Data-Flow-Based Normalization Generation Algorithm of R1CS for Zero-Knowledge Proof

Chenhao Shi, Hao Chen, Ruibang Liu, and Guoqiang Li[0000−0001−9005−7112]

School of Software, Shanghai Jiao Tong University, 200240, Shanghai
{ li.g Dennis.Chen undefeated}@sjtu.edu.cn

Abstract. In zero-knowledge proofs, the prover needs to construct a system that contains many constraints and prove that the system satisfies these constraints. Circom and R1CS are important components of the underlying toolchain that allow users to easily create, optimize, and verify zero-knowledge proof systems. Users can define various constraints easily with Circom and convert them to the R1CS format using a compiler, and then convert them to a verifiable form. However, due to the flexibility of the R1CS representation, R1CS compiled from Circom programs with the same semantics often differ significantly. This also makes it difficult to verify the correctness and scalability of R1CS. In this paper, we propose a data flow-based R1CS paradigm generation algorithm. The algorithm converts R1CS to a data flow graph and transforms equivalent R1CS into the same unique paradigm after abstract processing and weight calculation. Our simulation studies indicate that our algorithm can correctly generate paradigms in scenarios where equivalent R1CS are commonly produced.

Keywords: Zero-knowledge proof · R1CS · Data flow graph.

1 Introduction

Zero-knowledge proof is increasingly recognized for its importance in modern society, as more and more cryptographic communities seek to address some of the blockchain's biggest challenges: privacy and scalability. The heightened emphasis on information privacy and security, from both user and developer perspectives, has led to a greater appreciation for the privacy advantages offered by zero-knowledge proofs. As decentralized finance (DeFi) usage continues to grow, zero-knowledge applications that offer scalability and privacy advantages will have more opportunities to increase industry-wide adoption. However, not all computational problems can be directly addressed using zero-knowledge proofs. Instead, we must transform the problem into the correct form of computation, known as a "Quadratic Arithmetic Program (QAP)." In the specific process of a first-order zero-knowledge proof, we first convert the problem into Circom language, then into R1CS constraints, and finally from constraints to the QAP form.

However, the conversion from Circom to R1CS constraints in the underlying toolchain of zero-knowledge proofs faces many limitations, with the primary

issue being poor mergeability of R1CS. When merging A and B, the resulting R1CS has no formal relationship with the independently generated R1CS of A and B. This limitation is related to the inherent expressive power constraint of R1CS, where the program can generate multiple equivalent R1CS constraints. Therefore, it is necessary to propose a canonical form for R1CS constraints to facilitate the determination of equivalence and correctness for different R1CS constraints. This proposal would greatly benefit us in verifying program equivalence and correctness, including further research into the mergeability of R1CS.

This paper proposes a data flow-based algorithm for generating normalization of R1CS, which enables the conversion of different R1CS constraints into a unique normal form, facilitating the determination of equivalence and correctness. In this algorithm, we first transform R1CS into a data flow graph structure resembling an expression tree. We then segment and abstract the data flow graph, eliminating differences between equivalent R1CS constraints that may arise from the generation process. Next, we propose a set of sorting rules to sort the constraints and variables within R1CS, ultimately generating a unique normal form for equivalent R1CS.

In addition, based on the constraint generation logic of mainstream compilers and the expressiveness of R1CS, we classify and summarize the reasons and characteristics of the different equivalent R1CS generated. Moreover, based on the identified reasons for producing equivalent R1CS, we create a relatively complete benchmark. Currently, our proposed algorithm can pass all test cases in the benchmark, meaning that equivalent R1CS can be converted into a unique and identical canonical form under various circumstances.

This work contributes to the field of R1CS optimization by providing a novel algorithm for generating canonical forms of equivalent R1CS constraints. Our algorithm improves upon existing methods by eliminating unnecessary redundancy and normalizing representation, thus facilitating the analysis of equivalence and correctness. Furthermore, our benchmark provides a comprehensive evaluation of the proposed algorithm, demonstrating its effectiveness and practicality.

The rest of this paper is organized as follows. A brief background review of zero-knowledge proof and related components in its underlying toolchain is presented in the next section. Section 3 presents the process of the proposed algorithm in this paper. The detailed logic of the key steps, along with their implementation in code, is elucidated in Section 4 using a technical exposition. The specific categories of benchmarks and their corresponding experimental results can be found in Section 5. Finally, Section 6 and Section 7 summarize the related work and conclusions of the present study correspondingly.

2 Background

2.1 Zk-SNARKs

Zk-SNARKs is a specific type of zero-knowledge proof, with a working principle that can be simplified into several steps. First, when a user needs to verify

their information, they convert it into a mathematical problem, which is called computation. Computation can be implemented using any Turing-complete programming language, such as C, Python, and Solidity (used for Ethereum smart contract programming) because the zk-SNARKs algorithm does not depend on any specific programming language.

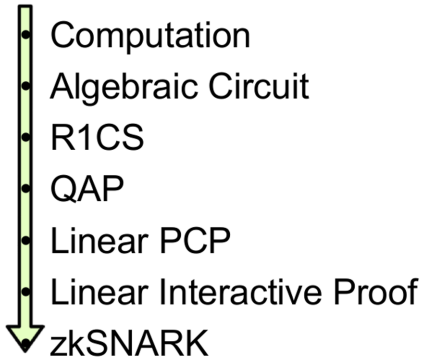

Fig. 1. The pipeline diagram of zk-SNARKs.

Next, the result of the computation is usually transformed into an arithmetic circuit, which is a data structure representing the calculation process. An arithmetic circuit consists of multiple gates, with each gate performing a basic arithmetic operation such as addition or multiplication. The entire arithmetic circuit can be used to generate a verifiable arithmetic circuit (R1CS), which takes the form of a constraint-based formula system (rank-1 constraint system) expressing the constraints of the arithmetic circuit. A verifiable arithmetic circuit is one of the inputs of the zk-SNARKs algorithm.

The next stage is the Quadratic Arithmetic Program (QAP), where the verifiable arithmetic circuit is further converted into QAP format. QAP is a formula system in which polynomials represent the behavior of the arithmetic circuit. QAP makes the implementation of the zk-SNARKs algorithm more efficient while also enhancing security.

Finally, there is the zk-SNARK stage, where the verifiable arithmetic circuit and QAP are used to generate proof.

zk-SNARKs are powerful privacy protection tools that can be utilized in digital payments, blockchain technology, and other fields. They can verify the authenticity of information while protecting the user's privacy. Although the working principle of zk-SNARKs is relatively complex, this technology has been widely applied, bringing higher security and privacy protection to the digital world.

2.2 Circom

Circom is one of the most important programming languages designed for building zero-knowledge applications. It provides a simple, declarative way to define constraint systems and generates corresponding R1CS (Rank-1 Constraint Systems) constraint systems. In this process, developers can use Circom to describe different constraints such as linear equations and inequalities. Circom is a domain-specific language (DSL) specifically designed for building zero-knowledge proof systems. DSL programming languages targeted at specific domains are typically more expressive and user-friendly than general-purpose programming languages. By providing a set of high-level abstract concepts, Circom allows developers to focus more on the algorithm itself without worrying too much about low-level implementation details.

The core idea of Circom is to represent an arithmetic circuit as a constraint system, where inputs, outputs, and computation processes are all represented as linear equations and inequalities. This approach allows developers to define arbitrarily complex computation processes and generate the corresponding R1CS constraint system. This constraint system can be used to implement various privacy-preserving protocols and zero-knowledge proof techniques, including digital asset transactions, cryptographic signatures, anonymous voting, and more.

A rich set of syntax and semantics was provided in Circom to help developers quickly define constraint systems. For example, it supports arithmetic operators (addition, subtraction, multiplication, division) and logical operators (AND, OR, NOT), which can rapidly construct complex computation processes. Additionally, Circom provides a standard library of functions, such as hash, signature, and Merkle tree, that can be used to build higher-level applications.

Circom can also be integrated with other tools such as SnarkJS and ZoKrates to build more complex zero-knowledge proof systems. These tools can compile Circom code into executable JavaScript code and provide a set of APIs for verifying proofs and generating evidence.

Circom is a robust programming language that facilitates the creation of sophisticated arithmetic circuits and R1CS constraint systems by developers. Its comprehensive range of features enables efficient and streamlined development of complex computational structures.It provides a simple, declarative way to define constraint systems, allowing developers to quickly implement various privacy-preserving protocols and zero-knowledge proof techniques. Circom has been widely used in cryptography, blockchain, and other security applications, providing critical support for protecting user privacy and data security.

2.3 R1CS

R1CS describes a set of constraint-based computation rules that can be verified using a set of public parameters and a private input sequence. In this model, computations are represented as a set of constraint conditions, namely linear equations and inequalities. Each equation has its own set of coefficients, while

each variable represents an input or output value. These equations and inequalities overall describe the limiting conditions of the computation, meaning that satisfying these conditions implies that the corresponding output result can be correctly calculated for the given input sequence.

An R1CS is a sequence of groups of three vectors (a, b, c), and the solution to an R1CS is a vector s, where s must satisfy the equation

$$(\vec{s} \cdot \vec{a}) * (\vec{s} \cdot \vec{b}) = \vec{s} \cdot \vec{c} \tag{1}$$

For example, a satisfied R1CS is shown in Fig.2:

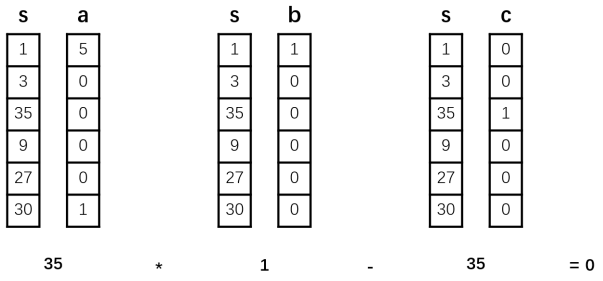

Fig. 2. A satisfied R1CS.

This constitutes a first-order constraint, which corresponds to a multiplication gate in the circuit. If we combine all constraints, we obtain a first-order constraint system.

Fundamentally, the R1CS constraint system employs the technique of generalized linear programming to solve problems, where the ultimate goal is to find a set of variable values that satisfy all constraints and the computed result matches the expected output. In this process, the prover needs to provide a set of responses so that the verifier can verify their claimed correctness.

Overall, R1CS is a powerful computational model that has been widely used in many practical applications. As an important component of verifiable algorithm design, it greatly expands our understanding of computer science and cryptography and provides critical support for various privacy protection measures.

2.4 Weighted Pagerank Algorithm

In this paper, we adopt the weighted PageRank algorithm to compute the weight of each node in the data flow graph [1].

Pagerank algorithm is a method used for computing the ranking of web pages in search engine results. It was originally proposed by Larry Page and Sergey

Brin, co-founders of Google, in 1998 and has since become one of the most important algorithms in the field of search engines.

The algorithm analyzes web pages on the internet to determine their weight values and uses these values to rank search results. The core idea behind Pagerank is that the weight of a web page depends on the number and quality of other web pages linking to it.

The main steps of the Pagerank algorithm are as follows:

1. Building the graph structure: Firstly, the web pages and links on the internet need to be converted into a graph structure. In this structure, each web page corresponds to a node and each link to a directed edge pointing to the linked web page.
2. Computing the initial scores of each page: In Pagerank, the initial score of each page is set to 1. This means that initially, each node has an equal score.

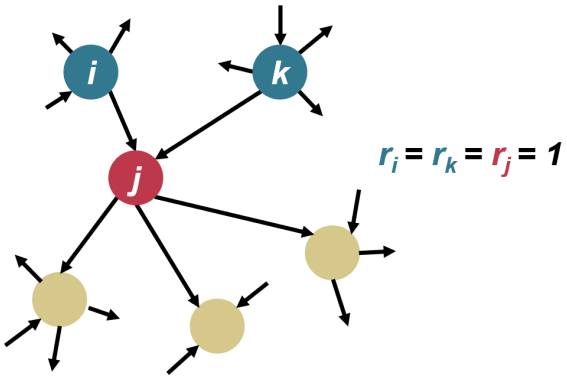

Fig. 3. The initial state of Pagerank algorithm.

3. Iteratively computing the scores of each page: Each node's score is iteratively computed based on its incoming links and averaged onto its outgoing links at each iteration.
4. Considering the number and quality of links: In addition to the relationships between nodes, Pagerank considers the number and quality of links pointing to a web page. Links from high-quality websites may carry more value than those from low-quality sites. Therefore, when computing scores, the algorithm weights links according to their number and quality.
5. Iterating until convergence: When the score of a node stabilizes, the algorithm stops iterating. This indicates that the final scores of all nodes have been determined and can be used to rank search results.

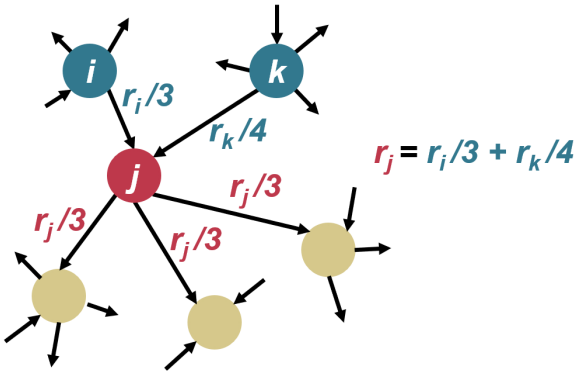

Fig. 4. The interation of Pagerank algorithm.

The Weighted PageRank algorithm differs from the standard PageRank algorithm in that it incorporates the weight of each link as a factor, resulting in a more precise evaluation of a webpage's importance. Considering the importance of pages, the original PageRank formula is modified as

$$PR(u) = (1-d) + d \sum_{v \in B(u)} PR(v) W^{in}_{(v,u)} W^{out}_{(v,u)} \tag{2}$$

In this equation, $W^{in}_{(v,u)}$ and $W^{out}_{(v,u)}$ are the weight of $link(v,u)$ calculated based on the number of inlinks of page u and the number of inlinks of all reference pages of page v.

In this paper, we aim to use this algorithm to obtain more accurate weight values for each node in the data flow graph.

3 Overview

In this section, we will introduce the procedure of the entire algorithm through the process of converting two equivalent R1CS constraint sets in the paradigm generation algorithm. We refer to these two constraint sets as Constraint Set A and Constraint Set B, respectively.

Constraint Set A:

$$A = \begin{pmatrix} 0\,1\,0\,0\,0\,0 \\ 0\,0\,0\,1\,0\,0 \\ 0\,1\,0\,0\,1\,0 \\ 5\,0\,0\,0\,0\,1 \end{pmatrix} B = \begin{pmatrix} 0\,1\,0\,0\,0\,0 \\ 0\,1\,0\,0\,0\,0 \\ 1\,0\,0\,0\,0\,0 \\ 1\,0\,0\,0\,0\,0 \end{pmatrix} C = \begin{pmatrix} 0\,0\,0\,1\,0\,0 \\ 0\,0\,0\,0\,1\,0 \\ 0\,0\,0\,0\,0\,1 \\ 0\,0\,1\,0\,0\,0 \end{pmatrix}$$

Constraint Set B:

$$A = \begin{pmatrix} 0\,1\,0\,0 \\ 0\,1\,0\,0 \end{pmatrix} B = \begin{pmatrix} 0\,1\,0\,0 \\ 0\,0\,0\,1 \end{pmatrix} C = \begin{pmatrix} 0 & 0 & 0\,1 \\ -5 & -1 & 1\,0 \end{pmatrix}$$

The Constraint Set B is generated by merging the last three constraints of Constraint Set A and performing some variable order swapping.

3.1 Construction of RNode Graph

The generation of the RNode Graph involves three main stages:

1. Transform each constraint into an equation in the form of $a * b = c$, as required by the R1CS constraints.
2. Convert each constraint in the original R1CS constraint set into such an equation.
3. Organize the resulting equations, which contain common sub-expressions, into a DAG-structured expression tree.

The structure of the RNode Graph generated by two sets of constraints is shown in Figure 5.

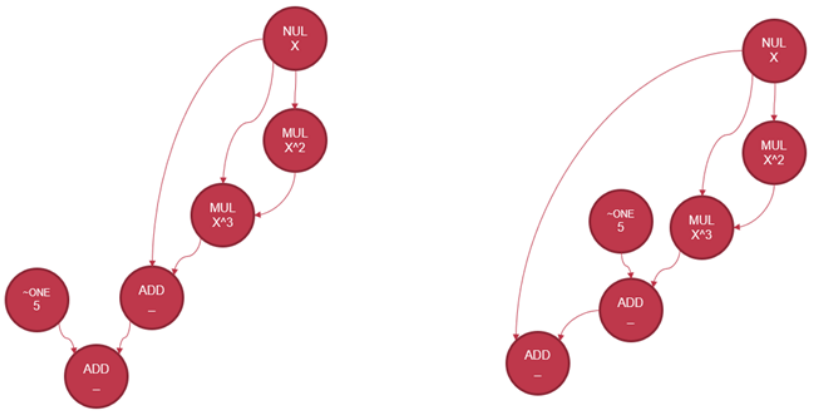

Fig. 5. The structure of the RNode Graph generated by two sets of constraints

The RNode is a data structure used to store information about the variables in an R1CS during the construction of the RNode Graph. Unlike typical nodes in an expression tree, each RNode stores information about both the variable and operator involved in a given operation, allowing each operator's output to be considered an intermediate variable and making it more closely aligned with the properties of an R1CS constraint set.

The second constraint in constraint set B is actually a combination of the last three constraints in constraint set A. In practice, the merging and splitting of constraints is one of the reasons why it is difficult to determine the equivalence of R1CS constraint sets. However, upon examining the two generated RNode Graphs, it is observed that this merging does not result in any significant differences. This is because when constraints are merged, one variable is subtracted from the original constraint set, for example

$$\begin{pmatrix} 1\ 1\ 0\ 0 \\ 0\ 1\ 1\ 0 \end{pmatrix} \cdot \begin{pmatrix} 1\ 0\ 0\ 0 \\ 1\ 0\ 0\ 0 \end{pmatrix} = \begin{pmatrix} 0\ 0\ 1\ 0 \\ 0\ 0\ 0\ 1 \end{pmatrix}$$

$$\downarrow$$

$$\begin{pmatrix} 1\ 2\ 0\ 0 \end{pmatrix} \cdot \begin{pmatrix} 1\ 0\ 0\ 0 \end{pmatrix} = \begin{pmatrix} 0\ 0\ 0\ 1 \end{pmatrix}$$

$$\downarrow$$

$$\begin{pmatrix} 1\ 2\ 0 \end{pmatrix} \cdot \begin{pmatrix} 1\ 0\ 0 \end{pmatrix} = \begin{pmatrix} 0\ 0\ 1 \end{pmatrix}$$

However, the variable that is subtracted will be added back into the RNode Graph as an intermediate node in the sum-product expression during the construction of the RNode Graph. The reverse is also true.

Upon examining the two RNode Graphs shown in Figure 1, the main difference observed is in the order of addition in constructing the sum-product expression $x^3 + x + 5 = out$. This is due to the different variable ordering in the constraint set, and we do not have sufficient information at this stage to determine the sequential execution order. In constraint set A, the expression is $(x^3 + x) + 5 = out$, while in constraint set B, it is $(x^3 + 5) + x = out$. Therefore, for such cases, further abstraction of the RNode Graph is required to eliminate this difference.

3.2 Tile Selection

Here, we categorize tiles into three types:

1. Quadratic: Tiles with the form $x * y = z$, where x, y, and z are variables.
2. MulLinear: Tiles whose root is obtained by multiplying its two parents, with at least one of the parents being a constant, such as $(5 * x) * 7 = z$.
3. AddLinear: Tiles whose root is obtained by adding its two parents, such as $x^3 + 5 + x = z$.

While AddLinear and MulLinear tiles are both generated by linear constraints and are essentially linear tiles, their logical processing differs significantly. Hence, we discuss them as two separate types. During tile selection, we divide the data flow graph from the previous step into these three types for various considerations:

1. We temporarily put aside the constraint merging step until we obtain more information about the tree in subsequent steps.

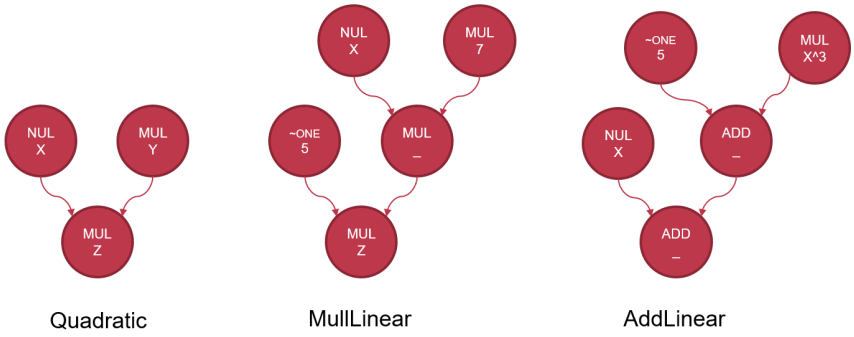

Fig. 6. The schematic diagrams for these three types of tiles.

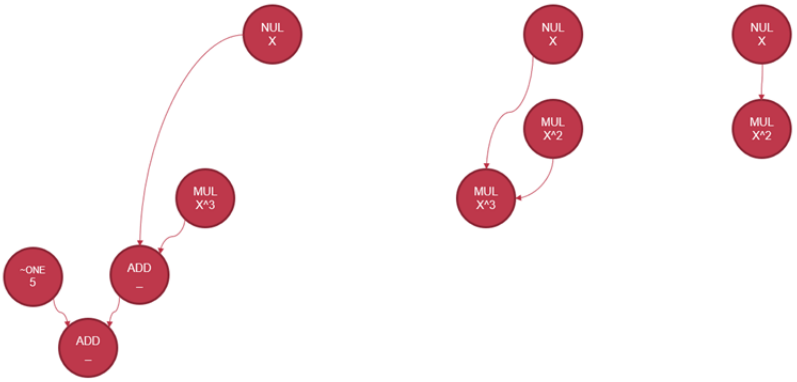

Fig. 7. Tiles selected from the RNode graph constructed from constraint set A.

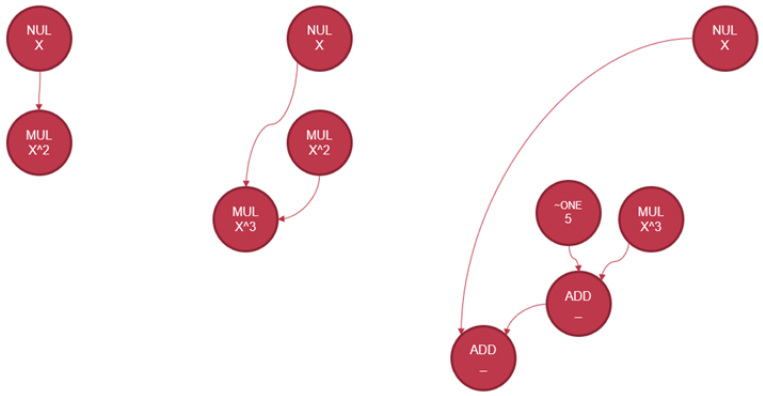

Fig. 8. Tiles selected from the RNode graph constructed from constraint set B.

2. If there is a need to generate merged formulas later, it can be achieved simply by applying a fixed algorithm to the unmerged formulas.
3. The implementation of the tile selection algorithm is relatively simple.

As we mentioned earlier, the difference between data flow graphs generated by equivalent R1CS constraint systems lies in the order of node additions when processing linear tiles. Upon examining the tiles selected from the RNode Graph generated by constraint sets A and B, it is apparent that their primary distinction lies in the specific internal structure of their linear tiles. However, if we consider the nodes added within a linear constraint as a set, they are actually equivalent. That is to say, the different addition order only means that the traversal order of the nodes is different. Therefore, if we regard the selected linear tiles as the products of the selected nodes and their respective coefficients, then the selected sets of linear tiles from equivalent R1CS constraint systems are the same. In other words, there is no difference between the selected sets of tiles for equivalent R1CS constraint systems.

Based on the selected tiles, we further abstract the data flow graph to eliminate the differences between various equivalent R1CS constraint systems. Specifically, we abstract linear tiles using a new node in the data flow graph. By doing so, we mask the difference in addition order within linear tiles in the RNode Graph and transform the relationship between external nodes and specific nodes within the linear tile into a relationship between external nodes and the tile to which the specific node belongs. In this abstracted data flow graph, the types of edges are as follows:

1. Non-linear tile abstract node to non-linear tile abstract node: The two vertices already existed in the original RNode graph, and this edge type remains consistent with the original RNode graph.

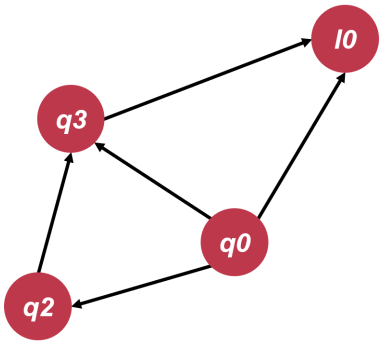

Fig. 9. The abstracted graph of RNode graph constructed from constraint set A and B.

2. Non-linear tile abstract node to linear tile abstract node: This edge type exists only if there are non-abstract nodes in the linear tile represented by the abstract node.
3. Linear tile abstract node to linear tile abstract node: This edge type exists only if the two abstract nodes represent linear tiles that share common non-abstract nodes.

3.3 Tile Weight Calculation

After the further abstraction of the data flow graph, the entire graph appears to be much simpler, with a significantly reduced number of nodes and edges. In specific constraint groups, symmetric scenarios may arise, causing certain nodes to receive identical weights in the algorithm, thereby making subsequent constraint sorting challenging. To address this issue, we employed the Weighted Pagerank algorithm, drawing inspiration from previous literature. The edge weights in the data flow graph were determined using the coefficient-normalized variance of linear tiles.

3.4 Constraint Generation

In this step, we generate constraints in descending order of weight, with tiles as the unit, starting with quadratic constraints and followed by linear constraints. Then, constraint group A and constraint group B are transformed according to the generated constraints.

$$A = \begin{pmatrix} 0\,1\,0\,0 & 0 \\ 0\,1\,0\,0 & 0 \\ 5\,1\,0\,1 & -1 \end{pmatrix} B = \begin{pmatrix} 0\,0\,1\,0\,0 \\ 0\,1\,0\,0\,0 \\ 1\,0\,0\,0\,0 \end{pmatrix} C = \begin{pmatrix} 0\,0\,0\,1\,0 \\ 0\,0\,1\,0\,0 \\ 0\,0\,0\,0\,0 \end{pmatrix}$$

3.5 Adjustment of Linear Constraints

At this stage, the partition and ordering of constraints within a constraint group have been established, and the ordering of variables that appeared in quadratic constraints has also been determined. However, it is necessary to adjust the ordering of newly introduced variables in linear tiles in this step. This is because, in the previous steps, the specific structures of linear tile constraints were abstracted to eliminate differences in the RNode Graph. Therefore, a new method is required to order the newly introduced variables in linear tiles.

For each new variable introduced in a linear tile, its weight is calculated as the sum of the absolute values of the products of its coefficient and the weights of other linear tiles, excluding itself. Then, the new variables can be sorted based on their weights. If the weights are the same, the coefficients of the variable in its own linear tile are considered for comparison. The appearance of new variables in other linear tiles to some extent reflects their importance in the entire constraint group. Additionally, if certain new variables only appear in their own constraints, their weights will be zero, and their ordering will only affect the constraints generated by their own linear tile, without changing the ordering of other constraints. Therefore, they can be sorted in descending order based on their coefficients alone.

4 Algorithm

Algorithm 1 Create the RNode representing the sum

Output: two RNodes
Input: the RNode N representing the product of the input RNodes
$N \leftarrow$ **new** $rightnode$
$N.operation \leftarrow Multiple$
$N.father \leftarrow \{rightnode1, rightnode2\}$
$rightnode1.child \leftarrow \{N\}$
$rightnode2.child \leftarrow \{N\}$

Algorithm 2 Create the RNode representing the sum

Output: two RNodes
Input: the RNode N representing the sum of the input RNodes
$N \leftarrow$ **new** $rightnode$
$N.operation \leftarrow Add$
$N.father \leftarrow \{rightnode1, rightnode2\}$
$rightnode1.child \leftarrow \{N\}$
$rightnode2.child \leftarrow \{N\}$

Algorithm 3 Create RNode Graph from R1CS‘

Output: A R1CS r of size $w \times l$
Input: the list res of RNode in the created graph
$res \leftarrow \emptyset$

for $i \leftarrow 1$ to w **do**
  $res.append($**new** $rightnode)$
end for
for $z \leftarrow 1$ to l **do**
  $node_a \leftarrow$ **None**
  $cons \leftarrow z^{th}$ constraint in r
  for $i \leftarrow 0$ to l **do**
    $node_a \leftarrow$ **None**
    if $cons$.a[i] $\neq 0$ **then**
      if i == 0 **then**
        $node_a \leftarrow CreateConstNode(cons.a[i])$
        $res.append(node_a)$
      else
        $tmp \leftarrow CreateConstNode($cons.$a[i])$
        $node_a \leftarrow Multiple(tmp, res[i])$
        $res.append(node_a)$
        $res.append(tmp)$
      end if
      for $j \leftarrow 0$ to l **do**
        $node_b \leftarrow$ **None**
        if $cons.b[j] \neq 0$ **then**
          if j == 0 **then**
            $node_b \leftarrow CreateConstNode($cons.$b[j])$
            $res.append(node_b)$
          else
            $tmp \leftarrow CreateConstNode(cons.b[j])$
            $node_a \leftarrow Multiple(tmp, res[j])$
            $res.append(node_b)$
            $res.append(tmp)$
          end if
          if i and j are the indices corresponding to the last non-zero elements in the constraint. **then**
            if only one element in the $cons$.c is not zero **then**
              if leftnode == **None then**
                result node $\leftarrow corespondingnodeofnone-zeroelementinc$
                result node.$operation \leftarrow Multiple$
                result node.$father \leftarrow \{node_a, node_b\}$
                $node_a.child.append\{$result node$\}$
                $node_b.childappend\{$result node$\}$
              else
                $rightnode \leftarrow Multiple(node_a, node_b)$
                $res.append(rightnode)$
                result node $\leftarrow corespondingnodeofnone-zeroelementinc$
                result node.$operation \leftarrow Add$
                result node.$father \leftarrow \{leftnode, rightnode\}$

```
            leftnode.child.append{result node}
            rightnode.childappend{result node}
        end if
    else
        for k ← 0 to l do
            if cons.c[k] ≠ 0 then
                if i == 0 then
                    node_c ← CreateConstNode(cons.c[k])
                    res.append(node_c)
                else
                    tmp ← CreateConstNode(cons.c[k])
                    node_c ← Multiple(tmp, res[k])
                    res.append(node_c)
                    res.append(tmp)
                end if
                if rightnode == None then
                    rightnode ← node_c
                else
                    rightnode ← Add(rightnode, node_c);
                    res.append(rightnode)
                end if
            end if
        end for
        if leftnode == None then
            leftnode ← Multiple(node_a, node_b)
            res.append(leftnode)
        else
            tmp ← Multiple(node_a, node_b)
            leftnode ← Add(leftnode, tmp)
            res.append(leftnode)
            res.append(tmp)
        end if
    end if
    gotonextconstraint
else
    if leftnode == None then
        leftnode ← Multiple(node_a, node_b)
        res.append(leftnode)
    else
        tmp ← Multiple(node_a, node_b)
        leftnode ← Add(leftnode, tmp)
        res.append(leftnode)
        res.append(tmp)
    end if
end if
```

```
            end if
        end for
    end if
  end for
end for
```

Algorithm 4 Select tiles from RNode graph

Output: the root RNode of the tile to be chosen and a flag
Input: set of the edges of the chosen tile, s

```
function GetTile(r, flag, s)
  if r has no predecessor nodes then
    return
  end if
  if flag == False & r.opretion == Multiple & both father nodes of r represents constant value then
    return
  end if
  if flag == True & r.opretion == Multiple & both father nodes of r represent variables then
    s.add(< r, father_left >)
    s.add(< r, father_right >)
    return
  end if
  if r.operation == Multiple then
    s.add(< r, father_left >)
    s.add(< r, father_right >)
    if father_left represents constant value then
      GetTile(r, false, father_left)
    end if
    if father_right represents constant value then
      GetTile(r, false, father_right)
    end if
  else
    s.add(< r, father_left >)
    s.add(< r, father_right >)
    GetTile(r, false, father_left)
    GetTile(r, false, father_right)
  end if
end function
```

5 Experiment

To evaluate the proposed algorithm in this paper, we implemented the entire process of paradigm generation using Python to verify its results. The simulations conducted included five main activities:

1. Generation of RNode Graph from any R1CS.
2. Selection of a set of tiles from the RNode Graph.
3. Abstraction of the RNode Graph based on the selected tiles.
4. Calculation of the weights of the tiles for subsequent sorting of variables in the R1CS paradigm.
5. Generation of the R1CS paradigm from the tiles.

Due to the lack of related research, there is currently no comprehensive benchmark in this field. Therefore, we summarized some rules for generating equivalent R1CS constraint groups based on the logic used by the mainstream Circom compiler to generate R1CS, and designed a more comprehensive benchmark based on these rules. The benchmark includes the following main categories depending on the reflected situation:

1. Replacement of variable order in R1CS.
2. Transformation of constraint order in R1CS.
3. Introduction of multiple new variables in a single linear constraint in R1CS.
4. Introduction of multiple new variables in multiple linear constraints in R1CS, with shared new variables.
5. Merging and splitting of constraints in R1CS.

The different categories in the benchmark correspond to the different reasons for generating equivalent R1CS. After abstraction of the RNode Graph, the differences in equivalent R1CS are eliminated. Furthermore, the Weighted PageRank Algorithm can correctly calculate the weight sequence of constraints and variables, thereby determining the order of arrangement of constraints and variables in the R1CS paradigm. In the experiments, all instances were correctly transformed to a unique paradigm.

6 Related Work

Historically, research investigating the factors associated with R1CS has focused on satisfiability. Eli et al. design, implement, and evaluate a zero knowledge succinct non-interactive argument (SNARG) for Rank-1 Constraint Satisfaction (R1CS) [2]. Jonathan et al. studies zero-knowledge SNARKs for NP, where the prover incurs finite field operations to prove the satisfiability of a n -sized R1CS instance [3]. Alexander et al. introduce Brakedown, the first built system that provides linear-time SNARKs for NP [4]. Collectively, these studies outline a critical role for simpler and more direct provement proof. The proposal of the R1CS paradigm clearly accelerates research in this area.

Considering Circom and R1CS as two languages before and after compilation, research on the generation of R1CS paradigm is actually more akin to research on semantic consistency in compilation. Currently, both domestic and foreign patent applications and research papers propose ideas and solutions for generating compilation paradigms in other languages, mainly exploring data flow [5],

syntax tree [6], or semantic mapping [7] aspects. these studies offer crucial insights into the fundamental information that semantically identical programs entail in the process of compilation. However, due to the inherent constraints embedded within the R1CS form itself, this paper ultimately elects to use data flow as a starting point for research.

7 Conclusion

R1CS is an indispensable component in the underlying toolchain for zk-SNARKS. However, the correctness and equivalence of R1CS constraint systems have long been difficult to study due to the diversity and flexibility of constraint construction methods. In this paper, we propose an algorithm based on data flow analysis to construct a paradigm for R1CS, which successfully eliminates the differences between equivalent R1CS constraint systems through a series of abstraction processes. Additionally, we propose a series of ordering methods for internal variables and constraints in R1CS, providing reference information for the final paradigm generation. Experimental results demonstrate that our algorithm can identify equivalent R1CS resulting from constraint merging, intermediate variable selection, and variable and constraint reordering, and transform them into a unique paradigm.

However, our study is limited by the sparsity of the generated paradigm matrix, resulting in storage waste. This is because we have not yet incorporated the step of constraint merging into the paradigm generation process. Furthermore, due to the lack of relevant research, there is currently no comprehensive benchmark in this field, and the benchmarks compiled in this paper may contain some omissions.

Further research is needed to establish rules for merging constraints and a more comprehensive benchmark. This requires us to conduct more in-depth research and exploration on the generation rules of R1CS, so as to continuously improve the algorithm proposed in this paper.

References

1. Xing W, Ghorbani A. Weighted pagerank algorithm[C]//Proceedings. Second Annual Conference on Communication Networks and Services Research, 2004. IEEE, 2004: 305-314.
2. Ben-Sasson E, Chiesa A, Riabzev M, et al. Aurora: Transparent succinct arguments for R1CS[C]//Advances in Cryptology–EUROCRYPT 2019: 38th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Darmstadt, Germany, May 19–23, 2019, Proceedings, Part I 38. Springer International Publishing, 2019: 103-128.
3. Lee J, Setty S, Thaler J, et al. Linear-time and post-quantum zero-knowledge SNARKs for R1CS[J]. Cryptology ePrint Archive, 2021.
4. Golovnev A, Lee J, Setty S, et al. Brakedown: Linear-time and post-quantum SNARKs for R1CS[J]. Cryptology ePrint Archive, 2021.

5. Muzi Yuan, Muyue Feng, Gu Ban, et al. Semantic Comparison Method and Device between a Kind of Source Code and Binary Code: China, CN110147235A[P], 2019-08-20.
6. Li Gao, Zhongqi Li, Dongsheng Yang, et al. Compiling method from intermediate language (IL) program to C language program of instruction list: China, CN103123590A[P], 2013-05-29.
7. Yongsheng Zhao, Zhiyong Chen, Rongtao Cui et al. A kind of assembly language is to the code conversion method of higher level language and device: China, CN103123590A[P], 2015-11-18.
8. Allen F E, Cocke J. A program data flow analysis procedure[J]. Communications of the ACM, 1976, 19(3): 137.
9. Lee E A. Consistency in dataflow graphs[J]. IEEE Transactions on Parallel and Distributed systems, 1991, 2(2): 223-235.
10. Ibrahim R. Formalization of the data flow diagram rules for consistency check[J]. arXiv preprint arXiv:1011.0278, 2010.
11. Belles-Munoz M, Isabel M, Munoz-Tapia J L, et al. Circom: A Circuit Description Language for Building Zero-knowledge Applications[J]. IEEE Transactions on Dependable and Secure Computing, 2022 (01): 1-18.
12. García Navarro H. Design and implementation of the Circom 1.0 compiler[J]. 2020.