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

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

School of Software, Shanghai Jiao Tong University, 200240, Shanghai, China
`{undefeated, Dennis.Chen, li.g, 628628}@sjtu.edu.cn`

Abstract. The zero-knowledge proof (ZKP) system, receiving extensive attention since it was proposed, is an essential branch of cryptography and computational complexity theory. Rank-1 Constraint Systems (R1CS) provide the bi-linear equations verifier for ZKPs. To represent the R1CS correctly and efficiently, various circulate languages such as Circom are proposed to automatically compile advanced programs to R1CS. However, due to the flexibility of the R1CS representation, R1CS forms compiled from Circom programs with the same semantics often differ significantly. This paper proposes a data-flow-based R1CS paradigm algorithm, generating a standard format of different R1CS with the same semantics. Furthermore, we provide an R1CS benchmark, and experimental evaluation shows the utilization of our methods.

Keywords: Zero-knowledge proof · Rank-1 constraint systems · Data flow graph · Circom · Normalization

1 Introduction

Zero-knowledge proof is increasingly recognized for its importance in modern society [1] as more and more cryptographic communities seek to address some of the blockchain's most significant challenges: privacy and scalability. It is also the essential technique in Zcash [2, 3]. 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 grows, zero-knowledge applications that provide 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 issue into the correct form of computation, known as a *Quadratic Arithmetic Program (QAP)*. R1CS describes the execution of statements written in higher-level programming languages and is used by many ZKP applications, but there is no standard way of representing them.[4] Circom is a novel domain-specific language for defining arithmetic circuits that can be used to generate zero-knowledge proofs.[5] 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 the 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.

Our Contributions 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, we create a relatively complete benchmark based on the identified reasons for producing equivalent R1CS. 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.

Related Work Eli et al. design, implement, and evaluate a zero-knowledge succinct non-interactive argument (SNARG) for Rank-1 Constraint System (R1CS) [6]. Historically, research investigating the factors associated with R1CS has focused on satisfiability. Jonathan et al. study zero-knowledge SNARKs for NP, where the prover incurs finite field operations to prove the satisfiability of an n -sized R1CS instance [7]. Alexander et al. introduce Brakedown, the first built system that provides linear-time SNARKs for NP [8]. These studies outline a critical role for more straightforward improvement proof. The proposal of the R1CS paradigm accelerates research in this area.

Considering Circom and R1CS as two languages before and after compilation, research on the generation of the R1CS paradigm is more akin to research on semantic consistency in the 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 [9], syntax tree [10], or semantic mapping [11] aspects. These studies offer crucial insights into the fundamental information semantically identical programs entail in the compilation process. However, due to the inherent constraints embedded within the R1CS form, this paper ultimately elects to use data flow as a starting point for research.

Paper Organization The rest of this paper is organized as follows. A brief background review of zero-knowledge proof and related tools 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 formal description 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 summarizes the conclusions of the present study.

2 Background

This section covers the basic concepts and principles of zero-knowledge proof and the role of R1CS and Circom in constructing zero-knowledge-proof systems. Additionally, it discusses some existing normalization techniques for R1CS and their limitations, which motivates the proposed data-flow-based normalization generation algorithm presented in this paper.

2.1 Zk-SNARKs

Zk-SNARK stands for Zero-Knowledge Succinct Non-interactive Argument of Knowledge. It was introduced in a 2012 paper [12]. It can allow one party to prove to another that they know a secret without revealing it. Zk-SNARKs is a type of zero-knowledge proof with a working principle that can be simplified into several steps. First, when users need 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.

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 both

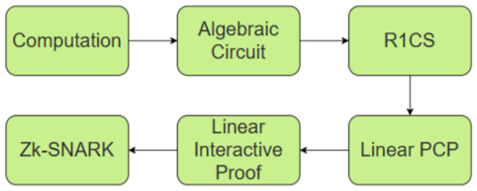

Fig. 1. The pipeline diagram of zk-SNARKs.

makes the implementation of the zk-SNARKs algorithm more efficient and enhance its security of it.

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 essential issues in the modern digital world, such as verifying identity without revealing users' private information and protecting privacy data from illegal exploitation, can be addressed using zero-knowledge proof (ZKP) technology. Arithmetic circuits are indispensable in describing and computing various complex operations within ZKP. Specifically, an arithmetic circuit comprises a series of logic gates that can describe and calculate different types of arithmetic operations, such as addition, multiplication, and division. Combining these basic operations can construct complex arithmetic circuits for various computational tasks.

The circuit mentioned in this paragraph is not an electronic circuit. It is the circuit in theory.

Here is the formal circuit definition in theoretical computer science [13, 14].

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 can describe various arithmetic circuits, such as linear constraint systems (LCS), bilinear pairings, and quantum circuits. This chapter mainly focuses on the Circom language, 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 will enable developers to focus more on the algorithm 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 and describe real-world puzzles, 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-critical applications, providing critical support for protecting user privacy and data security.

2.3 Rank-1 Constraint Systems (R1CS)

R1CS underlies real-world systems [2]. R1CS describes a set of constraint-based computation rules that can be verified using a set of public parameters and a private input sequence. This model represents computations 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 in Vitalik's blog [15].

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*

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

where $\cdot$ represents the dot product - in simpler terms, if we "zip together" a and s, multiply the two values in the same positions, and then take the sum of these products, then do the same to b and s and then c and s, then the third result equals the product of the first two results.

Each triplet of vectors in R1CS represents a mathematical constraint constructed in Circom. These vectors contain coefficients of variables at corresponding positions in the solution vector $\vec{s}$. The solution vector $\vec{s}$ for R1CS includes assignments for all variables appearing in the R1CS equations.

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

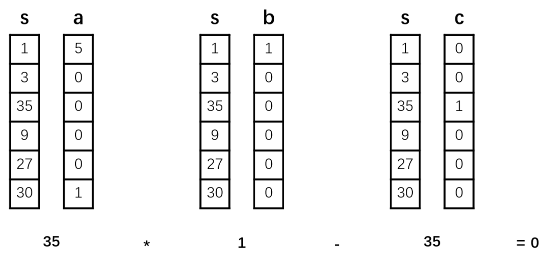

Fig. 2. A satisfied R1CS.

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

R1CS is a powerful computational model widely used in many practical applications. As an essential part of verifiable algorithm design, it dramatically 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 by imposing 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 have 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 an R1CS paradigm 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 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 three will be discussed in detail in later sections outlining the algorithm's steps.

2.4 Data Flow Graph

A data flow graph is a bipartite-directed graph in which links and actors are two node types.[16] In this model, the actors are used to describe the operations, and the links are used to receive data from one actor. Also, links can transmit values to actors by way of arcs. Readers are referred to the formal definition from Dennis' paper [16].

Definition 4. *A data flow graph is a bipartite labeled graph where the two types of nodes are called actors and links.*

$$G = \langle A \cup L, E \rangle \tag{2}$$

where

$$\begin{aligned} A &= a_1, a_2, \ldots, a_n && \textit{is the set of actors} \\ L &= l_1, l_2, \ldots, l_m && \textit{is the set of links} \\ E &\subseteq (A \times L) \cup (L \times A) && \textit{is the set of edges.} \end{aligned}$$

A more detailed description can be found in [17].

2.5 Weighted Pagerank Algorithm

In this paper, we adopt the weighted PageRank algorithm to compute the weight of each node in the data flow graph [18].

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: First, the web pages and links on the internet must be converted into a graph structure. In this structure, each web page corresponds to a node and each link to a directed edge that points 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.
3. Iteratively computing the scores of each page: Each node's score is iteratively calculated 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.

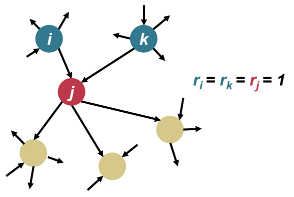

Fig. 3. The initial state of Pagerank algorithm.

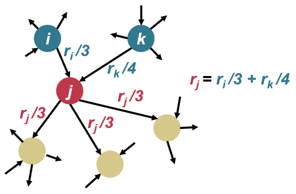

Fig. 4. The interation of Pagerank algorithm.

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

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

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 normalization through the process of converting R1CS introduced in Vitalik's blog [15] in the algorithm:

Constraint Set:

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

Firstly, the arithmetic tree generation process involves creating an arithmetic tree for each constraint within the input R1CS constraint group, which is subsequently merged. The resulting arithmetic tree comprises common subformulas stored in a *Directed Acyclic Graph (DAG)*. The constructed data flow graph's structure is shown in figure 5, which illustrates how the arithmetic trees are combined to form the data flow graph.

Subsequently, a tile selection algorithm is implemented based on the data flow graph, which divides the graph into tiles. The division of the entire graph into tiles is illustrated in figure 6, depicting the overall procedure of the tile selection algorithm. The specifics of the tile selection process, including the form and selection logic, will be elaborated upon in subsequent chapters.

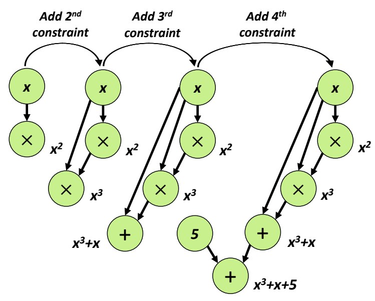

Fig. 5. The procedure of constructing the data flow graph.

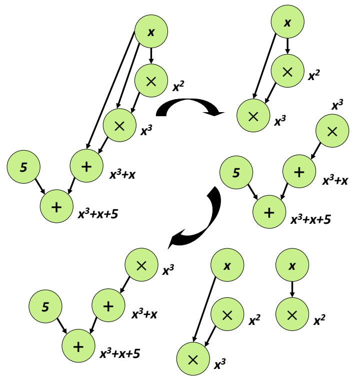

Fig. 6. The procedure of tile selection.

Next, the data flow graph is abstracted further with the selection of tiles as a reference. A new abstracted node in the data flow graph replaces linear constraints represented by tiles. The abstracted node can be represented as an affine mapping, which preserves the linear relationship between the variables, enabling faster computation of the intermediate values during the proof generation process. This abstraction procedure streamlines the proof generation process and reduces the computational cost of generating the proof.

We calculate the weight of each node with coefficients of the constraint. Then we calculate the weights of the selected individual tiles using the improved

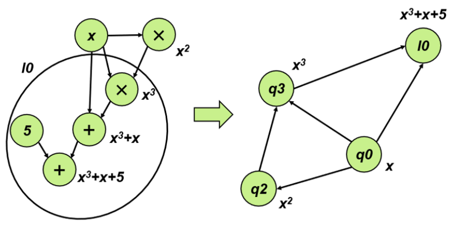

Fig. 7. The procedure of constructing the data flow graph.

Weighted PageRank algorithm. The convergence process of the PageRank values of four nodes in the abstract graph is depicted in figure 8.

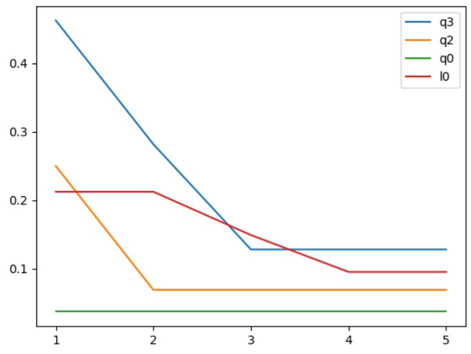

Fig. 8. The convergence process of each node.

Finally, constraints in the paradigm of R1CS are generated separately for each tile. And the constraints and variables are ranked according to the node weights computed in the previous steps.

Now we convert the input R1CS to its paradigm:

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

4 Normalization Algorithm

In this section, we formally introduce various steps of the normalization generation algorithm and several data structures defined within the algorithm.

4.1 Construction of RNode Graph

In our study, we designed 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 track how different variables are related to each other so that we can solve problems efficiently.

Now, we can formally define **RNode**.

Definition 5. *An RNode is a node of two types in the data flow graph constructed in this algorithm.*

$$\begin{aligned} RNode &= ConstNode \cup VarNode \\ ConstNode &= \{ConstValue, Operation, Father, Child\} \\ VarNode &= \{Operation, Father, Child\} \end{aligned} \tag{4}$$

where

$$\begin{aligned} &\forall c \textit{ is a ConstNode} \cap c.Operation = Null, c.child = \emptyset \\ &\forall c \textit{ is a ConstNode} \cap c.Operation \in \{Add, Mul\}, c.father = \emptyset \end{aligned}$$

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 intermediate variables generated during the construction of the arithmetic circuit, and the other representing constants in the data flow graph. Each RNode contains both an operator and a computed result, storing the calculation method between its two parent nodes and representing the calculated result of the subtree rooted in itself.

The generation of the RNode Graph involves three main stages:

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

The RNode Graph generation algorithm's core logic is quite similar to the procedure of constructing the RNode Graph, as we mentioned before. Detailed pseudocode is shown in the appendix A.

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.

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, in our RNode graph generation algorithm, it is observed that this merging or splitting 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 subtracted variable 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.

The main difference observed in the graphs generated by equivalent R1CS is in the order of addition in constructing the expression of continuous addition. 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. During the algorithm execution flow, two variables will be picked randomly and added together, resulting in a different structure in the graph. Therefore, further abstraction of the RNode Graph is required to eliminate this difference.

4.2 Tile Selection

Here, we categorize tiles into three types and give the formal definition of **tile**.

Definition 6. *Tile is a tree-like subgraph of the RNode Graph, representing 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.*

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

While AddLinear and MulLinear tiles are 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.

When the RNode graph is generated, we select a node with no successors 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 shown in the Appendix 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. However, considering the nodes added within a linear constraint as a set, they are equivalent. That is to say, the different addition order only means that the traversal order of the nodes is separate. Therefore, if we regard the selected linear tiles as the products of the selected nodes and their respective coefficients, the chosen 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.

4.3 Graph Abstraction

We further abstract the data flow graph based on the selected tiles to eliminate the differences between various equivalent R1CS constraint systems. Specifically, we abstract linear tiles using a new node. 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 particular 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. 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.

4.4 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 the abstraction of linear tiles. In the following step, constraints are generated on a tile-by-tile basis, and a criterion for tile order 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:

$$PR(u) = (1-d) + d \sum_{v \in B(u)} PR(v) W^{in}_{(v,u)} W^{out}_{(v,u)} \tag{5}$$

The primary 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:

$$\sum_{i=1}^{n} a_i b_i = c \tag{6}$$

Convert it to:

$$\sum_{i=1}^{n} a_i b_i - c = 0 \tag{7}$$

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

$$W = \frac{\sum_{i=1}^{n} \left(a_i - \frac{\sum_{i=1}^{n} a_i - 1}{n+1}\right)^2 + \left(-1 - \frac{\sum_{i=1}^{n} a_i - 1}{n+1}\right)^2}{\left(\frac{\sum_{i=1}^{n} a_i - 1}{n+1}\right)^2} \tag{8}$$

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

$$PR(u) = (1-d) + d \sum_{v \in B(u)} PR(v) W^u W^v \tag{9}$$

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

4.5 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.

In this phase, a portion of the variable ordering was determined through the 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. Still, 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. Figure 9 illustrates an example of equivalent terms for a quadratic constraint.

$$\boldsymbol{a \times b = c}$$
$$\boldsymbol{variable\ mapping = (\sim one, a, b, c)}$$

$$\boldsymbol{A = (0 \quad 1 \quad 0 \quad 0) B = (0 \quad 0 \quad 1 \quad 0)\ C = (0 \quad 0 \quad 0 \quad 1)}$$

$$\boldsymbol{A = (0 \quad 0 \quad 1 \quad 0) B = (0 \quad 1 \quad 0 \quad 0)\ C = (0 \quad 0 \quad 0 \quad 1)}$$

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 be assigned to the corresponding row vector in matrix A of the tripartite matrix group.

4.6 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 order of newly introduced variables in linear tiles in this step. As the example below demonstrates, introducing multiple new variables within a linear tile can result in disorderly sorting. This is because, in the previous actions, the specific structures of linear tile constraints were abstracted to eliminate differences in the RNode Graph.

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

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

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

$$weight = \sum_{other\ linear\ tiles} |field * weight\ of\ linear\ tile| \tag{10}$$

The new variables are sorted based on their weights. If the weights are the same, the coefficients of the variable in its 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, suppose certain new variables only appear in their constraints. In that case, their weights will be zero, and their ordering will only affect the constraints generated by their linear tile without changing the ordering of other constraints. Therefore, they can be sorted in descending order based on their coefficients alone.

5 Evaluation

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

5.1 Benchmark Design

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

Due to the lack of related research, this field has no comprehensive benchmark. Therefore, we summarized some rules for generating equivalent R1CS constraint groups based on the logic the mainstream Circom compiler used to create 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 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 other reasons for generating equivalent R1CS. Each category contains 2-3 basic R1CS constraints. To comprehensively test the robustness and correctness of the algorithm, 5-6 equivalent R1CS constraint groups are generated for each R1CS based on the respective reasons. The equivalent constraint groups of each constraint group are paired and inputted into the algorithm to verify whether the algorithm can generate consistent and R1CS-compliant output results defined in definition 3, when processing different equivalent constraint groups.

The data set used in this study is publicly available at the following GitHub repository: `https://github.com/Ash1sc/R1CS\_normalization\_benchmark`. The repository contains the raw data used for testing. The data set is licensed under the GNU General Public License version 3.0 (GPL-3.0), which allows for the free distribution, modification, and use of the data set and scripts, as long as any derivative works are also licensed under the GPL-3.0 and the original copyright and license information is retained. For more information about the GPL-3.0, please visit `https://www.gnu.org/licenses/gpl-3.0.en.html`.

5.2 Result Evaluation

Table 1 shows the result of the experiments.

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%

Observation shows that the generated paradigms meet the requirements of the R1CS paradigm mentioned above in 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 reordering constraints, the only difference in the resulting 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 other 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 RNodes representing initial variables in the R1CS are developed 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.

In the final step of the algorithm, we proposed a novel variable ordering method to solve the sorting confusion issue when multiple variables are introduced to a linear constraint.Experimental results demonstrate that our algorithm is capable of correctly identifying these variables and produces a variable mapping sequence that conforms to the definition.

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 the abstraction of the data flow graph. 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 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.

6 Conclusion

R1CS is an indispensable part of zk-SNARKS. The correctness and equivalence of R1CS constraint systems have long been challenging 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, eliminating the differences between equivalent R1CS constraint systems through a series of abstraction processes. We propose 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. Further work includes establishing rules for merging constraints and a more comprehensive benchmark. This requires us to conduct more in-depth research and exploration of the generation rules of R1CS to improve the algorithm.

References

1. Goldwasser, S., Micali, S., Rackoff, C.: The knowledge complexity of interactive proof-systems. In: Providing Sound Foundations for Cryptography: On the Work of Shafi Goldwasser and Silvio Micali, pp. 203–225 (2019)
2. ZCash Company, `https://z.cash/` (2014).
3. Sasson, E.B., Chiesa, A., Garman, C., Green, M., Miers, I., Tromer, E., Virza, M.: Zerocash: Decentralized anonymous payments from bitcoin. In: 2014 IEEE symposium on security and privacy, pp. 459–474 (2014)
4. Drevon, G.: J-R1CS – a JSON Lines format for R1CS, (2019).
5. Albert, E., Bellés-Muñoz, M., Isabel, M., Rodríguez-Núñez, C., Rubio, A.: Distilling constraints in zero-knowledge protocols. In: Computer Aided Verification: 34th International Conference, CAV 2022, Haifa, Israel, August 7–10, 2022, Proceedings, Part I, pp. 430–443 (2022)
6. Ben-Sasson, E., Chiesa, A., Riabzev, M., Spooner, N., Virza, M., Ward, N.P.: Aurora: Transparent succinct arguments for R1CS. In: 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, pp. 103–128 (2019)
7. Lee, J., Setty, S., Thaler, J., Wahby, R.: Linear-time and post-quantum zero-knowledge SNARKs for R1CS. Cryptology ePrint Archive (2021)
8. Golovnev, A., Lee, J., Setty, S., Thaler, J., Wahby, R.S.: Brakedown: Linear-time and post-quantum SNARKs for R1CS. Cryptology ePrint Archive (2021)
9. Muzi, Y., Muyue, F., Gu, B., Yang, X., Jiahuan, X., Chendong, Y., Yi, H., Wei, Z.: Semantic comparison method and device between a kind of source code and binary code, (2019).
10. Li, G., Zhongqi, L., Dongsheng, Y., Liu, Y.: Compiling method from intermediate language (IL) program to C language program of instruction list, (2013).
11. Yongsheng, Z., Zhiyong, C., Rongtao, C., Zhili, W.: A kind of assembly language is to the code conversion method of higher level language and device, (2015).
12. Bitansky, N., Canetti, R., Chiesa, A., Tromer, E.: From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again. In: Proceedings of the 3rd Innovations in Theoretical Computer Science Conference, pp. 326–349 (2012)
13. Vollmer, H.: Introduction to circuit complexity: a uniform approach. Springer Science & Business Media (1999)
14. Yang, K.: Integer circuit evaluation is PSPACE-complete. Journal of Computer and System Sciences **63**(2), 288–303 (2001)
15. Buterin, V.: Quadratic arithmetic programs: from zero to hero. URl: https://medium.com/@ VitalikButerin/quadratic-arithmetic-programs-from-zero-to-hero-f6d558cea649 (2016)
16. Dennis, J.B.: First version of a data flow procedure language. In: Programming Symposium: Proceedings, Colloque sur la Programmation Paris, April 9–11, 1974, pp. 362–376 (2005)
17. Treleaven, P.C., Brownbridge, D.R., Hopkins, R.P.: Data-driven and demand-driven computer architecture. ACM Computing Surveys (CSUR) **14**(1), 93–143 (1982)
18. Xing, W., Ghorbani, A.: Weighted pagerank algorithm. In: Proceedings. Second Annual Conference on Communication Networks and Services Research, 2004. Pp. 305–314 (2004)

A RNode Graph Generation Algorithm

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
```

B Tile Selection Algorithm

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**
```