

Language-Agnostic Detection of Computation-Constraint Inconsistencies in ZKP Programs via Value Inference

Arman Kolozyan^{1,2,3} Bram Vandenbogaerde³ Janwillem Swalens²
Lode Hoste² Stefanos Chaliasos^{4,5} Coen De Roover³

¹Max Planck Institute for Security and Privacy, ²Nokia Bell Labs, ³Vrije Universiteit Brussel,

⁴UCL Centre for Blockchain Technologies, ⁵zkSecurity

arman@kolozyan.com, {bram.vandenbogaerde, coen.de.roover}@vub.be,
{janwillem.swalens, lode.hoste}@nokia-bell-labs.com, stefanos@chaliasos.com

Abstract—Zero-knowledge proofs (ZKPs) allow a prover to convince a verifier of a statement’s truth without revealing any other information. In recent years, ZKPs have matured into a practical technology underpinning major applications. However, implementing ZKP programs remains challenging, as they operate over arithmetic circuits that encode the logic of both the prover and the verifier. Therefore, developers must not only express the computations for generating proofs, but also explicitly specify the constraints for verification. As recent studies have shown, this decoupling may lead to critical ZKP-specific vulnerabilities. Unfortunately, existing tools for detecting them are limited, as they: (1) are tightly coupled to specific ZKP languages, (2) are confined to the constraint level, preventing reasoning about the underlying computations, (3) target only a narrow class of bugs, and (4) suffer from scalability bottlenecks due to reliance on SMT solvers. To address these limitations, we propose a language-agnostic formal model, called the Domain Consistency Model (DCM), which captures the relationship between computations and constraints. Using this model, we provide a taxonomy of vulnerabilities based on computation-constraint mismatches, including novel subclasses overlooked by existing models. Next, we implement an IR-based bug detection tool, called CCC-CHECK, which is based on abstract interpretation. Our evaluation shows that CCC-CHECK is, on average, two orders of magnitude faster than the SoTA verification tool CIVER, while achieving comparable precision. Finally, using the DCM, we examine six widely adopted ZKP projects and uncover 15 previously unknown vulnerabilities. We reported these bugs to the projects’ maintainers, 13 of which have since been patched. Of these 15 vulnerabilities, 12 could not be captured by existing models.

1. Introduction

In today’s digital world, we are often required to prove facts about ourselves, such as our age, by uploading official documents like a passport. However, these documents often reveal far more than the single fact in question. Zero-knowledge proofs (ZKPs) are cryptographic protocols that allow one party (the prover) to convince another party (the verifier) that a statement is true, without revealing anything

beyond the validity of the statement itself [1]. For example, a ZKP makes it possible to prove the claim “I am older than 21” without disclosing an exact age, birth date, or other personal information. ZKPs have evolved from a theoretical curiosity in the 1980s to a practical technology that underpins major applications such as blockchain scalability [2], [3] and identity systems [4], [5]. Their power stems from two properties: zero-knowledge, guaranteeing privacy, and succinctness, allowing short proofs to be verified efficiently.

These applications are usually built using domain-specific languages (DSLs) like Circom [6]. In these languages, developers must write programs that include both the computations, which the prover executes to generate outputs from inputs, and the associated constraints, which (conceptually) the verifier checks to ensure the proof’s validity. These “dual” views significantly increase the complexity of ZKP programs and make them prone to subtle errors. Even a simple comparison checking if a number is less than another requires numerous constraints, and missing just one can lead to a vulnerability. Recent incidents highlight this risk: missing or incorrect constraints have enabled theft of user funds and double-spending attacks [7], [8]. A recent study reveals that over 95% of documented vulnerabilities in ZKP systems arise from such underconstrained programs [9].

To tackle these challenges, various tools have been implemented, employing static analysis [10], [11], dynamic analysis [12], and SMT-based formal verification [13], [14], [15], [16]. However, existing tools suffer from key limitations: they are tied to a single language, target one subclass of underconstrainedness, face scalability issues, and operate exclusively at the constraint level without taking computations into account [13], [15], [17]. To overcome these limitations, we make the following contributions:

- We introduce the *Domain Consistency Model (DCM)*, a formal model that captures the semantic relationship between computations and constraints in ZKP programs. The DCM provides a foundation for reasoning about the values that each side of a ZKP system may take: the computations and the corresponding constraint system.
- We develop a *taxonomy of semantic bug classes*

that arise from inconsistencies between computations and constraints. This taxonomy enables systematic reasoning about soundness violations that go beyond structural or syntactic errors. Built on top of the DCM, we provide an extensible framework for defining new classes of semantic bugs.

- We present *ZKP value inference*, a novel static analysis based on abstract interpretation that makes the DCM practical. By propagating and refining abstract value domains across arithmetic constraints, it efficiently detects semantic inconsistencies without relying on costly SMT reasoning.
- We implement the proposed techniques in a tool: CCC-CHECK (Computation-Constraint Consistency Checker). This tool takes the CIRC intermediate representation (IR) [18] as input, which currently supports ZoKrates, Noir, and Circom as front-ends. Compared to SMT-based approaches, CCC-CHECK is on average two orders of magnitude faster while achieving comparable precision. Using our techniques, we identify 15 previously unknown vulnerabilities across six widely used ZKP projects, 13 of which have been patched.

2. Background & Motivation

2.1. Background

ZKPs are protocols that enable a prover to convince a verifier that a statement is true without revealing anything beyond the validity of the statement [1]. ZKPs became practical with the advent of succinct proof systems such as *zk-SNARKs* (Zero-Knowledge Succinct Non-interactive Arguments of Knowledge) [19]. *zk-SNARKs* are attractive because they produce short proofs that can be verified in time sublinear to the size of the original computation. Typically, *zk-SNARK* systems represent the constraints that the verifier must check to validate a proof as a set of polynomial equations over a finite field, using an intermediate algebraic form such as *Rank-1 Constraint Systems* (R1CS) [20] or Plonk [21]. In SNARKs [19], both the prover and the verifier are generated from the statement description: the prover constructs a proof, while the verifier efficiently checks this proof without re-evaluating every constraint.

To facilitate programming, developers use higher-level *domain-specific languages (DSLs)* designed for ZKP development, such as Circom [6], Gnark [22], or Halo2 [23]. Among these, Circom is currently the most widely used [11]; hence, we will use Circom for the examples and evaluation in this paper. In Circom, developers write programs in two interleaved “views”. On one side, they define how intermediate and output values are computed from the inputs. Simultaneously, they define high-level constraints, which are then compiled into an algebraic form such as R1CS. The core idea is that a prover demonstrates to a verifier that it has executed the specific program correctly by proving that the intermediate values satisfy the constraints. Accordingly, a ZKP program is compiled into both a witness

generator and a constraint system. The witness generator produces the *witness*: a concrete assignment of values to all variables (input/intermediate/output). The constraint system defines the algebraic equations that must hold. The prover combines the witness and the constraints into a short *proof*, while the verifier checks this proof along with the program’s public outputs to determine whether to accept or reject it. Underlying this process is a *proof system*: a mathematical structure ensuring that (1) if the program was executed correctly, then the prover can produce a valid proof (*completeness*), and that (2) if the constraints are not satisfied, a dishonest prover cannot convince an honest verifier otherwise, except with a negligible probability (*soundness*). This process is illustrated in Fig. 1. For an in-depth background on ZKPs, we refer to [24].

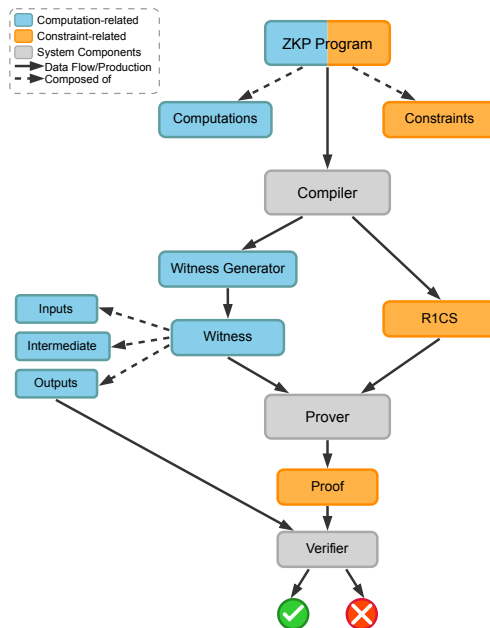


Figure 1: Pipeline of a modern ZKP system.

The soundness of the system heavily relies on the consistency between computations and constraints. If they differ, vulnerabilities can arise. The literature has focused on a specific type of inconsistency: *underconstrained circuits* [9], [13], where the same input can lead to multiple valid outputs. However, as this paper will show, inconsistencies can also stem from more subtle semantic mismatches.

2.2. Motivating Example

In Circom, programs (so-called *circuits*) are defined using *templates*, which are reusable building blocks. Templates can be parameterized and instantiated as *components*, supporting modularity. Inside a template, the developer declares *signals* as inputs, outputs, or as intermediate variables, and specifies their relationships through computations or constraints. Computations, expressed through `<-->`, detail how values are derived from other values and are used by the

```

1 template Divide() {
2   signal input a, b;
3   signal output rem, quot;
4   rem <-- a % b;
5   quot <-- a \ b;
6   a == b * quot + rem;
7 }

```

Listing 1: Division template in Circom with computations in red and the constraints in green.

prover. Constraints, expressed through `===`, define polynomial equalities that must hold for a proof to pass verification. When both sides of an operation are polynomials, Circom also offers the shorthand operator `<==`, which both defines a computation and enforces the corresponding constraint. Additionally, Circom supports `assert` statements to validate run-time conditions during witness generation, which cause the computation to abort if the assertion fails.

The reason for this separation is twofold. First, constraints in ZKPs must reduce to polynomial equations over a finite field because the proof system can handle these. Operations that are not inherently polynomial, like comparisons, need to be expressed through additional constraints that encode them algebraically. Second, performance is critical: developers may omit constraints already enforced elsewhere or replace some constraints with a “cheaper” version.

To illustrate, consider the division template in Listing 1. The developer’s intention is to implement integer division: given inputs a and b , the program computes the quotient `quot` and remainder `rem`. The division and modulus operations are expressed as computations (shown in red), describing how the prover derives `quot` and `rem`. However, since these operations are not polynomial equations, they cannot be checked directly by the verifier. To enable verification, the developer adds a constraint (shown in green), encoding $a = b \cdot \text{quot} + \text{rem}$.

Although this may seem sufficient to guarantee correctness, a problem arises if $b = 0$ in the computations, i.e., a division by zero. Yet, within the constraints, the equation $a = b \cdot \text{quot} + \text{rem}$ is satisfied for any value of `quot` as long as $\text{rem} = a$. This discrepancy shows how programs can become *underconstrained*: while a legitimate prover would never reach this state, a malicious prover can exploit missing constraints to create invalid but accepted proofs.

It is important to note that omitting the check for $b \neq 0$ can be a performance choice. This template might assume that signal `b` has already been constrained to be nonzero. Such assumptions are common in ZKP code due to performance considerations. Circom’s standard library includes many templates that omit checks for performance reasons [25], e.g., comparator templates that assume their inputs can be represented within a fixed number of bits. Further, Circom offers a feature called *signal tags* to make these assumptions explicit, which annotate signals with expected properties [26]. For our example, one could add a tag `nonZero` to signal `b`. Still, signal tags are underutilized as a protection mechanism and mainly serve as documentation;

the compiler does not enforce their correctness [26]. Circom relies on the developer to ensure that all necessary checks are implemented across the program.

2.3. Motivation for Our Approach

First, existing bug detectors for ZKP programs [10], [11], [13], [17] detect “underconstrainedness” (also known as “nondeterminism”). In the literature, an “underconstrained circuit” refers to the case where the same input can lead to multiple valid outputs due to missing constraints that ensure determinism. Yet, the vulnerabilities discussed in this work represent a broader class of *semantic mismatch bugs*, which occur when the computations and the constraints diverge more generally.

Second, detecting such mismatches requires reasoning about the values that each side (computation and constraint) accepts for each variable. For instance, the assignment $a = 2, b = 0, \text{rem} = 2, \text{quot} = 10$ satisfies the constraints but is an invalid computation, as division by zero is undefined.

Third, while SMT-based reasoning is a natural candidate for analyzing such inconsistencies, it faces severe scalability bottlenecks when applied to ZKPs. Modern solvers struggle with non-linear arithmetic over large finite fields, which characterizes ZKP circuits [27], [28], [29], [30]. As a result, verification times may range from minutes to hours for complex circuits, with many analyses timing out on production-scale programs [14]. This motivates the need for alternative approaches based on approximate value reasoning.

Finally, these issues are not unique to Circom; similar vulnerabilities have been identified in other DSLs such as Gnark, ZoKrates, Zirgen, and Noir. The growing diversity of the ZKP landscape calls for techniques that target multiple languages.

These observations motivate the need for a generic and scalable framework to reason about inconsistencies between computations and constraints.

3. Domain Consistency Model

Prior work focuses on nondeterminism instead of broader mismatches between computations and constraints. To address the detection of these mismatches, we introduce the *Domain Consistency Model (DCM)*, a formal framework that generalizes prior definitions [12], [13], [15], [16]. While more recent trace-based definitions consider abnormal program terminations by identifying inconsistencies between execution traces and constraints [12], DCM unifies and extends this model to general *semantic mismatches* between computations and constraints through pre- and postconditions. Its flexibility enables precise SMT-based reasoning and scalable over-approximating static analyses through value-domain inference (Sect. 4).

In Sect. 3.1, we define the core building blocks for our framework. Sect. 3.2 formalizes semantic consistency violations, including underconstrained circuits, overconstrained circuits, as well as a taxonomy of specific bug subclasses.

3.1. Building Blocks

We first introduce the building blocks that enable reasoning formally about vulnerabilities in ZKP programs. Recall that a ZKP program consists of two “views”: the *computation view*, which defines the intended semantics of witness generation, and the *constraint view*, which defines the behavior enforced by the verifier. Ideally, one is the “dual” of the other: the computation view defines how values are calculated, the constraint view checks if this happened correctly. In practice, bugs often originate from inconsistencies between these views [11], [14]. In the next few sections, we formally define these views and establish a framework for identifying and reasoning about such inconsistencies.

Definition 1 (ZKP Program). *A zero-knowledge program [12] is defined as a pair (P, C) where:*

- $P : \mathbb{F}^n \rightarrow (\mathbb{F}^k \times \mathbb{F}^m) \cup \{\perp\}$ is the computation logic (i.e., the witness generation code). It maps an input vector $\vec{x} \in \mathbb{F}^n$ to a pair (\vec{z}, \vec{y}) of intermediate values $\vec{z} \in \mathbb{F}^k$ (i.e., values that are not inputs nor outputs) and outputs $\vec{y} \in \mathbb{F}^m$, or to \perp in case of abnormal termination (e.g., assertion failure).
- $C : \mathbb{F}^n \times \mathbb{F}^k \times \mathbb{F}^m \rightarrow \{\text{true}, \text{false}\}$ is the constraint system, taking inputs \vec{x} , intermediate values \vec{z} , and outputs \vec{y} , and returns *true* if they satisfy all circuit constraints, and *false* otherwise.

Afterwards, the ZKP system compiles P and C into a prover and a verifier: the prover takes inputs and produces both outputs and a succinct proof π , while the verifier checks π against the inputs and outputs. In this work, we focus on the ZKP program itself from the developer’s perspective, aiming to detect bugs in the program logic rather than in the underlying proof system.

Definition 2 (Preconditions). *Preconditions specify semantic assumptions that must hold on the inputs for a ZKP program to behave as intended. A set of preconditions is represented as a set \mathcal{L} of arithmetic formulas (including relational operators) over the program inputs:*

$$\mathcal{L}(\vec{x}) = \{\text{Pre}_1(\vec{x}), \text{Pre}_2(\vec{x}), \dots, \text{Pre}_n(\vec{x})\},$$

where each $\text{Pre}_i(\vec{x})$ is a quantifier-free arithmetic formula (e.g., an inequality or range constraint) over the input vector $\vec{x} = (x_0, x_1, \dots)$.

Preconditions are part of the *computation semantics*, reflecting the developer’s intended behavior, but are not automatically enforced by constraints unless explicitly encoded. They can also be specified externally, for example, through signal tags in Circom such as `nonZero`. While such tags apply to individual signals, pre- and postconditions are more general as they can describe relationships among multiple signals and capture more correctness properties. Ensuring constraints respect preconditions is crucial for soundness.

Definition 3 (Postconditions). *Postconditions, also part of the computation semantics, define the semantic expectations that must hold after execution, typically on intermediate or*

output signals. They express what it means for a program to be correct, assuming that all preconditions were satisfied.

Formally, postconditions are denoted as a set \mathcal{T} of arithmetic formulas:

$$\mathcal{T}(\vec{x}, \vec{z}, \vec{y}) = \{\text{Post}_1(\vec{x}, \vec{z}, \vec{y}), \text{Post}_2(\vec{x}, \vec{z}, \vec{y}), \dots, \text{Post}_m(\vec{x}, \vec{z}, \vec{y})\}$$

where each $\text{Post}_j(\vec{x}, \vec{z}, \vec{y})$ is a quantifier-free arithmetic formula over the input signals \vec{x} , intermediate signals \vec{z} , and output signals \vec{y} .

3.1.1. Value Domains. To reason about mismatches between the computations and the constraint system, we introduce two *computation value domains* and two *constraint value domains*. These domains track the possible values for each signal in the program, either as determined by the computation logic or as restricted by the constraint system.

The computation value domain captures the values that a signal may have during execution of the computation code, taking into account preconditions and postconditions. Initially, a signal is assumed to range over all of \mathbb{F} , unless its domain is restricted by a precondition or postcondition (for example, a specification that it must be nonzero). The constraint value domain captures the values that are accepted by the constraint system.

To reason about executions, we define domain *instances*, which are assignments of values to all the signals in the program. Any program execution can be represented as a domain instance. These instances enable identifying whether a particular combination of values is valid according to the computation logic, the constraints, or both.

Definition 4 (Computation Value Domain). *Let \mathcal{D}_P denote the computation value domain. It maps each signal name s in the program P to the set of values $\mathcal{D}_P(s) \subseteq \mathbb{F}$ that the signal can take during execution of P . More formally, for a signal s in the complete signal tuple $(\vec{x}, \vec{z}, \vec{y}) \in \mathbb{F}^n \times \mathbb{F}^k \times \mathbb{F}^m$, we define:*

$$\begin{aligned} \mathcal{D}_P(s) &= \{\pi_s((\vec{x}, \vec{z}, \vec{y})) \mid \exists \vec{x} \in \mathbb{F}^n, \vec{z} \in \mathbb{F}^k, \vec{y} \in \mathbb{F}^m : \\ &\quad \mathcal{L}(\vec{x}) \wedge P(\vec{x}) = (\vec{z}, \vec{y}) \neq \perp \wedge \mathcal{T}(\vec{x}, \vec{z}, \vec{y})\}, \end{aligned}$$

where π_s denotes the projection onto the component corresponding to signal s in the tuple.

One method of computing \mathcal{D}_P would be to execute the program for every possible combination of inputs $\vec{x} \in \mathbb{F}^n$, and to collect all resulting values for signal s . However, as the field \mathbb{F} is extremely large (typically $p \approx 2^{254}$), exhaustive enumeration is infeasible. Instead, we provide a tractable approximation using abstract interpretation in Sect. 4.

Definition 5 (Conditional Computation Value Domain). *To express computation value domain dependencies, we define:*

$$\begin{aligned} \mathcal{D}_P(s \mid s_1 = v_1, \dots, s_k = v_k) &= \\ \{v \mid \exists (\vec{x}, \vec{z}, \vec{y}) \in \mathbb{F}^n \times \mathbb{F}^k \times \mathbb{F}^m : \mathcal{L}(\vec{x}) \wedge \\ P(\vec{x}) = (\vec{z}, \vec{y}) \neq \perp \wedge \mathcal{T}(\vec{x}, \vec{z}, \vec{y}) \wedge \\ \pi_s((\vec{x}, \vec{z}, \vec{y})) = v \wedge \forall i \in \{1, \dots, k\} : \pi_{s_i}((\vec{x}, \vec{z}, \vec{y})) = v_i\}. \end{aligned}$$

\mathcal{D}_P is the set of values that signal s can take across all executions where the conditioning signals s_1, \dots, s_k are

fixed to their respective values v_1, \dots, v_k . If the program does not produce any value for s under a given set of conditions (e.g., due to an assertion failure, violation of a precondition or a postcondition), then the corresponding conditional value for that signal is the empty set. For programs that, for example, have conditions on the divisor signal being zero, the conditional domain for the quotient signal would be \emptyset since the computation logic aborts on division by zero.

Definition 6 (Constraint Value Domain). *Let \mathcal{D}_C denote the constraint value domain. It maps each signal s to the set of values $\mathcal{D}_C(s) \subseteq \mathbb{F}$ that are accepted by the constraint system C . More precisely, for a signal s in the complete signal tuple $(\vec{x}, \vec{z}, \vec{y}) \in \mathbb{F}^n \times \mathbb{F}^k \times \mathbb{F}^m$, we define:*

$$\mathcal{D}_C(s) = \{\pi_s((\vec{x}, \vec{z}, \vec{y})) \mid \exists (\vec{x}, \vec{z}, \vec{y}) \in \mathbb{F}^n \times \mathbb{F}^k \times \mathbb{F}^m : C(\vec{x}, \vec{z}, \vec{y}) = \text{true}\}.$$

As with \mathcal{D}_P , we soundly over-approximate \mathcal{D}_C using the technique in Sect. 4.

Definition 7 (Conditional Constraint Value Domain). *As before, we also define the notion of conditional constraint value domain:*

$$\begin{aligned} \mathcal{D}_C(s \mid s_1 = v_1, \dots, s_k = v_k) &= \{\pi_s((\vec{x}, \vec{z}, \vec{y})) \mid \\ \exists (\vec{x}, \vec{z}, \vec{y}) \in \mathbb{F}^n \times \mathbb{F}^k \times \mathbb{F}^m : C(\vec{x}, \vec{z}, \vec{y}) &= \text{true} \wedge \\ \forall i \in \{1, \dots, k\} : \pi_{s_i}((\vec{x}, \vec{z}, \vec{y})) &= v_i\}. \end{aligned}$$

Example. Consider the division template from our motivating example in Listing 1. For $a = 7$ and $b = 3$, $\mathcal{D}_P(\text{quot} \mid a=7, b=3) = \{2\}$ and $\mathcal{D}_P(\text{rem} \mid a=7, b=3) = \{1\}$. However, if $b = 0$, the computation logic would abort due to division by zero, resulting in $\mathcal{D}_P(\text{quot} \mid a=7, b=0) = \emptyset$ and $\mathcal{D}_P(\text{rem} \mid a=7, b=0) = \emptyset$. In contrast, the constraint value domain \mathcal{D}_C only considers the constraint $a === b * \text{quot} + \text{rem}$. For $a = 7$ and $b = 0$, this constraint becomes $7 === 0 * \text{quot} + \text{rem}$, which is satisfied by any value of quot as long as $\text{rem} = 7$. Hence $\mathcal{D}_C(\text{quot} \mid a=7, b=0) = \mathbb{F}$ and $\mathcal{D}_C(\text{rem} \mid a=7, b=0) = \{7\}$.

3.1.2. Domain Instance. A *domain instance* is a mapping (i.e., every signal is mapped to a single value) $\nu : \mathcal{S} \rightarrow \mathbb{F}$, where \mathcal{S} is the set of all signal names in the program. If the signal names are sorted in a fixed order (e.g., lexicographically), then a domain instance ν can equivalently be represented as a tuple of values (v_1, v_2, \dots, v_n) , where each $v_i = \nu(s_i)$ for the i -th signal $s_i \in \mathcal{S}$. This representation simplifies notation when signal names are not relevant.

Definition 8 (Valid Computation Domain Instance). *We say that ν is a valid computation domain instance if and only if: $\forall s \in \mathcal{S}, \nu(s) \in \mathcal{D}_P(s \mid s_1 = \nu(s_1), \dots, s_k = \nu(s_k))$ for all other signals $s_1, \dots, s_k \in \mathcal{S} \setminus \{s\}$.*

In other words, the value assigned to each signal must lie within the allowed set given all other signal values, according to the computation logic.

Definition 9 (Valid Constraint Domain Instance). *Similarly, ν is a valid constraint domain instance if and only if: $\forall s \in$*

$\mathcal{S}, \nu(s) \in \mathcal{D}_C(s \mid s_1 = \nu(s_1), \dots, s_k = \nu(s_k))$, for all other signals $s_1, \dots, s_k \in \mathcal{S} \setminus \{s\}$.

Using these definitions, we define \mathcal{V}_P and \mathcal{V}_C as the set of all valid computation domain instances and valid constraint domain instances, respectively. These sets represent the values permitted by the computations and constraints.

Example. For the division template from Listing 1, $\nu = \{a \mapsto 7, b \mapsto 0, \text{quot} \mapsto 42, \text{rem} \mapsto 7\}$ would be a valid constraint domain instance (since it satisfies the constraint), but not a valid computation domain instance (since division by zero is undefined in the computation logic).

3.2. Semantic Consistency Violations

We now formalize the notion of semantic consistency violations, which occur due to mismatches between the computation logic and the constraint system. These violations can be categorized into two main classes: underconstrained circuits and overconstrained circuits. Our value-based formalization enables characterizing these main classes, as well as more specific subclasses that arise in practice.

3.2.1. Underconstrained Circuits. We define a circuit as *underconstrained* if its constraints accept values inconsistent with the program's intended behavior. This occurs when the constraints admit signal values that do not respect the computation code, preconditions, and postconditions.

The conditional value domains introduced in Sect. 3.1.1 enable a more precise definition. Let (P, C) be a ZKP program with preconditions \mathcal{L} and postconditions \mathcal{T} . Let $\vec{x} \in \mathbb{F}^n$ denote the input signals of the program, and $\vec{y} \in \mathbb{F}^m$ the output signals computed from those inputs. We say that the circuit is *underconstrained* if, for some input \vec{x} , the constraint system permits output values that do not correspond to the program's intended behavior. This can manifest in two, potentially overlapping, cases:

(i) **Nondeterministic output:** For a given input \vec{x} , there exists an output signal $o \in \vec{y}$ such that the constraint system allows multiple possible values for o . This means that the constraints do not enforce a unique output for that input: $\exists \vec{x} \in \mathbb{F}^n, \exists o \in \vec{y} : |\mathcal{D}_C(o \mid \vec{x})| > 1$. Recall that ZKPs require the prover to submit both the proof and the claimed output and public inputs. If the constraints do not uniquely tie the output to the input, a malicious prover can generate a proof for one valid output A , but then falsely claim that the output was B , and the verifier would still accept the proof.¹

(ii) **Phantom output:** For a given input \vec{x} , there exists an output signal $o \in \vec{y}$ such that the constraint system accepts values that the program does not produce for that input. Formally: $\exists \vec{x} \in \mathbb{F}^n, \exists o \in \vec{y} : |\mathcal{D}_C(o \mid \vec{x})| \geq 1 \wedge \mathcal{D}_C(o \mid \vec{x}) \cap \mathcal{D}_P(o \mid \vec{x}) = \emptyset$. This encompasses two scenarios: (a) the program produces no output for \vec{x} (i.e., $\mathcal{D}_P(o \mid \vec{x}) =$

1. While nondeterminism typically indicates a bug, there are rare cases where it may be intentional. For example, a circuit computing square roots may deliberately allow both positive and negative roots. Such intentional nondeterminism occasionally occurs in reusable circuit components, but is very rare for top-level circuits [13].

\emptyset), which occurs if the input violates preconditions \mathcal{L} , the program aborts during execution (e.g., due to an assertion), or the output violates postconditions \mathcal{T} ; or (b) the program and constraints both produce outputs, but they are disjoint.

3.2.2. Overconstrained Circuits. A circuit is said to be *overconstrained* if the constraint system rejects a value assignment that is semantically valid according to the program. In other words, the constraint system is too strict.

Let \mathcal{V}_P be the set of all valid computation instances as defined by the computation value domain \mathcal{D}_P , and let \mathcal{V}_C be the set of all valid constraint instances as defined by the constraint value domain \mathcal{D}_C . The circuit is *overconstrained* iff $\mathcal{V}_P \not\subseteq \mathcal{V}_C$. That is, there exists at least one domain instance ν that the program would consider valid, but is denied by the constraint system. Overconstrained bugs are less severe, as they do not allow invalid proofs to be accepted. Instead, they cause the system to reject valid proofs.

3.2.3. Taxonomy of Inconsistency Subclasses. Our value-based formalization of underconstrained circuits enables characterizing a wide range of bug subclasses that arise in practice. As noted earlier, prior work focuses primarily on nondeterministic outputs, missing vulnerabilities such as division-by-zero errors, which Wen et al. [11] report to account for approximately 40% of detected bugs. These bugs are not necessarily related to output ambiguity but instead arise because the denominator is not properly constrained to be nonzero. Although such issues clearly represent a mismatch between the intended program semantics and what the constraint system enforces, they are not considered in any of the existing definitions of underconstrainedness.

Our formalization addresses this limitation. It treats a signal as underconstrained not only when the output is ambiguous, but also when the constraint system permits values that would cause the program to abort or behave incorrectly due to violations of preconditions or postconditions.

In the remainder of this section, we introduce six subclasses of underconstrainedness that have not been supported by existing formalizations, explain how they are captured by our model, and illustrate each with a concrete example. While these subclasses may appear similar at a “symptom” level (unexpected values), they differ in what semantic assumption is being violated, the causing code pattern, and, consequently, what a detector should look for.

SCV 3.1: Arithmetic Overflows and Underflows. Arithmetic overflows and underflows occur when a signal is assumed to lie within a certain range, but the constraint system does not enforce this assumption. In an overflow, the signal may exceed the field modulus p , causing it to wrap around and to be interpreted as a small value. Symmetrically, an underflow occurs when a subtraction wraps below zero, producing a large value near p .

This vulnerability commonly appears in range-checking templates such as `LessThan(n)` from the standard Circom library, which assumes but does not check that its inputs are $\leq 2^n$. When range constraints are missing, an

attacker can provide out-of-bounds values that exploit modular arithmetic to bypass intended checks. For instance, in a withdrawal circuit, this could allow withdrawing more money than available in an account. Protection against arithmetic overflows and underflows can be achieved through the use of signal tags (such as `maxbit`) combined with range constraints. In our formalization, the computation logic would assume bounded inputs (e.g., through `maxbit` tags), leading to $\mathcal{D}_P(\text{signal} \mid \text{out-of-range input}) = \emptyset$. Still, the constraint system may accept these values: $|\mathcal{D}_C(\text{signal} \mid \text{out-of-range input})| \geq 1$. This case constitutes a *phantom output* and thus an underconstrained circuit. See Appendix A for a detailed example.

SCV 3.2: Division by Zero. Division-by-zero bugs arise when a denominator expression is assumed to be nonzero by the computation logic, but this assumption is not enforced by the constraint system. Such bugs result in undefined or underconstrained outputs, which allows attackers to inject arbitrary values in the circuit while satisfying all constraints.

This pattern frequently occurs in practice since division cannot be expressed directly as a constraint in DSLs such as Circom. This was illustrated by the motivating example from Listing 1. Developers can use signal tags (e.g., `nonZero`) combined with explicit nonzero constraints to prevent such issues. When the denominator is zero, the computation aborts, meaning there is no valid output, yet the constraint system still admits values. This constitutes a case of *phantom output*. Moreover, since the constraint system accepts multiple output values for the same input, it also qualifies as *nondeterministic output*.

SCV 3.3: Circuit Logic Assumptions. Some templates rely on semantic assumptions about their inputs to behave correctly. A common case is when arithmetic formulas are used to implement logical operators, such as `AND` or `XOR`.

For example, the `XOR` template from `circomlib` uses the formula `out <== a + b - 2ab`, which computes the correct result only for binary inputs $a, b \in \{0, 1\}$. However, when non-binary values are passed, the formula evaluates to a result that satisfies the constraint but contradicts the expected semantics.

In our formalization, when inputs violate the binary assumption (e.g., $a = 2$ and $b = 2$), the computation logic expects rejection ($\mathcal{D}_P(\text{out} \mid a = 2, b = 2) = \emptyset$), but the constraint system still accepts the values ($|\mathcal{D}_C(\text{out} \mid a = 2, b = 2)| = 1$). This results in a *phantom output*.

SCV 3.4: Special Signals. Some signals are semantically required to avoid specific values (e.g., zero). For example, a signal might represent a group element scalar or an inverse, where setting it to zero would invalidate the computation or allow trivial proofs. These assumptions are often implicit and may be annotated via tags such as `nonZero`.

An example of this bug class is described in the Spartan-ECDSA audit report [31]. The circuit in question computes the public key as follows: `pubKey <== s*T + U`. This assumes that the input signal s , a scalar multiplier, is nonzero. However, no constraint enforces this assumption. This opens the door to a trivial exploit: by setting s to zero and choosing $U = \text{pubKey}$, the entire scalar multiplication is nullified

and the constraint is trivially satisfied for every T . This allows a malicious prover to generate a seemingly valid proof without performing a proper signature computation.

This is reflected in the computation domain²: $\mathcal{D}_P(\text{out} \mid s = 0) = \emptyset$, but the constraint system still permits such values: $|\mathcal{D}_C(\text{out} \mid s = 0, U = \text{pubKey})| = 1$. This results in a *phantom output*, and thus in an underconstrained circuit.

SCV 3.5: Out-of-Bounds Indexing. When indexing into an array, the program assumes that the index lies within the valid bounds of that array. This assumption is an implicit precondition: if an array has length n , then an index signal i should satisfy $0 \leq i < n$. The Circom compiler detects such out-of-bounds accesses statically and refuses to compile the circuit. Furthermore, Circom does not allow signal-based indexing altogether (e.g., `in[a]`), since the actual value of a is not known at compile time.

However, other ZKP DSLs such as ZoKrates take a different approach. ZoKrates allows dynamic indexing using signal values and inserts additional constraints into the R1CS to ensure that the index lies within the valid bounds. While this aims to enforce safety at the constraint level, it relies on the correctness of the automatically generated checks. Moreover, such issues could be detected “earlier” in the pipeline and more efficiently, avoiding the need for extra constraints and the associated performance loss.

When an index is out of range, the computation logic would reject the execution ($\mathcal{D}_P(\text{out} \mid \text{out-of-range index}) = \emptyset$), but the constraint system may still accept such values ($|\mathcal{D}_C(\text{out} \mid \text{out-of-range index})| \geq 1$). This mismatch creates a *phantom output*, and therefore constitutes an underconstrained circuit in our framework.

SCV 3.6: Conditionals with Non-Binary Guard. In Circom, signals cannot be used as the condition in an `if` statement. Thus, conditionals are typically expressed using arithmetic formulas such as `out <== cond*a+(1-cond)*b`. This behaves as expected only if `cond` is binary.

A program can reject non-binary guards ($\mathcal{D}_P(\text{out} \mid \text{cond} = \text{non-binary}) = \emptyset$), but the constraint system may still accept them when corresponding constraints are missing ($|\mathcal{D}_C(\text{out} \mid \text{cond} = \text{non-binary})| \geq 1$). Such an example constitutes a *phantom output*.

4. CCC-Check

The DCM provides a precise, semantic characterization of the discrepancies between computations and constraints. However, in practice, directly applying the DCM to find such inconsistencies is infeasible. Calculating the exact set of reachable values for a signal would require iterating over all possible values of each signal, which means exploring the entire prime field \mathbb{F}_p . Since p is large, such enumeration is not scalable. This is one reason why SMT solvers tend to perform poorly in the ZKP context [32].

To make this analysis tractable, we employ a type of *abstract interpretation* [33]. We call this technique *value inference*, which calculates sound over-approximations of

2. We assume here that “out” denotes whether the check was successful.

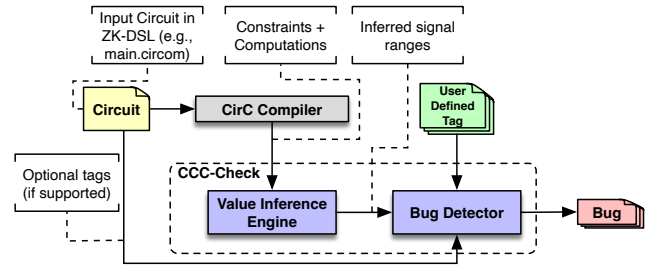


Figure 2: Overview of CCC-CHECK. Tags may be embedded in the program or supplied by the user.

run-time values instead of calculating them exactly. This enables static reasoning about the possible value ranges of signals and allows efficient checking of user-defined tags.

The main idea is to propagate and refine abstract value domains throughout the program’s code using a set of inference rules that capture common patterns found in ZKP circuits, such as arithmetic equalities. Each rule updates the possible value domain of a signal based on the semantics of the constraint in which it appears, gradually narrowing down the set of feasible values. By checking that the inferred domains satisfy the semantics of user-provided signal tags, we can identify vulnerabilities where constraints do not enforce the intended preconditions. Hence, CCC-CHECK extracts developer specifications (via signal tags) from the computations, and then checks whether the constraints enforce it. Additionally, CCC-CHECK inspects the computation to flag division-by-zero and out-of-bounds array indexing issues. This workflow enables the detection of the inconsistency classes described in Sect. 3.2.3.

This section explains how the value inference process works, and its application in bug detection using signal tags. We first present the fixpoint algorithm that guides the propagation, then define the abstract value domains, followed by the inference rules used to refine these domains. We conclude by explaining how the inferred domains are checked against tag specifications to detect vulnerabilities. Figure 2 illustrates the overall architecture of CCC-CHECK (Computation-Constraint Consistency Checker).

4.1. Value Inference Engine

4.1.1. Algorithm. The inference engine performs a monotone fixpoint computation over the constraints of the CIR intermediate representation (IR) of the input program. The analysis maintains a *constraint value domain* for each variable, representing the set of possible values that the variable can take. The algorithm starts with imprecise information and gradually refines it as constraints are discovered. When new information about a signal is obtained, the algorithm computes the meet (\sqcap) of its current abstract domain with the abstract domain derived from the new constraint to narrow the variable’s domain. A worklist of constraints guides the analysis: whenever a variable’s domain is refined, all constraints depending on this variable are added to the worklist for re-evaluation. These value domains are

Procedure: Infer

Require: Set of constraints \mathcal{C} , variable-to-constraint mapping $\mathcal{V} \rightarrow 2^{\mathcal{C}}$

Ensure: Variable states $\mathcal{S} : \mathcal{V} \rightarrow \mathcal{D}$ where \mathcal{D} represents value domains

```

1: Initialize variable domains  $\mathcal{S}$  to  $[0, p - 1]$ 
2: Apply input specification to refine  $\mathcal{S}$ 
3:  $\mathcal{W} \leftarrow \mathcal{C}$ 
4: while  $\mathcal{W} \neq \emptyset$  do
5:    $c \leftarrow \text{dequeue}(\mathcal{W})$ 
6:    $\text{changedVars} \leftarrow \emptyset$ 
7:   for variable  $v \in \text{Variables}(c)$  do
8:      $\mathcal{D}_{\text{new}} \leftarrow \text{InferDomain}(v, c, \mathcal{S})$ 
9:      $\mathcal{D}_{\text{refined}} \leftarrow \mathcal{S}[v] \sqcap \mathcal{D}_{\text{new}}$ 
10:    if  $\mathcal{D}_{\text{refined}} = \emptyset$  then
11:      report inconsistency
12:    end if
13:    if  $\mathcal{D}_{\text{refined}} \neq \mathcal{S}[v]$  then
14:       $\mathcal{S}[v] \leftarrow \mathcal{D}_{\text{refined}}$ 
15:       $\text{changedVars} \leftarrow \text{changedVars} \cup \{v\}$ 
16:    end if
17:  end for
18:  for variable  $v \in \text{changedVars}$  do
19:     $\mathcal{W} \leftarrow \mathcal{W} \cup \{c' \in \mathcal{C} : v \in \text{Variables}(c')\}$ 
20:  end for
21: end while
22: return  $\mathcal{S}$ 

```

Figure 3: Value Inference Algorithm

iteratively refined by propagating constraints and applying inference rules until the worklist is empty and no further refinement is possible.

The pseudocode is shown in Algorithm 3. It takes as input a set of constraints \mathcal{C} and a mapping from each variable to the constraints that reference it. The output is a map \mathcal{S} that assigns each variable a value domain \mathcal{D} of possible values it can take. The algorithm proceeds as follows:

(i) **Initialization:** The domain $\mathcal{S}[v]$ of each variable v is initially set to the full field range $[0, p - 1]$. If the circuit contains input specifications such as signal tags, these are applied at this stage to refine the initial domains. This is reasonable since the circuit assumes these properties about its signals for correct execution; it is up to the “caller” to provide constraints enforcing these assumptions.

(ii) **Worklist:** Constraints in \mathcal{C} are placed in a worklist \mathcal{W} .

(iii) **Propagation:** Until the worklist is empty, the algorithm dequeues constraint c and refines the domains of its variables. To this end, the algorithm computes a new candidate domain \mathcal{D}_{new} using a function $\text{InferDomain}(v, c, \mathcal{S})$, which applies a set of inference rules to derive a candidate domain for variable v from constraint c and the current variable domains \mathcal{S} . We discuss these rules and the function in more detail in Sect. 4.1.3. Finally, before the end of the propagation phase, the current domain $\mathcal{S}[v]$ is refined via the meet: $\mathcal{S}[v] \sqcap \mathcal{D}_{\text{new}}$. If the refined domain differs from the current one (i.e., it is strictly more precise, since the

meet can only tighten a domain), the change is recorded into the inferred domains \mathcal{S} .

(iv) **Re-queuing:** For each variable whose domain was refined, all constraints referencing that variable are added back to \mathcal{W} , as the refinement may allow further propagation in those constraints. Note that only the affected constraints are re-queued for performance reasons, as constraints unrelated to the refined variables cannot yield additional information.

(v) **Termination:** The algorithm stops when \mathcal{W} is empty, yielding the final inferred domains \mathcal{S} .

This design enables the engine to transitively propagate information from multiple constraints. For instance, if $x = y$ and $y \in \{0, 1\}$, the analysis will determine that $x \in \{0, 1\}$ even if no other constraint directly restricts x . If domain refinement produces an empty set (line 10), this indicates an inconsistency: the constraints impose mutually incompatible requirements on a variable, meaning no value can satisfy all constraints simultaneously.

Termination Guarantee. The refinement process is monotonically decreasing: Algorithm 3 only records a domain update when the refined domain differs from the current one (line 13). Since the meet always returns a subset of both operands, a changed result is necessarily strictly smaller, ensuring that domains only shrink or remain unchanged. This property, along with the fact that there are only finitely many possible domains for each variable (i.e., field \mathbb{F}_p is finite), satisfies the descending chain condition, which guarantees that the analysis will eventually reach a fixpoint and terminate.

4.1.2. Value Domains. Each variable in the IR is associated with an abstract *value domain*, representing the set of possible values it can assume. We define three domain types:

KnownValues(V). A finite, explicitly enumerated set of values $V = \{v_1, \dots, v_n\}$. This is used when the analysis can determine the exact possible values a variable may take. For instance, instead of representing a variable’s domain as the coarse interval $[0, 75]$, we might know that it can only take the values $\{0, 5, 75\}$, which is more precise.

BoundedValues(lb, ub, G). An interval $[lb, ub]$ (modulo p) with a set of excluded sub-intervals G (*gaps*). This allows compact representation of non-contiguous (e.g., $\text{BoundedValues}(0, 9, \{[2, 7]\})$ for $\{0, 1, 8, 9\}$).

ArrayDomain(M, D, s). For array signals, M maps specific indices to their element domains, D is the default domain for unspecified indices, and s is the array size.

The choice of domain is determined dynamically by the inference rules (e.g., `ROOT` yields `KnownValues`, `BASE-CONV` yields `BoundedValues`). The meet (\sqcap) and arithmetic operations handle mixed-domain combinations soundly; Appendix B details their implementation.

4.1.3. Inference Rules. The value inference engine refines value domains through a collection of *inference rules*. Each rule formalizes a recurring pattern commonly found in ZKP programs and is used to propagate and refine value domains during analysis. Conceptually, these rules build upon the

(a) Basic Expression / Arithmetic Rules

$$\begin{array}{c}
\frac{}{\nu \vdash x : \nu(x)} \text{(VAR)} \quad \frac{\text{Constant}(c)}{\nu \vdash c : \{c\}} \text{(CONST)} \\
\frac{\nu \vdash e : \Omega \quad \nu[x \mapsto \Omega] \vdash b : \Omega_b}{\nu \vdash \text{let } x = e \text{ in } b : \Omega_b} \text{(LET)} \quad \frac{\nu \vdash e_{\text{then}} : \Omega_1 \quad \nu \vdash e_{\text{else}} : \Omega_2}{\nu \vdash \text{ite}(c, e_{\text{then}}, e_{\text{else}}) : \Omega_1 \sqcup \Omega_2} \text{(ITE)} \\
\frac{\nu \vdash e_{1,2} : \Omega_{1,2}}{\nu \vdash e_1 \oplus e_2 : \{(v_1 \oplus v_2) \bmod p \mid (v_1, v_2) \in \Omega_1 \times \Omega_2\}} \text{(ADD/SUB/MUL)}
\end{array}$$

(b) Equation / Boolean Constraint Rules

$$\begin{array}{c}
\frac{\nu \vdash x : \Omega_x \quad \nu \vdash y : \Omega_y \quad x = y \in C}{\nu, C \vdash x, y : \Omega_x \sqcap \Omega_y} \text{(EQUALITY)} \quad \frac{\nu \vdash e : \Omega \quad c \neq 0 \quad c \times x - e = 0 \in C}{\nu, C \vdash x : \{v \times c^{-1} \bmod p \mid v \in \Omega\}} \text{(ASSIGN)} \\
\frac{\prod_{i=1}^n (x - c_i) = 0 \in C}{\nu, C \vdash x : \{c_1, \dots, c_n\}} \text{(ROOT)} \quad \frac{\sum_{i=0}^n c^i \times y_i = x \in C \quad c > 1 \quad \forall i. \nu(y_i) \subseteq [0, c-1]}{\nu, C \vdash x : [0, c^{n+1} - 1]} \text{(BASE-CONV)} \\
\frac{x + y = 1 \in C \quad x \in \{0, 1\}}{\nu, C \vdash y : \{0, 1\}} \text{(ZERO-ONE)} \quad \frac{a, b \in \{0, 1\} \quad \text{out} = a \times b \in C}{\nu, C \vdash \text{out} : \{0, 1\}} \text{(BOOL-AND)}
\end{array}$$

Analogous rules for OR, XOR, NOT, NAND, and NOR are given in Appendix C.

Figure 4: Summary of inference rules for value domain propagation.

ones provided by Pailoor et al. [13] and describe how the domain of one signal can be restricted based on the domains of others and on the semantics of the pattern that connects them. More specifically, we added rules to handle more expressive IR constructs (e.g., LET and ITE), rules to detect division-by-zero errors (e.g., NONZERO-PRODUCT), and array-family rules to detect out-of-bounds array indexing (e.g., ARRAY-SELECT). We present a rich set of rules in Fig. 4, and discuss implementation details in Appendix B.

To describe the rules formally, we denote $\nu, C \vdash e : \Omega$ to mean: “under domain instance ν and constraint set C , the expression e has domain Ω .” Here, ν is a mapping from variables to their inferred value domains, and C is the set of known constraints. Note that we omit C from the notation when it is not relevant to the specific rule. Using this notation, we can now formally define the `InferDomain` function used in Algorithm 3:

$$\text{InferDomain}(v, c, \mathcal{S}) = \prod_{\mathcal{S}, \{c\} \vdash v : D} D$$

That is, `InferDomain` pattern-matches c against the rules in Fig. 4 and applies the most specific matching rule. It is important to note that not every constraint necessarily updates every variable it references; `InferDomain` only considers variables whose domains can be refined by the specific semantics of the matching rule.

Basic Expression and Arithmetic Rules. Fig. 4a presents rules for basic expressions and arithmetic. The VAR, CONST, and LET rules are straightforward. The ITE rule soundly over-approximates by taking the join of both branches’ domains, since the actual execution will follow

(c) NonZero / Array Operation Rules

$$\begin{array}{c}
\frac{e_1 \times e_2 = c \in C \quad c \neq 0}{\nu, C \vdash e_1 : \nu(e_1) \setminus \{0\}, e_2 : \nu(e_2) \setminus \{0\}} \text{(NONZERO-PRODUCT)} \\
\text{Variants NONZERO-SUM and NONZERO-CONST are given in Appendix C.} \\
\frac{\nu \vdash \text{arr} : \text{ArrayDomain}(M, D, s) \quad \nu \vdash \text{idx} : \{i\} \quad 0 \leq i < s}{\nu \vdash \text{select}(\text{arr}, \text{idx}) : \begin{cases} M(i) & M(i) \neq \emptyset \\ D & \text{otherwise} \end{cases}} \text{(ARRAY-SELECT)} \\
\frac{\nu \vdash e_1, \dots, e_n : \Omega_1, \dots, \Omega_n}{\nu \vdash \#(\vec{e}) : \text{ArrayDomain}(\{i \rightarrow \Omega_{i+1}\}, [0, p-1], n)} \text{(ARRAY-LITERAL)} \\
\frac{\nu \vdash \text{arr} : \text{ArrayDomain}(M, D, s) \quad \nu \vdash \text{val} : \Omega \quad \nu \vdash \text{idx} : \{i\} \quad 0 \leq i < s}{\nu \vdash \text{store}(\text{arr}, \text{idx}, \text{val}) : \text{ArrayDomain}(M[i \mapsto \Omega], D, s)} \text{(ARRAY-STORE)} \\
\frac{\nu \vdash \text{val} : \Omega}{\nu \vdash \text{fill}(\text{val}, s) : \text{ArrayDomain}(\emptyset, \Omega, s)} \text{(ARRAY-FILL)}
\end{array}$$

exactly one branch. The ADD/SUB/MUL rules enumerate all pairwise combinations under modular arithmetic. They are sound as they consider all possible combinations of values from the operand domains.

Equation Constraint Rules. The rules in Fig. 4b refine domains using equality constraints. The EQUALITY rule is sound because if $x = y$, then both variables must take the same value, so their domains can be safely refined via the meet. The ASSIGN rule handles constraints of the form $c \times x = e$ where $c \neq 0$: it computes x ’s domain by dividing each possible value of e by c modulo p . This is sound because if $c \times x = v$ for some $v \in \Omega_e$, then $x = v \times c^{-1} \bmod p$. The ROOT rule applies when a polynomial $\prod_{i=1}^n (x - c_i)$ equals zero. This constraint is satisfied if and only if x equals one of the roots c_i , making the restriction to $\{c_1, \dots, c_n\}$ both sound and precise. The BASE-CONV rule handles base conversion constraints: if $x = \sum_{i=0}^n c^i \times y_i$ and each digit $y_i \in [0, c-1]$, then x is bounded by the maximum representable value $c^{n+1} - 1$. This gives a sound upper bound on x . Finally, the ZERO-ONE rule exploits the constraint $x + y = 1$ when x is binary: since $x \in \{0, 1\}$, we must have $y = 1 - x \in \{1 - 0, 1 - 1\} = \{0, 1\}$.

Boolean Constraint Rules. The boolean rules in Fig. 4b encode logical operations over binary signals. Without these rules, the engine would apply generic interval arithmetic to the polynomial encoding. For example, for OR ($\text{out} = a + b - a \times b$) with $a, b \in \{0, 1\}$, generic arithmetic yields $[0, 2] - [0, 1] = [-1, 2]$, which includes infeasible values and causes false positives.

NonZero Constraint Rules. The inference rules shown in Fig. 4c refine domains by identifying variables that cannot

be zero. The NONZERO-PRODUCT rule is sound because if $e_1 \times e_2 = c$ with $c \neq 0$, then neither e_1 nor e_2 can be zero. Two variants, NONZERO-SUM and NONZERO-CONST, handle algebraic rearrangements of the same pattern (e.g., $e_3 + e_1 \times e_2 = c$ with $e_3 = 0$, or $e_1 \times e_2 + c_1 = c_2$ with $c_1 \neq c_2$) and are listed in Appendix C. These rules can help identify division-by-zero patterns because they propagate nonzero information concerning operands that may occur in denominators on the computation side.

Array Operation Rules. Fig. 4c introduces rules for array-typed variables. The ARRAY-LITERAL, ARRAY-STORE, and ARRAY-FILL rules track per-index domains through construction and update operations. The ARRAY-SELECT rule returns the domain of the selected index (if tracked) or the default domain otherwise.

4.2. Bug Detection Using Signal Tags

Circum offers signal tags for expressing pre- and post-conditions on signals in the computation code (see Sect. 2.3). While tags document assumptions about a circuit, they are purely metadata and do not enforce constraints. This leaves the door open for mismatches between a signal’s intended behavior (as expressed via tags) and its actual allowed values (as defined by the constraints). Leveraging signal tags as a protection mechanism requires checking that the constraints actually enforce what the tags declare.

We focus specifically on automatically detecting the bugs described in Sect. 3.2.3. Bugs from these categories can be detected as violations against signal tags. For instance, an arithmetic overflow can be detected by annotating a signal with the `maxbit(n)` tag, which declares that its value must fit within n bits. It is worth noting that division-by-zero and out-of-bounds array indexing checks are performed even if the programmer does not annotate the relevant signals with explicit tags. For example, signals used for indexing arrays are implicitly checked against `maxvalue(arraylength-1)`. These implicit checks are not shown in Algorithm 3, but are performed as a post-processing step on the inferred domains.

Tags can be added by developers during circuit development or inserted by security experts for bug detection. To better understand the prevalence and variety of signal tags in real-world circuits, we conducted a study on all tagged programs provided by the CIVER benchmark dataset [14]. The results are summarized in Appendix D. CCC-CHECK supports checking inferred domains against all tags from the CIVER datasets. There are two kinds of tags: plain tags and tags with an associated value. Table 1 lists these tags along with their semantics. Note that for DSLs that do not support tags, users can directly provide tags to CCC-CHECK.

CCC-CHECK uses value inference to approximate signal values, and then checks whether these inferred domains satisfy the signal tags. Tags on input signals correspond to preconditions \mathcal{L} in the DCM: they are assumed to hold and are used to refine the initial domains (line 2 of Algorithm 3). Tags on intermediate and output signals correspond to post-conditions \mathcal{T} : they are checked against the inferred domains after the analysis completes.

TABLE 1: Supported signal tags, their type, and semantics.

Tag	Type	Semantics
binary	plain	$0 \leq \text{value} \leq 1$
maxbit	with value n	$0 \leq \text{value} \leq 2^n - 1$
maxvalue	with value n	$0 \leq \text{value} \leq n$
minvalue	with value n	$n \leq \text{value}$
max_abs	with value n	$-n \leq \text{value} \leq n$
maxbit_abs	with value n	$-2^n \leq \text{value} \leq 2^n$
powerof2	plain	value is a power of 2

To encode this formally, we define $\mathcal{T}_{\text{sat}}(\vec{x}, \vec{z}, \vec{y})$ as the set of postconditions satisfied by a given assignment $(\vec{x}, \vec{z}, \vec{y})$, and $\mathcal{T}_{\text{unsat}}(\vec{x}, \vec{z}, \vec{y}) = \mathcal{T} \setminus \mathcal{T}_{\text{sat}}(\vec{x}, \vec{z}, \vec{y})$ as its complement. The function `signals` extracts the signals referenced by a set of postconditions. The set of insufficiently constrained signals is then:

$$I = \{s \mid \exists \vec{x}, \vec{z}, \vec{y} : s \in \text{signals}(\mathcal{T}_{\text{unsat}}(\vec{x}, \vec{z}, \vec{y})) \wedge C(\vec{x}, \vec{z}, \vec{y})\}$$

That is, a signal s is in I if there exists a constraint-satisfying assignment under which some tag on s is violated. CCC-CHECK approximates I using the abstract domains computed by Algorithm 3:

$$\hat{I} = \{s \mid s \in \text{signals}(\hat{\mathcal{T}}_{\text{unsat}}(\mathcal{S})) \wedge \mathcal{S} = \text{Infer}(C)\}$$

where $\hat{\mathcal{T}}_{\text{unsat}}(\mathcal{S})$ returns the set of postconditions not satisfied by the abstract domains in \mathcal{S} .

4.3. Soundness

The value inference engine is sound with respect to bugs caused by unchecked signal tags. Formally, the set of derived unchecked signals \hat{I} must subsume the set of actual unchecked signals I : $I \subseteq \hat{I}$. To this end, the following property must hold: *Let \mathcal{S} be the variable states returned by Algorithm 3, let v be any variable, and γ the concretization function. Then, for every concrete assignment σ that satisfies all constraints C : $\forall v. \sigma(v) \in \gamma(\mathcal{S}[v])$.* This property holds under two assumptions: (i) each inference rule in Fig. 4 computes a sound over-approximation (i.e., for each variable v in each constraint c , the candidate domain \mathcal{D}_{new} produced by `InferDomain(v, c, \mathcal{S})` includes all concretely feasible values of v) and (ii) the meet operator (\sqcap) used to refine domains is itself sound (i.e., the concrete intersection is a subset of the abstract meet). Appendix B details why both assumptions hold for our analysis.

Practical Implications. This soundness property has important implications for our bug detector. For instance, if a signal is tagged with `binary`, and in reality can only take the values 0 and 1, but the analysis over-approximates the values as $\{0, 1, 2\}$, we may flag a false positive. Still, the converse will never happen: if a real violation exists, it will be detected, since over-approximation ensures that no actual behavior is excluded from the inferred domain. For tag verification, the value inference implementation is sound: if no violation is reported, the signal values indeed satisfy the corresponding specifications.

As for precision, one could extend the analysis to keep track of relationships between signals. For instance, if we know that $x = y$ and $z = x + y$, then z can only take even values ($z = 2x$). Without accounting for the relationship $x = y$, the analysis merely observes $x, y \in [0, 10]$ and concludes $z \in [0, 20]$, which also includes impossible odd values. Including such relational information comes at the cost of efficiency. The more relationships are tracked, the more the analysis approaches the computational cost of SMT solvers. One promising direction for improving precision is to extend our analysis with a lightweight relational layer that tracks simple equalities and linear relations between signals, inspired by ideas from concolic execution [34], [35]. This would refine value domains beyond our current per-variable analysis while avoiding the overhead of full SMT solving.

5. Evaluation

This section presents an empirical evaluation of DCM and CCC-CHECK. While our approach is based on an IR and generalizes to other ZKP DSLs (see Appendix F), we evaluate it on Circom since this is the most widely used DSL and enables direct comparison with existing tools. We start by comparing our formal model against existing formalizations of ZKP circuit vulnerabilities (RQ1). Next, we assess the precision and performance of CCC-CHECK by comparing our results against the SMT-based state-of-the-art tool CIVER [14] (RQ2). Then, we evaluate the effectiveness of our value inference approach by comparing it against PICUS [13], a verification tool that combines static analysis with SMT, regarding the number of variables successfully restricted and the precision of inferred ranges (RQ3). Finally, we analyze the prevalence of value-related bugs in six real-world ZKP projects, demonstrating the practical impact of our approach in detecting previously unknown vulnerabilities in widely deployed systems (RQ4).

Experimental Setup. All measurements were conducted on a *MacBook Air* equipped with an *Apple M2* chip, 16 GB of RAM, and running *macOS Sequoia 15.4.1*. We implemented CCC-CHECK in Haskell (with GHC version 9.6.6) and used the `criterion` library for high-precision microbenchmarking. For manual experiments outside of Criterion, we executed each benchmark for at least 15 iterations.

Implementation. The implementation of CCC-CHECK, written in Haskell ($\approx 3,500$ lines of code), is available at <https://github.com/ArmanKolozyan/CCC-Check>.

5.1. RQ1: How does our formal model compare to existing formalizations?

Previous formalizations of ZKP circuit vulnerabilities, such as those by Pailoor et al. [13] and Jiang et al. [15], have only focused on nondeterministic behavior in outputs. The TCCT model [12] extends this to include abnormal program terminations (i.e., the program aborts for a certain input while the constraints still accept the corresponding witness). More recently, Stephens et al. [16] introduced the *agreement*

TABLE 2: Comparison of formal models for vulnerability detection. U-C = Underconstrained, O-C = Overconstrained, A-T = Abnormal Termination, S-M = Semantic Mismatches. The half-filled circle (◐) indicates partial coverage.

Model	U-C	O-C	A-T	S-M
PICUS [13]	✓	✗	✗	✗
CONSCS [15]	✓	✓	✗	✗
ZKFUZZ [12]	✓	✓	✓	✗
ZEQUAL [16]	✓	✗	✗	◐
DCM (Ours)	✓	✓	✓	✓

model used in ZEQUAL, which formalizes consistency between the witness generator and the constraint system as follows: whenever the constraints accept a valuation, the witness generator must produce a corresponding valuation. This model captures underconstrainedness and some semantic mismatches that lead to observable disagreements between computations and constraints. However, it does not capture abnormal terminations and overconstrainedness. Furthermore, neither TCCT nor ZEQUAL considers pre- or postconditions and thus cannot detect semantic mismatches that preserve functional agreement but violate intended semantics (e.g., arithmetic overflows). The *DCM* addresses this critical gap. To the best of our knowledge, our model is the first formalization to incorporate semantic assumptions by explicitly modeling pre- and postconditions. Table 2 summarizes how our model differs from prior ones.

It is worth noting that while CIVER formalizes nondeterminism similarly to PICUS, the authors do not introduce a model of inconsistency classes. We therefore compare against CIVER in the next research question in terms of tag verification effectiveness and scalability.

5.2. RQ2: How effective is our approach in verifying tagged programs?

Methodology. Each program in the CIVER `circomlib` benchmark suite and the `circom-ecdsa` library is annotated with tags such as `binary` or `maxbit` that describe expected value ranges for selected variables. Our analysis verifies whether the inferred range for each tagged variable is a subset of its expected range. We then compare both precision and runtime against CIVER.

Results. Table 3 summarizes the results. Since CIVER does not check for division-by-zero or out-of-bounds array access, we compare both tools exclusively on tags. Nevertheless, we still report CCC-CHECK’s results on these additional checks in Table 3 to assess its performance. CCC-CHECK correctly verifies the tags of all 35 programs and outperforms CIVER on every benchmark, with speedups ranging from $3\times$ to five orders of magnitude.

The largest speedups occur on big-integer multiplication and modular reduction circuits, where polynomial multiplications produce large quantifier-free nonlinear arithmetic formulas (among the hardest classes of SMT problems). This is why CIVER timed out on `bigmult_23` [14], while CCC-CHECK completes the analysis in 0.012 ms. In 7 of

the programs, CCC-CHECK reports division-by-zero false positives due to over-approximation.

The results demonstrate that CCC-CHECK scales effectively to large Circom programs, but may produce false positives in cases where more precise abstractions such as symbolic reasoning or relational abstractions are necessary to verify correctness. We observed this in 7 out of 35 programs, where divisions by zero led to false positives. Despite this, the substantial performance gain demonstrates the practical viability of lightweight value inference for rapid ZKP circuit analysis.

5.3. RQ3: How does our value inference compare to existing approaches of restricting variable ranges?

PICUS is a hybrid verification framework that combines SMT solving with static analysis to detect underconstrained circuits in R1CS. To improve SMT efficiency, PICUS performs lightweight static analysis as a preprocessing step, which includes inferring variable ranges that are subsequently injected into the SMT queries. However, the value inference in PICUS is limited in scope: it only analyzes the constraint system without visibility into the witness computation logic and its inference rules are limited in expressiveness. While the paper evaluates how SMT solving performs without static analysis, it is unclear how effective the value inference phase is. To address this gap, we conducted experiments to answer the following questions:

(i) How many variables are successfully range-restricted by PICUS considering all of its benchmark programs?

(ii) How does CCC-CHECK compare in terms of the number of range-restricted variables?

Analysis of PICUS Range Inference. We wrote a script to extract value inference information from PICUS’s execution across all 559 benchmarks provided by the authors. The detailed statistics are presented in Appendix E. The results indicate that the majority of variables remain unrestricted in PICUS, with only 3.39% being restricted. Further, all constrained variables were restricted to the $\{0, 1\}$ range, indicating that only binary constraints are being propagated. This suggests the value inference mechanisms implemented in PICUS may leave the SMT solver with a large search space. Providing more precise ranges would likely further prune the search space and accelerate SMT solving.

Comparison with Our Inference Engine. To evaluate the relative effectiveness of our approach, we applied the inference engine of both tools to the benchmark suite used in RQ2. Since our analysis operates on an IR that retains a direct link to variables in the original source code, we extended the PICUS tool to preserve a mapping from R1CS variables back to their corresponding Circom variables. This enables a one-to-one comparison between the two tools.

Table 4 shows the results. Across all the programs, CCC-CHECK restricted 381 out of 575 signals (66.3%), while PICUS restricted 143 (24.9%). Additionally, CCC-CHECK inferred not only binary ranges, but also broader domains (e.g., $[0, 7]$), whereas PICUS failed to infer any

non-binary ranges. For variables that both tools restrict, the ranges inferred by CCC-CHECK are at least as precise as those reported by PICUS.

These results demonstrate that CCC-CHECK is significantly more effective at restricting variable ranges than PICUS. Hence, this technique could also be combined with an SMT-based system: by using it as a preprocessing step, it can reduce the search space more effectively and improve SMT solver performance.

5.4. RQ4: How prevalent are value-related bugs in real-world programs?

To assess the prevalence of value-related bugs in real-world settings, we examined the code of a selection of ZKP projects. This evaluation represents a DCM-guided analysis in which we manually tagged six projects, similar to the workflow of security auditors. We selected six projects based on their recognition within the developer community (i.e., high GitHub star counts and active commit history) and their representation of diverse application domains:

TikTok Trustless Attestation Verification [36]: ZKPs for verifying Trusted Execution Environment (TEE) attestations.

Microsoft Crescent [37]: A library for generating ZKPs of possession for mobile Driver’s Licenses (mDL).

Rooch [38]: A platform for building verifiable applications with transparent computation and state.

Self [39]: An application for creating ZKPs from government IDs with selective attribute disclosure (e.g., age).

ZKP2P [40]: A trustless peer-to-peer cryptocurrency exchange system using ZKPs for payment verification.

co-snarks [41]: Tooling for collaborative SNARKs, enabling multiple parties to jointly produce proofs.

We observed 15 issues spread across three bug types. The most frequent issue was *arithmetic overflows* due to missing range constraints (12 issues), followed by *invalid circuit logic assumptions* (2 issues) and *missing parameter assertions* (1 issue).

Table 5 summarizes bug categories and patch status per project. Among the analyzed projects, *Self* stands out as the one with the highest number of GitHub stars, reflecting significant adoption. The vulnerability found in *Self*, which had not been detected during an audit conducted by an industry-leading security company, earned us a bug bounty. The TikTok project showed the most issues; all others have arithmetic-overflow bugs. Out of the 15 vulnerabilities, 13 have already been patched by the project maintainers following our responsible disclosure (see Appendix G).

The fact that 12 out of 15 vulnerabilities were arithmetic overflows demonstrates that these bugs are not rare mistakes but systematic pitfalls in ZKP circuit development. This highlights the practical relevance of our work: the DCM is, to the best of our knowledge, the only formalization that captures such semantic mismatch bugs.

TABLE 3: Performance and verification power comparison between CCC-CHECK and CIVER on Circom programs. LOC = lines of code, #It = worklist iterations. Tags are checked by both tools, while Array-Bounds and Divisions are only reported for CCC-CHECK. “Verified?” reports whether each program is verified without false positives (✓) or with a false positive (✗).

Program	LOC	#It	CCC-CHECK (ms)	CIVER (ms)	Speedup	Verified?		
						Tags	Array-Bounds	Divisions
escalarmulfx	284	4,831	12.152	36.58	3×	✓	✓	✗
escalarmul	145	1,531	0.870	47.58	55×	✓	✓	✓
pedersen	181	3,379	1.870	79.98	43×	✓	✓	✓
mux1	27	15	0.002	5.75	3,194×	✓	✓	✓
mux1_1	27	83	0.004	5.63	1,522×	✓	✓	✓
mux2	59	17	0.002	3.55	2,088×	✓	✓	✓
mux2_1	59	146	0.005	3.50	745×	✓	✓	✓
mux3	72	27	0.003	3.65	1,141×	✓	✓	✓
mux3_1	72	207	0.008	3.60	444×	✓	✓	✓
mux4	103	49	0.007	3.95	573×	✓	✓	✓
mux4_1	103	390	0.014	3.84	270×	✓	✓	✓
pointbits_loopback	136	6,113	0.653	257.85	395×	✓	✓	✓
babybk	344	5,827	9.003	131.70	15×	✓	✓	✗
pedersen2	276	2,441	0.908	42.03	46×	✓	✓	✓
escalarmul_min	141	1,095	0.975	103.65	106×	✓	✓	✓
constants	24	67	0.003	3.85	1,426×	✓	✓	✓
check_bitify	37	91	0.005	23.89	4,594×	✓	✓	✓
escalarmulany	208	3,125	6.901	27.34	4×	✓	✓	✗
binsub	43	20	0.002	17.65	7,674×	✓	✓	✓
sum	51	20	0.003	20.47	7,059×	✓	✓	✓
aliascheck	63	1,320	0.121	110.98	920×	✓	✓	✓
sign	66	1,315	0.120	170.92	1,423×	✓	✓	✓
check_comps	79	236	0.258	37.33	145×	✓	✓	✗
decoder	18	9	0.018	19.86	1,128×	✓	✓	✗
bigmod_32	337	435	0.468	5,104.89	10,920×	✓	✓	✗
bigmod_22	337	410	0.388	5,101.05	13,140×	✓	✓	✗
bigsubmodp_32	221	423	0.029	87.81	3,018×	✓	✓	✓
bigmult_21	142	56	0.004	31.32	8,242×	✓	✓	✓
bigmult_22	142	77	0.007	184.75	25,660×	✓	✓	✓
bigmult_23	142	97	0.012	5,044.67	434,885×	✓	✓	✓
bigadd_15	93	129	0.011	34.64	3,268×	✓	✓	✓
bigadd_23	93	132	0.009	46.37	5,210×	✓	✓	✓
bigadd_2030	90	159	0.106	146.26	1,385×	✓	✓	✓
bigsub_23	133	71	0.005	57.62	11,081×	✓	✓	✓
bigsub_15	133	71	0.008	58.10	7,645×	✓	✓	✓

6. Related Work

ZKP Intermediate Representations (IRs). Several IRs have been proposed to decouple high-level ZKP languages from specific proving systems, enabling reusable compiler tooling [18], [42], [43], [44]. CIRC IR [18] was the only mature IR available during our research and provides a separation between the computations and the constraint system, which is crucial for reasoning about mismatches between them. LLZK [44], though promising, is still under development. ACIR [42] and Vamp-IR [43] are other mature IRs designed for ZKP programs, but they are not yet widely adopted beyond the Aztec and Anoma teams. Other approaches, such as *LLVM IR* used in ZKAP [11], were designed for traditional CPU programs and lack native support for field arithmetic. For these reasons, we opted for CIRC IR as the basis for our analysis. Nevertheless, our tool can support different IRs as it only requires a translation backend from the chosen IR to the value inference engine.

ZKP Static Analysis Tools. CIRCOMSPECT [10] combines syntactic and lightweight taint analysis to detect underconstrained signals but remains largely syntactic and lacks a

formal foundation. ZKAP [11] builds a dependency graph to enable more semantic reasoning. Not only are both tools limited to Circom, but they also lack value reasoning and therefore cannot detect discrepancies such as arithmetic overflows or division-by-zero.

ZKP Dynamic Analysis Tools. ZKFUZZ [12] applies fuzzing to mutated Circom circuits. While effective at uncovering specific bugs guided by predefined targets, fuzzing is inherently incomplete and its execution time grows quickly with program size because each mutation requires recomputing the witness and re-evaluating all constraints.

ZKP Hybrid Approaches. CIVER [14], PICUS [13], and ZEQUAL [16] combine lightweight static reasoning with SMT solvers to improve scalability. Since SMT solving remains an integral part of their analysis pipeline, these tools still suffer from performance bottlenecks. This stems mainly from two factors [27], [28], [29], [30]: (1) SMT solvers are not optimized for reasoning over finite fields and (2) many ZKP constraints involve non-linear arithmetic, which they handle poorly. In contrast, our approach avoids solver bottlenecks entirely through value inference, introduces an

TABLE 4: Number of signals with inferred range restrictions by CCC-CHECK and PICUS on the RQ2 benchmark programs.

Program	Total signals	CCC-CHECK	PICUS
mux1	3	1	0
mux1_1	2	2	1
mux2	7	2	0
mux2_1	7	6	1
mux3	11	3	0
mux3_1	11	10	2
mux4	22	4	0
mux4_1	22	20	4
check_bitify	5	5	2
check_comparators	22	20	14
decoder	5	2	0
binsub	3	3	2
binsum	2	2	1
aliascheck	5	5	2
sign	5	5	2
constants	1	1	0
bigadd_15	9	7	4
bigadd_23	9	7	4
bigadd_2030	9	7	4
bigsub_23	11	8	5
bigsub_15	11	8	4
bigmult_21	6	6	3
bigmult_22	7	7	4
bigmult_23	7	7	4
bigsubmodp_32	22	20	14
bigmod_32	30	28	21
bigmod_22	30	28	19
pointbits_loopback	24	24	16
escalarmul_min	28	13	1
escalarmul	29	13	1
escalarmulfix	55	31	1
escalarmulany	39	23	2
pedersen	30	12	1
pedersen2	30	13	1
babygbk	56	28	3

TABLE 5: Bug categories and patch status per project. AO = Arithmetic overflow, CLA = Circuit logic assumptions, MPA = Missing parameter assertions.

Project	Stars	AO	CLA	MPA	Total	Patched
TikTok TAV	72	4	2	1	7	6
Microsoft Crescent	39	4	0	0	4	4
Rooch	181	1	0	0	1	1
Self	1082	1	0	0	1	1
ZKP2P	314	1	0	0	1	0
co-snarks	183	1	0	0	1	1
Total	/	12	2	1	15	13/15

analysis that considers not only the constraints but also the computations, and thereby detects a broader range of inconsistency-related bug classes while remaining scalable. While recent work has focused on enhancing SMT solvers to be more efficient for finite fields [28], [30], our approach can further improve efficiency by serving as a preprocessing step to SMT-based tools, eliminating many cases that would otherwise require SMT reasoning. An advantage of SMT-based approaches is their precision, whereas our lightweight inference may signal false positives in cases requiring symbolic reasoning or relational constraints. The two approaches are therefore complementary: our tool provides rapid analysis

and scalability, while SMT solvers can be used to distinguish true positives from false positives when higher precision is needed by the developers or security researchers.

Abstract Interpretation. Abstract interpretation is an established technique for static analysis by over-approximating concrete semantics using abstract domains [33]. It has been applied in a wide range of domains [45], [46], [47], [48]. To the best of our knowledge, we are the first to apply abstract interpretation for ZKP bug detection.

7. Conclusion

This paper introduced both theoretical and practical advances toward improving the correctness of ZKP programs. We proposed the *Domain Consistency Model (DCM)*, a formal framework that captures the semantic relationship between computation and constraint systems. By generalizing prior notions of underconstrained circuits, the DCM provides a unified foundation for reasoning about a broad spectrum of semantic mismatch vulnerabilities. Further, we developed CCC-CHECK, a bug detection tool that makes the DCM tractable in real-world settings. It combines a value inference engine based on abstract interpretation with a flexible specification mechanism via signal tags. By working on an intermediate representation (IR), it is applicable across several ZKP languages. Our evaluation showed that the DCM captures vulnerabilities missed by existing analyzers while CCC-CHECK operates at a fraction of the run-time cost of SMT-based approaches. Moreover, we uncovered previously unknown vulnerabilities that can only be expressed within our semantic framework, underscoring both its effectiveness and practical relevance. While our work formalizes key classes of soundness violations and establishes a path toward automated reasoning about them, several open challenges remain. Future directions include enriching our analysis with relational abstract domains to reduce false positives and integrating emerging intermediate representations, such as LLZK IR, to extend support across diverse ZKP DSLs and ecosystems.

Acknowledgements

We thank the S&P reviewers for their detailed feedback, with special thanks to our anonymous shepherd for thoughtful guidance, which substantially improved the paper. We would also like to thank Nicolas Mohnblatt for helpful discussions around this topic. We are grateful to Clara Rodríguez and Hideaki Takahashi for discussing technical aspects of CIVER and ZKFUZZ, respectively. Our thanks further go to Rémi Colin for offering a bug bounty in response to our vulnerability report on Self, and to Donghang Lu from TikTok’s Privacy Innovation for being open in discussing the bugs. We also thank Giorgio Dell’Immagine for providing helpful feedback on the first drafts of this paper. Finally, we are grateful to the teams at Nokia Bell Labs, Veridise, zkSecurity, and Self Labs for their comments and feedback while we presented earlier versions of this work.

References

- [1] S. Goldwasser, S. Micali, and C. Rackoff, "The knowledge complexity of interactive proof-systems," in *Proceedings of the 17th Annual ACM Symposium on Theory of Computing (STOC)*, 1985, pp. 291–304.
- [2] Z. Community, "Zksync documentation," <https://docs.zksync.io>.
- [3] P. Labs, "Polygon zkEVM documentation," <https://docs.polygon.technology/zkEVM/>.
- [4] A. Stapelberg. (2025) Opening up "zero-knowledge proof" technology to promote privacy in age assurance. Google. [Online]. Available: <https://blog.google/technology/safety-security/>
- [5] (2025) Self docs. Self Labs. [Online]. Available: <https://docs.self.xyz>
- [6] M. Bellés-Muñoz, M. Isabel, J. L. Muñoz-Tapia, A. Rubio, and J. Baylina, "Circom: A circuit description language for building zero-knowledge applications," *IEEE Transactions on Dependable and Secure Computing*, vol. 20, no. 6, pp. 4733–4751, 2023.
- [7] Tornado Cash. Tornado.cash got hacked. by us. [Online]. Available: <https://tornado-cash.medium.com/tornado-cash-got-hacked-by-us-1e012a3c9a8>
- [8] M. Connor. Disclosure of recent vulnerabilities. [Online]. Available: <https://hackmd.io/@aztec-network/disclosure-of-recent-vulnerabilities>
- [9] S. Chaliasos, J. Ernstberger, D. Theodore, D. Wong, M. Jahanara, and B. Livshits, "SoK: What don't we know? understanding security vulnerabilities in SNARKs," in *Proceedings of the 33rd USENIX Conference on Security Symposium (USENIX Security)*, 2024, pp. 3855–3872.
- [10] F. Dahlgren. (2022) It pays to be circomspect. Trail of Bits. [Online]. Available: <https://blog.trailofbits.com/2022/09/15/it-pays-to-be-circomspect/>
- [11] H. Wen, J. Stephens, Y. Chen, K. Ferles, S. Pailoor, K. Charbonnet, I. Dillig, and Y. Feng, "Practical security analysis of zero-knowledge proof circuits," in *Proceedings of the 33rd USENIX Conference on Security Symposium (USENIX Security)*, 2024, pp. 1471–1487.
- [12] H. Takahashi, J. Kim, S. Jana, and J. Yang, "zkfuzz: Foundation and framework for effective fuzzing of zero-knowledge circuits," 2025.
- [13] S. Pailoor, Y. Chen, F. Wang, C. Rodríguez, J. Van Geffen, J. Morton, M. Chu, B. Gu, Y. Feng, and I. Dillig, "Automated detection of under-constrained circuits in zero-knowledge proofs," *Proceedings of the ACM on Programming Languages*, vol. 7, no. PLDI, 2023.
- [14] M. Isabel, C. Rodríguez-Núñez, and A. Rubio, "Scalable verification of zero-knowledge protocols," in *Proceedings of the 45th IEEE Symposium on Security and Privacy (SP)*, 2024, pp. 1794–1812.
- [15] J. Jiang, X. Peng, J. Chu, and X. Luo, "ConsCS: Effective and Efficient Verification of Circom Circuits," in *Proceedings of the 47th IEEE/ACM International Conference on Software Engineering (ICSE)*, 2025, pp. 737–737.
- [16] J. Stephens, S. Pailoor, and I. Dillig, "Automated verification of consistency in zero-knowledge proof circuits," in *Computer Aided Verification*, R. Piskac and Z. Rakamarić, Eds., 2025, pp. 315–338.
- [17] F. Wang. (2022) Ecne: Automated verification of zk circuits. [Online]. Available: <https://0xparc.org/blog/ecne>
- [18] A. Ozdemir, F. Brown, and R. S. Wahby, "Circ: Compiler infrastructure for proof systems, software verification, and more," in *Proceedings of the 43rd IEEE Symposium on Security and Privacy (SP)*, 2022, pp. 2248–2266.
- [19] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza, "Succinct non-interactive zero knowledge for a von neumann architecture," in *Proceedings of the 23rd USENIX Conference on Security Symposium (USENIX Security)*, 2014, pp. 781–796.
- [20] R. Gennaro, C. Gentry, B. Parno, and M. Raykova, "Quadratic span programs and succinct nizes without pcps," in *Proceedings of the Advances in Cryptology (EUROCRYPT)*, T. Johansson and P. Q. Nguyen, Eds., 2013, pp. 626–645.
- [21] A. Gabizon, Z. J. Williamson, and O. Ciobotaru, "PLONK: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge," Cryptology ePrint Archive, 2019.
- [22] ConsenSys. (2025) gnark documentation. [Online]. Available: <https://docs.gnark.consenSys.io/overview>
- [23] Electric Coin Company. (2023) Halo 2 book. [Online]. Available: <https://zcash.github.io/halo2/>
- [24] J. Thaler, "Proofs, arguments, and zero-knowledge," *Foundations and Trends® in Privacy and Security*, vol. 4, no. 2-4, pp. 117–660, 2022.
- [25] T. Hess, "Tag, you're it: Signal tagging in circom," *Trail of Bits Blog*, 2024. [Online]. Available: <https://blog.trailofbits.com/2024/01/02/tag-youre-it-signal-tagging-in-circom/>
- [26] *Signal Tags*, iden3, 2024. [Online]. Available: <https://docs.circom.io/circom-language/tags/>
- [27] T. Hader, D. Kaufmann, and L. Kovacs, "Smt solving over finite field arithmetic," *EPiC Series in Computing*, vol. 94, pp. 218–238, 2023.
- [28] A. Ozdemir, S. Pailoor, A. Bassa, K. Ferles, C. Barrett, and I. Dillig, "Split gröbner bases for satisfiability modulo finite fields," in *Proceedings of the Computer Aided Verification (CAV)*, A. Gurfinkel and V. Ganesh, Eds., 2024, pp. 3–25.
- [29] T. Hader and A. Ozdemir, "An smt-lib theory of finite fields," 2024.
- [30] A. Ozdemir, G. Kremer, C. Tinelli, and C. Barrett, "Satisfiability modulo finite fields," in *Proceedings of the Computer Aided Verification (CAV)*, C. Enea and A. Lal, Eds., 2023, pp. 163–186.
- [31] yAcademy Auditors. yacademy spartan-ecdsa review. Electi Security. [Online]. Available: https://github.com/electisec/spartan-ecdsa-audit-report#1-high--input-signal-s-is-not-constrained-in-eff_ecdsacircom
- [32] J. Liu, I. Kretz, H. Liu, B. Tan, J. Wang, Y. Sun, L. Pearson, A. Miltner, I. Dillig, and Y. Feng, "Certifying Zero-Knowledge Circuits with Refinement Types," in *Proceedings of the 45th IEEE Symposium on Security and Privacy (SP)*, 2024, pp. 1741–1759.
- [33] P. Cousot and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Proceedings of the Conference Record of the Fourth ACM Symposium on Principles of Programming Languages (POPL)*, 1977, pp. 238–252.
- [34] K. Sen, D. Marinov, and G. Agha, "Cute: a concolic unit testing engine for c," in *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2005, pp. 263–272.
- [35] P. Godefroid, N. Klarlund, and K. Sen, "Dart: directed automated random testing," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2005, pp. 213–223.
- [36] TikTok Privacy Innovation. (2024) Trustless attestation verification. [Online]. Available: <https://github.com/tiktok-privacy-innovation/trustless-attestation-verification-circom>
- [37] Microsoft. (2024) Crescent: Zero-knowledge proofs for jwts and mdls. [Online]. Available: <https://github.com/microsoft/crescent-credentials>
- [38] Rooch Network. (2024) Rooch: A platform for building verifiable applications. [Online]. Available: <https://github.com/rooch-network/rooch>
- [39] Self Labs. (2024) Self: Privacy-first application for zk proofs from government ids. [Online]. Available: <https://github.com/selfxyz/self>

- [40] ZKP2P. (2024) Zkp2p: Trustless peer-to-peer cryptocurrency exchange. [Online]. Available: <https://github.com/zkp2p/zkp2p-v1-monorepo>
- [41] Taceo Labs. (2024) co-snarks: Tooling for collaborative snarks. [Online]. Available: <https://github.com/TaceoLabs/co-snarks>
- [42] a. team, “acir: Arbitrary circuit intermediate representation,” <https://lib.rs/crates/acir>, 2025, accessed: 2025-11-10.
- [43] —, “Vamp-ir: a proof-system-agnostic language for writing arithmetic circuits,” <https://github.com/anoma/vamp-ir>, 2023, accessed: 2025-11-10.
- [44] *LLZK-lib Documentation: Project Overview*, Veridise, 2025. [Online]. Available: <https://veridise.github.io/llzk-lib/main/overview.html>
- [45] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival, “The astrÉE analyzer,” vol. 3444, 2005, pp. 21–30.
- [46] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski, “Frama-c: a software analysis perspective,” in *Proceedings of the 10th International Conference on Software Engineering and Formal Methods*, 2012, pp. 233–247.
- [47] B. Jeannet and A. Miné, “Apron: A library of numerical abstract domains for static analysis,” in *Computer Aided Verification*, A. Bouajjani and O. Maler, Eds., 2009, pp. 661–667.
- [48] C. Calcagno, D. Distefano, J. Dubreil, D. Gabi, P. Hooimeijer, M. Luca, P. O’Hearn, I. Papakonstantinou, J. Purbrick, and D. Rodriguez, “Moving fast with software verification,” in *NASA Formal Methods*, K. Havelund, G. Holzmann, and R. Joshi, Eds., 2015, pp. 3–11.
- [49] RISC Zero. (2025) Zirgen circuit compiler. <https://github.com/risc0/zirgen>.
- [50] Aztec Labs. (2022) Introducing noir: The universal language of zero-knowledge. [Online]. Available: <https://aztec.network/blog/introducing-noir-the-universal-language-of-zero-knowledge>
- [51] J. Eberhardt and S. Tai, “Zokrates - scalable privacy-preserving off-chain computations,” in *Proceedings of the IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, 2018, pp. 1084–1091.
- [52] Noir Contributors. (2024) Zk benchmark. [Online]. Available: https://github.com/noir-lang/zk_bench
- [53] J. Bruestle, P. Gafni, and R. Z. Team, “Risc zero zkvm: Scalable, transparent arguments of risc-v integrity,” RISC Zero, White Paper, 2023.
- [54] S. Labs. (2025) Sp1 docs. [Online]. Available: <https://docs.succinct.xyz/docs/sp1/introduction>
- [55] A. Arun, S. Setty, and J. Thaler, “Jolt: Snarks for virtual machines via lookups,” in *Proceedings of the Advances in Cryptology (EUROCRYPT)*, M. Joye and G. Leander, Eds., 2024, pp. 3–33.

Appendix A. Arithmetic Overflow Example

This section provides a detailed example of how arithmetic overflows can lead to underconstrained circuits, as explained in SCV 3.1. We then explain why only our DCM model covers this type of vulnerability, whereas traditional approaches fall short.

A canonical example of where this type of bug can arise is the `LessThan(n)` template from the standard Circom library, which, given two input values, checks whether one is less than the other. The comparison of a and b is performed by adding a constant offset 2^n , resulting in the expression

$z = 2^n + a - b$. This ensures that the most significant bit of the binary representation of z determines whether $a < b$.

Circom implements this idea in the `LessThan(n)` template (Listing 2), where we manually added tags to the template (note that the tag syntax is simplified for clarity).

```

1 template LessThan(n) {
2   assert(n <= 252);
3   signal input {maxbit: n} in[2];
4   signal output {binary} out;
5
6   component n2b = Num2Bits(n+1);
7   n2b.in <= in[0] + (1 << n) - in[1];
8
9   out <= 1 - n2b.out[n];
10 }

```

Listing 2: `LessThan(n)` template of circomlib.

The `LessThan` template uses `Num2Bits` to bit-decompose the value $z = 2^n + a - b$. The most significant bit of the resulting binary vector indicates whether $z \geq 2^n$, in which case $a \geq b$, and the output is set to 0. Otherwise, the output is 1, indicating $a < b$.

```

1 template Withdraw() {
2   signal input {maxbit: 2} withdrawAmount;
3   signal input {maxbit: 2} currentBalance;
4   signal output {binary} validWithdraw;
5
6   component It = LessThan(2);
7   It.in[0] <= withdrawAmount;
8   It.in[1] <= currentBalance;
9
10  validWithdraw <= It.out;
11 }

```

Listing 3: Simplified withdrawal circuit.

Now consider the withdrawal circuit in Listing 3. This circuit is intended to approve a withdrawal only if the amount is strictly less than the current balance. Since 2 is passed as parameter value to `LessThan`, it assumes that the values of its inputs will fit in 2 bits. However, due to the lack of range constraints, a prover can exploit this logic.

Suppose we use the following input: `withdrawAmount = p - 2` (which lies far outside the 2-bit range) and `currentBalance = 1`, where p is the field modulus. Then $z = 2^2 + (p - 2) - 1 = p + 1$. Since all arithmetic is modulo p , this reduces to $z = 1$.

The `Num2Bits(3)` component (see `Num2Bits(n+1)` in Listing 2) in `LessThan` decomposes this value as $[1, 0, 0]$, and the most significant bit is 0, meaning the comparison falsely concludes that `withdrawAmount < currentBalance`, and sets `validWithdraw` to 1. This effectively allows a withdrawal of $p - 2$ coins from an account with only 1 coin, thus creating money out of thin air.

According to the program’s specification, the input `withdrawAmount` is expected to be a 2-bit number (i.e., bounded by $2^2 - 1 = 3$). However, in the above example, the constraint system does not enforce this bound. As a result, the constraint domain includes the out-of-range value: $p - 2 \in \mathcal{D}_C(\text{withdrawAmount})$.

This value leads to a valid output `validWithdraw = 1` for this input, even though the program would not permit

this execution. Since the computation logic assumes that `withdrawAmount` is bounded, we have:

$$\mathcal{D}_P(\text{validWithdraw} \mid \text{withdrawAmount} = p - 2, \\ \text{currentBalance} = 1) = \emptyset$$

That is, the program would not produce any output for this input combination. However, the constraint system does produce exactly one valid output. This mismatch constitutes a *phantom output* and thus an underconstrained circuit.

Why Traditional Approaches Fall Short. Traditional bug detection techniques for ZKP circuits focus on identifying cases where the same input can lead to multiple valid outputs (nondeterminism). However, in this example, the constraint system is fully deterministic: given any specific input values, there is exactly one valid output.

More recent work, such as ZEQUAL [16], attempts to detect vulnerabilities by identifying discrepancies between the computation and constraint views. However, this approach also fails to catch this bug because it does not account for the intended semantics. To see why, consider the computation and constraint semantics without preconditions.

On the computation side, given inputs `withdrawAmount = p - 2` and `currentBalance = 1`, the witness generator evaluates $z = 4 + (p - 2) - 1 = p + 1 \equiv 1 \pmod{p}$. The bit decomposition of $z = 1$ yields $[1, 0, 0]$, with the most significant bit equal to 0. Therefore, `validWithdraw = 1 - 0 = 1`.

On the constraint side, the constraint system encodes the same arithmetic operations: the computation of $z = 4 + \text{withdrawAmount} - \text{currentBalance}$, the bit decomposition of z into a binary vector, and the formula `validWithdraw = 1 - msb(z)`. When instantiated with the same input values, the constraints are satisfied with `validWithdraw = 1`.

Since both the computation and the constraint views produce the same output (`validWithdraw = 1`) for the input `withdrawAmount = p - 2`, there is no observable discrepancy between them.

The vulnerability only becomes apparent when we incorporate the *precondition* that `withdrawAmount` should be a 2-bit value. The input `withdrawAmount = p - 2` is now invalid and should be rejected. As explained above, our DCM model detects this vulnerability by incorporating preconditions into the analysis.

Appendix B. Implementation Details of Inference Rules

This appendix discusses implementation details for the inference rules presented in Fig. 4. While the rules are formally presented using the `KnownValues` representation (i.e., as explicit sets), the actual implementation uses the more efficient `BoundedValues` representation wherever possible. For operations such as addition, subtraction, and multiplication, computing the result by explicitly enumerating all pairs $(v_1, v_2) \in \Omega_1 \times \Omega_2$ would be prohibitively expensive when the domains are large. Instead, the implementation leverages interval arithmetic on `BoundedValues` to

compute bounds on the result without explicit enumeration. Throughout, “sound” means *over-approximating*: a result contains every concretely reachable value, though it may also contain unreachable ones. Every abstract transformer follows a single contract: if the result can be tightly bounded, we compute it, and otherwise we return the full field $[0, p-1]$ (which is trivially sound).

B.1. Transformers

Given a `BoundedValues` domain $\Omega = \text{BoundedValues}(lb, ub, G)$, we write $\text{hull}(\Omega)$ for the cyclic interval $[lb, ub]$ obtained by removing all gaps G , and $\text{hullSize}(\Omega)$ for its size: $ub - lb + 1$ when $lb \leq ub$ (non-wrapping), and $p - lb + ub + 1$ when $lb > ub$ (wrapping). Arithmetic transformers operate on this hull interval. Since $\Omega \subseteq \text{hull}(\Omega)$, any result computed from two hulls is a superset of the result computed from the original domains, so this is a sound over-approximation.

Addition. Given $\Omega_1 = [lb_1, ub_1]$ and $\Omega_2 = [lb_2, ub_2]$ (possibly wrapping), each hull interval represents a set of consecutive values modulo p . Hence, their pairwise sums $\{(v_1 + v_2) \bmod p\}$ also form a set of consecutive values modulo p , of size $\text{hullSize}(\Omega_1) + \text{hullSize}(\Omega_2) - 1$. If $\text{hullSize}(\Omega_1) + \text{hullSize}(\Omega_2) - 1 \geq p$, the sum covers all of \mathbb{F}_p and we return $[0, p-1]$. Otherwise, we compute $lb_r = (lb_1 + lb_2) \bmod p$ and $ub_r = (ub_1 + ub_2) \bmod p$. If $lb_r \leq ub_r$, we return $[lb_r, ub_r]$. If $lb_r > ub_r$, we return the wrap-around interval representing $[lb_r, p-1] \cup [0, ub_r]$.

Subtraction. Analogous to addition, since pairwise differences of consecutive values modulo p again form consecutive values. The bounds become $(lb_1 - ub_2) \bmod p$ and $(ub_1 - lb_2) \bmod p$.

Multiplication. Unlike addition, multiplication of two intervals modulo p does not generally yield an interval. However, when both operands are non-wrapping ($lb \leq ub$) and all four corner products $lb_1 \times lb_2$, $lb_1 \times ub_2$, $ub_1 \times lb_2$, $ub_1 \times ub_2$ are $< p$, no modular reduction occurs for any product from the two intervals. Since integer multiplication is monotone on non-negative integers, the minimum and maximum of the four corners bound every product $v_1 \times v_2$, so we return the interval from the smallest to the largest corner product. When either condition fails, we return $[0, p-1]$.

Field-constant multiplication. Several rules (e.g., `ASSIGN`) require multiplying a domain by a field constant $c^{-1} \bmod p$. For `KnownValues`, this is computed exactly by mapping each element. For `BoundedValues`, multiplication by a field constant does not in general preserve interval structure (e.g., $[0, 2] \times 10 \bmod 17 = \{0, 10, 3\}$), so we return $[0, p-1]$.

Mixed-domain operations. When one operand is `KnownValues` S and the other is `BoundedValues`, we over-approximate S as $[\min(S), \max(S)]$ and apply the `BoundedValues` transformer. This is sound since $S \subseteq [\min(S), \max(S)]$.

B.2. Meet and Join

Meet (\sqcap). The meet operation is a sound over-approximation of the set intersection operation, i.e., for abstract elements $\mathcal{D}_1, \mathcal{D}_2$ with concretization γ : $\gamma(\mathcal{D}_1 \sqcap \mathcal{D}_2) \supseteq \gamma(\mathcal{D}_1) \cap \gamma(\mathcal{D}_2)$. Hence, the result may be less precise than exact set intersection, but it remains sound:

- **KnownValues** \sqcap **KnownValues**: We compute the exact set intersection.
- **KnownValues** \sqcap **BoundedValues**: We test each value for membership in the interval (respecting wrap-around and gaps).
- **BoundedValues** \sqcap **BoundedValues**: When both intervals are non-wrapping, we compute $[\max(lb_1, lb_2), \min(ub_1, ub_2)]$ with the union of both gap sets. If $\max(lb_1, lb_2) > \min(ub_1, ub_2)$, the intervals are disjoint and the result is empty (i.e., an inconsistency is reported). When one or both intervals wrap, we decompose each wrapping interval into its two non-wrapping halves ($[lb, p-1]$ and $[0, ub]$), compute pairwise intersections of the resulting non-wrapping sub-intervals, and take their union. For example, if Ω_1 wraps and Ω_2 does not, we intersect $[lb_1, p-1]$ and $[0, ub_1]$ each with $[lb_2, ub_2]$, yielding sub-intervals $L = [lb_2, \min(ub_1, ub_2)]$ (when $lb_2 \leq ub_1$) and $R = [\max(lb_1, lb_2), ub_2]$ (when $lb_1 \leq ub_2$). If both are non-empty, the result is **BoundedValues**($lb_2, ub_2, G_1 \cup G_2 \cup \{\min(ub_1, ub_2)+1, \max(lb_1, lb_2)-1\}$), adding one gap sub-interval for the hole between L and R . When both intervals wrap, we similarly decompose both and compute four pairwise intersections, adding at most two new gap sub-intervals.
- **ArrayDomain** \sqcap **ArrayDomain**: Given that the array sizes match, we compute the element-wise meet of per-index and default domains. Indices tracked in only one array are joined with the other's default domain.

Join (\sqcup). The join, used in the ITE rule, over-approximates the union:

- **KnownValues** \sqcup **KnownValues**: We compute the set union.
- **BoundedValues** \sqcup **BoundedValues**: If both are non-wrapping, we compute $[\min(lb_1, lb_2), \max(ub_1, ub_2)]$ with gaps dropped. This is sound since both inputs are subsets of the result. If either input wraps, we return $[0, p-1]$.
- **KnownValues** \sqcup **BoundedValues**: We over-approximate the finite set as $[\min(S), \max(S)]$, then apply the previous rule.
- **ArrayDomain** \sqcup **ArrayDomain**: Given that the array sizes match, we compute the element-wise join of per-index and default domains. Indices tracked in only one array are joined with the other's default domain.

Appendix C. Additional Inference Rules

This appendix presents additional inference rules that complement those shown in Fig. 4.

Boolean Operation Rules. Fig. 5 presents the rules that complement **BOOL-AND** from Fig. 4b. Each rule infers that the output is binary when the inputs are binary and the constraint matches the corresponding boolean formula.

$$\frac{a, b \in \{0, 1\} \quad out = a + b - a \times b \in C}{\nu, C \vdash out : \{0, 1\}} \text{ (BOOL-OR)}$$

$$\frac{a, b \in \{0, 1\} \quad out = a + b - 2 \times a \times b \in C}{\nu, C \vdash out : \{0, 1\}} \text{ (BOOL-XOR)}$$

$$\frac{a \in \{0, 1\} \quad out = 1 + a - 2 \times a \in C}{\nu, C \vdash out : \{0, 1\}} \text{ (BOOL-NOT)}$$

$$\frac{a, b \in \{0, 1\} \quad out = 1 - a \times b \in C}{\nu, C \vdash out : \{0, 1\}} \text{ (BOOL-NAND)}$$

$$\frac{a, b \in \{0, 1\} \quad out = (a \times b + 1) - (a + b) \in C}{\nu, C \vdash out : \{0, 1\}} \text{ (BOOL-NOR)}$$

Figure 5: Additional boolean constraint inference rules.

NonZero Constraint Rules. Fig. 6 presents the rules that complement **NONZERO-PRODUCT** from Fig. 4c.

$$\frac{e_3 + e_1 \times e_2 = c \in C \quad c \neq 0 \quad \nu(e_3) = \{0\}}{\nu, C \vdash e_1, e_2 : \nu(e_i) \setminus \{0\}} \text{ (NONZERO-SUM)}$$

$$\frac{e_1 \times e_2 + c_1 = c_2 \in C \quad c_1 \neq c_2}{\nu, C \vdash e_1 : \nu(e_1) \setminus \{0\}, e_2 : \nu(e_2) \setminus \{0\}} \text{ (NONZERO-CONST)}$$

Figure 6: Additional nonzero constraint inference rules.

Appendix D. Prevalence of Signal Tags

To better understand the prevalence and variety of signal tags in real-world circuits, we conducted a study on all tagged programs provided by the CIVER benchmark dataset [14]. This includes circuits from `circomlib`³, the `ECDSA` library⁴, and the `DarkForest` project⁵. The results are summarized in Table 6.

In total, we analyzed 147 Circom programs and found 128 tag instances across 6 unique tag types. The most frequently used tags are `binary` and `maxbit`, which together account for over 70% of all annotations. These results show that signal tags are mainly used to express binary properties. Our inference engine is particularly well suited for this task, given the rules we defined in Sect. 4.1.3.

3. https://github.com/costa-group/circomlib/tree/only_adding_tags

4. <https://github.com/costa-group/circom-ecdsa>

5. <https://github.com/costa-group/tagged-darkforest-v0.6>

TABLE 6: Signal tag usage statistics across the CIVER benchmark suite.

Tag	Occurrences	Percentage
binary	47	36.7%
maxbit	44	34.4%
maxvalue	16	12.5%
max_abs	12	9.4%
powerof2	8	6.2%
maxbit_abs	1	0.8%
Total	128	100%

Appendix E. Picus Range Inference Statistics

We analyzed the value inference capabilities of PICUS across all 559 benchmark programs provided by the authors, each with a timeout of 3 minutes. We then parsed the output to compute statistics about how many variables received finite value bounds. Table 7 summarizes the results.

TABLE 7: Range-restriction statistics for PICUS across 559 benchmark programs.

Metric	Value
Total benchmarks analyzed	559
Total variables analyzed	4,504,380
Total restricted variables	152,776
Percentage of restricted variables (global)	3.39%
Average % restricted variables per benchmark	18.43%
Inferred range for all restricted variables	{0, 1}

These results indicate that the vast majority of variables remain unconstrained in PICUS. Moreover, all constrained variables were restricted to the $\{0, 1\}$ range, indicating that only binary constraints are being propagated. The low percentage of restricted variables (3.39%) suggests that the value inference mechanisms currently implemented in PICUS are not fully effective, leaving the SMT solver with a significantly large search space.

Appendix F. Applicability of CCC-Check to Other ZKP Languages

While Circom served as the driving example due to its widespread adoption, we now discuss how our approach generalizes to other ZKP languages.

Low-Level DSLs with Explicit Constraint Writing. In most circuit DSLs, such as Gnark [22] and Zirgen [49], constraints need to be explicitly written by developers. This makes them directly susceptible to computation-constraint discrepancies. Hence, our approach is directly applicable.

High-Level DSLs with Automatic Constraint Generation. Higher-level DSLs such as Noir [50] and ZoKrates [51] aim to reduce developer burden by automatically generating constraints from computation code. While this reduces the likelihood of inconsistencies, it comes at a significant performance cost: the number of constraints generated by compilation is reported to be 3 to 300 times larger compared

to hand-written constraints [12], [52]. Consequently, many developers still opt to write constraints by hand, especially for performance-critical parts, using primitives provided by these languages. In Noir, developers can use `unsafe` blocks to write custom constraints, while ZoKrates provides `assembly` for similar purposes. These escape mechanisms reintroduce the risk of computation-constraint discrepancies, making our tool applicable to these higher-level DSLs.

Zero-Knowledge Virtual Machines (zkVMs). A more recent trend in the ZKP ecosystem is the adoption of zkVMs (Zero-Knowledge Virtual Machines), such as RISC Zero [53], SP1 [54], and Jolt [55]. These allow developers to write programs in general-purpose languages (e.g., Rust), which are then compiled and executed on a VM whose execution is proven using ZKPs. Although zkVMs abstract away circuit-level details, the VM’s instruction set is itself implemented using arithmetic circuits. Our tool is applicable at this level, detecting inconsistencies between the VM’s execution semantics and the constraints that enforce correctness of the execution trace.

Appendix G. Ethical Considerations

We responsibly disclosed all previously unknown vulnerabilities found in public projects, adhering to established ethical standards for security research. When available, we followed the security vulnerability reporting procedures of the projects. Otherwise, we first contacted maintainers privately via email. In most cases, they then granted us permission to open GitHub issues and submit corresponding pull requests.

All vulnerability disclosures were made at least 60 days before paper submission, providing maintainers with sufficient time to review, patch, and deploy fixes. We maintained open communication with project maintainers throughout the disclosure process. When needed, we held meetings with maintainers to discuss the vulnerabilities in detail. We also provided guidance on protection mechanisms such as signal tags to help prevent similar issues in the future.

Table 8 lists the public issue reports and pull requests for each affected project.

Project	GitHub Issue IDs	GitHub PR IDs
TikTok TAV [36]	4–16	17–20
Microsoft Crescent [37]	125, 128, 131, 133	129, 130, 132, 134
Rooch [38]	3599	3600
Self [39]	578	579
ZKP2P [40]	428	429

TABLE 8: References to GitHub issue reports and pull requests for disclosed vulnerabilities in the affected projects.

Appendix H. Meta-Review

The following meta-review was prepared by the program committee for the 2026 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

H.1. Summary

This paper presents a theoretical formalism (dubbed DCM) allowing to capture inconsistencies between the prover computation and the constraint system checked by the verifier in a ZKP program. This model is used to characterize a number of inconsistency bugs. The DCM formalism is claimed to be more precise than existing formalisms targeting computation/constraints inconsistencies. Then, the paper proposes CCC-Check, an abstract interpretation based tool allowing to detect such bugs. Finally, CCC-Check and the DCM framework are evaluated.

H.2. Scientific Contributions

- Creates a New Tool to Enable Future Science.
- Identifies an Impactful Vulnerability.
- Provides a Valuable Step Forward in an Established Field.

H.3. Reasons for Acceptance

- 1) This paper works on an important problem in zero-knowledge proofs: making sure the program's computations and its verification constraints match correctly.
- 2) The proposed approach is practical and fast, which makes it useful for finding certain types of ZKP bugs.
- 3) The paper found vulnerabilities in real world programs.