Bounded Verification for Finite-Field-Blasting

In a Compiler for Zero Knowledge Proofs

Alex Ozdemir[1](✉), Riad S. Wahby[2], Fraser Brown[2], and Clark Barrett[1]

[1] Stanford University, Stanford, USA
aozdemir@cs.stanford.edu
[2] Carnegie Mellon University, Pittsburgh, USA

Abstract. Zero Knowledge Proofs (ZKPs) are cryptographic protocols by which a prover convinces a verifier of the truth of a statement without revealing any other information. Typically, statements are expressed in a high-level language and then compiled to a low-level representation on which the ZKP operates. Thus, *a bug in a ZKP compiler can compromise the statement that the ZK proof is supposed to establish.* This paper takes a step towards ZKP compiler correctness by partially verifying a *field-blasting* compiler pass, a pass that translates Boolean and bit-vector logic into equivalent operations in a finite field. First, we define correctness for field-blasters and ZKP compilers more generally. Next, we describe the specific field-blaster using a set of encoding rules and define verification conditions for individual rules. Finally, we connect the rules and the correctness definition by showing that if our verification conditions hold, the field-blaster is correct. We have implemented our approach in the CirC ZKP compiler and have proved bounded versions of the corresponding verification conditions. We show that our partially verified field-blaster does not hurt the performance of the compiler or its output; we also report on four bugs uncovered during verification.

1 Introduction

Zero-Knowledge Proofs (ZKPs) are powerful tools for building privacy-preserving systems. They allow one entity, the *prover* $\mathcal{P}$, to convince another, the *verifier* $\mathcal{V}$, that some secret data satisfies a public property, *without revealing anything else about the data.* ZKPs underlie a large (and growing!) set of critical applications, from billion-dollar private cryptocurrencies, like Zcash [24,53] and Monero [2], to research into auditable sealed court orders [20], private gun registries [26], privacy-preserving middleboxes [23], and zero-knowledge proofs of exploitability [11]. This breadth of applications is possible because of the generality of ZKPs. In general, $\mathcal{P}$ knows a secret *witness* w, whereas $\mathcal{V}$ knows a *property* ϕ and a public *instance* x. $\mathcal{P}$ must show that $\phi(x, w) = \top$. Typically, x and w are vectors of variables in a finite field $\mathbb{F}$, and ϕ can be any system of equations over the variables, using operations $+$ and $\times$. Because ϕ itself is an

C. Enea and A. Lal (Eds.): CAV 2023, LNCS 13966, pp. 154–175, 2023.
https://doi.org/10.1007/978-3-031-37709-9_8

input to $\mathcal{P}$ and $\mathcal{V}$, and because of the expressivity of field equations, a single implementation of $\mathcal{P}$ and $\mathcal{V}$ can serve many different purposes.

Humans find it difficult to express themselves directly with field equations, so they use *ZKP compilers*. A ZKP compiler converts a high-level predicate ϕ' into an equivalent system of field equations ϕ. In other words, a ZKP compiler *generalizes* a ZKP: by compiling ϕ' to ϕ and then using a ZKP for ϕ, one obtains a ZKP for ϕ'. There are many industrial [3,5,6,14,21,45,55,66] and academic [4,18,28,29,46,48,50,54,63] ZKP compilers.

The correctness of a ZKP compiler is critical for security— a bug in the compiler could admit proofs of false statements— but verification is challenging for three reasons. First, the definition of correctness for a ZKP compiler is nontrivial; we discuss later in this section. Second, ZKP compilers span multiple domains. The high-level predicate ϕ' is typically expressed in a language with common types such as Booleans and fixed-width integers, while the output ϕ is over a large, prime-order field. Thus, any compiler correctness definition must span these domains. Third, ZKP compilers are evolving and performance-critical; verification must not inhibit future changes or degrade compiler performance.

In this work, we develop tools for automatically verifying the *field-blaster* of a ZKP compiler. A ZKP compiler's field-blaster is the pass that converts from a formula over Booleans, fixed-width integers, and finite-field elements, to a system of field equations; as a transformation from bit-like types to field equations, the field-blaster exemplifies the challenge of cross-domain verification.

Our paper makes three contributions. First, we formulate a precise correctness definition for a ZKP compiler. Our definition ensures that a correct compiler preserves the completeness and soundness of the underlying ZK proof system.[1] More specifically, given a ZK proof system where statements are specified in a low-level language L, and a compiler from a high-level language H to L, if the compiler is correct by our definition, it extends the ZK proof system's soundness and completeness properties to statements in H. Further, our definition is preserved under sequential composition, so proving the correctness of each compiler pass individually suffices to prove correctness of the compiler itself.

Second, we give an architecture for a verifiable field-blaster. In our architecture, a field-blaster is a set of "encoding rules." We give verification conditions (VCs) for these rules, and we show that if the VCs hold, then the field-blaster is correct. Our approach supports *automated* verification because (bounded versions of) the VCs can be checked automatically. This reduces both the up-front cost of verification and its maintenance cost.

Third, we do a case study. Using our architecture, we implement a new field-blaster for CirC [46] ("SIR-see"), an infrastructure used by state-of-the-art ZKP compilers. We verify bounded versions of our field-blaster's VCs using SMT-based finite-field reasoning [47], and show that our field blaster does not compromise CirC's performance. We also report on four bugs that our verification effort uncovered, including a soundness bug that allowed the prover to "lie" about the results of certain bit-vector comparisons. We note that the utility of

[1] Roughly speaking, a ZK proof system is complete if it is possible to prove every true statement, and is sound if it is infeasible to prove false ones.

our techniques is not limited to CirC: most ZKP compilers include something like the field-blaster we describe here.

In the next sections, we discuss related work (Sect. 1.1), give background on ZKPs and CirC (Sect. 2), present a field-blasting example (Sect. 3), describe our architecture (Sect. 4), give our verification conditions (Sect. 5), and present the case study (Sect. 6).

1.1 Related Work

Verified Compilers. There is a rich body of work on verifying the correctness of traditional compilers. We focus on compilation for ZKPs; this requires different correctness definitions that relate bit-like types to prime field elements. In the next paragraphs, we discuss more fine-grained differences.

Compiler verification efforts fall into two broad categories: *automated*—verification leveraging automated reasoning solvers—and *foundational*—manual verification using proof assistants (e.g., Coq [8] or Isabelle [44]). CompCert [36], for example, is a Coq-verified C compiler with verified optimization passes (e.g., [40]). Closest to our work is backend verification, which proves correct the translation from an intermediate representation to machine code. CompCert's lowering [37] is verified, as is CakeML's [31] lowering to different ISAs [19,57]. While such foundational verification offers strong guarantees, it imposes a heavy proof burden; creating CompCert, for example, took an expert team eight years [56], and any updates to compiler code require updates to proofs.

Automated verification, in contrast, does not require writing and maintaining manual proofs.[2] Cobalt [34], Rhodium [35], and PEC [32] are domain-specific languages (DSLs) for writing automatically-verified compiler optimizations and analyses. Most closely related to our work is Alive [39], a DSL for expressing verified peephole optimizations, local rewrites that transform snippets of LLVM IR [1] to better-performing ones. Alive addresses transformations over fixed types (while we address lowering to finite field equations) and formulates correctness in the presence of undefined behavior (while we formulate correctness for ZKPs). Beyond Alive, Alive2 [38] provides translation validation [41,51] for LLVM [33], and VeRA [10] verifies range analysis in the Firefox JavaScript engine.

There is also work on verified compilation for domains more closely related to ZKPs. The Porcupine [15] compiler automatically synthesizes representations for fully-homomorphic encryption [62], and Gillar [58] proves that optimization passes in the Qiskit [60] quantum compiler are semantics-preserving. While these works compile from high-level languages to circuit representations, the correctness definitions for their domains do not apply to ZKP compilers.

Verified Compilation to Cryptographic Proofs. Prior works on verified compilation for ZKPs (or similar) take the foundational approach (with attendant proof maintenance burdens), and they do not formulate a satisfactory definition of compiler correctness. PinocchioQ [18] builds on CompCert [36]. The

[2] Automated verification generally leverages solvers. This is a particularly appealing approach in our setting, since CirC (our compiler infrastructure of interest) already supports compilation to SMT formulas.

authors formulate a correctness definition that preserves the *existential soundness* of a ZKP but does not consider completeness, knowledge soundness, or zero-knowledge (see Sect. 2.2). Leo [14] is a ZKP compiler that produces (partial) ACL2 [27] proofs of correct compilation; work to emit proofs from its field-blaster is ongoing.

Recent work defines security for *reductions of knowledge* [30]. These let $\mathcal{P}$ convince $\mathcal{V}$ that it knows a witness for an instance of relation $\mathcal{R}_1$ by proving it knows a witness for an instance of an easier-to-prove relation $\mathcal{R}_2$. Unlike ZKP compilers, $\mathcal{P}$ and $\mathcal{V}$ *interact* to derive $\mathcal{R}_2$ using $\mathcal{V}$'s randomness (e.g., proving that two polynomials are nonzero w.h.p. by proving that a random linear combination of them is), whereas ZKP compilers run ahead of time and non-interactively.

Further afield, Ecne [65] is a tool that attempts to verify that the input to a ZKP encodes a *deterministic* computation. It does not consider any notion of a specification of the intended behavior. A different work [25] attempts to automatically verify that a "widget" given to a ZKP meets some specification. They consider widgets that could be constructed manually or with a compiler. Our focus is on verifying a compiler pass.

2 Background

2.1 Logic

We assume usual terminology for many-sorted first-order logic with equality ([17] gives a complete presentation). We assume every signature includes the sort Bool, constants True and False of sort $\mathtt{Bool}$, and symbol family $\approx_\sigma$ (abbreviated $\approx$) with sort $\sigma \times \sigma \rightarrow \mathsf{Bool}$ for each sort σ. We also assume a family of conditionals: symbols ite_σ ("if-then-else", abbreviated ite) of sort $\mathsf{Bool} \times \sigma \times \sigma \rightarrow \sigma$.

A *theory* is a pair $\mathcal{T} = (\Sigma, \mathbf{I})$, where Σ is a signature and $\mathbf{I}$ is a class of Σ-interpretations. A *Σ-formula* is a term of sort Bool. A Σ-formula ϕ is *satisfiable* (resp., *unsatisfiable*) in $\mathcal{T}$ if it is satisfied by some (resp., no) interpretation in $\mathbf{I}$. We focus on two theories. The first is $\mathcal{T}_{BV}$, the SMT-LIB theory of bit-vectors [52,61], with signature Σ_{BV} including a bit-vector sort $\mathsf{BV}_{[n]}$ for each $n > 0$ with bit-vector constants $c_{[n]}$ of sort $\mathsf{BV}_{[n]}$ for each $c \in [0, 2^n - 1]$, and operators including & and | (bitwise and, or) and $+_{[n]}$ (addition modulo 2^n). We write $t[i]$ to refer to the i^{th} bit of bit-vector t, where $t[0]$ is the least-significant bit. The other theory is $\mathcal{T}_{F_p}$, which is the theory corresponding to the finite field of order p, for some prime p [47]. This theory has signature Σ_{F_p} containing the sort FF_p, constant symbols $0, \ldots, p-1$, and operators $+$ and $\times$.

In this paper, we assume all interpretations interpret sorts and symbols in the same way. We write $\mathsf{dom}(v)$ for the set interpreting the sort of a variable v. We assume that Bool, True, and False are interpreted as $\{\top, \bot\}$, $\top$, and $\bot$, respectively; Σ_{BV}-interpretations follow the SMT-LIB standard; and Σ_{F_p}-interpretations interpret symbols as the corresponding elements and operations in $\mathbb{F}_p$, a finite field of order p (for concreteness, this could be the integers modulo p). Note that only the values of variables can vary between two interpretations.

For a signature Σ, let t be a Σ-term of sort σ, with free variables $x_1, \ldots, x_n$, respectively of sort $\sigma_1, \ldots, \sigma_n$. We define the function $\hat{t} : \mathsf{dom}(x_1) \times \cdots \times$

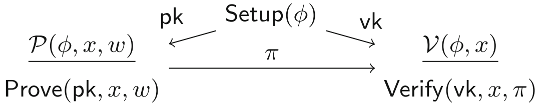

Fig. 1. The information flow for a zero-knowledge proof.

$\mathsf{dom}(x_n) \to \mathsf{dom}(t)$ as follows. Let $\mathbf{x} \in \mathsf{dom}(x_1) \times \cdots \times \mathsf{dom}(x_n)$. Let $\mathcal{M}$ be an interpretation that interprets each x_i as x_i. Then $\hat{t}(\mathbf{x}) = t^{\mathcal{M}}$ (i.e., the interpretation of t in $\mathcal{M}$). For example, the term $t = a \wedge \neg a$ defines $\hat{t} : \mathsf{Bool} \to \mathsf{Bool} = \lambda\, x. \bot$. In the following, we follow the convention used above in using the standard font (e.g., x) for logical variables and a sans serif font (e.g., x) to denote meta-variables standing for values (i.e., elements of $\sigma^{\mathcal{M}}$ for some σ and $\mathcal{M}$). Also, abusing notation, we'll conflate single variables (of both kinds) with vectors of variables when the distinction doesn't matter. Note that a formula ϕ is *satisfiable* if there exist values x such that $\hat{\phi}(\mathsf{x}) = \top$. It is *valid* if for all values x, $\hat{\phi}(\mathsf{x}) = \top$.

For terms s, t and variable x, $t[x \mapsto s]$ denotes t with all occurrences of x replaced with s. For a sequence of variable-term pairs, $S = (x_1 \mapsto s_1, \ldots, x_n \mapsto s_n)$, $t[S]$ is defined to be $t[x_1 \mapsto s_1] \cdots [x_n \mapsto s_n]$.

2.2 Zero Knowledge Proofs

As mentioned above, Zero-knowledge proofs (ZKPs) make it possible to prove that some secret data satisfies a public property—without revealing the data itself. See [59] for a full presentation; we give a brief overview here, and then describe how general-purpose ZKPs are used.

Overview and Definitions. In a cryptographic proof system, there are two parties: a *verifier* $\mathcal{V}$ and a *prover* $\mathcal{P}$. $\mathcal{V}$ knows a public *instance* x and asks $\mathcal{P}$ to show that it has knowledge of a secret *witness* w satisfying a public *predicate* $\phi(x, w)$ from a predicate class Φ (a set of formulas) (i.e., $\hat{\phi}(\mathsf{x}, \mathsf{w}) = \top$). Figure 1 illustrates the workflow. First, a trusted party runs an efficient (i.e., polytime in an implicit security parameter λ) algorithm $\mathsf{Setup}(\phi)$ which produces a *proving key* pk and a *verifying key* vk. Then, $\mathcal{P}$ runs an efficient algorithm $\mathsf{Prove}(\mathsf{pk}, \mathsf{x}, \mathsf{w}) \to \pi$ and sends the resulting *proof* π to $\mathcal{V}$. Finally, $\mathcal{V}$ runs an efficient verification algorithm $\mathsf{Verify}(\mathsf{vk}, \mathsf{x}, \pi) \to \{\top, \bot\}$ that accepts or rejects the proof. A zero-knowledge argument of knowledge for class Φ is a tuple $\Pi = (\mathsf{Setup}, \mathsf{Prove}, \mathsf{Verify})$ with three informal properties for every $\phi \in \Phi$ and every $\mathsf{x} \in \mathsf{dom}(x), \mathsf{w} \in \mathsf{dom}(w)$:

- *perfect completeness*: if $\hat{\phi}(\mathsf{x}, \mathsf{w})$ holds, then $\mathsf{Verify}(\mathsf{vk}, \mathsf{x}, \pi)$ holds;
- *computational knowledge soundness* [9]: an efficient adversary that does not know w cannot produce a π such that $\mathsf{Verify}(\mathsf{vk}, \mathsf{x}, \pi)$ holds; and
- *zero-knowledge* [22]: π reveals nothing about w, other than its existence.

Technically, the system is an "argument" rather than a "proof" because soundness only holds against efficient adversaries. Also note that knowledge soundness requires that an entity must "know" a valid w' to produce a proof; it is not enough for a valid w' to simply exist. We give more precise definitions in Appendix A.

Representations for ZKPs. As mentioned above, ZKP applications are manifold (Sect. 1)—from cryptocurrencies to private registries. This breadth of applications is possible because ZKPs support a broad class of predicates. Most commonly, these predicates are expressed as *rank-1 constraint systems* (R1CSs). Recall that $\mathbb{F}_p$ is a prime-order finite field (also called a *prime field*). We will drop the subscript p when it is not important. In an R1CS, x and w are vectors of elements in $\mathbb{F}$; let $\mathsf{z} \in \mathbb{F}^m$ be their concatenation. The function $\hat{\phi}$ can be defined by three matrices $\mathsf{A}, \mathsf{B}, \mathsf{C} \in \mathbb{F}^{n \times m}$; $\hat{\phi}(\mathsf{x}, \mathsf{w})$ holds when $\mathsf{Az} \circ \mathsf{Bz} = \mathsf{Cz}$, where $\circ$ is the element-wise product. Thus, ϕ can be viewed as n conjoined *constraints*, where each constraint i is of the form $(\sum_j a_{ij} z_j) \times (\sum_j b_{ij} z_j) \approx (\sum_j c_{ij} z_j)$ (where the a_{ij}, b_{ij} and c_{ij} are constant symbols from Σ_{F_p}, and the z_j are a vector of variables of sort FF_p). That is, each constraint enforces a single non-linear multiplication.

2.3 Compilation Targeting Zero Knowledge Proofs

To write a ZKP about a high-level predicate ϕ, that predicate is first compiled to an R1CS. A *ZKP compiler* from class Φ (a set of Σ-formulas) to class Φ' (a set of Σ'-formulas) is an efficient algorithm $\mathsf{Compile}(\phi \in \Phi) \to (\phi' \in \Phi', \mathsf{Ext}_x, \mathsf{Ext}_w)$. Given a predicate $\phi(x, w)$, it returns a predicate $\phi'(x', w')$ as well as two efficient and deterministic algorithms, instance and witness *extenders*: $\mathsf{Ext}_x : \mathsf{dom}(x) \to \mathsf{dom}(x')$ and $\mathsf{Ext}_w : \mathsf{dom}(x) \times \mathsf{dom}(w) \to \mathsf{dom}(w')$.[3] For example, CirC [46] can compile a Boolean-returning C function (in a subset of C) to an R1CS.

At a high-level, ϕ and ϕ' should be "equisatisfiable", with Ext_x and Ext_w mapping satisfying values for ϕ to satisfying values for ϕ'. That is, for all $\mathsf{x} \in \mathsf{dom}(x)$ and $\mathsf{w} \in \mathsf{dom}(w)$ such that $\hat{\phi}(\mathsf{x}, \mathsf{w}) = \top$, if $\mathsf{x}' = \mathsf{Ext}_x(\mathsf{x})$ and $\mathsf{w}' = \mathsf{Ext}_w(\mathsf{x}, \mathsf{w})$, then $\hat{\phi}'(\mathsf{x}', \mathsf{w}') = \top$. Furthermore, for any x, it should be impossible to (efficiently) find w' satisfying $\hat{\phi}'(\mathsf{Ext}_x(\mathsf{x}), \mathsf{w}') = \top$ without knowing a w satisfying $\hat{\phi}(\mathsf{x}, \mathsf{w}) = \top$. In Sect. 5.1, we precisely define correctness for a predicate compiler.

One can build a ZKP for class Φ from a compiler from Φ to Φ' and a ZKP for Φ'. Essentially, one runs the compiler to get a predicate $\phi' \in \Phi'$, as well as Ext_x and Ext_w. Then, one writes a ZKP to show that $\hat{\phi}'(\mathsf{Ext}_x(\mathsf{x}), \mathsf{Ext}_w(\mathsf{x}, \mathsf{w})) = \top$. In Appendix A, we give this construction in full and prove it is secure.

Optimization. The primary challenge when using ZKPs is cost: typically, Prove is at least three orders of magnitude slower than checking ϕ directly [64]. Since Prove's cost scales with n (the constraint count), it is *critical* for the compiler to minimize n. The space of optimizations is large and complex, for two reasons. First, the compiler can introduce fresh variables. Second, only equisatifiability—not logical equivalence—is needed. Compilers in this space exploit equisatisfiability heavily to efficiently represent high-level constructs (e.g., Booleans, bit-vectors, arrays, ...) as an R1CS.

[3] For technical reasons, the runtime of Ext_x and the size of its description must be $\mathsf{poly}(\lambda, |x|)$—not just $\mathsf{poly}(\lambda)$ (Appendix A). .

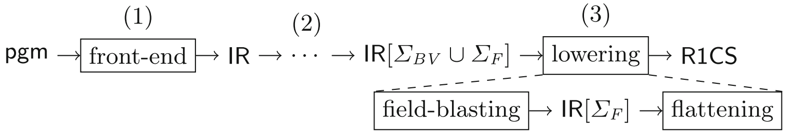

Fig. 2. The architecture of CirC

As a (simple!) example, consider the Boolean computation $a \approx c_1 \vee \cdots \vee c_k$. Assume that $c'_1, \ldots, c'_k$ are variables of sort FF and that we add constraints $c'_i(1 - c'_i) \approx 0$ to ensure that c'_i has to be 0 or 1 for each i. Assume further that $(c'_i \approx 1)$ encodes c_i for each i. How can one additionally ensure that a' (also of sort FF) is also forced to be equal to 0 or 1 and that $(a' \approx 1)$ is a correct encoding of a? Given that there are $k - 1$ ORs, natural approaches use $\Theta(k)$ constraints. One clever approach is to introduce variable x' and enforce constraints $x'(\sum_i c'_i) \approx a'$ and $(1 - a')(\sum_i c'_i) \approx 0$. In any interpretation where any c_i is true, the corresponding interpretation for a' must be 1 to satisfy the second constraint; setting x' to the sum's inverse satisfies the first. If all c_i are false, the first constraint ensures a' is 0. This technique assumes the sum does not overflow; since ZKP fields are typically large (e.g., with p on the order of 2^{255}), this is usually a safe assumption.

CirC. CirC [46] is an infrastructure for building compilers from high-level languages (e.g., a C subset), to R1CSs. It has been used in research projects [4,12], and in industrial R&D. Figure 2 shows the structure of an R1CS compiler built with CirC. First, the front-end of the compiler converts the source program into CirC-IR. CirC-IR is a term IR based on SMT-LIB that includes: Booleans, bit-vectors, fixed-size arrays, tuples, and prime fields.[4] Second, the compiler optimizes and simplifies the IR so that the only remaining sorts are Booleans, bit-vectors, and the target prime field. Third, the compiler lowers the simplified IR to an R1CS predicate over the target field. For ZKPs built with CirC, *the completeness, soundness, and zero-knowledge of the end-to-end system depend on the correctness of CirC itself.*

3 Overview and Example

To start, we view CirC's lowering pass as two passes (Fig. 2). The first pass, "(finite-)field-blasting," converts a many-sorted IR (representable as a $(\Sigma_{BV} \cup \Sigma_F)$-formula) to a conjunction of field equations (Σ_F-equations). The second pass, "flattening," converts this conjunction of field equations to an R1CS.

Our focus is on verifying the first pass. We begin with a worked example of how to field-blast a small snippet of CirC-IR (Sect. 3.1). This example will illustrate four key ideas (Sect. 3.2) that inspire our field-blaster's architecture.

[4] We list all CirC-IR operators for Booleans, bit-vectors, and prime fields in Appendix C. Almost all are from SMT-LIB.

Table 1. New variables and assertions when compiling the example ϕ.

clause	term from ϕ	assertions	new variables	notes
1	x_0		x'_0	
	w_0	$w'_0(w'_0 - 1) \approx 0$	w'_0	
	$x_0 \oplus w_0$	$1 \approx 1 - w'_0 - x'_0 + 2w'_0x'_0$		
2	x_1		$x'_{1,u}$	
	w_1	$w'_{1,i}(w'_{1,i} - 1) \approx 0$	$w'_{1,i}$	$i \in [0, 3]$
	$x_1 +_{[4]} w_1$	$s' \approx x'_{1,u} + \sum_{i=0}^{3} 2^i w'_{1,i}$	s'	
		$s'_i(s'_i - 1) \approx 0$	s'_i	$i \in [0, 4]$
		$s' \approx \sum_{i=0}^{4} 2^i s'_i$		
	$x_1 +_{[4]} w_1 \approx w_1$	$s'_i \approx w'_{1,i}$		$i \in [0, 3]$
3	x_2		$x'_{2,u}$	
	x_2 (bits)	$x'_{2,i}(x'_{2,i} - 1) \approx 0$	$x'_{2,i}$	$i \in [0, 3]$
		$x'_{2,u} \approx \sum_{i=0}^{3} 2^i x'_{2,i}$		
	x_2 & $w_1 \approx x_2$	$x'_{2,i}w'_{1,i} \approx x'_{2,i}$		$i \in [0, 3]$
4	x_3, w_2		x'_3, w'_2	
	$x_3 \approx w_2 \times w_2$	$x'_3 \approx w'_2 \times w'_2$		

3.1 An Example of Field-Blasting

We start with an example CirC-IR predicate expressed as a $(\Sigma_{BV} \cup \Sigma_F)$-formula:

$$\phi \triangleq (x_0 \oplus w_0) \wedge (w_1 +_{[4]} x_1 \approx w_1) \wedge (x_2 \mathbin{\&} w_1 \approx x_2) \wedge (x_3 \approx w_2 \times w_2) \qquad (1)$$

The predicate includes: the XOR of two Booleans ("$\oplus$"), a bit-vector sum, a bit-vector AND, and a field product. x_0 and w_0 are of sort `Bool`, x_1, x_2, and w_1 are of sort $\mathsf{BV}_{[4]}$, and x_3 and w_2 are of sort FF_p. We'll assume that $p \gg 2^4$. Table 1 summarizes the new variables and assertions we create during field-blasting; we describe the origin of each assertion and new variable in the next paragraphs.

Lowering Clause One (Booleans). We begin with the Boolean term $(x_0 \oplus w_0)$. We will use 1 and 0 to represent $\top$ and $\bot$. We introduce variables x'_0 and w'_0 of sort FF_p to represent x_0 and w_0 respectively. To ensure that w'_0 is 0 or 1, we assert: $w'_0(w'_0 - 1) \approx 0$. [5] $x_0 \oplus w_0$ is then represented by the expression $1 - x'_0 - w'_0 + 2x'_0w'_0$. Setting this equal to 1 enforces that $x_0 \oplus w_0$ must be true. These new assertions and fresh variables are reflected in the first three rows of the table.

Lowering Clause Two and Three (Bit-vectors). Before describing how to bit-blast the second and third clauses in ϕ, we discuss bit-vector representations in

[5] Later (Sect. 5), we will see that "well-formedness" constraints like this are unnecessary for instance variables, such as x_0. .

general. A bit-vector t can be viewed as a sequence of b bits or as a non-negative integer less than 2^b. These two views suggest two natural representations in a prime-order field: first, as one field element t'_u, whose unsigned value agrees with t (assuming the field's size is at least 2^b); second, as b elements $t'_0, \ldots, t'_{b-1}$, that encode the bits of t as 0 or 1 (in our encoding, t'_0 is the least-significant bit and t'_{b-1} is the most-significant bit). The first representation is simple, but with it, some field values (e.g., 2^b) don't corresponding to any possible bit-vector. With the second approach, by including equations $t'_i(t'_i - 1) \approx 0$ in our system, we ensure that any satisfying assignment corresponds to a valid bit-vector. However, the extra b equations increase the size of our compiler's output.

We represent ϕ's w_1 bit-wise: as $w'_{1,0}, \ldots, w'_{1,3}$, and we represent the instance variable x_1 as $x'_{1,u}$.[6] For the constraint $w_1 +_{[4]} x_1 \approx w_1$, we compute the sum in the field and bit-decompose the result to handle overflow. First, we introduce new variable s' and set it equal to $x'_{1,u} + \sum_{i=0}^{3} 2^i w'_{1,i}$. Then, we bit-decompose s', requiring $s' \approx \sum_{i=0}^{4} 2^i s'_i$, and $s'_i(s'_i - 1) \approx 0$ for $i \in [0, 4]$. Finally, we assert $s'_i \approx w'_{1,i}$ for $i \in [0, 3]$. This forces the lowest 4 bits of the sum to be equal to w_1.

The constraint x_2 & $w_1 \approx x_2$ is more challenging. Since x_2 is an instance variable, we initially encode it as $x'_{2,u}$. Then, we consider the bit-wise AND. There is no obvious way to encode a bit-wise operation, other than bit-by-bit. So, we convert $x'_{2,u}$ to a bit-wise representation: We introduce witness variables $x'_{2,0}, \ldots, x'_{2,3}$ and equations $x'_{2,i}(x'_{2,i} - 1) \approx 0$ as well as equation $x'_{2,u} \approx \sum_{i=0}^{3} 2^i x'_{2,i}$. Then, for each i we require $x'_{2,i} w'_{1,i} \approx x'_{2,i}$.

Lowering the Final Clause (Field Elements). Finally, we consider the field equation $x_2 \approx w_2 \times w_2$. Our target is also field equations, so lowering this is straightforward. We simply introduce primed variables and copy the equation.

3.2 Key Ideas

This example highlights four ideas that guide the design of our field-blaster:

1. *fresh variables and assertions*: Field-blasting uses two primitive operations: creating new variables in ϕ' (e.g., w'_0 to represent w_0) and adding new assertions to ϕ' (e.g., $w'_0(w'_0 - 1) \approx 0$).
2. *encodings*: For a term t in ϕ, we construct a field term (or collection of field terms) in ϕ' that represent the value of t. For example, the Boolean w_0 is represented as the field element w'_0 that is 0 or 1.
3. *operator rules*: if t is an operator applied to some arguments, we can encode t given encodings of the arguments. For example, if t is $x_0 \oplus w_0$, and x_0 is encoded as x'_0 and w_0 as w'_0, then t can be encoded as $1 - x'_0 - w'_0 + 2x'_0 w'_0$.
4. *conversions*: Some sorts can be represented by encodings of different kinds. If a term has multiple possible encodings, the compiler may need to convert between them to apply some operator rule. For example, we converted x_2 from an unsigned encoding to a bit-wise encoding before handling an AND.

[6] We represent w_1 bit-wise so that we can ensure the representation is well-formed with constraints $w'_{1,i}(w'_{1,i} - 1) \approx 0$. As previously noted, such well-formedness constraints are not needed for an instance variable like x_1.(See footnote 5).

Table 2. Encodings for each term sort. Only bit-vectors have two encoding kinds.

Variant Contents			**Semantics**
encoded_term	kind	terms	Validity Condition
t: Bool	`bit`	f	$f \approx ite(t, 1, 0)$
t: $\mathsf{BV}_{[b]}$	`uint`	f	$f \approx \sum_i ite(t[i] \approx 1_{[1]}, 2^i, 0)$
t: $\mathsf{BV}_{[b]}$	`bits`	$f_0, \ldots, f_{b-1}$	$\bigwedge_i f_i \approx ite(t[i] \approx 1_{[1]}, 1, 0)$
t: FF	`field`	f	$t \approx f$

4 Architecture

In this section, we present our field-blaster architecture. To compile a predicate ϕ to a system of field equations ϕ', our architecture processes each term t in ϕ using a post-order traversal. Informally, it represents each t as an "encoding" in ϕ': a term (or collection of terms) over variables in ϕ'. Each encoding is produced by a small algorithm called an "encoding rule".

Below, we define the type of encodings Enc (Sect. 4.1), the five different types of encoding rules (Sect. 4.2), and a calculus that iteratively applies these rules to compile all of ϕ (Sect. 4.3).

4.1 Encodings

Table 2 presents our tagged union type Enc of possible term encodings. Each variant comprises the term being encoded, its tag (the *encoding kind*), and a sequence of field terms. The encoding kinds are `bit` (a Boolean as 0/1), `uint` (a bit-vector as an unsigned integer), `bits` (a bit-vector as a sequence of bits), and `field` (a field term trivially represented as a field term). Each encoding has an intended semantics: a condition under which the encoding is considered valid. For instance, a `bit` encoding of Boolean t is valid if the field term f is equal to $ite(t, 1, 0)$.

4.2 Encoding Rules

An encoding rule is an algorithm that takes and/or returns encodings, in order to represent some part of the input predicate as field terms and equations.

Primitive Operations. A rule can perform two primitive operations: creating new variables and emitting assertions. In our pseudocode, the primitive function $\mathsf{fresh}(\mathsf{name}, t, \mathsf{isInst}) \rightarrow x'$ creates a fresh variable. Argument isInst is a Boolean indicating whether x' is an instance variable (as opposed to a witness). Argument t is a field term (over variables from ϕ and previously defined primed variables) that expresses how to compute a value for x'. For example, to create a field variable w' that represents Boolean witness variable w, a rule can call $\mathsf{fresh}(w', ite(w, 1, 0), \bot)$. The compiler uses t to help create the Ext_x and Ext_w algorithms. A rule asserts a formula t' (over primed variables) by calling $\mathsf{assert}(t')$.

```
$\text{fn variable}(\mathrm{t}, \mathsf{isInst}) \to \mathsf{Enc}:$
  $\text{if } \mathsf{isInst}:$
    $t' \leftarrow \mathsf{fresh}(\mathsf{name}(\mathrm{t}) \,\|\, `u\text{'},$
        $\sum_i ite(t[i] \approx 1_{[1]}, 2^i, 0), \top)$
    $\text{return } t, \texttt{uint}, t'$
  $\text{else}:$
    $\text{for } i \text{ in } [0, \mathsf{size}(\mathsf{sort}(t)) - 1]:$
      $t'_i \leftarrow \mathsf{fresh}(\mathsf{name}(t) \,\|\, i,$
        $ite(t[i] \approx 1_{[1]}, 1, 0), \bot)$
      $\mathsf{assert}(t'_i(t'_i - 1) = 0)$
    $\text{return } t, \texttt{bits}, t'_0, \ldots, t'_{\mathsf{size}(\mathsf{sort}(t))-1}$

$\text{fn const}(t) \to \mathsf{Enc}:$
  $\text{for } i \text{ in } [0, \mathsf{size}(\mathsf{sort}(t)) - 1]:$
    $t'_i \leftarrow ite(t[i] \approx 1_{[1]}, 1, 0)$
  $\text{return } t, \texttt{bits}, t'_0, \ldots, t'_{\mathsf{size}(\mathsf{sort}(t))-1}$
```

```
$\text{fn assertEq}(e : \mathsf{Enc}, e' : \mathsf{Enc}):$
  $\text{if } \mathsf{kind}(e) = \texttt{bits}:$
    $\text{for } i \text{ in } [0, \mathsf{size}(\mathsf{terms}(e)) - 1]:$
      $\mathsf{assert}(\mathsf{terms}(e)[i] \approx \mathsf{terms}(e')[i])$
  $\text{elif } \mathsf{kind}(e) = \texttt{uint}:$
    $\mathsf{assert}(\mathsf{terms}(e)[0] \approx \mathsf{terms}(e')[0])$

$\text{fn convert}(e : \mathsf{Enc}, \mathsf{kind}' : \mathsf{Kind}) \to \mathsf{Enc}:$
  $t \leftarrow \mathsf{encoded\_term}(e)$
  $\text{if } \mathsf{kind}(e) = \texttt{bits} \text{ and } \mathsf{kind}' = \texttt{uint}:$
    $\text{return } t, \texttt{uint}, \sum_i 2^i \mathsf{terms}(e)[i]$
  $\text{elif } \mathsf{kind}(e) = \texttt{uint} \text{ and } \mathsf{kind}' = \texttt{bits}:$
    $e' \leftarrow \mathsf{variable}(t, \bot)$
    $\mathsf{assert}(\mathsf{terms}(e)[0] \approx \sum_i 2^i \mathsf{terms}(e')[i])$
    $\text{return } e'$
```

Fig. 3. Pseudocode for some bit-vector rules: variable uses a `uint` encoding for instances and bit-splits witnesses to ensure they're well-formed, const bit-splits the constant it's given, assertEq asserts unsigned or bit-wise equality, and convert either does a bit-sum or bit-split.

Rule Types. There are five types of rules: (1) Variable rules variable(t, isInst) → e take a variable t and its instance/witness status and return an encoding of that variable made up of fresh variables. (2) Constant rules const(t) → e take a constant term t and produce an encoding of t comprising terms that depend only on t. Since t is a constant, the terms in e can be evaluated to field constants (see the calculus in Sect. 4.3).[7] The const rule cannot call fresh or assert. (3) Equality rules assertEq(e, e') take two encodings of the same kind and emit assertions that equate the underlying terms. (4) Conversion rules convert(e, kind$'$) → e' take an encoding and convert it to an encoding of a different kind. Conversions are only non-trivial for bit-vectors, which have two encoding kinds: `uint` and `bits`. (5) Operator rules apply to terms t of form $o(t_1, \ldots, t_n)$. Each operator rule takes t, o, and encodings of the child terms t_i and returns an encoding of t. Some operator rules require specific kinds of encodings; before using such an operator rule, our calculus (Sect. 4.3) calls the convert rule to ensure the input encodings are the correct kind. Figure 3 gives pseudocode for the first four rule types, as applied to bit-vectors. Figure 4 gives pseudocode for two bit-vector operator encoding rules. A field blaster uses many operator rules: in our case study (Sect. 6) there are 46.

[7] Having const(t) return terms that depend on t (rather than directly returning constants) is useful for constructing verification conditions for const.

```
fn bvZeroExt($t, o$ : Op, $e$ : Enc) :
  if kind($e$) = bits:
    $w \leftarrow$ size(terms($e$))
    for $i$ in $[0, w-1]$:
      $t'_i \leftarrow$ terms($e$)[$i$]
    for $i$ in $[0, o.\text{newBits} - 1]$:
      $t'_{w+i} \leftarrow 0$
    return $t$, bits, $t'_0, \dots, t'_{w+o.\text{newBits}-1}$
  else:
    return $t$, kind($e$), terms($e$)
```

```
fn bvMulUint($t, o$ : Op, $\vec{e}$ : [Enc]) :
  $w \leftarrow$ size(sort(encoded_term($e[0]$)))
  $W \leftarrow$ size($\vec{e}$) $\times w$
  assume $W < \lfloor \log_2 p \rfloor$
  $s' \leftarrow \prod_i$ terms($e_i$)[0]
  $b \leftarrow$ ff2bv($W, s'$)
  for $i$ in $[0, W-1]$:
    $t'_i \leftarrow$ fresh($i$, $ite(b[i], 1, 0)$, $\bot$)
    assert($t'_i(t'_i - 1) \approx 0$)
  assert($s' \approx \sum_{i=0}^{W-1} 2^i t'_i$)
  return $t$, bits, $t'_0, \dots, t'_{w-1}$
```

Fig. 4. Pseudocode for some bit-vector operator rules. bvZeroExt zero-extends a bit-vector; for bit-wise encodings, it adds zero bits, and for unsigned encodings, it simply copies the original encoding. bvMulUint multiplies bit-vectors, all assumed to be unsigned encodings. We show only the case where the multiplication cannot overflow in the field: in this case the rule performs the multiplication in the field, and bit-splits the result to implement reduction modulo 2^b. The rules use ff2bv, which converts from a field element to a bit-vector (discussed in Sect. 6.1).

4.3 Calculus

We now give a non-deterministic calculus describing how our field-blaster applies rules to compile a predicate $\phi(x, w)$ into a system of field equations.

A calculus state is a tuple of three items: (E, A, F). The *encoding store* E is a (multi-)map from terms to sets of encodings. The *assertions formula* A is a conjunction of all field equations asserted via assert. The *fresh variable definitions sequence* F is a sequence consisting of pairs, where each pair (v, t) matches a single call to fresh$(v, t, \dots)$.

Figure 5 shows the transitions of our calculus. We denote the result of a rule as $A', F', e' \leftarrow r(\dots)$, where A' is a formula capturing any new assertions, F' is a sequence of pairs capturing any new variable definitions, and e' is the rule's return value. We may omit one or more results if they are always absent for a particular rule. For encoding store E, $E \cup (t \mapsto e)$ denotes the store with e added to t's encoding set.

There are five kinds of transitions. The Const transition adds an encoding for a constant term. The const rule returns an encoding e whose terms depend on the constant c; e' is a new encoding identical to e, except that each of its terms has been evaluated to obtain a field constant. The Var transition adds an encoding for a variable term. The Conv transition takes a term that is already encoded and re-encodes it with a new encoding kind. The kinds operator returns all legal values of kind for encodings of a given sort. The Op_r transition applies operator rule r. This transition is only possible if r's operator kind agrees with o, and if its input encoding kinds agree with $\vec{e}$. The Finish transition applies when ϕ has been encoded. It uses const and assertEq to build assertions that hold when $\phi = \top$. Rather than producing a new calculus state, it returns the outputs of the calculus: the assertions and the variable definitions.

$$\frac{\text{constant term } c \qquad e \leftarrow \mathsf{const}(c) \qquad e' \leftarrow \mathsf{map}(\mathsf{eval}, e)}{E := E \cup (c \mapsto e')}\ \mathsf{Const}$$

$$\frac{\text{variable term } v \qquad A', F', e \leftarrow \mathsf{variable}(v, \mathsf{isInst}(v))}{E := E \cup (v \mapsto e), \quad A := A \wedge A', \quad F := F \parallel F'}\ \mathsf{Var}$$

$$\frac{(t \mapsto e) \in E \qquad \mathsf{kind} \in \mathsf{kinds}(\mathsf{sort}(t)) \qquad A', F', e' \leftarrow \mathsf{convert}(e, \mathsf{kind})}{E := E \cup (t \mapsto e'), \quad A := A \wedge A', \quad F := F \parallel F'}\ \mathsf{Conv}$$

$$\frac{(t_i \mapsto e_i) \in E \qquad t = o(\vec{t}) \qquad A', F', e' \leftarrow r(t, o, \vec{e})}{E := E \cup (t \mapsto e'), \quad A := A \wedge A', \quad F := F \parallel F'}\ \mathsf{Op}_r$$

$$\frac{(\phi \mapsto e) \in E \qquad e_\top \leftarrow \mathsf{const}(\top) \qquad A', F' \leftarrow \mathsf{assertEq}(e, e_\top)}{\text{return } (A \wedge A', \quad F \parallel F')}\ \mathsf{Finish}$$

Fig. 5. The transition rules of our rewriting calculus.

To meet the requirements of the ZKP compiler, our calculus must return two extension function: Ext_x and Ext_w (Sect. 2.2). Both can be constructed from the fresh variable definitions F. One subtlety is that $\mathsf{Ext}_x(x)$ (which assigns values to fresh instance variables) is a function of x only—it cannot depend on the witness variables of ϕ. We ensure this by allowing fresh instance variables to only be created by the $\mathsf{variable}$ rule, and only when it is called with $\mathsf{isInst} = \top$.

Strategy. Our calculus is non-deterministic: multiple transitions are possible in some situations; for example, some conversion is almost always applicable. The strategy that decides which transition to apply affects field blaster performance (Appendix D) but *not* correctness.

5 Verification Conditions

In this section, we first define correctness for a ZKP compiler (Sect. 5.1). Then, we give verification conditions (VCs) for each type of encoding rule (Sect. 5.2). Finally, we show that if these VCs hold, our calculus is a correct ZKP compiler (Sect. 5.3).

5.1 Correctness Definition

Definition 1 (Correctness). *A ZKP compiler* $\mathsf{Compile}(\phi) \rightarrow (\phi', \mathsf{Ext}_x, \mathsf{Ext}_w)$ *is* ***correct*** *if it is demonstrably complete and demonstrably sound.*

- *demonstrable completeness*: For all $\mathsf{x} \in \mathsf{dom}(x), \mathsf{w} \in \mathsf{dom}(w)$ such that $\hat{\phi}(\mathsf{x}, \mathsf{w}) = \top$,

$$\hat{\phi}'(\mathsf{Ext}_x(\mathsf{x}), \mathsf{Ext}_w(\mathsf{x}, \mathsf{w})) = \top$$

- *demonstrable soundness*: There exists an efficient algorithm $\mathsf{Inv}(\mathsf{x}', \mathsf{w}') \rightarrow \mathsf{w}$ such that for all $\mathsf{x} \in \mathsf{dom}(x), \mathsf{w}' \in \mathsf{dom}(w')$ such that $\hat{\phi}'(\mathsf{Ext}_x(\mathsf{x}), \mathsf{w}') = \top$,

$$\hat{\phi}(\mathsf{x}, \mathsf{Inv}(\mathsf{Ext}_x(\mathsf{x}), \mathsf{w}')) = \top$$

Demonstrable completeness (respectively, soundness) requires the existence of a witness for ϕ' (resp., ϕ) when a witness exists for ϕ (resp., ϕ'); this existence is *demonstrated* by an efficient algorithm Ext_w (resp., Inv) that computes the witness.

Correct ZKP compilers are important for two reasons. First, since sequential composition preserves correctness, one can prove a multi-pass compiler is correct pass-by-pass. Second, a correct ZKP compiler from Φ to Φ' can be used to generalize a ZKP for Φ' to one for Φ. We prove both properties in Appendix A.

Theorem 1 (Compiler Composition). *If* $\mathsf{Compile}'$ *and* $\mathsf{Compile}''$ *are correct, then the compiler* $\mathsf{Compose}(\mathsf{Compile}', \mathsf{Compile}'')$ *(Appendix A) is correct.*

Theorem 2 (ZKP Generalization). *(informal) Given a correct ZKP compiler* $\mathsf{Compile}$ *from* Φ *to* Φ' *and a ZKP for* Φ'*, we can construct a ZKP for* Φ*.*

5.2 Rule VCs

Recall (Sect. 4) that our language manipulates encodings through five types of encoding rules. We give verification conditions for each type of rule. Intuitively, these capture the correctness of each rule in isolation. Next, we'll show that they imply the correctness of a ZKP compiler that follows our calculus.

Our VCs quantify over valid encodings. That is, they have the form: "for any valid encoding e of term t, ..." We can quantify over an encoding e by making each $t_i \in \mathsf{terms}(e)$ a fresh variable, and quantifying over the t_i. Encoding validity is captured by a predicate $valid(e, t)$, which is defined to be the validity condition in Table 2. Each VC containing encoding variables $\boldsymbol{e}$ implicitly represents a conjunction of instances of that VC, one for each possible tuple of kinds of $\boldsymbol{e}$, which is fixed for each instance. If a VC contains $valid(e, t)$, the sort of t is constrained to be *compatible* with $\mathsf{kind}(e)$. For a kind and a sort to be compatible, they must occur in the same row of Table 2. We define the equality predicate $equal(e, e')$ as $\bigwedge_i \mathsf{terms}(e)[i] \approx \mathsf{terms}(e')[i]$.

Encoding Uniqueness. First, we require the uniqueness of valid encodings, for any fixed encoding kind. Table 3 shows the VCs that ensure this. Each row is a formula that must be valid, for all compatible encodings and terms. The first two rows ensure that there is a bijection from terms to their valid encodings (in the first row, we consider only instances for which $\mathsf{kind}(e) = \mathsf{kind}(e')$). The function $fromTerm(t, \mathsf{kind}) \rightarrow e$ maps a term and an encoding kind to a valid encoding of that kind, and the function $toTerm(e) \rightarrow t$ maps a valid encoding to its encoded term. The third and fourth rows ensure that *fromTerm* and *toTerm* are correctly defined. We will use *toTerm* in our proof of calculus soundness (Appendix B) and we will use *fromTerm* to optimize VCs for faster verification (Sect. 6.1).

Table 3. VCs related to encoding uniqueness.

Property	Condition
valid encoding uniqueness	$(\mathit{valid}(e,t) \land \mathit{valid}(e',t)) \to \mathit{equal}(e,e')$
valid encoding uniqueness	$(\mathit{valid}(e,t) \land \mathit{valid}(e,t')) \to t \approx t'$
fromTerm correctness	$\mathit{valid}(\mathit{fromTerm}(t, \mathsf{kind}), t)$
toTerm correctness	$\mathit{valid}(e, \mathit{toTerm}(e))$

Table 4. VCs for encoding rules.

Rule	Property	Condition
Operator	Sound	$(A \land \bigwedge_i \mathit{valid}(e_i,t_i)) \to \mathit{valid}(e', o(\boldsymbol{t}))$
$e' \leftarrow r_o(\boldsymbol{e})$	Complete	$\big((\bigwedge_i \mathit{valid}(e_i,t_i)) \to (A \land \mathit{valid}(e', o(\boldsymbol{t})))\big)\ [F]$
Equality	Sound	$(A \land \bigwedge_i \mathit{valid}(e_i,t_i)) \to (t_1 \approx t_2)$
$r_=(e_1,e_2)$	Complete	$\big(((t_1 \approx t_2) \land \bigwedge_i \mathit{valid}(e_i,t_i)) \to A\big)\ [F]$
Conversion	Sound	$(A \land \mathit{valid}(e,t)) \to \mathit{valid}(e',t)$
$e' \leftarrow r_\to(e)$	Complete	$((\mathit{valid}(e,t)) \to (A \land \mathit{valid}(e',t)))\ [F]$
Variable	Sound ($t \in w$)	$A \to \exists t'.\ \mathit{valid}(e',t')$
	Sound ($t \in x$)	$(A \to \mathit{valid}(e',t))[F_x]$
$e' \leftarrow r_v(t)$	Complete	$(A \land \mathit{valid}(e',t))[F]$
Constant	—	$\mathit{valid}(e,t)$
$e \leftarrow r_c(t)$		

For an example of the *valid*, *fromTerm*, and *toTerm* functions, consider a Boolean b encoded as an encoding e with kind bit and whose terms consist of a single field element f. Validity is defined as $\mathit{valid}(e,b) = f \approx \mathit{ite}(b,1,0)$, $\mathit{toTerm}(f)$ is defined as $f \approx 1$, and $\mathit{fromTerm}(b, \texttt{bit})$ is $(b, \texttt{bit}, \mathit{ite}(b,1,0))$.

VCs for Encoding Rules. Table 4 shows our VCs for the rules of Fig. 5. For each rule application, A and F denote, respectively, the assertions and the variable declarations generated when that rule is applied. We explain some of the VCs in detail.

First, consider a rule r_o for operator o applied to inputs $t_1, \ldots, t_k$. The rule takes input encodings $e_1, \ldots, e_k$ and returns an output e'. It is sound if the validity of its inputs and its assertions imply the validity of its output. It is complete if the validity of its inputs implies its assertions and the validity of its output, after substituting fresh variable definitions.

Second, consider a variable rule. Its input is a variable term t, and it returns e', a putative encoding thereof. Note that e' does not actually contain t, though the substitutions in F may bind the fresh variables of e' to functions of t. For the rule to be sound when t is a witness variable ($t \in w$), the assertions must imply that e' is valid for *some* term t'. For the rule to be sound when t is an instance variable ($t \in x$), the assertions must imply that e' is valid for t, when the instance variables in e' are replaced with their definition (F_x denotes F, restricted to its declarations of instance variables).[8] For the variable rule to be complete (for an instance or a witness), the assertions and the validity of e' for t must follow from F.

Third, consider a constant rule. Its input is a constant term t, and it returns an encoding e. Recall that the terms of e are always evaluated, yielding e' which only contains constant terms. Thus, correctness depends only on the fact that e is always a valid encoding of the input t. This can be captured with a single VC.

5.3 A Correct Field-Blasting Calculus

Given rules that satisfy these verification conditions, we show that the calculus of Sect. 4.3 is a correct ZKP compiler. The proof is in Appendix B.

Theorem 3 (Correctness). *With rules that satisfy the conditions of Sect. 5.2, the calculus of Sect. 4.3 is demonstrably complete and sound (Def. 1).*

6 Case Study: A Verifiable Field-Blaster for CirC

We implemented and partially verified a field-blaster for CirC [46]. Our implementation is based on a refactoring of CirC's original field blaster to conform to our encoding rules (Sect. 4.2) and consists of ≈850 lines of code (LOC).[9] As described below, **we have (partially) verified our encoding rules**, but trust our calculus (Sect. 4.3, ≈150 LOC) and our flattening implementations (Fig. 2, ≈160 LOC).

While porting rules, **we found 4 bugs in CirC's original field-blaster** (see Appendix G), including a severe soundness bug. Given a ZKP compiled with CirC, the bug allowed a prover to incorrectly compare bit-vectors. The prover, for example, could claim that the unsigned value of `0010` is greater than *or less than* that of `0001`. A patch to fix all 4 bugs (in the original field blaster) has been upstreamed, and we are in the process of upstreaming our new field blaster implementation into CirC.

6.1 Verification Evaluation

Our implementation constructs the VCs from Sect. 5.2 and emits them as SMT-LIB (extended with a theory of finite fields [47]). We verify them with cvc5, because it can solve formulas over bit-vectors and prime fields [47]. The verification is partial in that it is bounded in two ways. We set $b \in \mathbb{N}$ to be the maximum bit-width of any bit-vector and $a \in \mathbb{N}$ to be the maximum number of arguments to any n-ary operator. In our evaluation, we used $a = 4$ and $b = 4$. These bounds are small, but they were sufficient to find the bugs mentioned above.

[8] The different soundness conditions for instance and witness variables play a key role in the proof of Theorem 3. Essentially: since the condition for instances replaces variables with their definitions, the validity of the encodings of instance variables need not be explicitly enforced in A. This is why some constraints could be omitted in our field-blasting example.(See footnote 5).

[9] Our implementation is in Rust, as is CirC.

Optimizing Completeness VCs. Generally, cvc5 verifies soundness VCs more quickly than completeness VCs. This is surprising at first glance. To see why, consider the soundness (S) and completeness (C) conditions for a conversion rule from e to e' that generates assertions A and definitions F:

$$S \triangleq (A \wedge \mathit{valid}(e,t)) \rightarrow \mathit{valid}(e',t) \qquad C \triangleq (\mathit{valid}(e,t) \rightarrow (A \wedge \mathit{valid}(e',t)))[F]$$

In both, t is a variable, e contains variables, and there are variables in e' and A that are defined by F. In C, though, some variables are replaced by their definitions in F—which makes the number of variables (and thus the search space)—seem smaller for C than S. Yet, cvc5 is slower on C.

The problem is that, while the field operations in A are standard (e.g., +, ×, and =), the definitions in F use a CirC-IR operator that (once embedded into SMT-LIB) is hard for cvc5 to reason about. That operator, `(ff2bv` b`)`, takes a prime field element x and returns a bit-vector v. If x's integer representative is less than 2^b, then v's unsigned value is equal to x; otherwise, v is zero.

The `ff2bv` operator is trivial to evaluate but hard to embed. cvc5's SMT-LIB extension for prime fields only supports +, × and =, so no operator can directly relate x to v. Instead, we encode the relationship through b Booleans that represent the bits of v. To test whether $x < 2^b$, we use the polynomial $f(x) = \prod_{i=0}^{2^b-1}(x-i)$, which is zero only on $[0, 2^b-1]$. The bit-splitting essentially forces cvc5 to guess v's value; further, f's high degree slows down the Gröbner basis computations that form the foundation of cvc5's field solver.

To optimize verification of the completeness VCs, we reason about CirC-IR directly. First, we use the uniqueness of valid encodings and the *fromTerm* function. Since the VC assumes $\mathit{valid}(e,t)$, we know e is equal to $\mathit{fromTerm}(t, \mathsf{kind}(e))$. We use this equality to eliminate e from the completeness VC, leaving:

$$(A \wedge \mathit{valid}(e',t))[F][e \mapsto \mathit{fromTerm}(t, \mathsf{kind}(e))]$$

Since F defines all variables in A and e', the only variable after substitution is t. So, when t is a Boolean or small bit-vector, an exhaustive search is very effective;[10] we implemented such a solver in 56 LOC, using CirC's IR as a library.

For soundness VCs, this approach is less effective. The *fromTerm* substitution still applies, but if F introduces fresh field variables, they are not eliminated and thus, the final formula contains field variables, so exhaustion is infeasible.

Verification Results. We ran our VC verification on machines with Intel Xeon E5-2637 v4 CPUs.[11] Each attempt is limited to one physical core, 8GB memory, and 30 min. Figure 6 shows the number of VCs verified by cvc5 and our exhaustive solver. As expected, the exhaustive solver is effective on completeness VCs for Boolean and bit-vector rules, but ineffective on soundness VCs for rules that introduce fresh field variables. There are four VCs that neither solver verifies

[10] So long as the exhaustive solver reasons directly about all CirC-IR operators.

[11] We omit the completeness VCs for `ff2bv`. See Appendix C.

Type	Prop.	VCs	Verified			Unver.
			cvc5	exhaust	either	
const	—	6	6	5	6	0
conv	C	8	8	8	8	0
conv	S	8	8	4	8	0
eq	C	10	10	9	10	0
eq	S	10	10	9	10	0
op	C	259	247	247	259	0
op	S	263	259	126	259	4
uniq	—	40	40	0	40	0
var	C	12	12	10	12	0
var	S	6	6	0	6	0

Fig. 6. VCs verified by different solvers. 'uniq' denotes the VCs of Table 3; others are from Table 4. 'C' denotes completeness; 'S': soundness.

Metric	Unverified	Verified
Time (s)	27.27	25.05
Mem. (GB)	6.56	6.42
Constraints	559445	559445

Fig. 7. The performance of CirC with the verified and unverified field-blaster. Metrics are summed over the 61 functions in the Z# standard library.

within 30 min: `bvadd` with ($b = 4$, $a = 4$), and `bvmul` with ($b = 3$, $a = 4$) and ($b = 4$, $a \geq 3$). Most other VCs verify instantly. In Appendix E, we analyze how VC verification time depends on a and b.

6.2 Performance and Output Quality Evaluation

We compare CirC with our field-baster ("Verified") against CirC with its original field-blaster ("Unverified")[12] on three metrics: compiler runtime, memory usage, and the final R1CS constraint count. Our benchmark set is the standard library for CirC's Z# input language (which extends ZoKrates [16,68] v0.6.2). Our testbed runs Linux with 32GB memory and an AMD Ryzen 2700.

There is no difference in constraints, but the verified field-blaster slightly improves compiler performance: –8% time and –2% memory (Fig. 7). We think that the small improvement is unrelated to the fact that the new field blaster is verified. In Appendix E, we discuss compiler performance further.

7 Discussion

In this work, we present the first automatically verifiable field-blaster. We view the field-blaster as a set of rules; if some (automatically verifiable) conditions hold for each rule, then the field-blaster is correct. We implemented a performant and partially verified field-blaster for CirC, finding 4 bugs along the way.

Our approach has limitations. First, we require the field-blaster to be written as a set of encoding rules. Second, we only verify our rules for bit-vectors of bounded size and operators of bounded arity. Third, we assume that each rule is a pure function: for example, it doesn't return different results depending on

[12] After fixing the bugs we found. See Sect. 6.

the time. Future work might avoid the last two limitations through bit-width-independent reasoning [42,43,67] and a DSL (and compiler) for encoding rules. It would also be interesting to extend our approach to: a ZKP with a non-prime field [7,13], a compiler IR with partial or non-deterministic semantics, or a compiler with correctness that depends on computational assumptions.

Acknowledgements. We appreciate the help and guidance of Andres Nötzli, Dan Boneh, and Evan Laufer.

This material is in part based upon work supported by the DARPA SIEVE program and the Simons foundation. Any opinions, findings, and conclusions or recommendations expressed in this report are those of the author(s) and do not necessarily reflect the views of DARPA. It is also funded in part by NSF grant number 2110397 and the Stanford Center for Automated Reasoning.

A Zero-Knowledge Proofs and Compilers

This appendix is available in the full version of the paper [49].

B Compiler Correctness Proofs

This appendix is available in the full version of the paper [49].

C CirC-IR

This appendix is available in the full version of the paper [49].

D Optimizations to the CirC Field-Blaster

This appendix is available in the full version of the paper [49].

E Verified Field-Blaster Performance Details

This appendix is available in the full version of the paper [49].

F Verifier Performance Details

This appendix is available in the full version of the paper [49].

G Bugs Found in the CirC Field Blaster

This appendix is available in the full version of the paper [49].

References

1. LLVM language reference manual. https://llvm.org/docs/LangRef.html
2. Monero technical specs. https://monerodocs.org/technical-specs/ (2022)
3. Airscript. https://github.com/0xPolygonMiden/air-script
4. Angel, S., Blumberg, A.J., Ioannidis, E., Woods, J.: Efficient representation of numerical optimization problems for SNARKs. In: USENIX Security (2022)
5. Bellés-Muñoz, M., Isabel, M., Muñoz-Tapia, J.L., Rubio, A., Baylina, J.: Circom: a circuit description language for building zero-knowledge applications. IEEE Transactions on Dependable and Secure Computing (2022)
6. Bellman. https://github.com/zkcrypto/bellman
7. Ben-Sasson, E., Bentov, I., Horesh, Y., Riabzev, M.: Scalable zero knowledge with no trusted setup. In: CRYPTO (2019)
8. Bertot, Y., Castéran, P.: Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-662-07964-5
9. Blum, M., Feldman, P., Micali, S.: Non-interactive zero-knowledge and its applications. In: STOC (1988)
10. Brown, F., Renner, J., Nötzli, A., Lerner, S., Shacham, H., Stefan, D.: Towards a verified range analysis for JavaScript JITs. In: PLDI (2020)
11. Campanelli, M., Gennaro, R., Goldfeder, S., Nizzardo, L.: Zero-knowledge contingent payments revisited: attacks and payments for services. In: CCS (2017)
12. Chen, E., Zhu, J., Ozdemir, A., Wahby, R.S., Brown, F., Zheng, W.: Silph: a framework for scalable and accurate generation of hybrid MPC protocols (2023)
13. Chiesa, A., Hu, Y., Maller, M., Mishra, P., Vesely, N., Ward, N.: Marlin: preprocessing zkSNARKs with universal and updatable SRS. In: EUROCRYPT (2020)
14. Chin, C., Wu, H., Chu, R., Coglio, A., McCarthy, E., Smith, E.: Leo: a programming language for formally verified, zero-knowledge applications (2021). https://ia.cr/2021/651
15. Cowan, M., Dangwal, D., Alaghi, A., Trippel, C., Lee, V.T., Reagen, B.: Porcupine: a synthesizing compiler for vectorized homomorphic encryption. In: PLDI (2021)
16. Eberhardt, J., Tai, S.: ZoKrates–scalable privacy-preserving off-chain computations. In: IEEE Blockchain (2018)
17. Enderton, H.B.: A mathematical introduction to logic. Elsevier (2001)
18. Fournet, C., Keller, C., Laporte, V.: A certified compiler for verifiable computing. In: CSF (2016)
19. Fox, A., Myreen, M.O., Tan, Y.K., Kumar, R.: Verified compilation of CakeML to multiple machine-code targets. In: CPP (2017)
20. Frankle, J., Park, S., Shaar, D., Goldwasser, S., Weitzner, D.: Practical accountability of secret processes. In: USENIX Security (2018)
21. Goldberg, L., Papini, S., Riabzev, M.: Cairo - a Turing-complete STARK-friendly CPU architecture (2021). https://ia.cr/2021/0163
22. Goldwasser, S., Micali, S., Rackoff, C.: The knowledge complexity of interactive proof-systems. In: STOC (1985)
23. Grubbs, P., Arun, A., Zhang, Y., Bonneau, J., Walfish, M.: Zero-knowledge middleboxes. In: USENIX Security (2022)
24. Hopwood, D., Bowe, S., Hornby, T., Wilcox, N.: Zcash protocol specification. https://raw.githubusercontent.com/zcash/zips/master/protocol/protocol.pdf (2016)

25. Jiang, K., Chait-Roth, D., DeStefano, Z., Walfish, M., Wies, T.: Less is more: refinement proofs for probabilistic proofs. IEEE S&P (2023)
26. Kamara, S., Moataz, T., Park, A., Qin, L.: A decentralized and encrypted national gun registry. In: IEEE S&P (2021)
27. Kaufmann, M., Manolios, P., Moore, J.S.: Computer-aided reasoning: ACL2 case studies, vol. 4. Springer, NY (2013). https://doi.org/10.1007/978-1-4757-3188-0
28. Kosba, A., Papadopoulos, D., Papamanthou, C., Song, D.: MIRAGE: succinct arguments for randomized algorithms with applications to universal zk-SNARKs. In: USENIX Security (2020)
29. Kosba, A., Papamanthou, C., Shi, E.: xJsnark: A framework for efficient verifiable computation. In: IEEE S&P (2018)
30. Kothapalli, A., Parno, B.: Algebraic reductions of knowledge (2022). https://ia.cr/2022/009
31. Kumar, R., Myreen, M.O., Norrish, M., Owens, S.: CakeML: A verified implementation of ML. In: POPL (2014)
32. Kundu, S., Tatlock, Z., Lerner, S.: Proving optimizations correct using parameterized program equivalence. In: PLDI (2009)
33. Lattner, C., Adve, V.: LLVM: a compilation framework for lifelong program analysis & transformation. In: CGO (2004)
34. Lerner, S., Millstein, T., Chambers, C.: Automatically proving the correctness of compiler optimizations. In: PLDI (2003)
35. Lerner, S., Millstein, T., Rice, E., Chambers, C.: Automated soundness proofs for dataflow analyses and transformations via local rules. In: POPL (2005)
36. Leroy, X.: Formal verification of a realistic compiler. Commun. ACM **52**(7), 107–115 (2009)
37. Leroy, X.: A formally verified compiler back-end. J. Autom. Reason. **43**(4), 363–446 (2009)
38. Lopes, N.P., Lee, J., Hur, C.K., Liu, Z., Regehr, J.: Alive2: bounded translation validation for LLVM. In: PLDI (2021)
39. Lopes, N.P., Menendez, D., Nagarakatte, S., Regehr, J.: Provably correct peephole optimizations with Alive. In: PLDI (2015)
40. Mullen, E., Zuniga, D., Tatlock, Z., Grossman, D.: Verified peephole optimizations for CompCert. In: PLDI (2016)
41. Necula, G.C.: Translation validation for an optimizing compiler. In: PLDI (2000)
42. Niemetz, A., Preiner, M., Reynolds, A., Zohar, Y., Barrett, C., Tinelli, C.: Towards bit-width-independent proofs in SMT solvers. In: CADE (2019)
43. Niemetz, A., Preiner, M., Reynolds, A., Zohar, Y., Barrett, C., Tinelli, C.: Towards satisfiability modulo parametric bit-vectors. J. Autom. Reason. **65**(7), 1001–1025 (2021)
44. Nipkow, T., Wenzel, M., Paulson, L.C.: Isabelle/HOL: a proof assistant for higher-order logic. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45949-9
45. Noir. https://noir-lang.github.io/book/index.html
46. Ozdemir, A., Brown, F., Wahby, R.S.: CirC: Compiler infrastructure for proof systems, software verification, and more. In: IEEE S&P (2022)
47. Ozdemir, A., Kremer, G., Tinelli, C., Barrett, C.: Satisfiability modulo finite fields. In: submission (2022). https://ia.cr/2023/091
48. Ozdemir, A., Wahby, R., Whitehat, B., Boneh, D.: Scaling verifiable computation using efficient set accumulators. In: USENIX Security (2020)
49. Ozdemir, A., Wahby, R.S., Brown, F., Barrett, C.: Bounded verification for finite-field-blasting. Cryptology ePrint Archive (2023) (Full version)

50. Parno, B., Howell, J., Gentry, C., Raykova, M.: Pinocchio: nearly practical verifiable computation. Commun. ACM **59**(2), 103–112 (2016)
51. Pnueli, A., Siegel, M., Singerman, E.: Translation validation. In: TACAS (1998)
52. Ranise, S., Tinelli, C., Barrett, C.: SMT fixed size bit-vectors theory. https://smtlib.cs.uiowa.edu/theories-FixedSizeBitVectors.shtml (2017)
53. Sasson, E.B., Chiesa, A., Garman, C., Green, M., Miers, I., Tromer, E., Virza, M.: Zerocash: decentralized anonymous payments from Bitcoin. In: IEEE S&P (2014)
54. Setty, S., Braun, B., Vu, V., Blumberg, A.J., Parno, B., Walfish, M.: Resolving the conflict between generality and plausibility in verified computation. In: EuroSys (2013)
55. Snarky. https://github.com/o1-labs/snarky
56. Stewart, G., Beringer, L., Cuellar, S., Appel, A.W.: Compositional CompCert. In: POPL (2015)
57. Tan, Y.K., Myreen, M.O., Kumar, R., Fox, A., Owens, S., Norrish, M.: The verified CakeML compiler backend. J. Funct. Programm. **29**, E2 (2019)
58. Tao, R., et al.: Giallar: push-button verification for the Qiskit quantum compiler. In: PLDI (2022)
59. Thaler, J.: Proofs, Arguments, and Zero-Knowledge. Manuscript (2022)
60. The Qiskit authors and maintainers: Qiskit: an open-source framework for quantum computing (2021). https://doi.org/10.5281/zenodo.2573505. The Qiskit maintainers request that the full list of Qiskit contributors be included in any citation. Regretfully, we cannot comply, as the list is two pages long
61. Tinelli, C.: SMT core theory. https://smtlib.cs.uiowa.edu/theories-Core.shtml (2015)
62. Viand, A., Jattke, P., Hithnawi, A.: SoK: fully homomorphic encryption compilers. In: IEEE S&P (2021)
63. Wahby, R.S., Setty, S., Howald, M., Ren, Z., Blumberg, A.J., Walfish, M.: Efficient RAM and control flow in verifiable outsourced computation. In: NDSS (2015)
64. Walfish, M., Blumberg, A.J.: Verifying computations without reexecuting them. Commun. ACM **58**(2), 74–84 (2015)
65. Wang, F.: Ecne: automated verification of ZK circuits (2022). https://0xparc.org/blog/ecne
66. Zinc. https://zinc.matterlabs.dev/
67. Zohar, Y., et al.: Bit-precise reasoning via Int-blasting. In: CADE (2022)
68. ZoKrates. https://zokrates.github.io/