While the proximate motivation for this and other improvements was the ZoKrates pipeline, we emphasize two key points: first, each change is modular and is not particular to one front-end or back-end. Second, each change also results in an immediate benefit for the C-to-R1CS pipeline—and each is likely to help future pipelines, too.

**Discussion.** Because Circify handles the details, our front-end is much simpler than the ZoKrates reference compiler. Furthermore, our front-end is more extensible: with Circify, adding data-dependent control-flow is easy—we did so in four lines of code, similar to lines 6–10 of Figure 4 (§3). In contrast, extending the reference compiler in a similar way appears to require a substantial redesign.

5 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 measure 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? (§5.1)
- Emit SMT circuits that capture C program semantics with enough fidelity to find simple bugs? (§5.2)

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 [25] verifier competition.

5.1 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 (§2.3; [76, 100, 130]).

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 [103], a state-of-the-art compiler from C to R1CS that builds on a long line of work [31, 110, 111, 130]. We use 6 benchmarks from the Pequin software distribution covering a representative sample of control-flow patterns and primitive operations. 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 Pequin and CirC produce, which is higher when CirC performs better.

Figure 6 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 because of 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 [76]), whereas Pequin uses a single memory representation that is asymptotically cheap yet concretely costly for small arrays.

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. 15), CirC outperforms Pequin slightly.

ZoKrates-to-R1CS. We evaluate CirC’s ZoKrates-to-R1CS pipeline relative to the ZoKrates compiler (v0.6.1). 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). As above, we report the ratio of constraint counts.

Figure 7 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.

**Circom-To-R1CS.** Circom [15] 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 (§3). We evaluate CirC against the Circom compiler (v0.0.30) on a representative subset of Circom’s standard library, including: utilities (e.g., binary arithmetic, multiplexers) hashes, elliptic curve operations, and signatures.

The compilers perform identically on all benchmarks, reflecting the fact that Circom programs explicitly describe constraints, and the compilers apply identical optimizations.

### 5.2 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, we can run CirC on sv-comp benchmarks.

We run CirC on two benchmark categories: `signedintegeroverflow-regression`, which tests the precision with which overflow is modeled, and `bitvector-loops`, which tests the precision with which conditional branches, stack arrays, and basic pointers-to-stack-arrays are modeled. We choose these categories because they exercise the majority of CirC’s front-end support for C semantics (§3). CirC is fully 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 [106]). 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 [39]). We discuss combining CirC with existing verifiers and high-performance verification strategies in Section 8.

### 6 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 much more difficult to understand—and can sometimes be surprising [17, 41]. 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, (§6.1), and that CirC’s **oblivious array elimination** pass (§6.2) and **granular array modeling** (§6.3) help both solver and proof system back-ends.

#### 6.1 Constant folding

**SMT term rewriting**—replacing one SMT term with an equivalent one to assist in SMT solving—is an old technique [6] used in all major solvers [13, 35, 36, 53, 54, 97]. **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 a combination of 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.\(^6\)

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 $x$.

Figure 8 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.\(^7\)

#### 6.2 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 $N$ ints (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

---

\(^6\)The hash is loosely inspired by SHA1, using bit rotations, sums, and XORs.

\(^7\)Because the SMT solver perfoms a learned, 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.
ax and proof system performance when operating on the predicate \( \exists x. H(x)[0..8] = 0 \) (§6.1). Constant folding improves performance for both back-ends.

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 9 shows the results with and without the optimization, with varied \( N \). Both targets perform better when array elimination is enabled.

6.3 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 [34], it is generally more expensive for SMT solvers to reason about [37, 132]. 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 implemented as an array, so applying the permutation is just an array index operation (the program assumes that the input is in bounds). To evaluate the effect of array granularity, we apply a source-level transformation that fuses separate permutations into shared arrays, which simulates coarser- or finer-grained arrays (e.g., fusing all arrays simulates the “stack” approach discussed above).

Figure 10 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 11 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.

7 Crossover applications and techniques

CirC’s different targets serve substantially different purposes, opening the door to applications that combine targets, and to techniques that use one target to help another. In this section, we discuss two such cross-overs: zero-knowledge detection and proof of bugs, and SMT-driven optimization of R1CS size. CirC’s common infrastructure makes cross-overs easy: in ≈100 LOC, we create the first pipeline for automatically finding bugs and proving their existence in zero knowledge (§7.1); in sixteen LOC, we build the first tool to use SMT queries to optimize R1CS circuits (§7.2).

```c
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);
}
```

Figure 10. Array granularity benchmark: this C program applies two permutations, which are implemented as arrays, to its input (§6.3).
Figure 11. Effect of array granularity on solver and proof system performance (§6.3). Increasing x-axis corresponds to increasingly coarse-grained array representations, which increase costs for both the SMT and proof back-ends.

7.1 Automatically finding and proving bugs

Over the past decade, many companies have started bug bounty programs. These programs offer cash rewards in exchange for legitimate bug reports, incentivizing researchers to report—rather than exploit—zero-day vulnerabilities. Although bounty programs have been successful in practice [85], the literature presents lingering concerns about everything from economic incentives to fairness [32, 33, 78, 79]. For example, once a reporter discloses a bug, they lose their bargaining power: if the vendor doesn’t pay the promised bounty, the reporter has little (or slow) recourse (e.g., [4, 124]).

One proposal to address some of these problems is to use zero-knowledge proofs (§2.3) to report bugs without revealing their details [104, 121]; a recent DARPA program solicited solutions to precisely this problem [12]. This could work roughly as follows: first, a company offering a bounty indicates which program properties they expect to hold, e.g., by embedding assertions in the code. Then, when security researchers find a bug, they construct a zero-knowledge proof that there exists an execution path on which some assertion is violated, and submit the proof to the vendor. Upon verifying the proof, the company is convinced that the researchers have found a bug, and the parties negotiate details and payment. This requires additional machinery (e.g., smart contracts); we discuss in Section 8.

While prior work [32, 33, 121] constructs manual proof-of-bug pipelines, none of them can automatically detect bugs and then automatically prove 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 R1CS. In fact, even proving the presence of many types of bugs is beyond the reach of existing R1CS compilers like Pequin [103] (§5.1), because they model language semantics too imprecisely. In contrast, CirC makes both bug proving and bug finding easy: CirC can model language semantics precisely (§5.2) and can embed those semantics into both SMT (to find bugs) and R1CS (to prove bugs’ existence).

As a proof of concept, we augment CirC’s C front-end with support for SMT assertions over C integers (∼60 LOC) and some glue code (∼40 LOC). This lets CirC automatically (1) detect and (2) write a zero-knowledge proof for the bug underlying CVE-2014-3570 [50, 99] in OpenSSL. The bug is in the macro mul_add_c2 (Fig. 12), which is intended to compute \( c = c + 2*a*b \), where \( c \) is a multi-precision integer.
### 7.2 Optimizing R1CS using SMT

SMT-guided optimization is an old idea, and SMT solvers have optimized everything from code [108] to smart contracts [2] to tensorflow graphs [72]. Circify 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 15 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 (§2). 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 Circify to use an SMT solver to determine the maximum number of iterations of a loop. Mechanically, we add nine lines of code to Circify and add a function to Circify that asks the SMT solver whether the current path condition is feasible; we add seven lines to the C front-end to stop unrolling loops in the case of an infeasible path, plus a few lines to Circify’s commandline interface to control this feature. Obviously this approach cannot work for all programs, but it is quite effective for some: for the \( \text{u32log2} \) function of Figure 15, Circify and the SMT solver determine that \( N = 32 \) in well under one second.

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.

#### 8 Discussion, future work, and conclusion

**Targeting other applications.** Circify has applications beyond SMT solvers and proof systems. As one example, Circify could support *multi-party computation* (MPC), which enables mutually distrusting parties to collaboratively evaluate a function while revealing only the result [87]. MPC frameworks require the function to be expressed as a boolean [134], or arithmetic circuit [16, 20, 63], so extending Circify 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 [70].

As another example, Circify could support *optimization problems* (e.g., integer linear programming, or ILP), which maximize an objective function subject to a set of constraints [38]. Augmenting Circify with a notion of objective functions and adding a back-end for an appropriate constraint format would enable compiling high-level languages to optimization problems. One intriguing application of an optimization back-end is in service of the compilation process itself, i.e., adding optimization passes very roughly analogous to SMT-guided optimization (§7.2). Exploring this is future work.

**Program analysis infrastructure.** Circify supports IR-level optimizations, but sophisticated static analysis infrastructure—at both the language and IR level—would improve most compilation pipelines. For example, Circify 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 [38, 68, 127, 129]. Designing new analyses of this kind is also future work.

**Combining Circify with existing verifiers.** It might also be interesting to combine Circify with modern verification machinery. For example, Circify could benefit from SMACK’s [106] front-end-based optimizations, while Boogie front-ends [11] could benefit from targeting cryptographic applications.

**Challenges for proof-based bug reporting.** While the idea of bug bounties using zero-knowledge proofs has seen recent interest [12, 104, 121], proofs are not a drop-in replacement for existing bug-reporting mechanisms. First proof-based bounties require extra machinery to enforce fairness (e.g., forcing the reporter to reveal bug details before payment; this is not a problem today because reporting the bug naturally reveals the details). Existing work proposes automating this using smart contracts [32, 33, 121]; cryptographic fair exchange protocols may offer another solution [101].

Second, proof-based bounties require the definition of a bug to be precise enough that, once compiled to an EQC, any satisfying assignment encodes a real bug. This is a departure because, in traditional bounties, reporters and triage teams...
often interact, both to determine whether a report represents a real bug and to subsequently reproduce and fix it.

Third, existing bounty programs often support massive software artifacts (e.g., Mozilla Firefox, which comprises 21 million LOC [1, 92]). Compiling such a large program would yield an immense EQC. Meanwhile, proof system costs grow with EQC size (§2.3)—so a naive approach (i.e., compiling the entire program to prove one bug) is far too costly to be feasible for today’s proof systems [130, §5]. Better proof systems and new approaches seem necessary.

**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**

[8] Thomas Ball, Ella Bouunimova, Vladimir Levin, Rahul Kumar, and Jakob Lichtenberg. 2010. The static driver verifier research platform. In CAV.
Unifying Compilers for SNARKs, SMT, and More


Nuno P. Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. 2015. Provably correct peephole optimizations with alive. In PLDI.


David Menendez, Santosh Nagarakatte, and Aarti Gupta. 2016. Alive-FP: Automated verification of floating point based peephole optimizations in LLVM. In SAS.


Eric Mullen, Daryl Zuniga, Zachary Tatlock, and Dan Grossman. 2016. Verified peephole optimizations for CompCert. In PLDI.


Andres Nötzli and Fraser Brown. 2016. Lifecjacket: verifying precise floating-point optimizations in LLVM. In SOAP.


Pequin: A system for verifying outsourced computations and apply-...


