2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)

ScaleCirc: Scaling the Analysis over Circom Circuits

Jinan Jiang
Hong Kong Polytechnic University
jinan.jiang@connect.polyu.hk

Haoran Qin
Hong Kong Polytechnic University
hao-ran.qin@connect.polyu.hk

Xiapu Luo
Hong Kong Polytechnic University
csxluo@comp.polyu.edu.hk

***Abstract*—Zero-knowledge proof (ZKP) circuits implemented in programming languages like Circom are fundamental to blockchain and privacy-preserving applications. These code often suffer from constraint-related issues where constraints fail to accurately specify intended computations. While existing analysis tools have been proposed, they struggle with large-scale circuits containing complex template embeddings. We present SCALECIRC, a novel framework that addresses such limitations through: 1) systematic management of analysis redundancy via circuit deduplication strategies; 2) constrainedness propagation methods leveraging source code semantic information; and 3) a generalizable framework for different circuit analysis tasks. Evaluation on 691 real-world circuits shows SCALECIRC demonstrates higher efficiency, and successfully analyzes many Circom programs that existing works failed on.**

I. INTRODUCTION

Zero-knowledge proof (ZKP) circuits, implemented as specialized software primitives, are fundamental components in blockchain and privacy-preserving applications. A ZKP allows one party to prove to another that a statement is true without revealing any additional private information beyond the statement's validity. For example, a user could prove they are over 18 years old by presenting a ZKP of their birth date, without revealing their actual birth date or any other personal information. One notable software implementation of ZKP is zkRollups [1]–[4], a scaling solution for increasing transaction throughput in blockchain systems. In zkRollups, multiple transactions are processed privately and a ZKP that proves the correctness of all transaction executions is provided. Users can verify this proof to trust the execution results without needing access to the transaction details or intermediate computational states. The ZKP software ecosystem has expanded to enable various privacy-preserving applications, such as verification of identity [5], voting systems [6], [7], proving geographical location [8], machine learning (ML) model training [9], ML inference [10]–[14], etc.

Circom has emerged as one of the most popular programming languages for specifying ZKP circuits. These circuits compile into R1CS (Rank-1 Constraint System) format, a system of finite field polynomial constraints that encode the relationships between circuit signals and specify correctness requirements. Circom circuits are increasingly deployed in critical applications handling financial value or sensitive data, making it critical to ensure their correctness. *Under-constrained circuits* is the most commonly found [15] issue associated with real-world Circom code. It occurs when the constraints specified

Corresponding author: Xiapu Luo.

```
 1 template Sha256_2() {
 2     signal input a;
 3     component bits2num = Bits2Num(216);
 4     component num2bits[2];
 5     num2bits[0] = Num2Bits(216);
 6     num2bits[1] = Num2Bits(216);
 7     component s = Sha256compression();
 8     s.inp[432] <== 1;
 9     ...}
10 template Sha256compression() {
11     signal input hin[256];
12     signal output out[256];
13     var i; component t2[64];
14     for (i=0; i<64; i++) t2[i] = T2();
15     component suma[64];
16     for (i=0; i<64; i++) suma[i] = BinSum(32,2);
17     component sume[64];
18     for (i=0; i<64; i++) sume[i] = BinSum(32,2);
19     component fsum[8];
20     for (i=0; i<8; i++) fsum[i] = BinSum(32,2);
21     ...}
22 template T2() {
23     signal input a[32];
24     component sum = BinSum(32, 2);
25     ...}
```

Fig. 1: A simplified example Circom circuit [21].

by Circom programmers fail to fully capture the intended computations, allowing some fake proofs to pass verification. This type of bug can be exploited by malicious actors, and multiple real-world exploits have been reported [16]–[20], resulting in huge financial losses. Therefore, it is important to be able to automatically verify the correctness of Circom code.

Multiple existing tools [22]–[26] have been proposed to analyze Circom code for constraint-related issues, leveraging approaches such as formal methods, pattern matching, and graph-based methods. However, they face several limitations:

Limitation 1 (L1): Existing works analyze the entire compiled R1CS circuit without leveraging the inherent structural redundancy in Circom code. Consider the motivating example shown in Figure 1, which shows a simplified SHA-256 hash circuit from `circomlib` [21], a widely-adopted library. The circuit exhibits both: (1) *intra-circuit redundancy*, where identical components are repeated within the same template level, such as the 128 copies of `BinSum(32, 2)` in `suma` and `sume` arrays in lines 15-18 and 8 copies in the `fsum` array (lines 19-20); and (2) *inter-circuit redundancy*, where identical components are repeated across different template embedding layers, such as the 64 copies of `T2` (lines 13-14), each containing one `BinSum(32, 2)` (lines 22-24). Overall, over 200 identical `BinSum` components are instantiated within

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00124

a single SHA-256 hash circuit, but current analysis tools compile all these duplicate components into over 200,000 R1CS polynomial constraints, and perform full-scale analysis (e.g., SMT queries) over all of them. This overwhelms their SMT solvers (e.g., Z3 [27], CVC5 [28]), leading to slower analysis and often timeouts that leave many real-world circuits unverifiable. These redundancies are commonly found in real-world Circom code. Managing this redundancy is a challenge as it exists across multiple template layers through recursive embeddings, and any deduplication effort that modifies the circuit must also preserve the signal constrainedness properties across the entire compiled R1CS circuit.

Limitation 2 (L2): Current under-constrained circuit detection tools for Circom operate at the compiled R1CS level, discarding crucial semantic information from source code. When compiling Circom code to mathematical R1CS polynomials, important contexts (e.g., variable types, operation semantics, control flow, circuit embedding relationships) are eliminated, which could otherwise provide informative context for circuit analysis [29], [30]. Unlike R1CS that follows a uniform polynomial structure, analyzing source code presents challenges due to its complex programming constructs and grammar patterns.

Limitation 3 (L3): Existing approaches struggle to scale with circuit complexity, with lower success rate on larger circuits. For instance, Picus's solved rate falls from 84% for small circuits ($<$100 constraints) to 20% on larger ones ($\geq$1000 constraints) on the classical `circomlib` benchmark [25]. Modern Circom circuits often contain hundreds of thousands of constraints with complex signal wirings, making them difficult to analyze efficiently. For example, the SHA-256 circuit in Figure 1 contains over 200,000 constraints with complex template embedding relationships, rendering all existing works fail to report an under-constrainedness analysis result on it.

In this paper, we present SCALECIRC, a comprehensive approach to scaling the analysis of Circom code. Our approach introduces a source code-based general analysis framework and effective management of circuit analysis result reuse. We handle both intra- and inter-circuit redundancy via novel circuit deduplication approaches (addressing L1), and introduce source code-level propagation of constrainedness (addressing L2). Experimental results show that our approach successfully analyzes many large circuits that existing works failed on. For example, on the classical `circomlib` benchmark, SCALECIRC successfully analyzes 98% of the small circuits and 72% of the large ones, outperforming existing methods (e.g., Picus solved only 84% and 20%, repectively) (addressing L3).

Overall, we extract a simplified DSL for Circom (§III-B), and develop a generic analysis framework (§ III-C). We then detail our implementation and inference rules for under-constrainedness analysis (§III-D), and demonstrate SCALECIRC's generalizability through two additional circuit analysis tasks (§III-E and §III-F). To evaluate SCALECIRC, we compare its performance on under-constrainedness analysis to existing methods (§IV-B), as well as on two additional circuit analysis tasks to show its generalizability (§IV-C). In our comparison, we discuss both the improvements and limitations of SCALECIRC compared to baseline tools, along with case studies for better illustration. We also conduct an ablation study on SCALECIRC to examine the contribution of its components (§IV-D) and discuss its resource overhead (§IV-E).

In summary, our contributions are as follows:

1) We propose a novel framework called SCALECIRC for systematically managing the redundancy within the analysis of Circom code, largely enhancing the efficiency and effectiveness of Circom analysis tasks.
2) We develop novel constrainedness propagation methods for under-constrainedness analysis that leverage fine-grained semantic information from source code.
3) We demonstrate SCALECIRC's generalizability by showing its performance across multiple circuit analysis tasks.
4) On our benchmark of 691 real-world circuits (largest among existing works), SCALECIRC successfully analyzes 70 circuits that Picus failed on and 56 that ConsCS failed.

II. BACKGROUND

A. The Circom Language

A Zero-Knowledge Proof (ZKP) allows one party to prove to another that a statement is true without revealing any additional private information beyond the statement's validity. Circom has emerged as one of the most popular programming languages for specifying ZKP circuits. These circuits are compiled into R1CS (Rank-1 Constraint System) format, a system of finite field polynomial constraints that encode the relationships between circuit signals and specify computation correctness requirements for generating ZKPs. Consider the example Circom program implementing SHA-256 shown in Figure 1, demonstrating several key language concepts of Circom. It consists of multiple templates (e.g., `Sha256_2`), where each template is a parametric circuit blueprint that defines reusable circuit logic. Given concrete parameter values, templates can be instantiated within other templates (which we call *parent templates*) to create components, which are concrete instances of the template. For example, `Sha256_2` is a parent **template** circuit that instantiates a `Bits2Num` template (line 3), creating a **component** stored in the identifier `bits2num`. Another important distinction is between signals and variables [31]. **Signals** are immutable tokens containing field elements in $\mathbb{F}_p$ that are used to generate constraints. In our example circuit, line 12 defines an array input signal `hin`, and line 13 defines an array output signal `out`. Signals use special operators like <== and === for assignments that generate R1CS constraints. In contrast, **variables** (declared with `var`) are mutable and store intermediate calculations, as shown in line 14. When generating constraints, variables are replaced by their computational expressions. To transform these language constructs into ZKP circuits, the Circom compiler [32] translates the circuit source code into R1CS constraints and generates witness generators (in WebAssembly or C++) that compute values for all signals given the input signal assignments. The compiler also produces a symbol file that maps signals to R1CS variables. The compiled circuit and computed witness values can then be used with snarkjs [33] to generate and verify ZKPs.

B. Constrainedness of Arithmetic Circuits

One single R1CS constraint is defined as $\left(\sum_j a_j x_j\right) \cdot \left(\sum_j b_j x_j\right) - \sum_j c_j x_j \equiv 0 \pmod{p}$ [24], where a_j, b_j, c_j are coefficients in the finite field $\mathbb{F}_p$, and x_j represents witness variables. Each R1CS constraint is a degree-2 polynomial, as it involves the product of two linear terms minus a third linear term. A Circom circuit is compiled into a system of such polynomials that specify the intended computation. For example, a Circom statement $s_1 \ + \ s_2 \ === \ 3$ is translated to the R1CS constraint: $(s_2+s_1)\cdot 1-3 \equiv 0 \pmod{p}$. The witness is a satisfying solution to this system of equations, which contains the private information (e.g., private input signals) that the prover wants to prove knowledge of without revealing. An important property of Circom circuits is their constrainedness. A circuit is **properly constrained** if for any valid input assignment, its output signal are deterministically bound to a specific value [24]. Conversely, a circuit is **under-constrained** if for some valid input assignment, at least one output signal can take multiple valid values, meaning that the output signals are not properly constrained by the input signals. Any pair of valid witnesses where their input signals are identical but output signals differ constitutes a **counter-example** that shows a circuit to be under-constrained by definition. Under-constrained circuits represent a challenge in Circom development, as they typically indicate missing constraints that could lead to bugs in its code. In fact, under-constrained circuits have been identified as the most commonly found real-world programming issue in the Circom language [15].

III. Methodology

A. Overview

ScaleCirc is a novel framework that scales zero-knowledge circuit analysis by exploiting their template-based nature. Its generic workflow (§III-C) is as follows: **(1)** Parse input code into an Abstract Syntax Tree (AST), capturing hierarchical structure and syntactic relationships within the code; **(2)** Preprocess the AST to extract task-specific information (e.g., loop handling, assignment/range history, constrainedness) that guides subsequent transformations; **(3)** For each component, try retrieving cached analysis results, leveraging the template-based nature of circuits to avoid redundant computation; **(4)** Upon cache misses, recursively analyze the component and cache new results; **(5)** Propagate component results to the parent circuit through transformations that preserve essential properties while reducing problem size, helping achieve scalability; **(6)** Post-process to aggregate information into final analysis results and update the cache. A detailed example illustrating this workflow is in Section III-D4. We then present our implementation and inference rules for under-constrainedness analysis (§III-D), and demonstrate ScaleCirc's generalizability through additional circuit analysis tasks (§III-E and §III-F).

B. The DSL Syntax

Motivation. We present a concise domain-specific language (DSL) that captures the essential elements of the Circom

$$
\begin{aligned}
\mathit{Template} : \mathcal{T}(\vec{\tau}) &::= \mathcal{S}^{+} \\
\mathit{Function} : \mathcal{F}(\vec{\tau}) &::= \mathcal{S}^{+} \\
\mathit{Statement} : \mathcal{S} &::= \textbf{if } (\mathcal{E}) \textbf{ then } (\mathcal{S}^{+}) \textbf{ else } ((\mathcal{S}^{*}) \mid \\
&\qquad \textbf{while } (\mathcal{E}) \textbf{ do } (\mathcal{S}^{+}) \mid \\
&\qquad \textbf{for } (\mathcal{S}, \mathcal{E}, \mathcal{S}) \textbf{ do } (\mathcal{S}^{+}) \mid \\
&\qquad \mathcal{E} \odot \mathcal{E} \mid \textbf{return } (\mathcal{E}) \\
&\qquad \textbf{log } (\mathcal{E}^{*}) \mid \textbf{assert } (\mathcal{E}) \mid \\
&\qquad \textbf{new}(\tau) \mid \textbf{new}(\tau) \odot \mathcal{E} \\
\mathit{Expression} : \mathcal{E} &::= \tau \mid \kappa.\sigma \mid \alpha\langle\tau\rangle[\vec{\mathcal{E}}] \mid \mathcal{F}(\vec{\mathcal{E}}) \mid \\
&\qquad \mathcal{T}(\vec{\mathcal{E}}) \mid \mathcal{E} \oplus \mathcal{E} \mid \phi \\
\mathit{Token} : \tau &::= c \mid \sigma \mid \nu \mid \kappa \mid \alpha\langle\tau\rangle \\
\mathit{Signals} : \sigma &::= \sigma_{int} \mid \sigma_{in} \mid \sigma_{out} \\
\mathit{StmtOp} : \odot &::= \odot_{con} \mid \texttt{<--} \mid \texttt{=} \\
\mathit{ConstraintOp} : \odot_{con} &::= \texttt{<==} \mid \texttt{===} \\
\mathit{Operators} : \oplus &::= + \mid - \mid * \mid ** \mid > \mid == \mid \% \mid \&\& \mid \ldots
\end{aligned}
$$

Fig. 2: A Simplified DSL Syntax of Circom Programs.

language needed for our analysis.

1) The syntax: Our simplified grammar, shown in Figure 2, focuses on fundamental constructs while abstracting away some features (e.g., pragma and include statements) that do not impact our target problems. We use superscripts of $*$ to denote ≥ 0 occurrences, and $+$ to denote ≥ 1 occurrences.

Templates ($\mathcal{T}$) and *functions* ($\mathcal{F}$) are building blocks in Circom. Templates act as circuit generators that create components when instantiated with parameters, while functions provide auxiliary computational logic. They both contain sequences of *statements* ($\mathcal{S}^{+}$). In Figure 1, *Sha256_2*, *Sha256compression*, and *T2* (in lines 1, 10, and 22) are three example templates.

Statements ($\mathcal{S}$) include control flow constructs (conditionals, loops), signal and variable assignments, component instantiations, etc. For example, lines 2 through 8 in Figure 1 are all valid statements. Assignment operators ($\odot$) are categorized into ones that generate R1CS constraints ($\odot_{con}$: <==, ===) and ones (<--, =) that specify witness computation without generating constraints. For example, line 8 in Figure 1 is a constraint-generating statement. *Template instantiation* follows the syntax **new**(τ), where *tokens* (τ) can be one of the five types: *constants* (c), *signals* (σ), *variables* (ν), *components* (κ), or multi-dimensional *arrays* of tokens ($\alpha\langle\tau\rangle$). In Circom, each array must be filled with elements of the same type, which is why we mark arrays with the type indicator $\langle\tau\rangle$. Our recursive definition of arrays naturally supports multidimensional arrays. Template instantiation can also be coupled with an initializing expression $\textbf{new}(\tau) \odot \mathcal{E}$. *Expressions* ($\mathcal{E}$) represent computations that evaluate to values, and can be one of the following: tokens (τ), component signal access ($\kappa.\sigma$), array access ($\alpha\langle\tau\rangle[\vec{\mathcal{E}}]$) (which support multi-dimensional array access ($\vec{\mathcal{E}}$)), template or function initialization ($\mathcal{F}(\mathcal{E}^{*})$, $\mathcal{T}(\mathcal{E}^{*})$), and arithmetic operations between expressions ($\mathcal{E} \oplus \mathcal{E}$). Line 3 in Figure 1 is an example of template instantiation.

A *signal* is divided into 3 types: *intermediate* (σ_{int}), *input* (σ_{in}), and *output* (σ_{out}). Signal visibility follows strict rules:

only input and output signals of a component are accessible from outside via the dot notation (e.g., $\kappa.\sigma$), while intermediate signals remain private and accessible only within the template that defines them. In Figure 1, lines 11 to 12 is an example of input and output signal declaration.

In this DSL, we abstract away details such as anonymous components by unrolling them into equivalent regular component operations, and rightward assignment operators (--> and ==>) by equivalently representing them as their leftward counterparts (<--, <==). This has the benefit of focusing on the most important details while abstracting away others.

We modified the Circom parser [32] (written in Rust) to output structured AST in the JSON format, which is originally extracted based on a set of pre-defined Lalrpop grammar. The extracted AST contains detailed information such as location tracking for language constructs and symbol information (e.g., type, dimensions, declaration order) of our DSL. Syntax sugar (e.g., anonymous components, tuple expressions) is automatically handled by the Circom parser. The parsed AST serves as input to our subsequent analysis phases.

Algorithm 1 high level scheme of analyze_template

```
1: procedure GENERAL_ANALYZE($\mathcal{T}, \vec{\pi}$)
2:     $ast \leftarrow Parse(\mathcal{T})$                              ▷ ①
3:     $Preprocess(ast)$                                   ▷ ②
4:     for $\kappa \in ast.\vec{\kappa}$ do
5:         $(\mathcal{T}_\kappa, \vec{\pi}_\kappa) \leftarrow extract\_template\_info(\kappa)$
6:         if $(T_\kappa, \vec{\pi}_\kappa) \in dom(\mathbb{C})$ then
7:             $\mathcal{R}_\kappa \leftarrow \mathbb{C}(T_\kappa, \vec{\pi}_\kappa)$               ▷ ③
8:         else
9:             $\mathcal{R}_\kappa \leftarrow generall\_analyze(T_\kappa, \vec{\pi}_\kappa)$    ▷ ④
10:            $\mathbb{C}(\mathcal{T}_\kappa, \vec{\pi}_\kappa) \leftarrow \mathcal{R}_\kappa$
11:        end if
12:        $Propagate(\mathcal{R}_\kappa, \kappa)$                          ▷ ⑤
13:    end for
14:    $Postprocess(\mathcal{T}, \vec{\pi})$                               ▷ ⑥
15:    return $\mathbb{C}(\mathcal{T}, \vec{\pi})$
16: end procedure
```

C. The General Analysis Scaling Scheme

Motivation. We introduce a generic framework in Algorithm 1 that enhances the effectiveness and efficiency of circuit analysis tools through a systematic approach. The generic algorithm implements template-based recursive analysis with caching, which can be adapted to scale various circuit analysis problems.

1) The framework: The algorithm consists of six major steps (marked as ① through ⑥ in Algorithm 1), and three generic operations that are to be implemented for each specific circuit analysis task: `Preprocess`, `Propagate`, and `Postprocess`. The workflow proceeds as follows: ① (line 2) The algorithm takes as input a template $\mathcal{T}$ and its parameters $\vec{\pi}$ and parses them into an AST representation, following the DSL introduced in Section III-B and Figure 2. ② (line 3) The `Preprocess` operation analyzes the AST to extract preliminary information specific to the type of circuit analysis task being performed, such as signal constrainedness, access patterns, component locations, etc. Next, for each component κ, after extracting its template $\mathcal{T}\kappa$ and parameters $\vec{\pi}\kappa$ (line 5), the algorithm either: ③ (line 7) Retrieves cached results $\mathcal{R}_\kappa$ for κ if available (caching is implemented as partial function $\mathbb{C} : \mathcal{T} \rightarrow \mathcal{R}$, where $\mathcal{R}$ contains both analysis results and auxiliary information (e.g., counter-examples for under-constrained circuits, graph structures), enabling efficient reuse and combination of analysis results), or ④ (line 9) Recursively analyzes κ using Algorithm 1 and caches the result (line 10). ⑤ (line 12) After an analysis result for κ is obtained, the `Propagate` operation maps it back into the parent circuit, applying transformations to reduce the problem size while preserving essential properties for the target analysis. ⑥ (line 14) Finally, the `Postprocess` operation aggregates the accumulated information into the final analysis result and updates the cache accordingly. We transform circuits only for analysis, and preserve the original circuits for downstream processing, ensuring compatibility with other tools.

In summary, SCALECIRC requires implementation of three generic operations for each circuit analysis task: **(1)** `Preprocess`: extracts preliminary circuit information that could be helpful for later stages. **(2)** `Propagate`: integrates known component analysis results and applies necessary transformations, in order to reduce the problem size. **(3)** `Postprocess`: generates final analysis results and manages cache updates. Specific implementations of these generic operations for several popular circuit analysis tasks are presented below in Sections III-D, III-E, and III-F.

D. Under-constrainedness Detection

Motivation. We present a framework for scaling the analysis of under-constrained circuits. Under-constrained circuits represent a challenge as the most prevalent real-world programming issue [15], making their scalable analysis crucial for practical applications. Furthermore, we demonstrate the generalizability of our framework across additional circuit analysis tasks in Sections III-E and III-F.

1) Source-code based constrainedness propagation: We present $ConstProp(\mathcal{T})$, which is the initial stage of analysis that propagates and identifies signal constrainedness at the source code level. This approach represents a novel contribution compared to existing R1CS-based methods. Source-level analysis (e.g., on arrays/loops) enables efficient constrainedness propagation capturing program structure while avoiding R1CS redundancy. Constrainedness propagation throughout the circuit, i.e., $ConstProp(\mathcal{T})$, is a key component of our `Preprocess` generic operation implementation (detailed later in Section III-D2). Our approach uses inference rules (IRs) to propagate constrainedness information throughout the circuit. The analysis maintains five key data structures:

- *Template analysis result cache* ($\mathbb{C} : \mathcal{T} \rightarrow \mathcal{R}$): Maps templates ($\mathcal{T}$) to their analysis results ($\mathcal{R}$).
- *Constrainedness store* (Γ_c : Set($\mathcal{E}$)): Tracks expressions inferred to be constrained. It is initialized with input signals and template parameters, as input signals are

$$\frac{\sigma \odot_{con} \mathcal{E} \quad \mathcal{E} \in \Gamma_c}{\Gamma_c \leftarrow \Gamma_c \cup \{\sigma\}} \quad (1) \qquad \frac{\nu = \mathcal{E} \quad \mathcal{E} \in \Gamma_c}{\Gamma_c \leftarrow \Gamma_c \cup \{\nu\}} \quad (2) \qquad \frac{\mathcal{E}_1 \in \Gamma_c \quad \mathcal{E}_2 \in \Gamma_c}{\Gamma_c \leftarrow \Gamma_c \cup \{\mathcal{E}_1 \oplus \mathcal{E}_2\}} \quad (3) \qquad \frac{Set(\vec{\mathcal{E}}) \subseteq \Gamma_c}{\Gamma_c \leftarrow \Gamma_c \cup \{\mathcal{F}(\vec{\mathcal{E}})\}} \quad (4)$$

$$\frac{\alpha\langle\tau\rangle[\vec{\mathcal{E}}] \in \Gamma_c}{\Gamma_c \leftarrow \Gamma_c \cup \{\alpha\langle\tau\rangle[\vec{\mathcal{E}} \| *]\}} \quad (5) \qquad \frac{\forall i \in [0, |\alpha\langle\tau\rangle[\vec{\mathcal{E}}]|),\ \alpha\langle\tau\rangle[\vec{\mathcal{E}}][i] \in \Gamma_c}{\Gamma_c \leftarrow \Gamma_c \cup \{\alpha\langle\tau\rangle[\vec{\mathcal{E}}]\}} \quad (6)$$

$$\frac{\mathbb{C}(\kappa.\mathcal{T}, \kappa.\vec{\pi}) = \textbf{safe} \quad \kappa.\sigma_{in} \subseteq \Gamma_c}{\forall \sigma \in \kappa.\sigma_{out},\ \Gamma_c \leftarrow \Gamma_c \cup \{\sigma\}} \quad (7) \qquad \frac{\textbf{if } (\mathcal{E}) \textbf{ then } (\mathcal{S}_1^+) \textbf{ else } (\mathcal{S}_2^+) \quad \mathcal{S}_1^+ \Vdash \tau \quad \mathcal{S}_2^+ \Vdash \tau}{\Gamma_c \leftarrow \Gamma_c \cup \{\tau\}} \quad (8)$$

$$\frac{\textbf{for } i \in [l, u] \textbf{ do } \mathcal{S}^+ \quad \sigma[\vec{\mathcal{E}}_{i=l}] \in \Gamma_c \quad (\mathcal{S}^+, \sigma[\vec{\mathcal{E}}_{i=k}] \in \Gamma_c) \Vdash \sigma[\vec{\mathcal{E}}_{i=k+1}]}{\Gamma_c \leftarrow \Gamma_c \cup \{\sigma[\vec{\mathcal{E}}_{i=k}] \mid k \in [l, u]\}} \quad (9)$$

$$\frac{\nu = \mathcal{E}}{\begin{array}{c}\Gamma_a(\nu) \leftarrow \Gamma_a(\nu) \cup \{\mathcal{E}\} \\ \Gamma_r(\nu) \leftarrow \Gamma_r(\nu) \cup \{[\top]\}\end{array}} \quad (10) \qquad \frac{\sigma \odot_{con} \mathcal{E}}{\begin{array}{c}\Gamma_a(\sigma) \leftarrow \Gamma_a(\sigma) \cup \{\mathcal{E}\} \\ \Gamma_r(\sigma) \leftarrow \Gamma_r(\sigma) \cup \{[\top]\}\end{array}} \quad (11) \qquad \frac{\alpha\langle\tau_1\rangle[\vec{\mathcal{E}_1}] = \mathcal{E} \quad \odot \neq \texttt{<--}}{\begin{array}{c}\Gamma_a(\alpha\langle\tau_1\rangle) \leftarrow \Gamma_a(\alpha\langle\tau_1\rangle) \cup \{\mathcal{E}\} \\ \Gamma_r(\alpha\langle\tau_1\rangle) \leftarrow \Gamma_r(\alpha\langle\tau_1\rangle) \cup \{[\vec{\mathcal{E}_1}]\}\end{array}} \quad (12)$$

$$\frac{\forall \mathcal{E} \in \Gamma_a(\tau),\ \mathcal{E} \in \Gamma_c \qquad \bigcup_{[\vec{r}] \in \Gamma_r(\tau)} [\vec{r}] = \bigcup_{\vec{x} \in \mathbb{Z}^{\dim(\tau)} | 0 \le x_i < \text{size}_i(\tau)} [\vec{x}]}{\Gamma_c \leftarrow \Gamma_c \cup \{\tau\}} \quad (13) \qquad \frac{\forall \sigma \in \sigma_{out}, \sigma \in \Gamma_c}{\mathbb{C}(\mathcal{T}, \vec{\pi}) \leftarrow \textbf{safe}} \quad (14)$$

Fig. 3: Inference Rules (IRs) for Constrainedness Propagation $ConstProp(\mathcal{T})$

inherently constrained (being uniquely self-determined) and parameters are initialized as constants.

- *Assignment history store* ($\Gamma_a : \tau \rightarrow \text{Set}(\tau)$): Records all historical assignments to each variable.
- *Range history store* ($\Gamma_r : \tau \rightarrow [\mathcal{E}^{\dim(\tau)}]$): Tracks multi-dimensional access history of variables.
- *Location store* ($\Gamma_{loc} : \kappa \rightarrow (s, e)$): Maps components to their source code locations, where (s, e) denotes the start and end character indices in the source code.

Figure 3 presents our inference rules (IRs) for Constrainedness Propagation. These are applied as we traverse the circuit's AST. The IRs provide a formal and accessible description of our constrainedness propagation system.

IR 1 and IR 2 handle basic constrainedness propagation through constraints ($\odot_{con}$) for signals, and assignments ($=$) for variables. When an expression $\mathcal{E}$ is constrained ($\mathcal{E} \in \Gamma_c$), its assigned variable or signal is therefore constrained and added to Γ_c. They also apply to signals or variables that are array elements. IR 3 extends to arithmetic and logical operations, where constrained operands produce constrained results. Similarly, IR 4 handles function calls, where constrained inputs deterministically yield constrained outputs. Note that IR 1 does not consider <-- assignments as they only specify witness generation without affecting the constraint system.

IR 5 and IR 6 consider multi-dimensional arrays. When an array access $\alpha\langle\tau\rangle[\vec{\mathcal{E}}]$ (e.g., for a[0][1], $\vec{\mathcal{E}}$ is represented as the <0,1> vector) is constrained, all nested accesses $\alpha\langle\tau\rangle[\vec{\mathcal{E}} \| *]$ inherit this property (IR 5). Conversely, when all elements at a dimension $\alpha\langle\tau\rangle[\vec{\mathcal{E}}][i]$ are constrained, the entire dimension $\alpha\langle\tau\rangle[\vec{\mathcal{E}}]$ becomes constrained (IR 6), where $|\alpha\langle\tau\rangle[\vec{\mathcal{E}}]|$ denotes the size of the immediate nested dimension.

IR 7 handles component-level propagation. When a constrained component κ and all its inputs are constrained, then all its outputs are constrained by definition. IR 8 manages if-else statements. A token τ is constrained only if it's constrained in all branches, regardless of the execution path. This applies to complete if-else structures (with nested conditions handled recursively), but not to those without else branches (thus the $+$ notation in $\mathcal{S}_2^+$). By convention [25], we use the notation $\mathcal{S}_1^+ \Vdash \tau$ to indicate that analyzing $\mathcal{S}_1^+$ renders τ constrained.

IR 9 handles loop-based constrainedness through induction. For a loop iterating from l to u, the base case is when a signal $\sigma[\vec{\mathcal{E}}_{i=l}]$ is constrained at the initial iteration. The induction step is that if signal $\sigma[\vec{\mathcal{E}}_{i=k}]$ is constrained at iteration k, executing the loop body $\mathcal{S}^+$ renders the signal to be constrained at the next iteration $\sigma[\vec{\mathcal{E}}_{i=k+1}]$. Then by induction, $\sigma[\vec{\mathcal{E}_i}]$ is constrained throughout $i \in [l, u]$.

IR 10, 11, and 12 maintain assignment and access history. When an expression $\mathcal{E}$ is assigned to a variable ν (IR 10), a signal σ (IR 11), or an array access $\alpha\langle\tau_1\rangle[\vec{\mathcal{E}_1}]$ (IR 12), it is added to their respective assignment history stores (Γ_a). For signals and variables, $[\top]$ is added to their range store (Γ_r) to indicate complete range access, while for array accesses, its specific access is recorded in Γ_r.

IR 13 combines range (Γ_r) and assignment (Γ_a) history to infer constrainedness. A token τ is constrained when all its assignments are constrained ($\forall \mathcal{E} \in \Gamma_a(\tau),\ \mathcal{E} \in \Gamma_c$) and its access history (i.e., union of all recorded accesses $\bigcup_{[\vec{r}] \in \Gamma_r(\tau)} [\vec{r}]$ covers all dimensions $\bigcup_{\vec{x} \in \mathbb{Z}^{\dim(\tau)} | 0 \le x_i < \text{size}_i(\tau)} [\vec{x}]$). Lastly, IR 14 marks the parent template as constrained when all its output signals are constrained.

2) Implementations: Equations 15 through 18 show our implementation of the generic operations from Algorithm 1 of Section III-C. Equation 15 implements `Preprocess` through two actions: first applying constrainedness propagation `ConstProp`($\mathcal{T}$) (as introduced in Section III-D1), and then updating the location store $\Gamma_{loc}(\kappa)$ by recording locations of all expressions containing component κ (i.e., locations of $\mathcal{T}.\mathcal{E}_\kappa$). This step propagates constrainedness information and enables precise component replacement during later propagation.

Equation 16 defines propagation of a **safe** constrained com-

$$\frac{\texttt{Preprocess}(\mathcal{T})}{\texttt{ConstProp}(\mathcal{T}) \quad \forall \mathcal{E} \in \mathcal{T}.\mathcal{E}_{\kappa}, \Gamma_{loc}(\kappa) \leftarrow \Gamma_{loc}(\kappa) \cup \{loc(\mathcal{E})\}} \tag{15}$$

$$\frac{\texttt{Propagate}(\textbf{safe}, \kappa) \quad \forall \sigma \in \kappa.\sigma_{in} : \sigma \in \Gamma_c}{\texttt{AddSig()} \quad \texttt{DelComp()} \quad \texttt{RepSig()}} \tag{16}$$

$$\frac{\begin{array}{c}\texttt{Propagate}(\textbf{unsafe}, \kappa) \\ \texttt{GetWit}(\mathcal{T}, \kappa.wit_1)_{out} \neq \texttt{GetWit}(\mathcal{T}, \kappa.wit_2)_{out}\end{array}}{\mathbb{C}(\mathcal{T}, \vec{\pi}) \leftarrow \textbf{unsafe}} \tag{17}$$

$$\frac{\texttt{Postprocess}(\mathcal{T})}{\mathbb{C}(\mathcal{T}, \vec{\pi}) \leftarrow \texttt{ArtifactInv}(\mathcal{T})} \tag{18}$$

ponent κ with constrained input signals ($\forall \sigma \in \kappa.\sigma_{in} : \sigma \in \Gamma_c$). We transform the circuit while preserving constrainedness by: **(1)** `AddSig()`: Transforms κ's input signals $\kappa.\sigma_{in}$ into new intermediate signals $\textbf{new}_{\sigma_{int}}(\sigma)$ (i.e., $\mathcal{T}.\mathcal{S} \leftarrow \{\textbf{new}_{\sigma_{int}}(\sigma) \mid \sigma \in \kappa.\sigma_{in}\} \| \mathcal{T}.\mathcal{S}$, where $\mathcal{T}.\mathcal{S}$ denotes the sequence of all statements in $\mathcal{T}$), and output signals $\kappa.\sigma_{out}$ into new input signals (i.e., $\mathcal{T}.\mathcal{S} \leftarrow \{\textbf{new}_{\sigma_{in}}(\sigma) \mid \sigma \in \kappa.\sigma_{out}\} \| \mathcal{T}.\mathcal{S}$) at the beginning of the parent template. **(2)** `DelComp()`: Removes κ's declaration from the parent template ($\mathcal{T}.\mathcal{S} \leftarrow \mathcal{T}.\mathcal{S} \backslash (\textbf{new}(\kappa))$). **(3)** `RepSig()`: Uses $\Gamma_{loc}(\kappa)$ to replace κ's signal occurrences with their new counterparts ($\textbf{new}_{\sigma_{int}}$ and $\textbf{new}_{\sigma_{in}}$), preserving original circuit constraint wirings.

These operations preserve the circuit's signal structure and constrainedness property while eliminating κ, reducing the circuit size. However, if we find the modified circuit to be under-constrained, this result isn't conclusive. This is because constrainedness is a universal ($\forall$) property over all witness assignments, while under-constrainedness is existential ($\exists$) as proved by a counter-example. Specifically, a found counter-example $w' \in (\mathbb{F}_p^{|\sigma_{in}|} \times \mathbb{F}_p^{|\kappa.\sigma_{out}|})$ from here might exist only in our expanded domain but not in the original restricted sub-domain $(\mathbb{F}_p^{|\sigma_{in}|} \times \mathbb{F}_p^{|\kappa.\sigma_{out}|})|_{\kappa.\sigma_{out}=f(\sigma_{in})}$. Due to this existential nature, we verify under-constrained results with an additional artifact call on the sub-domain to ensure correctness.

Equation 17 shows the propagation of **unsafe** under-constrained component κ. For unsafe components whose inputs connect directly to σ_{in} via constraint operators ($\odot_{con}$), we examine whether κ's counterexamples ($\kappa.wit_1$, $\kappa.wit_2$) can generate witnesses with identical inputs but different outputs in the parent template, as a proof of under-constrainedness. The witness generation process (`GetWit`) consists of three steps. First, assign to input signal with values from the given witness that directly connects to σ_{in}. Second, remove the unsafe component (using `AddSig()`, `DelComp()`, and `RepSig()` from Equation 16) and assign the counter-example outputs to the replaced output signal via constraints (i.e., via the $===$ operator). Third, generate witnesses for $\kappa.wit_1$ and $\kappa.wit_2$, respectively, through the WebAssembly witness calculator generated by the Circom compiler. We then compare the output parts of the two generated witness. Since identical inputs are supplied, different outputs in witness constitute a proof of under-constrainedness by definition. When same outputs occur, our inference aborts without further search. This approach tests whether a component's known counter-examples can induce

```
1  template ShaComp() {
2   signal input inp[256];
3   signal output o[48];
4   signal w[64];
5   var i;  var k;
6   component s[48];
7   for (i=0; i<48; i++) {
8   s[i] = Sigma();}
9   for (i=0; i<64; i++) {
10   w[i] <== inp[i];}
11  for (i=0; i<48; i++) {
12   for (k=0; k<32; k++){
13    s[i].i2[k]<==w[i];}
14   o[i] <== s[i].out;}}
```

```
15 template ShaComp() {
16  signal input inp[256];
17  signal output o[48];
18  signal w[64];
19  var i;  var k;
20  signal i2[48][32];
21  signal input out[48];
22  for (i=0; i<64; i++) {
23   w[i] <== inp[i];}
24  for (i=0; i<48; i++) {
25   for (k=0; k<32; k++){
26    i2[i][k] <== w[i];
27   o[i] <== out[i];}}
```

Fig. 4: Example: code before/after transformation on left/right.

under-constrainedness in the parent template. Equation 18 implements `Postprocess` as a direct artifact invocation (e.g., a call to Picus [25] or ConsCS [24]) on the modified circuit, with results cached for future queries.

3) Correctness: For *safe* circuits, the correctness follows from a key property: for a constrained component κ with constrained inputs $\kappa.\sigma_{in}$, its outputs $\kappa.\sigma_{out}$ are constrained. Converting $\kappa.\sigma_{out}$ to template inputs σ_{in} creates a *safe over-approximation*, since previously $\kappa.\sigma_{out}$ are uniquely determined by σ_{in} (i.e., $\sigma_{in} \Vdash \kappa.\sigma_{out}$), and after the transformation, the already constrained $\kappa.\sigma_{out}$ can take any free value from $\mathbb{F}_p^{|\kappa.\sigma_{out}|}$; therefore, if the modified circuit remains constrained for all possible witness under the expanded domain $(\mathbb{F}_p^{|\sigma_{in}|} \times \mathbb{F}_p^{|\kappa.\sigma_{out}|})$, it must be constrained in the original, more restricted sub-domain $(\mathbb{F}_p^{|\sigma_{in}|} \times \mathbb{F}_p^{|\kappa.\sigma_{out}|})|_{\kappa.\sigma_{out}=f(\sigma_{in})}$. For *unsafe* circuits, correctness is proven through ScaleCirc's counter-examples, demonstrating how identical inputs lead to different outputs, directly proving under-constrainedness by definition. An *unknown* result occurs when ScaleCirc cannot definitively determine whether an *entire* circuit is safe or unsafe after exhausting all inference rules and analysis methods. Following standard practice [24], [25], these cases are reported as *unsolved* in our metrics, preserving soundness by indicating analysis failure. For example, with 100 circuits in a benchmark where 75 are safe and 25 unknown, the solved rate would be 75%.

4) Example Transformation: Figure 4 presents an example to illustrate our transformation process. Initially, $\texttt{Preprocess}(\mathcal{T})$ and its constrainedness propagation $\texttt{ConstProp}(\mathcal{T})$ are executed using the IRs shown in Figure 3 to infer that the constrainedness of the *inp* input signal propagates to the *w* array, and consequently to the input signals *i2* of all *Sigma* components. Next, following Equation 16, given all input signals of all *Sigma* components and the *Sigma* components themselves are retrieved as constrained from the cache, we can transform the *Sigma* templates through these steps: (1) `DelComp()`: Remove the *Sigma* template array instantiation (lines 7-8) from the parent circuit; (2) `AddSig()`: Extract *Sigma*'s `i2` and `out` signals to the parent template (lines 20-21) to preserve the circuit's intermediate signal structure; (3) `RepSig()`: Replace operations on *Sigma*'s signals with their new counterparts (from lines 13-14 to lines 26-27) to preserve wiring connections of the circuit. These operations

$$\frac{\texttt{Preprocess}(\mathcal{T})}{\forall \mathcal{S} \in \mathcal{T}.\mathcal{S}_{\kappa \odot \mathcal{T}(\vec{\mathcal{E}})}, \Gamma_{loc}(\kappa) \leftarrow \Gamma_{loc}(\kappa) \cup \{loc(\mathcal{S})\}} \quad (19)$$

$$\frac{\texttt{Propagate}((\mathcal{G}_{\kappa-d}, \mathcal{G}_{\kappa-a}), \kappa)}{\begin{array}{c}\texttt{RepTemplate}()\\ (\mathcal{G}_{\text{aux}-d}, \mathcal{G}_{\text{aux}-a}) \leftarrow (\mathcal{G}_{\text{aux}-d} \cup \mathcal{G}_{\kappa-d}, \mathcal{G}_{\text{aux}-a} \cup \mathcal{G}_{\kappa-a})\end{array}} \quad (20)$$

$$\frac{\texttt{Postprocess}(\mathcal{T})}{\mathbb{C}(\mathcal{T}, \vec{\pi}) \leftarrow \texttt{ArtifactInv}(\mathcal{T}, \mathcal{G}_{\text{aux}-d}, \mathcal{G}_{\text{aux}-a})} \quad (21)$$

preserve the circuit's signal structure and constrainedness while reducing circuit size through *Sigma* elimination.

E. Generalization 1: Discrepancy and Lack of Constraint Detection

1) Overview: ZKAP [23] is an artifact that analyzes Circom code using a **Circuit Dependence Graph** (CDG). The CDG models signal relationships through: (1) *Data flow edges* (E_d): Directed edges representing computational dependencies from <-- and <== operators, capturing the intended witness computation flow; (2) *Constraint edges* (E_c): Undirected edges representing arithmetic constraints from === and <== operators, capturing the R1CS constraint relationships. Formally, a CDG is defined as $\mathcal{G} = (V, E_d, E_c)$ where V represents nodes for signals and constants. ZKAP excels at detecting the following constraint issues with high accuracy: (1) **Unconstrained Circuit Output (UCO)**: Circuit outputs (σ_{out}) that lack constraints. Detection involves checking if any σ_{out} is transitively constrained to constants or input signals via constraint edges; (2) **Unconstrained Sub-circuit Input (USCI)**: Component input signals ($\kappa.\sigma_{in}$) lacking constraints. Detection involves checking if any $\kappa.\sigma_{in}$ have no transitive constraint dependencies on external signals or constants; (3) **Dataflow-Constraint Discrepancy (DCD)**: Mismatches between witness computation and R1CS constraints, detected by comparing discrepancy between transitive closures of data flow and constraint edges. ZKAP operates in two steps. First, it converts Circom code to LLVM IR using the circom2llvm tool from Veridise [34]. Second, it analyzes the CDG using custom LLVM optimization passes written in C++, guided by vulnerability patterns specified in a custom Vulnerability Description Language (VDL). For each circuit being analyzed, ZKAP produces a report of signals that fall into any of these above categories.

2) Implementations: Our implementation is specified in Equations 19, 20, and 21, following the general framework presented in Section III-C. The `Preprocess` stage (Equation 19) identifies component instantiation locations by recording component declarations ($\mathcal{T}.\mathcal{S}_{\kappa \odot \mathcal{T}(\vec{\mathcal{E}})}$, i.e., retrieves component declarations from all statements) in the Circom code. These locations enable component replacement during propagation in subsequent stages. For the `Propagate` stage (Equation 20), its input consists of the component κ's data flow graph $\mathcal{G}_{\kappa-d}$ and constraint graph $\mathcal{G}_{\kappa-a}$. The propagation involves two steps. First, `RepTemplate` creates a simplified template $\mathcal{T}_{new}$ from κ that preserves only its input and output signal declarations (i.e., $\mathcal{T}_{new}.\mathcal{S} \leftarrow \kappa.\mathcal{T}.\mathcal{S}_{\textbf{new}(\sigma_{in})} \| \kappa.\mathcal{T}.\mathcal{S}_{\textbf{new}(\sigma_{out})}$). This simplified template has the benefit of maintaining κ's signal wirings in the parent template, while saving the efforts of rebuilding κ's graph and transitive closure of edges, since such information are already provided in the inputs $(\mathcal{G}_{\kappa-d}, \mathcal{G}_{\kappa-a})$. Then, the location store $\Gamma_{loc}(\kappa)$ recorded from `Preprocess` is looked up to replace the original component instantiation with $\mathcal{T}_{new}$, completing the replacement. Second, the graphs supplied in the input are merged into the auxiliary graphs $(\mathcal{G}_{\text{aux-d}}, \mathcal{G}_{\text{aux-a}})$, which records edges of all replaced components for later use. The subsequent `Postprocess` stage, shown in Equation 21, merges these auxiliary graphs into the main graph that ZKAP builds from the main template. Lastly, a direct artifact invocation to ZKAP over the main graph is applied, obtaining an analysis report and the template's overall graphs. These produced graphs are further cached for potential reuse. We modified ZKAP's C++ graph building code to support this functionality.

3) Correctness: Let $\mathcal{G}_{orig} = (V_{orig}, E_{d-orig}, E_{c-orig})$ and $\mathcal{G}_{new} = (V_{new}, E_{d-new}, E_{c-new})$ be graphs from original ZKAP and our framework, respectively; subscripts d and c denote data flow and constraint edges. Since our transformation preserves signal definitions, both approaches process same signal declarations to produce nodes, thus $V_{orig} = V_{new}$. For edges, original ZKAP merges freshly built edges from sub-circuits with the parent graph: $E_{orig} = (\bigcup_\kappa E_\kappa) \cup E_{parent}$. Our framework instead loads cached sub-circuit graphs (computing them if not cached) into an auxiliary graph $\mathcal{G}_{\text{aux}} = \bigcup_\kappa E_{\kappa-cache}$ and merges with the parent graph: $E_{new} = \mathcal{G}_{\text{aux}} \cup E_{parent}$. Since cached graphs are built from the same ZKAP process, then $E_{\kappa-cache} = E_\kappa$ for each component κ. Therefore: $E_{new} = \mathcal{G}_{\text{aux}} \cup E_{parent} = (\bigcup_\kappa E_{\kappa-cache}) \cup E_{parent} = (\bigcup_\kappa E_\kappa) \cup E_{parent} = E_{orig}$, so our approach produces identical edges with original ZKAP. Thus, $\mathcal{G}_{orig} = \mathcal{G}_{new}$.

F. Generalization 2: Unconstrained Detection

1) Overview: The unconstrained signal detector [22], [32] provided with the Circom compiler identifies signals that are declared but not constrained. The detector is implemented via the *--inspect* option in the compiler. Specifically, two scenarios are considered: (1) signals declared but not constrained within their template scope, and (2) array signals where some or all elements are not constrained. For each circuit being analyzed, the detector produces an *unconstrained signal report* indicating which signals or arrays of signals are unconstrained.

2) Implementations: We describe our implementation of the general framework from Section III-C. Our specifications are presented in Equations 22, 23, and 24. In particular, Equation 22 shows how `Preprocess` is implemented, which

$$\frac{\texttt{Preprocess}(\mathcal{T})}{\forall \mathcal{S} \in \mathcal{T}.\mathcal{S}_{\kappa \odot \mathcal{T}(\vec{\mathcal{E}})}, \Gamma_{loc}(\kappa) \leftarrow \Gamma_{loc}(\kappa) \cup \{loc(\mathcal{S})\}} \quad (22)$$

$$\frac{\texttt{Propagate}(\mathcal{R}_{\textbf{msgs}}, \kappa)}{\texttt{RepTemplate}() \quad \Gamma_{err} \leftarrow \Gamma_{err} \cup \mathcal{R}_{\textbf{msgs}}} \quad (23)$$

$$\frac{\texttt{Postprocess}(\mathcal{T})}{\mathbb{C}(\mathcal{T}, \vec{\pi}) \leftarrow \texttt{ArtifactInv}(\mathcal{T}) \cup \Gamma_{err}} \quad (24)$$

is identical to the implementation of ZKAP as introduced in Section III-E. For `Propagate`, shown in Equation 23, its input includes unconstrained signal reports $\mathcal{R}_{\textbf{msgs}}$ for component κ from the cache. The propagation involves two steps. First, `RepTemplate` creates a new simplified template $\mathcal{T}_{new}$ from κ, preserving only its input and output signal declarations, which is identical to the implementation for ZKAP (Section III-E). This has the benefit of perserving κ's circuit wirings while saving all the efforts of rebuilding a vulnerability report for κ, largely saving analysis efforts. Second, the unconstrained signal reports for κ are added to an error message store Γ_{err}, where Γ_{err} stores analysis reports of all components and the parent template. Equation 24 shows how `Postprocess` is implemented. It performs an artifact invocation on the modified template and combines its *unconstrained signal report* with the previously accumulated Γ_{err}, making the final report account for all components. This final report is added to the cache for potential later reuse.

3) Correctness: Let $\mathcal{R}_{orig}$ and $\mathcal{R}_{new}$ be unconstrained signal reports from original analysis and our framework, respectively. Original analysis merges each component's report with parent template's report: $\mathcal{R}_{orig} = (\bigcup_{\kappa} \mathcal{R}_{\kappa}) \cup \mathcal{R}_{parent}$. Our framework instead accumulates cached component reports (computing them if not cached) into $\Gamma_{err} = \bigcup_{\kappa} \mathcal{R}_{\kappa-cache}$ and merges with parent report: $\mathcal{R}_{new} = \Gamma_{err} \cup \mathcal{R}_{parent}$. Since cached reports used the same analysis process, $\mathcal{R}_{\kappa-cache} = \mathcal{R}_{\kappa}$ for each component κ. Therefore, $\Gamma_{err} = \bigcup_{\kappa} \mathcal{R}_{\kappa-cache} = \bigcup_{\kappa} \mathcal{R}_{\kappa}$, implying $\mathcal{R}_{orig} = \mathcal{R}_{new}$.

IV. Evaluation

We present a comprehensive evaluation addressing the following research questions in Sections IV-B through IV-E: (1) **RQ1:** Is ScaleCirc Able to Outperform Existing Works and Analyze Circuits that Were Previously Unsolvable? (2) **RQ2:** Is ScaleCirc Able to Generalize to Additional Circuit Analysis Tasks? (3) **RQ3:** How do Propagation and Deduplication Components Contribute to ScaleCirc's Overall Performance? (4) **RQ4:** What is the Storage Overhead of ScaleCirc?

A. Experiment Setup

We constructed a dataset of 691 circuits, with an average of 225,327 R1CS constraints per circuit. Our dataset construction process is as follows: (1) First, we adopt the widely used `circomlib` (including `utils` and `core`) benchmark built by Picus [25], containing 183 fundamental circuits from the standard Circom library. It has been extensively tested in prior work as a standard benchmark. (2) Second, we collected 508 circuits from 19 active Circom projects on GitHub. These were selected from the 100 most recently updated repositories (as of December 24, 2024) containing valid *.circom* files and having at least one star, ensuring code quality and community interest. Combined with the classical `circomlib` benchmark, our dataset totals 691 circuits, which is the largest among existing works. The dataset spans diverse zero-knowledge proof applications such as privacy systems, core cryptographic primitives, blockchain infrastructure and verification systems, etc. This diversity enables comprehensive evaluation of ScaleCirc across varied real-world scenarios. For simplicity, we use abbreviated benchmark names, and group benchmarks with ≤ 5 circuits into an "Other" category. Following existing categorization standards [24], [25], [35], our benchmark contains 307/128/101/155 circuits across Small(<100)/Medium(100–1000)/Large(1000-10000)/Very Large(>10000) constraint ranges. Our code is available at https://github.com/jinan789/ScaleCirc. A uniform 600-second timeout is adopted for all experiments. Our experiments are conducted on an Intel Xeon Gold 6226R CPU server with 503 GB RAM, running Ubuntu 18.04.6 LTS.

B. RQ1: Is ScaleCirc Able to Outperform Existing Works and Analyze Circuits that Were Previously Unsolvable?

1) Motivation and Approach: We evaluate ScaleCirc's capability to scale up under-constrained circuit analysis over two existing works Picus [25] and Conscs [24], using two metrics: (1) *solved*: the percentage of successfully verified circuits; (2) *time*: the time spent on analyzing each circuit. The experiment uses the setup outlined in Section IV-A. Table I presents the results, divided into two main sections comparing the baseline tools against their ScaleCirc counterparts. For each tool (columns) and benchmark (rows), we report both metrics and improvement percentages (highlighted in shaded columns) of ScaleCirc, calculated as $\frac{scaled-base}{base} \times 100\%$.

2) Improvements: ScaleCirc demonstrates noticeable improvements in both metrics across both baseline tools, suggesting it is able to scale the analysis of existing works and verify circuits that are previously unverifiable.

For Picus, solving time is improved by up to 99.7% and solved rates by up to 333.3%. ConsCS shows similar improvements, with time improvements up to 99.6% and solved rate increases up to 44.44%. Specifically, ScaleCirc successfully verified the correctness of 56 Circom circuits (averaging 1,474,524 constraints) that Conscs failed on, and 70 circuits (averaging 684,493 constraints) that Picus failed on. Such sizes significantly exceed the overall dataset's average of 225,327 constraints per circuit, demonstrating ScaleCirc's capability in handling large circuits, which posed challenges for existing works. For example, Picus [25] reports a drop of solved rate from 84% in smaller benchmarks (<100 constraints) to 20% in larger ones ($\geq$1000 constraints) in `circomlib`, while for our work this number is from 98% to 72%. Moreover, we newly solved 5/23/5/37 circuits that Picus failed on (17/12/7/20 for ConsCS) in the Small/Medium/Large/Very Large circuit categories. (as defined in Section IV-A). Most improvements occur in larger circuits, despite small circuits dominating the benchmark (44.4%). This shows that our approach effectively makes contributions to the domain of verifying larger circuits. Such enhanced performance stems from ScaleCirc's novel source-code based analysis and systematic deduplication strategies as introduced in Section III-D, which efficiently handle the complex signal wirings and component interactions typical in large circuits. Although we already matched the maximum timeout of prior works (ConsCS used 30s and Picus

TABLE I: Results of applying ScaleCirc to Picus and ConsCS.

	Picus		ScaleCirc		Improvement		ConsCS		ScaleCirc		Improvement	
Benchmark	**solved**	**time**	**solved**	**time**	**solved**	**time**	**solved**	**time**	**solved**	**time**	**solved**	**time**
aes	68.97%	188.35s	89.66%	62.73s	+30.0%	+66.7%	75.86%	113.81s	89.66%	62.26s	+18.2%	+45.3%
n-body	63.16%	149.70s	71.05%	120.73s	+12.5%	+19.4%	64.86%	122.24s	78.38%	54.55s	+20.8%	+55.4%
c-plonk	89.66%	99.27s	100.00%	0.26s	+11.5%	+99.7%	93.10%	7.61s	100.00%	0.03s	+7.4%	+99.6%
circom-dl	49.03%	303.23s	52.26%	271.39s	+6.6%	+10.5%	50.31%	186.86s	54.09%	235.29s	+7.5%	-25.9%
playground	66.67%	233.59s	100.00%	1.82s	+50.0%	+99.2%	100.00%	10.31s	100.00%	0.42s	+0.0%	+95.9%
circomlib	63.93%	162.00s	70.49%	122.71s	+10.3%	+24.3%	79.78%	37.22s	90.16%	19.49s	+13.0%	+47.6%
e-zkvm	38.46%	377.55s	38.46%	341.29s	+0.0%	+9.6%	42.31%	253.44s	42.31%	228.95s	+0.0%	+9.7%
maci	11.11%	537.99s	48.15%	318.66s	+333.3%	+40.8%	33.33%	392.24s	48.15%	268.89s	+44.4%	+31.4%
parser	44.44%	346.25s	71.43%	128.31s	+60.7%	+62.9%	60.32%	210.73s	69.84%	127.07s	+15.8%	+39.7%
tav	80.00%	128.94s	93.33%	43.75s	+16.7%	+66.1%	80.00%	136.18s	86.67%	80.08s	+8.3%	+41.2%
wrap	71.43%	180.39s	71.43%	90.53s	+0.0%	+49.8%	71.43%	55.93s	85.71%	7.12s	+20.0%	+87.3%
zk-regex	9.30%	544.86s	9.30%	544.19s	+0.0%	+0.1%	6.98%	285.49s	9.30%	297.34s	+33.3%	-4.1%
zk-sym	62.50%	211.60s	66.67%	182.52s	+6.7%	+13.7%	75.00%	134.10s	75.00%	125.08s	+0.0%	+6.7%
Other	57.14%	265.71s	78.57%	129.49s	+37.5%	+51.3%	64.29%	143.30s	78.57%	128.70s	+22.2%	+10.2%

used 600s), to further investigate whether our 600s timeout limits baseline tools' performance, we additionally tried to extend baseline timeouts to 6000s (10× longer) for circuits that only ScaleCirc could verify. Even with this extension, ConsCS and Picus still failed to verify 94.64% and 84.29% of these circuits respectively, demonstrating ScaleCirc's effectiveness in verifying previously unsolvable circuits across timeout settings.

We present a case study to better illustrate ScaleCirc's improvements. The *Pedersen(254)* hash circuit (6,152 constraints) from `circomlib` embeds 64 *BabyAdd* sub-circuits (6 constraints each) through complex recursive paths across multiple loops. Existing approaches can only verify the small individual 6-constraint *BabyAdd* circuits with an SMT call, and fails when all 64 of them are recursively embedded, due to the high level of total complexity in the resulting SMT query. In contrast, ScaleCirc successfully verifies the entire large circuit by effectively propagating constrainedness information of individual *BabyAdd* circuits across recursive layers at the fine-grained level. Furthermore, both the *sha256_2* (204,462 constraints) and *Sha256compression* circuits (202,784 constraints) shown in Figure 1 are additional large circuit examples that only ScaleCirc could verify, while existing works failed on, demonstrating ScaleCirc's successful analysis of large and complex circuits that were previously unsolvable.

3) Limitations: ScaleCirc underperforms in primarily two unusual causes: (1) benchmarks containing multiple circuits dependent on certain unsolvable common sub-circuit, and (2) circuits with no sub-circuits where deduplication doesn't apply. The latter is particularly unusual given Circom's reuse-oriented design philosophy, where even simple operations (e.g., equality checking) are typically modularized as reusable circuits. During removal of a component κ, we preserve end-to-end input-output signal constrainedness properties of κ via signal transformation, and discard its internal constraints. However, in rare cases, these intermediate constraints could provide useful context for full analysis. For instance, `tav`'s *qwords_to_bytes* template is constrained, but removing multiple *Num2Bits* components eliminates the binary bit conversion constraints helpful for full template analysis, yielding an *unknown* result (this is rare, which was the only case in the benchmark). Section III-D3 provides a detailed discussion on *unknown* results.

4) Bug-finding capabilities: ScaleCirc reported 3 bugs that baseline tools missed in our experiments (although these have been reported elsewhere [35], [36]). As a case study, consider the *Window4@pedersen.circom* circuit from Circomlib [37], which contains a buggy *component dbl2 = MontgomeryDouble()* sub-circuit. ScaleCirc assigns *dbl2.in[2]*'s counter-example inputs to parent template's witness, replaces the sub-circuit with its signals, and constrains its replaced signal to the counter-example outputs via `===` operators (see details in Section III-D), thereby identifying that *dbl2*'s counter-example induces its parent template to be under-constrained. This demonstrates how ScaleCirc leverages results from smaller sub-circuits to analyze larger ones, aligning with our core scaling methodology. While ScaleCirc found some known bugs, its primary focus is verifying circuit correctness and scaling analysis. In contrast, zkFuzz [35] is a recent tool specializing in bug detection through sophisticated fuzzing techniques, capturing both intermediate computations and program abort bugs. On our benchmark, zkFuzz identified 93 bugs (29 non-deterministic, 64 computation abort), averaging 152.44 seconds per bug circuit. In comparison, ScaleCirc identified 24 non-deterministic bugs, averaging 10.59 seconds per bug circuit. zkFuzz found 7 non-deterministic bugs that ScaleCirc missed, while ScaleCirc found 2 that zkFuzz missed. Overall, zkFuzz identifies more bugs, but ScaleCirc is more efficient. zkFuzz's strong bug detection capabilities are due to its sophisticated fuzzing techniques and ability to detect computation abort bugs, a novel category not covered by other works. However, unlike ScaleCirc, it cannot verify circuit correctness. At the other end of the bug-finding spectrum, Ecne [38] is an earlier tool focusing solely on verification without bug-finding capabilities. Our comparison shows ScaleCirc verified 66 circuits (taking 320.93s) that Ecne failed to verify (taking 10,060.17s, 30.35× slower). In summary, ScaleCirc's primary strength lies in verifying circuit correctness and scaling analysis speed to provide stronger correctness guarantees.

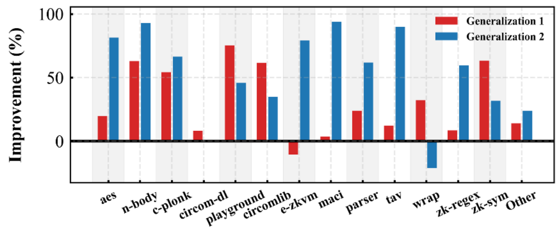

Fig. 5: Performance on generalization implementations.

Answer to RQ1: On our diverse benchmark, which is the largest among existing works, SCALECIRC largely enhances the solving time and solved rate compared to existing works, and is able to verify 70 circuits that Picus failed on, and 56 that ConsCS failed on.

C. RQ2: Is SCALECIRC Able to Generalize to Additional Circuit Analysis Tasks?

1) Motivation and Approach: We evaluate SCALECIRC's generalization capabilities beyond under-constrained analysis by examining its performance on two additional circuit analysis tasks introduced in Section III-E (ZKAP, detecting lack of constraint and discrepancy issues) and III-F (the Circom compiler, detecting un-constrained signal issues). This experiment uses the same benchmark as in RQ1 (Section IV-B).

2) Improvements: We measure the performance improvement compared to baseline tools when applying SCALECIRC, with results shown in Figure 5. The figure uses red bars (Generalization 1) and blue bars (Generalization 2) to show solving time improvements across different benchmarks. The bolded line at $y = 0$ indicates the threshold of zero performance improvement, bars above it indicate performance improvement and below is degradation. The results demonstrate that SCALECIRC introduces mostly positive performance improvements (above the $y = 0$ line) across both analysis tasks. For Generalization 2, we observe noticeable improvements, with several benchmarks achieving over 80% speedup. Notable benchmarks include `maci` with 94.09% improvement and `n-body` with 92.90%. Generalization 1 shows similar results, with `playground` achieving 75.35% improvement and `zk-sym` showing 63.22%. Overall, the figure shows that SCALECIRC can be successfully generalized to scale up multiple circuit analysis tasks.

3) Limitations: However, there are also cases where SCALECIRC resulted in performance degradation: (1) The `e-zkvm` benchmark shows negative improvement (-10.53%) in Generalization 1. This is due to the special structure of the *DigitReader* circuit in `e-zkvm`, which introduces large overhead due to merging 16 large *ArgMax* graphs into the main graph of *DigitReader*, each containing 16,254 edges. Overall, a total of 260,064 edges are inserted into the main graph from the cache, resulting in significant graph processing and cache management overhead in SCALRCIRC. The baseline tool took 81.95s on this circuit, while SCALRCIRC took 203.29s. (2) The `wrap` benchmark shows -21.23% degradation in Generalization 2. This is due to the *PrivacyToken* circuit, where the baseline

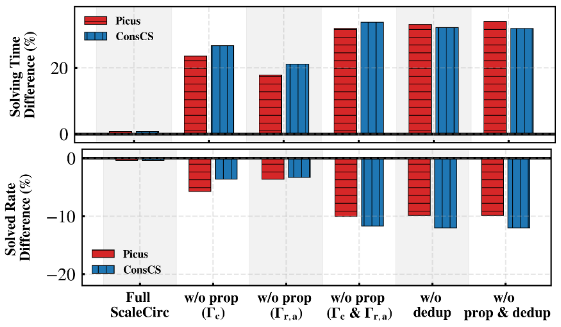

Fig. 6: Performance difference in solving time and solved rate.

took 1.77s, while SCALRCIRC took 2.69s. This is because it involves excessive recursive analysis over 13 recursive circuits such as *MultiplyPolynomials*, *UnpackArray*, *VerifyEncrypt*, etc. Managing these recursive calls introduces overhead such as AST parsing, source code analysis, cache management, etc. Despite these two cases, SCALRCIRC shows positive improvements across most benchmarks.

Answer to RQ2: SCALECIRC successfully generalizes to additional circuit analysis tasks, achieving up to 94% improvement in Generalization 2 and 75% in Generalization 1.

D. RQ3: How do Propagation and Deduplication Components Contribute to SCALECIRC's Overall Performance?

1) Motivation and Approach: SCALECIRC contains two major modules: **(1)** *Deduplication*: implemented through caching ($\mathbb{C}$) and cache-based analysis result reuse (IR 7). **(2)** *Propagation*: contains two parts, including direct propagation (IR 1-6, 8-9) on only constrainedness store (Γ_c), and propagation aided by additional paired range and assignment information (Γ_r, Γ_a) (IR 10-13). We conduct an ablation study isolating deduplication and propagation with these 6 SCALECIRC variants: **(1)** *Full* SCALECIRC: all components enabled, **(2)** *w/o prop* (Γ_c): propagation (basic constrainedness store) removed, **(3)** *w/o prop* ($\Gamma_{r,a}$): propagation (paired range and assignment store) removed, **(4)** *w/o prop* (Γ_c *&* $\Gamma_{r,a}$): entire propagation phase removed, **(5)** *w/o dedup*: deduplication removed, **(6)** *w/o prop & dedup*: both propagation and deduplication removed.

2) Results: Figure 6 shows the average performance difference relative to full SCALECIRC, adopting the same metric as the *improvement* column of Table I from RQ1. Higher bars in solving time (slower analysis) and lower bars in solved rate (lower success rate) indicate greater performance degradation. The full version achieves best performance in both metrics. Disabling either Γ_c or $\Gamma_{r,a}$ causes moderate degradation (medium-height bars), with Γ_c removal showing greater impact. This shows both propagation methods contribute meaningfully, with basic propagation on Γ_c carrying slightly more weight. Their complementary nature allows partial functionality when either is disabled. Removing full propagation, deduplication, or both causes noticeable degradation. Without both, SCALECIRC reduces to baseline performance, running only the

TABLE II: Average cache sizes by implementation type.

Type	ConsCS	Picus	G-1	G-2
Size (KB)	0.28	0.23	7.72	0.69

final *postprocess* step (Line 14, Algorithm 1). The significant performance drop when either component is removed is due to their sequential dependency: as shown in Equations 16 and 7, deduplication requires constrainedness information from propagation to determine feasibility; without propagation determining component input signals' constrainedness, deduplication cannot be performed due to unsatisfied antecedent predicate. Moreover, removing deduplication by itself prevents leveraging sub-circuit analysis results, thereby eliminating our core scaling capabilities and showing no improvement. These patterns remain consistent across both Picus and ConsCS over our large benchmark, demonstrating the generality of our results.

Answer to RQ3: Both propagation and deduplication are essential to SCALECIRC's performance. Propagation on basic Γ_c carries slightly more weight than on $\Gamma_{r,a}$.

E. RQ4: What is the Storage Overhead of SCALECIRC?

1) Motivation and Approach: We analyze the storage overhead introduced by SCALECIRC's caching mechanism across different circuit analysis tasks. Table II shows the average cache entry size per circuit for each implementation type, including Generalization-1 (G-1) and Generalization-2 (G-2) as introduced in § III-E and § III-F.

2) Results: Applying SCALECIRC over ConsCS and Picus demonstrates efficient storage usage, requiring only 0.28KB and 0.23KB per circuit, respectively. G-1 shows a higher storage demand (7.72KB) due to the need to store complete graph structures for each circuit's data flow graph $\mathcal{G}_{\kappa-d}$ and constraint graph $\mathcal{G}_{\kappa-a}$. G-2 has moderate storage requirements (0.69KB) as it only stores brief intermediate error messages. The total cache size for all 691 circuits in our benchmark across all 4 implementations sums to only 3.1MB, indicating a manageable overall storage overhead. Given this manageable storage requirements, analysis tools can easily distribute pre-computed results for commonly used circuits along with their artifacts, to largely enhance practical analysis effectiveness.

Answer to RQ4: SCALECIRC's caching mechanism introduces minimal storage overhead, making it practical for real-world deployment.

V. DISCUSSION

A. Failure Reasons and Limitations of SCALECIRC

SCALECIRC has these limitations: (1) complex cryptographic circuits (e.g., elliptic curve arithmetic like circomlib's *BabyPbk*) introduce challenges. Their analysis could reduce to proving basic cryptographic properties, making complete analysis theoretically challenging. We only handle them at the individual constraint level. (2) our analysis does not capture certain runtime-dependent information (e.g., values dependent on complex branching conditions or runtime context), potentially missing some analysis opportunities. (3) for under-constrainedness analysis, we modify signal wirings of circuits. While effective overall, it struggles with self-recursive circuits, as they assign values to new inputs that weren't in the original one. In such rare cases, we conservatively fall back to unmodified artifact invocations to ensure correctness. (4) our deduplication cannot be applied to Circomspect [39] since it only analyzes circuit at the topmost level, without recursively analyzing sub-circuits. (5) like other tools, SCALECIRC assumes toolchain correctness (e.g., compiler), leaving toolchain vulnerabilities out of scope.

VI. RELATED WORKS

A. Arithmetic Circuit Analysis

Several domain-specific languages have been developed for specifying arithmetic circuits, including Circom [40], Leo [41], Cairo [42], Zokrates [43]. With their growing importance in blockchain applications, various tools have been proposed to automatically analyze these circuits, with under-constrained circuits being the most prevalent issue [15]. Existing analysis approaches employ diverse methodologies such as SMT-based solutions (Picus [25], ConsCS [24]), graph-based analysis (ZKAP [23]), pattern matching (circomspect [39]), deduction rules (Ecne [38]). Some additional circuit analysis tools include Civer [26], CODA [44], Aleo [45], [46], etc.

B. Zero Knowledge Proof

Zero Knowledge Proofs (ZKPs) allow a prover to convince a verifier of a statement's truth without revealing private information. ZKPs have seen continuous recent development through works like Groth [47], Plonk [48], Libra [49], Pianist [50], [51]. Blockchain scaling solutions, particularly Zero Knowledge Virtual Machines (ZKVMs), represent a major real-world application of ZKPs. ZKEVMs like Polygon [3] and Scroll [4] focus on Ethereum compatibility for smart contract execution, while general-purpose implementations like RISC Zero [52], sp1 [53], and Jolt [54]–[56] enable proving execution of RISC-V CPU instructions. Besides developing ZKP systems, verification of ZK circuit-processing pipelines have also been proposed [57]–[59].

VII. CONCLUSION

In this paper, we presented SCALECIRC, a comprehensive framework for scaling the analysis of Circom circuits. SCALECIRC introduces systematic management of analysis redundancy, novel constrainedness propagation methods, and effective deduplication strategies. Our evaluation on 691 real-world circuits demonstrates significant improvements over existing tools from multiple perspectives.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their insightful comments and suggestions to help improve the quality of this paper. This work is supported by Hong Kong ITF Project (No. PRP/005/23FX) and Hong Kong RGC Projects (PolyU15224121, PolyU15231223).

References

[1] "Zero-knowledge rollups," https://ethereum.org/en/developers/docs/scaling/zk-rollups/, 2025.

[2] "zksync overview," https://docs.lite.zksync.io/userdocs/intro/, 2024.

[3] "Polygon overview," https://docs.polygon.technology/zkEVM/overview/, 2025.

[4] "Scroll overview," https://docs.scroll.io/en/getting-started/overview/, 2025.

[5] M. Rosenberg, J. White, C. Garman, and I. Miers, "zk-creds: Flexible anonymous credentials from zksnarks and existing identity infrastructure," in *2023 IEEE Symposium on Security and Privacy (SP)*, 2023, pp. 790–808.

[6] A. Ekbatanifard and G. Ekbatanifard, "Z-voting: A zero knowledge based confidential voting on blockchain," in *2024 8th International Conference on Smart Cities, Internet of Things and Applications (SCIoT)*, 2024, pp. 100–107.

[7] G. Misiakoulis, H. Niavis, S. Kundig, and K. Loupos, "Enhancing security and scalability in electronic voting through privacy-preserving cryptography and efficient data structures," in *2024 IEEE International Conference on Blockchain (Blockchain)*, 2024, pp. 631–636.

[8] J. Ernstberger, C. Zhang, L. Ciprian, P. Jovanovic, and S. Steinhorst, " Zero-Knowledge Location Privacy via Accurate Floating-Point SNARKs ," in *2025 IEEE Symposium on Security and Privacy (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2025, pp. 3199–3218. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SP61157.2025.00057

[9] K. Abbaszadeh, C. Pappas, J. Katz, and D. Papadopoulos, "Zero-knowledge proofs of training for deep neural networks," in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 4316–4330. [Online]. Available: https://doi.org/10.1145/3658644.3670316

[10] H. Sun, J. Li, and H. Zhang, "zkllm: Zero knowledge proofs for large language models," in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 4405–4419. [Online]. Available: https://doi.org/10.1145/3658644.3670334

[11] B. Feng, L. Qin, Z. Zhang, Y. Ding, and S. Chu, "ZEN: An optimizing compiler for verifiable, zero-knowledge neural network inferences," Cryptology ePrint Archive, Paper 2021/087, 2021. [Online]. Available: https://eprint.iacr.org/2021/087

[12] T. Liu, X. Xie, and Y. Zhang, "zkcnn: Zero knowledge proofs for convolutional neural network predictions and accuracy," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 2968–2985. [Online]. Available: https://doi.org/10.1145/3460120.3485379

[13] H. Sun, T. Bai, J. Li, and H. Zhang, "zkdl: Efficient zero-knowledge proofs of deep learning training," *IEEE Transactions on Information Forensics and Security*, vol. 20, pp. 914–927, 2025.

[14] B.-J. Chen, S. Waiwitlikhit, I. Stoica, and D. Kang, "Zkml: An optimizing system for ml inference in zero-knowledge proofs," in *Proceedings of the Nineteenth European Conference on Computer Systems*, ser. EuroSys '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 560–574. [Online]. Available: https://doi.org/10.1145/3627703.3650088

[15] 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*, ser. SEC '24. USA: USENIX Association, 2024.

[16] "Bigmod incorrectly omits range checks on the remainder," https://github.com/0xPARC/circom-ecdsa/pull/10, 2022.

[17] "Coreverifypubkeyg1 does not enforce lt checks on input." https://github.com/yi-sun/circom-pairing/pull/21/commits/c686f0011f8d18e0c11bd87e0a109e9478eb9e61, 2022.

[18] "Fix maci 1.0 processmessages circuit to prevent message censorship by the coordinator," https://github.com/privacy-scaling-explorations/maci/issues/320, 2021.

[19] "Disclosure of recent vulnerabilities," https://hackmd.io/@aztec-network/disclosure-of-recent-vulnerabilities, 2022.

[20] "Tornado.cash got hacked. by us." https://tornado-cash.medium.com/tornado-cash-got-hacked-by-us-b1e012a3c9a8, 2024.

[21] "Circomlib," https://github.com/iden3/circomlib, 2025.

[22] "Inspect option - circom 2 documentation," https://docs.circom.io/circom-language/code-quality/inspect/, 2025.

[23] 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 *33rd USENIX Security Symposium (USENIX Security 24)*. Philadelphia, PA: USENIX Association, Aug. 2024, pp. 1471–1487. [Online]. Available: https://www.usenix.org/conference/usenixsecurity24/presentation/wen

[24] J. Jiang, X. Peng, J. Chu, and X. Luo, " ConsCS: Effective and Efficient Verification of Circom Circuits ," in *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2025, pp. 737–737. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ICSE55347.2025.00200

[25] 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," *Proc. ACM Program. Lang.*, vol. 7, no. PLDI, jun 2023. [Online]. Available: https://doi.org/10.1145/3591282

[26] M. Isabel, C. Rodríguez-Núñez, and A. Rubio, "Scalable verification of zero-knowledge protocols," in *2024 IEEE Symposium on Security and Privacy (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, may 2024, pp. 136–136. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SP54263.2024.00133

[27] "Z3," https://github.com/Z3Prover/z3, 2025.

[28] A. Ozdemir, "Cvc5-ff," https://github.com/alex-ozdemir/CVC4/tree/ff, 2022.

[29] K. Zhao, Z. Li, W. Chen, X. Luo, T. Chen, G. Meng, and Y. Zhou, "Recasting type hints from webassembly contracts," in *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering (FSE)*, 2025, pp. 2665–2688.

[30] K. Zhao, Z. Li, J. Li, H. Ye, X. Luo, and T. Chen, "Deepinfer: Deep type inference from smart contract bytecode," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2023, pp. 745–757.

[31] "Signals - circom 2 documentation," https://docs.circom.io/circom-language/signals/, 2025.

[32] "The circom github repository," https://github.com/iden3/circom, 2025.

[33] "snarkjs," https://github.com/iden3/snarkjs, 2025.

[34] "circom2llvm," https://github.com/Veridise/circom2llvm, 2025.

[35] H. Takahashi, J. Kim, S. Jana, and J. Yang, "zkfuzz: Foundation and framework for effective fuzzing of zero-knowledge circuits," 2025. [Online]. Available: https://arxiv.org/abs/2504.11961

[36] A. C. or Firm, "Auditing Report FOR circom-bigint (circomlib)," https://veridise.com/wp-content/uploads/2023/02/VAR-circom-bigint.pdf, 2022.

[37] "Circom 2 documentation," https://docs.circom.io/, Accessed: 2024-06-03.

[38] "Ecneproject," https://github.com/franklynwang/EcneProject, 2022.

[39] "It pays to be circomspect," https://blog.trailofbits.com/2022/09/15/it-pays-to-be-circomspect/, 2022.

[40] 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.

[41] C. Chin, H. Wu, R. Chu, A. Coglio, E. McCarthy, and E. Smith, "Leo: A programming language for formally verified, zero-knowledge applications," Cryptology ePrint Archive, Paper 2021/651, 2021, https://eprint.iacr.org/2021/651. [Online]. Available: https://eprint.iacr.org/2021/651

[42] L. Goldberg, S. Papini, and M. Riabzev, "Cairo – a turing-complete STARK-friendly CPU architecture," Cryptology ePrint Archive, Paper 2021/1063, 2021, https://eprint.iacr.org/2021/1063. [Online]. Available: https://eprint.iacr.org/2021/1063

[43] J. Eberhardt and S. Tai, "Zokrates - scalable privacy-preserving off-chain computations," in *2018 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.

[44] 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)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2024, pp. 1741–1759. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SP54263.2024.00078

[45] 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. [Online]. Available: https://eprint.iacr.org/2023/1278

[46] A. Coglio, E. McCarthy, and E. W. Smith, "Formal verification of zero-knowledge circuits," *Electronic Proceedings in Theoretical Computer Science*, vol. 393, p. 94–112, Nov. 2023. [Online]. Available: http://dx.doi.org/10.4204/EPTCS.393.9

[47] J. Groth, "On the size of pairing-based non-interactive arguments," in *Advances in Cryptology – EUROCRYPT 2016*, M. Fischlin and J.-S. Coron, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 305–326.

[48] A. Gabizon, Z. J. Williamson, and O. Ciobotaru, "PLONK: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge," Cryptology ePrint Archive, Paper 2019/953, 2019, https://eprint.iacr.org/2019/953. [Online]. Available: https://eprint.iacr.org/2019/953

[49] T. Xie, J. Zhang, Y. Zhang, C. Papamanthou, and D. Song, "Libra: Succinct zero-knowledge proofs with optimal prover computation," in *Advances in Cryptology – CRYPTO 2019: 39th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18–22, 2019, Proceedings, Part III*. Berlin, Heidelberg: Springer-Verlag, 2019, p. 733–764. [Online]. Available: https://doi.org/10.1007/978-3-030-26954-8_24

[50] T. Liu, T. Xie, J. Zhang, D. Song, and Y. Zhang, " Pianist: Scalable zkRollups via Fully Distributed Zero-Knowledge Proofs ," in *2024 IEEE Symposium on Security and Privacy (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2024, pp. 1777–1793. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SP54263.2024.00035

[51] C. Li, P. Zhu, Y. Li, C. Hong, W. Qu, and J. Zhang, " HyperPianist: Pianist with Linear-Time Prover and Logarithmic Communication Cost ," in *2025 IEEE Symposium on Security and Privacy (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2025, pp. 3142–3160. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SP61157.2025.00202

[52] "risc0," https://github.com/risc0/risc0, 2025.

[53] "succinctlabs/sp1," https://github.com/succinctlabs/sp1, 2025.

[54] A. Arun, S. Setty, and J. Thaler, "Jolt: Snarks for virtual machines via lookups," in *Advances in Cryptology – EUROCRYPT 2024*, M. Joye and G. Leander, Eds. Cham: Springer Nature Switzerland, 2024, pp. 3–33.

[55] S. Setty, J. Thaler, and R. Wahby, "Unlocking the lookup singularity with lasso," in *Advances in Cryptology – EUROCRYPT 2024: 43rd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zurich, Switzerland, May 26–30, 2024, Proceedings, Part VI*. Berlin, Heidelberg: Springer-Verlag, 2024, p. 180–209. [Online]. Available: https://doi.org/10.1007/978-3-031-58751-1_7

[56] C. Kwan, Q. Dao, and J. Thaler, "Verifying jolt zkVM lookup semantics," Cryptology ePrint Archive, Paper 2024/1841, 2024. [Online]. Available: https://eprint.iacr.org/2024/1841

[57] D. Xiao, Z. Liu, Y. Peng, and S. Wang, "Mtzk: Testing and exploring bugs in zero-knowledge (zk) compilers," in *NDSS*, 2025.

[58] C. Hochrainer, A. Isychev, V. Wüstholz, and M. Christakis, "Fuzzing processing pipelines for zero-knowledge circuits," 2024. [Online]. Available: https://arxiv.org/abs/2411.02077

[59] X. Peng, Z. Sun, K. Zhao, Z. Ma, Z. Li, J. Jiang, X. Luo, and Y. Zhang, "Automated soundness and completeness vetting of polygon {zkEVM}," in *Proceedings of the 34th USENIX Security Symposium (USENIX SEC)*, 2025, pp. 4093–4108.