Compositional Formal Verification of Zero-Knowledge Circuits

Alessandro Coglio[a,b], Eric McCarthy[a,b], Eric Smith[b],
Collin Chin[a], Pranav Gaddamadugu[a], and Michel Dellepere[a]

[a]Aleo Systems, Inc.
[b]Kestrel Institute

Abstract

We provide a preliminary report of our ongoing work in formally defining and verifying, in a compositional way, the R1CS gadgets generated by Aleo's snarkVM. The approach is applicable to other systems that generate gadgets in a similar manner, and that may use non-R1CS representations.

1 Introduction

In zkSNARKs (Section A.3), computations are represented as zero-knowledge circuits in forms such as Rank-1 Constraint Systems (R1CS)[1] (Section A.2). These constraints are fairly low-level, often distant from the higher-level computations they represent. This raises the issue of ensuring the correctness of the representations: if a computation is represented incorrectly, a zero-knowledge proof may not quite prove what was intended [0xP]. To developers, the computation might be overspecified in the constraint system, forcing them to pay more constraints than necessary. To users, the computation may be underconstrained, allowing an attacker to print an infinite amount of assets [SWB19, Sem19, Tor19, Bas20].

As in other applications, formal verification can provide strong guarantees that the computation is represented correctly. Given formal models of the computation and of the circuit representation, correctness can be established by proving a theorem asserting the desired relation between the two models.[2]

This paper is a preliminary report on our ongoing work to formally verify the R1CSes generated by Aleo's snarkVM (Section A.2). We use the ACL2 theorem prover [KM] to formalize snarkVM's R1CS constructions and to prove that the constructions match the specifications of the computations they are intended to represent. Our formal verification approach is compositional: we reify the notion of R1CS gadget, informally used in the literature to refer to R1CS components, and verify each gadget using the theorems about its sub-gadgets.

This work is part of our overarching plan to apply formal verification to every aspect of the Aleo programming stack [Aleb], with initial emphasis on the correct compilation of Leo [CWC+21] to R1CS via Aleo instructions (Section A.4), which are the assembly-like language used to represent program code in the Aleo blockchain. The generation of R1CS is the last step of this compilation process: the work described in this paper aims at verifying the R1CS constructions performed in this last step, providing the basis for verifying the correct compilation of Aleo instructions to R1CS, which is future work. The correct compilation of Leo to Aleo instructions is also mainly future work, but we have already started some effort in this direction [CWC+21].

[1] In this paper, we use 'R1CS' either to mean a specific set of constraints or to denote the general form of the constraints. We also use 'R1CSes' to mean a set of Rank-1 Constraint Systems.

[2] Note the difference between zero-knowledge proofs, which provide statistically overwhelming evidence of certain facts without revealing any details about those facts, and the proofs of theorems in formal verification, which provide logically absolute evidence of certain facts without hiding any information. Any use of the unqualified terms 'proof', 'prove', etc. in this paper should be easily disambiguated by context.

We believe that the R1CS gadget verification work described in this paper is more general than snarkVM and more general than R1CS. The way snarkVM generates R1CS is analogous to other systems [GN20, OBW20], whether driven by direct user programming [Bow] or part of a compilation process [GN20]; our compositional verification approach applies to this way of generating R1CS. Furthermore, as discussed in the later part of this paper (Section 5) we are moving towards the use of a richer circuit representation than R1CS, called PFCS, which can represent without overhead not only R1CS but also other forms such as CCS [STW23], Plonkish [GWC19][CBBZ22], and AIR [GPR21]; thus, our approach should be applicable to those other forms as well.

Section 2 presents the formal framework of our formal verification work. Section 3 overviews its mechanization in ACL2, whose results are summarized in Section 4. Section 5 discusses our approach to scale the verification. Related and future work are described in Section 6 and Section 7, while Section 8 concludes. Background material is in Section A in the appendix, along with the evaluation details in Section B.

2 Formal Framework

2.1 Computations

For our purposes, a (deterministic) *computation* is 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 an error $\mathcal{E}$ that is distinct from (tuples of) inputs and output. The error $\mathcal{E}$ can model the occurrence of events such as division by zero (if that is regarded as erroneous in the modeled computation), or in general the application of the computation on disallowed inputs.

The case $n = 0$ of no inputs is not very interesting, but causes no additional difficulties to the framework. The case $m = 0$ of no outputs can model assertion-like computations that return either nothing (more precisely, the empty tuple $\langle\,\rangle$ of outputs) or the error if they fail. We regard a 1-tuple $\langle a \rangle$ equivalent to its only element a, and thus speak of tuples of inputs or outputs also when f has a single input or output.

This model only considers the functional input/output behavior of computations, not their performance and other non-functional characteristics. This is adequate for representing computations as R1CSes.

2.2 Data Encodings

The input and output sets I_i and O_j may correspond to high-level data types. To represent the computation f in R1CS form, the input and output data must be represented as elements in a prime field $\mathbb{F}$. That is, there must be injective *encoding* functions

$$e_i^{\mathrm{I}} : I_i \to \mathbb{F}^{n_i} \qquad e_j^{\mathrm{O}} : O_j \to \mathbb{F}^{m_j}$$

that represent each input $\iota \in I_i$ as $n_i \geq 1$ field elements and each output $\omega \in O_j$ as $m_j \geq 1$ field elements.

This leads to considering a version of the computation on encoded data

$$\widehat{f} : \mathbb{F}^N \to \mathbb{F}^M \cup \{\mathcal{E}\},$$

where $N = \sum_i n_i$ and $M = \sum_j m_j$, such that

$$\begin{array}{lll} \forall \iota_1, \ldots, \iota_n, \omega_1, \ldots, \omega_m. & [f(\iota_1, \ldots, \iota_n) = \langle \omega_1, \ldots, \omega_m \rangle & \implies \widehat{f}(e_1^{\mathrm{I}}(\iota_1), \ldots, e_n^{\mathrm{I}}(\iota_n)) = \langle e_1^{\mathrm{O}}(\omega_1), \ldots, e_m^{\mathrm{O}}(\omega_m) \rangle] \\ \forall \iota_1, \ldots, \iota_n. & [f(\iota_1, \ldots, \iota_n) = \mathcal{E} & \implies \widehat{f}(e_1^{\mathrm{I}}(\iota_1), \ldots, e_n^{\mathrm{I}}(\iota_n)) = \mathcal{E}] \end{array}$$

i.e. if f returns non-error outputs on some inputs then $\widehat{f}$ returns those encoded outputs on those encoded inputs, and if f returns an error on some inputs then $\widehat{f}$ returns the error on those encoded inputs;[3] $\widehat{f}$ returns $\mathcal{E}$ when applied outside the range of the input encoding functions.

In the rest of this paper, for simplicity, we consider computations already operating on field elements, leaving the input and output encodings as a separate issue. Thus, we regard a computation as a function

$$f : \mathbb{F}^n \to \mathbb{F}^m \cup \{\mathcal{E}\}$$

from $n \geq 0$ input field elements to $m \geq 0$ output field elements or to an error $\mathcal{E}$.

[3] For notational brevity, in this paper we liberally flatten nested tuples.

2.3 R1CS Gadgets

As explained in Section A.2, an R1CS is a finite sequence of constraints. Given an ordering $v_1, \ldots, v_r$ of all the variables used in the constraints, the R1CS determines an r-ary relation over the field

$$R \subseteq \mathbb{F}^r$$

that holds exactly when all the constraints hold, assigning the field elements to the variables in order.

When an R1CS represents a computation f of the kind characterized at the end of Section 2.2, the r variables can be divided into three disjoint sets:

- n *input variables*, corresponding to the n inputs of f.
- m *output variables*, corresponding to the m outputs of f.
- $l = r - n - m$ *auxiliary variables*, for internal use.

It is convenient for the variable ordering $v_1, \ldots, v_r$ to have the form $v_1^{\mathrm{I}}, \ldots, v_n^{\mathrm{I}}, v_1^{\mathrm{O}}, \ldots, v_m^{\mathrm{O}}, v_1^{\mathrm{A}}, \ldots, v_l^{\mathrm{A}}$, where $v_1^{\mathrm{I}}, \ldots, v_n^{\mathrm{I}}$ are the input variables, $v_1^{\mathrm{O}}, \ldots, v_m^{\mathrm{O}}$ are the output variables, and $v_1^{\mathrm{A}}, \ldots, v_l^{\mathrm{A}}$ are the auxiliary variables. Then the relation determined by the R1CS can be viewed as a ternary one

$$R \subseteq \mathbb{F}^n \times \mathbb{F}^m \times \mathbb{F}^l$$

over the tuples of input, output, and auxiliary field elements. This also determines an input/output binary relation

$$\widetilde{R} \subseteq \mathbb{F}^n \times \mathbb{F}^m$$
$$\widetilde{R}(\iota_1, \ldots, \iota_n, \omega_1, \ldots, \omega_m) = \exists \alpha_1, \ldots, \alpha_l . [R(\iota_1, \ldots, \iota_n, \omega_1, \ldots, \omega_m, \alpha_1, \ldots, \alpha_l)]$$

obtained from R by existentially quantifying the auxiliaries.

We define an *R1CS gadget* as an R1CS (i.e. sequence of constraints) accompanied by an ordering and an input/output/auxiliary designation of the variables as above, determining a relation over (tuples of) field elements as above. This definition is biased towards the idea that an R1CS gadget represents a computation, in the sense made precise in Section 2.4.

Larger gadgets can be constructed from smaller gadgets, hierarchically. The interaction between the sub-gadgets of a gadget is realized via shared variables: for instance, the same variable can be an output of a sub-gadget and an input of another sub-gadgets.

2.4 Correctness

An R1CS gadget as characterized above is *correct* with respect to a computation f exactly when

$$\forall \iota_1, \ldots, \iota_n, \omega_1, \ldots, \omega_m . \ [\widetilde{R}(\iota_1, \ldots, \iota_n, \omega_1, \ldots, \omega_m) \iff f(\iota_1, \ldots, \iota_n) = \langle \omega_1, \ldots, \omega_m \rangle].$$

The left-to-right ('only if') implication

$$\forall \iota_1, \ldots, \iota_n, \omega_1, \ldots, \omega_m . \ [\widetilde{R}(\iota_1, \ldots, \iota_n, \omega_1, \ldots, \omega_m) \implies f(\iota_1, \ldots, \iota_n) = \langle \omega_1, \ldots, \omega_m \rangle]$$

is *soundness*. Expanding the definition of $\widetilde{R}$ and turning the existential quantification on the premise into a universal quantification on the implication yields

$$\forall \iota_1, \ldots, \iota_n, \omega_1, \ldots, \omega_m, \alpha_1, \ldots, \alpha_l . \ [R(\iota_1, \ldots, \iota_n, \omega_1, \ldots, \omega_m, \alpha_1, \ldots, \alpha_l) \implies f(\iota_1, \ldots, \iota_n) = \langle \omega_1, \ldots, \omega_m \rangle],$$

i.e. every solution of the constraints corresponds to the instance of the computation that returns the outputs from the inputs. In other words, the gadget represents *only* correct computation instances.

The right-to-left ('if') implication

$$\forall \iota_1, \ldots, \iota_n, \omega_1, \ldots, \omega_m . \ [f(\iota_1, \ldots, \iota_n) = \langle \omega_1, \ldots, \omega_m \rangle \implies \widetilde{R}(\iota_1, \ldots, \iota_n, \omega_1, \ldots, \omega_m)]$$

is *completeness*. Expanding the definition of $\widetilde{R}$ yields

$$\forall \iota_1, \ldots, \iota_n, \omega_1, \ldots, \omega_m . \ [f(\iota_1, \ldots, \iota_n) = \langle \omega_1, \ldots, \omega_m \rangle \implies \exists \alpha_1, \ldots, \alpha_l . [R(\iota_1, \ldots, \iota_n, \omega_1, \ldots, \omega_m, \alpha_1, \ldots, \alpha_l)]],$$

i.e. for every instance of the computation there exist auxiliaries that together with the inputs and outputs form a solution to the constraints. In other words, the gadget represents *all* correct computation instances.

Soundness alone is not sufficient, because a gadget that has no satisfying assignment is trivially sound. Completeness ensures that the gadget has a satisfying assignment for each input that does not yield an error.

3 Formalization and Verification in ACL2

We provide an overview of our mechanization in ACL2 of the formal framework presented in Section 2. Some of the ACL2 code is slightly simplified in order to focus the exposition on the essential ideas.

3.1 Specifications, the Units of Verification

To verify that a given R1CS correctly implements a computation f of the kind characterized at the end of Section 2.2, we need a *specification* of that computation, which we write in ACL2.

We want to verify the R1CS that is input to the zero-knowledge proving system, which is often very large. If that R1CS can be split into component computations, we verify those, and compose them into a verification of the whole.

In an Aleo instructions program, each instruction is a computation. Often an instruction is composed of sub-gadgets that are computations. We verify these bottom-up.

To understand the composite structure, we examine the snarkVM code used to build an R1CS from an Aleo instruction and from its sub-gadgets. We write an *ACL2 gadget definition*, an alternate implementation in ACL2 of the snarkVM code that builds the R1CS. We test this definition by showing that the R1CS it creates, for selected parameters, is equivalent to R1CS samples extracted from snarkVM.[4]

We verify an R1CS gadget by proving a theorem saying that the R1CS generated by the ACL2 gadget definition is correct with respect to the ACL2 specification. We verify a composite gadget by proving a theorem that the composed ACL2 gadget definition is correct with respect to an ACL2 specification of the composed computation.

3.2 Formal Specification

When writing ACL2 specifications for Aleo instructions, we abstract some of the details that affect how the instruction is compiled to R1CS. For example, for an operation on fields, the field order is included as a parameter to the functional specification. Also, the specification does not distinguish between a constant argument and a register argument. However, we do write different specifications for different operand types.

For example, the Aleo instruction

```
div r0 r1 into r2; // divide register 0 by register 1 and store result in register 2
```

where `r0` and `r1` contain field elements, can be modeled (specified) by the function

```
(defun field-div (x y p)
  (if (equal y 0)
      (error)
    (pfield::div x y p)))
```

which returns an error (i.e. $\mathcal{E}$ in Section 2.1) if the divisor is 0, otherwise returns the result of the field division operation `pfield::div`, which is from the ACL2 prime fields library.[5]

To explicate the domain and range of the specification, we prove type theorems about them (not shown). This is important because limitations on arguments are relevant for completeness and must be heeded by callers when composing verifications. For `field-div`, these follow straightforwardly from the definition.

3.3 R1CS Formalization

To give formal status to R1CS gadgets, we formalize the R1CS syntax and semantics.

Our R1CS (abstract) *syntax* is approximately described by the following ABNF grammar:

[4] These are just tests to validate our constructions at this time. Section 7 discusses how we will ensure that we verify the exact same gadgets generated by snarkVM.

[5] acl2.org/manual topic: prime-fields

```
constant = <an integer among ..., -2, -1, 0, 1, 2, ...>
variable = <a symbol, distinct from the integers>
monomial = constant / constant variable ; constant, or variable with coefficient
polynomial = *monomial ; sum of zero or more monomials, i.e. a linear combination
constraint = polynomial polynomial polynomial ; 1st * 2nd = 3rd
r1cs = *constraint ; zero or more constraints
```

The actual definition in ACL2[6] is expressed differently, but it is essentially equivalent to this grammar. For instance, a constraint is defined as an aggregate data structure with three components `a`, `b`, `c` that are sparse vectors (another name for a polynomial as defined in the grammar above):

```
(defaggregate r1cs-constraint
  ((a sparse-vectorp)
   (b sparse-vectorp)
   (c sparse-vectorp)))
```

Our R1CS *semantics* starts with the concept of *assignment*, which is a finite map from the symbols used for variables to the integers, i.e. an integer is assigned to each variable.

Given an assignment for all the variables in a polynomial (i.e. sparse vector), we define the *evaluation* of the polynomial by multiplying and adding:

```
(defun dot-product (vector assignment prime) ...) ; details omitted
```

This is called `dot-product` based on the view of a dot product between the vector of the coefficients of the polynomial and the vector of (the values assigned to) the variables of the polynomial (the constant monomial is regarded as multiplying a pseudo-variable '1'). The argument `prime` is the prime p that defines the prime field, needed to perform addition and multiplication in the field.

Based on that, given an assignment and a prime as above, we define the *satisfaction* of a constraint as equality between the product of the `a` and `b` polynomials and the `c` polynomial:

```
(defun r1cs-constraint-holdsp (constraint assignment prime)
  (equal (mul (dot-product (r1cs-constraint->a constraint) assignment prime)
              (dot-product (r1cs-constraint->b constraint) assignment prime)
              prime)
         (dot-product (r1cs-constraint->c constraint) assignment prime)))
```

The `mul` function is the multiplication in the prime field, which depends on the prime.

The satisfaction of a sequence of constraints is defined as the satisfaction of all the constraints:

```
(defun r1cs-constraints-holdp (constraints assignment prime)
  (or (endp constraints)
      (and (r1cs-constraint-holdsp (first constraints) assignment prime)
           (r1cs-constraints-holdp (rest constraints) assignment prime))))
```

3.4 R1CS Gadget Formalization

Given the R1CS formalization just described, we formalize R1CS gadgets by constructing their abstract syntax, mimicking the corresponding snarkVM code.

For example, the field division gadget

$$\begin{array}{rcl} (y)\ (w) & = & (1) \\ (x)\ (w) & = & (z) \end{array}$$

[6] acl2.org/manual topic: r1cs

where x and y are input variables (dividend and divisor), z is an output variable (quotient), and w is an auxiliary variable (inverse of y, if not zero),[7] can be formalized as

```
(defun field-div-gadget (x y z w)
  (list ; list of constraints
   (make-r1cs-constraint ; constructs an r1cs-constraint aggregate
    :a (list (list 1 y))     ; polynomial (y)
    :b (list (list 1 w))     ; polynomial (w)
    :c (list (list 1 1)))    ; polynomial (1)
   (make-r1cs-constraint ; constructs an r1cs-constraint aggregate
    :a (list (list 1 x))     ; polynomial (x)
    :b (list (list 1 w))     ; polynomial (w)
    :c (list (list 1 z)))))  ; polynomial (z)
```

This is actually a family of gadgets, parameterized over the specific choice of variable symbols, which are passed as arguments to `field-div-gadget` (`x`, `y`, `z`, and `w` are meta-variables). Since the exact variables of each field division gadget generated by snarkVM vary (as snarkVM generates variables with globally increasing numeric indices), this parameterized gadget formulation captures all the possible gadgets.

The field division gadget is actually formalized by composing a field inversion gadget with a field multiplication gadget:

```
(defun field-div-gadget (x y z w)
  (append (field-inverse-gadget y w)
          (field-mul-gadget x w z)))
```

where `field-inverse-gadget` and `field-mul-gadget` are defined as consisting of the first and second constraint shown above. This is a simple example of hierarchical gadget construction; the interaction between the two sub-gadgets is via the shared variable `w`, which is an output variable for the field inversion gadget and an input variable for the field multiplication gadget.

While the three (parameterized) gadgets presented above have fixed numbers of variables and constraints, other gadget constructions have a varying number of variables and constraints. An example is the field-to-bits gadget

$$\begin{aligned}
(y_0)\ (1 - y_0) &= (0) \\
&\cdots \\
(y_{n-1})\ (1 - y_{n-1}) &= (0) \\
(x)\ (1) &= (y_0 + 2y_1 + 4y_2 + \cdots + 2^{n-1}y_{n-1}) \\
&\cdots
\end{aligned}$$

where x is an input variable, $y_0, \ldots, y_{n-1}$ are output variables, the first n constraints say that each y_i is a bit (i.e. 0 or 1), the other constraint says that x is the weighted sum of the y_i bits according to powers of 2, and the final ellipsis omits (for brevity) additional constraints to ensure that the weighted sum is below the prime; the varying number n is the number of bits of the prime p. This is formalized as a family of gadgets

```
(defun field-to-bits-gadget (x ys ... prime)
  (append (boolean-check-gadget-list ys)
          (pow2sum-gadget x ys)
          ...)) ; sub-gadgets to check that the weighted sum of ys is below prime
```

where `ys` is a list of variable symbols of appropriate length, `prime` is the prime that defines the field, and the ellipses stand for additional variables and constraints used to check that the weighted sum of the `ys` bits is below the prime. Here `boolean-check-gadget-list` constructs a list of boolean check gadgets, and `pow2sum-gadget` constructs a gadget that equates `x` to the weighted sum of `ys`. Functions to construct gadgets with varying number of variables or constraints are defined recursively in ACL2, or call

[7] The auxiliary variable w serves to ensure soundness. The deceitfully equivalent single constraint $(y)\ (z) = (x)$ is satisfied for every value of z when $x = y = 0$.

functions defined recursively; in this case, `boolean-check-gadget-list` is defined recursively in terms of `boolean-check-gadget`, and `pow2sum-gadget` is defined recursively based on `ys`.

3.5 Correctness Theorems

Given gadget definitions and the corresponding computation specifications, we prove correctness via ACL2 theorems that correspond to soundness and completeness in Section 2.4.

For instance, the soundness of the field division gadget in Section 3.4 is expressed as

```
(defthm field-div-soundness
  (implies ...; hypotheses omitted
           (let ((xval (lookup x asg)) (yval (lookup y asg)) (zval (lookup z asg)))
             (implies (r1cs-constraints-holdp (field-div-gadget x y z w) asg prime)
                      (equal zval (field-div xval yval prime))))))
```

which says that if the constraints of the gadget hold for an assignment `asg` then the value `zval` assigned to the variable `z` is the result of the division of the values `xval` and `yval` assigned to the variables `x` and `y`. The omitted hypotheses of the theorem say that `asg` assigns values to all the variables including `w`, as well as other structural facts that are not important here. Since `zval` is a field element and not an error, this theorem also implicitly says that the divisor `yval` is not 0. The `(r1-constraints-holdp ...)` assertion corresponds to $R(\ldots)$ in Section 2.4.

The completeness of the field division gadget is a little more awkward to formulate, because of the auxiliary variable `w`, which is existentially quantified. One approach is to introduce a `defun-sk` (see Section A.5) that existentially quantifies over the value `wval` assigned to `w`, closely following the formulation in Section 2.4, but that involves an extra function definition. Another approach, when the values of the auxiliary variables are determined by the input variables as is often the case, is to eschew the explicit existential quantification in favor of assertions that functionally determine the values of the auxiliary variables (cf. Skolemization in logic). For instance, the completeness of the field division gadget can be expressed as

```
(defthm field-div-completeness
  (implies ... ; hypotheses and let bindings omitted)
           (implies (and (equal zval (field-div xval yval prime))
                         (equal wval (field-inv yval prime)))
                    (r1cs-constraints-holdp (field-div-gadget x y z w) asg prime)))
```

but that involves an extra assertion on `wval`, which is tangential to the essence of the field division gadget. See Section 5 for an approach to overcome this awkwardness.

The correctness theorems above prove the soundness and completeness not just of a single gadget, but of the whole family constructed by `field-div-gadget`, and not just for a specific prime field, but for every prime field. The same generality applies to theorems for gadgets with varying numbers of variables and constraints, such as the field-to-bits gadget in Section 3.4: the proved correctness holds for any number of variables and constraints.

When a gadget construction is defined recursively, its correctness is proved by induction in ACL2. When a gadget includes sub-gadgets, the correctness theorems for the sub-gadgets are used to prove the correctness of the gadget (e.g. the correctness theorems of field inversion and multiplication are used to prove the correctness of field division): the verification is compositional, according to the gadget structure. When constraints are directly involved (i.e. not just sub-gadgets), the definition of `r1cs-constraints-holdp` and other ACL2 functions for the formal semantics of R1CS are expanded, reducing the proof task to showing that the prime field calculations achieve the specification, i.e. the core correctness argument of the gadget. The difficulty of these proofs range from easy and automatic in ACL2 to needing some fairly elaborate lemmas and intermediate definitions from us to guide ACL2.

4 Evaluation

We applied the mechanization of the formal framework, described in Section 3, to the gadgets used in snarkVM (Section A.4). The goal of this evaluation was to maintain parity with the gadget implementations so that they can be verified as they evolve. To that end, we constructed an end-to-end pipeline which extracts R1CS samples from the snarkVM gadget implementations, and then verifies the extracted R1CSes against the formal specifications. This infrastructure exists in snarkVM and has been used to verify the original gadgets as well as bug fixes and optimizations that we discovered during the course of this work.

At the time of this writing, we have formalized and verified all the gadgets for boolean operations, all the gadgets for field operations, and most of the the gadgets for integer operations. In the process, we identified two bugs and three classes of optimizations, whose details are given in Section B. While these bugs and optimizations could have been discovered without formal verification by carefully examining the gadgets, it is our general experience that the close scrutiny demanded by formal verification naturally encourages and facilitates this kind of discoveries: we discovered one of the bugs directly because we could not prove the correctness of the buggy gadget; we discovered one of the optimizations directly because our correctness proof did not use the constraints that was then optimized away.

5 Towards a More Structured Representation

5.1 Challenges in the Current Approach

In the parameterized gadget constructions described in Section 3.4, the hierarchical structure of the gadgets is captured not in the gadgets themselves, which are flat sequences of constraints, but in the definitions of the ACL2 constructor functions for the gadgets. This supports well the compositional verification of soundness, but as explained in Section 3.5 completeness is more awkward, due to the presence of the auxiliary variables, which must be existentially quantified.

Another issue is the increasing number of parameters for the auxiliary variables. For example, the function `field-div-gadget` has parameters not only for its inputs (`x` and `y`) and output (`z`), but also for its auxiliary (`w`), which is passed to both `field-inv-gadget` and `field-mul-gadget`. As another example, the function `field-to-bits-gadget` has parameters (indicated by an ellipsis) for auxiliary variables to pass to the sub-gadgets for checking that the weighted sum is below the prime. In general, a gadget construction function must have parameters for all the variables used in its constraints and sub-gadgets, which does not scale well as the gadgets grow in size and complexity.

5.2 Overcoming the Challenges

We have developed *Prime Field Constraint Systems* (*PFCS*), which extend (and subsume) R1CS by allowing (1) equality constraints between arbitrary expressions and (2) explicitly hierarchical structuring of the constraints.[8] We have formalized the PFCS syntax and semantics in ACL2.[9]

Our PFCS (abstract) *syntax* is approximately described by the following ABNF grammar:

```
constant = <an integer among ..., -2, -1, 0, 1, 2, ...>
variable = <a symbol, distinct from the integers>
name = <a symbol>
expression = constant
           / variable
           / expression "+" expression
           / expression "*" expression
constraint = expression "=" expression
           / name "(" expression* ")"
relation = name "(" *variable ")" "{" *constraint "}"
```

[8] For our current purpose of verifying R1CSes, we only need the second extension. However, PFCS is more general, and could be applied to other forms than R1CS, taking advantage of the first extension.

[9] acl2.org/manual topic: prime-field-constraint-systems

Expressions are built out of constants, variables, additions, and multiplications. Equality constraints are between two expressions. Finite sequences of constraints can be grouped into named relations with variable parameters, which may be used as constraints by replacing the parameters with expressions (i.e. they may be "called" on expression arguments).

Our PFCS *semantics* is similar to, but more complex than, our R1CS semantics. It defines the evaluation of expressions given an assignment and a prime, and the satisfaction of equality constraints, as expected, similarly to the R1CS semantics. The satisfaction of a relation constraints `rel(exp1,exp2,...)` by an assignment `asg` is defined as follows: evaluate the expressions to values (field elements) `val1`, `val2`, etc. using `asg`; form an assignment `asg1` of the values to the parameters of `rel`; the constraint is satisfied if there exists an assignment `asg2` of field elements to the non-parameter variables of `rel` such that all the constraints that define `rel` are satisfied by the combination of `asg1` and `asg2`.

A *PFCS gadget* is a defined relation according to the above grammar, accompanied by a partitioning of the relation's parameters into input and output variables. The non-parameter variables are auxiliary variables, which the aforementioned semantics existentially quantifies over. A PFCS gadget semantically corresponds directly to $\widetilde{R}$ in Section 2.3; the defining constraints of the relation semantically correspond to R in Section 2.3.

We believe that the PFCS formalism is general enough to describe more complex arithmetizations such as Customizable Constraint Systems [STW23]. If we limit the PFCS equality constraints to have the R1CS form (i.e. equality between a product of two linear combinations and a linear combination), the PFCS relations provide an explicit hierarchical structuring of the R1CS constraints, which can be thus regarded as *structured R1CS*.

Given the formalization in ACL2 of the PFCS formalism just sketched, PFCS gadgets in ACL2 can be defined similarly to Section 3.4, by constructing their abstract syntax. For example, the field division gadget and sub-gadgets can be defined as

```
(defun field-mul-gadget ()
  (pfdef 'field-mul                                    ; name
         (list 'x 'y 'z)                               ; parameters
         (pf= (pf* (pfvar 'x) (pfvar 'y)) (pfvar 'z)))) ; constraint
(defun field-inv-gadget ()
  (pfdef 'field-inv                                     ; name
         (list 'x 'y)                                   ; parameters
         (pf= (pf* (pfvar 'x) (pfvar 'y)) (pfconst 1)))) ; constraint
(defun field-div-gadget ()
  (pfdef 'field-div                                             ; name
         (list 'x 'y 'z)                                        ; parameters
         (pfcall 'field-inv (pfvar 'y) (pfvar 'w))              ; constraint
         (pfcall 'field-mul (pfvar 'x) (pfvar 'w) (pfvar 'z)))) ; constraint
```

where the quotes `'` denote ACL2 symbols for relation and variable names, and where the `pf...` notations construct PFCS expressions and definitions. There are some crucial differences with Section 3.4:

- The hierarchical structure is not in the ACL2 function definitions, which do not call each other, but in the PFCS abstract syntax.
- Since PFCS supports the instantiation of parameters with arguments (e.g. the call to `'field-inv` in `'field-div` instantiates the parameters `'x` and `'y` with the arguments `'y` and `'w`), the gadgets use fixed variable symbols, obviating the need to parameterize the gadgets on the variable names, avoiding the scalability issues pointed out earlier.
- In `'field-div` the variable `'w` is auxiliary, because it is not part of the parameters.

While the above examples are unparameterized gadgets, gadgets with varying numbers of variables and constraints need to be parameterized, but only over one or more non-negative numbers that specify numbers of variables and constraints as appropriate. Gadgets with fixed or varying numbers of variables and constraints may need to be parameterized over the prime. However, this is a much smaller and scalable parameterization than for the corresponding R1CS constructions, because variable names do not need to be parameters. For example, a PFCS definition of the field-to-bits gadget in Section 3.4 is parameterized over

the prime, whose number of bits can be obtained in the ACL2 definition and used to construct a suitable number of variables (corresponding to `ys`) with a common prefix and a numeric suffix (`'y0`, `'y1`, `'y2`, etc.).

Since the PFCS semantics existentially quantifies the auxiliary variables, correctness theorems, including both soundness and completeness, can be stated more naturally than in the R1CS form, overcoming the other challenge pointed out earlier. For instance, the correctness theorem for the field division gadget is

```
(defthm field-div-correctness
  (implies ... ; details omitted
           (equal (definition-satp 'field-div defs (list x y z) prime)
                  (equal z (field-div x y p)))))
```

which says that `'field-div` is satisfied by the field elements `x`, `y`, and `z` if and only if `z` is the (non-error) result of `field-div` on `x` and `y`; note that there is no mention of any `w` value for the auxiliary variable.[10] A theorem of this form can be used as an ACL2 rewrite rule in theorems about gadgets that use field division as a sub-gadget; in fact, the proof of `field-div-correctness` makes use of the similarly phrased correctness theorems for the field inversion and multiplication sub-gadgets.

6 Related Work

The formal verification of R1CS and similar constraint systems is a relatively new area of research, and therefore there is not a large amount of literature yet [MT].

In previous work, we used the Axe toolkit[11] to verify the functional correctness of the large and complex R1CS of the BLAKE2s hash [CWC+21]. This verification was performed on the R1CS as a whole, extracted from an earlier version of snarkVM, while in our current approach we are building and verifying gadgets compositionally.

In earlier work at Kestrel, we used ACL2 and Axe to verify R1CS components of Ethereum's Semaphore[12] and of Zcash.[13] This provided inspiration for many of the ideas presented in this paper.

Our work focuses on the formal verification of R1CS functional correctness using theorem provers. Complementary approaches, described below, include using more automated techniques to detect issues in circuits, or to prove higher-level properties (such as circuit equivalence and compliance with specifications) as far as scalability allows. Used in conjunction, these approaches together with our approach can form a comprehensive toolkit for formally verifying properties about zero-knowledge circuits. Where the more automated techniques fall short, the techniques outlined in this paper offer a more scalable and efficient approach, at the cost of requiring more human guidance.

The QED^2 tool [PCW+23] 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 in our approach, since the specification of a gadget is that of a functional computation. Their approach works on individual circuits, not on parameterized circuit families like ours.

The SMT solver for finite fields described in [OKTB23], namely cvc5 with a theory of finite fields, 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). Since our circuit specifications prescribe computations, in a way that may be similar to the source code input to a circuit compiler, their soundness proofs are analogous to ours (with determinism implied by soundness, as noted above); but their work does not cover completeness proofs. Their approach works on individual circuits, not on parameterized circuit families like ours. For example, in our work, an unsigned n-bit integer addition circuit family is verified once, quickly, for every possible n, 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

[10] The proof of the theorem must suitably handle the existential quantification of course: PFCS does not sweep it under the rug, but provides a structure in which the existential quantification can be handled more easily.

[11] acl2.org/manual topic: r1cs-verification-with-axe

[12] acl2.org/manual topic: semaphore

[13] acl2.org/manual topic: zcash

separately, taking increasing resources as n grows. Another advantage of verifying parameterized circuit families is that the definitions of these families are essentially formal models of the circuit construction libraries and therefore help validate the libraries' design and implementation. In summary, the tradeoff between their approach and our approach is automation versus generality.

The Ecne tool [Wan] 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 auxiliary 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 auxiliary variables is unnecessary for correctness, but it could be addressed with our techniques.

Similarly to our compilation of Leo to R1CS (which forms the context for the work described in this paper), there exist other domain-specific languages for zero-knowledge applications that are compiled to circuits expressed as constraint systems, some of which involve the formal verification of at least some phases of the compilation process: examples of the latter are PinocchioQ [FKL16], Coda [LKL+23], and CirC [OWBB23]. Thus, there are opportunities for cross-fertilization, even though many of the details depend on the specifics of the languages, compilers, and verification tools. The purpose is a little different though: the approach described in this paper is for verifying circuits constructed by libraries like snarkVM, 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 gadget construction libraries.

Work such as [BM23] serves to ensure that the zero-knowledge proof systems work as intended. It is orthogonal to our work, which serves to ensure that gadgets work as intended.

As an additional remark, the notion of existentially quantified circuits (EQCs) [OBW22] is related to the existential quantification of auxiliary variables in PFCS.

7 Future Work

We plan to formalize and verify all the remaining snarkVM gadgets, using the more scalable PFCS form discussed in Section 5, into which we are also migrating the already formalized and verified gadgets. We also plan to formalize and verify PFCS transformations that capture certain optimizations performed by snarkVM, which blend conceptually separate gadgets into gadgets with fewer variables and constraints. To compare the ACL2 PFCS constructions with the snarkVM R1CS constructions, we plan to formalize and verify a PFCS-to-R1CS flattener.

We will use the resulting definitions and theorems to formally verify the correctness of snarkVM's compilation of Aleo instructions to R1CS. Like we mimicked snarkVM's R1CS gadget constructions in ACL2 (Section 3.4), we will mimic snarkVM's translation of Aleo instructions to R1CS in ACL2; this translation will also generate ACL2 correctness theorems between the input Aleo instructions to the output R1CS, which will be proved using the gadget correctness theorems, combined with a formal semantics of Aleo instructions and of encodings of Aleo instruction data types as in Section 2.2. This is a *verifying compiler* approach, as opposed to a *verified compiler* approach: we do not verify the compilation process, but instead we have it generate a theorem each time is run. In order to relate the ACL2-generated and snarkVM-generated gadgets, we will compare them for equivalence each time the compiler is run, similarly to how we are currently testing gadget samples (Section 3), but in a more rigorous and comprehensive way that leaves no gaps in the formal correctness argument.

After that, our next goal is to verify the correctness of the compilation from Leo to Aleo instructions. This work has already started [CWC+21], but needs to be developed much further. We will also use a verifying compiler approach. Once this capability is in place, we will compose it with the one discussed above to obtain an end-to-end formal correctness argument for the whole compilation from Leo to R1CS.

We also plan to apply formal verification to other parts of the Aleo blockchain, including the zero-knowledge proof system, the consensus protocol, and application-level correctness.

8 Conclusion

We have described our ongoing and future work in formalizing and verifying the R1CS gadgets generated by snarkVM, and how that fits in the broader picture of formal verification applied to the Aleo blockchain. Our technical contributions are more general, applicable to other systems that generate R1CS gadgets, and also to systems that use different gadget representations. The notion of PFCS may be of independent interest.

References

[0xP] 0xParc Foundation. Zk bug tracker. `https://github.com/0xPARC/zk-bug-tracker`.

[Alea] Aleo Team. Aleo instructions documentation. `https://developer.aleo.org/aleo`.

[Aleb] Aleo Team. Aleo, where applications become private. `https://aleo.org`.

[AZ16] Jeremy Avigad and Richard Zach. The epsilon calculus. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, summer 2016 edition, 2016. `https://plato.stanford.edu/archives/sum2016/entries/epsilon-calculus/`.

[Bas20] Baseline. Potential security bug with the zk-snark verifier. `https://github.com/ethereum-oasis/baseline/issues/34`, 2020.

[BBC+17] Eli Ben-Sasson, Iddo Bentov, Alessandro Chiesa, Ariel Gabizon, Daniel Genkin, Matan Hamilis, Evgenya Pergament, Michael Riabzev, Mark Silberstein, Eran Tromer, and Madars Virza. Computational integrity with a public random string from quasi-linear PCPs. In *Proceedings of the 36th Annual International Conference on Theory and Application of Cryptographic Techniques*, EUROCRYPT '17, pages 551–579, 2017.

[BCCT12] Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, ITCS '12, pages 326–349, 2012.

[BCG+13] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. SNARKs for C: Verifying program executions succinctly and in zero knowledge. In *Proceedings of the 33rd Annual International Cryptology Conference*, CRYPTO '13, pages 90–108, 2013.

[BCG+14] Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from Bitcoin. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, pages 459–474, 2014.

[BCG+15] Eli Ben-Sasson, Alessandro Chiesa, Matthew Green, Eran Tromer, and Madars Virza. Secure sampling of public parameters for succinct zero knowledge proofs. In *Proceedings of the 36th IEEE Symposium on Security and Privacy*, S&P '15, pages 287–304, 2015.

[BCG+20] Sean Bowe, Alessandro Chiesa, Matthew Green, Ian Miers, Pratyush Mishra, and Howard Wu. Zexe: Enabling decentralized private computation. In *Proceedings of the 2020 IEEE Symposium on Security and Privacy*, SP '20, 2020. `https://eprint.iacr.org/2018/962`.

[BCI+13] Nir Bitansky, Alessandro Chiesa, Yuval Ishai, Rafail Ostrovsky, and Omer Paneth. Succinct non-interactive arguments via linear interactive proofs. In *Proceedings of the 10th Theory of Cryptography Conference*, TCC '13, pages 315–333, 2013.

[BCS16] Eli Ben-Sasson, Alessandro Chiesa, and Nicholas Spooner. Interactive oracle proofs. In *Proceedings of the 14th Theory of Cryptography Conference*, TCC '16-B, pages 31–60, 2016.

[BCTV14] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Succinct non-interactive zero knowledge for a von Neumann architecture. In *Proceedings of the 23rd USENIX Security Symposium*, Security '14, pages 781–796, 2014. Extended version at `http://eprint.iacr.org/2013/879`.

[BFR+13] Benjamin Braun, Ariel J. Feldman, Zuocheng Ren, Srinath Setty, Andrew J. Blumberg, and Michael Walfish. Verifying computations with state. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles*, SOSP '13, pages 341–357, 2013.

[BGG16] Sean Bowe, Ariel Gabizon, and Matthew D. Green. ZCash parameter generation. `https://z.cash/technology/paramgen.html`, 2016. Accessed: 2017-09-28.

[BGG18] Sean Bowe, Ariel Gabizon, and Matthew D. Green. A multi-party protocol for constructing the public parameters of the pinocchio zk-SNARK, 2018.

[BGM17] Sean Bowe, Ariel Gabizon, and Ian Miers. Scalable multi-party computation for zk-snark parameters in the random beacon model. Cryptology ePrint Archive, Report 2017/1050, 2017.

[BM79] Robert S. Boyer and J Strother Moore. *The Boyer-Moore Theorem Prover*. Academic Press, 1979. `https://www.cs.utexas.edu/users/boyer/ftp/nqthm/`.

[BM23] Bolton Bailey and Andrew Miller. Formalizing soundness proofs of SNARKs. Cryptology ePrint Archive, Paper 2023/656, 2023.

[Bow] Sean Bowe. bellman - a crate for building zk-snark circuits. `https://github.com/zkcrypto/bellman`.

[CBBZ22] Binyi Chen, Benedikt Bünz, Dan Boneh, and Zhenfei Zhang. Hyperplonk: Plonk with linear-time prover and high-degree custom gates. Cryptology ePrint Archive, Paper 2022/1355, 2022.

[CFH+15] Craig Costello, Cédric Fournet, Jon Howell, Markulf Kohlweiss, Benjamin Kreuter, Michael Naehrig, Bryan Parno, and Samee Zahur. Geppetto: Versatile verifiable computation. In *Proceedings of the 36th IEEE Symposium on Security and Privacy*, S&P '15, pages 250–273, 2015.

[CHM+19] Alessandro Chiesa, Yuncong Hu, Mary Maller, Pratyush Mishra, Noah Vesely, and Nicholas Ward. Marlin: Preprocessing zksnarks with universal and updatable srs. Cryptology ePrint Archive, Report 2019/1047, 2019. `https://eprint.iacr.org/2019/1047`.

[CWC+21] Collin Chin, Howard Wu, Raymond Chu, Alessandro Coglio, Eric McCarthy, and Eric Smith. Leo: A programming language for formally verified, zero-knowledge applications. Cryptology ePrint Archive, Report 2021/651, 2021. `https://eprint.iacr.org/2021/651.pdf`.

[DFKP13] George Danezis, Cedric Fournet, Markulf Kohlweiss, and Bryan Parno. Pinocchio Coin: building Zerocoin from a succinct pairing-based proof system. In *Proceedings of the 2013 Workshop on Language Support for Privacy Enhancing Technologies*, PETShop '13, 2013.

[DFKP16] Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, and Bryan Parno. Cinderella: Turning shabby X.509 certificates into elegant anonymous credentials with the magic of verifiable computation. In *Proceedings of the 37th IEEE Symposium on Security and Privacy*, S&P '16, pages 235–254, 2016.

[FKL16] Cédric Fournet, Chantal Keller, and Vincent Laporte. A certified compiler for verifiable computing. In *Proc. 29th IEEE Computer Security Foundations Symposium (CSF)*, 2016.

[FL14] Matthew Fredrikson and Benjamin Livshits. Zø: An optimizing distributing zero-knowledge compiler. In *Proceedings of the 23rd USENIX Security Symposium*, Security '14, pages 909–924, 2014.

[GGPR13] Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. Quadratic span programs and succinct NIZKs without PCPs. In *Proceedings of the 32nd Annual International Conference on Theory and Application of Cryptographic Techniques*, EUROCRYPT '13, pages 626–645, 2013.

[GM17] Jens Groth and Mary Maller. Snarky signatures: Minimal signatures of knowledge from simulation-extractable snarks. In *Proceedings of the 37th Conference on the Theory and Applications of Cryptographic Techniques*, CRYPTO '17, pages 581–612, 2017.

[GN20] Hermenegildo García Navarro. Design and implementation of the Circom 1.0 compiler, 2020.

[GPR21] Lior Goldberg, Shahar Papini, and Michael Riabzev. Cairo - a Turing-complete STARK-friendly CPU architecture. Cryptology ePrint Archive, Paper 2021/1063, 2021.

[Gro10] Jens Groth. Short pairing-based non-interactive zero-knowledge arguments. In *Proceedings of the 16th International Conference on the Theory and Application of Cryptology and Information Security*, ASIACRYPT '10, pages 321–340, 2010.

[Gro16] Jens Groth. On the size of pairing-based non-interactive arguments. In *Proceedings of the 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques*, EUROCRYPT '16, pages 305–326, 2016.

[GW11] Craig Gentry and Daniel Wichs. Separating succinct non-interactive arguments from all falsifiable assumptions. In *Proceedings of the 43rd Annual ACM Symposium on Theory of Computing*, STOC '11, pages 99–108, 2011.

[GWC19] Ariel Gabizon, Zachary J. Williamson, and Oana Ciobotaru. Plonk: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge. Cryptology ePrint Archive, Report 2019/953, 2019. `https://eprint.iacr.org/2019/953`.

[HBHW18] Daira Hopwood, Sean Bowe, Taylor Hornby, and Nathan Wilcox. Zcash protocol specification, 2018.

[JKS16] Ari Juels, Ahmed E. Kosba, and Elaine Shi. The ring of Gyges: Investigating the future of criminal smart contracts. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 283–295, 2016.

[KM] Matt Kaufmann and J Strother Moore. The ACL2 theorem prover: Web site. `http://acl2.org`.

[KMS+16] Ahmed E. Kosba, Andrew Miller, Elaine Shi, Zikai Wen, and Charalampos Papamanthou. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In *Proceedings of the 2016 IEEE Symposium on Security and Privacy*, SP '16, pages 839–858, 2016.

[KPP+14] Ahmed E. Kosba, Dimitrios Papadopoulos, Charalampos Papamanthou, Mahmoud F. Sayed, Elaine Shi, and Nikos Triandopoulos. TRUESET: Faster verifiable set computations. In *Proceedings of the 23rd USENIX Security Symposium*, Security '14, pages 765–780, 2014.

[LF19] Daniel Lubarov and Brendan Farmer. R1cs programming: Zk0x04 workshop notes, October 2019.

[Lip12] Helger Lipmaa. Progression-free sets and sublinear pairing-based non-interactive zero-knowledge arguments. In *Proceedings of the 9th Theory of Cryptography Conference on Theory of Cryptography*, TCC '12, pages 169–189, 2012.

[LKL+23] Junrui Liu, Ian Kretz, Hanzhi Liu, Bryan Tan, Jonathan Wang, Yi Sun, Luke Pearson, Anders Miltner, Işıl Dillig, and Yu Feng. Certifying zero-knowledge circuits with refinement types. Cryptology ePrint Archive, Paper 2023/547, 2023.

[MBKM19] Mary Maller, Sean Bowe, Markulf Kohlweiss, and Sarah Meiklejohn. Sonic: Zero-knowledge snarks from linear-size universal and updateable structured reference strings. Cryptology ePrint Archive, Report 2019/099, 2019. https://eprint.iacr.org/2019/099.

[Mic00] Silvio Micali. Computationally sound proofs. *SIAM Journal on Computing*, 30(4):1253–1298, 2000. Preliminary version appeared in FOCS '94.

[MT] Thomas Morgan and Ventali Tan. Formal verification of zk constraint systems. Medium Post.

[NT16] Assa Naveh and Eran Tromer. Photoproof: Cryptographic image authentication for any set of permissible transformations. In *Proceedings of the 2016 IEEE Symposium on Security and Privacy*, SP '16, pages 255–271, 2016.

[OBW20] Alex Ozdemir, Fraser Brown, and Riad S. Wahby. Unifying compilers for snarks, smt, and more. Cryptology ePrint Archive, Report 2020/1586, 2020. https://eprint.iacr.org/2020/1586.

[OBW22] Alex Ozdemir, Fraser Brown, and Riad S. Wahby. CirC: Compiler infrastructure for proof systems, software verification, and more. In *Proc. 43rd IEEE Symposium on Security and Privacy (SP)*, pages 2248–2266, May 2022.

[OKTB23] Alex Ozdemir, Gereon Kremer, Cesare Tinelli, and Clark Barrett. Satisfiability modulo finite fields. Cryptology ePrint Archive, Paper 2023/091, 2023.

[OWBB23] Alex Ozdemir, Riad S. Wahby, Fraser Brown, and Clark Barrett. Bounded verification for finite-field-blasting. In *Proc. 35th International Conference on Computer Aided Verification (CAV), Part III*, volume 13966 of *Lecture Notes in Computer Science*, pages 154–175, July 2023.

[PCW+23] Shankara Pailoor, Yanju Chen, Franklyn Wang, Clara Rodríguez, Jacob Van Gaffen, Jason Morton, Michael Chu, Brian Gu, Yu Feng, and Isil Dillig. Automated detection of underconstrained circuits for zero-knowledge proofs. Cryptology ePrint Archive, Paper 2023/512, 2023.

[PGHR13] Brian Parno, Craig Gentry, Jon Howell, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. In *Proceedings of the 34th IEEE Symposium on Security and Privacy*, Oakland '13, pages 238–252, 2013.

[Sem19] Semaphore. Vulnerability allowing double spend. https://github.com/appliedzkp/semaphore/issues/16, 2019.

[Ste84] Guy L. Steele. *Common Lisp the Language*. Digital Press, 1984.

[STW23] Srinath Setty, Justin Thaler, and Riad Wahby. Customizable constraint systems for succinct arguments. Cryptology ePrint Archive, Report 2021/651, 2023. https://eprint.iacr.org/2023/552.pdf.

[SWB19] Josh Swihart, Benjamin Winston, and Sean Bowe. Zcash counterfeiting vulnerability successfully remediated. https://electriccoin.co/blog/zcash-counterfeiting-vulnerability-successfully-remediated/, 2019.

[Tor19] Tornado19. Tornado.cash got hacked. by us. https://tornado-cash.medium.com/tornado-cash-got-hacked-by-us-b1e012a3c9a8, 2019.

[Wan] Franklyn Wang. Ecne: Automated verification of ZK circuits.

[WSR+15] Riad S. Wahby, Srinath Setty, Zuocheng Ren, Andrew J. Blumberg, and Michael Walfish. Efficient RAM and control flow in verifiable outsourced computation. In *Proceedings of the 22nd Network and Distributed System Security Symposium*, NDSS '15, 2015.

[ZPK14] Yupeng Zhang, Charalampos Papamanthou, and Jonathan Katz. Alitheia: Towards practical verifiable graph processing. In *Proceedings of the 21st ACM Conference on Computer and Communications Security*, CCS '14, pages 856–867, 2014.

A Background

A.1 Prime Fields

If p is a prime number,[14] the set $\mathbb{F}_p = \{0, \ldots, p-1\}$ of the non-negative integers below p forms a *prime field*, whose arithmetic operations are

$$\begin{array}{ll} \text{addition:} & x \oplus_p y = (x + y) \bmod p \\ \text{multiplication:} & x \otimes_p y = (x \times y) \bmod p \end{array}$$

which are modular versions of the ones on the integers.[15] The prime field contains additive and multiplicative inverses (the latter only for non-zero elements), and thus in effect it also has the arithmetic operations

$$\begin{array}{ll} \text{subtraction:} & x \ominus_p y = (x - y) \bmod p \\ \text{division:} & x \oslash_p y = z, \text{ where } x = y \otimes_p z, \text{ if } y \neq 0 \end{array}$$

where subtraction is a modular version of the one on the integers, while the division of x by y is the unique z that yields x when multiplied by y, provided that y is not 0.[16] The prime field operations satisfy many of the same properties as the corresponding integer operations, e.g. addition is commutative and associative.

In this paper, as common in the literature, for brevity we denote the prime field arithmetic operations with the same symbols as the integer arithmetic operations, i.e. $+$, $-$, $\times$, $/$; context should always disambiguate the notation. Also following common practice in arithmetic, we may omit the multiplication symbol altogether, e.g. $(x+1)(y-1)$ stands for $(x+1) \times (y-1)$, which in turns may stand for $(x \oplus_p 1) \otimes_p (y \ominus_p 1)$; the omission should be always clear from context. We may also just write $\mathbb{F}$, leaving p implicit.

A.2 R1CS

A *Rank-1 Constraint System* (*R1CS*) is a finite sequence of *constraints*, each of the form

$$(a_0 + a_1 x_1 + \cdots + a_n x_n)\ (b_0 + b_1 y_1 + \cdots + b_m y_m) = (c_0 + c_1 z_1 + \cdots + c_l z_l)$$

where $n, m, l \geq 0$, each a_i, b_j, c_k is an integer, and each x_i, y_j, z_k is a variable. That is, an R1CS constraint is an equality between the product of two polynomials and a polynomial, where each polynomial is multivariate of degree 1, also called a *linear combination*. The three linear combinations may have variables in common (e.g. some of the x_i variables may be the same as some of the y_j variables).[17] Some of the integer coefficients may be 0, in which case they are normally omitted (along with the variable that they multiply, if any).

The variables in an R1CS range over a prime field $\mathbb{F}$; the addition and multiplication operations are interpreted as $\oplus_p$ and $\otimes_p$. An *assignment* of values (i.e. elements of the prime field) to variables may or may not *satisfy* a constraint (we always assume that assignments cover all the variables in the constraints of interest), i.e. the equation may or may not hold when the variables are replaced by the assigned values and all the additions and multiplications are carried out to reduce each side of the equality to a field element. Given a system (i.e. sequence) of constraints, an assignment may or may not satisfy all the constraints of the system (the relative order of constraints in the sequence does not matter for this). In other words, an R1CS may or may not have *solutions*, i.e. assignments that satisfy all the constraints.

Good examples of how R1CS can represent basic computations are in [LF19] and [HBHW18].

A.3 zkSNARKs

Cryptographic proofs offering strong privacy and efficiency guarantees, known as *zero-knowledge Succinct Non-interactive ARguments of Knowledge (zkSNARKs)*, have received significant interest from academia

[14] A *prime number* is a positive integer that has exactly two distinct divisors, namely 1 and itself. Thus, the prime numbers are 2, 3, 5, 7, 11, etc.

[15] If a and b are integers with $b \neq 0$, there exist unique integers q and r such that $a = b \times q + r$ and $0 \leq r < b$. The *modulus* of a and b is $a \bmod b = r$.

[16] In a prime field, this z always exists and is unique, when $y \neq 0$, because y has a multiplicative inverse in the field.

[17] In fact, other presentations of R1CS in the literature use the same variables in all three polynomials, with the understanding that some (often many, in fact) of the integer coefficients are 0.

and industry [Gro10, Lip12, BCI+13, GGPR13, PGHR13, BCG+13, BFR+13, DFKP13, BCTV14, KPP+14, ZPK14, BCG+14, CFH+15, WSR+15, CFH+15, Gro16, JKS16, KMS+16, NT16, DFKP16, BCS16, GM17, BBC+17, MBKM19, CHM+19, GWC19].

A zkSNARK allows a *prover* to convince a *verifier* of a statement of the form "*given a function F and input x, I know a secret w such that $F(x, w) = \mathsf{true}$*". The notion of a zkSNARK, formulated in [Mic00, GW11, BCCT12], has several definitions, and we consider one known as a *publicly-verifiable preprocessing zkSNARK* (see [BCI+13, GGPR13]). This definition of a zkSNARK consists of three algorithms—a *setup*, a *prover*, and a *verifier*:

- $\mathsf{Setup}(F) \to (\mathsf{pk}_F, \mathsf{vk}_F)$ – The setup takes as input a predicate F and outputs a proving key pk_F and a verification key vk_F. The setup is often considered *trusted* as its intermediate computation steps involve values that must remain secret; however ongoing work, such as [BCG+15, BGG16, BGM17, BGG18], offers mitigations for this requirement. The setup outputs keys $(\mathsf{pk}_F, \mathsf{vk}_F)$ which are used as public parameters. The setup only needs to be run once.
- $\mathsf{Prover}(\mathsf{pk}_F, x, w) \to \pi$ – The prover takes as input the proving key pk_F, a public input x for F, and a private input w for F, and outputs a proof π. The proof π attests to the statement "*given F and x, I know a secret w such that $F(x, w) = \mathsf{true}$*", while revealing no information about w beyond what is implied by the statement. The prover can be run by anyone who wishes to prove their claim to the statement.
- $\mathsf{Verifier}(\mathsf{vk}_F, x, \pi) \to \mathsf{true}/\mathsf{false}$ – The verifier takes as input the verification key vk_F, the public input x for F, and the proof π, and outputs true if the proof is valid and correct, otherwise outputs false. The verifier can be run by anyone who wishes to verify a claim to the statement.

In a zkSNARK, the predicate F is represented as a *zero-knowledge circuit*, i.e. a system of constraints that are equalities over arithmetic expressions interpreted in a prime field (see Section A.1). Depending on the zkSNARK's proof system, the constraints have specific forms, such as the R1CS form (see Section A.2).

zkSNARKs output "short" proofs, in the sense that the size of the proof is logarithmic in the size of the circuit C, i.e. $|\pi| = O_\lambda(\log |C|)$, where λ is a security parameter related to the size of the involved cryptographic keys. zkSNARKs verify proofs "succinctly", in the sense that the time to verify a proof is linear in the size of the public input x and logarithmic in the size of the circuit C, i.e. $t = O_\lambda(|x|, \log |C|)$, where λ is the same security parameter mentioned just above.

zkSNARKs allow applications to provide strong privacy guarantees to users. One example is Zcash [BGG16, BCG+14], a popular cryptocurrency that relies on zkSNARKs as its fundamental technology. Zcash is specifically designed to protect the privacy of users' payments. In this cryptocurrency example, w is the private payment details, x is the encryption of the payment details, and F is a predicate that checks whether x is a valid encryption of w and that w is a legitimate payment.

In recent years, there have been several efforts to develop *preprocessing zkSNARKs* with a *universal* and *updatable* structured reference string (SRS) [MBKM19, CHM+19, GWC19]. Various real-world applications require the SRS to be generated through cryptographic "ceremonies" [BGG16] to ensure that no single party has exclusive control over the SRS sampling process.

Recent work [CHM+19] has demonstrated that generating proofs for an R1CS instance ϕ using a universal SRS achieves practical performance capabilities for both the zkSNARK prover and verifier. It is worth noting that the difference in efficiency between universal SRS and state-of-the-art circuit-specific SRS [Gro16] is competitive. For the purpose of this paper, it suffices to state that there exists performant (*and universal*) proof systems for the circuit-specific SRS to be used in.

A.4 snarkVM

At a high level, *snarkVM* is a virtual machine for zero-knowledge execution which allows users to produce publicly verifiable transactions that attest to the offline computation of programs through the use of succinct, zero-knowledge proofs. Validators can run constant-time verification algorithms to verify these proofs rather than relying on expensive re-executions. Aleo is based on the notion of *decentralized private computation* (DPC), a cryptographic primitive introduced in ZEXE [BCG+20].

Leo [CWC+21] is the programming language of Aleo, a high-level, functional, feature-rich language that allows developers to write general purpose ZK applications. Leo program compilation generates Aleo instructions [Alea], a lower level intermediate representation (IR) language. One of snarkVM's primary

functions is to compile Aleo instructions programs into R1CS. Interpreting programs in Aleo instructions, rather than Leo code, carries several benefits:

- Simplification: Aleo instructions simplify the high-level Leo code into a form that is easier to translate into R1CS gadgets. They remove complex control structures, function calls, etc., and boil down the code to simpler operations.
- Abstraction: Aleo instructions provide a layer of abstraction between the high-level Leo language and the low-level R1CS gadgets. This makes it possible to optimize the generation of R1CS gadgets without having to touch the high-level Leo code.
- Optimization: By having an IR, it is possible to apply optimizations at this level before generating the R1CS gadgets. This can make the resulting zero-knowledge proofs more efficient.
- Analysis: The IR can also be used to analyze the code, for instance to check for certain properties or to optimize the code.

The actual process of generating R1CS happens in the Rust code of snarkVM. First, the virtual machine parses a program into opcodes which are pushed onto a stack data structure. Next, a series of cryptographic checks verifies that the caller is authorized to execute all program functions and value is conserved. Each opcode in Aleo instructions corresponds to an R1CS gadget that can be called to accumulate constraints from the stack. The resulting R1CS is essentially a set of linear equations with a specific form. Although the code may have a hierarchical structure for creating these constraints, the resulting constraint system is flat, with variables assigned with increasing indices. This linear form is important for subsequent steps in the zero-knowledge proof generation process.

It is important to note that this approach to generating constraints-—-compiling from a higher-level language or generating them programmatically [GN20, OBW20, FL14]-—-is not unique to snarkVM or Aleo. This is a common strategy in the area of zero-knowledge proofs and cryptographic distributed systems, which means that the formal verification techniques for snarkVM described in this paper may be broadly extended to other systems.

A.5 ACL2

ACL2 [KM] is a general-purpose interactive theorem prover based on an untyped first-order logic of total functions that is an extension of a purely functional subset of Common Lisp [Ste84]. Predicates are functions and formulas are terms; they are false when their value is `nil`, and true when their value is `t` or anything non-`nil`.

The ACL2 syntax is consistent with Lisp. A function application is a parenthesized list consisting of the function's name followed by the arguments, e.g. $x + 2 \times f(y)$ is written `(+ x (* 2 (f y)))`. Names of constants start and end with `*`, e.g. `*limit*`. Comments extend from semicolons to line endings.

The user interacts with ACL2 by submitting a sequence of theorems, function definitions, etc. ACL2 attempts to prove theorems automatically, via algorithms similar to NQTHM [BM79], most notably simplification and induction. The user guides these proof attempts mainly by (i) proving lemmas for use by specific proof algorithms (e.g. rewrite rules for the simplifier) and (ii) supplying theorem-specific 'hints' (e.g. to case-split on certain conditions).

The factorial function can be defined as

```
(defun fact (n)
  (if (zp n)
      1
    (* n (fact (- n 1)))))
```

where `zp` tests if `n` is 0 or not a natural number. Thus `fact` treats arguments that are not natural numbers as 0. ACL2 functions often handle arguments of the wrong type by explicitly or implicitly coercing them to the right type—since the logic is untyped, in ACL2 a 'type' is just any subset of the universe of values. The function `fact` is defined in the formal logic of ACL2 and thus can be reasoned about (see below); it is also a Lisp function that can be executed.

To preserve logical consistency, recursive function definitions must be proved to terminate via a measure of the arguments that decreases in each recursive call according to a well-founded relation. For `fact`, ACL2

automatically finds a measure and proves that it decreases according to a standard well-founded relation, but sometimes the user has to supply a measure.

A theorem saying that `fact` is above its argument can be introduced as

```
(defthm above
  (implies (natp n)
           (>= (fact n) n)))
```

where `natp` tests if `n` is a natural number. ACL2 proves this theorem automatically (if a standard arithmetic library is loaded), finding and using an appropriate induction rule—the one derived from the recursive definition of `fact`, in this case.

ACL2 provides logical-consistency-preserving mechanisms to axiomatize new functions, such as indefinite description functions. A function constrained to be strictly below `fact` can be introduced as

```
(defchoose below (b) (n)
  (and (natp b)
       (< b (fact n))))
```

where `b` is the variable bound by the indefinite description. This introduces the logically conservative axiom that, for every `n`, `(below n)` is a natural number less than `(fact n)`, if any exists—otherwise, `(below n)` is unconstrained. This function has a logical definition, but does not have an executable Lisp counterpart.

ACL2's Lisp-like macro mechanism provides the ability to extend the language with new constructs defined in terms of existing constructs. For instance, despite the lack of built-in quantification in the logic, functions with top-level quantifiers can be introduced. The existence of a value strictly between `fact` and `below` can be expressed by a predicate as

```
(defun-sk between (n)
  (exists (m)
    (and (natp m)
         (< (below n) m)
         (< m (fact n)))))
```

where `defun-sk` is a macro defined in terms of `defchoose` and `defun`, following a known construction [AZ16].

B Evaluation Details

B.1 Discovering Bugs

By verifying samples of R1CS extracted from snarkVM, we discovered two vulnerabilities in the gadget implementations. We also identified fixes for each of these bugs, and verified that each fix resolved the vulnerability. We describe each of these bugs and their fixes below.

B.1.1 Underconstrained Bitwise Decomposition

We discovered a bug in the bitwise decomposition gadgets: `Field::to_bits_le` and `Scalar::to_bits_le`. Both of these gadgets decompose a field element, either in the base field or in the scalar field, into a sequence of bits. In the original implementation, the bits of the decomposition are provided as witnesses to the gadget. Note that the witness variables are outputs, per the formalism in Section 2. The gadget then verifies that the reconstructed decomposition is equal to the original field element. A general form of this gadget, expressed as constraints, is given in Section 3.4.

In checking samples of these gadgets against the formal specification, we found that these gadgets were unsound and underconstrained. That is, there may exist two distinct assignments to the witness variables that

satisfy the constraints defined by the original gadget. Consider the case where we invoke `Field::to_bits_le` on the zero field element. Two valid assignments to the witness variables are

$$\omega_0 = 0, \omega_1 = 0, \omega_2 = 0, \ldots, \omega_{252} = 0$$

$$\omega_0 = m_0, \omega_1 = m_1, \omega_2 = m_2, \ldots, \omega_{252} = m_{252}$$

where m_i is the ith bit of the prime of the field (the prime that defines our field consists of 253 bits). The first assignment is the bitwise decomposition of the zero field element, and the second assignment is the bitwise decomposition of the prime. Both assignments satisfy the bit constraints, that is, each variable is either zero or one. The first assignment clearly sums to zero, and the second assignment sums to the prime, which wraps around to zero in the prime field.

To resolve this issue, we add constraints requiring that the value of the resulting bits of `Field::to_bits_le` or `Scalar::to_bits_le`, not reduced modulo the prime, is less than the prime. This less-than check is implemented by a bitwise comparison.

B.1.2 Underconstrained Square Root

We discovered a bug in the `Field::sqrt` gadget. This gadget computes the square root of a field element. The gadget witnesses the square root of the field element and verifies that the square of the witness is equal to the original field element. The gadget, defined as constraints, is given below.

$$(\omega_0)\ (\omega_0) \quad = \quad (\iota_0)$$

where ω_0 is the witness variable and ι_0 is the field element to take the square root of.

In checking samples of this gadget against the formal specification, we found that this gadget was also underconstrained. The square root in a field, if it exists, is usually not unique. For example, 1 and $p - 1$, where p is the prime number that defines the field, are both square roots of 1. This may not be a problem if it is used as part of a larger computation, but because there is a `sqrt` Aleo instructions opcode, the gadget that the opcode compiles into must be a deterministic function.

There are a number of possible fixes for this issue. One fix is to require that the witness variable be less than or equal to $(p - 1)/2$. This requires a bitwise comparison, adding significant cost to the gadget.

B.2 Optimizing Existing Gadgets

In defining formal specifications and verifying the snarkVM gadgets, we discovered a number of optimizations. A number of these optimizations have been implemented in snarkVM and verified against the formal specifications. We describe a few of these optimizations below.

B.2.1 Optimizing Unsigned Division

The `Integer::div_wrapped` gadget implements unsigned wrapping division by witnessing the quotient and remainder of the division and enforcing that the quotient and remainder satisfy the constraints of the division:

$$\begin{aligned} (\iota_1) &\neq (0) \\ (\omega_0 * \iota_1 + \omega_1) &= (\iota_0) \\ (\omega_1) &< (\iota_1) \end{aligned}$$

where ι_0 is the dividend, ι_1 is the divisor, ω_0 is the quotient, and ω_1 is the remainder. Note that these constraints are not in the R1CS form, but in an abstracted short-hand. ι_0 and ι_1 are unsigned integers, each of which is a sequence of variables (in an R1CS) constrained to be either zero or one. ω_0 and ω_1 are also unsigned integers, as sequences of bit variables.

The first constraint checks that the divisor is not zero. This is a gadget in itself. The second constraint checks that the quotient and remainder form a valid decomposition of the dividend. The third constraint checks that the remainder is less than the divisor.

Observe that the first constraint is redundant. If the divisor is zero, then the third constraint will fail since the remainder is unsigned integer and must be less than the divisor. We have removed the first constraint from the gadget, which reduces the number of constraints by one.

Originally, for the second constraint, each of ι_0, ι_1, ω_0, and ω_1 was cast into a field element via the `Field::from_bits` gadget, and the check was performed in the field. Due to the size of the field prime, `u128` integers could not use this method as the multiplication in field may wrap around the prime. Instead, the gadget originally performed binary long division directly on the bits of the dividend and divisor. This was a costly operation, and was replaced with a check of the second constraint in the integer domain using the `Integer::mul_checked` and `Integer::add_checked` gadgets. This optimization reduced the cost of the gadget for `u128` integers by an order of magnitude.

B.2.2 Implicitly Implementing Overflow Checks

The `Integer::add_checked` and `Integer::mul_checked` gadgets implement addition and multiplication of unsigned integers, checking for overflow. In both cases, the gadget computes the result in the field and extracts the resulting bits (including the carry bits) from the field elements via `Field::to_bits_le`. The carry bits are checked to be zero and the result bits are constructed into an integer. In the case of checked addition, there is one carry bit and N result bits, where N is the number of bits in the integer. In the case of checked multiplication, there are N carry bits and N result bits, where N is the number of bits in the integer.

To illustrate this optimization, consider the constraints for the `Integer::add_checked` gadget, for `u8`.

$$\begin{aligned}
(\iota_0)\ (1-\iota_0) &= (0)\\
&\dots\\
(\iota_7)\ (1-\iota_7) &= (0)\\
(\iota_8)\ (1-\iota_8) &= (0)\\
&\dots\\
(\iota_{15})\ (1-\iota_{15}) &= (0)\\
(\omega_0)\ (1-\omega_0) &= (0)\\
&\dots\\
(\omega_8)\ (1-\omega_8) &= (0)\\
(\iota_0 + 2\iota_1 + \dots + 128\iota_7 + \iota_8 + 2\iota_9 + \dots + 128\iota_{15})\ (1) &= (\omega_0 + 2\omega_1 + \dots + 256\omega_8)
\end{aligned}$$

The first set of constraints check that the input bits of the first operand, $\iota_0, \dots, \iota_7$, are either zero or one. The second set of constraints check that the input bits of the second operand, $\iota_8, \dots, \iota_15$, are either zero or one. The third set of constraints check that the output bits, $\omega_0, \dots, \omega_7$, and the carry bit, ω_8, are either zero or one. The final constraint checks that the result of the addition in the field is correct. Note that the final constraint is sound as long as the linear combination and the product in the field do not overflow the prime of the field. Therefore, this optimization does not hold for checked unsigned multiplication for `u128` integers.

Instead of explicitly computing the carry bits to check for overflow, we can implicitly check for overflow by checking that the result of the addition in the field domain is equal to the witnessed result in the field domain.

The constraints for the optimized `Integer::add_checked` gadget, for `u8`, are given below.

$$\begin{aligned}
(\iota_0)\ (1-\iota_0) &= (0)\\
&\dots\\
(\iota_7)\ (1-\iota_7) &= (0)\\
(\iota_8)\ (1-\iota_8) &= (0)\\
&\dots\\
(\iota_{15})\ (1-\iota_{15}) &= (0)\\
(\omega_0)\ (1-\omega_0) &= (0)\\
&\dots\\
(\omega_7)\ (1-\omega_7) &= (0)\\
(\iota_0 + 2\iota_1 + \dots + 128\iota_7 + \iota_8 + 2\iota_9 + \dots + 128\iota_{15})\ (1) &= (\omega_0 + 2\omega_1 + \dots + 128\omega_7)
\end{aligned}$$

The final constraint implicitly checks that the carry bit is zero. This optimization saves one constraint in the case of checked unsigned addition. Similar optimizations can be made for checked multiplication, saving cN constraints, for some constant factor c.

Table 1: Constraint counts after fixes and optimizations

Gadget Name	**original # of constraints**	**updated # of constraints**
Field::to_bits_le	254	507
Scalar::to_bits_le	252	503
u16::add_checked	18	16
u32::add_checked	35	33
u64::mul_checked	194	65
u128::mul_checked	1235	328
u128::div_wrapped	16641	845
i128::div_wrapped	17544	1620
u32::pow_checked	7229	5245
u64::pow_checked	14175	10333
u8::shl_checked	1886	1096
u16::shl_checked	3646	2622

B.2.3 Concisely Checking that Bits Are Zero

A number of the integer gadgets check that a sequence of bits are zero. Originally, this check was implemented by applying logical ORs in chain over the bits in question, asserting that the final expression was equal to zero. This implementation requires $O(N)$ constraints, where N is the number of bits checked.

Instead, we can convert each bit into a field element and check that the sum of the field elements is zero. This implementation requires a single constraint but does introduce a number of additional variables. While this reduces overall proving cost, there are some nontrivial costs associated with the additional variables as they increase the density of the R1CS matrices.

B.3 Metrics

Preliminary metrics on changes in the number of constraints after fixes and optimizations are in Table 1.