ConsCS: Effective and Efficient Verification of Circom Circuits

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

Xinghao Peng
Hong Kong Polytechnic University
xing-hao.peng@connect.polyu.hk

Jinzhao Chu
Hong Kong Polytechnic University
jinzhao.chu@polyu.edu.hk

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

***Abstract*—Circom is a popular programming language for writing arithmetic circuits that can be used to generate zero-knowledge proofs (ZKPs) like zk-SNARKS. ZKPs have received tremendous attention in protocols like zkRollups. The Circom circuits are compiled to Rank-1 Constraint Systems (R1CS) circuits, based on which zk-SNARK proofs are generated. However, one major challenge associated with R1CS circuits is the problem of under-constrained circuits, which are susceptible to allowing incorrect computations to pass verification due to insufficient constraints, potentially leading to security vulnerabilities. In this paper, we propose a novel framework CONSCS to automatically verify Circom circuits. Our contributions are threefold: 1) we propose novel circuit inference rules to help reduce the size of circuits and to extract more comprehensive information than existing works; 2) we introduce the novel Binary Property Graph (BPG) as a highly efficient reasoning engine, outperforming all existing tools in effectiveness and efficiency; 3) we leverage fine-grained domain-specific information to guide the SMT solving to address non-linear constraints, increasing the success rate of SMT queries of existing works from 2.68% to 48.84%. We conduct experiments to show that CONSCS enhances the solved rate of existing works from around 50-60% to above 80%.**

I. INTRODUCTION

In the field of blockchain technology, Zero-Knowledge Proofs (ZKPs), and specifically Zero-Knowledge Succinct Non-Interactive Arguments of Knowledge (zk-SNARKs), are critical in enhancing blockchain scalability via techniques like zkRollups. In zkRollups, operators process batches of transactions off-chain and subsequently only post minimal summary data along with succinct zk-SNARK validity proofs to the blockchain [1]. A fundamental component in the construction of zk-SNARKs is the arithmetic circuits, which encode computation into a series of polynomial constraints. These constraints, coupled with private inputs, are used to generate proofs which convince verifiers that the computation was performed correctly, without revealing the private states. Among the various issues associated with arithmetic circuits, **under-constrained circuits** pose a significant challenge [2]–[6]. These circuits are susceptible to incorrect computations passing the verification as correct due to insufficient constraints, which could potentially lead to security vulnerabilities. Circom is a domain-specific programming language designed specifically for defining and compiling arithmetic circuits into Rank-1 Constraint Systems (R1CS) [7], [8]. R1CS is a set of polynomials representing the constraints of the circuit, which are used by zk-SNARKs to generate zero-knowledge proofs.

Multiple real-world instances of under-constrained Circom circuits have been reported [9]–[12]. The Tornado Cash incident [13] is a famous example where an under-constrained bug in one of the Circomlib circuits was exploited by an attacker, allowing the attacker to drain the funds. Thus, it is important to devise automatic methods to verify the constrainedness of R1CS circuits. Several tools have been proposed to address this issue, including Picus (also known as QED^2) [5] and Civer [6]. However, these methods suffer from several limitations.

First, Civer, which has been integrated into the Circom compiler [14], utilizes a coarse-grained approach that relies on only a single SMT query to directly encode the entire circuit to solve the constrainedness, which lacks utilization of more fine-grained information that could be extracted from the circuits. Picus, on the other hand, performs more granular verification at the level of individual variables, but still only directly encodes the entire circuit, achieving a 2.68% success rate of SMT queries. This is challenging because reasoning over non-linear constraints over large prime fields is intractable for SMT solvers [3], [6], [15]. To address these challenges, we propose an assumption-guided SMT solving method (§III-D), which leverages fine-grained circuit information at the constraint level to guide the SMT solving to address non-linear constraints, improving the success rate of SMT queries to 48.84%, a 1,822% increase compared to Picus.

Second, existing works typically rely on a combination of inference rules followed by SMT queries. A full reliance on SMT solvers has the issue that it tends to be more time-consuming and is less explainable since the solver is used as a blackbox without a clear reasoning path. However, it is not a trivial task to design an alternative reasoning framework for R1CS circuits that achieves efficiency, genericness, and explainability. To enhance the efficiency and explanability of the solving process, we introduce the Binary Property Graph (BPG) algorithm (§III-C) as an intermediate processing stage, a highly efficient and scalable reasoning engine. We show via experiments that it outperform all existing tools across various

Corresponding author: Xiapu Luo.

1558-1225/25/$31.00 ©2025 IEEE
DOI 10.1109/ICSE55347.2025.00200

sizes of circuits, and only accounts for 3.82% of the overall solving time per circuit.

Third, another challenge is that existing works performed analysis over the entirety of circuits without simplifications. Circuit simplification is a non-trivial task, since it must ensure a bijective relationship between the original circuit and the simplified version, in order to allow for the preservation of variable relationships and reconstruction of counter-examples. To address this challenge, we propose novel and generic replacement and information extraction inference rules, allowing for simplifying the problem by reducing the number of constraints as well as variables while still preserving essential properties in the reduced circuits (§III-B).

To evaluate our framework, ConsCS (**Cons**trained R1**CS**), we conduct a comparison to show how it improves over existing works across circuits of various sizes (§IV-B). In addition, we conduct an ablation study to demonstrate the effectiveness of different components within ConsCS, as well as a coverage analysis to show that these components generically cover a wide variety of circuits (§IV-C). Furthermore, we discuss the effects of different hyperparameter choices on the BPG algorithm (§IV-A) and present case studies to illustrate how each component contributed to determining the constrainedness of circuits that existing works could not solve (§IV-C4).

In summary, our main contributions are as follows:

- We introduce novel inference rules that are shown to be applicable to a wide range of circuits, enhancing circuit simplification and information extraction.
- We design the Binary Property Graph (BPG) and reasoning algorithms over it, to systematically handle and reduce R1CS constraints of any level of specified complexity, which largely enhances the efficiency, solved rate, and explainability over existing methods.
- We propose an assumption-guided SMT solving method that leveraged fine-grained information at the constraint level to effectively address non-linear constraints, largely enhancing the success rate of SMT queries.
- Overall, as demonstrated by the evaluation, our proposed novel framework, ConsCS, enhances the solved rate of state-of-the-art tools from around 50-60% to above 80%.

II. Background

A. Motivating Example

Consider the motivating example from the Circomlib [16] library shown in Fig. 1a. This Circom program involves point addition on the Montgomery curve [17], a commonly used elliptic curve in cryptography. In Circom, the assignment and equality operators (e.g., $\Leftarrow==$ and $===$) are transformed into polynomial constraints within the R1CS, ensuring that the cryptographic proofs generated reflect the intended computations of the Circom program. Fig. 1b displays the R1CS circuit compiled from the Circom program. It has three constraints, translated from the three Circom assignment or equality statements shown in Fig. 1a. In this example, the input signals at lines 2 and 3 define two points on the curve, in_1 and in_2, and the output signal at line 4 represents the addition

```
1  template MontgomeryAdd() {
2      signal input in1[2];
3      signal input in2[2];
4      signal output out[2];
5      var a = 168700;
6      var d = 168696;
7      var A = (2 * (a + d)) / (a - d);
8      var B = 4 / (a - d);
9
10     signal lamda;
11     lamda <-- (in2[1] - in1[1]) / (in2[0] - in1[0]);
12     lamda * (in2[0] - in1[0]) === (in2[1] - in1[1]);
13     out[0] <== B*lamda*lamda - A - in1[0] -in2[0];
14     out[1] <== lamda * (in1[0] - out[0]) - in1[1];
15 }
```

(a) The Circom program.

$$\begin{cases} 0 + (p-1)\cdot x_3x_7 + x_5x_7 + x_4 + (p-1)\cdot x_6 & \equiv 0 \pmod p, \\ 168698 + (p-1)\cdot x_7x_7 + x_1 + x_3 + x_5 & \equiv 0 \pmod p, \\ 0 + x_1x_7 + (p-1)\cdot x_3x_7 + x_2 + x_4 & \equiv 0 \pmod p. \end{cases}$$

(b) The compiled R1CS circuit. λ is translated to x_7, in_1 to (x_3, x_4), in_2 to (x_5, x_6), out to (x_1, x_2), $p-1$ is congruent to -1.

Fig. 1: The motivating example

result. The first and second elements of these signals are the x and y coordinates, respectively. At line 11, there is a critical computation of a λ value, defined as $\lambda = \frac{in_2[1]-in_1[1]}{in_2[0]-in_1[0]}$, which is needed for point addition under the assumption that in_1 and in_2 are distinct points. However, its correctness is actually verified by the equality statement $\lambda(in_2[0] - in_1[0]) = in_2[1] - in_1[1]$ specified at the next line, line 12, and is translated into the R1CS constraint at the first equation of Fig. 1b. It is worth noting that this R1CS constraint misses one important component: its equality allows $in_2[0] - in_1[0] = 0$ to pass the verification, by making both sides trivially 0, but it is an invalid input for the computation of λ, since 0 does not have an inverse in finite fields. This is a misalignment between the intended computation and the correctness check specified by the programmer. Due to this, the compiled R1CS circuit allows identical input points to pass the verification, which is incorrect, as addition of identical points (i.e., point doubling) should be handled differently. This is an example of an **under-constrained circuit** with insufficient constraints due to programming negligence.

B. Zero Knowledge Proofs

Zero-Knowledge Proof (ZKP) is a cryptographic technique that enables one party (the prover) to prove to another party (the verifier) that a certain statement is true, without revealing any information beyond the statement's validity [18]. Among the types of ZKPs, zk-SNARK (Zero-Knowledge Succinct Non-interactive Argument of Knowledge) stands out for its efficiency and minimal communication cost. zk-SNARKs are used in many cryptographic implementations such as Ligero [19], Marlin [20], Groth [21], Plonk [22], Libra [23], and Pianist [24]. To illustrate zk-SNARKs, consider Alice who wants to prove to Bob that she knows the solution to a public Sudoku puzzle, but without revealing the solution numbers. Both know the puzzle, but only Alice knows the solution. Using a zk-SNARK, Alice (the prover) can convince Bob

(the verifier) of her knowledge while ensuring Bob learns nothing about the solution. One notable application of zk-SNARK in software systems is in blockchain technology. As demand on blockchain networks grows, the main chain can become congested, struggling to process an increasing number of transactions efficiently. zkRollup [1] addresses this challenge by processing batches of transactions off-chain. In a zkRollup, only the final state changes and an efficiently verifiable zero-knowledge proof are posted to the main chain. This approach uses a single, succinct proof to ensure that batches of transactions are correctly processed without overloading the main blockchain, significantly enhancing scalability and throughput. Examples of zkRollup implementations include zkSync [25], Polygon [26], and Scroll [27]. Beyond blockchain, zk-SNARK has potential applications in software systems such as private voting systems [28], [29] and digital identity verification [30], where privacy and security are critical.

C. Arithmetic Circuits and Circom

Arithmetic circuits, a central component of ZKPs, represent computational problems using addition and multiplication gates. They function like complex puzzles where the prover knows the solution (i.e., a satisfying assignment to the circuit gates) but wants to prove its existence without revealing it. These circuits can model a wide variety of problems, such as verifying password hashes, validating transactions, or performing computations over encrypted data in privacy-preserving protocols. Circom [8] is a popular programming language designed for creating these arithmetic circuits. It allows developers to write succinct, high-level code that compiles into the R1CS (Rank-1 Constraint System) form, a set of equations over a finite field representing the arithmetic circuit. By providing satisfying assignments to the circuit's inputs, the R1CS constraints can be used to generate a zk-SNARK proof, revealing nothing about the private inputs. Circom's syntax resembles that of Java and C++, using curly braces to define code blocks, semicolons to terminate statements, and similar comment syntax. However, Circom introduces unique operators like <== and === to generate R1CS constraints. For instance, the statement $a === b * c;$ compiles to the R1CS constraint $a - bc \equiv 0 \pmod{p}$, enforcing that a equals b multiplied by c modulo a prime field p. Circom supports control flow statements (*if*, *for*, *while*) and basic data types (arrays, integers). All operations are over a finite field with a specified large prime. An example Circom program and its compiled R1CS constraints are shown in Figure 1a and Figure 1b. Circom has a growing developer community and an ecosystem of libraries and tools, such as *snarkjs* [31], which facilitate the generation and verification of ZKP proofs from circuits defined in Circom.

D. Constrainedness of Circuits

In an R1CS arithmetic circuit, each constraint is a degree-two multivariate polynomial defined over $\mathbb{F}_p$. The general form of one constraint [22] is given by:

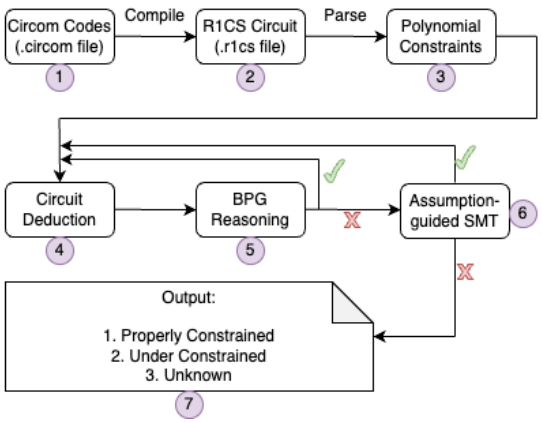

Fig. 2: The workflow of ConsCS.

$$\left(\sum_{j\in[m]} a_j x_j\right) \cdot \left(\sum_{j\in[m]} b_j x_j\right) - \sum_{j\in[m]} c_j x_j \equiv 0 \pmod{p} \quad (1)$$

In this setup, each degree-two polynomial constraint is defined over the finite field $\mathbb{F}_p$ and involves a certain number of variables, referred to as signals or wires. A small example of three such constraints is shown in Fig. 1b. In practice, the number of variables and constraints could scale up to tens of thousands. One R1CS constraint as shown in Equation (1) can be succinctly represented as $f_i = 0$, and thus an entire circuit with n constraints as $\{f_1 = 0, \ldots, f_n = 0\}$ [5]. A certain witness satisfies the circuit if $\bigwedge_{i=1}^{n} f_i = 0$. A critical aspect in the design and verification of arithmetic circuits is the determination of the constrainedness of circuits [2]–[6], which is the primary focus of this paper. Below, we introduce the formal definition of the constrainedness of circuits.

Properly Constrained Circuit. A circuit is *properly constrained* if:

$$\forall v_i \in \mathcal{V}_{out}, \mathcal{C}_{in} \in \mathbb{F}_p^{|\mathcal{V}_{in}|} : \mathcal{V}_{in} \equiv \mathcal{C}_{in} \Rightarrow v_i \equiv c_i \quad (2)$$

Here, $\mathcal{C}_{in} \in \mathbb{F}_p^{|\mathcal{V}_{in}|}$ represents a valid assignment to the input signals, and $v_i \equiv c_i$ indicates that each output signal v_i is uniquely determined to a specific value c_i. In other words, for every valid input signal assignment, each output signal in the circuit is uniquely determined.

Under-constrained Circuit. A circuit is *under-constrained* if:

$$\exists v_i \in \mathcal{V}_{out}, \mathcal{C}_{in} \in \mathbb{F}_p^{|\mathcal{V}_{in}|} : \mathcal{V}_{in} \equiv \mathcal{C}_{in} \Rightarrow \bigvee_{j=1}^{n} (v_i \equiv c_j) \quad (3)$$

Here, $c_{j_1} \neq c_{j_2}$ for $j_1 \neq j_2$, and $\{c_1, c_2, \ldots, c_n\}$ is the set of valid values that the output variable v_i could take. In other words, given some valid assignment to the input signals, there is an output signal that can take multiple different valid values.

III. Method

A. Overview

Fig. 2 shows the high-level overview of ConsCS. Initially, Circom codes (stage ①) are compiled into R1CS circuits using

the Circom compiler (stage ②), from which the polynomial constraints $\{f_1 = 0, \ldots, f_n = 0\}$ are parsed (stage ③). These constraints are then processed through a pipeline comprising three stages (i.e., stages ④, ⑤, and ⑥), each dedicated to extracting constrainedness information at the variable level. When new information of a variable is identified, such as being constrained or binary, a backward update, indicated by a green ✓, is initiated to re-process the constraints that involve the variable. This is because the newly identified information could also potentially be helpful for previous stages. If no new information is identified at a stage, indicated by a red × in the figure, it proceeds to the next stage. These three stages are presented in Sections III-B, III-C, and III-D respectively. The final outputs are either the constrainedness of the circuit or *Unknown*. While we focus on Circom circuits, ConsCS could be readily adapted to other zk-SNARK languages that generate R1CS circuits by substituting stages ① and ②.

Notation. One R1CS constraint is represented using the general form $\sum_{i=1}^{m} c_i v_{2i-1} v_{2i} + c_{m+1} \equiv 0 \pmod p$, expanded from Equation (1), where v_i denotes a single variable and c_i represents a constant. A *term* represents both degree-two variable products (e.g., $v_1 v_2$) and single variables (e.g., v_1). $\mathcal{V}_{in}$ and $\mathcal{V}_{out}$ represent the set of input and output signal variables, respectively. $\mathcal{T}_{cons}$ denotes the set of *terms* that are inferred to be properly constrained, and is initialized to include all input signals (i.e., $\mathcal{T}_{cons} \Leftarrow \mathcal{V}_{in}$), since input variables are inherently uniquely determined by themselves. $\mathcal{V}_{bin}$ represents the set of single variables that are inferred to be binary, taking the value of either zero or one. Therefore, a circuit is *properly constrained* if all output variables are constrained (i.e., $\mathcal{V}_{out} \subseteq \mathcal{V}_{cons}$).

$$\frac{v_1 \in \mathcal{T}_{cons} \wedge v_2 \in \mathcal{T}_{cons}}{\mathcal{T}_{cons} \leftarrow \mathcal{T}_{cons} \cup \{v_1 v_2\}} \tag{4}$$

$$\frac{c_1 v_i + c_2 v_j + c_3 \equiv 0, i < j, v_j \notin \mathcal{V}_{in} \cup \mathcal{V}_{out}}{v_j \leftarrow (-c_3 - c_1 v_i) c_2^{-1}} \tag{5}$$

$$\frac{\begin{array}{l}\sum_{i=1}^{m} c_i v_{2i-1} v_{2i} + c_{m+1} \equiv 0, \\ \exists m\text{: } v_{2m-1} v_{2m} \notin \mathcal{T}_{cons}, \forall n \neq m\text{: } v_{2n-1} v_{2n} \in \mathcal{T}_{cons}\end{array}}{\mathcal{T}_{cons} \leftarrow \mathcal{T}_{cons} \cup \{v_{2m-1} v_{2m}\}} \tag{6}$$

$$\frac{\begin{array}{l}\sum_{i=1}^{n} c_i v_i + \sum_{j=n+1}^{m} c_j v_{2j-1} v_{2j} + c_{m+1} \equiv 0, \\ \forall i \in [1, n]\text{: } v_i \in \mathcal{V}_{bin}, c_1 \equiv 1, c_i \equiv 2 \times c_{i-1}, \\ \forall j \in [n+1, m]\text{: } v_{2j-1} v_{2j} \in \mathcal{T}_{cons}\end{array}}{\mathcal{T}_{cons} \leftarrow \mathcal{T}_{cons} \cup \{v_i \mid i \in [1, n]\}} \tag{7}$$

$$\frac{\begin{array}{l}v_1 \times (c_1 + \sum_{i=2}^{n} c_i v_i) + \sum_{i=n+1}^{m} c_i v_{2i-1} v_{2i} + c_{m+1} \equiv 0, \\ v_1 \notin \mathcal{T}_{cons}, \forall i \in [2, 2m]\text{: } v_i \in \mathcal{T}_{cons}\end{array}}{\mathcal{T}_{as} \leftarrow \mathcal{T}_{as} \cup \{(v_1, c_1 + \sum_{i=2}^{n} c_i v_i \equiv 0)\}} \tag{8}$$

B. The Circuit Deduction Rules

Motivation. In this section, we describe stage ④ of ConsCS. This crucial pre-processing step involves 5 inference rules (IRs) aimed at both extracting information and simplifying circuits. Pre-processing on circuits to extract useful information is a commonly adopted approach in existing works. For instance, Civer [6] utilizes 11 IRs to infer range information (e.g., lower/upper bounds) of variables, thereby trimming the search space. Similarly, Picus [5] adopts 13 IRs. Below, we introduce each of our 5 IRs as shown in Equations (4) through (8), and how they compare to existing works. All arithmetic is done in $\mathbb{F}_p$.

1) IR 1: Single to Double: : As shown in Equation (4), given two variables that are already inferred to be constrained (i.e., $v_1 \in \mathcal{T}_{cons} \wedge v_2 \in \mathcal{T}_{cons}$), it can be inferred that their product is also constrained. This inference propagates the constrainedness from individual variables to their product terms.

2) IR 2: Simple Solve: : As shown in Equation (5), simple linear constraints in the form of $c_1 v_i + c_2 v_j + c_3 \equiv 0 \pmod p$ enable direct variable substitution, with v_j removed after the replacement. This type of simple linear constraint is commonly compiled from assignment operations in Circom. The replacement of v_j is performed on all constraints involving v_j, reducing the number of variables and hence complexity of the constraint system, while maintaining a one-to-one correspondence with the original system, ensuring that the constrainedness is preserved.

3) IR 3: Single Unknown: : In Circom, it is common to assign a linear combination of known constrained terms to an unknown term. As shown in Equation (6), if all terms except one ($v_{2m-1} v_{2m}$) are known to be constrained, this means the unknown term is constrained to the linear combination of known terms. This rule is crucial for efficiently assigning constrainedness to previously unknown terms.

4) IR 4: Binary Expression: : Binary expressions are common components in Circom circuits. For a binary expression with all its variables being binary bits, it poses a one-to-one relationship with its represented value. Therefore, the constrainedness of the set of binary bits is implied by that of the value they represent, as shown in Equation (7).

5) IR 5: Assumptions Extraction: : As outlined in Equation (8), this rule addresses non-linear constraints by isolating one variable v_1 in a constraint as the assumption variable, and the linear polynomial it multiplies (i.e., $c_1 + \sum_{i=2}^{n} c_i v_i$) as the assumption condition, which are put together as a tuple and are stored in the set of all recorded assumptions $\mathcal{T}_{as}$. Stage ⑥ leverages an SMT solver on $\mathcal{T}_{as}$ to infer the constrainedness of the isolated assumption variable. We delegate the detailed discussion of how $\mathcal{T}_{as}$ is utilized to Section III-D.

C. The Binary Property Graph (BPG)

Motivation. As introduced in Section I, existing works utilize the methodology of circuit deduction followed by SMT queries, which fully relies on the SMT solver to determine the constrainedness of circuits. However, there are several limitations. First, existing approaches only use the direct coarse-grained encoding of the entire circuit, without leveraging fine-grained constraint-level information. Second, the SMT solver operates as a black box, and a sole reliance on it obscures the rationale behind the inferred information. Third, as

$$\frac{\texttt{deduce}(\sum_{i=1}^{1} c_i v_{2i-1} v_{2i} + c_2 \equiv 0 \pmod p)}{\begin{array}{l}(\mathcal{P}_0(c_2) \Rightarrow (\overline{\mathcal{P}}_0(v_1) \Rightarrow \mathcal{P}_0(v_2))) \\ \wedge(\overline{\mathcal{P}}_0(c_2) \Rightarrow (\overline{\mathcal{P}}_0(v_1) \wedge \overline{\mathcal{P}}_0(v_2)))\end{array}} \tag{9}$$

$$\frac{\texttt{deduce}(\sum_{i=1}^{2} c_i v_{2i-1} v_{2i} + c_3 \equiv 0 \pmod p)}{\begin{array}{l}(\overline{\mathcal{P}}_0(v_1) \wedge \overline{\mathcal{P}}_0(v_3) \Rightarrow (\mathcal{P}_c(v_2) \Leftrightarrow \mathcal{P}_c(v_4))) \\ \wedge(\mathcal{P}_0(c_1 + c_2) \wedge \mathcal{P}_1(v_4) \wedge v_1 = v_2 \wedge v_1 = v_3 \wedge \mathcal{P}_0(c_3) \\ \quad \Rightarrow (\overline{\mathcal{P}}_0(v_1) \Rightarrow \mathcal{P}_1(v_1))) \\ \wedge(\mathcal{P}_0(v_1) \vee \mathcal{P}_0(v_2) \Rightarrow \texttt{deduce}(\sum_{i=2}^{2} c_i v_{2i-1} v_{2i} + c_3)) \\ \wedge(\overline{\mathcal{P}}_0(v_1) \wedge \overline{\mathcal{P}}_0(v_2) \Rightarrow \texttt{deduce}(\sum_{i=2}^{2} c_i v_{2i-1} v_{2i} + {c_3}^*)) \\ \wedge(\mathcal{P}_1(v_3) \wedge \mathcal{P}_0(c_1 + c_2) \wedge v_1 = v_4 \\ \quad \Rightarrow \texttt{deduce}(\sum_{i=1}^{1} c_i v_{2i-1}(v_{2i} - 1) + c_3))\end{array}} \tag{10}$$

$$\frac{\texttt{deduce}(\sum_{i=1}^{3} c_i v_{2i-1} v_{2i} + c_4 \equiv 0 \pmod p)}{\begin{array}{l}(\mathcal{P}_0(v_1) \vee \mathcal{P}_0(v_2) \Rightarrow \texttt{deduce}(\sum_{i=2}^{3} c_i v_{2i-1} v_{2i} + c_4)) \\ \wedge(\overline{\mathcal{P}}_0(v_1) \wedge \overline{\mathcal{P}}_0(v_2) \Rightarrow \texttt{deduce}(\sum_{i=2}^{3} c_i v_{2i-1} v_{2i} + {c_4}^*))\end{array}} \tag{11}$$

$$\frac{\texttt{deduce}(\sum_{i=1}^{m} c_i v_{2i-1} v_{2i} + c_{m+1} \equiv 0 \pmod p)}{\begin{array}{l}(\mathcal{P}_0(v_1) \vee \mathcal{P}_0(v_2) \Rightarrow \texttt{deduce}(\sum_{i=2}^{m} c_i v_{2i-1} v_{2i} + c_{m+1})) \\ \wedge(\overline{\mathcal{P}}_0(v_1) \wedge \overline{\mathcal{P}}_0(v_2) \Rightarrow \texttt{deduce}(\sum_{i=2}^{m} c_i v_{2i-1} v_{2i} + c^*_{m+1}))\end{array}} \tag{12}$$

$$\frac{\phi}{\begin{array}{l}(\mathcal{P}_0(v_1) \Rightarrow \overline{\mathcal{P}}_1(v_1) \wedge \mathcal{P}_c(v_1)) \\ \wedge(\mathcal{P}_1(v_1) \Rightarrow \overline{\mathcal{P}}_0(v_1) \wedge \mathcal{P}_c(v_1)) \\ \wedge(\overline{\mathcal{P}}_0(v_1) \Rightarrow \mathcal{P}_c(v_1)) \\ \wedge(\overline{\mathcal{P}}_1(v_1) \Rightarrow \mathcal{P}_c(v_1))\end{array}} \tag{13}$$

demonstrated in our experiments in Section IV, the direct use of SMT queries for checking constrainedness is less time-efficient. To address these issues, we introduce a novel, lightweight reasoning approach called the Binary Property Graph (BPG), which significantly improves both efficiency and effectiveness compared to current methods.

1) Preliminaries: The BPG is structured around the binary properties of variables. Binary properties are fundamental to the structure of circuits, and are very common in Circom circuits, such as transformations into or from binary bits, mux selectors, addition of binaries, bit shifting, etc. It is also common for variables to be constrained as binary in Circom. For instance, since the release of Circom 2.1.0, signal tags [32] have been introduced at the native programming language level, allowing programmers to explicitly specify signals as binary via a {*binary*} tag. Furthermore, our analysis in Section IV indicates that binary properties are found in 85.23% of the Circomlib circuits. Specifically, we define the binary property function $\mathcal{P}_c(v_i)$ as follows:

$$\mathcal{P}_c(v_i) = \begin{cases} \text{True} & \text{if } v_i = c \\ \text{False} & \text{otherwise} \end{cases}, \text{ where } c \in \{0, 1\}$$

Additionally, its negation $\overline{\mathcal{P}}_c(v_i)$ checks the inequality condition. For instance, $\mathcal{P}_1(v_2)$ is True if $v_2 = 1$ and False otherwise, while $\overline{\mathcal{P}}_0(v_3)$ is True if $v_3 \neq 0$. This notation is also extended to use $\mathcal{P}_c(v_i)$ to indicate that v_i is some constant from $\mathbb{F}_p$. Therefore, there are five types of binary properties: $\overline{\mathcal{P}}_0(v_1), \mathcal{P}_0(v_1), \overline{\mathcal{P}}_1(v_1), \mathcal{P}_1(v_1), \mathcal{P}_c(v_1)$.

2) Construction of the Graph: The nodes and edges of BPG are defined as follows:

a) *Node*: Each node is a tuple $(v_i, \mathcal{P}_c(v_i))$, where v_i is a variable and $\mathcal{P}_c(v_i)$ represents one of its binary properties.
b) *Edge*: Each edge is a tuple (*from*, *to*, *edge_condition*), where the property of the *from* node implies the property of the *to* node under the *edge_condition*. The *edge_condition* is a list of multiple binary properties of different variables. An empty *edge_condition*, denoted as ϕ, indicates that the edge is a direct implication without conditions.

For instance, an edge $([v_1, \mathcal{P}_0(v_1)], [v_2, \overline{\mathcal{P}}_0(v_2)], \phi)$ denotes that if $v_1 \equiv 0$, it is directly inferred that $v_2 \neq 0$ with no conditions. We define **depth** as the number of variable terms in the constraint. For example, $v_1 v_2 + c \equiv 0$ has depth 1, $v_1 v_2 + v_3 + c \equiv 0$ has depth 2, and $v_1 v_2 + v_3 + v_4 v_5 + c \equiv 0$ has depth 3. The BPG is structured based on a set of binary property inference rules (IRs), ranging from specific (depth-1 constraints) to general (depth-m constraints), as shown in Equations (9) through (12).

Edges are constructed directly from implications (i.e., $\Rightarrow, \Leftrightarrow$). The **edge condition** of an edge is a list that encodes antecedent conditions which cannot fit in one node. Edge conditions are used for checking that the traversal between nodes satisfies all antecedents. Consider the three illustrative examples below:

Example 1. For *direct implications* such as $\mathcal{P}_0(v_1) \Rightarrow (\overline{\mathcal{P}}_0(v_2) \Rightarrow \mathcal{P}_c(v_3))$, we create an edge $\mathcal{P}_0(v_1) \xrightarrow{\phi} \overline{\mathcal{P}}_0(v_2)$ without conditions, and another edge $\overline{\mathcal{P}}_0(v_2) \xrightarrow{[\mathcal{P}_0(v_1)]} \mathcal{P}_c(v_3)$ with the condition $[\mathcal{P}_0(v_1)]$, meaning $\mathcal{P}_0(v_1)$ must be satisfied for $\overline{\mathcal{P}}_0(v_2)$ to traverse to $\mathcal{P}_c(v_3)$.

Example 2. For *disjunctive connectives* such as $(\mathcal{P}_0(v_1) \vee \overline{\mathcal{P}}_0(v_2)) \Rightarrow \mathcal{P}_c(v_3)$, we add two edges, $\mathcal{P}_0(v_1) \xrightarrow{\phi} \mathcal{P}_c(v_3)$ and $\overline{\mathcal{P}}_0(v_2) \xrightarrow{\phi} \mathcal{P}_c(v_3)$, both without conditions. This is because either $\mathcal{P}_0(v_1)$ or $\overline{\mathcal{P}}_0(v_2)$ could independently derive $\mathcal{P}_c(v_3)$.

Example 3. For *conjunctive connectives* such as $(\mathcal{P}_0(v_1) \wedge \overline{\mathcal{P}}_0(v_2) \wedge \mathcal{P}_c(v_3)) \Rightarrow \mathcal{P}_1(v_4)$, we introduce three edges, $\mathcal{P}_0(v_1) \xrightarrow{[\overline{\mathcal{P}}_0(v_2), \mathcal{P}_c(v_3)]} \mathcal{P}_1(v_4)$, $\overline{\mathcal{P}}_0(v_2) \xrightarrow{[\mathcal{P}_0(v_1), \mathcal{P}_c(v_3)]} \mathcal{P}_1(v_4)$, and $\mathcal{P}_c(v_3) \xrightarrow{[\mathcal{P}_0(v_1), \overline{\mathcal{P}}_0(v_2)]} \mathcal{P}_1(v_4)$, meaning that all components of the conjunction antecedent are required to make a traversal.

Longer reasoning chains are handled recursively, reducing to base cases as in the examples shown above. The inference rules used to build edges are specified using the `deduce` function, which takes as input a constraint, and recursively infers new information and adds inferential relations to the BPG graph. We explain each of the inference rules (IRs) below.

a) **Depth-1 constraints:** Inference rule (9) addresses depth-1 (i.e., single-term) constraint polynomials. Specifically, if $v1v2 \equiv 0 \pmod p$, then one variable must be zero (i.e., $\mathcal{P}_0(v_1) \vee \mathcal{P}_0(v_2)$, equivalently $\overline{\mathcal{P}}_0(v_1) \Rightarrow \mathcal{P}_0(v_2)$). Conversely, if $v1v2 \neq 0 \pmod p$, then neither variable can be zero (i.e., $\overline{\mathcal{P}}_0(v_1) \wedge \overline{\mathcal{P}}_0(v_2)$). This IR serves as the

base case for the deduce recursive calls, ensuring the termination of the recursion.

b) **Depth-2 constraints:** Inference rule (10) addresses depth-2 (i.e., double-term) constraints. Specifically, if the constraint is of the form $c_1v_2 + c_2v_4 + c_4 \equiv 0 \pmod p$, then the constrainedness of v_2 or v_4 implies the other's constrainedness (i.e., $\mathcal{P}_c(v_2) \Leftrightarrow \mathcal{P}_c(v_4)$). Additionally, an equation like $c_1v_1v_1 - c_1v_1 \equiv 0 \pmod p$ implies $v_1 \in \{0, 1\}$, hence $\overline{\mathcal{P}}_0(v_1) \Rightarrow \mathcal{P}_1(v_1)$. Reduction to a depth-1 constraint occurs if one variable in a term is zero, prompting a recursive call to deduce. If terms consist solely of non-zero constants, they merge with c_3, getting turned into c_3^*, indicating a dirty constant with an unknown binary property. Furthermore, the resursive call where a variable $(v_1 - 1)$ appears in the place of v_1 is also considered. This is special in the sense that their binary properties are reversed: $\mathcal{P}_1(v_1) \Leftrightarrow \mathcal{P}_0(v_1 - 1)$, $\overline{\mathcal{P}}_1(v_1) \Leftrightarrow \overline{\mathcal{P}}_0(v_1 - 1)$, etc.

c) **General (depth-m) constraints:** Extending the reasoning to depth-3 equations, Inference Rule (11) follows a similar reduction approach as the second rule, considering one of the terms being turned into either zero or a dirty constant. These reductions continue recursively, handled by Inference Rule (12), which systematically reduces depth-m equations to depth-(m-1) equations, terminating at the base case of depth-1. Thus, **depth** is an adjustable hyperparameter. The impact of varying the depth is discussed via experiments in Section IV-A.

Furthermore, Equation (13) specifies the unconditional inference with empty premise ϕ. All expressions involving variables v_1 and v_2 are commutative. Additionally, unless specified otherwise, we by default assume all coefficients c_i to be non-zero, and $v_i \neq v_j$ for $i \neq j$. For example, the constraint $x_1 \times (x_2 - 1) \equiv 0 \pmod p$ encodes that either $x_1 \equiv 0$ or $x_2 \equiv 1$, on which Equation (10) serves as an entry point of inference, and Equation (9) is recursively called.

3) Traversal of the Graph: Traversal over the graph is a lightweight reasoning process for R1CS circuits, designed to exploit the complementary nature of binary properties recorded in the BPG. The intuition is that given that one of two complementary cases must happen, if certain properties of a variable consistently holds in both cases, then such property is inferred to be universal. Specifically, the complementary pairs we consider are $[\overline{\mathcal{P}}_0(v_1), \mathcal{P}_0(v_1)]$, $[\overline{\mathcal{P}}_1(v_1), \mathcal{P}_1(v_1)]$, and $[\overline{\mathcal{P}}_0(v_1), \overline{\mathcal{P}}_1(v_1)]$. For example, if $\overline{\mathcal{P}}_0(v_1) \Rightarrow \mathcal{P}_0(v_2)$ and $\overline{\mathcal{P}}_1(v_1) \Rightarrow \mathcal{P}_1(v_2)$, then v_2 must be binary; similarly, their contrapositives also imply v_1 to be binary. Since $\mathcal{P}_c(v)$ is an abstract identifier indicating v is inferred to constrain to some constant, this binary property does not have a complement.

The BPG solving is shown in Algorithm 1. It includes a Depth First Search (DFS) over each pair of complementary nodes (line 4), collecting nodes that can be inferred from each binary property of the pair, excluding the starting node. Each set of the collected nodes via DFS is called the *consequence*, denoted by C_i. The comparison of consequences from these searches determines the inferred universal properties

Algorithm 1 solve_BPG

```
1:  procedure SOLVE_BPG(G_bpg)   ▷ G_bpg: the BPG graph
2:      N_comp ← get_complementary_pairs()
3:      for (N_1, N_2) ∈ N_comp do   ▷ N_comp: a list of pairs
4:          C_1, C_2 = DFS(N_1), DFS(N_2) ▷ C_i: consequence
5:          for v_i ∈ V do   ▷ V: set of all variables
6:              if v_i ∈ C_1.vars ∧ v_i ∈ C_2.vars ∧ N_1 ∈ T_cons ∧
    N_2 ∈ T_cons  then
7:                  T_cons ← T_cons ∪ {v_i}
8:              end if
9:              if (P_0(v_i) ∈ C_1 ∧ P_1(v_i) ∈ C_2) ∨ (P_1(v_i) ∈
    C_1 ∧ P_0(v_i) ∈ C_2) then
10:                 V_bin ← V_bin ∪ {v_i}
11:             end if
12:             if (P_0(v_i) ∈ C_1 ∧ P_0(v_i) ∈ C_2) then
13:                 v_i ← 0
14:             end if
15:             if (P_1(v_i) ∈ C_1 ∧ P_1(v_i) ∈ C_2) then
16:                 v_i ← 1
17:             end if
18:         end for
19:     end for
20: end procedure
```

of variables. If a variable appears as constrained in both consequences and their starting nodes are both constrained, then the variable must be consistently constrained (line 6). A variable appearing as zero in one consequence and one in the other is inferred to be binary (line 9). If it consistently appears as either zero or one in both consequences, it is assigned that particular value (lines 12 and 15). Overall, the possible outcomes of this algorithm is to infer variables as being zero, one, binary, or constrained.

4) Complexity Analysis: Denote the number of variables (i.e., wires) in a circuit by n. Nodes represent variables tagged with one of the five binary properties $(\overline{\mathcal{P}}_0(v_1), \mathcal{P}_0(v_1), \overline{\mathcal{P}}_1(v_1), \mathcal{P}_1(v_1), \mathcal{P}_c(v_1))$, so the number of nodes is proportional to n, scaling with $O(n)$. In the worst-case, each node infers all other nodes, so the number of edges scales quadratically with $O(n^2)$. The analysis algorithm consists of DFS traversals combined with a loop over all variables, so the theoretical complexity of our algorithm is $O(n + n^2 + n)$, simplifying to $O(n^2)$. Despite this quadratic theoretical worst-case complexity, empirical observations suggest a sparse graph with linear performance across various sizes of circuits, which we discuss in more detail in Section IV-D.

D. The Assumption-Guided SMT solving

Motivation. This stage is to enhance the reasoning of non-linear constraints over the finite field. Non-linear constraints are essential for the security of circuits, because they are very challenging to reason with; currently, directly solving non-linear constraints over large prime fields is intractable for SMT solvers [3], [6], [15]. To address this challenge, we leverage assumptions to provide fine-grained guidance for the solver

Algorithm 2 solve_assumption

```
1: procedure SOLVE_ASSUMPTION(V_as, E_cond, solver)
2:     counter_example ← ∅
3:     solver.push()
4:     solver.add(E_cond ≡ 0 (mod p))
5:     res ← solver.check()
6:     if res ≡ UNSAT then
7:         T_cons ← T_cons ∪ {V_as}
8:     else if res ≡ SAT then
9:         counter_example ← solver.find_counter_example()
10:    end if
11:    solver.pop()
12:    return counter_example    ▷ empty (φ) if none found
13: end procedure
```

to reason about the constrainedness of variables. Currently, existing tools only conduct direct queries without any guidance.

Equation (8) from Section III-B specifies how the assumption tuples are extracted. Specifically, let $\mathcal{S}$ be the set of all R1CS constraints in the circuit, this stage handles each non-linear constraint s of the form: $s : v_1 \times \mathcal{E}_{cond} = \mathcal{E}_{other}, s \in \mathcal{S}$, where an assumption variable $\mathcal{V}_{as} = v_1$ can be isolated to multiply a known constrained linear expression $\mathcal{E}_{cond} = c_1 + \sum_{i=2}^{n} c_i v_i$, equating to another known constrained expression $\mathcal{E}_{other} = -\sum_{i=n+1}^{m} c_i v_{2i-1} v_{2i} + c_{m+1}$. This leads to the formation of an assumption tuple $(\mathcal{V}_{as}, \mathcal{E}_{cond})$. If $\mathcal{E}_{cond}$ or $\mathcal{E}_{other}$ were not known to be constrained, then s would not be considered. Algorithm 2 shows the solving process for each assumption tuple. A Z3 [33], [34] *solver* object with all circuit constraints (i.e., $\mathcal{S}$) already added is also provided as input. Then, the solver state is pushed (line 3) to utilize Z3's incremental solving feature, which allows for the reuse of learned clauses and avoids re-encoding the entire circuit for each call [35].

Next, the condition for checking the assumption tuple (i.e., $\mathcal{E}_{cond} \equiv 0$) is added (line 4) and its satisfiability, along with the entire circuit encoding, is checked (line 5). The two specific queries are:

$$\mathcal{Q}_1 : \textit{UNSAT}(\mathcal{S} \wedge \mathcal{E}_{cond} \equiv 0) \qquad \mathcal{Q}_2 : \textit{SAT}(\mathcal{S})$$

If $\mathcal{Q}_1 \equiv$ UNSAT and $\mathcal{Q}_2 \equiv$ SAT are returned, it implies that $(\mathcal{S} \wedge \mathcal{E}_{cond} == 0) \equiv False$, leading to $(\neg\mathcal{S} \vee \mathcal{E}_{cond} \neq 0) \equiv True$, whose logical equivalent is $(\mathcal{S} \Rightarrow \mathcal{E}_{cond} \neq 0) \equiv True$. Given $\mathcal{Q}_2 \equiv$ SAT, it follows from this implication that $\mathcal{E}_{cond} \neq 0$ under the R1CS system. This renders $\mathcal{E}_{cond}$ invertible, so v_1 can be expressed as $\mathcal{E}_{other} \times \mathcal{E}_{cond}^{-1}$, thereby constrained since both expressions $\mathcal{E}_{cond}$ and $\mathcal{E}_{other}$ are known to be constrained, then v_1 must be constrained.

Conversely, a SAT result with $\mathcal{Q}_1 \equiv$ SAT and $\mathcal{Q}_2 \equiv$ SAT implies that one satisfying assignment to all wires is found, and $\mathcal{E}_{cond}$ and $\mathcal{E}_{other}$ would both be 0. This implies that v_1 can take any arbitrary value and yet still satisfy the constraint $s : v_1 \times \mathcal{E}_{cond} = \mathcal{E}_{other}$, potentially introducing under-constrainedness. Therefore, a second SMT query is ran (line 9) to look for an alternative assignment with the same inputs reported from $\mathcal{Q}_1$, but different outputs. If one is found, it serves as a counter-example and by definition proves the under-constrainedness of the circuit. Otherwise, *None* is returned, suggesting that no new information was identified. Two example circuits are provided at Section IV-C4 as a further case study.

We also apply several optimizations via Groebner basis and domain-specific conflict-driven clause learning (CDCL). First, we incorporate an advanced encoding technique to augment the circuit representation $\mathcal{S}$, helping unearth variable-level information. Specifically, we leverage the Groebner basis of $\mathcal{S}$, computed via Sympy's *f5b* method [36]. The Groebner basis can be viewed as a generalization of the classical Gaussian elimination on linear matrices to multivariate systems, which has the benefit of providing a simplified and equivalent set of polynomials. Groebner polynomials are ranked by complexity, where simpler ones tend to reveal helpful information such as internal conflicts and potential solutions. Second, we leverage domain-specific conflict-driven clause learning (CDCL) to alleviate the workload of the solver in terms of reasoning with non-linear constraints. For example, when performing the $\mathcal{Q}_1$ query at line 5 of Algorithm 2, the condition $S \wedge (\mathcal{E}_{cond} \equiv 0)$ might produce some unsatisfiable Groebner polynomial like $v_1 \times v_1 \equiv c$, where c is a quadratic nonresidue, meaning c has no "square roots" mod p. In this case, $E_{cond} \neq 0$ is recorded as a learned clause. Quadratic nonresidues are checked via Euler's criterion [37]. In addition, if a linear or quadratic equation is solvable, the disjunction of its solutions are recorded as learned clauses. Since we are dealing with a finite field of characteristic-p with a large prime p, solutions are efficiently derived via Tonelli–Shanks algorithm [38] The above information derived from the Groebner basis is not directly revealed from the inherent Groebner representation of SMT solvers like CVC5 [39] or Z3 [33], which is why we explicitly compute it.

E. Novelty

This section discusses the novelty of ConsCS compared to the existing works Picus [5] and Civer [6]. Our key contribution and main performance boost come from stage ⑤, the BPG reasoning engine. Both Picus and Civer rely on expensive SMT queries that encode the exact definition of circuit constrainedness, which is less effective on larger circuits due to computational complexity. To address this challenge, we develop a lightweight BPG reasoning engine that exhibits linear empirical complexity. Our approach substantially reduces reliance on expensive SMT queries, which are performance bottlenecks in existing works. For instance, *Num2BitsNeg(n=1)* is a circuit with 3 constraints, which is easily solved by Picus in 5.23s and by Civer in 0.090s. However, they both fail on the larger versions like *Num2BitsNeg(n=128)* and *Num2BitsNeg(n=256)* (with 132 and 260 constraints, respectively) even with 600s timeouts, due to the expensive SMT queries. In contrast, leveraging our BPG reasoning in stage ⑤, ConsCS quickly solves the *(n=128)* and *(n=256)* versions in 0.053s and 0.195s, respectively. This demonstrates that direct SMT queries become less feasible on larger circuits, whereas our BPG algorithm handles them efficiently. More such

examples of large circuits that both Picus and Civer fail to solve but ConsCS succeeds include *SigmaPlus@sigmaplus* (1018 constraints, solved in 4.063s), *T2@t2* (706 constraints, solved in 1.920s), *Sign@sign* (521 constraints, solved in 1.154s), etc. When BPG is disabled, all these circuits time out. We discuss more examples in Section IV-C4. Stage ④ and ⑥ are designed to facilitate stage ⑤ by simplifying problems and extracting variable-level information. In stage ④, while our IR1 is adopted from Picus, we extend the pre-processing with IR2 through IR5 to enhance analysis coverage towards a more complex domain. IR2 and IR5 are novel contributions introducing bijective replacements and assumption variables to better handle non-linear constraints. IR3 and IR4 extend Picus by covering both double-variable terms (e.g., $v_1 v_2$) and single signals (e.g., v_1), whereas Picus only addresses the latter. Such pre-processing stage for inferring constrainedness is new to Civer. In stage ⑥, we employ a novel SMT querying method focusing on the separability of non-linear constraints, facilitating the main reasoning engine in stage ⑤. This is a fundamentally different approach from existing works that rely on encoding the exact definition of circuit constrainedness to obtain final analysis results.

IV. Evaluation

In this section, we introduce the experiments that we conducted to evaluate ConsCS. The codes are available at https://github.com/jinan789/ConsCS. Specifically, we assess the performance of our proposed framework in terms of determining the constrainedness of R1CS circuits, as well as how it compares to existing works (§Section IV-B). Additionally, we conduct an ablation study and a coverage analysis to investigate the impact of each design component of ConsCS (§Section IV-C). Furthermore, we examine the effect of varying the hyperparameter *depth* in the BPG algorithm (§Section IV-A). We also present case studies (§Section IV-C4) and discuss the empirical complexity of the BPG algorithm (§Section IV-D).

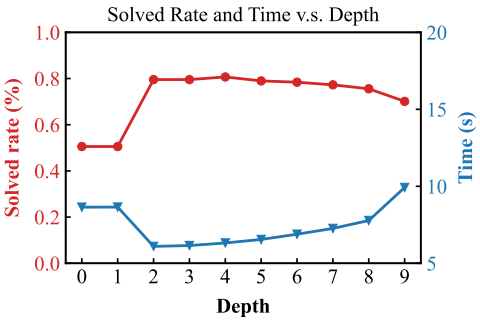

Fig. 3: The solved rate and time at different depths.

A. RQ1: How does the choice of different depths affect the effectiveness of the BPG algorithm?

1) Motivation: The tunable hyperparameter *depth*, introduced at Section III-C, plays a critical role in configuring the BPG algorithm. This RQ aims to show how varying the depth affects the effectiveness of the BPG algorithm, as well as to identify the optimal depth. It also explains how we configured the hyperparameters for the following research questions.

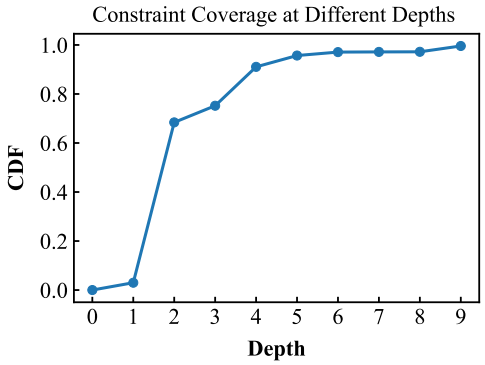

Fig. 4: The CDF plot of constraint coverage at different depths. For example, the point at depth 4 indicates that this depth covers 91.06% of the constraints in our benchmark.

2) Approach: We conduct two experiments. First, we run ConsCS at different depths from 0 to 9, and Fig. 3 shows the solved rate and solving time across different depths. Second, we examine the percentage of constraints covered by each depth via the Cumulative Distribution Function (CDF) shown in Fig. 4. It illustrates the coverage across varied depths. This CDF plot also provides empirical insights into the composition and complexity of the Circom R1CS constraints.

3) Results: Fig. 4 shows that at depth = 0, 0% of the constraints are covered. It effectively bypasses the BPG stage since it covers no constraints, achieving the lowest solved rate. At depth = 1, only 2.99% of constraints are covered, still with very low solved rate. Overall, these two depths exhibit the worst performance. At depth 2, it shows noticeable enhancement in both solved rate and time, covering 68.41% of the constraints, a substantial improvement from depth 1. This enhancement comes from less SMT queries; at depth = 1, BPG involves only 2.99% of all constraints and becomes basically useless, leaving heavy workloads to expensive SMT queries. However, at depth = 2, BPG covers 68.41% of constraints, leveraging its efficiency to largely reduce solving time. Beyond depth 2, while solving time increases, the solved rate peaks at depth 4. The gradually declining solved rate can be explained by more timeouts due to increased complexity of the analysis. The optimal tradeoff point is identified at depth 4, at which the solved rate is highest, and 91.06% of constraints are covered.

> Overview for RQ 1: Depth = 4 is identified as the optimal tradeoff point, achieving the highest solved rate and covers 91.06% of all the polynomial constraints.

B. RQ2: What is the performance of ConsCS? How does it compare to existing tools?

1) Motivation: We aim to evaluate the performance of our proposed framework, ConsCS, and compare it to state-of-the-art tools, specifically focusing on how it outperforms these existing methods.

2) Approach: We compare to two state-of-the-art tools: Civer [6] and Picus [5]. Civer has been integrated into the Circom compiler [14]. We adapt the benchmarks from Picus,

TABLE I: The performance metrics. $\mathcal{U}$ and $\mathcal{C}$ denote the `utils` and the `core` benchmark, respectively. Subscripts denote the sizes of the corresponding benchmark. "*# Var*" and "*# Cons*" denote the number of variables and constraints, respectively.

Metric	Category	$\mathcal{U}_{small}$	$\mathcal{U}_{medium}$	$\mathcal{U}_{large}$	$\mathcal{U}_{overall}$	$\mathcal{C}_{small}$	$\mathcal{C}_{medium}$	$\mathcal{C}_{large}$	$\mathcal{C}_{overall}$	**Overall**
# Var	In	5	98	145	35	27	39	146	54	47
	Out	2	2	57	9	10	78	36	30	22
	Witness	13	343	14,165	1,983	11	397	32,579	6,905	5,059
	Overall	19	443	14,367	2,027	48	513	32,761	6,989	5,129
# Cons	Linear	7	181	8,701	1,214	5	200	26,116	5,507	3,897
	Nonlinear	7	163	5,529	779	12	274	6,540	1,434	1,188
	Overall	14	344	14,230	1,993	17	474	32,657	6,941	5,086
Time	Picus	8.06s	15.68s	22.55s	10.96s	9.18s	20.50s	27.03s	15.39s	13.73s
	Civer	1.34s	6.12s	21.54s	4.67s	1.25s	5.51s	15.31s	5.12s	4.95s
	ConsCS	3.52s	7.52s	11.88s	5.15s	1.90s	6.90s	21.18s	7.02s	6.32s
Solved	Picus	87.76%	62.50%	33.33%	77.27%	82.54%	45.83%	13.04%	60.00%	66.48%
	Civer	83.67%	12.50%	0.00%	63.64%	84.13%	20.83%	4.35%	53.64%	57.39%
	ConsCS	**89.80%**	**75.00%**	**77.78%**	**86.36%**	**95.24%**	**70.83%**	**34.78%**	**77.27%**	**80.68%**

including the `utils` with 66 utility circuits, and the `core` with 110 circuits representing the most security-critical templates from Circomlib [16]. Circomlib is provided as part of the Circom compiler, representing the mostly commonly used features and is widely adopted in existing projects. It is the primary library for the Circom language, and has been adopted as the standard testing benchmarks in existing works like Picus and Civer. The circuits are categorized into sizes based on the number of constraints $|C|$: small ($|C| < 100$), medium ($100 \leq |C| < 1000$), and large ($1000 \leq |C|$). Table I presents the variable and constraint counts for these categories, showing an increase in both counts with circuit size. For example, in the `utils` benchmark, the variable count grows from 19 in small circuits ($\mathcal{U}_{small}$) to 14,367 in large ones ($\mathcal{U}_{large}$), indicating a substantial increase in complexity as circuit size scales. Our experimental setup involves an Intel Xeon Platinum 8375C CPU server with 128 GB RAM, running Ubuntu 22.04.4 LTS. We adopt the identical timeout (30 seconds per circuit) and the same set of benchmarks for all of the tools. We determined the optimal timeout value by searching for a threshold above which our compared tools are just timeouts. We tested different timeouts ranging from 5 to 600 seconds, and found 30 to be the ideal choice. The experiments are conducted sequentially without parallel processing. We evaluate the tools based on two metrics. First, *Time* is the average duration for solving a circuit. Second, *Solved Rate* is the percentage of circuits that were successfully solved, inferred as either *properly constrained* or *under-constrained*. The *overall* time is an arithmetic mean from a full run of all 176 circuits in the benchmark. The *overall* solved rate is the percentage of solved circuits out of all circuits.

3) Results: As shown in Table I, ConsCS demonstrates superior performance, achieving the highest solved rate among all compared tools, across circuits of all sizes. Specifically, ConsCS enhances the solved rate by 21.4% over Picus (i.e., $\frac{80.68-66.48}{66.48} \times 100\%$) and by 40.6% over Civer (i.e., $\frac{80.68-57.39}{57.39} \times 100\%$). Moreover, ConsCS and Civer are both more efficient than Picus, being approximately two to three times faster. With respect to Civer, ConsCS shows a small improvement, but not in all cases. Overall, it shows that, across a wide range of sizes of circuits, ConsCS is consistently highly effective in terms of solving the constrainedness of circuits compared to existing works.

> Overview for RQ 2: In terms of solved rate, ConsCS consistently outperforms both the state-of-the-art existing tools, across all benchmarks of different sizes. For efficiency, while ConsCS is slightly slower than Civer in some instances, it is consistently much faster than Picus, showcasing its effectiveness and efficiency.

TABLE II: The coverage analysis results.

Stage	Component	Percentage
④	IR 1: Single to Double	78.41%
	IR 2: Simple Solve	54.55%
	IR 3: Single Unknown	80.11%
	IR 4: Binary Expression	26.14%
	IR 5: Assumptions Extraction	32.95%
⑤	BPG: Nodes Exist	85.23%
	BPG: Variable Solved	33.52%
⑥	Assumption: Variable Solved	39.46%

C. RQ3: What is the impact and coverage of each major component within ConsCS?

1) Motivation: Our objective is to assess the influence and necessity of the major components within ConsCS. This involves evaluating the individual effectiveness and the overall coverage of different components in terms of the percentage of circuits in which they are shown to be effective. The components we evaluate are the Circuit Deduction, BPG

TABLE III: The ablation study results. A cross × indicates a removed stage. A checkmark ✓ indicates a present stage.

④	⑤	⑥	Solved (%)	Time (s)
✓	✓	✓	**80.68%**	**6.32s**
✓	✓	×	71.59%	3.12s
✓	×	✓	46.86%	10.28s
✓	×	×	42.61%	2.38s
×	✓	✓	14.77%	9.46s
×	✓	×	11.93%	2.40s
×	×	✓	6.29%	6.79s

Reasoning, as well as the Assumption-guided SMT Solving, which are respectively numbered stage ④, ⑤, and ⑥ in Fig. 2.

2) Approach: We characterize different components of ConsCS from two perspectives. First, an ablation study is conducted by removing each component to observe their effect on the framework's performance. The findings are presented in Table III. Second, we assess the coverage of each component's usage in solving the constrainedness of circuits as follows. For stage ④, we record the percentage of circuits where the corresponding rules were applied and successfully inferred new information. For stage ⑤, we report both the percentage of circuits with a non-trivial BPG graph at the "*BPG: Nodes Exist*" column, and the percentage of circuits where the BPG algorithm was utilized to solve new information at the "*BPG: Variable Solved*" column. For stage ⑥, we report the percentage of circuits that entered this stage and successfully utilized the assumptions to infer new information. Table II shows the results.

3) Results: The results of the ablation study (Table III) show that the presence of all three stages yields the highest solved rate (80.68%). Notably, the absence of stage ④ results in a significant decrease in the solved rate to around 10%, showing the essential role of circuit simplification and deduction. Furthermore, when only stages ④ and ⑤ are present, ConsCS already achieves a 71.59% solved rate and 3.12 solving time, which is more efficient and effective than all existing tools. Although Stage ④ alone solves 42.61% of circuits, these tend to be simpler ones that benefit from straightforward preprocessing, and more complex circuits tend to be addressed by stage ⑤ and ⑥. In addition, when BPG is disabled, the performance of ConsCS drops significantly by 41.9% (i.e., $\frac{80.68-46.86}{80.68} \times 100\%$), highlighting the BPG algorithm's contribution. Table II quantifies each component's coverage. Over half (54.55%) of the circuits benefit from our newly proposed circuit simplification and reduction IR (Equation (5)), showing its wide applicability. In addition, BPG nodes were identified in the vast majority (85.23%) of circuits, contributing to solving the constrainedness of variables in about a third (33.52%) of all circuits. For stage ⑥, by leveraging assumption tuples, 39.47% of the circuits that entered this stage solved the constrainedness of at least one variable, indicating its wide application.

4) Case studies: Below we discuss three case studies, one for each of our three main stages, to illustrate the effectiveness of each component of ConsCS. All selected examples are those that none of the existing detection tools could solve.

Stage 4: *T1@t1@circomlib.circom* This circuit is from the Circomlib implementation of the SHA-256 hash. In this example, the *simple solve* deduction rule introduced in Section III-B contributes to significant reduction of circuit size. Specifically, 640 intermediate variables are removed, reducing 82.9% of the polynomial constraints to trivial constraints, i.e., $0 \equiv 0 \pmod p$, which are always satisfied. This reduces the circuit's complexity by nearly tenfold, enhancing ConsCS's ability to efficiently solve the circuit.

Stage 5: *Num2BitsNeg@bitify@circomlib_256.circom* In this circuit, the BPG graph contains 538 nodes and 1,037 edges. Our BPG algorithm took only 0.0051 seconds to infer 257 variables to be binary and 1 variable to be constrained. In contrast, the result reported by Picus for our inferred constrained variable is unknown. Furthermore, our ablation study from Section IV-C shows that when BPG is disabled, ConsCS times out without finding a solution, showing the significance of the BPG algorithm.

Stage 6: *BabyAdd@babyjub@circomlib.circom* Both Picus and Civer struggled with inferring information from the constraint $x_1 + 168696 * x_1x_{10} \equiv x_7 + x_8 \pmod p$. The difficulty stems from its non-linearity. Notably, our assumption-guided SMT query effectively infers the constrainedness of the circuit via $\mathcal{Q}_1 \equiv$ `UNSAT` and $\mathcal{Q}_2 \equiv$ `SAT` query results. On the other hand, *Pedersen@pedersen.circom* is an example where $\mathcal{Q}_1 \equiv$ `SAT` and $\mathcal{Q}_2 \equiv$ `SAT` results were leveraged to infer under-constrainedness of the circuit, which no existing works could solve, either. Our ablation study shows that when the assumption-guided SMT solving module is removed, both of these circuits time out and cannot be solved.

5) Cases where ConsCS fails: There are also cases where ConsCS fails. We classify them into 2 main categories. First, handling large circuits such as *Sha256_2@sha256_2* is a challenge, where ConsCS often times out during the parsing stage, before even entering analysis. As shown in Table I, this issue is not unique to our approach, but is a common challenge among SOTA tools, which also exhibit decreased performance with large circuits. Second, ConsCS struggles with analyzing circuits from the Pedersen hash family. Excluding these circuits from the benchmark, our solved rate would increase to 91.36%. The Pedersen hash maps a sequence of bits to a point on the Baby-Jubjub elliptic curve to produce a hash value [40], involving complex cryptographic operations that are challenging to reason with. Notably, existing tools also struggle with solving such circuits, suggesting that further efforts is needed for future works.

> Overview for RQ 3: Each of our proposed novel component contributes to identifying the constrainedness of Circom R1CS circuits, across various sizes of circuits. Furthermore, the variant of ConsCS with stage ⑥ removed outperforms both Picus and Civer in both solved rate and solving time.

Size	# Var	Nodes/Var	Edges/Var	Time/Var
small	36	2.70	18.80	0.24 ms
medium	496	3.10	23.78	0.60 ms
large	27587	2.68	20.61	0.67 ms

TABLE IV: Empirical complexity of the BPG algorithm.

D. Empirical complexity of the BPG algorithm

As discussed in Section III-C, the theoretical worst-case complexity of the BPG algorithm is $O(n^2)$, where n is the number of variables in the circuit, assuming maximum connectivity between nodes. However, empirical data shown in Table IV display a different trend. Specifically, it can be observed that despite a significant increase in the number of variables from small to large circuits, the average number of nodes per variable remains steady at around 3. Similarly, the number is around 20 for edges per variable. This indicates that the number of nodes and edges both empirically grow with a linear trend, instead of quadratic, suggesting sparse graphs. The time efficiency of the BPG phase further supports this linear trend, with time per variable ranging from 0.70 ms to 1.09 ms across different sizes, which is much lower than expected for a worst-case quadratic complexity. Additionally, on average, the BPG phase accounts for only 3.82% of the total solving time per circuit (i.e., $\frac{0.2413}{6.32} \times 100\%$), highlighting its efficiency.

V. Discussion & Limitations

ConsCS does not generate false positives during evaluation. When a circuit is reported as under-constrained, ConsCS provides a counter-example that is verified against all circuit constraints, thus proving under-constrainedness by definition. The rationale is analogous (although not identical) to the classical NP-complete circuit satisfiability problem, where finding a satisfying assignment is non-trivial, which ConsCS provides, but verifying one is easy. The primary limitation of ConsCS is that it fails to solve certain types of circuits, resulting in unknown or timeout results. In particular, as discussed in Section IV-C4, ConsCS struggles with extremely large circuits and those that incorporate complex cryptographic operations like the Pedersen hash. These issues highlight areas for potential enhancement in future research.

VI. Related works

A. Construction of arithmetic circuits

Several domain-specific languages (DSLs) for the construction of arithmetic circuits for zk-SNARK have been proposed. Some such languages include Circom [8], CODA [4], Leo [41], Lurk [42], and Zokrates [43]. On the other hand, Cairo [44] is a framework that supports STARK, a different approach for zero-knowledge proofs. These tools provide support for verifiable computation via zero-knowledge proofs on the blockchain. At a high level, they offer a convenient interface for users to specify their problems, along with an associated compiler that converts programs into arithmetic circuits, which are then used to generate zero-knowledge proofs. Additionally, beyond R1CS circuits, Pinocchio [45], [46] is a system that targets verifiable computation with support for Quadratic Arithmetic Programs (QAP) and Quadratic Span Programs (QSP). Circ [47] proposes a more generalized approach by building a shared compiler infrastructure for compiling their C-style programs to constraint circuit representations; Otti [48] is an example compiler that extends Circ to support the compilation of numerical optimization problems.

B. Analysis of arithmetic circuits

Several works for the analysis of arithmetic circuits have proposed. Picus [5] and Civer [6] both address the issue of under-constrained circuits in the Circom framework. They both rely on inference rules to infer range information of wires, followed by SMT queries to determine the constrainedness of circuits. In addition, the Circom compiler [7] provides native support for identifying potential lack of constraints via the *–inspect* tag, though it is limited to reporting signals that do not appear in any constraints, lacking deeper logical reasoning. Circomspect [49] is a patten-matching based approach for identifying several types of bugs, but suffers from the problem of false positives and false negatives. Other than Circom, further analysis have explored the under-constrainedness of Halo2 circuits [50]. Additionally, several works have investigated common vulnerabilities in Circom and zk-SNARK protocols [2], [3], providing detailed analysis and datasets for various vulnerability issues related to arithmetic circuits. Beyond vulnerability analysis, several works have been proposed to facilitate a more convenient understanding and analysis of circuits. These include distilling essential constraints out of a constraint system [15] and generating standardized formats for different R1CS instances with identical semantics [51].

VII. Conclusion

In this paper, we introduced ConsCS, a novel framework aimed at enhancing the verification of Circom for blockchain security. ConsCS introduces novel inference rules, the Binary Property Graph (BPG) as a reasoning engine, and an assumption-guided SMT solving approach that achieves a 48.84% increase in success rate compared to prior methods. Our experimental results demonstrate that ConsCS achieves the highest effectivenss among existing tools, enhancing the solved rate of existing works from around 50-60% to above 80%. In addition, we also demonstrate that our proposed methods are widely applicable to a wide range of circuits across various sizes, with the BPG algorithm being a highly efficient and effective light-weight reasoning engine.

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 RGC Project (PolyU15231223).

REFERENCES

[1] "Zero-knowledge rollups," https://ethereum.org/en/developers/docs/scaling/zk-rollups/, Accessed: 2024-07-11.

[2] 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. [Online]. Available: https://arxiv.org/abs/2402.15293

[3] 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, https://eprint.iacr.org/2023/190. [Online]. Available: https://eprint.iacr.org/2023/190

[4] 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," Cryptology ePrint Archive, Paper 2023/547, 2023, https://eprint.iacr.org/2023/547. [Online]. Available: https://eprint.iacr.org/2023/547

[5] 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

[6] 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

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

[8] 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.

[9] "Bigmod incorrectly omits range checks on the remainder," https://github.com/0xPARC/circom-ecdsa/pull/10, Accessed: 2024-07-26.

[10] "Coreverifypubkeyg1 does not enforce lt checks on input." https://github.com/yi-sun/circom-pairing/pull/21/commits/c686f0011f8d18e0c11bd87e0a109e9478eb9e61, Accessed: 2024-07-26.

[11] "Fix maci 1.0 processmessages circuit to prevent message censorship by the coordinator," https://github.com/privacy-scaling-explorations/maci/issues/320, Accessed: 2024-07-26.

[12] "Disclosure of recent vulnerabilities," https://hackmd.io/@aztec-network/disclosure-of-recent-vulnerabilities, Accessed: 2024-07-26.

[13] "Tornado.cash got hacked. by us." https://tornado-cash.medium.com/tornado-cash-got-hacked-by-us-b1e012a3c9a8, Accessed: 2024-07-26.

[14] "Circom civer," https://github.com/costa-group/circom_civer, Accessed: 2024-07-11.

[15] E. Albert, M. Bellés-Muñoz, M. Isabel, C. Rodriguez, and A. Rubio, *Distilling Constraints in Zero-Knowledge Protocols*, 08 2022, pp. 430–443.

[16] "Circomlib," https://github.com/iden3/circomlib, Accessed: 2024-07-09.

[17] P. L. Montgomery, "Speeding the pollard and elliptic curve methods of factorization," *Mathematics of Computation*, vol. 48, pp. 243–264, 1987. [Online]. Available: https://api.semanticscholar.org/CorpusID:4262792

[18] J. Thaler, "Proofs, arguments, and zero-knowledge," https://people.cs.georgetown.edu/jthaler/ProofsArgsAndZK.html, 2022, accessed: 2023-11-09.

[19] S. Ames, C. Hazay, Y. Ishai, and M. Venkitasubramaniam, "Ligero: Lightweight sublinear arguments without a trusted setup," Cryptology ePrint Archive, Paper 2022/1608, 2022, https://eprint.iacr.org/2022/1608. [Online]. Available: https://eprint.iacr.org/2022/1608

[20] A. Chiesa, Y. Hu, M. Maller, P. Mishra, P. Vesely, and N. Ward, "Marlin: Preprocessing zkSNARKs with universal and updatable SRS," Cryptology ePrint Archive, Paper 2019/1047, 2019, https://eprint.iacr.org/2019/1047. [Online]. Available: https://eprint.iacr.org/2019/1047

[21] J. Groth, "On the size of pairing-based non-interactive arguments," Cryptology ePrint Archive, Paper 2016/260, 2016, https://eprint.iacr.org/2016/260. [Online]. Available: https://eprint.iacr.org/2016/260

[22] 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

[23] T. Xie, J. Zhang, Y. Zhang, C. Papamanthou, and D. Song, "Libra: Succinct zero-knowledge proofs with optimal prover computation," Cryptology ePrint Archive, Paper 2019/317, 2019, https://eprint.iacr.org/2019/317. [Online]. Available: https://eprint.iacr.org/2019/317

[24] T. Liu, T. Xie, J. Zhang, D. Song, and Y. Zhang, "Pianist: Scalable zkRollups via fully distributed zero-knowledge proofs," Cryptology ePrint Archive, Paper 2023/1271, 2023, https://eprint.iacr.org/2023/1271. [Online]. Available: https://eprint.iacr.org/2023/1271

[25] "zksync overview," https://docs.lite.zksync.io/userdocs/intro/, Accessed: 2024-07-12.

[26] "Polygon overview," https://docs.polygon.technology/zkEVM/overview/, Accessed: 2024-07-12.

[27] "Scroll overview," https://docs.scroll.io/en/getting-started/overview/, Accessed: 2024-07-12.

[28] 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.

[29] 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.

[30] 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.

[31] "snarkjs," https://github.com/iden3/snarkjs, Accessed: 2024-07-16.

[32] "Signal tags," https://docs.circom.io/circom-language/tags/, Accessed: 2024-07-17.

[33] "Z3," https://github.com/Z3Prover/z3, Accessed: 2024-07-18.

[34] L. De Moura and N. Bjørner, "Z3: an efficient smt solver," in *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, ser. TACAS'08/ETAPS'08. Berlin, Heidelberg: Springer-Verlag, 2008, p. 337–340.

[35] "Programming z3," https://theory.stanford.edu/~nikolaj/programmingz3.html, Accessed: 2024-07-18.

[36] "Sympy," https://www.sympy.org/en/index.html, Accessed: 2024-07-18.

[37] E. Lehmer, "On euler's criterion," *Journal of the Australian Mathematical Society*, vol. 1, no. 1, p. 64–70, 1959.

[38] V. Diekert, M. Kufleitner, G. Rosenberger, and U. Hertrampf, *Discrete Algebraic Methods: Arithmetic, Cryptography, Automata and Groups*. De Gruyter, 2016.

[39] A. Ozdemir, "Cvc5-ff," https://github.com/alex-ozdemir/CVC4/tree/ff, 2022, accessed: 2024-11-15.

[40] J. Baylina and M. Bellés, "4-bit window pedersen hash on the baby jubjub elliptic curve," https://iden3-docs.readthedocs.io/en/latest/_downloads/4b929e0f96aef77b75bb5cfc0f832151/Pedersen-Hash.pdf, iden3 and Universitat Pompeu Fabra, 2023, accessed: 2024-07-26.

[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] N. Amin, J. Burnham, F. Garillot, R. Gennaro, C. Künzang, D. Rogozin, and C. Wong, "LURK: Lambda, the ultimate recursive knowledge," Cryptology ePrint Archive, Paper 2023/369, 2023, https://eprint.iacr.org/2023/369. [Online]. Available: https://eprint.iacr.org/2023/369

[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] 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

[45] B. Parno, J. Howell, C. Gentry, and M. Raykova, "Pinocchio: Nearly practical verifiable computation," in *2013 IEEE Symposium on Security and Privacy*, 2013, pp. 238–252.

[46] C. Fournet, C. Keller, and V. Laporte, "A certified compiler for verifiable computing," in *2016 IEEE 29th Computer Security Foundations Symposium (CSF)*, 2016, pp. 268–280.

[47] A. Ozdemir, F. Brown, and R. S. Wahby, "CirC: Compiler infrastructure for proof systems, software verification, and more," Cryptology ePrint Archive, Paper 2020/1586, 2020, https://eprint.iacr.org/2020/1586. [Online]. Available: https://eprint.iacr.org/2020/1586

[48] S. Angel, A. J. Blumberg, E. Ioannidis, and J. Woods, "Efficient representation of numerical optimization problems for SNARKs," in *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 4273–4290. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/angel

[49] "It pays to be circomspect," https://blog.trailofbits.com/2022/09/15/it-pays-to-be-circomspect/, Accessed: 2024-07-25.

[50] 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, https://eprint.iacr.org/2023/1051. [Online]. Available: https://eprint.iacr.org/2023/1051

[51] C. Shi, R. Liu, H. Chen, G. Li, and S. Gao, "Rna: R1cs normalization algorithm based on data flow graphs for zero-knowledge proofs," *Form. Asp. Comput.*, may 2024, just Accepted. [Online]. Available: https://doi.org/10.1145/3665339