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
[2] Carnegie Mellon

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 [25, 52] and Monero [2],to research into auditable sealed court orders [21], private gun registries [26], privacy-preserving middleboxes [24], and zero-knowledge proofs of exploitability [12]. 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 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, 15, 22, 45, 54, 64] and academic [4, 19, 28, 29, 46, 48, 49, 53, 62] 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 non-trivial; 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.[3] 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 our techniques is not limited to CirC: most ZKP compilers include something like the field-blaster we describe here.

[3] 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.

In the next sections, we discuss related work (§1.1), give background on ZKPs and CirC (§2), present a field-blasting example (§3), describe our architecture (§4), give our verification conditions (§5), and present the case study (§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 [20, 56]. While such foundational verification offers strong guarantees, it imposes a heavy proof burden; creating CompCert, for example, took an expert team eight years [55], and any updates to compiler code require updates to proofs.

Automated verification, in contrast, does not require writing and maintaining manual proofs.[4] 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, 50] for LLVM [33], and VeRA [11] verifies range analysis in the Firefox JavaScript engine.

There is also work on verified compilation for domains more closely related to ZKPs. The Porcupine [16] compiler automatically synthesizes representations for fully-homomorphic encryption [61], and Gillar [57] proves that optimization passes in the Qiskit [59] 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 [19] builds on CompCert [36]. The authors formulate a correctness definition that preserves the *existential soundness* of a ZKP

[4] 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.

but does not consider completeness, knowledge soundness, or zero-knowledge (see Section 2.2). Leo [15] 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.

2 Background

2.1 Logic

We assume usual terminology for many-sorted first-order logic with equality ([18] 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 $\sigma \times \sigma \to \mathsf{Bool}$ for each sort σ. We also assume a family of conditionals: symbols ite_σ ("if-then-else", abbreviated ite) of sort $\mathsf{Bool} \times \sigma \times \sigma \to \sigma$.

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 [51, 60], 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 $\hat{t} : \mathsf{dom}(x_1) \times \cdots \times \mathsf{dom}(x_n) \to \mathsf{dom}(t)$ as follows. Let $\vec{\mathsf{x}} \in \mathsf{dom}(x_1) \times \cdots \times \mathsf{dom}(x_n)$. Let $\mathcal{M}$ be an interpretation that interprets each x_i as x_i. Then $\hat{t}(\vec{\mathsf{x}}) = t^{\mathcal{M}}$ (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,

$$\begin{tikzcd}[column sep=small] \underline{\mathcal{P}(\phi, x, w)} & & \mathsf{Setup}(\phi) \arrow[ll, "\mathsf{pk}"'] \arrow[rr, "\mathsf{vk}"] & & \underline{\mathcal{V}(\phi, x)} \\ \mathsf{Prove}(\mathsf{pk}, x, w) \arrow[rrrr, "\pi"] & & & & \mathsf{Verify}(\mathsf{vk}, x, \pi) \end{tikzcd}$$

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

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 [58] 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* [10]: 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* [23]: 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 (§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 $\mathsf{Ext}_w : \mathsf{dom}(x) \times \mathsf{dom}(w) \to \mathsf{dom}(w')$.[5] 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 Section 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 [63]. 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.

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

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

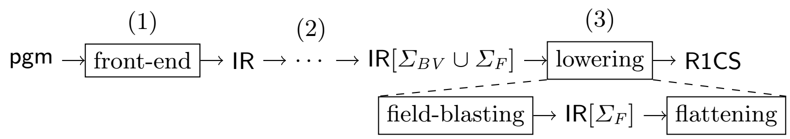


Fig. 2: The architecture of CirC

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, 13], 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.[6] 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 (§3.1). This example will illustrate four key ideas (§3.2) that inspire our field-blaster's architecture.

3.1 An example of field-blasting

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

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

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

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	
	$x_3 \approx w_2 \times w_2$	$x'_3 \approx w'_2 \times w'_2$		

Table 3: New variables and assertions when compiling the example ϕ.

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 3 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: $w'_0(w'_0 - 1) \approx 0$.[7] $x_0 \oplus w_0$ is then represented by the expression $1 - x'_0 - w'_0 + 2x'_0 w'_0$. 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 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

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

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 $w_{1,0}', \dots, w_{1,3}'$, and we represent the instance variable x_1 as $x_{1,u}'$.[8] 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 $x_{2,0}', \dots, x_{2,3}'$ 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.

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

[8] 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.[7]

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	$t \approx f$

Table 4: Encodings for each term sort. Only bit-vectors have two encoding kinds.

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 (§4.1), the five different types of encoding rules (§4.2), and a calculus that iteratively applies these rules to compile all of ϕ (§4.3).

4.1 Encodings

Table 4 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 $\mathsf{fresh}(\mathsf{name}, t, \mathsf{isInst}) \to x'$ 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 or 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 $\mathsf{assert}(t')$.

Rule Types. There are five types of rules: (1) Variable rules $\mathsf{variable}(t, \mathsf{isInst}) \to e$ take a variable t and its instance/witness status and return an encoding of that

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

$$
\begin{array}{l}
\text{fn assertEq}(e : \mathsf{Enc}, e' : \mathsf{Enc}): \\
\quad \text{if } \mathsf{kind}(e) = \mathtt{bits}: \\
\qquad \text{for } i \text{ in } [0, \mathsf{size}(\mathsf{terms}(e)) - 1]: \\
\qquad\quad \mathsf{assert}(\mathsf{terms}(e)[i] \approx \mathsf{terms}(e')[i]) \\
\quad \text{elif } \mathsf{kind}(e) = \mathtt{uint}: \\
\qquad \mathsf{assert}(\mathsf{terms}(e)[0] \approx \mathsf{terms}(e')[0]) \\
\\
\text{fn convert}(e : \mathsf{Enc}, \mathsf{kind}' : \mathsf{Kind}) \to \mathsf{Enc}: \\
\quad t \leftarrow \mathsf{encoded\_term}(e) \\
\quad \text{if } \mathsf{kind}(e) = \mathtt{bits} \text{ and } \mathsf{kind}' = \mathtt{uint}: \\
\qquad \text{return } t, \mathtt{uint}, \sum_i 2^i \mathsf{terms}(e)[i] \\
\quad \text{elif } \mathsf{kind}(e) = \mathtt{uint} \text{ and } \mathsf{kind}' = \mathtt{bits}: \\
\qquad e' \leftarrow \mathsf{variable}(t, \bot) \\
\qquad \mathsf{assert}(\mathsf{terms}(e)[0] \approx \sum_i 2^i \mathsf{terms}(e')[i]) \\
\qquad \text{return } e'
\end{array}
$$

Fig. 5: 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.

variable made up of fresh variables. (2) Constant rules $\mathsf{const}(t) \to e$ 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 Section 4.3).[9] The const rule cannot call fresh or assert. (3) Equality rules $\mathsf{assertEq}(e, e')$ take two encodings of the same kind and emit assertions that equate the underlying terms. (4) Conversion rules $\mathsf{convert}(e, \mathsf{kind}') \to e'$ 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 (Sec. 4.3) calls the convert rule to ensure the input encodings are the correct kind. Figure 5 gives pseudocode for the first four rule types, as applied to bit-vectors. Figure 6 gives pseudocode for two bit-vector operator encoding rules. A field blaster uses many operator rules: in our case study (Sec. 6) there are 46.

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.

---

[9]Having $\mathsf{const}(t)$ 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 ← size(terms(e))
    for i in [0, w − 1]:
      t'_i ← terms(e)[i]
    for i in [0, o.newBits − 1]:
      t'_{w+i} ← 0
    return t, bits, t'_0, ..., t'_{w+o.newBits−1}
  else:
    return t, kind(e), terms(e)
```

```
fn bvMulUint(t, o : Op, ⃗e : [Enc]) :
  w ← size(sort(encoded_term(e[0])))
  W ← size(⃗e) × w
  assume W < ⌊log_2 p⌋
  s' ← ∏_i terms(e_i)[0]
  b ← ff2bv(W, s')
  for i in [0, W − 1]:
    t'_i ← fresh(i, ite(b[i], 1, 0), ⊥)
    assert(t'_i(t'_i − 1) ≈ 0)
  assert(s' ≈ ∑_{i=0}^{W−1} 2^i t'_i)
  return t, bits, t'_0, ..., t'_{w−1}
```

Fig. 6: 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 Section 6.1).

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 $\mathsf{fresh}(v, t, \dots)$.

Figure 7 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.

To meet the requirements of the ZKP compiler, our calculus must return two extension function: Ext_x and Ext_w (Sec. 2.2). Both can be constructed from the fresh variable definitions F. One subtlety is that $\mathsf{Ext}_x(x)$ (which assigns values

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

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

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

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

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

Fig. 7: The transition rules of our rewriting calculus.

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 (App. D) but *not* correctness.

5 Verification conditions

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

5.1 Correctness definition

Definition 1 (Correctness). *A ZKP compiler* $\mathsf{Compile}(\phi) \to (\phi', \mathsf{Ext}_x, \mathsf{Ext}_w)$ *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 $\mathsf{Inv}(\mathsf{x}', \mathsf{w}') \to \mathsf{w}$ such that for all $\mathsf{x} \in \mathsf{dom}(x), \mathsf{w}' \in \mathsf{dom}(w')$ such that $\hat{\phi}'(\mathsf{Ext}_x(\mathsf{x}), \mathsf{w}') = \top$,
$$\hat{\phi}(\mathsf{x}, \mathsf{Inv}(\mathsf{Ext}_x(\mathsf{x}), \mathsf{w}')) = \top$$

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

Table 8: VCs related to encoding uniqueness.

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* $\mathsf{Compose}(\mathsf{Compile}',\mathsf{Compile}'')$ *(Fig. 12, App. A.2) 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 (Sec. 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 $\mathit{valid}(e,t)$, which is defined to be the validity condition in Table 4. Each VC containing encoding variables $\vec{e}$ implicitly represents a conjunction of instances of that VC, one for each possible tuple of kinds of $\vec{e}$, which is fixed for each instance. If a VC contains $\mathit{valid}(e,t)$, 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 4. We define the equality predicate $\mathit{equal}(e,e')$ 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 8 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

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

Table 9: VCs for encoding rules.

first row, we consider only instances for which $\mathsf{kind}(e) = \mathsf{kind}(e')$). The function $\mathit{fromTerm}(t, \mathsf{kind}) \rightarrow e$ maps a term and an encoding kind to a valid encoding of that kind, and the function $\mathit{toTerm}(e) \rightarrow t$ 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 (App. B.1) and we will use *fromTerm* to optimize VCs for faster verification (Sec. 6.1).

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 $\mathit{fromTerm}(b, \mathtt{bit})$ is $(b, \mathtt{bit}, \mathit{ite}(b, 1, 0))$.

VCs for encoding rules. Table 9 shows our VCs for the rules of Figure 7. 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).[10] 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 Section 4.3 is a correct ZKP compiler. The proof is in Appendix B.

Theorem 3 (Correctness). *With rules that satisfy the conditions of Section 5.2, the calculus of Section 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 (§4.2) and consists of ≈850 lines of code (LOC).[11] As described below, **we have (partially) verified our encoding rules**, but trust our calculus (§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 App. 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 Section 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

[10] 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.[7]

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

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.

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:

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

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., $+$, $\times$, 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 $+$, $\times$ 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:

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

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;[12] 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.[13] Each attempt is limited to one physical core, 8GB memory, and 30 minutes. Table 10 shows the number of VCs verified by cvc5 and our

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

[13] 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. 10: VCs verified by different solvers.'uniq' denotes the VCs of Table 8; others are from Table 9. 'C' denotes completeness; 'S': soundness.

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

Fig. 11: The performance of CirC with the verified and unverified field-blaster. Metrics are summed over the 61 functions in the Z# standard library.

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 within 30 minutes: `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")[14] 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 [17, 66] 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 (Table 11). 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.

[14] After fixing the bugs we found. See Section 6.

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 the time. Future work might avoid the last two limitations through bit-width-independent reasoning [42, 43, 65] 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, 14], 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.

Bibliography

[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 Science & Business Media (2013)

[9] Bitansky, N., Canetti, R., Chiesa, A., Goldwasser, S., Lin, H., Rubinstein, A., Tromer, E.: The hunting of the SNARK. Journal of Cryptology **30**(4), 989–1066 (Oct 2017)

[10] Blum, M., Feldman, P., Micali, S.: Non-interactive zero-knowledge and its applications. In: STOC (1988)

[11] Brown, F., Renner, J., Nötzli, A., Lerner, S., Shacham, H., Stefan, D.: Towards a verified range analysis for JavaScript JITs. In: PLDI (2020)

[12] Campanelli, M., Gennaro, R., Goldfeder, S., Nizzardo, L.: Zero-knowledge contingent payments revisited: Attacks and payments for services. In: CCS (2017)

[13] 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)

[14] Chiesa, A., Hu, Y., Maller, M., Mishra, P., Vesely, N., Ward, N.: Marlin: preprocessing zkSNARKs with universal and updatable SRS. In: EUROCRYPT (2020)

[15] Chin, C., Wu, H., Chu, R., Coglio, A., McCarthy, E., Smith, E.: Leo: A programming language for formally verified, zero-knowledge applications (2021), Preprint at https://ia.cr/2021/651

[16] Cowan, M., Dangwal, D., Alaghi, A., Trippel, C., Lee, V.T., Reagen, B.: Porcupine: A synthesizing compiler for vectorized homomorphic encryption. In: PLDI (2021)

[17] Eberhardt, J., Tai, S.: ZoKrates—scalable privacy-preserving off-chain computations. In: IEEE Blockchain (2018)

[18] Enderton, H.B.: A mathematical introduction to logic. Elsevier (2001)
[19] Fournet, C., Keller, C., Laporte, V.: A certified compiler for verifiable computing. In: CSF (2016)
[20] Fox, A., Myreen, M.O., Tan, Y.K., Kumar, R.: Verified compilation of CakeML to multiple machine-code targets. In: CPP (2017)
[21] Frankle, J., Park, S., Shaar, D., Goldwasser, S., Weitzner, D.: Practical accountability of secret processes. In: USENIX Security (2018)
[22] Goldberg, L., Papini, S., Riabzev, M.: Cairo – a Turing-complete STARK-friendly CPU architecture (2021), Preprint at https://ia.cr/2021/0163
[23] Goldwasser, S., Micali, S., Rackoff, C.: The knowledge complexity of interactive proof-systems. In: STOC (1985)
[24] Grubbs, P., Arun, A., Zhang, Y., Bonneau, J., Walfish, M.: Zero-knowledge middleboxes. In: USENIX Security (2022)
[25] Hopwood, D., Bowe, S., Hornby, T., Wilcox, N.: Zcash protocol specification. https://raw.githubusercontent.com/zcash/zips/master/protocol/protocol.pdf (2016)
[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 Science & Business Media (2013)
[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), Preprint at 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. Communications of the ACM **52**(7), 107–115 (2009)
[37] Leroy, X.: A formally verified compiler back-end. Journal of Automated Reasoning **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. Journal of Automated Reasoning **65**(7), 1001–1025 (2021)
[44] Nipkow, T., Wenzel, M., Paulson, L.C.: Isabelle/HOL: a proof assistant for higher-order logic. Springer (2002)
[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 (2022), In submission, Preprint at 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] Parno, B., Howell, J., Gentry, C., Raykova, M.: Pinocchio: Nearly practical verifiable computation. Communications of the ACM **59**(2), 103–112 (2016)
[50] Pnueli, A., Siegel, M., Singerman, E.: Translation validation. In: TACAS (1998)
[51] Ranise, S., Tinelli, C., Barrett, C.: SMT fixed size bit-vectors theory. https://smtlib.cs.uiowa.edu/theories-FixedSizeBitVectors.shtml (2017)
[52] 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)
[53] 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)
[54] Snarky. https://github.com/o1-labs/snarky
[55] Stewart, G., Beringer, L., Cuellar, S., Appel, A.W.: Compositional CompCert. In: POPL (2015)
[56] Tan, Y.K., Myreen, M.O., Kumar, R., Fox, A., Owens, S., Norrish, M.: The verified CakeML compiler backend. Journal of Functional Programming **29** (2019)
[57] Tao, R., Shi, Y., Yao, J., Li, X., Javadi-Abhari, A., Cross, A.W., Chong, F.T., Gu, R.: Giallar: push-button verification for the Qiskit quantum compiler. In: PLDI (2022)
[58] Thaler, J.: Proofs, Arguments, and Zero-Knowledge. Manuscript (2022)
[59] 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 contributers be included in any citation. Regretfully, we cannot comply, as the list is two pages long.
[60] Tinelli, C.: SMT core theory. https://smtlib.cs.uiowa.edu/theories-Core.shtml (2015)
[61] Viand, A., Jattke, P., Hithnawi, A.: SoK: Fully homomorphic encryption compilers. In: IEEE S&P (2021)
[62] Wahby, R.S., Setty, S., Howald, M., Ren, Z., Blumberg, A.J., Walfish, M.: Efficient RAM and control flow in verifiable outsourced computation. NDSS (2015)
[63] Walfish, M., Blumberg, A.J.: Verifying computations without reexecuting them. Commun. ACM **58**(2), 74–84 (2015)
[64] Zinc. https://zinc.matterlabs.dev/
[65] Zohar, Y., Irfan, A., Mann, M., Niemetz, A., Nötzli, A., Preiner, M., Reynolds, A., Barrett, C., Tinelli, C.: Bit-precise reasoning via int-blasting. In: CADE (2022)
[66] ZoKrates. https://zokrates.github.io/

A Zero-knowledge proofs & compilers

A.1 Definition of a zkSNARK

A zero-knowledge succinct argument of knowledge (zkSNARK) is a triple of algorithms that is Complete, Succinct, Knowledge Sound, and Zero Knowledge, as defined below. The below definitions are adapted from Bitansky et al. [9].

Definition 2 (zkSNARK Π' for predicate class Φ').

$\mathsf{Setup}'(\hat{\phi}', 1^\lambda) \rightarrow (\mathsf{pk}', \mathsf{vk}')$ *This algorithm generates proving key* pk' *and verification key* vk' *for predicate* $\phi' \in \Phi'$ *and security parameter* $\lambda \in \mathbb{N}$.[15]

$\mathsf{Prove}'(\mathsf{pk}', \mathsf{x}', \mathsf{w}') \rightarrow \pi$ *This algorithm generates a proof* π *given a proving key* pk'*, instance* x'*, and witness* w'*.*

$\mathsf{Verify}'(\mathsf{vk}', \mathsf{x}', \pi) \rightarrow \{\bot, \top\}$ *This algorithm returns* $\top$ *if* π *is a valid proof for statement* x' *under the verification key* vk'*, and* $\bot$ *otherwise.*

In the following definitions, let $\phi' \in \Phi'$ and $\lambda \in \mathbb{N}$.

Completeness means, informally, that an honest prover can generate an accepting proof for a true statement.

Definition 3 (Completeness). $\forall \mathsf{x}', \mathsf{w}' : \hat{\phi}'(\mathsf{x}', \mathsf{w}') = \top$,

$$Pr\left[\mathsf{Verify}'(\mathsf{vk}', \mathsf{x}', \pi) = \top : \begin{matrix} (\mathsf{pk}', \mathsf{vk}') \leftarrow \mathsf{Setup}'(\hat{\phi}', 1^\lambda) \\ \pi \leftarrow \mathsf{Prove}'(\mathsf{pk}', \mathsf{x}', \mathsf{w}') \end{matrix}\right] = 1$$

and Prove' *runs in time poly*$(\lambda, |\mathsf{x}'|, |\hat{\phi}'|)$*.*

[15] The security parameter λ is encoded in unary to allow the setup algorithm to run in time polynomial in the *value* of λ rather than polynomial in the *size of its representation* (say, in binary), which is exponentially smaller.

Succinctness means, informally, that proofs are very short and easy to check.

Definition 4 (Succinctness). *The length of the proof* π *generated by* Prove' *and the running time of* Verify' *is bounded by* $poly(\lambda + |\mathsf{x}'| + \log |\hat{\phi}'|)$.

Knowledge soundness, informally, means that any prover that generates an accepting proof for predicate ϕ' on instance x' must *actually know* a witness w' such that $\hat{\phi}'(\mathsf{x}', \mathsf{w}') = \top$. Here, knowledge is formalized via an algorithm that, given (exceptional) access to a prover that is able to generate convincing proofs for an instance, outputs the corresponding witness. (In other words: "if you know something, I can get you to tell it to me.") This definition of soundness is strictly stronger than *existential soundness*, which ensures only that generating a proof for a false statement is infeasible. A succinct proof system meeting this weaker definition is called a succinct non-interactive argument or *SNARG*. Moreover, since knowledge soundness and existential soundness both hold only against polynomially bounded provers, the "proof" π is called an *argument*.

Definition 5 (Knowledge Soundness). *There exists a polynomial-time extractor* $\mathcal{E}$ *such that, for any polynomially bounded (potentially cheating) prover* $\mathcal{P}^\star$ *and any large enough security parameter* $\lambda \in \mathbb{N}$,

$$Pr\left[\begin{array}{c}(\mathsf{pk}', \mathsf{vk}') \leftarrow \mathsf{Setup}'(\hat{\phi}', 1^\lambda) \\ (\mathsf{x}', \pi) \leftarrow \mathcal{P}^\star(\mathsf{pk}') \\ \mathsf{Verify}'(\mathsf{vk}', \mathsf{x}', \pi) = \top\end{array} \quad \wedge \quad \begin{array}{c}\mathsf{w}' \leftarrow \mathcal{E}^{\mathcal{P}^\star}(\mathsf{pk}', \mathsf{x}') \\ \hat{\phi}'(\mathsf{x}', \mathsf{w}') = \bot\end{array}\right] \leq \mathsf{negl}(\lambda)$$

Here, $\mathcal{E}^{\mathcal{P}^\star}$ *means that the extractor algorithm is given oracle access to* $\mathcal{P}^\star$*, including the ability to rewind the prover to a previous state and resume execution.*

Remark 1. Definition 5 is one widely used notion of knowledge soundness, but others are known. The differences among these definitions include the implementation of the extractor (e.g., it is sometimes defined as a circuit of polynomial size rather than an algorithm with polynomial runtime) and the ordering of quantifiers (e.g., for extractors defined as circuits, the cheating prover is quantified before the extractor, which allows the extractor circuit to include an implementation of the prover). While these differences are cryptographically significant, they are essentially equivalent from this work's perspective. We believe that the proof in Appendix A.3 applies to any reasonable definition of knowledge soundness.

Zero knowledge, informally, means that a proof reveals no information about w' that is not otherwise revealed by the truth of $\hat{\phi}'$ on instance x'. This notion is formalized via an algorithm that generates simulated proofs that are indistinguishable from real proofs—*without* access to the witness w'.

Definition 6 (Zero knowledge). *For* $(\mathsf{pk}', \mathsf{vk}') \leftarrow \mathsf{Setup}'(\hat{\phi}', 1^\lambda)$*, there exists a polynomial-time simulator* $\mathcal{S}$ *such that* $\forall \mathsf{x}', \mathsf{w}' : \hat{\phi}'(\mathsf{x}', \mathsf{w}') = \top$*, the following distributions are indistinguishable:*

$$\left\{\mathsf{Prove}'(\mathsf{pk}', \mathsf{x}', \mathsf{w}')\right\} \;\approx\; \left\{\mathcal{S}(\mathsf{pk}', \mathsf{x}')\right\}$$

def $\mathsf{Compose}(\mathsf{Compile}', \mathsf{Compile}'')(\phi \in \Phi) \to (\phi'' \in \Phi, \mathsf{Ext}_x, \mathsf{Ext}_w)$:
  $\phi', \mathsf{Ext}'_x, \mathsf{Ext}'_w \leftarrow \mathsf{Compile}'(\phi)$
  $\phi'', \mathsf{Ext}''_x, \mathsf{Ext}''_w \leftarrow \mathsf{Compile}''(\phi)$
  def $\mathsf{Ext}_x(\mathsf{x})$: return $\mathsf{Ext}''_x(\mathsf{Ext}'_x(\mathsf{x}))$
  def $\mathsf{Ext}_w(\mathsf{x}, \mathsf{w})$: return $\mathsf{Ext}''_w(\mathsf{Ext}'_x(\mathsf{x}), \mathsf{Ext}'_w(\mathsf{x}, \mathsf{w}))$
  return $(\phi'', \mathsf{Ext}_x, \mathsf{Ext}_w)$

Fig. 12: The composition of compilers $\mathsf{Compile}'$ and $\mathsf{Compile}''$.

If these distributions are perfectly (resp., statistically, computationally) indistinguishable, we say that the zkSNARK is perfectly (resp., statistically, computationally) zero knowledge.

A.2 Proof of compiler composition theorem

Restating Theorem 1 of Section 5.1:

Theorem (Compiler Composition). *If* $\mathsf{Compile}'$ *from class* Φ *to class* Φ' *and* $\mathsf{Compile}''$ *from class* Φ' *to class* Φ'' *are correct, there exists a compiler* $\mathsf{Compose}(\mathsf{Compile}', \mathsf{Compile}'')$ *from class* Φ *to class* Φ'' *that is correct.*

Proof. Figure 12 gives the compiler $\mathsf{Compose}(\mathsf{Compile}', \mathsf{Compile}'')$. We now argue demonstrable completeness and demonstrable soundness.

Demonstrable Completeness (Def. 1, *Demonstrable Completeness*). Fix x, w such that $\hat{\phi}(\mathsf{x}, \mathsf{w}) = \top$. Let $\mathsf{x}' = \mathsf{Ext}'_x(\mathsf{x})$, $\mathsf{w}' = \mathsf{Ext}'_w(\mathsf{x}, \mathsf{w})$, $\mathsf{x}'' = \mathsf{Ext}''_x(\mathsf{x}')$, and $\mathsf{w}'' = \mathsf{Ext}''_w(\mathsf{x}', \mathsf{w}')$. We must show that $\hat{\phi}''(\mathsf{Ext}_x(\mathsf{x}), \mathsf{Ext}_w(\mathsf{x}, \mathsf{w})) = \top$; substituting, we must show that

$$\hat{\phi}''(\mathsf{x}'', \mathsf{w}'') = \top \tag{2}$$

From the demonstrable correctness of $\mathsf{Compile}'$, $\hat{\phi}(\mathsf{x}, \mathsf{w}) = \top$ implies that:

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

From the demonstrable correctness of $\mathsf{Compile}''$, this implies our goal (2).

Demonstrable Soundness (Def. 1, *Demonstrable Soundness*). Fix x, w'' such that $\hat{\phi}''(\mathsf{Ext}_x(\mathsf{x}), \mathsf{w}'') = \top$. Let Inv' be the algorithm guaranteed to exists by the demonstrable soundness of $\mathsf{Compile}'$, and let Inv'' be the algorithm guaranteed to exists by the demonstrable soundness of $\mathsf{Compile}''$. Further, let $\mathsf{x}' = \mathsf{Ext}'_x(\mathsf{x})$, $\mathsf{w}' = \mathsf{Inv}''(\mathsf{x}', \mathsf{w}'')$ and $\mathsf{w} = \mathsf{Inv}'(\mathsf{x}, \mathsf{w}')$. Define $\mathsf{Inv}(\mathsf{x}, \mathsf{w}'') = \mathsf{w}$; we must show that $\hat{\phi}(\mathsf{x}, \mathsf{Inv}(\mathsf{x}, \mathsf{w}'')) = \top$; substituting, we must show that

$$\hat{\phi}(\mathsf{x}, \mathsf{w}) = \top \tag{3}$$

By the demonstrable soundness of $\mathsf{Compile}''$, $\hat{\phi}''(\mathsf{Ext}_x(\mathsf{x}), \mathsf{w}'') = \top$ implies that

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

By the demonstrable soundness of $\mathsf{Compile}'$, this implies our goal (3).

$\mathsf{Setup}(\hat{\phi}, 1^\lambda) \to (\mathsf{pk}, \mathsf{vk})$:
$(\hat{\phi}', \mathsf{Ext}_x, \mathsf{Ext}_w) \leftarrow \mathsf{Compile}(\hat{\phi})$
$(\mathsf{pk}', \mathsf{vk}') \leftarrow \mathsf{Setup}'(\hat{\phi}', 1^\lambda)$
$\mathsf{pk} \leftarrow (\mathsf{Ext}_x, \mathsf{Ext}_w, \mathsf{pk}')$
$\mathsf{vk} \leftarrow (\mathsf{Ext}_x, \mathsf{vk}')$

$\mathsf{Prove}(\mathsf{pk}, \mathsf{x}, \mathsf{w}) \to \pi$:
$(\mathsf{Ext}_x, \mathsf{Ext}_w, \mathsf{pk}') \leftarrow \mathsf{pk}$
$\mathsf{x}' \leftarrow \mathsf{Ext}_x(\mathsf{x})$
$\mathsf{w}' \leftarrow \mathsf{Ext}_w(\mathsf{x}, \mathsf{w})$
$\pi \leftarrow \mathsf{Prove}'(\mathsf{pk}', \mathsf{x}', \mathsf{w}')$

$\mathsf{Verify}(\mathsf{vk}, \mathsf{x}, \pi) \to \{\bot, \top\}$:
$(\mathsf{Ext}_x, \mathsf{vk}') \leftarrow \mathsf{vk}$
$\mathsf{x}' \leftarrow \mathsf{Ext}_x(\mathsf{x})$
$\mathsf{Verify}'(\mathsf{vk}', \mathsf{x}', \pi)$

Fig. 13: A ZKP $\Pi = (\mathsf{Setup}, \mathsf{Prove}, \mathsf{Verify})$ for predicate class Φ and security parameter λ, based on compiler $\mathsf{Compile}$ from class Φ to class Φ' and a ZKP $\Pi' = (\mathsf{Setup}', \mathsf{Prove}', \mathsf{Verify}')$ for predicate class Φ'.

A.3 Proof of zkSNARK generalization

Restating Theorem 2 of Section 5.1:

Theorem (zkSNARK Generalization). *Given a zkSNARK Π' for predicate class Φ' and a correct ZKP compiler* $\mathsf{Compile}$ *from class Φ to class Φ', there exists a zkSNARK Π for predicate class Φ.*

Proof. Figure 13 gives the algorithms $(\mathsf{Setup}, \mathsf{Prove}, \mathsf{Verify})$ comprising Π. By inspection, these match the zkSNARK syntax given in Definition 2.

Completeness (Def. 3): By the demonstrable completeness of $\mathsf{Compile}$ (Def. 1, §5.1) and the definitions of ϕ', x', and w' in Figure 13, we have that $\hat{\phi}'(\mathsf{x}', \mathsf{w}') = \top$ whenever $\hat{\phi}(\mathsf{x}, \mathsf{w}) = \top$. Since Π' is complete, this implies that the output of Prove' is an accepting proof as long as the value of x' is the same in the Prove and Verify algorithms; this is true because Ext_x is deterministic.

Succinctness (Def. 4): $\mathsf{Compile}$ and Ext_x are efficient, so $|\mathsf{x}'| \in \mathrm{poly}(|\mathsf{x}|)$ and $|\hat{\phi}'| \in \mathrm{poly}(|\hat{\phi}|)$. $\mathrm{poly}(\lambda + |\mathsf{x}'| + \log|\hat{\phi}'|)$ and $\mathrm{poly}(\lambda + |\mathsf{x}| + \log|\hat{\phi}|)$ are thus equivalent. The output π of Prove is identically the output of Prove' and Π' is succinct, so $|\pi| \in \mathrm{poly}(\lambda + |\mathsf{x}| + \log|\hat{\phi}|)$. The succinctness of Π' implies that $|\mathsf{vk}'| \in \mathrm{poly}(\lambda + |\mathsf{x}'| + \log|\hat{\phi}'|)$ by the bound on the runtime of Verify', implying a corresponding runtime to unpack vk in the first step of Verify. The efficiency bound on Ext_x (Note 5) guarantees that the size of Ext_x's description and its runtime are $\mathrm{poly}(|\mathsf{x}|)$. Verify thus runs in time $\mathrm{poly}(\lambda + |\mathsf{x}| + \log|\hat{\phi}|)$ as required.

Knowledge soundness (Def. 5): The extractor for Π works as follows. On inputs $(\mathsf{pk}, \mathsf{x})$, unpack pk, compute $\mathsf{x}' \leftarrow \mathsf{Ext}_x(\mathsf{x})$, and invoke the extractor for Π' on $(\mathsf{pk}', \mathsf{x}')$ to obtain w', then output $\mathsf{w} \leftarrow \mathsf{Inv}(\mathsf{x}', \mathsf{w}')$. By the knowledge soundness of Π', $\hat{\phi}'(\mathsf{x}', \mathsf{w}') = \top$ except with negligible probability. By the demonstrable soundness of $\mathsf{Compile}$, $\mathsf{Inv}(\mathsf{x}', \mathsf{w}')$ returns a value w satisfying $\hat{\phi}(\mathsf{x}, \mathsf{w}) = \top$. Thus, the extractor for Π succeeds except when the extractor for Π' fails, which happens with negligible probability as required.

Zero knowledge (Def. 6): The simulator for Π works as follows. On inputs $(\mathsf{pk}, \mathsf{x})$, unpack pk, compute $\mathsf{x}' \leftarrow \mathsf{Ext}_x(\mathsf{x})$, and invoke the simulator for Π' on $(\mathsf{pk}', \mathsf{x}')$ to obtain a simulated proof $\pi^\star$. Ext_x is deterministic and the output of Prove is $\pi \leftarrow \mathsf{Prove}'(\mathsf{pk}', \mathsf{x}', \mathsf{w}')$, so the distribution of $\pi^\star$ is indistinguishable from the distribution of π by the zero-knowledge property of Π'.

Remark 2. By a similar argument, a correct ZKP compiler generalizes a non–zero-knowledge SNARK: in this case, neither Π' nor Π have a simulator, the zero-knowledge property does not hold, and the other properties are unchanged.

Likewise, for a (zk or non-zk) SNARG the knowledge soundness property is replaced by the weaker existential soundness property (see discussion immediately before Def. 5), meaning that it is infeasible for any efficient prover to generate an accepting proof for a false statement. The equisatisfiability of $\mathsf{Compile}$ guarantees that $\hat{\phi}'(\mathsf{Ext}_x(\mathsf{x}), \mathsf{Ext}_w(\mathsf{x}, \mathsf{w})) = \bot$ whenever $\hat{\phi}(\mathsf{x}, \mathsf{w}) = \bot$. Thus, if Π' is existentially sound, Π must also be.

B Compiler correctness proofs

Here, we present proofs of our main theorems.

We begin with a note on notation. We will sometimes denote the function $\hat{f}$ defined by a term f as $\overline{f}$; this alternate notation is clearer when the term has syntactic structure, e.g., $\overline{f_1 = f_2}$, whose range is $\{\bot, \top\}$. If $\phi(x)$ is a formula in variable x, and x is a value, by $\phi[x \mapsto \mathsf{x}]$ we denote ϕ with its variable x replaced with a constant term of value x. Note that for our theories of interest (bit-vectors and prime fields), all values have a corresponding constant. When the variable x and the value being substituted x are denoted with the same letter, we will abbreviate $\phi[x \mapsto \mathsf{x}]$ as $\phi[\mathsf{x}]$. Finally, for a term ϕ which contains no variables, we denote the equation $\hat{\phi} = \mathsf{z}$ by $\phi \Downarrow \mathsf{z}$. Thus, $\Downarrow$ denotes the evaluation relation for variable-free terms and values. Evaluation for variable-free terms is unique because we allow interpretations to differ only in variables (Sec. 2.1).

B.1 Theorem 3: Demonstrable Soundness

> A compiler that implements the calculus of Section 4.3 with rules that satisfy the conditions of Section 5.2 is demonstrably sound.

Proof. Let the compiler take $\phi(x, w)$ as input and produce as output: $\phi'(x', w')$, Ext_x, and Ext_w. Fix an instance x for ϕ, and let $\mathsf{x}' \leftarrow \mathsf{Ext}_x(\mathsf{x})$. Fix a w' such that $\hat{\phi}'(\mathsf{x}', \mathsf{w}') = \top$, and let $\mathsf{z}' = (\mathsf{x}', \mathsf{w}')$. Let $\mathcal{T}$ (the "trace") be the sequence of transitions that the compiler takes. For each witness variable $w_i \in w$, there is a transition in $\mathcal{T}$ that encodes w_i as some e_i. We define a function $\mathsf{Inv}(\mathsf{z}') \rightarrow \mathsf{w}$. For variable $w_i \in w$, it outputs the evaluation of $\mathit{toTerm}(e_i)[\mathsf{z}']$. Let w denote the output of Inv and let $\mathsf{z} = (\mathsf{x}, \mathsf{w})$. It suffices to show that $\hat{\phi}(\mathsf{z}) = \top$.

By inducting over $\mathcal{T}$, we will show that for each $(t \mapsto e) \in E$ (where E is the final encoding store in $\mathcal{T}$), $\mathit{valid}(e[\mathsf{z}'], t[\mathsf{z}]) \Downarrow \top$. Initially, E is empty, so the property holds. We have one inductive case for each transition rule (except equality).

Operator rules are our first inductive case. Consider a transition for operator o applied to arguments $\vec{t}$, with operator rule r: $A', F', e' \leftarrow r(o, \vec{e})$. By the inductive hypothesis, we have that for all i, $\mathit{valid}(e_i[\mathsf{z}'], t_i[\mathsf{z}]) \Downarrow \top$. Furthermore, we have that $A'[\mathsf{z}'] \Downarrow \top$. Operator soundness requires that $(A \wedge \bigwedge_i \mathit{valid}(e_i, t_i)) \rightarrow$

$\mathit{valid}(e', o(\vec{t}))$ is valid. Then, substitutions and evaluations give the desired result:

$$
\begin{aligned}
(A \wedge \bigwedge_i \mathit{valid}(e_i, t_i)) \to \mathit{valid}(e', o(\vec{t})) \text{ is valid} && \textit{(operator soundness)}\\
(A \wedge \bigwedge_i \mathit{valid}(e_i, t_i)) \to \mathit{valid}(e', o(\vec{t}))[\mathsf{z}, \mathsf{z}'] \Downarrow \top && \textit{(sub \& eval)}\\
(A[\mathsf{z}'] \wedge \bigwedge_i \mathit{valid}(e_i[\mathsf{z}'], t_i[\mathsf{z}])) \to \mathit{valid}(e'[\mathsf{z}'], o(\vec{t})[\mathsf{z}]) \Downarrow \top && \textit{(move subs to vars)}\\
(\top \wedge \bigwedge_i \top) \to \mathit{valid}(e'[\mathsf{z}'], o(\vec{t})[\mathsf{z}]) \Downarrow \top && \textit{(eval)}\\
\mathit{valid}(e'[\mathsf{z}'], o(\vec{t})[\mathsf{z}]) \Downarrow \top && \textit{(eval)}
\end{aligned}
$$

Variables are our second inductive case. For a witness variable w_i, encoded as e_i, the value of w_i in w is that of $\mathit{toTerm}(e_i)[\mathsf{z}']$. So, we must show that $\mathit{valid}(e_i[\mathsf{z}'], \mathit{toTerm}(e_i)[\mathsf{z}']) \Downarrow \top$. Variable soundness (for witnesses) implies that there exists a term constant t of value t such that $\mathit{valid}(e_i[\mathsf{z}'], t)$ is valid. Instantiating the correctness of toTerm, we have that $\mathit{valid}(e_i[\mathsf{z}'], \mathit{toTerm}(e_i[\mathsf{z}']))$ is valid. Instantiating the uniqueness of valid encodings, the validity of two different terms for the same encoding implies that the terms are equal, i.e., $t = \mathit{toTerm}(e_i[\mathsf{z}'])$ is valid, so $\mathit{toTerm}(e_i[\mathsf{z}']) \Downarrow \mathsf{t}$. Using this equality, we eliminate t from our previous valid term, giving that $\mathit{valid}(e_i[\mathsf{z}'], \mathit{toTerm}(e_i[\mathsf{z}']))$ is valid; since in is variable-free, it also evaluates to $\top$, which is the desired conclusion.

For an instance variable x_i, encoded as e_i with definitions F', $z' \mapsto \mathsf{z}'$ agrees with F' on the instance variables of F'. Thus, instantiating variable soundness (for instances) gives $\mathit{valid}(e_i[\mathsf{z}'], x_i[\mathsf{z}])$, which is what we need.

The recursive cases for constants and conversions hold *mutatis mutandis*. These cases complete the induction.

Now, consider the final transition in $\mathcal{T}$. By construction, it is an equality transition: $e_\top \leftarrow \mathsf{const}(\top)$; $A', F' \leftarrow \mathsf{assertEq}(e_\top, e_\phi)$. By the constant completeness condition, $\mathit{valid}(e_\top, \top) \Downarrow \top$. By induction, we have $\mathit{valid}(e_\phi[\mathsf{z}'], \phi[\mathsf{z}]) \Downarrow \top$. Through instantiating equality soundness, substituting, and evaluating, we find:

$$
\begin{aligned}
(A \wedge \mathit{valid}(e_\phi, \phi) \wedge \mathit{valid}(e_\top, \top)) \to \phi = \top \text{ is valid} && \textit{(eq. sound)}\\
((A \wedge \mathit{valid}(e_\phi, \phi) \wedge \mathit{valid}(e_\top, \top)) \to \phi = \top)[\mathsf{z}, \mathsf{z}'] \Downarrow \top && \textit{(sub \& eval)}\\
((A[\mathsf{z}'] \wedge \mathit{valid}(e_\phi[\mathsf{z}'], \phi[\mathsf{z}]) \wedge \mathit{valid}(e_\top[\mathsf{z}'], \top)) \to \phi[\mathsf{z}] = \top) \Downarrow \top && \textit{(move subs)}\\
((\top \wedge \top \wedge \top) \to \phi[\mathsf{z}] = \top) \Downarrow \top && \textit{(eval)}\\
\phi[\mathsf{z}] \Downarrow \top && \textit{(eval)}
\end{aligned}
$$

This is exactly what we sought to show. Thus, our compiler is demonstrably sound.

B.2 Theorem 3: Demonstrable Completeness

A compiler that implements the calculus of Section 4.3 with rules that satisfy the conditions of Section 5.2 is demonstrably complete.

Proof. Let the compiler take $\phi(x,w)$ as input and produce as output: $\phi'(x',w')$, Ext_x, and Ext_w. Fix x and w such that $\phi(\mathsf{x},\mathsf{w}) = \top$. Let $\mathsf{z} \leftarrow (\mathsf{x},\mathsf{w})$. We must show that for $\mathsf{z}' \leftarrow (\mathsf{Ext}_x(\mathsf{x}), \mathsf{Ext}_w(\mathsf{z}))$, $\phi'[\mathsf{z}'] \Downarrow \top$. Again, let $\mathcal{T}$ be the compiler's transition sequence. Let A be the final assertion set and let F be the final sequence of fresh variable definitions. It suffices to show that $A[F][\mathsf{z}] = \top$, since the substitutions z' are equivalent to the substitution sequence defined by F and z, because of z's definition in terms of the Ext_x and Ext_w functions.

We proceed by induction on $\mathcal{T}$. Our inductive hypothesis is that after each transition, $A[F][\mathsf{z}] \Downarrow \top$ and for each $(t \mapsto e) \in E$, $\mathit{valid}(e[F],t)[\mathsf{z}] \Downarrow \top$. Initially, A and E are empty, so this holds. There is one inductive case for each transition type (except equality). The transition begins with the calculus in state (E,A,F). It might create new constraints A', new definitions F', and a new encoding e for some term t. It suffices to show that $A'[F][F'][\mathsf{z}] \Downarrow \top$ and that $\mathit{valid}(e[F][F'],t)[\mathsf{z}] \Downarrow \top$.

First, consider a transition for operator o applied to arguments $\vec{t}$, with operator rule r: $A', F', e' \leftarrow r(o, \vec{e})$. The inductive hypothesis gives that for all i, $\mathit{valid}(e_i[F],t_i)[\mathsf{z}] \Downarrow \top$. We instantiate operator completeness and proceed:

$$((\bigwedge_i \mathit{valid}(e_i,t_i)) \rightarrow (A' \wedge \mathit{valid}(e',t)))[F'] \text{ is valid} \qquad \textit{(op completeness)}$$
$$((\bigwedge_i \mathit{valid}(e_i,t_i)) \rightarrow (A' \wedge \mathit{valid}(e',t)))[F,F',\mathsf{z}] \Downarrow \top \qquad \textit{(sub \& eval)}$$
$$((\bigwedge_i \mathit{valid}(e_i[F],t_i)[\mathsf{z}]) \rightarrow (A'[F][F'] \wedge \mathit{valid}(e'[F][F'],t))[\mathsf{z}]) \Downarrow \top \qquad \textit{(move subs)}$$
$$((\bigwedge_i \top) \rightarrow (A'[F][F'] \wedge \mathit{valid}(e'[F][F'],t))[\mathsf{z}]) \Downarrow \top \qquad \textit{(eval)}$$
$$A'[F][F'][\mathsf{z}] \wedge \mathit{valid}(e'[F][F'],t)[\mathsf{z}] \Downarrow \top \qquad \textit{(eval)}$$

which is what we sought to show.

Second, consider a transition for variable z_i. Variable completeness states that the conjunction $(A' \wedge \mathit{valid}(e',z_i))[F']$ holds for all values of z_i, thus it evaluates to $\top$ under z, as desired.

The inductive cases for conversion rules, and constant rules also hold, *mutatis mutandis*. These cases complete the induction.

Finally, consider the calculus's final equality transition: $e_\top \leftarrow \mathsf{const}(\top)$; $A', F' \leftarrow \mathsf{assertEq}(e_\top, e_\phi)$. By constant completeness, $e_\top \Downarrow \top$. Moreover, $\phi[\mathsf{z}] \Downarrow \top$. By the inductive hypothesis, $\mathit{valid}(e_\phi[F],\phi)[\mathsf{z}] \Downarrow \top$. We instantiate equality completeness and proceed:

$$((\phi = \top \wedge \mathit{valid}(e_\phi,\phi) \wedge \mathit{valid}(e_\top,\top)) \rightarrow A')[F'] \text{ is valid} \qquad \textit{(eq. complete)}$$
$$((\phi = \top \wedge \mathit{valid}(e_\phi,\phi) \wedge \mathit{valid}(e_\top,\top)) \rightarrow A')[F,F',\mathsf{z}] \Downarrow \top \qquad \textit{(sub \& eval)}$$
$$((\phi[\mathsf{z}] = \top \wedge \mathit{valid}(e_\phi[F],\phi)[\mathsf{z}] \wedge \mathit{valid}(e_\top,\top)) \rightarrow A'[F][F'][\mathsf{z}]) \Downarrow \top \qquad \textit{(move subs)}$$
$$((\top = \top \wedge \top \wedge \top) \rightarrow A'[F][F'][\mathsf{z}]) \Downarrow \top \qquad \textit{(eval)}$$
$$A'[F][F'][\mathsf{z}]) \Downarrow \top \qquad \textit{(eval)}$$

Operators	Signature	Semantics
$\neg$	$\texttt{Bool} \to \texttt{Bool}$	
$\to$	$\texttt{Bool} \times \texttt{Bool} \to \texttt{Bool}$	
$\wedge, \oplus, \vee$	$\texttt{Bool}^* \to \texttt{Bool}$	
`maj`	$\texttt{Bool} \times \texttt{Bool} \times \texttt{Bool} \to \texttt{Bool}$	majority vote
$+, \times$	$\mathbb{F}_p^* \to \mathbb{F}_p$	
`ffrecip`	$\mathbb{F}_p \to \mathbb{F}_p$	zero on input zero
`ffneg`	$\mathbb{F}_p \to \mathbb{F}_p$	
`bvsub`, `bvudiv`, `bvurem`	$\texttt{BV}_b \times \texttt{BV}_b \to \texttt{BV}_b$	
`bvshl`, `bvashr`, `bvlshr`	$\texttt{BV}_b \times \texttt{BV}_b \to \texttt{BV}_b$	
`bvadd`, `bvmul`	$\texttt{BV}_b^* \to \texttt{BV}_b$	
`bvxor`, `bvand`, `bvor`	$\texttt{BV}_b^* \to \texttt{BV}_b$	
`bvneg`, `bvnot`	$\texttt{BV}_b \to \texttt{BV}_b$	
`bv{s,u}{l,g}{e,t}`	$\texttt{BV}_b \times \texttt{BV}_b \to \texttt{BV}_b$	
`(bvextract` h l`)`	$\texttt{BV}_b \to \texttt{BV}_{h-l+1}$	
`(bv{s,u}ext` i`)`	$\texttt{BV}_b \to \texttt{BV}_{b+i}$	
`bvconcat`	$\texttt{BV}_{b_1} \times \cdots \times \texttt{BV}_{b_k} \to \texttt{BV}_{\sum_i b_i}$	
`bool2bv`	$\texttt{Bool} \to \texttt{BV}_1$	
`(bvbit` i`)`	$\texttt{BV}_b \to \texttt{Bool}$	i^{th} bit, $i < b$
`ff2bv`	$\mathbb{F}_p \to \texttt{BV}_b$	zero on input $\geq 2^b$
`bv2ff`	$\mathbb{F}_b \to \texttt{BV}_p$	$2^b \leq p$
$=$	$\texttt{?} \times \texttt{?} \to \texttt{Bool}$	
ite	$\texttt{Bool} \times \texttt{?} \times \texttt{?} \to \texttt{?}$	

Table 14: CirC-IR operators

That is exactly what we sought to show. Thus, our compiler is demonstrably complete.

C CirC-IR

In Table 14, we list all CirC-IR operators concerning Booleans, bit-vectors, or prime-field elements. Most operators are from SMT-LIB and have the same semantics. For prime-field operators $+$, $\times$ and negation, we follow the semantics of prior work [47]. For operators that are not in SMT-LIB nor in prior work, we give any semantic notes under "Semantics".

One unusual operator is `ff2bv` which maps a prime field element $f \in \mathbb{F}_p$ to a bit-vector v of length b. When the unsigned integer that represents f is less than 2^b, v has the same unsigned value. Otherwise, v is zero. The field-blaster for CirC is intensionally incomplete for `ff2bv`, when the input is $\geq 2^b$. It simply bit-splits a `uint` encoding of f into b bits. Incompleteness is acceptable because `ff2bv` is only included in predicates by internal compiler components that ensure the input is in-bounds. This, `ff2bv`'s use in predicates is quite restricted. However, as discussed in the main text (Sec. 6.1), `ff2bv` is widely used in variable definitions.

D Optimizations to the CirC Field-Blaster

D.1 Policy

Our calculus is non-deterministic because in some situations, multiple transitions are applicable. For instance, a conversion can apply at virtually any time. Some policy is needed to decide which transitions to apply.

Our implementation's policy follows three rules. First, prioritize operator rules for un-encoded terms. Second, if only one rule applies to some term's operator, then perform the necessary conversions to meet the constraints on that operator's input encoding types. Third, if multiple rules might apply to a term's operator, call the choose function.

We implement a function $\mathsf{choose}(t, E) \to \mathsf{id}$ that takes the current encoding store and the term to encode, and returns an identifier for which operator rule to apply. choose only needs to handle terms that can match multiple operator rules. In our field-blaster, the only ambiguity is between different rules for encoding bit-vectors extensions. There is one rule when the input is bit-wise encoded and another when the input's unsigned value is encoded. For brevity, Figure 6 shows pseudocode for both rules in a single function: bvZeroExt.

D.2 Equality Assertions

As an optimization, our implemented field-blaster deviates slightly from our calculus as described in Section 4.3. Essentially, it "pushes down" equality assertions. That is, for an input predicate $\phi = \bigwedge_i (t_i = t'_i) \wedge \bigwedge_i b_i$, the field-blaster *does not* encode ϕ. Instead, the field-blaster encodes all t_i, t'_i, and b_i and then asserts that the encodings for each t_i and t'_i are equal, and that each $b_i = \top$. Since asserting equalities is often cheaper (i.e. uses fewer field multiplications) than encoding their result, this is an optimization.

It is straightforward—but tedious—to modify the calculus and its proof of correctness to show that this optimization yields a correct field blaster. Thus, in the main body of the paper, we do not consider it.

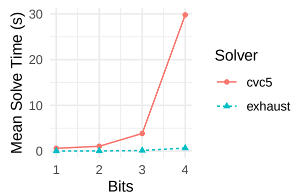
E Verified Field-Blaster Performance Details

In this appendix, we give further details on the performance of the verified field-baster in our case study (Sec. 6).

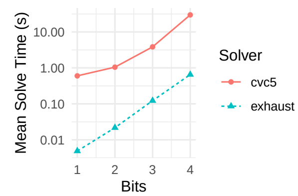
In Figure 15, we show compiler runtime, memory usage, and final constraint count for every function in the Z# standard library, with and without the verified field blaster. We also show the number of encoding rules applied in the verified field-blaster; this is generally similar to the final number of constraints.

	Seconds		Memory (kB)		Constraint		
File	Ver.	Unver.	Ver.	Unver.	Ver.	Unver.	Rules
ecc/edwardsAdd.zok	0.00	0.00	7048	6960	17	17	30
ecc/edwardsCompress.zok	0.09	0.09	21604	21636	511	511	772
ecc/edwardsNegate.zok	0.00	0.00	6792	6840	2	2	7
ecc/edwardsOnCurve.zok	0.00	0.00	6848	6884	6	6	16
ecc/edwardsOrderCheck.zok	0.01	0.00	7304	8988	57	57	72
ecc/edwardsScalarMult.zok	0.55	0.62	127064	100748	9206	9206	10496
ecc/proofOfOwnership.zok	0.57	0.49	126428	100832	9392	9392	10419
EMBED.zok	0.00	0.00	6872	7056	1	1	3
field.zok	0.00	0.00	6760	6940	1	1	3
hashes/mimc7/mimc7R20.zok	0.01	0.01	7840	8052	80	80	124
hashes/mimcSponge/mimcFeistel.zok	0.15	0.17	20080	20004	662	662	1325
hashes/pedersen/512bitBool.zok	0.54	1.14	48664	48672	3573	3573	8617
hashes/pedersen/512bit.zok	0.59	0.55	52540	52744	4083	4083	8424
hashes/sha256/1024bitPadded.zok	3.96	4.59	750852	841956	84727	84727	49749
hashes/sha256/1024bit.zok	2.55	2.99	546200	550216	57382	57382	33242
hashes/sha256/1536bit.zok	3.93	4.78	772816	863696	86253	86253	49953
hashes/sha256/256bitPadded.zok	1.20	1.36	276076	273748	27645	27645	16442
hashes/sha256/512bitPacked.zok	2.65	3.03	551276	541852	55897	55897	34692
hashes/sha256/512bitPadded.zok	2.45	2.85	550252	542844	55856	55856	33036
hashes/sha256/512bit.zok	1.25	1.50	281908	277880	28505	28505	16523
hashes/sha256/embed/IVconstants.zok	0.01	0.02	9068	8804	256	256	514
hashes/sha256/shaRound.zok	1.21	1.55	283432	281228	29057	29057	16879
hashes/utils/256bitsDirectionHelper.zok	0.00	0.00	7456	7188	16	16	65
utils/casts/bool_128_to_u32_4.zok	0.03	0.04	21616	21612	4	4	272
utils/casts/bool_256_to_u32_8.zok	0.05	0.07	21620	21632	8	8	544
utils/casts/field_to_u16.zok	0.03	0.02	21604	21604	17	17	38
utils/casts/field_to_u32.zok	0.02	0.03	21604	21548	33	33	70
utils/casts/field_to_u64.zok	0.03	0.03	21604	21628	66	66	134
utils/casts/field_to_u8.zok	0.02	0.02	21600	21600	9	9	22
utils/casts/u16_from_bits.zok	0.00	0.00	7144	6824	1	1	4
utils/casts/u16_to_bits.zok	0.00	0.00	7144	7056	17	17	50
utils/casts/u16_to_field.zok	0.00	0.00	7112	7108	1	1	3
utils/casts/u16_to_u32.zok	0.00	0.00	8720	6632	1	1	4
utils/casts/u16_to_u64.zok	0.00	0.00	8356	6796	1	1	4
utils/casts/u32_4_to_bool_128.zok	0.06	0.05	21572	21572	132	132	392
utils/casts/u32_8_to_bool_256.zok	0.06	0.08	21620	21620	264	264	784
utils/casts/u32_from_bits.zok	0.01	0.00	6988	6960	1	1	4
utils/casts/u32_to_bits.zok	0.00	0.00	7580	6956	33	33	98
utils/casts/u32_to_field.zok	0.00	0.00	6952	6776	1	1	3
utils/casts/u32_to_u64.zok	0.00	0.00	7184	6976	1	1	4
utils/casts/u64_from_bits.zok	0.00	0.00	7152	7208	1	1	4
utils/casts/u64_to_bits.zok	0.01	0.00	8160	7704	65	65	194
utils/casts/u64_to_field.zok	0.00	0.00	6996	8680	1	1	3
utils/casts/u8_from_bits.zok	0.00	0.00	6584	6876	1	1	4
utils/casts/u8_to_bits.zok	0.00	0.00	7080	7064	8	8	17
utils/casts/u8_to_field.zok	0.00	0.00	8796	6932	1	1	3
utils/casts/u8_to_u16.zok	0.00	0.00	6984	8708	1	1	4
utils/casts/u8_to_u32.zok	0.00	0.00	7064	6700	1	1	4
utils/casts/u8_to_u64.zok	0.00	0.00	6796	6704	1	1	4
utils/multiplexer/lookup1bit.zok	0.00	0.00	6708	8416	1	1	6
utils/multiplexer/lookup2bit.zok	0.00	0.00	7044	7080	3	3	20
utils/multiplexer/lookup3bitSigned.zok	0.00	0.00	9100	6788	4	4	26
utils/pack/bool/nonStrictUnpack256.zok	0.06	0.06	21612	21552	257	257	769
utils/pack/bool/pack128.zok	0.06	0.03	21544	21632	1	1	388
utils/pack/bool/pack256.zok	0.06	0.07	21612	21556	1	1	772
utils/pack/bool/unpack128.zok	0.06	0.06	21604	21660	129	129	386
utils/pack/u32/nonStrictUnpack256.zok	0.08	0.11	21616	21604	255	255	543
utils/pack/u32/pack128.zok	0.12	0.10	21620	21628	129	129	396
utils/pack/u32/pack256.zok	0.06	0.07	21640	21604	257	257	788
utils/pack/u32/unpack128.zok	0.10	0.10	21632	21628	129	129	274

Fig. 15: Compile time, memory usage, and constraint count for all standard library functions.



(a) Linear time scaling.



(b) Logarithmic time scaling.

Fig. 16: How average VC verification time depends on bit-width b. Includes only VCs that are verified by the pertinent solver at all bit-widths.

The number of constraints is always exactly equal, but we see a slight improvement in compiler runtime and memory usage. The difference appears to be orthogonal to the verifiability of our field-blaster. The original field blaster emitted many constraints of form $x \times x = x$, while the new version emits $x \times (x - 1) = 0$. These are mathematically equivalent, but the latter has one fewer variable. This appears to slightly improve performance during R1CS optimization (a downstream compiler pass that is out of scope).

F Verifier Performance Details

In this appendix, we give further details on the performance of cvc5 and our exhaustive verifier in our case study (Sec. 6).

In Figure 16a we show how VC verification time scales with the bit-width b of the rule that the VC models. Each data point is the average solve time at a particular bit-width over all VC operator rules that the solver verifiers at bit-width 4. We see that cvc5's run time grows substantially with the bit-size. The exhaustive solver is far faster on average (though it verifies only easy completeness VCs). Figure 16b shows the same data with a logarithmic time scale. Both solvers appear to take time exponential in the bit-width.

The dependence on the arity a of the operator that the VC is written for. Figure 17a shows how VC verification time grows with arity. As before, we include only VCs that the solver verifies at arity 4. Note that this excludes binary operators and ternary operators like if-then-else. The trend is similar to that observed with bit-width: solver runtime grows quickly with the arity. Figure 17b shows the same data with a logarithmic scale.

G Bugs found in the CirC field blaster

In this appendix, we explain the four bugs we found in CirC's field-blaster. We found these bugs by observing VC violations while porting operator rules from CirC's field-blaster to our verifiable field blaster.

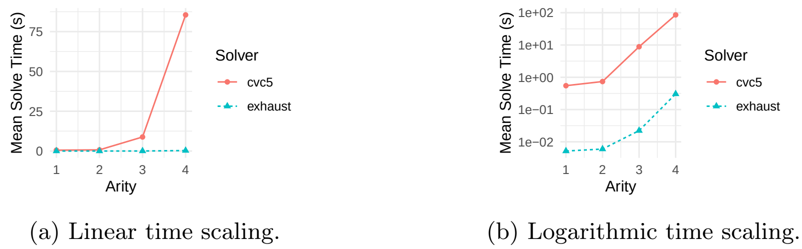


Fig. 17: How average VC verification time depends on operator arity a. Includes only VCs that are verified by the pertinent solver at arity 4.

Incompleteness: finite field division. CirC-IR includes a finite field division operator: $z \leftarrow x/y$. In the IR evaluator, if y is zero, then z is set to be zero. However, the field-blaster emits the equation $zy = x$; this is unsatisfiable for $y = 0, x \neq 0$. This is an incompleteness bug: it makes the whole output predicate unsatisfiable. The completeness VC for the finite-field division rules catches this bug. Our fix is to emit the equation $zyy = xy$.

Incompleteness: bit-shifts. CirC-IR includes bit-vector shifts (left shift, arithmetic right shift, and logical right shift). Following SMT-LIB, overshifting should *saturate*: i.e., shifting a length-b bit-vector by $\geq b$ bits should be equivalent to shifting by $b-1$ bits. However, the field-blaster required the shift amount to be less than b. This is an incompleteness bug, and it is caught by the completeness VCs for the shift rules.

Non-determinism: unsigned bit-vector division. CirC-IR includes unsigned bit-vector division: $z \leftarrow x/y$. Following SMT-LIB, when $y = 0$, the value of z should be $2^b - 1$. However, when $y = 0$, the field-blaster emits equations that allow z to have any value in $\{0, \ldots, 2^b - 1\}$. The soundness VC for the unsigned bit-vector division rule catches this bug.

Unsoundness: bit-vector comparisons. CirC-IR includes operators for signed and unsigned bit-vector comparisons: $z \leftarrow x \bowtie y$. For example, signed $\geq$ and unsigned $<$. For all these operators, the field-blaster uses a utility that computes $\Delta = x - y$ (the difference of signed values) and tested whether $\Delta \geq 0$. This test reduces to testing whether Δ "fit in b unsigned bits". To test this, the field-blaster emits bit-constrained fresh variables $\Delta_0, \ldots, \Delta_{b-1}$ and tests $\Delta = \sum_i 2^i \Delta_i$. This approach is unsound: while the Δ_i variables are *supposed* to be set to the bits of Δ (if it is non-negative), this is not ensured. By setting the bits to the wrong decomposition, the equality doesn't hold, even if $0 \leq \Delta < 2^b$. In the context of a ZKP, this allows a malicious prover to equivocate about whether $x \geq y$. We first caught this bug with the soundness VC for the operator rule for signed bit-vector $\geq$. However, the subroutine "fits in bits" is also used in all bit-vector comparisons, as well as division and remainder.

fn $\mathsf{bvUgeBuggy}(t_x, t_y, x : \mathsf{Enc}, y : \mathsf{Enc})$:
  $\mathsf{kind}(x) = \texttt{uint} \land \mathsf{kind}(y) = \texttt{uint}$
  $b \leftarrow \mathsf{size}(\mathsf{sort}(t_x))$
  $\Delta \leftarrow x - y$
  $\Delta' \leftarrow \mathsf{terms}(x) - \mathsf{terms}(y)$
  for i in $[0, b-1]$:
    $\Delta'_i \leftarrow \mathsf{fresh}(i, ite(\Delta[i], 1, 0), \bot)$
    $\mathsf{assert}(\Delta'_i(\Delta'_i - 1) \approx 0)$
  return $\mathsf{ffEq}(\Delta', \sum_{i=0}^{b-1} 2^i \Delta'_i)$

fn $\mathsf{bvUgeCorrect}(t_x, t_y, x : \mathsf{Enc}, y : \mathsf{Enc})$:
  $\mathsf{kind}(x) = \texttt{uint} \land \mathsf{kind}(y) = \texttt{uint}$
  $b \leftarrow \mathsf{size}(\mathsf{sort}(t_x))$
  $\Delta \leftarrow bvUext(x, 1) - bvUext(y, 1)$
  for i in $[0, b]$:
    $\Delta'_i \leftarrow \mathsf{fresh}(i, ite(\Delta[i], 1, 0), \bot)$
    $\mathsf{assert}(\Delta'_i(\Delta'_i - 1) \approx 0)$
  $\mathsf{assert}(\Delta' \approx -2^b \Delta'_b + \sum_{i=0}^{b-1} 2^i \Delta'_i)$
  return $(\texttt{bit}, 1 - \Delta'_b)$

Fig. 18: Pseudocode for buggy and correct bit-vector unsigned $\geq$. All subroutines have been inlined, except ffEq which produces a boolean `bit` encoding of whether the inputs are equal, as field elements.

To fix the bug, observe that Δ *always* fits in $b+1$ *signed* bits. After *enforcing* (not testing!) the signed bit decomposition, the sign bit indicates whether $\Delta \geq 0$. In Figure 18 we show the buggy rule and the correct rule.

Significance. In most applications of ZKPs, safety properties (e.g., the solvency of an exchange) depend on soundness, while liveness properties (e.g., whether a specific transaction completes) depend on completeness. Thus, the soundness bugs are more serious. The bug in unsigned division bug only affects predicates that assume SMT-LIB semantics for division by zero. The comparison bug is much more serious: it affects any predicate that compares or divides bit-vectors.