Formal Verification of Zero-Knowledge Circuits

Alessandro Coglio Eric McCarthy Eric W. Smith

Kestrel Institute `https://kestrel.edu`
Aleo Systems Inc. `https://aleo.org`

Zero-knowledge circuits are sets of equality constraints over arithmetic expressions interpreted in a prime field; they are used to encode computations in cryptographic zero-knowledge proofs. We make the following contributions to the problem of ensuring that a circuit correctly encodes a computation: a formal framework for circuit correctness; an ACL2 library for prime fields; an ACL2 model of the existing R1CS (Rank-1 Constraint Systems) formalism to represent circuits, along with ACL2 and Axe tools to verify circuits of this form; a novel PFCS (Prime Field Constraint Systems) formalism to represent hierarchically structured circuits, along with an ACL2 model of it and ACL2 tools to verify circuits of this form in a compositional and scalable way; verification of circuits, ranging from simple to complex; and discovery of bugs and optimizations in existing zero-knowledge systems.

1 Introduction

In cryptography, a *zero-knowledge proof* is a method by which a *prover* can convince a *verifier* that they know a secret x that satisfies a computable predicate P, without revealing x and without involving third parties [22, 10]. Spurred by recent advances that have greatly improved their efficiency [9, 23, 14, 7, 13], zero-knowledge proofs are finding increasingly wide application, particularly in the blockchain world [20, 32, 8, 19, 12, 1], holding promise to rebalance privacy on the Internet [37, 18].

While most of the technical details of zero-knowledge proofs are irrelevant to this paper, the one crucial fact is that the predicate P must be expressed as a *zero-knowledge circuit*, which can be defined[1] as a set of equality constraints over integer variables where the only operations are addition and multiplication modulo a large prime number. This is a low-level representation P_{L} of P, at odds with the need for P to be clearly understood by both prover and verifier, who we presume would understand a higher-level representation P_{H} of P, e.g. expressed in a conventional programming language. Unless P_{L} and P_{H} denote the same P, the zero-knowledge proof may not quite prove what is expected.

This leads to the mathematically well-defined problem of formally proving that a zero-knowledge circuit correctly represents a higher-level description. Note the difference between formal proofs, which provide logic-based unconditional evidence of mathematical assertions, and zero-knowledge proofs, which provide cryptography-based statistically overwhelming evidence of computational assertions. Besides formal proofs about zero-knowledge proofs, which is the topic of this paper, one could imagine doing zero-knowledge proofs of formal proofs (i.e. prove a theorem without revealing the proof, which may have interesting applications), but we have not explored that yet. Given the above characterization of the problem in terms of zero-knowledge circuits, the zero-knowledge proof aspect is largely irrelevant here; the unqualified ‘proof’ and similar words in the rest of this paper have the familiar meaning.

This paper describes our endeavors, in the course of various projects, to tackle the zero-knowledge circuit verification problem, using ACL2 [26] and tools built on it. Our contributions are:

(a) A general formal framework for zero-knowledge circuit correctness, i.e. that $P_{\mathrm{L}} \Longleftrightarrow P_{\mathrm{H}}$.
(b) A library of rules to reason about *prime fields*—the arithmetic basis for zero-knowledge circuits.

[1]There seems to be no universal definition of zero-knowledge circuits and of some related notions in the literature.

(c) A formal model of *Rank-1 Constraint Systems* (*R1CS*), an existing formalism commonly used to represent zero-knowledge circuits.
(d) Rules and tools to verify R1CS circuits, including a new specialized version of Axe [38, axe].
(e) A formal model of *Prime Field Constraint Systems* (*PFCS*), a novel formalism developed by us that generalizes R1CS with richer forms of constraints and with hierarchical structure.
(f) Rules and tools to verify PFCS circuits, in a compositional and scalable way.
(g) Verification of zero-knowledge circuits, ranging from simple to complex, in R1CS and PFCS form.
(h) Discovery of two bugs and several optimizations in a zero-knowledge circuit construction library.

Section 2 provides the necessary background on zero-knowledge circuits. Section 3 describes contribution (a). Section 4 describes contribution (b). Section 5 describes contributions (c), (d), (g), and (h). Section 6 describes contributions (e), (f), and (g). Related work is discussed in Section 7. Future work is outlined in Section 8. Some conclusions are drawn in Section 9.

2 Background

A *prime field* is a set $\mathbb{F}_p = \{0,\ldots,p-1\}$, consisting of the natural numbers below p, where p is a prime number. The *arithmetic operations* on $\mathbb{F}_p$ are:

addition:	$x \oplus_p y = (x+y) \bmod p$
subtraction:	$x \ominus_p y = (x-y) \bmod p$
multiplication:	$x \otimes_p y = (x \times y) \bmod p$
division:	$x \oslash_p y = z$, where $x = y \otimes_p z$, if $y \neq 0$

That is, all the operations are modular versions of the ones on the integers, except that the division of x by y yields the unique z (which always exists) that yields x when multiplied by y, provided that $y \neq 0$. We may denote the prime field arithmetic operations with the same symbols as the integer arithmetic operations, i.e. $+$, $-$, $\times$, $/$. We may also omit the multiplication symbol altogether, e.g. $(x+1)(y-1)$ may stand for $(x+1)\times(y-1)$, which in turn may stand for $(x \oplus_p 1) \otimes_p (y \ominus_p 1)$. We may also just write $\mathbb{F}$, leaving p implicit. Context should always disambiguate these commonly used abbreviations.

A *zero-knowledge circuit* is a set of *constraints* that are equalities between *expressions* built out of *variables*, *constants*, *additions*, and *multiplications*, all interpreted in $\mathbb{F}$. By designating certain variables as *inputs* and *outputs*, the constraints can represent a computation of outputs from inputs. Zero-knowledge circuits generalize *arithmetic circuits*, which are like boolean circuits, except that wires carry integers instead of booleans, and gates perform arithmetic operations instead of boolean ones.

For reasons that depend on the details of zero-knowledge proofs, such constraints must be written in specific forms [15]. A popular formalism is *Rank-1 Constraint Systems* (*R1CS*), whose constraints have the form $(a_0 + a_1x_1 + \cdots + a_nx_n)\ (b_0 + b_1y_1 + \cdots + b_my_m) = (c_0 + c_1z_1 + \cdots + c_lz_l)$, where $n,m,l \geq 0$, each a_i, b_j, c_k is a *coefficient* in $\mathbb{F}$, and each x_i, y_j, z_k is a *variable* ranging over $\mathbb{F}$. That is, an R1CS constraint is an equality between the product of two polynomials and a polynomial, each polynomial having zero or more variables with exponent 1, i.e. a *linear combination*. An R1CS circuit is a set of these constraints. Literature definitions of R1CS are usually in terms of vectors and matrices; the definition just given here is more like an abstract syntax of R1CS.

For example, if w is *boolean*, i.e. either 1 or 0, the circuit in the left part of Figure 1 represents the computation in the right part, which sets z to x or y based on whether w is 1 or 0. If $w = 0$, the left side of the constraint is 0 and thus $z = y$; if

$$(w)\ (x-y)\ =\ (z-y) \qquad\qquad z := \begin{cases} x & \text{if } w = 1 \\ y & \text{if } w = 0 \end{cases}$$

Figure 1: An ‘if-then-else’ conditional.

$w = 1$, the $-y$ cancels and thus $z = x$.

As another example, the circuit in the left part of Figure 2 represents the computation in the right part, which sets w to 1 or 0 based on whether $u = v$ or not. If $u = v$, the first constraint makes $w = 1$, and the second constraint is satisfied because $0 \times 1 = 0$. If $u \neq v$, the second constraint makes $w = 0$, and the first constraint is satisfied by $s = 1/(u-v)$.

$$\begin{array}{lcl} (u-v)\ (s) & = & (1-w) \\ (u-v)\ (w) & = & (0) \end{array} \qquad w := \begin{cases} 1 & \text{if } u = v \\ 0 & \text{if } u \neq v \end{cases}$$

Figure 2: An equality test.

These and other examples can be found in the literature, e.g. [30] and [24, Appendix A]. Circuits vary in size and complexity. Even the ones in Figure 1 and Figure 2 require a little thought to understand.

Larger circuits are built from smaller ones by joining their constraints and sharing some variables. For instance, combining Figure 1 and Figure 2 yields a circuit that represents the computation that sets z to x or y based on whether $u = v$ or not; the variable w is shared, with Figure 2 guaranteeing that it is boolean as assumed in Figure 1. In this kind of hierarchical construction, a *gadget* is a circuit with a well-defined purpose, usable as a component of larger gadgets, and possibly made of smaller gadgets. The zero-knowledge circuit P_L in Section 1 is a top-level gadget; it represents the computation, described in some high-level way P_H, of the predicate P on the secret input x.

While all the variables in the gadget in Figure 1 are involved in the represented computation, the variable s in the gadget in Figure 2 is not. It is an *internal* variable, while the other ones are *external* variables; the latter are divided into *input* and *output* variables according to the represented computation. When the two gadgets are combined as just described, the shared external variable w becomes internal to the combined gadget. The distinction between external and internal variables, and between input and output variables, is not captured in the R1CS formalism, but it is arguably implicit in the notion of gadget. In general, internal variables cannot be avoided in gadgets; attempts to eliminate them often result in subtly non-equivalent constraints that fail to adequately represent the intended computation.

Although direct support is limited to $\mathbb{F}$ as a data type and (field) addition and multiplication as operations, R1CS circuits are at least as expressive as boolean circuits: if x and y are boolean variables (like w earlier), the constraint $(x)(y) = (1-z)$ represents a 'nand' gate with output z (and similarly simple constraints represent other logical gates); and higher-level data types can be always encoded as bits. But more efficient representations (fewer variable and constraints) are often possible.

For example, two unsigned n-bit integers, encoded as the bits $x_0, \ldots, x_{n-1}$ and $y_0, \ldots, y_{n-1}$ in little endian order, can be added via the gadget in Figure 3. The first $n+1$ constraints force $z_0, \ldots, z_n$ to be boolean. The last constraint forces them to be the bits of the sum, in little endian order, where z_n is the carry. This assumes that the prime p has at least $n+2$ bits, so that the field operations do not wrap around p; a typical p has about 250 bits, sufficient for fairly large integers.

$$\begin{array}{rcl} (z_0)\ (1-z_0) & = & (0) \\ & \vdots & \\ (z_n)\ (1-z_n) & = & (0) \\ (\sum_{i=0}^{n} 2^i z_i)\ (1) & = & (\sum_{i=0}^{n-1} 2^i x_i + \sum_{i=0}^{n-1} 2^i y_i) \end{array}$$

Figure 3: An unsigned n-bit integer addition.

R1CS circuits are normally constructed programmatically using libraries [3, 4, 6, 27, 28] that provide facilities to build gadgets hierarchically. These libraries are invoked directly, by programs written to build specific circuits, or indirectly, by compilers of higher-level languages to R1CS [2, 5, 25, 31, 11, 29]. As these libraries are invoked, they generate growing sequences of the R1CS constraints that form the gadgets.[2] Separate instances of the same gadget have different variables, which are typically generated

[2]The final sequence consists of the constraints for the predicate P in Section 1, and is part of the zero-knowledge proof.

via monotonically increasing indices. The gadgets' hierarchical structure, reflected in both the static organization and the dynamic execution of the libraries, is lost in the generated flat sequence of constraints; this is not an issue for zero-knowledge proofs, but it can be for formal proofs, as elaborated later.

3 Formal Framework

An R1CS circuit, along with an ordering of the r variables that occur in it, determines a relation $R \subseteq \mathbb{F}^r$ consisting of the r-tuples that satisfy all the constraints, when assigned element-wise to the variables. If additionally the variables are partitioned into q external ones and $r-q$ internal ones (in the sense of Section 2), and ordered so that the former precede the latter, a relation $\widetilde{R} \subseteq \mathbb{F}^q$ is also determined, defined as $\widetilde{R} = \{\langle \phi_1, \ldots, \phi_q \rangle \mid \exists \langle \phi'_1, \ldots, \phi'_{r-q} \rangle . R(\phi_1, \ldots, \phi_q, \phi'_1, \ldots, \phi'_{r-q})\}$, i.e. consisting of the q-tuples that satisfy all the constraints, when assigned element-wise to the external variables, for some $(r-q)$-tuples assigned element-wise to the internal variables. The tuples are *solutions* of the constraints. The informal notion of *gadget* described in Section 2 can be more precisely defined as an R1CS circuit accompanied by an ordering and designation of its variables as just described; $\widetilde{R}$ is the semantics of the gadget. For example, for the gadget in Figure 2, $R = \{\langle u, v, w, s \rangle \mid (u-v)\,s = 1-w \wedge (u-v)\,w = 0\}$ and $\widetilde{R} = \{\langle u, v, w \rangle \mid \exists s. R(u, v, w, s)\}$.

Given this semantic characterization, it is natural to use a relation $S \subseteq \mathbb{F}^q$ as *specification* of the gadget, and to express *correctness* of the gadget as $\widetilde{R} = S$. The specification S may be defined in whichever high-level way that is convenient (more on this later), but in any case it denotes the set of q-tuples that must be the solutions of the gadget. Correctness consists of *soundness* $\widetilde{R} \subseteq S$ (i.e. every solution of the gadget satisfies the specification) and *completeness* $S \subseteq \widetilde{R}$ (i.e. everything satisfying the specification is a solution of the gadget). To prove soundness and completeness, the definition of $\widetilde{R}$ must be expanded, to expose the constraints that define R. To prove soundness, the existential quantification over the antecedent can be turned into a universal quantification over the implication, leading to the quantifier-free formula $R(\phi_1, \ldots, \phi_q, \phi'_1, \ldots, \phi'_{r-q}) \Longrightarrow S(\phi_1, \ldots, \phi_q)$. To prove completeness, no such move is possible: the formula $S(\phi_1, \ldots, \phi_q) \Longrightarrow \exists \langle \phi'_1, \ldots, \phi'_{r-q} \rangle . R(\phi_1, \ldots, \phi_q, \phi'_1, \ldots, \phi'_{r-q})$ demands dealing with the existential quantification explicitly, typically by exhibiting witnesses for the internal variables $\phi'_1, \ldots, \phi'_{r-q}$.

When a gadget represents a computation (as in Section 2), the specification S must specify the computation. For this, a *computation* is modeled as a function $f : I_1 \times \cdots \times I_n \to (O_1 \times \cdots \times O_m) \cup \{\mathcal{E}\}$ from $n \geq 0$ inputs to $m \geq 0$ outputs or to a distinct error $\mathcal{E}$. The case $n = 0$ is uninteresting but unproblematic. The case $m = 0$ models assertion-like computations, e.g. for each gadget $(z_i)(1-z_i) = (0)$ that checks whether z_i is a bit, used in Figure 3: the computation represented by the gadget returns the empty tuple of outputs $\langle\rangle$ if z_i is boolean, or $\mathcal{E}$ otherwise. The function f always models a deterministic computation, which is appropriate for zero-knowledge applications.[3] The function f only captures the computation's input/output behavior, not other aspects of its execution; this is consistent with the fact that R1CS constraints only express relations among variables. For example, for the gadget in Figure 2, $I_1 = I_2 = O_1 = \mathbb{F}$, $f(u,v) = 1$ if $u = v$, $f(u,v) = 0$ if $u \neq v$, and thus $f(u,v) \neq \mathcal{E}$ always.

To represent f in R1CS form, its inputs and outputs must be represented as field elements, via injective encoding functions $e^{\mathrm{I}}_i : I_i \to \mathbb{F}^{n_i}$, where e^{I}_i maps each input $x_i \in I_i$ to some number n_i of field elements, and $e^{\mathrm{O}}_j : O_j \to \mathbb{F}^{m_j}$, where e^{O}_j maps each output $y_j \in O_j$ to some number m_j of field elements. This leads to a computation on encoded inputs and outputs $\widehat{f} : \mathbb{F}^N \to \mathbb{F}^M \cup \{\mathcal{E}\}$, with $N = \sum_i n_i$ and $M = \sum_j m_j$, defined as follows: (1) if $f(x_1, \ldots, x_n) = \langle y_1, \ldots, y_m \rangle$ then $\widehat{f}(e^{\mathrm{I}}_1(x_1), \ldots, e^{\mathrm{I}}_n(x_n)) = \langle e^{\mathrm{O}}_1(y_1), \ldots, e^{\mathrm{O}}_m(y_m) \rangle$;

[3] While zero-knowledge proofs themselves involve non-deterministic computations, normally they (probabilistically) prove facts about deterministic computations.

(2) if $f(x_1,\ldots,x_n) = \mathcal{E}$ then $\widehat{f}(e^{\mathrm{I}}_1(x_1),\ldots,e^{\mathrm{I}}_n(x_n)) = \mathcal{E}$; and (3) $\widehat{f}$ returns $\mathcal{E}$ outside the range of the input encodings. For example, for the gadget in Figure 2, $e^{\mathrm{I}}_1 = e^{\mathrm{I}}_2 = e^{\mathrm{O}}_1 = id$ (identity) and $\widehat{f} = f$.

The computation $\widehat{f}$ is represented by a gadget with $q = N+M$ external variables for the inputs and outputs. The specification of $\widehat{f}$ is $S(x_1,\ldots,x_N,y_1,\ldots,y_M) = [\widehat{f}(x_1,\ldots,x_N) = \langle y_1,\ldots,y_M\rangle]$. Thus, soundness means that every solution of the gadget corresponds to a non-erroneous instance of the computation, and completeness means that every non-erroneous instance of the computation corresponds to a solution of the gadget. Soundness alone is not sufficient for correctly representing a computation: a gadget without solutions is trivially sound; completeness ensures that there is a solution for every input for which the computation is not erroneous. For example, for the gadget in Figure 2, $S = \{\langle u,v,w\rangle \mid f(u,v) = w\}$.

For a top-level gadget P_{L} that represents P in a zero-knowledge proof (see Section 1 and Section 2), whose formal semantics is a relation $\widetilde{R}_P$ as above, the specification S_P is derived from the high-level description P_{H} of P. If P_{H} is a program in a higher-level language [2, 5, 25, 31, 11, 29], a function f_P that denotes the execution of the program is formally defined, based on a formalization of the language, and a specification relation S_P is derived from it as above. If P_{H} is a description of a fixed application-specific computation (e.g. Zcash shielded transactions [19]), f_P is defined by formalizing that description, and S_P is derived from it as above. For sub-gadgets of P_{L}, specifications may be written in any formal form that is convenient; these specifications play a role in the formal verification of P_{L} (see Section 5.5 and Section 8), but they are not directly exposed to the zero-knowledge prover and verifier mentioned in Section 1. The understandability of P_{H} by prover and verifier, mentioned in Section 1, as with all complex technologies, boils down to trusting authoritative high-level informal descriptions for users from the general public, analyzing the aforementioned programs for users who are also software developers, and examining the formalizations and theorems for users who are also formal verification specialists.

4 Prime Fields

Starting with the recognizer of prime numbers `primep` from [39, `[books]/projects/numbers`], the *prime fields library* [38, `prime-fields`] introduces a recognizer `fep` of field elements, and functions `add`, `sub`, `mul`, `div`, `neg`, `inv`, `pow`, and `minus1` for field operations. These are all parameterized over a prime `p`, e.g. `(fep x p)` checks if `x` is in $\mathbb{F}_{\mathsf{p}}$, and `(add x y p)` returns $\mathsf{x} \oplus_{\mathsf{p}} \mathsf{y}$ (see Section 2).

The recognizer and operations are executable. The multiplicative inverse `inv` is calculated via `pow`, according to the known equation $1 \oslash_p x = x^{p-2}$ `mod` p; we prove that this definition indeed yields the multiplicative inverse. The definition of `pow` is an `mbe` whose `:logic` is recursively repeated multiplication and whose `:exec` is fast modular exponentiation `mod-expt-fast` from [38, `arithmetic-3`].

The library provides basic theorems, such as all the standard field axioms (e.g. commutativity of addition); these theorems often suffice for relatively simple reasoning. The library also provides collections of rules that realize certain normalization strategies, useful for more elaborate reasoning.

5 Rank-1 Constraint Systems

5.1 Model

Based on the prime fields library described in Section 4, the *R1CS library* [38, `r1cs`] provides an ACL2 model of R1CS. It follows the nomenclature of literature definitions of R1CS, which are in terms of vectors and matrices. The model consists of a *dense* formulation, where linear combinations have monomials for all the involved variables (many with zero coefficients), and a *sparse* formulation, where linear combinations may omit monomials with zero coefficients. The dense formulation is of intellectual interest but impractical for verification; the rest of this paper focuses on the sparse formulation.

The model formalizes a *pseudo-variable* as either a *variable* (an ACL2 symbol) or the number 1. A linear combination is formalized as a (sparse) *vector*, i.e. an ACL2 list of pairs (ACL2 lists of length 2) where each pair consists of a *coefficient* (a field element) and a pseudo-variable; the notion of pseudo-variable provides uniformity between monomials of degrees 1 and 0 in linear combinations. A *constraint* is formalized as an aggregate [38, `defaggregate`] with components `a`, `b`, and `c` that are the three linear combinations; this corresponds to the equality $(\mathtt{a})\,(\mathtt{b}) = (\mathtt{c})$, referring to Section 2. Finally, a (*rank-1 constraint*) *system* is formalized as an aggregate consisting of a prime, a list of variables, and a list of constraints. The model also defines well-formedness conditions on this aggregate and its sub-structures, e.g. that all the variables in the constraints are also in the list of variables of the R1CS aggregate. This aggregate and its sub-structures form the model's formal *syntax* of R1CS.

The *semantics* of R1CS is formalized in terms of satisfaction of constraints by *valuations*, which are ACL2 alists from variables to field elements, i.e. assignments of field elements to variables. Given a valuation, a linear combination evaluates to a field element, in the obvious way; the model defines this evaluation in terms of *dot product* of vectors, as in the literature. Given a valuation, a constraint evaluates to a boolean, in the obvious way. A valuation *satisfies* a system iff it makes all its constraints true.

Besides basic theorems about the syntax and semantics sketched above, the R1CS library also includes rules for reasoning about R1CS, for both ACL2 and Axe (see below).

5.2 Extraction

To verify gadgets using the R1CS model described above, the gadgets must be represented in the syntactic form defined by the model. The gadget construction libraries mentioned in Section 2 produce R1CS constraints that are not in that form, and sometimes they do not provide facilities to export them in any form. Thus, the approach to extract gadgets for verification is case by case.

In a Kestrel project funded by the Ethereum Foundation, we worked on the verification of Ethereum's Semaphore circuit [38, `semaphore`]. Since Semaphore was written in the high-level language Circom [25], whose compiler had a facility to export the R1CS constraints in JSON format, we developed an ACL2 converter from that format [39, `[books]/kestrel/ethereum/semaphore/json-to-r1cs`], and we used that along with our ACL2 JSON parser [39, `[books]/kestrel/json-parser`]. Taking advantage of the modularity of the Circom code, we extracted not only the complete circuit gadget, but also several sub-gadgets.

In a Kestrel project funded by the Tezos Foundation, we worked on the verification of Zcash's Jubjub elliptic curve operation circuits [38, `zcash`]. Since these circuits were generated programmatically in Rust, we instrumented that Rust code, with help from the Zcash team, to export R1CS constraints directly as s-expressions in the R1CS model's format. We extracted the top-level gadgets and several sub-gadgets, by invoking the library at different points.

At Aleo, we are working on the verification of the snarkVM gadgets [17]. With our Aleo colleagues, we have instrumented snarkVM's Rust code to export R1CS constraints in JSON format (different from Circom's). We have developed an ACL2 converter from that format to the model's format, which we are using along with the ACL2 JSON parser. We are extracting gadgets at varying levels of granularity.

In the current situation, in all the above cases, the gadget extraction is trusted: an error in our instrumentation of the gadget construction libraries, or in our conversion to ACL2, may cause us to unwittingly verify a different gadget from the real one. However, the top-level gadget P_{L} (discussed in Section 1, Section 2, and Section 3) is part of the zero-knowledge proof, which has a well-defined (protocol-dependent) format, and is generated by tools like snarkVM [3] that use or include gadget construction libraries, That format can be formalized in ACL2, and the formal verification can be applied directly to (the P_L gadget in) the zero-knowledge proof. In this eventual situation, the extraction of sub-gadgets of P_L via instrumentation and conversion will merely provide building blocks for the top-level formal proof of P_L (especially in the compositional approach in Section 5.5), but will no longer be trusted.

5.3 Verification in ACL2

Regardless of the exact approach, the result of the above extraction is an ACL2 constant, say `*gadget*`, whose value is an R1CS aggregate of the form described in Section 5.1. The model confers semantics to this aggregate, amounting to the relation R in Section 3. More precisely, the model provides a predicate over an assignment of field elements to variables: `(r1cs-holdsp *gadget* asg)` means that the assignment `asg` satisfies all the constraints in `*gadget*`, given the prime that is part of the `*gadget*` aggregate (which is left implicit in R). This can be turned into a finitary relation over the field elements assigned to the variables, like R, by specializing `r1cs-holdsp` with an assignment to the gadget's specific variables, e.g. if the variables in `*gadget*` are `'x0`, `'x1`, ..., the relation R is formalized as

```
(defun gadget (x0 x1 ...)
  (r1cs-holdsp *gadget* (list (cons 'x0 x0) (cons 'x1 x1) ...))
```

where each `'xi` is a variable and each `xi` is a field element.

To state and prove correctness, a specification is written in ACL2, amounting to S in Section 3:

```
(defun spec (x0 x1 ...) ...)  ; this can be defined in any form
```

If the gadget has no internal variables, correctness is stated as

```
(defthm gadget-correctness
  (implies (and ...   ; boilerplate hypotheses
                ...)  ; preconditions (if applicable)
           (equal (gadget x0 x1 ...) (spec x0 x1 ...))))  ; R = S
```

where the boilerplate hypotheses say that `x0`, `x1`, ... are field elements, and where examples of preconditions are that some `xi` is boolean or that some `xi` is non-zero. If the gadget has internal variables, `spec` has fewer parameters, but soundness can be stated and proved similarly, using `implies` in place of `equal`. Completeness is less straightforward; it is discussed, in a more general context, in Section 5.5.

The proofs are carried out by first enabling certain functions of the R1CS semantics, so that the (evaluated) constraints *deeply embedded* in ACL2 are rewritten to ACL2 terms involving prime field operations, i.e. constraints *shallowly embedded* in ACL2. Then the core of the proof is handled via other hints and lemmas, of varying complexity, that depend on the details of the constraints and specification.

After verifying, in the manner just described, the correctness of a number of Semaphore sub-gadgets for elliptic curve operations and data multiplexing [39, `[books]/kestrel/ethereum/semaphore`], two related issues became apparent. One issue was that the numbers of variables and constraints and the resulting ACL2 terms grew quickly as we moved from simpler to more complex gadgets, making the proofs harder and less efficient. Another issue was that because each gadget was extracted in isolation, with its own specific variable names generated by the gadget construction libraries (typically via monotonically increasing indices), it was not easy to use proofs of sub-gadgets in proofs of super-gadgets: the same sub-gadget could appear with different variable names in different super-gadgets, or in different instantiations within the same super-gadget, but the proof for the separate sub-gadget used different variable names than would be seen in any of these instantiations.

5.4 Verification in Axe

To combat the growth of terms mentioned in Section 5.3, we turned to the Axe toolkit [38, `axe`]. The *Axe Rewriter* is functionally similar to the ACL2 rewriter, but it represents terms as directed acyclic graphs (DAGs) instead of trees: these DAGs share sub-terms, affording the practical handling of very large terms, such as fully unrolled AES implementations.

We developed a specialization of the *Axe Lifter* for R1CS, which turns deeply embedded constraints into shallowly embedded ones, similarly to what is described in Section 5.3, and also performs some simplifications of the lifted constraints using the Axe Rewriter. This specialized lifter [38, `lift-r1cs`] generates an ACL2 constant whose value is a DAG representing the simplified lifted constraints.

We developed a specialization of the *Axe Prover* for R1CS, which, given a DAG from the lifter as above and a specification like `spec` in Section 5.3, attempts to prove soundness [38, `verify-r1cs`] or completeness (via a more general event macro to prove implications). This specialized prover uses rewriting and variable elimination via substitution, and it supports applying different sets of rewrite rules in sequence. Substitution is enabled by the fact that certain constraints essentially equate certain variables to expressions over other variables, though rewriting must often be performed first to make this explicit by solving the constraints. A constraint is a candidate for substitution if it equates a variable with some sub-DAG not involving that variable. Large R1CS proofs can involve hundreds or thousands of substitution steps, and we optimized Axe to apply many substitutions at once when possible. For each round of substitution, Axe substitutes a set of variables each of which is equated to a sub-DAG not involving any variables in the set. The set of equalities used in the round is then removed from the assumptions of the proof. Repeated substitution of intermediate variables can incrementally turn a large unstructured conjunction of constraints into a deeply nested operator tree (represented in DAG form), of the kind commonly verified by Axe. The ability to apply the rewriting tactic with different sets of rewrite rules supports the staging of inter-dependent proof steps, which depend on previous steps and enable subsequent steps. Suitable rewrite rules can recognize R1CS idioms and turn them into equivalent higher-level formulations that may facilitate the rest of the proof. Similarly, certain sub-gadgets may also be recognized and raised in abstraction using rewrite rules based on the correctness properties of such sub-gadgets; this partially addresses the second issue described in Section 5.3.

The Axe verification of the soundness of an R1CS gadget looks like

```
(lift-r1cs *gadget-dag*  ; name of the generated defconst
           '(x0 x1 ...)  ; variables of the gadget
           ...           ; constraints of the gadget
           ...           ; prime of the gadget
           ...)          ; options
(verify-r1cs *gadget-dag*       ; gadget (simplified and lifted, in DAG form)
             (spec x0 x1 ...)   ; specification
             :tactic ...        ; proof tactics, e.g. (:rep :rewrite :subst)
             ...)               ; other information and options
```

We used this approach to verify the soundness, and in some cases also the completeness, of a number of Semaphore and Zcash (see Section 5.2) sub-gadgets that perform fixed-size integer operations, elliptic curve operations, instances of the MiMC cipher, and parts of the BLAKE2s hash [39, `[books]/kestrel/ethereum/semaphore`] [39, `[books]/kestrel/zcash/gadgets`]; these range from relatively small and simple to relatively large and complex. We also verified the soundness of the large and complex BLAKE2s hash gadget generated by (an earlier version of) snarkVM [3]; this is currently not open-source, but it will be in the future.

While using Axe helps address the term growth problem, the sub-gadget proof re-use problem remained largely unsolved. The recognition and rewriting of sub-gadgets mentioned above, which worked for certain cases, in general may need to recover the sub-gadgets from a sea of constraints. Each sub-gadget may consist of multiple constraints, some of which may even have the same form across different sub-gadgets, requiring the exploration of multiple recovery paths. Furthermore, constraint optimizations, such as the ones performed by the Circom compiler and by snarkVM, which blend gadgets under certain conditions, may greatly complicate, or defeat altogether, the recovery of sub-gadgets. Solving these problems is not necessarily impossible, but it is challenging; as a data point, the aforementioned soundness verification of snarkVM's BLAKE2s took several person-days to develop and takes several machine-hours to run.

5.5 Compositional Verification

As mentioned in Section 2, the hierarchical structure in the gadget construction libraries gets flattened away in the generated R1CS constraints. Thus, as discussed in Sections 5.3 and 5.4, the gadgets extracted from the libraries are verified as wholes, with limited ability to discern their hierarchical structure and leverage proofs of their sub-gadgets, resulting in difficult and slow proofs.

More scalability can be achieved via *compositional verification*, where the proof of a gadget uses the proofs of its sub-gadgets and is used in the proofs of its super-gadgets. This could be accomplished by extending the gadget construction libraries to generate such compositional proofs along with the gadgets, but doing so is impractical due to the libraries' complexity and ownership. A viable approach is to (1) replicate the gadget constructions in the theorem prover, (2) verify correctness properties of the constructions, and (3) validate the replicated gadget constructions by checking that the constructed gadgets are the same as the ones extracted from the libraries. We propound the term *detached proof-generating extension* for this kind of solution.

The gadget constructions are formalized by ACL2 functions that take variable names as inputs and return lists of R1CS constraints as outputs. The constraints are built either directly or by calling functions that build sub-gadgets, concatenating all the resulting constraints together. These functions return lists of constraints, which are readily composable by concatenation; they do not return R1CS aggregates (see Section 5.1), which are not readily composable. The gadget hierarchy corresponds to the function hierarchy. The parameterization over the variable names is critical, because separate instances of the same gadget have different variables, as mentioned in Sections 2 and 5.3.

To *validate* that these constructions are consistent with the libraries, we extract *sample gadgets* from the libraries as in Section 5.2, and we formulate ACL2 ground theorems saying that the extracted R1CS constraints are identical to the ones built by the ACL2 functions when passed suitable variable names as arguments. Currently this validation process amounts to testing our constructions against the libraries. Eventually, this validation will be performed every time the libraries are run to generate a zero-knowledge proof, as explained in Section 8.

The correctness of the ACL2 gadget constructions is proved for generic variable names and generic prime p (sometimes under restricting hypotheses). The proof opens the function definition and uses the theorems for any called functions, whose definitions are unopened; if the function builds some constraints directly, certain semantic functions of the R1CS model are also opened, lifting those constraints to equalities and prime field operations. Given this proof setup, the correctness of the gadget (family) built by the function is proved by reasoning over the specifications of the sub-gadgets (not the sub-gadgets' constraints) and/or the constraints of the gadget; the details depend on the gadget, and may involve hints and lemmas of varying complexity.

For example, a gadget to force a variable to be boolean as in Section 2 is constructed as

```
(defun boolean-assert-gadget (x)
  (list (make-r1cs-constraint :a (list (list 1 x))             ; (x)
                              :b (list (list 1 1) (list -1 x))  ; (1 - x)
                              :c nil)))                          ; (0)
```

where x is the variable name to use. Correctness (soundness and completeness) is expressed as

```
(defthm boolean-assert-gadget-correctness
  (implies ...  ; boilerplate hypotheses
           (equal (r1cs-constraints-holdp (boolean-assert-gadget x) asg p)  ; R
                  (bitp (lookup-equal x asg)))))                             ; S
```

where asg assigns field elements to variables, lookup-equal retrieves them, and bitp is the specification of this gadget; in the notation of Section 3, this theorem rewrites R ($= \widetilde{R}$ in this case) to S.

As another example, the gadget in Figure 2 is constructed as

```
(defun equality-test-gadget (u v w s)
  (append (list (make-r1cs-constraint ...))    ; (u - v) (s) = (1 - w)
          (list (make-r1cs-constraint ...))))  ; (u - v) (w) = (0)
```

Soundness is expressed as

```
(defthm equality-test-gadget-soundness
  (implies (and ...  ; boilerplate hypotheses
                (r1cs-constraints-holdp (equality-test-gadget u v w s) asg p))  ; R
           (equal (lookup-equal w asg)                                          ; S
                  (if (equal (lookup-equal u asg) (lookup-equal v asg)) 1 0))))
```

where the specification of this gadget is that the value of w is 1 or 0 based on whether the values of u and v are equal or not; in the notation of Section 3, this theorem derives S from R ($\neq \widetilde{R}$ in this case).

The gadget described in Section 2 as the combination of Figure 2 and Figure 1 is constructed as

```
(defun if-equal-then-else-gadget (u v x y z w s)
  (append (if-then-else-gadget w x y z)
          (equality-test-gadget u v w s)))
```

which calls the functions for the sub-gadgets (the definition of if-then-else-gadget is not shown).

To exemplify varying numbers of variables and constraints, the gadget in Figure 3 is constructed as

```
(defun addition-gadget (xs ys zs)
  ...  ; guard requires (len xs) = (len ys) = (len zs) - 1
  (append (boolean-assert-list-gadget zs)
          (list (make-r1cs-constraint
                 :a (append (pow2sum-vector xs) (pow2sum-vector ys))
                 :b (list (list 1 1))
                 :c (pow2sum-vector zs)))))
```

where xs, ys, and zs are lists of variables, boolean-assert-list-gadget constructs boolean constraints for all the variables in zs, and pow2sum-vector constructs a powers-of-two weighted sum. The parameterization covers not only the names of the variables, but also the number of bits n in Figure 3, which is the length of xs and ys. Correctness is expressed as

```
(defthm addition-gadget-correctness
  (implies (and ...                                   ; boilerplate hypotheses
                (< (1+ (len xs)) (integer-length p))   ; restriction on n
                (bit-listp (lookup-equal-list xs asg))   ; precondition
```

```
          (bit-listp (lookup-equal-list ys asg)))  ; precondition
     (equal (r1cs-constraints-holdp (addition-gadget xs ys zs) asg p)
            (and (bit-listp (lookup-equal-list zs asg))
                 (equal (lebits=>nat (lookup-equal-list zs asg))
                        (+ (lebits=>nat (lookup-equal-list xs asg))
                           (lebits=>nat (lookup-equal-list ys asg))))))))
```

where `lebits=>nat` turns a list of bits into the integer they denote in little endian order, and where the restriction on n ensures that the modular weighted sums can be turned into non-modular sums. This is proved for every n, using a property of `pow2sum` proved by induction. While the proofs for the previously exemplified gadgets are straightforward, this gadget takes a little more work.

The details of the examples above, and of the other ones in Section 2, are in the supporting materials, in [39, `[books]/workshops/2023/coglio-mccarthy-smith`]. Other examples are in the R1CS library, in [39, `[books]kestrel/crypto/r1cs/sparse/gadgets`], where in particular the proofs in `range-check.lisp` were quite laborious.

We have employed this approach to verify compositionally a substantial portion of the snarkVM gadgets [17], specifically most of the ones for boolean, field, and integer operations. In the process, we have discovered two bugs in the gadgets, which have been fixed:[4] (i) the gadget to convert a field element into its bits failed to constrain the integer value of the bits to be below the prime, leading to indeterminacy (e.g. the field element 0 could be converted to not only all zero bits as expected, but also to the bits that form the prime, since $p \text{ mod } p = 0$); and (ii) the gadget to calculate square root allowed both positive and negative roots (when the input is a non-zero square), leading to indeterminacy. We have also identified some possible optimizations, which have been or are being applied, saving a large number of constraints in some cases. Our ACL2 work on snarkVM is currently not open-source, but it will be in the future.

But even this approach eventually runs into a scalability issue, due to the internal variables of gadgets. The names of these variables are exposed as function parameters of not only the gadgets that directly use them to build constraints, but also any super-gadgets that contain (possibly many instances of) those sub-gadgets. As increasingly large gadgets are constructed, the function parameters for variable names keep growing, including all the internal variables at every level. Furthermore, while soundness theorems like `equality-test-gadget-soundness` above can ignore internal variables in the consequent of the implication, completeness theorems need to say something about the internal variables. In the notation of Section 3, a gadget correctness theorem $\widetilde{R} = S$ reduces to $R = S$ if there are no internal variables, which is a good rewrite rule, as in `boolean-assert-gadget-correctness` above. But internal variables cannot be existentially quantified in the gadget construction functions, because these functions must return the gadgets given all their variables. Instead, the specification S over the external variables must be extended to a specification S' over all variables, including the internal ones at every level. This exposure of internal variables violates modularity and impedes compositionality.

6 Prime Field Constraint Systems

The scaling issue discussed in Section 5.5 is addressed by *Prime Field Constraint Systems* (*PFCS*), a formalism introduced by the authors. PFCS generalizes R1CS in two ways: (1) constraints can be equalities between any expressions, built out of variables, constants, additions, and multiplications; and (2) constraints can be grouped into *named relations with parameters*, and these relations can be used as constraints with the parameters replaced by argument expressions (as in function calls).

[4]The Aleo blockchain mainnet had not been launched yet, so these bugs did not affect real applications and assets.

The first extension is useful to represent zero-knowledge circuit formalisms different from R1CS, but is not especially relevant to verifying R1CS gadgets. The second extension is important for verifying R1CS and other kinds of gadgets, because it explicitly captures their hierarchical structure. A PFCS relation formalizes a *gadget*; the relation's parameters are the gadget's external variables, while the other variables in the relation's defining body are the gadget's internal variables.[5] PFCS explicitly handles the existential quantification that takes R to $\widetilde{R}$: while R is the semantics of a PFCS relation's body, $\widetilde{R}$ is the semantics of the PFCS relation itself. The internal variables of a gadget are taken into consideration when proving the correctness $\widetilde{R} = S$ of a gadget, which involves R, but can be ignored when proving the correctness of super-gadgets that include that sub-gadget, whose semantics $\widetilde{R}$ can be rewritten to S in proofs for the super-gadgets; no extended specification S' (see end of Section 5.5) is needed.

Our development and use of PFCS is still somewhat preliminary. It is overviewed here, but it will be described in more detail in future publications. More information is in the PFCS library [38, `pfcs`].

6.1 Model

The *syntax* of PFCS is approximately described by the grammar in Figure 4, consistently with the informal description above. A relation R consists of a name N, a sequence of parameters N*, and a defining body that is a sequence of constraints C*. The abstract syntax is formalized via recursive types [38, `fty`]. The concrete syntax is formalized via an ABNF grammar [38, `abnf`] complemented by some (upcoming) restricting predicates.

names	N	::=	⟨letter then letters/digits/underscores⟩
integers	I	::=	`...` | `-2` | `-1` | `0` | `+1` | `+2` | `...`
expressions	E	::=	N | I | E`+`E | E`*`E
constraints	C	::=	E`=`E | N`(`E*`)`
relations	R	::=	N`(`N*`){`C*`}`

Figure 4: PFCS syntax.

The *semantics* of PFCS is approximately described by the inference rules in Figure 5, which inductively define when an assignment α (a finite map from variables to field elements) satisfies a constraint c in the context of a set of relations ρ, written $\alpha \vdash_\rho c$. The first rule says that α satisfies an equality constraint $e_1 = e_2$ when the evaluations $\alpha(e_1)$ and $\alpha(e_2)$ yield the same field element; α extends from variables to expressions in the obvious way. The second rule says that α satisfies a relation constraint $r(e_1 \cdots e_n)$ when ρ includes the relation $r(v_1 \cdots v_n)\{c_1 \cdots c_m\}$ and each constraint c_i in its body is satisfied by an assignment α' that extends the assignment of each evaluated argument expression $\alpha(e_j)$ to the corresponding parameter v_j of the relation; besides the parameters, α' must assign field elements to the other variables (if any) in the body of the relation, which are internal to the gadget. Since α' appears in the premises but not in the conclusion, it is existentially quantified; since the values of the relation's parameters are prescribed by the rule, the existential quantification reduces to the values assigned to the internal variables (if any), capturing exactly the existential quantification in $\widetilde{R}$. The prime p is left implicit in Figure 5.

$$\frac{\alpha(e_1) = \alpha(e_2)}{\alpha \vdash_\rho e_1 = e_2}$$

$$\frac{\begin{array}{c} r(v_1 \cdots v_n)\{c_1 \cdots c_m\} \in \rho \\ \alpha' \supseteq \{v_1 \mapsto \alpha(e_1), \ldots, v_n \mapsto \alpha(e_n)\} \\ \forall i \in \{1, \ldots, m\}.\ \alpha' \vdash_\rho c_i \end{array}}{\alpha \vdash_\rho r(e_1 \cdots e_n)}$$

Figure 5: PFCS semantics.

Since ACL2 disallows mutually recursive `defun` and `defun-sk`, the PFCS semantics is formalized, over the PFCS abstract syntax, via (1) proof trees for the inference rules in Figure 5 and (2) a proof

[5] PFCS does not distinguish between input and output external variables. This distinction only matters to the formulation of the specification S, which is still always a relation over the external variables, as is the semantics $\widetilde{R}$ of the gadget.

checker for those proof trees; that is, a mini-logic is formalized in ACL2. Since this definition is inconvenient for reasoning about gadgets, ACL2 rules are provided that capture the inference rules more directly, without proof trees and proof checker, as if `defun` and `defun-sk` were mutually recursive.

6.2 Verification

In the PFCS framework, gadget constructions are formalized by ACL2 functions that take no or few inputs and return (abstract syntax of) PFCS relations as outputs. Gadgets with fixed numbers of variables and constraints are built by ACL2 functions with no inputs. Gadgets with varying numbers of variables or constraints are built by ACL2 functions whose inputs are non-negative integers that specify those varying numbers. None of these ACL2 functions take variable names as inputs, because variables in PFCS relations are local to the relations and can be fixed for each gadget: the external variables, i.e. the parameters, can be replaced when the relations are called; and the internal variables are existentially quantified. These ACL2 functions do not call each other, unlike the ones that construct R1CS gadgets; the gadget hierarchy is captured directly in the PFCS relations.

Correctness is proved for generic prime p and (if applicable) for generic numbers of variables and constraints (sometimes under restricting hypotheses). The *deeply embedded* PFCS relations built by the ACL2 functions are lifted to *shallowly embedded* PFCS relations, which are ACL2 predicates over field elements, with parameters for the external variables and an existential quantification (via `defun-sk`) for the internal variables. These predicates are defined as conjunctions of (1) calls of other predicates, one per sub-gadget, and (2) equalities between terms involving prime field operations, one per equality constraint; the predicates' call graph corresponds to the gadget hierarchy. For gadgets with fixed numbers of variables and constraints, a deep-to-shallow *lifter* automatically generates the predicates, along with theorems connecting the deep and shallow formulations; for gadgets with varying numbers of variables and constraints, currently the predicate and theorem are manually generated, but a future extension of the lifter may automate these as well. Correctness of a shallowly embedded PFCS relation is proved by opening the predicate definition, using the called predicates' correctness theorems as rewrite rules, and using other hints and lemmas of varying complexity as needed. Correctness is extended to the deeply embedded PFCS relation via the lifting theorem, in a way that may be automated in the future.

For example, a PFCS version of `boolean-assert-gadget` in Section 5.5 is constructed as

```
(defun boolean-assert-gadget ()  ; deeply embedded PFCS relation
  (pfdef "boolean_assert"        ; name
         (list "x")              ; parameter
         (pf= (pf* (pfvar "x")                          ; (x)
                   (pf+ (pfconst 1) (pfmon -1 "x")))    ; (1 - x)
              (pfconst 0))))                            ; (0)
```

The lifter call `(lift (boolean-assert-gadget))` generates the predicate

```
(defun boolean-assert (x p)  ; shallowly embedded PFCS relation
  (and (equal (mul x (add (mod 1 p) (mul (mod -1 p) x p) p) p)
              (mod 0 p))))
```

and the lifting theorem

```
(defruled definition-satp-of-boolean-assert-to-shallow
  (implies ... ; boilerplate hypotheses
           (equal (definition-satp "boolean_assert" defs (list x) p)  ; deep
                  (boolean-assert x p))))                             ; shallow
```

where `(definition-satp` r ρ `(list` ϕ_1 $\cdots$ ϕ_n`)` p`)` formalizes $\{v_1 \mapsto \phi_1, \ldots, v_n \mapsto \phi_n\} \vdash_\rho r(v_1 \cdots v_n)$. The correctness of the predicate is expressed as

```
(defthm boolean-assert-correctness
  (implies ...  ; boilerplate hypotheses
           (equal (boolean-assert x p)  ; R (shallow)
                  (bitp x))))           ; S
```

which is extended to the gadget via the lifting theorem as

```
(defthm boolean-assert-gadget-correctness
  (implies ...  ; boilerplate hypotheses
           (equal (definition-satp "boolean_assert" defs (list x) p)  ; R (deep)
                  (bitp x))))                                         ; S
```

As another example, a PFCS version of `if-equal-then-else-gadget` in Section 5.5 is built as

```
(defun if-equal-then-else-gadget ()
  (pfdef "if_equal_then_else"
         (list "u" "v" "x" "y" "z")
         (pfcall "if_then_else" (pfvar "w") (pfvar "x") (pfvar "y") (pfvar "z"))
         (pfcall "equality_test" (pfvar "u") (pfvar "v") (pfvar "w"))))
```

The lifter generates the predicate

```
(defun-sk if-equal-then-else (u v x y z p)
  (exists (w)
          (and (fep w p)
               (and (if-then-else w x y z p)
                    (equality-test u v w p)))))
```

which existentially quantifies `w` and which calls the lifted predicates for its sub-gadgets (not shown here). Correctness is expressed as

```
(defthm if-equal-then-else-gadget-correctness
  (implies ...  ; boilerplate hypotheses
           (equal (definition-satp "if_equal_then_else" defs (list u v x y z) p)  ; R
                  (equal z (if (equal u v) x y)))))                               ; S
```

which rewrites R to S without involving the internal variable w.

To exemplify varying numbers of variables and constraints, a PFCS version of `boolean-assert-list-gadget` mentioned (but not shown) in Section 5.5 is constructed as

```
(defun boolean-assert-list-gadget (n)
  (pfdef (iname "boolean_assert_list" n)  ; "boolean_assert_list_<n>"
         (iname-list "x" n)               ; (list "x_0" "x_1" ...)
         (boolean-assert-list-gadget-aux  ; ((pfcall "boolean_assert" (pfvar "x_0"))
          (iname-list "x" n)))))          ;  (pfcall "boolean_assert" (pfvar "x_1"))
                                          ;  ...)
(defun boolean-assert-list-gadget-aux (vars)
  (cond ((endp vars) nil)
        (t (cons (pfcall "boolean_assert" (pfvar (car vars)))
                 (boolean-assert-list-gadget-aux (cdr vars))))))
```

where `iname` constructs an indexed name, `iname-list` constructs a list of indexed names, and the auxiliary function constructs a list of PFCS relations calls for generic variable names (which is useful for induction), which the main function instantiates to specific variable names. Correctness is expressed as

```
(defthm boolean-assert-list-gadget-correctness
  (implies ...  ; boilerplate hypotheses
           (equal (definition-satp "boolean_assert_list" defs xs p)  ; R
                  (bit-listp xs))))                                  ; S
```

The details of the examples above, and of the other ones in Section 2, are in the supporting materials, in [39, [books]/workshops/2023/coglio-mccarthy-smith]. Other examples are in the PFCS library, in [39, [books]kestrel/crypto/pfcs/examples.lisp].

We are porting the verified snarkVM gadgets mentioned in Section 5.5 from R1CS form to PFCS form, which we will also use for the remaining snarkVM gadgets.

6.3 Validation

The PFCS gadget constructions in ACL2 are built in the same way as the R1CS gadget constructions in Section 5.5, namely by replicating what the gadget construction libraries do. For the ACL2 R1CS constructions, different choices of function call graph are possible, so long as they produce the same R1CS constraints as the libraries. For the ACL2 PFCS constructions, different choices of PFCS hierarchy are possible, so long as, when flattened, they produce the same R1CS constraints as the libraries.

We plan to develop a *flattener* of PFCS to R1CS, which will also generate theorems of correct flattening, i.e. that the flattened R1CS constraints are equivalent to the PFCS constraints. The flattener will inline all the relation constraints, resulting in a sequence of equalities, all of which have the R1CS form because our PFCS constructions use equality constraints of the R1CS form.

The PFCS gadget constructions in ACL2 will be validated against sample gadgets from the libraries in the same way as explained in Section 5.5 for the R1CS gadget constructions in ACL2, with the addition of the aforementioned PFCS flattener.

7 Related Work

The authors are not aware of any other work to formally verify zero-knowledge circuits using ACL2; the paper [17] describes our snarkVM verification work in more detail. There is work using other tools, discussed below.

The QED^2 tool [36] is a specialized verifier that combines a dedicated algorithm with an SMT solver to automatically establish whether the outputs of a zero-knowledge circuit are uniquely determined by the inputs, or are instead under-constrained; it may also fail to find an answer. Their approach is automated, but our work addresses a stronger property (correctness); the unique determination of outputs from inputs is implied by soundness, when the specification of a gadget is that the gadget represents a computation (see Section 3). Their approach works on individual circuits like the ones in Sections 5.3 and 5.4, not on parameterized circuit families like the ones in Sections 5.5 and 6.2.

The SMT solver for finite fields described in [34] has been used to verify automatically whether circuits produced by certain compilers are sound (with respect to the compilation source) and deterministic (i.e. the outputs are uniquely determined by the inputs, as in [36]). Since our circuit specifications prescribe computations, in a way that may be similar to the sources of circuit compilers, their soundness proofs are analogous to ours (with determinism implied by soundness, at least in our case, as noted above); but their work does not cover completeness proofs. Their approach works on individual circuits, not on parameterized circuit families (as also noted above for [36]). For example, in our work, an unsigned n-bit integer addition circuit family as in Figure 3 is verified once, quickly, for every possible n (see Section 5.5), and can be used to verify correct compilation via a syntactic check; in contrast, in their work, instances of that family for different values of n are verified separately, taking increasing resources as n grows. Another advantage of verifying parameterized circuit families is that their definitions are essentially formal models of the circuit construction libraries and therefore help validate the libraries' design and implementation. The tradeoff between their and our approach is automation versus generality.

The Ecne tool [40] uses a dedicated algorithm to perform weak verification (their term to mean that the outputs are uniquely determined by the inputs) and witness verification (their term to mean that the outputs and the internal variables are uniquely determined by the inputs); their paper also discusses strong verification (i.e. correctness in our work), but only as future work. As already noted, the determinism of output variables is a consequence of soundness in our work. The determinism of internal variables is unnecessary for correctness, but it becomes a consequence of correctness if the latter is stated with respect to an extended specification S' that includes the internal variables (as in Section 5.5) and prescribes their computation from the inputs; Ecne's witness verification can be thus addressed with our techniques.

There is work on verifying the compilation of higher-level languages to zero-knowledge circuits [21, 35, 29]. While there is probably overlap with our work, and thus the opportunity for cross-fertilization, the purpose is a bit different: we verify the circuits constructed by existing libraries, which may be used as compilation targets, or for more general purposes such as programmatic construction of circuits; as noted above, our approach also helps validate the libraries.

As a final remark, the notion of existentially quantified circuits (EQCs) in [33] is related to the existential quantification of internal variables in PFCS.

8 Future Work

The main thread of future work is the continued verification of the snarkVM gadgets at Aleo, extending and improving the ACL2 PFCS library along the way. We plan to extend the PFCS lifter to work on parameterized gadgets, which requires a leap in sophistication in order to operate on the ACL2 functions that construct those gadgets rather than on the PFCS abstract syntax produced by the ACL2 functions. We also plan to build a proof-generating flattener of PFCS to R1CS form, to enable validation against samples extracted from snarkVM (see Section 6.3). To handle the gadget optimizations in snarkVM, we plan to develop proof-generating PFCS-to-PFCS transformations that correspond to those optimizations: these can be composed with the proofs for the vanilla (unoptimized) gadgets to obtain proofs of the optimized gadgets. The end goal is to verify all the snarkVM gadgets, including complex ones for cryptographic operations. These gadgets are being verified against specifications written in ACL2, which are not directly exposed to prover and verifier (cf. end of Section 3); they are building blocks for the next steps described below.

After reaching the above goal, the next verification target is the snarkVM compiler from Aleo instructions (the assembly-like language used to represent program code in the Aleo blockchain) to R1CS constraints, which uses the snarkVM gadget constructions to generate the constraints. The approach will be a detached proof-generating extension of the snarkVM compiler, built on the detached proof-generating extension of the snarkVM gadget constructions; every time the compiler is run, its generated constraints will be syntactically compared to the ones generated in ACL2, including flattening and optimizations as described above, to ensure that they are identical and thus that the proof applies to that exact compilation, as in a verifying compiler. The specification for the R1CS constraints generated by the snarkVM compiler is the source Aleo instructions program, which software developers can read and understand; the formal proof relies on a formalization of Aleo instructions that we are building in ACL2.

The same detached proof-generating extension approach will then be used for the compilation from

Leo [2] (a high-level programming language for the Aleo blockchain) to Aleo instructions, providing an end-to-end verifying compiler functionality from Leo to R1CS constraints via Aleo instructions. The specification for the R1CS constraints generated by the Leo and snarkVM compilers is the Leo source program, which software developers can read and understand; the formal proof relies on a formalization of Leo that we are building in ACL2 [16].

The roadmap delineated above is part of our overarching work to apply formal verification to ideally every aspect of the Aleo blockchain and ecosystem. The aforementioned formalizations of Aleo instructions and of Leo have more general value than their role in the formal verification of the compilation.

Alongside the PFCS-based compositional verification approach, it would also be interesting to continue exploring the Axe-based whole-gadget verification approach, in particular to improve the ability to recover sub-gadgets. There are tradeoffs between the two approaches: the first one keeps the proofs more manageable and efficient, but requires the formalization of the gadget constructions; the second one does not require that formalization, but needs to recover some of that structure during the proofs.

It would also be interesting to investigate the use or specialization of Axe for PFCS-based compositional proofs. Although PFCS aims at keeping proofs relatively small via parameterization and composition, Axe may come handy in case some large proof tasks arise. Axe's tactics may also be useful for certain proofs regardless of size, and not only for zero-knowledge circuits.

While interactive theorem proving is needed to verify parameterized circuit families with efficiency and generality, automated tools like SMT solvers could be useful for certain proof sub-tasks. ACL2 already has facilities to interface with automated reasoning tools.

There may be opportunities to partially automate the replication of the gadget constructions in the theorem prover, in the detached proof-generating extension approach (see Section 5.5). One avenue is the abstraction and translation of code in the gadget construction libraries. Another avenue, suggested by a reviewer, is to leverage any structure that can be recovered from generated gadgets.

9 Conclusion

Our exploration of the zero-knowledge circuit verification problem has shed more light into the problem, created and improved libraries and tools of more general use (e.g. the prime fields library), and evaluated increasingly sophisticated solution approaches. The PFCS-based compositional approach is promising, but completing the verification of the snarkVM gadgets will provide a more definitive validation.

The inherent restrictions on zero-knowledge circuits might initially lead to think of their verification as more tractable than general verification. Our exploration shows the opposite. As the size and complexity of the circuits grows, one eventually hits the "program verification wall". This should not be surprising, for a formalism that can describe sufficiently general computations. Although zero-knowledge circuits are not Turing-complete, and their verification is technically decidable because of their finiteness, the constraint solution space is so large that their verification is "practically undecidable".

Our exploration also confirms the importance of *structure* in formal verification. Preserving and leveraging the structure that is naturally available when the circuits are built promotes more manageable and efficient proofs, compared to losing that structure and then attempting to recover it.

Acknowledgements

Thanks to the Ethereum Foundation for funding Kestrel's work on the Semaphore circuit verification. Thanks to the Tezos Foundation for funding Kestrel's work on the Zcash circuit verification. Thanks to the Zcash team for their help in extracting the Zcash R1CS constraints. Thanks to our colleagues Pranav Gaddamadugu and Collin Chin at Aleo for their help in extracting the snarkVM R1CS constraints. Thanks to the reviewers for comments that led to improvements to the paper.

References

[1] Aleo Systems Inc.: *The Aleo Blochchain*. Available at `https://aleo.org`.

[2] Aleo Systems Inc.: *The Leo Language*. Available at `https://leo-lang.org`.

[3] Aleo Systems Inc.: *snarkVM*. Available at `https://github.com/AleoHQ/snarkVM`.

[4] *The ark-r1cs-std Library*. Available at `https://github.com/arkworks-rs/r1cs-std`.

[5] Aztec: *The Noir Language*. Available at `https://aztec.network/noir`.

[6] *The bellman Library*. Available at `https://github.com/zkcrypto/bellman`.

[7] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh & Michael Riabzev (2018): *Scalable, transparent, and post-quantum secure computational integrity*. Cryptology ePrint Archive, Paper 2018/046. Available at `https://eprint.iacr.org/2018/046`.

[8] Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer & Madars Virza (2014): *Zerocash: Decentralized Anonymous Payments from Bitcoin*. In: *Proc. 35th IEEE Symposium on Security and Privacy (SP)*, pp. 459–474, doi:10.1109/SP.2014.36.

[9] Nir Bitansky, Ran Canetti, Alessandro Chiesa & Eran Tromer (2011): *From Extractable Collision Resistance to Succinct Non-Interactive Arguments of Knowledge, and Back Again*. Cryptology ePrint Archive, Paper 2011/443. Available at `https://eprint.iacr.org/2011/443`.

[10] Manuel Blum, Paul Feldman & Silvio Micali (1988): *Non-interactive Zero-Knowledge and Its Applications*. In: *Proc. 20th Annual ACM Conference on Theory of Computing (STOC)*, pp. 103–112, doi:10.1145/62212.62222.

[11] Dan Bogdanov, Joosep Jääger, Peeter Laud, Härmel Nestra, Martin Pettai, Jaak Randmets, Ville Sokk, Kert Tali & Sandhra-Mirella Valdma (2022): *ZK-SecreC: a Domain-Specific Language for Zero Knowledge Proofs*, doi:10.48550/arXiv.2203.15448.

[12] Sean Bowe, Alessandro Chiesa, Matthew Green, Ian Miers, Pratyush Mishra & Howard Wu (2020): *ZEXE: Enabling Decentralized Private Computation*. In: *41st IEEE Symposium on Security and Privacy (SP)*, pp. 947–964, doi:10.1109/SP40000.2020.00050.

[13] Sean Bowe, Jack Grigg & Daira Hopwood (2019): *Recursive Proof Composition without a Trusted Setup*. Cryptology ePrint Archive, Paper 2019/1021. Available at `https://eprint.iacr.org/2019/1021`.

[14] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille & Greg Maxwell (2017): *Bulletproofs: Short Proofs for Confidential Transactions and More*. Cryptology ePrint Archive, Paper 2017/1066. Available at `https://eprint.iacr.org/2017/1066`.

[15] Vitalik Buterin (2016): *Quadratic Arithmetic Programs: from Zero to Hero*. Available at `https://medium.com/@VitalikButerin/quadratic-arithmetic-programs-from-zero-to-hero-f6d558cea649`.

[16] Collin Chin, Howard Wu, Raymond Chu, Alessandro Coglio, Eric McCarthy & Eric Smith (2021): *Leo: A Programming Language for Formally Verified, Zero-Knowledge Applications*. Cryptology ePrint Archive, Report 2021/651. Available at `https://eprint.iacr.org/2021/651`.

[17] Alessandro Coglio, Eric McCarthy, Eric Smith, Collin Chin, Pranav Gaddamadugu & Michel Dellepere (2023): *Compositional Formal Verification of Zero-Knowledge Circuits*. Cryptology ePrint Archive, Paper 2023/1278. Available at `https://eprint.iacr.org/2023/1278`. Presented at the Science of Blockchain Conference (SBC) 2023.

[18] Gilad Edelman (2021): *What is Web3, Anyway? WIRED*. Available at `https://www.wired.com/story/web3-gavin-wood-interview`.

[19] Electric Coin Company & Zcash Foundation: *The Zcash Blockchain*. Available at `https://z.cash`.

[20] Ethereum: *Zero-Knowledge Rollups*. Available at `https://ethereum.org/en/developers/docs/scaling/zk-rollups`.

[21] Cédric Fournet, Chantal Keller & Vincent Laporte (2016): *A Certified Compiler for Verifiable Computing*. In: *Proc. 29th IEEE Computer Security Foundations Symposium (CSF)*, pp. 268–280, doi:10.1109/CSF.2016.26.

[22] Shafi Goldwasser, Silvio Micali & Charles Rackoff (1985): *The Knowledge Complexity of Interactive Proof Systems*. In: *Proc. 17th Annual ACM Conference on Theory of Computing (STOC)*, pp. 291–304, doi:10.1145/22145.22178.

[23] Jens Groth (2016): *On the Size of Pairing-based Non-interactive Arguments*. Cryptology ePrint Archive, Paper 2016/260. Available at `https://eprint.iacr.org/2016/260`.

[24] Daira Hopwood, Sean Bowe, Taylor Hornby & Nathan Wilcox (2022): *Zcash Protocol Specification*. Available at `https://github.com/zcash/zips/blob/master/protocol/protocol.pdf`.

[25] Iden3: *The Circom Language*. Available at `https://github.com/iden3/circom`.

[26] Matt Kaufmann & J Strother Moore: *The ACL2 Theorem Prover*. Available at `http://acl2.org`.

[27] *The librustzcash Library*. Available at `https://github.com/zcash/librustzcash`.

[28] *The libsnark Library*. Available at `https://github.com/scipr-lab/libsnark`.

[29] Junrui Liu, Ian Kretz, Hanzhi Liu, Bryan Tan, Jonathan Wang, Yi Sun, Luke Pearson, Anders Miltner, Işıl Dillig & Yu Feng (2023): *Certifying Zero-Knowledge Circuits with Refinement Types*. Cryptology ePrint Archive, Report 2023/547. Available at `https://eprint.iacr.org/2023/547`.

[30] Daniel Lubarov & Brendan Farmer (2019): *R1CS Programming: ZK0x04 Workshop Notes*. Available at `https://github.com/mir-protocol/r1cs-workshop/tree/master/workshop.pdf`.

[31] Lurk Team: *The Lurk Language*. Available at `https://lurk-lang.org`.

[32] Ian Miers, Christina Garman, Matthew Green & Aviel D. Rubin (2013): *Zerocoin: Anonymous Distributed E-Cash from Bitcoin*. In: *Proc. 34th IEEE Symposium on Security and Privacy (SP)*, pp. 397–411, doi:10.1109/SP.2013.34.

[33] Alex Ozdemir, Fraser Brown & Riad S. Wahby (2022): *CirC: Compiler Infrastructure for Proof Systems, Software Verification, and More*. In: *Proc. 43rd IEEE Symposium on Security and Privacy (SP)*, pp. 2248–2266, doi:10.1109/SP46214.2022.9833782.

[34] Alex Ozdemir, Gereon Kremer, Cesare Tinelli & Clark Barrett (2023): *Satisfiability Modulo Finite Fields*. In: *Proc. 35th International Conference on Computer Aided Verification (CAV), Part II*, *Lecture Notes in Computer Science* 13965, pp. 163—-186, doi:10.1007/978-3-031-37703-7_8.

[35] Alex Ozdemir, Riad S. Wahby, Fraser Brown & Clark Barrett (2023): *Bounded Verification for Finite-Field-Blasting*. In: *Proc. 35th International Conference on Computer Aided Verification (CAV), Part III*, *Lecture Notes in Computer Science* 13966, pp. 154–175, doi:10.1007/978-3-031-37709-9_8.

[36] Shankara Pailoor, Yanju Chen, Franklyn Wang, Clara Rodríguez, Jacob Van Geffen, Jason Morton, Michael Chu, Brian Gu, Yu Feng & Işıl Dillig (2023): *Automated Detection of Under-Constrained Circuits in Zero-Knowledge Proofs*. *Proc. ACM Program. Lang.* 7(PLDI), doi:10.1145/3591282.

[37] Alex Pruden (2020): *How Zero-Knowledge is Rebalancing the Scales of the Internet*. Available at `https://aleo.org/post/how-zero-knowledge-is-rebalancing-the-scales-of-the-internet`.

[38] The ACL2 Community: *The ACL2 Theorem Prover and Community Books: Documentation*. Available at `http:/acl2.org/manual`.

[39] The ACL2 Community: *The ACL2 Theorem Prover and Community Books: Source Code*. Available at `http://github.com/acl2/acl2`.

[40] Franklyn Wang (2022): *Ecne: Automated Verification of ZK Circuits*. OxPARC Blog. Available at `https://0xparc.org/blog/ecne`.