Automated Verification of Consistency in Zero-Knowledge Proof Circuits

Jon Stephens[1,2], Shankara Pailoor[2], Isil Dillig[1,2]

[1]The University of Texas at Austin. {jon, isil}@cs.utexas.edu
[2]Veridise Inc. {jon, shankara, isil}@veridise.com

Abstract. Circuit languages like Circom and Gnark have become essential tools for programmable zero-knowledge cryptography, allowing developers to build privacy-preserving applications. These domain-specific languages (DSLs) encode both the computation to be verified (as a *witness generator*) and the corresponding *arithmetic circuits*, from which the prover and verifier can be automatically generated. However, for these programs to be correct, the witness generator and the arithmetic circuit need to be mutually consistent in a certain technical sense, and inconsistencies can result in security vulnerabilities. This paper formalizes the consistency requirement for circuit DSLs and proposes the first automated technique for verifying it. We evaluate the method on hundreds of real-world circuits, demonstrating its utility for both automated verification and uncovering errors that existing tools are unable to detect.

1 Introduction

Zero-knowledge (ZK) proofs are cryptographic protocols that allow a prover to convince a verifier of the truth of a statement without disclosing confidential information. These protocols have become crucial building blocks in implementing privacy-preserving applications. Recent advancements in cryptography have introduced a powerful class of *succinct* zero-knowledge proofs [1, 2] that enable verification of complex computations while keeping both proof sizes and verification times minimal. This innovation has enabled a wide range of applications, including anonymous whistleblowing [3], image authentication [4], private digital transactions [5], and ensuring computational integrity in blockchains [6].

However, successfully integrating zero-knowledge proofs into an application requires developers to create two interdependent artifacts: (1) a set of constraints, formulated as polynomial equations over a finite field and (2) a *witness generator* that produces satisfying values for these constraints. Crucially, a fundamental assumption about these artifacts is that the constraints should only accept values that are produced by the witness generator. Failure to meet this requirement can lead to severe security vulnerabilities, allowing true statements to become unprovable or, even more seriously, false statements to be incorrectly verified.

To aid developers in defining these two artifacts, several domain-specific languages (DSLs) have been developed. These "circuit DSLs", such as Circom and Gnark, streamline the process of maintaining consistency between constraints and witness generators. For example, several DSLs include constructs that allow developers to define constraints and assignments simultaneously, reducing the risk of errors. Despite these tools, inconsistencies remain a major challenge, with over 95% of known ZK vulnerabilities arising from such issues [7].

This paper aims to address these challenges by introducing a formal definition of consistency between constraints and witness computation and presenting the first automated algorithm to verify this notion of consistency in circuit DSLs. Informally, a circuit is *consistent* if its constraints accept only those witnesses generated by the witness generator. While prior work has focused on detecting specific instances of inconsistency (such as non-determinism) [8–11], none have provided an automated solution for verifying the stronger notion consistency formalized in this paper.

From a technical perspective, the consistency verification problem poses two key challenges. First, it requires proving a relational property between a witness generator and an arithmetic circuit, which oftentimes defines non-linear finite field equations that tend to challenge SMT solvers. Second, both of these artifacts are typically derived from *templates* containing arrays and complex loops, necessitating the inference of complex quantified loop invariants.

This paper tackles the consistency verification problem through a two-step approach. First, given a *product program* that simultaneously performs constraint generation and witness construction, our approach uses lightweight static reasoning to infer pair-wise equivalences between array elements and simplifies the initial product program as much as possible. In the subsequent verification step, our method infers more complicated quantified invariants over arrays to discharge the verification conditions needed for proving equivalence.

We have implemented our method in a tool called ZEQUAL, which verifies consistency of circuit templates written in Circom, a widely-used circuit DSL and the target language of several other ZK circuit verifiers and bug-finding tools in prior literature [8–10, 12]. We evaluate ZEQUAL on hundreds of real-world benchmarks taken from popular projects and show that ZEQUAL can verify consistency in a majority of these templates. We also demonstrate that ZEQUAL is capable of finding bugs that cannot be detected using prior methods.

To summarize, contributions of this paper include: (1) a formal definition of the *consistency verification problem* for ZK circuits; (2) first automated technique for verifying consistency of ZK circuit templates; (3) evaluation of the method on several hundred real-world circuit templates.

2 Background

Zero-Knowledge Proofs (ZKPs) are protocols that allow a prover to demonstrate the truth of a statement to a verifier without revealing any additional information beyond the statement's validity. Traditionally, ZKPs were developed for specific statements such as the three-coloring problem or graph isomorphism; however, new advancements in Programmable ZKP have enabled greater flexi-

```
template MultiMux() {
    signal input x[5]; // Constants
    signal input y[5]; // Constants
    signal input s; // Selector
    signal output out[5];
    for (var i=0; i<5; i++) {
        out[i] <== (x[i] - y[i])*s + x[i];
    }
}
```

```
template IsZero() {
    signal input in;
    signal output out;
    signal inv;
    inv <-- in!=0 ? 1/in : 0;
    out <== -in*inv +1;
    in*out === 0; }
```

Fig. 2: Examples of Circom templates.

bility. For example, proof frameworks like zk-SNARKs [1] or zk-STARKs [2] can generate verifiable proofs that $P(x) = y$ for some computation P and private input x. A key property of programmable ZKPs is that a prover and verifier can be automatically generated from a set of polynomial equations over a finite field $\mathbb{F}_p$. The variables in these equations are referred to as signals, and addition and multiplication are performed modulo a large prime p. The details of how the prover and verifier are generated are beyond the scope of this paper, but we refer the interested reader to [1, 13, 14] for additional background.

Once the prover and verifier are set up, proofs are generated and verified as illustrated in Figure 1. The process begins with the witness generator, a program that produces a *witness*—a mapping of signals to values that satisfy the constraints. Conceptually, the witness generator executes the computation P for which the user seeks to prove correctness. The prover then uses this witness, along with the constraints, to generate a proof, which is sent to the verifier. Crucially, the proof conceals the values of all private signals. The verifier receives the proof and only the *public* values, verifying its correctness or rejecting it. Notably, witnesses do not need to be generated solely by running the witness generator; any satisfying assignment to the constraints can produce a verifiable proof. As a result, it is essential that the constraints accurately reflect the intended computation and align with the witness generator. Inconsistencies between these artifacts could lead to invalid proofs being accepted or valid computations failing to produce verifiable proofs. Indeed, most security vulnerabilities in real-world implementations of these protocols stem from such inconsistencies [7].

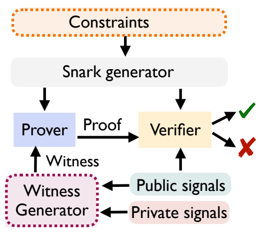

Fig. 1: ZK workflow.

Circom DSL. Manually encoding a computation as a set of constraints is a time-consuming and error-prone task. To address these challenges, several domain-specific languages have been developed [15–17], with Circom being one of the most popular [16]. Circom allows developers to specify the witness generation logic and constraints in a unified language (with concepts similar to a hardware description language (HDL)), and the compiler automatically generates the witness generator along with the constraints. Tools like SnarkJS [18] can then be used to generate the prover and verifier from these constraints.

A Circom program consists of a list of *templates* with one template distinguished as the entry point. A template is similar to a sub-circuit module in a

```
template Divide() {
   signal input numerator, denominator;
   signal output remainder, quotient;
   isZero := IsZero();
   isZero.in <== denominator;
   isZero.out == 0; // ensure denominator is non-zero
   remainder <-- numerator % denominator;
   quotient <-- numerator \ denominator;
   numerator === denominator * quotient + remainder;
}
```

```
template Reward() {

  signal input inp;
  signal output out;
  var gwei = 10 ** 6;
  out <-- inp \ gwei;
  out * gwei === inp;
}

```

Fig. 3: Non-determinism bug on left; inconsistency bug on right.

HDL as it defines a reusable block of code that is instantiated upon invocation by connecting the declared input and output signals within the caller. All output signals in Circom are public and other signals are private by default unless otherwise specified. Witness generation is expressed using the `<--` and `-->` operators for signal assignment, while the `===` operator expresses constraints. Additionally, the `==>` and `<==` operators can be used to combine assignment and constraint declaration when the left and right hand sides are polynomials, helping developers maintain consistency between computation and constraints. For example, the program on the left in Figure 2 only performs addition and multiplication over signals—operations that are directly expressible as polynomials over finite fields. As such, it is able to exclusively use the `<==` operator, keeping the witness generation and constraints consistent. In contrast, the program in Figure 2(b) does not use the `<==` operator because comparisons cannot be expressed directly as polynomials. Instead, the corresponding constraints utilize an intermediate signal, `inv`, relying on the field axiom that every nonzero element has a multiplicative inverse.

Circuit Bugs. Languages like Circom help developers maintain consistency between witness generation and constraints, but implementations often contain bugs due to discrepancies. A common issue, known as "Unintended Nondeterminism" or "Underconstrained Circuits", arises when constraints $C(in, out)$ allow multiple outputs o and o' for the same input i. The program on the left of Figure 3 provides an example of a nondeterminism bug: the template takes a numerator and denominator as inputs and returns the quotient and remainder. The witness generation code performs this logic directly using the integer division operator and the modulo operator `%`. However, the constraints only assert that $\texttt{quotient} * \texttt{numerator} + \texttt{remainder} = \texttt{numerator}$, lacking a crucial check that the remainder is *less than* the denominator when interpreted as integers. Without this check, there are multiple solutions for a given input.

However, not all discrepancies manifest themselves as nondeterminism bugs: constraints can be deterministic, yet their satisfying assignments may still diverge from the witness values, as illustrated on the right of Figure 3[1]. This program takes an input currency amount `inp`, and the witness generator returns a reward `out` by dividing `inp` by a constant `gwei` using integer division. Here, the computation and constraints only align when `inp` is a multiple of `gwei` but

[1]This example was taken from [9] and is a simplified version of a bug found in a real-world closed-source project.

```
 1 template DecomposeProduct(N) {
 2   signal input u8_xs[N], u8_ys[N];
 3   signal u16s[N];
 4   signal output high[N], low[N];
 5   component bits_low[N], bits_high[N];
 6 
 7   for (var i = 0; i < N; i++) {
 8     u16s[i] <== u8_xs[i] * u8_ys[i];
 9     low[i] <-- u16s[i] % 0x100;
10     bits_low[i] = Num2Bits(8);
11     bits_low[i].in <== low[i];
12     high[i] <-- (u16s[i] \ 0x100) % 0x100;
13     bits_high[i] = Num2Bits(8);
14     bits_high[i].in <== high[i];
15     u16s[i] === low[i] + 0x100 * high[i];
16   }
17 }
```

Fig. 4: Code for motivating example

```
 1 template DecomposeProduct_Expanded(N) {
 2   signal input u8_xs[N], u8_ys[N];
 3   signal u16s_w[N], u16s_c[N];
 4   signal output high_w[N], high_c[N];
 5   signal output low_w[N], low_c[N];
 6   component bits_low[N], bits_high[N];
 7 
 8   for (var i = 0; i < N; i++) {
 9     u16s_w[i] <-- u8_xs[i] * u8_ys[i];
10     u16s_c[i] === u8_xs[i] * u8_ys[i];
11     low_w[i] <-- u16s_w[i] % 0x100;
12     bits_low[i] = Num2Bits(8);
13     bits_low[i].in_w <-- low_w[i];
14     bits_low[i].in_c === low_c[i];
15     high_w[i] <-- (u16s_w[i] \ 0x100) % 0x100;
16     bits_high[i] = Num2Bits(8);
17     bits_high[i].in_w <-- high_w[i];
18     bits_high[i].in_c === high_c[i];
19     u16s_c[i] === low_c[i] + 0x100 * high_c[i];
20   }
21 }
```

Fig. 5: Product program

diverge in all other cases. For instance, if `inp` is 1, the witness will assign `out` to 0, while the satisfying assignment for the constraints sets `out` to $\texttt{gwei}^{-1}$. This results in the output (as defined by the constraints) being even larger than the input, which could allow gaining more reward than intended.

3 Motivating Example

Figure 4 presents a Circom template which takes two arrays, `u8_xs` and `u8_ys` of length N, each containing field elements whose integer representation is an unsigned 8-bit integer. The template computes the element-wise product, capturing the result as 16-bit unsigned integers in `u16s`. Each entry in `u16s` is then decomposed into two output arrays, `low` and `high`, which store the lower and higher 8 bits, respectively.

At first glance, it is not immediately obvious why the witness generator and the constraints expressed by this program align with each other. For instance, the witness generator computes the low and high bytes in a standard way using integer division and the modulo operator (lines 9, 12), whereas the constraints enforce this decomposition at the bit level, validating each byte component individually through binary representations (Num2Bits) (lines 10, 13). Given the nontrivial differences between the witness generator and the constraints, establishing their equivalence over all possible template instantiations is challenging. In the following discussion, we explain the key ideas that enable ZEQUAL to perform verification successfully, along with the observations that inspired these ideas.

Observation #1: *Many circuit DSLs provide a "pseudo-product-program" with good alignment.* A common approach to verifying equivalence is to reduce the relational verification task to checking assertions in a *product program* [19]. A crucial step in this reduction is establishing a suitable *alignment* — a mapping that pairs corresponding operations between the two programs. In generalpurpose languages, achieving this alignment can be difficult because the program structures may differ significantly.

Fortunately, circuit DSLs like Circom, Gnark, and Zirgen implicitly define a "pseudo-product-program" with good alignment where corresponding operations between the witness and constraint components are grouped within the same basic block. We call this a *pseudo-product* because it uses the *same variables* to represent signals in both the constraints and the witness generator even when they may *not* be equivalent. Hence, in order to instrument the program with suitable assertions that ensure equivalence, our method modifies the Circom program by introducing two copies s_c, s_w for each (non-input) signal. For instance, as shown in Figure 5, `high[N]` is replaced with two arrays `high_w[N]` and `high_c[N]` denoting the constraint and witness generation components respectively.

Observation #2: *Lightweight static analysis can help infer pair-wise equivalence between signals, simplifying the verification task.* As mentioned in Section 2, circuit DSLs often include constructs that generate equivalent witness generation logic and constraints from a single statement. For instance, in Circom, the statement `x <== e` serves as shorthand for both the assignment `x <- e` and the constraint `x === e`. Other circuit languages also have similar features.[2]

This duality between witness generator logic and constraints is commonly leveraged by circuit developers to maintain consistency across the two artifacts. This makes it feasible to infer pair-wise equality between the two copies s_c, s_w of a signal s through *lightweight static analysis* of the product program. For example, at line 9 of Figure 5, `u16s_w[i]` is assigned to some expression `e` and line 10 includes constraint `u16s_c[i] === e` using the same expression `e`. Because `e` only involves input signals, we can conclude that `u16s_w[i]` and `u16s_c[i]` are pair-wise equivalent and then propagate this information to potentially prove equivalence of more signals. Finally, for any pair of signals proven equivalent, we can simplify the product program by deleting assignments and replacing the two copies of a signal with a single variable. In this example, we can delete the assignment at line 9 and replace `u16s_w` and `u16s_c` with a single variable `u16s`.

Observation #3: *Invariants needed for verification often have a particular shape.* After simplifying the product program, our approach instruments it with assumptions and assertions (see Figure 6) and tries to discharge the latter. However, because circuit programs often contain loops, Zequal needs to synthesize quantified array invariants. While this is a challenging problem in general, most required invariants involve predicates of the form $\forall \bar{i}.\ \phi(\bar{i}) \rightarrow s_w[f(\bar{i})] = s_c[f(\bar{i})]$ where ϕ is a guard predicate over the loop counters and f is an arithmetic function. Our method generates candidate invariants of this shape through static analysis and computes the strongest conjunction over this universe of predicates

[2]For example, in Gnark and Cairo, every operation, *by default*, is translated both into a statement (in the witness generator) and a field equation; when this is not possible, the developer needs to employ so-called "hints". Similarly, in Zirgen, computation performed in a component generates equivalent witness generator logic and constraints while computation performed in an extern is only performed in the witness generator.

```
template DecomposeProduct_Instrumented(N) {
    signal input u8_xs[N], u8_ys[N];
    signal u16s[N];
    signal output high_w[N], high_c[N];
    signal output low_w[N], low_c[N];
    component bits_low[N], bits_high[N];

    for (var i = 0; i < N; i++) {
      assume(u16s[i] = u8_xs[i]*u8_ys[i]);
      low_w[i] <-- u16s_w[i] % 0x100;
      bits_low[i] = Num2Bits(8);
      bits_low[i].in_w <-- low_w[i];
      assume(bits_low[i].in_c = low_c[i]);
      assert(bits_low[i].in_w = low_w[i]);
      high_w[i] <-- (u16s_w[i] \ 0x100) % 0x100;
      bits_high[i] = Num2Bits(8);
      bits_high[i].in_w <-- high_w[i];
      assume(bits_high[i].in_c = high_c[i]);
      assert(bits_low[i].in_w = low_w[i]);
      assume(u16s[i] = low_c[i] + 0x100 * high_c[i]);
      assert(u16s[i] = low_w[i] + 0x100 * high_w[i]);
    }
    assert(low_w = low_c && high_w = high_c);
}
```

Fig. 6: Instrumented product

through monomial predicate abstraction [20]. For our example, the following invariant is sufficient for successful verification:

$$\forall x.\ 0 \leq x < i \leq N \rightarrow (\texttt{low\_w}[x] = \texttt{low\_c}[x] \wedge \texttt{high\_w}[x] = \texttt{high\_c}[x])$$

4 Problem Statement

In this paper, we adopt a formalization that captures commonalities between most circuit DSLs, such as Circom [16], Gnark [15], Zirgen [21], and Zokrates [22].

Definition 1 (Circuit program). *A ZK circuit program P is a tuple $(\Gamma, \mathbb{F}_p, \Phi, \omega)$ where:*

- *Γ is a set of signal declarations $s : \tau$, where $\tau \in \{\mathsf{In}, \mathsf{Out}, \mathsf{Temp}\}$;*
- *$\mathbb{F}_p$ is the prime field of the circuit – i.e., $s \in \mathbb{F}_p$ for any signal s;*
- Constraint *Φ is a formula in the* theory of finite fields *[23]. The precise language for such formulas is shown on the left side of Figure 7.*
- Witness computation *ω is a loop-free program over signals in Γ. The language for expressing witness computation is shown on the right side of Figure 7.*

Given a program $P = (\Gamma, \mathbb{F}_p, \Phi, \omega)$, we write P_Φ, P_ω to denote Φ and ω respectively, and, for a signal s, $\Gamma(s)$ denotes the type of s. We refer to signals of type In, Out, and Temp as input, output, and intermediate signals respectively. Witness computation ω can *only* perform assignments to output and intermediate signals but never to inputs. Furthermore, ω can never perform multiple assignments to the same signal within the same execution.

The semantics of ZK circuit programs are defined over *signal valuations* σ that map signals to concrete values. The output of a program is a tuple (b, σ') where b is a boolean and σ' is the output valuation. Given a program $P =$

$$
\begin{array}{lll}
\text{Constraints } \Phi & ::= & \Phi_1 \land \Phi_2 \mid P_1 == P_2 \\
\text{Polynomial } P & ::= & P_1 + P_2 \\
 & & \mid P_1 * P_2 \mid s \mid c
\end{array}
\qquad
\begin{array}{lll}
\text{Program } \omega & ::= & s \leftarrow E \mid \omega_1; \omega_2 \\
 & & \mid \texttt{if}(C)\ \omega_1\ \texttt{else}\ \omega_2 \\
\text{Conditional } C & ::= & E_1 \otimes E_2 \mid \neg C \\
\text{Expression } E & ::= & E_1 \oplus E_2 \mid s \mid c \mid E^{-1}
\end{array}
$$

Fig. 7: (Left) Arithmetic circuit DSL. P is a polynomial equation over a finite field $\mathbb{F}_p$. (Right) Witness generation DSL over $\mathbb{F}_p$. E^{-1} computes the multiplicative inverse of E, and $\oplus, \otimes$ denote arithmetic and logical operators respectively.

$(\Gamma, \mathbb{F}_p, \Phi, \omega)$ we write $(\sigma, P) \Downarrow (b, \sigma')$ iff $[\![\Phi]\!](\sigma)$ evaluates to b and executing ω on σ yields new environment σ'. Next, we define a notion of *agreement* between the witness computation and constraint in ZK circuit programs.

Definition 2 (Agreement). *Let $P = (\Gamma, \mathbb{F}_p, \Phi, \omega)$ be a program and let σ be a signal valuation such that $(\sigma, P) \Downarrow (b, \sigma')$. We say that constraint Φ and witness computation ω* agree *on σ, denoted $\sigma \vdash \Phi \sim \omega$, iff the following holds:*

$$b \rightarrow ([\![\Phi]\!](\sigma') \land \forall (s : \mathsf{Out}) \in \Gamma.\ \ \sigma(s) = \sigma'(s)) \tag{1}$$

According to this definition, if the constraint Φ "accepts" some valuation σ (meaning b is true), then (1) the output valuation σ' produced by the witness generator should also be accepted by Φ, and (2) the two valuations σ and σ' should agree on the values of all output signals. In other words, if a given witness σ with inputs I is accepted by the constraints Φ, then the witness generator ω should also be able to produce a satisfying witness σ' with the same inputs I such that the output signals in σ and σ' match. This ensures that the constraints will only accept valuations that are produced by the witness generator, which in turn captures the desired behavior of a ZK circuit program.

Definition 3 (Consistency). *Let $P = (\Gamma, \mathbb{F}_p, \Phi, \omega)$ be a ZK circuit program. P is* correct *iff for all $\sigma \in \mathbb{F}_p^{|\Gamma|}$, we have $\sigma \vdash \Phi \sim \omega$.*

The definition of consistency expresses the desired notion of correctness as it states that the constraint Φ agrees with the witness computation function ω for all possible signal valuations. This definition is similar to the definition of *strong safety* from prior work [16]. In particular, strong safety states that for every input to the circuit, (a) there is exactly one solution to the constraints and (b) that solution is equal to the witness produced by the witness generator on that input. However, we note that *strong safety* is too strong a requirement in practice: for many real world circuits, there can be multiple witnesses that satisfy the constraints which disagree on their assignment to the *intermediate signals* but agree on their assignments to the output signals. In contrast, our notion of consistency allows the two witnesses to disagree on the assignment to intermediate signals as long as both witnesses satisfy the constraints.

$$\begin{aligned}
\text{Skeleton } \mathcal{P} &::= D; S \\
\text{Var Decl } D &::= \texttt{var } v[E]; \mid D_1; D_2 \\
\text{Statement } S &::= \square_i \mid v \leftarrow E \mid S_1; S_2 \mid \texttt{if}(C)\ S_1\ \texttt{else}\ S_2 \mid \texttt{while}(C)\ S \\
\text{Expression } E &::= E_1 \oplus E_2 \mid s \mid v \mid \alpha \mid E_1[E_2] \mid c \\
\text{Conditional } C &::= E_1 \otimes E_2 \mid \neg C
\end{aligned}$$

Fig. 8: Template syntax. c, s, v, α denote constants, signals, variables, and parameters respectively where $\oplus \in \{+, -, *, /, \backslash, \&, |\}$ and $\otimes \in \{\leq, <, >, \geq, \neq, =\}$. Holes are instantiated with terms from Fig. 9 and 10.

$$\begin{aligned}
\text{Constraint } \phi &::= E'_1 == E'_2 \\
\text{Expression } E' &::= s \mid v \mid \alpha \mid c \mid E'_1[E'_2] \\
&\mid E'_1 + E'_2 \mid E'_1 * E'_2
\end{aligned}$$

Fig. 9: Constraint DSL

$$\begin{aligned}
\text{Assignments } A &::= L \leftarrow E \mid A; A \\
\text{LHS Expression } L &::= s \mid L[E]
\end{aligned}$$

Fig. 10: Witness DSL

ZK circuit templates. While ZK circuit programs cannot contain unbounded constraints or computation, many DSLs *do* allow developers to implement *templates* containing loops as long as these templates can be instantiated to ZK circuit programs at *compile time*. However, in general, it is often desirable to reason about the correctness of the template itself rather than its specific instantiations. For example, many libraries provide circuit templates that are instantiated by clients of that library, so reasoning about the correctness of a library in isolation requires the ability to reason about template correctness. Thus, we also generalize our formalization to circuit templates as follows.

Definition 4 (Template). *A template $\mathcal{T}$ is a tuple $(V, \Gamma, \mathbb{F}_p, \mathcal{P}, \Sigma_\Phi, \Sigma_\omega)$ where:*

- *V is a set of template parameters;*
- *Γ is a set of signal declarations $s : \tau$, and $\mathbb{F}_p$ is the prime field for the circuit;*
- *$\mathcal{P}$ is a program with holes belonging to the syntax of Figure 8;*
- *Σ_Φ is a mapping from holes in $\mathcal{P}$ to constraints, as defined in Figure 9;*
- *Σ_ω is a mapping from holes in $\mathcal{P}$ to statements, as defined in Figure 10*

We refer to $\mathcal{P}[\Sigma_\Phi]$ as a constraint template *and $\mathcal{P}[\Sigma_\omega]$ as a* witness template*.*

Template parameters $\alpha \in V$ are instantiated at compile-time; thus, they can never be modified. Signals can have type $\mathsf{Array}(\tau)$, and we refer to a type τ as an *input type*, denoted $\mathsf{IsInput}(\tau)$, if $\tau = \mathsf{In}$ or $\tau = \mathsf{Array}(\tau')$ and $\mathsf{IsInput}(\tau')$.

In most DSLs, the constraint and witness templates are required to have the same control flow structure; thus, we represent them using a shared *control-flow skeleton* $\mathcal{P}$. As shown in Figure 8, the skeleton involves loops, conditionals, constants, and importantly, *mutable* data variables denoted v. Typically, data variables are used as loop counters and for configuring different parts of the circuit. The constraint and witness templates are obtained by filling the holes in the shared control-flow skeleton with constraints (Fig. 9) or statements that manipulate signals (Fig. 10) respectively. Statements in the witness DSL can only perform assignments to signals, but not to data variables, which can only be modified in the shared control-flow skeleton.

```
1: procedure VerifyCircuitTemplate($\mathcal{T}$)
   input: A ZK circuit template $\mathcal{T} = (V, \Gamma, \mathbb{F}_p, \mathcal{P}, \Sigma_\Phi, \Sigma_\omega)$
2:     # Step 1: Construct product of circuit and witness programs
3:     $S \leftarrow \{s \mid (s : \tau) \in \Gamma, \neg\mathsf{IsInput}(\tau)\}$
4:     $\mathcal{P}_\otimes \leftarrow \mathcal{P}[\{(\Box_i \mapsto A_i[S_w/S]; \phi_i[S_c/S]) \mid \Sigma_\Phi[\Box_i] = \phi_i \wedge \Sigma_\omega[\Box_i] = A_i\}]$
5:     # Step 2: Perform static analysis to simplify $P_\otimes$ into $P_\star$
6:     $\mathcal{E} \leftarrow$ InferEquivalences($P_\otimes, S$)
7:     $\mathcal{P}_\star \leftarrow$ Simplify($\mathcal{P}_\otimes, \mathcal{E}$)
8:     # Step 3: Verify product program
9:     $\mathcal{P}_I \leftarrow$ Instrument($\mathcal{P}_\star$)
10:    return VerifyProduct($\mathcal{P}_I$)
```

Fig. 11: Top-level algorithm. Given signals S, we write S_x to denote $\{v_x \mid v \in S\}$

Definition 5 (Instantiation). *A* template instantiation *is a pair* $(\mathcal{T}, \nu)$ *where* $\mathcal{T}$ *is a template, and* ν *maps template parameters* V *to values in* $\mathbb{F}_p$*, and* $(\mathcal{T}, \nu) \hookrightarrow P$ *indicates that* P *is the ZK circuit program obtained by evaluating* $\mathcal{T}$ *under* ν*.*

Definition 6 (Correctness of template). *A ZK circuit template* $\mathcal{T}$ *is correct if, for* every ν *such that* $(\mathcal{T}, \nu) \hookrightarrow P$*,* P *is correct according to Definition 3.*

5 Verification Algorithm

Figure 11 shows our top-level algorithm for verifying the correctness of a circuit template. The algorithm starts by constructing a *product program* [19] that encodes the simultaneous execution of both the witness template $\mathcal{P}[\Sigma_\omega]$ and the constraint template $\mathcal{P}[\Sigma_\Phi]$ (lines 3–4). Because circuit languages force programmers to write constraint and witness templates with the same shared control-flow structure, constructing a suitable product program is very easy: We introduce two different copies S_c, S_w of the *non-input* signals and replace each hole $\Box_i$ in $\mathcal{P}$ with $A_i[S_w/S]; \phi_i[S_c/S]$ where $A_i = \Sigma_\omega[\Box_i]$ and $\phi_i = \Sigma_\Phi[\Box_i]$.

The second step of the algorithm statically analyzes the constructed product program $\mathcal{P}_\otimes$ and tries to infer invariants of the form $s_c = s_w$ establishing that two copies of the same signal s have the same value. If so, we can treat signal s as an input signal after deleting all assignments to s_w and renaming the different copies s_c and s_w in $\mathcal{P}_\otimes$ both as s (performed by Simplify at line 7). Finally, the algorithm instruments the resulting program to contain suitable assertions and assumptions by calling Instrument at line 9 and attempts to discharge the assertions by calling VerifyProduct at line 10.

Assumptions. The rest of this section assumes that all signals and data variables are modeled as (possibly flattened) single-dimensional arrays. Following existing circuit languages, we also disallow multiple assignments to signals.

5.1 Inference of Equivalent Signals via Static Analysis

In this section, we present the InferEquivalences algorithm from Figure 12, which uses lightweight static analysis to infer equivalent signal pairs $(s_c, s_w) \in (S_w \times S_c)$. The algorithm first constructs a so-called *symbolic store* Σ to represent values of array elements as symbolic expressions over the inputs

```
1: procedure InferEquivalences($P_\otimes$, $S$)
2:     $\Sigma \leftarrow$ ConstructSymbolicStore($P_\otimes$);    $\mathcal{E} \leftarrow \varnothing$
3:     for $s \in S$ do
4:         if IsEquivalent($\Sigma, s_w, s_c$) then $\mathcal{E} \leftarrow \mathcal{E} \cup \{s\}$
5:     return $\mathcal{E}$
```

Fig. 12: Algorithm for inferring equivalent signals

$$\frac{\varsigma \text{ fresh variable}}{\Sigma \vdash \star : \varsigma}\,(\textsc{Unknown}) \qquad \frac{\Sigma \vdash E : \varphi \quad (\upsilon, \varphi) \in dom(\Sigma)}{\Sigma \vdash \upsilon[E] : \Sigma(\upsilon, \varphi)}\,(\textsc{Arr-1})$$

$$\frac{\begin{array}{c}\Sigma \vdash E : \varphi \quad (\upsilon, \varphi) \notin dom(\Sigma) \\ \neg\mathsf{IsInput}(\upsilon) \quad \Sigma \vdash \star : \varphi'\end{array}}{\Sigma \vdash \upsilon[E] : \varphi'}\,(\textsc{Arr-2}) \qquad \frac{\begin{array}{c}\Sigma \vdash E : \varphi \quad (\upsilon, \varphi) \notin dom(\Sigma) \\ \mathsf{IsInput}(\upsilon)\end{array}}{\Sigma \vdash \upsilon[E] : \upsilon[\varphi]}\,(\textsc{Arr-3})$$

$$\frac{\mathsf{IsConst}(E) \ \vee \ \mathsf{IsTemplParam}(E)}{\Sigma \vdash E : E}\,(\textsc{Basic}) \qquad \frac{\Sigma \vdash E_i : \varphi_i}{\Sigma \vdash f(E_1, \ldots, E_n) : f(\varphi_1, \ldots, \varphi_n)}\,(\textsc{Arith})$$

Fig. 13: Look-up rules. Every application of Unknown produces a fresh variable.

(line 2) and then checks pair-wise equality between them (line 4). The symbolic expressions φ in our analysis are defined by the grammar:

$$\varphi := \alpha \mid c \mid \varsigma \mid s_{in}[\varphi] \mid \varphi \oplus \varphi$$

Here, s_{in} is an input signal, α a template parameter, c a constant, ς an unknown, and $\oplus$ an arithmetic operator. Symbolic expressions represent immutable values, ensuring that syntactically identical expressions are equal. To improve precision and enable relational reasoning, fresh variables ς are used to represent unknowns instead of the standard top element $\top$ in abstract interpretation. For a symbolic expression φ, $\mathsf{Unknown}(\varphi)$ indicates that it corresponds to a fresh variable ς.

The symbolic store Σ maps array elements (υ, φ) to symbolic expressions, where υ is a signal or data variable, and φ represents an array index. For instance, $(s_w, 0) \mapsto s_{\mathsf{in}}[1]$ indicates that index 0 of s_w is (transitively) assigned to the second element of s_{in}. Figure 14 defines a flow-sensitive analysis for constructing Σ, using lookup rules from Figure 13 to perform on-demand initialization of input arrays, as their sizes are not statically known. In Figure 14, the rule Asgn-Sig processes assignments to *signals*, updating the entry (s_w, φ_1) with φ_2 via a *strong update* [24], since circuit languages ensure signals are assigned only once. For data variables, the Asgn-Var rule adopts a conservative approach, handling potential aliasing by updating (υ, φ) and invalidating other entries (υ, φ_i) that might alias φ. The Cstr rule tracks equality constraints on signals s_c. The final three rules handle sequencing, conditionals, and loops, where the join operator is defined in Figure 15. Finally, the While rule requires the symbolic store Σ_1 to be a fixed-point solution for the loop.

Once the symbolic store is constructed, the IsEquivalent procedure (summarized in Figure 16) uses these symbolic expressions to deduce that two signals s_c, s_w are equivalent, denoted $\Sigma \vdash s_c \sim s_w$. According to Figure 16, signals s_c, s_w are equivalent if (a) they have the same set of written indices and (b) their values

$$\frac{\Sigma \vdash E_1 : \varphi_1 \quad \Sigma \vdash E_2 : \varphi_2}{\Sigma \vdash s_w[E_1] \leftarrow E_2 : \Sigma[(s_w, \varphi_1) \leftarrow \varphi_2]} \text{(Asgn-Sig)}$$

$$\frac{\Sigma \vdash \star : \varphi'}{\Sigma \vdash \mathsf{Havoc}(v, \varphi) : \Sigma[(v, \varphi) \leftarrow \varphi']} \text{(Havoc)}$$

$$\frac{\begin{array}{c}\Sigma \vdash E_1 : \varphi \quad \Sigma \vdash E_2 : \varphi' \\ \Theta = \{(\varphi_i \mid (v, \varphi) \in \mathsf{Dom}(\Sigma) \ \wedge \ \mathsf{SAT}(\varphi = \varphi_i)\} \\ \Sigma \vdash \mathsf{Havoc}(v, \varphi_1); \ldots; \mathsf{Havoc}(v, \varphi_{|\Theta|}) : \Sigma'\end{array}}{\Sigma \vdash v[E_1] \leftarrow E_2 : \Sigma'[(v, \varphi) \leftarrow \varphi']} \text{(Asgn-Var)}$$

$$\frac{\Sigma \vdash E : \varphi \quad \Sigma \vdash E' : \varphi' \quad \Sigma' = \Sigma[(s_c, \varphi) \leftarrow \varphi']}{\Sigma \vdash s_c[E] == E' : \Sigma'} \text{(Cstr)}$$

$$\frac{\begin{array}{c}\Sigma \vdash S_1 : \Sigma_1 \\ \Sigma_1 \vdash S_2 : \Sigma_2\end{array}}{\Sigma \vdash S_1; S_2 : \Sigma_2} \text{(Seq)} \qquad \frac{\begin{array}{c}\Sigma \vdash S_1 : \Sigma_1 \\ \Sigma \vdash S_2 : \Sigma_2 \quad \Sigma' \leftarrow \Sigma_1 \sqcup \Sigma_2\end{array}}{\Sigma \vdash \mathtt{if}(C)\ S_1\ \mathtt{else}\ S_2 : \Sigma'} \text{(If)} \qquad \frac{\begin{array}{c}\Sigma \sqsubseteq \Sigma_1 \\ \Sigma_1 \vdash S : \Sigma_2 \\ \Sigma_2 \sqsubseteq \Sigma_1\end{array}}{\Sigma \vdash \mathtt{while}(C)\ S : \Sigma_1} \text{(While)}$$

Fig. 14: Symbolic store construction. $\Sigma_1 \sqsubseteq \Sigma_2$ holds if, for all elements $(v, \varphi) \in \mathsf{Dom}(\Sigma_1)$, $\Sigma_1(v, \varphi) \sqsubseteq \Sigma_1(v, \varphi)$. $\varphi_1 \sqsubseteq \varphi_2$ holds if $\varphi_1 = \varphi_2$ or $\mathsf{Unknown}(\varphi_2)$.

$$\frac{\varphi_1 = \varphi_2}{\varphi_1 \sqcup \varphi_2 = \varphi_1} \text{(SymExp-1)} \qquad \frac{\varphi_1 \neq \varphi_2 \quad \Sigma \vdash \star : \varphi}{\varphi_1 \sqcup \varphi_2 : \varphi} \text{(SymExp-2)}$$

$$\frac{(v, \varphi) \notin \mathsf{Dom}(\Sigma_i) \quad \Sigma_i \vdash \star : \varphi'}{(\Sigma_1 \sqcup \Sigma_2)[(v, \varphi)] = \varphi'} \text{(One)} \qquad \frac{(v, \varphi) \in \mathsf{Dom}(\Sigma) \cap \mathsf{Dom}(\Sigma')}{(\Sigma \sqcup \Sigma')[(v, \varphi)] = \Sigma[(v, \varphi)] \sqcup \Sigma'[(v, \varphi)]} \text{(Both)}$$

Fig. 15: Rules defining join operation on symbolic stores

at these indices are equal. Here, we use the notation $\varphi \equiv \varphi'$ to indicate that φ and φ' are equivalent expressions (e.g., $\alpha + 1 \equiv 1 + \alpha$). The theorems stating the soundness of our analysis can be found in the extended version of the paper [25], along with their proofs.

5.2 Product Program Simplification and Instrumentation

The next step in the algorithm uses the static analysis results to simplify the product program. Figure 17 presents the Simplify procedure as inference rules $\mathcal{E} \vdash S \rightsquigarrow S'$, where S simplifies to S' under inferred equivalences $\mathcal{E}$. The procedure deletes assignments like $s_w[E] \leftarrow E$ if s_w and s_c are equivalent (S-Asg1) and substitutes all occurrences of s_c and s_w with s for any $s \in \mathcal{E}$ (S-Expr). Next, given the simplified program $\mathcal{P}_\star$, the Instrument procedure generates the instrumented program $\mathcal{P}_I$ such that $\mathcal{P}_I$ is safe if and only if the original template is correct. This procedure is summarized in Figure 18 and appends an assertion at the end of the product program stating pair-wise equivalence between the output signals. It also recursively instruments the body of $\mathcal{P}_\star$, as summarized in the first three rows of Figure 18. In particular, constraints of the form $e_1[S_c] == e_2[S'_c]$ are rewritten into an assertion $\mathtt{assert}(e_1[S_w] = e_2[S'_w])$ on the witness variables and an assumption $\mathtt{assume}(e_1[S_c] = e_2[S'_c])$ on the constraint variables. The as-

$$\frac{\begin{array}{c}\forall(s_w,\varphi)\in dom(\Sigma).\ \exists(s_c,\varphi')\in dom(\Sigma).\quad \varphi\equiv\varphi'\ \wedge\ \Sigma(s_c,\varphi)\equiv\Sigma(s_w,\varphi)\\ \forall(s_c,\varphi)\in dom(\Sigma).\ \exists(s_w,\varphi')\in dom(\Sigma).\quad \varphi\equiv\varphi'\ \wedge\ \Sigma(s_w,\varphi)\equiv\Sigma(s_c,\varphi)\end{array}}{\Sigma\vdash s_c\sim s_w}\ (\textsc{Sig-Equiv})$$

Fig. 16: Rule describing the IsEquivalent procedure

$$\frac{S=\{s\mid s\in \mathsf{Sigs}(E)\wedge s\in\mathcal{E}\}}{\mathcal{E}\vdash E\rightsquigarrow E[S/S_c,S/S_w]}\ (\textsc{S-Expr})\qquad \frac{s\in\mathcal{E}}{\mathcal{E}\vdash s[E_1]\leftarrow E_2\rightsquigarrow \texttt{skip}}\ (\textsc{S-Asg1})$$

$$\frac{s\notin\mathcal{E}\quad \mathcal{E}\vdash E_1\rightsquigarrow E_1'\quad \mathcal{E}\vdash E_2\rightsquigarrow E_2'}{\mathcal{E}\vdash s[E_1]\leftarrow E_2\rightsquigarrow s[E_1']\leftarrow s[E_2']}\ (\textsc{S-Asg2})$$

Fig. 17: Simplify procedure. $\mathsf{Sigs}(E)$ returns the normalized (i.e., without c, w subscripts) versions of the signals in E. We omit the trivial rules (e.g., for sequencing).

sumption ensures counterexamples are limited to executions where $[\![\Phi]\!](\sigma)$ holds, enforcing the antecedent in Def. 2. On the other hand, the assertion enforces that witnesses satisfy constraints $[\![\Phi]\!](\sigma')$, corresponding to the first conjunct in the consequent Def. 2. Finally, the second conjunct in the consequent of Def. 2 is enforced by the assertions placed at the end of the product program.

5.3 Verification of Instrumented Product Program

The verification algorithm (Figure 19) operates in two phases. In the first phase, it populates an environment Λ with information about loop counters, calling InferBounds (line 7) to compute lower and upper bounds for each counter κ via interval analysis [26] and invoking InferStepSize to determine κ's step size. The second phase (lines 9–15) computes inductive loop invariants, initializing them in the first iteration and weakening each invariant until soundness is ensured. Specifically, for each loop L, InferLoopInv is invoked at line 13 to compute an inductive invariant for L *assuming* that $\Psi[L]$ over-approximates L's true precondition. These loop pre-conditions (Ψ) are computed through standard post-condition computation, using invariants $\mathcal{I}$ for preceding loops. However, since the "invariants" computed in the first iteration may be stronger than the true invariant, these pre-conditions may also be too strong. Hence, the algorithm weakens both the pre-conditions and the invariants until a fixed-point is reached. Upon convergence, it generates verification conditions and checks their validity.

Inferring loop invariants. Figure 20 outlines our procedure for generating loop invariants, based on a standard Houdini-style approach [27] with two distinguishing features. First, it uses a novel technique (ConjecturePredicates) to construct the predicate universe; second, it includes a boolean parameter check-pre to control whether predicates are filtered using pre-condition ψ. Dur-

$$\begin{array}{ll}\mathsf{Instr}(e_1[S_c]==e_2[S_c']) & \triangleq \mathtt{assume}(e_1[S_c]=e_2[S_c'])\mathtt{assert}(e_1[S_w]=e_2[S_w']);\\ \mathsf{Instr}(s[E_1]\leftarrow E_2) & \triangleq s[E_1]\leftarrow E_2\\ \mathsf{Instr}(S_1;S_2) & \triangleq \mathsf{Instr}(S_1);\mathsf{Instr}(S_2)\\ \textsc{Instrument}(P_\otimes) & \triangleq \mathsf{Instr}(P_\otimes);\mathtt{assert}(\forall s\in\mathsf{OutputSignals}(P_\otimes).\ s_w\equiv s_c)\end{array}$$

Fig. 18: Instrumentation procedure, Instrument

```
1:  procedure VerifyProduct($P_\otimes$ )
2:      # $\mathcal{I}, \Psi$ map each loop to their invariant and pre-conditions respectively
3:      $(\mathcal{I}, \Psi) \leftarrow (\varnothing, \varnothing)$; $\Lambda \leftarrow \emptyset$
4:      # Phase 1: Infer information about loop counters
5:      for $L \in \mathsf{Loops}(P_\otimes)$ do
6:          $(\kappa, \mathcal{I}[L]) \leftarrow (\mathsf{InferLoopCounter}(L), \mathsf{true})$
7:          $\Lambda[\kappa] \leftarrow (\mathsf{InferBounds}(\kappa, L), \mathsf{InferStepSize}(\kappa, L))$
8:      # Phase 2: Weaken invariants until they are sound
9:      $\mathsf{done} \leftarrow \mathsf{false}; \mathsf{first} \leftarrow \mathsf{true}$
10:     while $\neg\mathsf{done}$ do
11:         $(\Psi, \mathsf{done}) \leftarrow (\mathsf{ComputeSP}(P_\otimes, \mathcal{I}), \mathsf{true})$
12:         for $L \in \mathsf{Loops}(P_\otimes)$ do
13:             $I \leftarrow$ InferLoopInv$(L, \Psi[L], \Lambda, \neg\mathsf{first})$
14:             if $I \neq \mathcal{I}[L]$ then $\mathsf{done} \leftarrow \mathsf{false}; \mathcal{I}[L] \leftarrow I$
15:         $\mathsf{first} \leftarrow \mathsf{false}$
16:     # Generate VCs and check their validity
17:     return $\mathsf{Valid}(\mathsf{GenVC}(P_\otimes, \mathcal{I}))$
```

Fig. 19: Verification algorithm; procedures in sans serif font are explained in text.

```
1:  procedure InferLoopInv($L$, $\psi$, $\Lambda$, check-pre)
    input: Loop $L$ with condition $C$ and body $B$, pre-condition $\psi$,
2:          counter environment $\Lambda$, boolean check-pre
3:      # Conjecture predicates that are likely to be part of the invariant
4:      $\Pi \leftarrow$ ConjecturePredicates$(L, \Lambda)$
5:      # Remove predicates not implied by pre-condition
6:      if check-pre then $\Pi \leftarrow \{\varphi \mid \varphi \in \Pi \ \wedge \ \psi \Rightarrow \varphi\}$
7:      # Houdini-style fixed-point computation; done initialized to false
8:      while $\neg\mathsf{done}$ do
9:          $\mathsf{done} \leftarrow \mathsf{true}$
10:         for $\varphi \in \Pi$ do
11:             if $\not\vdash \{C \wedge \bigwedge_{p_i \in \Pi} p_i\}\ B\ \{\varphi\}$ then $\Pi \leftarrow \Pi \backslash \{\varphi\}; \mathsf{done} \leftarrow \mathsf{false}$
12:     return $\bigwedge_{p \in \Pi} p$
```

Fig. 20: Invariant inference algorithm. If Π is empty, then we define $\bigwedge_{p \in \Pi} p$ to be True.

ing the first invocation, check-pre is **false**, so the inferred invariant could be too strong (satisfying consecution but not initiation). In subsequent calls, check-pre is **true**, ensuring that predicates failing initiation are eventually filtered out.

ConjecturePredicates (Figure 21) generates candidate predicates for a loop L, returning a set of *guarded array equality* (GAE) predicates which are of the form $\forall \overline{x}.\ \phi(\overline{x}) \rightarrow s_c[f(\overline{x})] = s_w[f(\overline{x})]$. To explain our algorithm, we utilize the example in Figure 22 for which the required invariant is:

$$\forall x, y.\ (0 \leq x < i \leq N \wedge 0 \leq y < M) \vee (x = i \wedge 0 \leq y < j \leq M) \rightarrow s_w[N * x + y] = s_c[N * x + y]$$

Here, the guard captures all loop counter values c_x, c_y for which $s_w[N * c_x + c_y]$ was written before the current iteration. To infer such predicates, the al-

```
for (var i = 0; i < N; i++) {
 for (var j = 0; j < M; j++) {
   s_w[N*i + j] <-- a;
   s_c[N*i + j] == a;}}
```

Fig. 22: Loop Example

```
1: procedure ConjecturePredicates($L, \Lambda$)
   input: Loop $L$ and counter environment $\Lambda$
   output: Candidate guarded array equality predicates $\Pi$
2:     $\mathcal{W} \leftarrow \mathsf{SignalWrites}(L) \quad \Pi \leftarrow \{\}$;
3:     for $(s_w, f(\kappa_1, \dots, \kappa_n)) \in \mathcal{W}$ do
4:         # Infer constraints over counters $\kappa_1, \dots, \kappa_n$, sorted outermost to innermost
5:         $(\varphi, \varphi_a) \leftarrow (\mathsf{false}, \mathsf{true})$
6:         for $\kappa_i \in [\kappa_1, \dots, \kappa_n]$ do
7:             # generate constraints $\varphi_{\kappa_i}$ on $\kappa_i$ assuming $\varphi_a = \bigwedge_{i=1}^{n} x_i = \kappa_i$
8:             $(l_i, u_i, s_i) \leftarrow \Lambda[\kappa_i]$
9:             $\varphi_{\kappa_i} \leftarrow \phi_a \wedge (l_i \leq x_i < \kappa \leq u_i \wedge x_i - l_i \textbf{ mod } s_i = 0)$
10:            for $\kappa_j \in [\kappa_{i+1}, \dots, \kappa_n]$ do
11:                $(l_j, u_j, s_j) \leftarrow \Lambda[\kappa']$
12:                $\varphi_{\kappa_i} \leftarrow \varphi_{\kappa_i} \wedge (l_j \leq x_j < u_j \wedge x_j - l_j \textbf{ mod } s_j = 0)$
13:            # Add constraints $\varphi_{\kappa_i}$ to $\varphi$ and add assumption $x_i = \kappa_i$ to $\varphi_a$
14:            $(\varphi, \varphi_a) \leftarrow (\varphi \vee \varphi_\kappa, \varphi_a \wedge x_i = \kappa_i)$;
15:        $\Pi \leftarrow \Pi \cup \{\forall x_1, \dots, x_n.\ \varphi \rightarrow s_w[f(x_1, \dots, x_n)] = s_c[f(x_1, \dots, x_n)]\}$
16:    return $\Pi$
```

Fig. 21: Algorithm for generating guarded array equality predicates.

gorithm first identifies all writes in the loop (line 2),
represented as tuples $(s_w, f(\kappa_i, ..., \kappa_j))$, where $f(\kappa_i, ..., \kappa_j)$ is the symbolic expression for the index written in s_w. In this example, the only write is to s_w at $N * i + j$, so the algorithm conjectures $s_w[N * i + j] = s_c[N * i + j]$ as the array equality component of the GAE predicate.

For each inferred array equality predicate, the algorithm constructs its guard φ by iterating over loop counter variables from outermost to innermost. For each counter κ_i, it synthesizes a predicate φ_{κ_i} capturing all feasible values for counters $[\kappa_i, \dots, \kappa_n]$, assuming parent counters hold their current iteration values (φ_a). Lines 10–14 compute φ_{κ_i} using loop bounds and step sizes from Λ, ensuring the predicate accounts for all inner counter values during successful executions of prior iterations. The final guard φ is updated as $\varphi \vee \varphi_{\kappa_i}$, while φ_a adds the assumption $x_i = \kappa_i$. Going back to the example, the counter environment maps i to $[0, N)$ and step size 1, and j to $[0, M)$ and step size 1. The procedure first generates φ_i as $0 \leq x_i < i \leq N \wedge 0 \leq x_j < M$. Next, it updates φ_a with $x_i = i$ and computes φ_j for the inner loop as $x_i = i \wedge 0 \leq x_j < j \leq M$. Finally, φ becomes $\varphi_i \vee \varphi_j$, yielding the desired guard:

$$(0 \leq x_i < i \leq N \wedge 0 \leq x_j < M) \vee (x_i = i \wedge 0 \leq x_j < j \leq M)$$

We refer the interested reader to the extended version of the paper [25] for an example demonstrating a more complex loop invariant.

6 Implementation

We implemented our approach in a tool called Zequal and built it on top of the Circom compiler [28]. Zequal discharges SMT queries via Z3 [29] in the theory of integers. Finite field operations are modeled by performing all operations modulo the BN254 prime. For expressions composed of additions, subtractions, or multiplications, the modulo reduction is moved to the outermost expression to reduce the number of large mod operations. In cases where an operation is not supported by the theory of integers (e.g. bitwise arithmetic, exponentiation, finite field inverse), we model these operations with uninterpreted functions and include appropriate axioms.

Although Section 5 describes our procedure as operating on a single product program, real-world circuits and witness generators are often composed of subcircuit invocations. ZEQUAL verifies such circuits by assuming the consistency of any invoked subcircuit. A circuit can therefore be verified by checking the consistency of all invoked sub-circuits in addition to the circuit itself.

7 Evaluation

In this section, we present the results of an experimental evaluation that is designed to answer the following research questions: (1) Can ZEQUAL be used to verify real-world circuits? (2) Does ZEQUAL uncover buggy ZK circuit templates in the wild? (3) How important is the proposed static analysis for successful verification? (4) Can static analysis be used *on its own* to prove consistency? To answer questions (1) and (2), we analyze what percentage of circuits ZEQUAL can verify/falsify. To answer the latter two questions, we perform two ablation studies that disable static analysis and deductive verification respectively.

Benchmarks. We evaluate ZEQUAL on Circom programs gathered from the 20 most popular Circom projects (measured by GitHub stars). After excluding four repositories that use unsupported features (e.g., incompatible with ZEQUAL's version of the Circom compiler), we retain 464 templates from 16 unique repositories, spanning over 290K lines of code. Collectively, these projects span a variety of applications, including cryptographic primitives, games, and machine learning components. For all projects, ZEQUAL attempts to verify the consistency of each template declared within the source code. Since all templates rely on primitives provided by circomlib, which serves as a foundational library for Circom development, ZEQUAL employs stubs of circomlib circuits when analyzing other projects. These stubs capture the behavior of circomlib components, enabling ZEQUAL to reason about templates in other repositories without requiring their full implementation. To evaluate ZEQUAL on these benchmarks, we make *minor* semantics-preserving modifications to a small subset of the templates in order to ensure that they comply with ZEQUAL's assumptions (e.g., our implementation assumes that array sizes only reference template parameters rather than local variables).

Experimental Setup. The experiments were conducted on a MacBook Pro® with an M1 Max CPU, 64 GB of memory, and MacOS 12.3.1. Each benchmark was allocated a runtime of 10 minutes. Each experiment requires verifying the correctness of a specific template, assuming all external templates are correct.

Main Results. Table 1 summarizes the results of our evaluation. Out of 464 benchmarks, ZEQUAL successfully verifies the correctness of 306 templates, meaning it can verify correctness for *any* instantiation of the templates in 66% of the

Project	# templates	LoC	# verified	# cex	# failed	Verif. time
tornado-core	5	102	5 (100%)	0 (0%)	0 (0%)	0.40
semaphore	2	50	2 (100%)	0 (0%)	0 (0%)	2.41
circomlib	103	28,660	69 (67%)	24 (23%)	10 (10%)	7.28
maci	52	2,171	40 (77%)	9 (17%)	3 (6%)	1.74
stealthdrop	36	161,251	17 (47%)	13 (36%)	6 (17%)	13.58
circom-ecdsa	63	68,359	36 (57%)	22 (35%)	5 (8%)	1.65
hydra-s1-zkps	4	152	4 (100%)	0 (0%)	0 (0%)	2.31
zkshield	5	238	2 (40%)	1 (20%)	2 (40%)	6.14
circomlib-ml	37	897	34 (92%)	3 (8%)	0 (0%)	5.75
eigen-zkvm	21	23,856	11 (52%)	6 (29%)	4 (19%)	10.85
zk-nullifier-sig	44	1,762	24 (55%)	13 (30%)	7 (16%)	16.49
tornado-nova	7	190	5 (71%)	2 (29%)	0 (0%)	1.77
darkforest-eth	22	699	13 (59%)	8 (36%)	1 (5%)	3.84
ed25519-circom	20	2,186	8 (40%)	10 (50%)	2 (10%)	3.86
unirep	14	595	12 (86%)	1 (7%)	1 (7%)	1.42
zk-hunt	29	995	24 (83%)	4 (14%)	1 (3%)	1.68
Overall	464	292,163	306 (66%)	116 (25%)	42 (9%)	5.71

Table 1: Results. The column called "#verified" shows the number of benchmarks for which Zequal is able to prove correctness, "# cex" shows the number of benchmarks for which Zequal is unable to prove the validity of the generated VC and "# failed" shows the number of benchmarks for which Zequal times out or returns unknown. The last column called "verif. time" shows the average running time of Zequal for those benchmarks that were successfully verified.

benchmarks. Zequal is also highly efficient, with an average verification time of just 5.71 seconds per template. For the remaining 158 benchmarks, verification does not succeed. In 42 cases (9%), Zequal exceeds the 10-minute time limit or the tool returns unknown, primarily due to the Z3 timing out on SMT queries. In the remaining 116 benchmarks (25%), Zequal identifies counterexamples to the validity of the generated VC. These counterexamples occur for two reasons: (1) some template instantiations are genuinely incorrect, or (2) the template is correct, but verification fails due to analysis imprecision, such as weak loop invariants.

Failure analysis. Next, we examine the scenarios in which Zequal is unable to prove correctness. The findings from this failure analysis are summarized in Figure 23. Among the cases where verification fails, 22% are true positives, which are discussed in more detail below. For the remaining benchmarks, Zequal fails to verify correctness for four primary reasons:

Reason for failure	# (%)
True positive	26 (22%)
Insufficient loop invariant	61 (53%)
Imprecise loop bound	10 (9%)
Imprecise template stub	8 (7%)
Imprecise operation modeling	11 (9%)

Fig. 23: Results of the failure analysis.

– *Insufficient loop invariant:* For 61 of the failure cases (53% of false positives), the loop invariant inferred by Zequal is too weak. For instance, several benchmarks require invariants that capture precise valuations of s_c and s_w

to establish the consistency of a signal s across *multiple* loops. We refer the interested reader to the extended version of the paper [25] for a representative example.

– *Imprecise loop bounds:* Recall that our approach relies on static analysis of loop counters to infer bounds and step sizes as a precursor to loop invariant generation. 10 of the failure cases (i.e., 9% of false positives) are due to imprecision in the static analysis of loop counters.
– *Imprecise template stubs:* Recall that our approach utilizes stubs of circomlib templates when analyzing other projects. In some cases, the stubs are not sufficiently precise, accounting for 7% of false positives.
– *Imprecise modeling:* As discussed in Section 6, Zequal uses uninterpreted functions to model operations that are not supported by the theory of integers. Roughly 9% of false positives are caused by imprecision in the SMT encoding.

True positive analysis. Among the 116 potentially vulnerable circuits we manually inspected, 26 were confirmed to correspond to real bugs. However, since Zequal assumes that external template invocations are safe, these 26 templates could also have cascading effects on other benchmarks. We therefore reran Zequal without this assumption, and identified an additional 20 buggy benchmarks. Hence, Zequal identified a total of 46 buggy templates in the wild.

Of the 46 falsified templates, 41 exhibit violations of the determinism property explored in prior work [8, 10] under *some* instantiation of the template parameters. On the other hand, 5 benchmarks are incorrect despite being deterministic and fail to properly validate their inputs. In practice, an attacker can easily take advantage of such bugs in the validation logic to trick the verifier into accepting proofs that should be classified as invalid. An example of an incorrect but deterministic benchmark can be found in the extended version of the paper [25].

Comparison with prior work . As discussed previously, 41 out of the 46 incorrect templates also violate the determinism property studied in prior work [8, 10]. Therefore, *in principle*, these 41 bugs could be uncovered by an existing tools like QED^2 and CIVER, which specialize in refuting determinism. However, existing tools for determinism checking are limited to analyzing fully instantiated circuits and cannot directly handle *templates*.

To evaluate how Zequal compares with existing determinism checkers, we instantiated the 41 non-deterministic templates with the *smallest* parameter values that expose the bug. This approach avoids scalability issues caused by larger parameter values while ensuring that the bug remains detectable. Using state-of-the-art tools, QED^2 and CIVER, we then attempted to identify the non-determinism bugs within a time limit of 10 minutes. Among the 41 non-deterministic circuits, QED^2 successfully detected the issue in 22 (54%) circuits but failed to do so for the remaining 19 (46%) cases. Similarly, CIVER successfully detected the issue in 27 (66%) circuits but failed to do so for the remaining 14 (34%) cases.

These results highlight a significant limitation of existing open source determinism checking tools: their ability to refute determinism diminishes in practice due to scalability challenges, even when working with specific circuit instantiations. In contrast, ZEQUAL directly analyzes templates and verifies a much stronger property for *any* instantiation, without requiring manually-crafted parameter templates. However, ZEQUAL cannot automatically refute correctness, even when restricted to determinism, making it a complementary approach rather than a replacement.

Ablation Study. To assess the significance of the key design choices in ZEQUAL, we compare its performance against two ablated versions:

– ZEQUAL-NSA: This variant disables the static analysis described in Section 5.1. Without static analysis, ZEQUAL-NSA bypasses simplifications of the product program and directly invokes the verification procedure. This evaluation highlights the contribution of static analysis in reducing verification complexity.
– ZEQUAL-OSA: This variant relies solely on the static analysis results to determine correctness. Specifically, it uses the ISEQUIVALENT procedure to verify whether *all* signals in the ZK circuit are equivalent. If this equivalence holds, all instrumented assertions reduce to *true*, and the circuit is deemed correct. This approach examines the standalone utility of static analysis without leveraging the full verification pipeline in ZEQUAL.

The results of the ablation study, shown in Figure 24, highlight the importance of both the static analysis and the proposed verification pipeline. Specifically, ZEQUAL-NSA verifies over 100 fewer benchmarks than ZEQUAL due to the increased complexity of the product program, which leads to a significantly higher number of timeouts. On the other hand, ZEQUAL-OSA is much faster since it relies solely on lightweight static analysis, but it verifies only half as many benchmarks as ZEQUAL. These results demonstrate that static analysis alone is insufficient for proving correctness, reinforcing the necessity of the full verification pipeline.

Tool	**# verified**
ZEQUAL-NSA	205 (44%)
ZEQUAL-OSA	150 (32%)
ZEQUAL	306 (66%)

Fig. 24: Ablation study

8 Related Work

Formal Verification of ZK Circuits. Recent work has explored using formal verification to check the correctness of ZK circuits [8, 10–12, 30–33]. One line of work uses interactive theorem provers to verify functional correctness [12, 30, 31]. For example, CODA [12] uses a custom circuit DSL embedded in Coq which is equipped with a rich type system for specifying circuit properties. Its type checker uses a combination of tactics along with manually proved lemmas to check correctness in Coq. Another line of work uses SMT solvers for automated verification [8, 10, 11, 32, 33], focusing on specific circuit properties rather than full functional correctness. The most related work to ours is QED^2 [8], which checks the determinism of ZK circuits using a combination of static analysis and SMT solvers like ZEQUAL. However, unlike ZEQUAL, QED^2 checks a weaker property and is limited to instantiated circuits (without loops), whereas ZEQUAL can directly analyze circuit templates.

Relational Verification. Checking relational properties between programs often involves building product programs [34–38], and a key challenge is determining a suitable alignment between programs. For instance, Churchill et al. [35] check program equivalence by using manually provided tests to generate program traces and synthesize alignment predicates from the trace states. They then lift the alignment predicates back to the source code to construct the product program. In contrast, ZEQUAL leverages the natural alignment between constraints and witness generators in circuit DSLs and performs static analysis to substantially simplify the product program.

Array Invariants. There is an extensive body of work on generating quantified invariants for loops with array variables [39–44]. For example, Flanagan et al. [40] use predicate abstraction and Skolemization to construct quantified invariants, while Gulwani et al. [42] employ a quantified abstract domain to express universally quantified formulas over arrays. Although these techniques could be applied in our setting, we leverage a domain-specific insight: Most loop invariants express equality between the witness and constraint components of signal arrays. This allows us to generate high-quality templates and efficiently construct loop invariants in most cases.

9 Conclusion

This paper presents ZEQUAL, a framework for verifying the consistency of zero-knowledge circuit templates by combining static analysis and deductive verification. ZEQUAL formalizes the consistency requirement for circuit DSLs and introduces a verification method that directly operates on templates, providing guarantees for any circuit instantiated from them. We have evaluated the proposed approach through an extensive experimental study on 464 real-world benchmarks drawn from popular Circom projects. The results show that ZEQUAL terminates on 91% of the benchmarks within a 10-minute time limit and successfully verifies 72% of the cases where it terminates. For benchmarks where verification fails, a systematic failure analysis reveals that roughly 22% of counterexamples correspond to true positives. ZEQUAL also uncovers several bugs that a state-of-the-art tool, QED^2, cannot detect. This capability arises from two key advantages: First, ZEQUAL is not limited to verifying determinism, and, second, it can reason about all possible instantiations of the template in a scalable manner. Additionally, ablation studies validate the design choices behind ZEQUAL, showing that the combination of lightweight static analysis and deductive verification is critical for making the approach effective.

Acknowledgments. This work was conducted in a research group supported by NSF awards CCF-1762299, CCF-1918889, CNS-1908304, CCF-1901376, CNS-2120696, CCF- 2210831, and CCF-2319471.

Disclosure of Interests. All authors are employed by Veridise, a company that provides formal verification services. The authors have no other competing interests to declare that are relevant to the content of this article.

Bibliography

[1] J. Groth, "Short non-interactive zero-knowledge proofs," in *Advances in Cryptology - ASIACRYPT 2010* (M. Abe, ed.), (Berlin, Heidelberg), pp. 341–358, Springer Berlin Heidelberg, 2010.

[2] E. Ben-Sasson, I. Bentov, Y. Horesh, and M. Riabzev, "Scalable, transparent, and post-quantum secure computational integrity." Cryptology ePrint Archive, Paper 2018/046, 2018.

[3] K. W. Jie, "Openclimate-sg/datawhistleblowing: Work done for the yale open climate collabathon."

[4] T. Datta, B. Chen, and D. Boneh, "VerITAS: Verifying image transformations at scale." Cryptology ePrint Archive, Paper 2024/1066, 2024.

[5] E. Ben Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza, "Zerocash: Decentralized anonymous payments from bitcoin," in *2014 IEEE Symposium on Security and Privacy*, pp. 459–474, 2014.

[6] C. Smith, "Zero-knowledge rollups."

[7] S. Chaliasos, J. Ernstberger, D. Theodore, D. Wong, M. Jahanara, and B. Livshits, "Sok: What don't we know? understanding security vulnerabilities in snarks," 2024.

[8] S. Pailoor, Y. Chen, F. Wang, C. Rodríguez, J. V. Gaffen, J. Morton, M. Chu, B. Gu, Y. Feng, and I. Dillig, "Automated detection of underconstrained circuits for zero-knowledge proofs." Cryptology ePrint Archive, Paper 2023/512, 2023.

[9] 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." Cryptology ePrint Archive, Paper 2023/190, 2023.

[10] M. Isabel, C. Rodriguez-Nunez, and A. Rubio, " Scalable Verification of Zero-Knowledge Protocols ," in *2024 IEEE Symposium on Security and Privacy (SP)*, (Los Alamitos, CA, USA), pp. 1794–1812, IEEE Computer Society, May 2024.

[11] F. H. Soureshjani, M. Hall-Andersen, M. Jahanara, J. Kam, J. Gorzny, and M. Ahmadvand, "Automated analysis of halo2 circuits." Cryptology ePrint Archive, Paper 2023/1051, 2023.

[12] 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 *2024 IEEE Symposium on Security and Privacy (SP)*, pp. 1741–1759, 2024.

[13] B. Parno, C. Gentry, J. Howell, and M. Raykova, "Pinocchio: Nearly practical verifiable computation." Cryptology ePrint Archive, Paper 2013/279, 2013. https://eprint.iacr.org/2013/279.

[14] V. Buterin, "Quadratic arithmetic programs: From zero to hero." https://medium.com/@VitalikButerin/quadratic-arithmetic-programs-from-zero-to-hero-f6d558cea649, Dec 2016.

[15] ConsenSys, *gnark: A Framework for Zero-Knowledge Proofs*, 2024. Documentation for circuit concepts in gnark.

[16] 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, 2022.

[17] "halo2 - the halo2 book." https://zcash.github.io/halo2/index.html.

[18] iden3, "Iden3/snarkjs: Zksnark implementation in javascript and wasm." https://github.com/iden3/snarkjs.

[19] G. Barthe, J. M. Crespo, and C. Kunz, "Relational verification using product programs," in *International Symposium on Formal Methods*, pp. 200–214, Springer, 2011.

[20] S. K. Lahiri and S. Qadeer, "Complexity and algorithms for monomial and clausal predicate abstraction," in *International Conference on Automated Deduction*, pp. 214–229, Springer, 2009.

[21] RISC Zero, "ZirGen: Risc zero circuit generator." https://github.com/risc0/zirgen, 2024. Accessed: 2024-11-01.

[22] ZoKrates Team, *ZoKrates: A Toolbox for zkSNARKs on Ethereum*, 2024. Official Documentation.

[23] J. Ax, "The elementary theory of finite fields," *Annals of Mathematics*, vol. 88, no. 2, pp. 239–271, 1968.

[24] I. Dillig, T. Dillig, and A. Aiken, "Fluid updates: Beyond strong vs. weak updates," in *Programming Languages and Systems: 19th European Symposium on Programming, ESOP 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings 19*, pp. 246–266, Springer, 2010.

[25] J. Stephens, S. Pailoor, and I. Dillig, "Automated verification of consistency in zero-knowledge proof circuits." Cryptology ePrint Archive, Paper 2025/916, 2025. https://eprint.iacr.org/2025/916.

[26] 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 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pp. 238–252, 1977.

[27] C. Flanagan and K. R. M. Leino, "Houdini, an annotation assistant for esc/java," in *International Symposium of Formal Methods Europe*, pp. 500–517, Springer, 2001.

[28] "Circom compiler." https://github.com/iden3/circom.

[29] L. de Moura and N. Bjørner, "Z3: An efficient smt solver," in *Tools and Algorithms for the Construction and Analysis of Systems* (C. R. Ramakrishnan and J. Rehof, eds.), (Berlin, Heidelberg), pp. 337–340, Springer Berlin Heidelberg, 2008.

[30] A. Coglio, E. McCarthy, E. Smith, C. Chin, P. Gaddamadugu, and M. Dellepere, "Compositional formal verification of zero-knowledge circuits." Cryptology ePrint Archive, Paper 2023/1278, 2023.

[31] C. Kwan, Q. Dao, and J. Thaler, "Verifying jolt zkVM lookup semantics." Cryptology ePrint Archive, Paper 2024/1841, 2024.

[32] M. Stronati, D. Firsov, A. Locascio, and B. Livshits, “Clap: a semantic-preserving optimizing edsl for plonkish proof systems,” 2024.

[33] H. Chen, G. Li, M. Chen, R. Liu, and S. Gao, “Ac4: Algebraic computation checker for circuit constraints in zkps,” 2024.

[34] R. Sharma, E. Schkufza, B. Churchill, and A. Aiken, “Data-driven equivalence checking,” *SIGPLAN Not.*, vol. 48, p. 391–406, Oct. 2013.

[35] B. Churchill, O. Padon, R. Sharma, and A. Aiken, “Semantic program alignment for equivalence checking,” in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, (New York, NY, USA), p. 1027–1040, Association for Computing Machinery, 2019.

[36] M. Dahiya and S. Bansal, “Black-box equivalence checking across compiler optimizations,” in *Programming Languages and Systems* (B.-Y. E. Chang, ed.), (Cham), pp. 127–147, Springer International Publishing, 2017.

[37] R. Dickerson, P. Mukherjee, and B. Delaware, “Kestrel: Relational verification using e-graphs for program alignment,” 2024.

[38] J. Chen, J. Wei, Y. Feng, O. Bastani, and I. Dillig, “Relational verification using reinforcement learning,” *Proc. ACM Program. Lang.*, vol. 3, Oct. 2019.

[39] S. Gulwani and A. Tiwari, “An abstract domain for analyzing heap-manipulating low-level software,” in *Proceedings of the 19th International Conference on Computer Aided Verification*, CAV’07, (Berlin, Heidelberg), p. 379–392, Springer-Verlag, 2007.

[40] C. Flanagan and S. Qadeer, “Predicate abstraction for software verification,” *SIGPLAN Not.*, vol. 37, p. 191–202, Jan. 2002.

[41] D. Gopan, T. Reps, and M. Sagiv, “A framework for numeric analysis of array operations,” *SIGPLAN Not.*, vol. 40, p. 338–350, Jan. 2005.

[42] S. Gulwani, B. McCloskey, and A. Tiwari, “Lifting abstract interpreters to quantified logical domains,” *SIGPLAN Not.*, vol. 43, p. 235–246, Jan. 2008.

[43] L. Kovács and A. Voronkov, “Finding loop invariants for programs over arrays using a theorem prover,” in *Fundamental Approaches to Software Engineering* (M. Chechik and M. Wirsing, eds.), (Berlin, Heidelberg), pp. 470–485, Springer Berlin Heidelberg, 2009.

[44] I. Dillig, T. Dillig, and A. Aiken, “Symbolic heap abstraction with demand-driven axiomatization of memory invariants,” *SIGPLAN Not.*, vol. 45, p. 397–410, Oct. 2010.