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, China
{undefeated, Dennis.Chen, 628628, li.g}@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 with 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. It 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. It 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 most significant challenges: privacy and scalability. From both user and developer perspectives, the heightened emphasis on information privacy and security 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 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 R1CS optimization by providing a novel algorithm for generating canonical forms of equivalent R1CS constraints. Our algorithm improves existing methods by eliminating unnecessary redundancy and normalizing representation, thus facilitating the analysis of equivalence and correctness. Furthermore, our benchmark comprehensively evaluates 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 offers the process of the proposed algorithm in this paper. The detailed logic of the critical steps and 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 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 called computation. Computation can be implemented using any Turing-complete programming language, such as C, Python, and Solidity (used for Ethereum innovative contract programming) because the zk-SNARKs algorithm does not depend on any specific programming language.

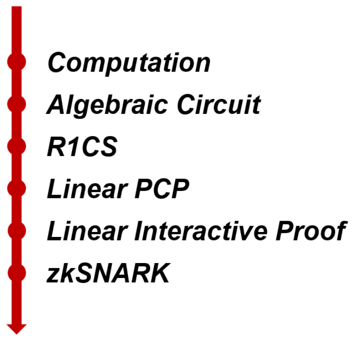


Fig. 1. The pipeline diagram of zk-SNARKs.

Next, the computation result is usually transformed into an arithmetic circuit, a data structure representing the calculation process. An arithmetic circuit consists of multiple gates, each performing an essential 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 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 enhancing security.

Finally, there is the zk-SNARK stage, where the verifiable arithmetic circuit and QAP are used to generate a 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 Circuit Language

Many important issues in today's digital world, such as how to verify identity without revealing private information and how to protect privacy data from illegal exploitation, can be addressed through the use of zero-knowledge proof (ZKP) technology. Among these, arithmetic circuits play an indispensable role in describing and computing various complex operations within ZKP. Specifically, an arithmetic circuit is composed of a series of logic gates that can be used to describe and calculate various types of arithmetic operations, such as addition, multiplication, and division. By combining these basic operations, complex arithmetic circuits can be constructed for accomplishing various computational tasks.

Here, we introduce the formal definition of circuit given in Wiki.

Definition 1. *A circuit is a triple* (M, L, G)*, where*

· M *is a set of values,*
· L *is a set of gate labels, each of which is a function from* M^i *to* M *for some non-negative integer* i *(where* i *represents the number of inputs to the gate), and*
· G *is a labelled directed acyclic graph with labels from* L.

Within the framework of ZKP systems, there exist several commonly used arithmetic circuit description languages, including Arithmetica, libsnark DSL, and Circom, which are typically employed for building and verifying ZKP systems. These languages are capable of describing various types of arithmetic circuits, such as linear constraint systems (LCS), bilinear pairings, and quantum circuits. In this chapter, we mainly focus on the Circom language, which is employed at the core of zk-SNARKs for describing arithmetic circuits.

The core idea of Circom is to represent an arithmetic circuit as a constraint system, where inputs, outputs, and computation processes are all described as linear equations and inequalities. This approach allows developers to define arbitrarily complex computation processes and generate the corresponding R1CS constraint system. 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.

Circom's comprehensive range of features enables the efficient and streamlined development of complex computational structures. It provides a simple, declarative way to define constraint systems, allowing developers to implement various privacy-preserving protocols and zero-knowledge proof techniques quickly. Circom has been widely used in cryptography, blockchain, and other security applications, providing critical support for protecting user privacy and data security.

2.3 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 initially proposed by Larry Page and Sergey Brin, co-founders of Google, in 1998 and has since become one of the most essential algorithms in the field of search engines.

The algorithm analyzes web pages online 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.

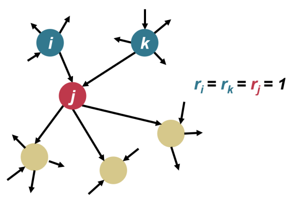


Fig. 2. The initial state of Pagerank algorithm.

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.

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

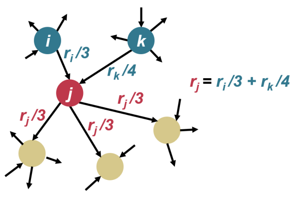


Fig. 3. The interation of Pagerank algorithm.

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

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 and outlinks 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 introduced in Vitalik's blog [2] 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 Basic Knowledge and Definitions of 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. Here is the formal definition of R1CS given in Vitalik's blog [2].

Definition 2. *An R1CS is a sequence of groups of three vectors $(\vec{a}, \vec{b}, \vec{c})$, and the solution to an R1CS is a vector $\vec{s}$, where s must satisfy the equation*

$$(\vec{s} \cdot \vec{a}) * (\vec{s} \cdot \vec{b}) = \vec{s} \cdot \vec{c} \tag{2}$$

Each three vectors in R1CS corresponds to a constraint constructed in Circom, where the values in the triplet are coefficients of the variables at corresponding indices in the solution vector $\vec{s}$. The solution vector $\vec{s}$ for R1CS consists of assignments for all variables that appear in the R1CS.

For example, a satisfied R1CS is shown in Fig.4:

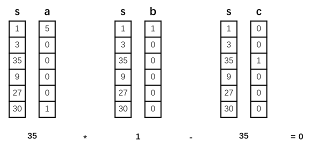


Fig. 4. A satisfied R1CS.

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.

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.

In this paper, we propose the R1CS paradigm for constraint groups through the imposition of constraints on the form and ordering of variable constraints.

Definition 3. *R1CS paradigm is a R1CS satisfies following requirements:*

1. *If a constraint in the R1CS paradigm contains multiplication between variables, it cannot contain any other operators.*
2. *If a constraint in the R1CS paradigm does not contain multiplication between variables, it cannot contain intermediate variables generated by other linear constraints.*
3. *The ordering of constraints and variables in the R1CS paradigm must be consistent with the ordering method proposed in this paper.*

The specific adjustments required for converting a general constraint system into R1CS paradi will be explained through examples with concrete constraints.

Requirement 1 suggests that complex quadratic constraints in the R1CS constraint system should be split into simpler forms. For instance,

$$a \times b + c + d = f \Longrightarrow\ a \times b = r, r + c + d = f$$
$$5 \times a \times b = c \Longrightarrow 5 \times a = r, r \times b = c$$

Requirement 2 indicates that linear constraints in the R1CS constraint system must be eliminated by removing intermediate variables defined by other linear constraints. For example,

$$a + b = c, c + d = e \Longrightarrow a + b + d = e$$

The specific sorting methods in requirement 3 will be discussed in detail in later sections outlining the algorithm's steps.

3.2 Construction of RNode Graph

In our study, we design a new data structure called RNode. An RNode is a type of data structure that we use to represent the relationships between variables in the arithmetic circuit of an R1CS. It helps us keep track of how different variables are related to each other so that we can solve problems efficiently.

Depending on the variables they represent, RNodes can be divided into two categories: one representing original variables in the solution vector of the R1CS, and the other representing intermediate variables generated during the construction of the arithmetic circuit. Each RNode contains both an operator and a computed result, storing the calculation method between its two parent nodes while also representing the computed result of the subtree rooted at itself.

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.

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.

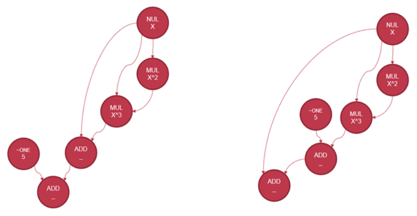


Fig. 5. The structure of the RNode Graph generated by two sets of constraints

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.

3.3 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 $x^3 + 5 + x = z$.

Tile is essentially a set of linear equations that are generated by applying certain constraints to variables in the R1CS. We can use tiles as building blocks to construct a normalize R1CS that is both correct and scalable.

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.
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.

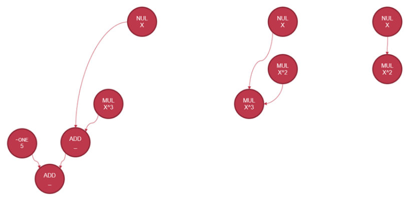


Fig. 6. Tiles selected from the RNode graph constructed from constraint set A.

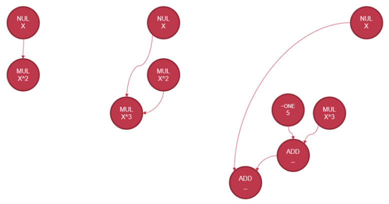


Fig. 7. Tiles selected from the RNode graph constructed from constraint set B.

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.

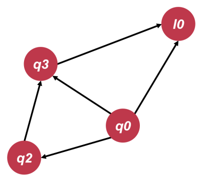


Fig. 8. The abstracted graph of RNode graph constructed from constraint set A and B.

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.
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.

3.4 Tile Weight Calculation

In this step, we use the Weighted PageRank algorithm to calculate the scores of each vertex in the abstracted data flow graph.

The previous steps eliminated the differences in the data flow graphs generated by equivalent R1CS through abstraction of linear tiles. In the following step, constraints are generated on a tile-by-tile basis, and a criterion for tile ordering is proposed to sort the generated constraints.

In the algorithm proposed in this paper, the Weighted PageRank algorithm is used to calculate the weights of each node in the abstracted data flow graph, which are then used as the basis for calculating the weights of the corresponding constraints for each tile. Compared to the traditional PageRank algorithm, this algorithm assigns weights to every edge in the graph and adjusts the iterative formula for node weights. In the Weighted PageRank algorithm, as mentioned in the former section, the formula for calculating node scores is defined as follows:

$$PR(u) = (1-d) + d \sum_{v \in B(u)} PR(v) W^{in}_{(v,u)} W^{out}_{(v,u)} \tag{3}$$

The main purpose of using the Weighted PageRank algorithm in this algorithm is to reduce the symmetry of the abstracted data flow graph. In the previous step, the linear tiles were simplified, resulting in a significant simplification of the structure of the data flow graph. There are symmetric nodes in some structures in the graph. If general algorithms are used to calculate the weights of nodes in the graph, nodes with symmetry in the graph may be assigned the same weight, which can cause problems in subsequent constraint generation and sorting. Therefore, the Weighted PageRank algorithm is used to calculate weights for different nodes, increasing the asymmetry of the graph and avoiding the occurrence of nodes with the same score as much as possible.

In this algorithm, the iterative formula for scores in the Weighted PageRank algorithm is further adjusted. The node weights retained in the original data flow graph are set to 1, while for nodes abstracted from linear constraints, their weights are calculated through a series of steps.

First, for a linear constraint:

$$\sum_{i=1}^{n} a_i b_i = c \tag{4}$$

Convert it to:

$$\sum_{i=1}^{n} a_i b_i - c = 0 \tag{5}$$

Finally, the variance of the normalized coefficients is utilized as the weight of the abstract node representing the linear constraint.

$$W = \frac{\sum_{i=1}^{n} \left(a_i - \frac{\sum_{i=1}^{n} a_i - 1}{n+1} \right)^2 + \left(-1 - \frac{\sum_{i=1}^{n} a_i - 1}{n+1} \right)^2}{\left(\frac{\sum_{i=1}^{n} a_i - 1}{n+1} \right)^2} \tag{6}$$

In this algorithm, the iterative formula for the node score in Weighted PageRank algorithm is given by:

$$PR(u) = (1-d) + d \sum_{v \in B(u)} PR(v) W^u W^v \tag{7}$$

After calculating the scores for each node, the ranking of each constraint in the R1CS form is determined based on the scores.

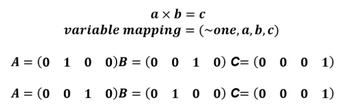
3.5 Constraint Generation

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.

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

In this phase, a portion of the variable ordering was determined through quadratic tile. In the tripartite matrix group of the R1CS paradigm, the three row vectors corresponding to quadratic constraints each have only one non-zero coefficient, but in matrices A and B, the two row vectors corresponding to the same constraint are equivalent in the constraint representation, consistent with the commutativity of multiplication. This results in two equivalent expressions for the same quadratic constraint.

Figure9 illustrates an example of equivalent expressions for a quadratic constraint.



$$a \times b = c$$
$$variable\ mapping = (\sim one, a, b, c)$$
$$A = (0 \quad 1 \quad 0 \quad 0) B = (0 \quad 0 \quad 1 \quad 0)\ C = (0 \quad 0 \quad 0 \quad 1)$$
$$A = (0 \quad 0 \quad 1 \quad 0) B = (0 \quad 1 \quad 0 \quad 0)\ C = (0 \quad 0 \quad 0 \quad 1)$$

Fig. 9. Two equivalent expressions of a quadratic constraint.

In the abstract data flow graph, all vertices representing variables appearing in quadratic constraints are retained, making it possible to determine the choice of non-zero coefficients in the corresponding row vectors of matrices A and B in the tripartite matrix group of the R1CS paradigm based on the weights of each variable. Specifically, variables with higher weights are assigned to matrix A and given smaller indices in the variable mapping.

The sorting rules for the ordering of variables that appear in quadratic constraints can be summarized as follows:

1. Sort variables based on the highest weight value among all quadratic constraints in which they appear, such that variables with higher weight values have smaller indices in the variable mapping.

2. For variables that appear in the same constraint and have the same highest weight value, sort them based on their scores in the Weighted PageRank algorithm for the nodes in the data flow graph corresponding to the variables. Variables with higher node scores will have smaller indices in the variable mapping and will be assigned to the corresponding row vector in matrix A of the tripartite matrix group.

3.6 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. As demonstrated in the example below, introducing multiple new variables within a linear tile can result in disorderly sorting.

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

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

Therefore, we introduce a new method to order the newly introduced variables in linear tiles. For each new variable introduced in a linear tile, its weight is calculated as

$$weight = \sum_{other\ linear\ tiles} |field * weight\ of\ linear\ tile| \tag{8}$$

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

Now, we can give a formal definition of **RNode**.

Definition 4. *An RNode is a vertex in the arithmetic circuit constructed for the R1CS and serves as a storage data structure for the variable relationships in the R1CS.*

Also, we can give the definition of **tile**.

Definition 5. *Tile is a tree-like subgraph of the RNode Graph, which can be represented as a constraint in R1CS.*

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.*
3. *AddLinear: Tiles whose root is obtained by adding its two parents.*

With the above definition, then we give the algorithm of creating RNode Graph from R1CS. The core logic of the algorithm is quite similar with the procedure of constructing RNode Graph, we mentioned before. Detailed pseudocode is shwon in the appendix section of the paper.

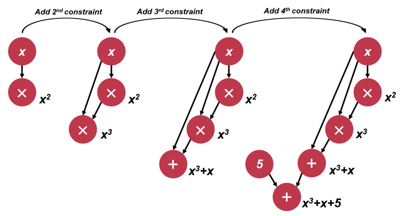


Fig. 10. The procedure of creating RNode Graph from the R1CS constraint set A in Section Overview.

When the RNode graph is generated, we select a node having no successor nodes and use it as the root to partition a subgraph from the RNode Graph as a tile. We then remove this tile from the RNode Graph, and repeat this loop until the entire graph is partitioned. Detailed pseudocode is shwon in the appendix section of the paper.

5 Experiment

In this section, we introduced the self-designed benchmark used in this paper and evaluated the effectiveness of the paradigm generation algorithm through analyzing the test results and the intermediate outputs of the algorithm.

5.1 Benchmark Design

To evaluate the proposed algorithm in this paper, we implemented the entire process of paradigm generation explained in former section using Python to verify its results.

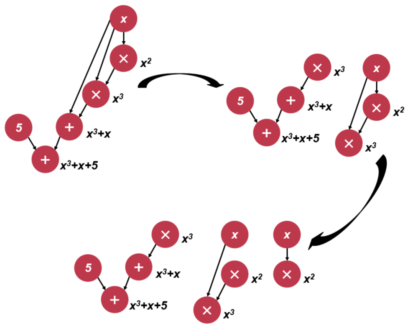


Fig. 11. The procedure of selecting tiles from the RNode Graph generated from constraint set A in Section Overview.

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.

5.2 Result Evaluation

Table 1 shows the result of the experiments.

Inspection shows that the generated paradigms all meet the defined requirements of this paper and have the same semantics as the original R1CS constraint groups.

Through analysis of the intermediate outputs at each stage of the conversion process, it was found that for equivalent R1CS constraint groups generated by

Table 1. Experimental Results of Equivalent R1CS Constraint Group Conversion

Reasons for Generating Equivalent R1CS Constraints	Number of Groups	Successfully Generated Groups	Pass Rate
Replacement of variable order in R1CS.	55	55	100%
Reordering of constraint sequences in R1CS.	21	21	100%
Introduction of multiple new variables in a single linear constraint in R1CS.	15	15	100%
Introduction of multiple new variables with shared usage in multiple linear constraints in R1CS.	15	15	100%
Merging and splitting of constraints in R1CS.	6	6	100%

reordering constraints, the only difference in the generated data flow graphs lies in the order in which nodes representing intermediate variables are created. This is due to different processing orders of each constraint during traversal, leading to different orders of introducing intermediate variables in each constraint.

For equivalent R1CS constraint groups generated by variable replacement, the differences lie in the order in which nodes representing initial variables are generated, and in the order of addition in the summation chain structure caused by differences in variable order in linear constraints. However, these changes do not affect the selected tile set and the same data flow graph is obtained after abstraction.

Equivalent R1CS constraint groups generated by introducing multiple new variables in linear constraints all result in the same data flow graph after abstraction operations on the linear constraint group, and the sorting of newly introduced variables also produces a variable mapping sequence that conforms to the definition.

For equivalent R1CS constraint groups generated by merging and splitting constraints, the only difference in the generated data flow graphs lies in whether vertices representing intermediate variables represent originally existing variables or intermediate variables introduced during the creation of the data flow graph. However, this does not affect the structure of the data flow graph. The splitting and merging of linear constraints cause changes in the order of addition in the summation chain in the data flow graph, but this is resolved after abstraction of the data flow graph.

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) [3]. 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 [4]. Alexander et al. introduce Brakedown, the first built system that provides linear-time SNARKs for NP [5]. 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 [6], syntax tree [7], or semantic mapping [8] 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.

8 Appendix

Algorithm 1 Create RNode Graph from R1CS

```
Output: A R1CS $r$ consisting of $w$ three vectors, and the length of each vector
  is $l$
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$\}$
                $rightnode.child.append\{$result node$\}$
            end if
        else
            for $k \leftarrow 0$ to $l$ do
                if $cons.$c[k] $\neq 0$ then
                    if i $== 0$ then
                        $node_c \leftarrow CreateConstNode($cons.$c[k])$
                        $res.append(node_c)$
                    else
                        $tmp \leftarrow CreateConstNode($cons.$c[k])$
                        $node_c \leftarrow Multiple(tmp, res[k])$
                        $res.append(node_c)$
                        $res.append(tmp)$
                    end if
                    if $rightnode ==$ None then
                        $rightnode \leftarrow node_c$
                    else
                        $rightnode \leftarrow Add(rightnode, node_c)$;
                        $res.append(rightnode)$
                    end if
                end if
            end for
            if $leftnode ==$ None then
                $leftnode \leftarrow Multiple(node_a, node_b)$
                $res.append(leftnode)$
            else
                $tmp \leftarrow Multiple(node_a, node_b)$
                $leftnode \leftarrow Add(leftnode, tmp)$
                $res.append(leftnode)$
                $res.append(tmp)$
            end if
        end if
        $gotonextconstraint$
    else
        if $leftnode ==$ None then
            $leftnode \leftarrow Multiple(node_a, node_b)$
            $res.append(leftnode)$
```

```
                    else
                        $tmp \leftarrow Multiple(node_a, node_b)$
                        $leftnode \leftarrow$ Add(leftnode, tmp)
                        $res.append(leftnode)$
                        $res.append(tmp)$
                    end if
                end if
            end if
        end for
    end if
  end for
end for
```

Algorithm 2 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
```

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. Vitalik Buterin. Quadratic Arithmetic Programs: from Zero to Hero[EB/OL].(2016-12-12)[2023-05-08].https://medium.com/@VitalikButerin/quadratic-arithmetic-programs-from-zero-to-hero-f6d558cea649
3. 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.
4. Lee J, Setty S, Thaler J, et al. Linear-time and post-quantum zero-knowledge SNARKs for R1CS[J]. Cryptology ePrint Archive, 2021.
5. Golovnev A, Lee J, Setty S, et al. Brakedown: Linear-time and post-quantum SNARKs for R1CS[J]. Cryptology ePrint Archive, 2021.
6. 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.
7. 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.
8. 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.
9. Allen F E, Cocke J. A program data flow analysis procedure[J]. Communications of the ACM, 1976, 19(3): 137.
10. Lee E A. Consistency in dataflow graphs[J]. IEEE Transactions on Parallel and Distributed systems, 1991, 2(2): 223-235.
11. Ibrahim R. Formalization of the data flow diagram rules for consistency check[J]. arXiv preprint arXiv:1011.0278, 2010.
12. 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.
13. García Navarro H. Design and implementation of the Circom 1.0 compiler[J]. 2020.