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

Chenhao Shi, Hao Chen, Ruibang Liu, Guoqiang Li†
Shanghai Jiao Tong University, Shanghai 200240, China
{undefeated, Dennis.Chen, 628628, li.g}@sjtu.edu.cn

***Abstract*—The introduction of zero-knowledge proofs (ZKPs) has had a profound impact on the blockchain and distributed ledger communities. ZKPs require the utilization of Rank-1 Constraint Systems (R1CS), which serve as verifiers for bilinear equations. However, the flexibility of R1CS representation leads to notable variations in the compiled R1CS forms derived from circuit language programs with identical semantics. To tackle this challenge, this paper proposes a data-flow-based R1CS paradigm algorithm, producing a standardized format for different R1CS instances with the identical semantics. By adopting the normalized R1CS format circuits, the complexity of circuits' verification can be reduced. Furthermore, this paper presents an R1CS normalization algorithm benchmark, and our experimental evaluation demonstrates the effectiveness and accuracy of our methods.**

***Index Terms*—Zero-knowledge proof; Rank-1 constraint systems; Data flow graph; ZKP programming; Normalization**

I. Introduction

The significance of *zero-knowledge proofs (ZKPs)* in contemporary cryptography is widely acknowledged [1], particularly as various cryptographic communities strive to tackle the critical challenges posed by blockchain technology, namely privacy and scalability. It is also the essential technique in Zcash [2], [3]. Both users and developers have increasingly recognized the importance of information privacy and security, resulting in a heightened appreciation for the privacy advantages offered by zero-knowledge proofs. The *rank-1 constraint system (R1CS)* captures the execution of statements written in high-level programming languages and serves as a fundamental building block for many ZKP applications, but there is currently no standardized representation for them [4]. Circom is a novel domain-specific language for transforming computational problems into R1CS format circuits[5].

The flexible nature of R1CS representation, combined with differences in program organizations and compiler optimization levels, can lead to substantial variations in the compiled R1CS forms derived from circuit language programs with identical underlying semantics.

This paper proposes a data-flow-based algorithm for generating normalization of R1CS, enabling the conversion of different R1CS constraints into a normal form, facilitating the determination of equivalence and correctness. To achieve this, the algorithm starts by transforming an R1CS into a data flow graph structure resembling an expression tree. It then segments and abstracts the data flow graph, eliminating differences between equivalent R1CS constraints that may have emerged during the generation process. Finally, sorting rules are proposed to order the constraints and variables within R1CS, ultimately resulting in a unique normal form for equivalent R1CS.

† Corresponding author.

Moreover, we classify and summarize the reasons and characteristics of the different equivalent R1CS generated, based on the constraint generation logic employed by mainstream compilers and the expressive nature of R1CS.

This work contributes to R1CS by providing a algorithm by generating canonical forms of equivalent R1CS constraints.

Related Work Eli et al. design, implement, and evaluate a zero-knowledge succinct non-interactive argument (SNARG) R1CS [6]. Historically, research investigating the factors associated with R1CS has focused on satisfiability. In paper [7], 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]. The studies outline a vital role for more straightforward improvement proof. The proposal of the R1CS paradigm accelerates research in this area.

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.

II. Preliminaries

This section introduces the basic concepts and principles of zero-knowledge proof and discusses the role of R1CS and Circom in zero-knowledge-proof systems. It also explores the limitations of existing normalization techniques for R1CS and presents the proposed data-flow-based normalization generation algorithm, which is motivated by these limitations.

Zk-SNARK, which stands for Zero-Knowledge Succinct Non-interactive Argument of Knowledge, is a type of zero-knowledge proof introduced in a 2014 paper [12]. The objective of zk-SNARK is to enable one party to prove to another that they possess specific knowledge without revealing the knowledge itself, and to do so concisely and efficiently. The working principle of zk-SNARK can be simplified into several steps. First, users convert the information into a mathematical problem called a computation with a high-level language.

R1CS, a common arithmetic circuit [13] format underlies real-world systems [2] and an important part of the zk-SNARKS algorithm groth16 [14], represents computations as a set of constraint conditions, namely linear equations and inequalities.

Data Flow Graph, a bipartite-directed graph links and actors are two node types [15]. 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. A more detailed description can be found in [16].

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 [17].

III. Normalization Algorithm of R1CS

This section provides formal definitions of the data structures used in the algorithm, as well as a detailed description of the algorithm's specific procedure.

A. Formal Definitions

Definition III.1. A arithmetic circuit is a map $C : \mathbb{F}^n \to \mathbb{F}$, where $\mathbb{F}$ is a finite field.

1) It is a directed acyclic graph (DAG) where internal nodes are labeled $+, -,$ or $\times$ and inputs are labeled $1, x_1, \ldots, x_n$, the edges are wires or connections.
2) It defines an n-variable polynomial with an evaluation recipe.
3) Where $|C| = \#$ gates in C.

Definition III.2. R1CS is a format for ZKP ACs. An R1CS is a conjunction of constraints, each of the form:

$$(\vec{a} \cdot \vec{x}) \times (\vec{b} \cdot \vec{x}) = (\vec{b} \cdot \vec{x})$$

where $\vec{a}, \vec{b}$ and $\vec{c}$ are vectors of coefficients (elements of the prime field consisting of the integers modulo some prime), and $vecx$ is a vector of distinct "pseudo-variables". Each pseudo-variable is either a variable, representing an element of the field, or the special symbol 1, representing the field element 1. $\cdot$ represents taking the dot product of two vectors except that all additions and multiplications are done $\pmod p$, and $\times$ represents the product of 2 scalars $\pmod p$. Using a pseudo-variabale of 1 allows a constant addend to be represented in a dot product.

Definition III.3. R1CS paradigm is an R1CS satisfies the 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 (defined in **Definition III.2**) in the R1CS paradigm must be consistent with the ordering method ($\times$ is in front of $+$) 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 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 later sections outlining the algorithm's steps.

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

$$G = \langle A \cup L, E \rangle \tag{1}$$

where

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

Example III.1. For the purpose of facilitating comprehension, we have chosen the R1CS discussed in Vitalik's blog [18] to provide a more intuitive visualization of the conversion process within the algorithm's specific flow:

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

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

B. Construction of RNode Graph

Our study presents a novel data structure called RNode, which that represents variable relationships within an R1CS arithmetic circuit. This structure, *RNode*, facilitates efficient problem-solving by tracking interconnections among variables, which is formally defined as follows,

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

$$
\begin{aligned}
RNode &= ConstNode \cup VarNode \\
ConstNode &= \{ConstValue, Operation, Father, Child\} \\
VarNode &= \{Operation, Father, Child\}
\end{aligned} \tag{2}
$$

where

$\forall c$ is a ConstNode $\cap\, c.Operation = Null, c.child = \emptyset$

$\forall c$ is a ConstNode $\cap\, c.Operation \in \{Add, Mul\}, c.father = \emptyset$

RNodes can be categorized into two types based on the variables they represent. The first category represents the original variables in the solution vector of the R1CS and the intermediate variables produced during the construction of the arithmetic circuit. The second category represents the constants in the data flow graph. Each RNode includes both an operator and a computed result, which store the calculation method between its two parent nodes and represent the calculated result of the subtree rooted in itself.

The generation of the RNode Graph involves 3 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.

Example III.2. The constructed structure of the data flow graph is depicted in Fig. 1. The construction process follows a similar procedure to the one mentioned earlier for generating the RNode Graph. Unlike typical nodes in an expression tree, each RNode in this graph stores information about both the variable and operator involved in a given operation. This allows the output of each operator to be treated as an intermediate variable, aligning it more closely with the properties of an R1CS constraint set.

In practice, the merging and splitting of constraints poses a challenge in determining the equivalence of R1CS constraint sets. However, our RNode graph generation algorithm has observed that this merging or splitting does not lead to substantial differences. When constraints are merged, one variable is subtracted from the original constraint set.

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.

Further abstraction of the RNode Graph is necessary to eliminate the difference observed in the graphs generated by equivalent R1CS due to variable ordering in the constraint set. The primary disparity lies in the order of addition when constructing the expression of continuous addition. At this stage, there is insufficient information available to determine the sequential execution order. During the algorithm execution flow, two variables are randomly selected and combined, resulting in variations in the graph structure.

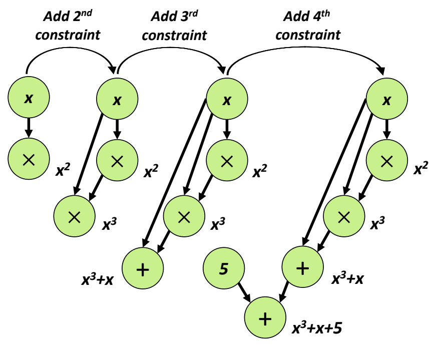

Fig. 1. The procedure of constructing the data flow graph.

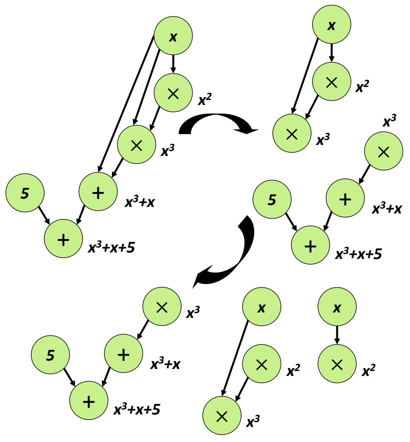

Fig. 2. The procedure of tile selection.

C. Tile Selection

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

Definition III.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.

During tile selection, we divide the data flow graph from the previous step into 3 types. Although AddLinear and MulLinear tiles are generated by linear constraints and are fundamentally linear tiles, their logical processing differs significantly. Therefore, we treat them as distinct types.

1) The constraint merging step is temporarily deferred until we gather additional information about the tree in

subsequent steps.

2) If there is a requirement to generate merged formulas later, it can be accomplished easily by applying a predefined algorithm to the unmerged formulas.
3) The implementation of the tile selection algorithm is relatively simple.

Example III.3. Fig. 2 illustrates the division of the entire graph generated from the provided example into tiles. After the generation of the RNode graph, we select a node with no successors as the root and partition a subgraph from the RNode Graph as a tile. We then remove this tile from the RNode Graph and repeat this process until the entire graph is partitioned. Detailed code will be posted on GitHub.

The difference between data flow graphs generated by equivalent R1CS constraint systems lies in the order of node additions when processing linear tiles. However, the nodes added within a linear constraint, when considered as a set, are equivalent. This implies that the different addition order only affects the traversal order of the nodes. Therefore, the selected sets of linear tiles from equivalent R1CS constraint systems are the same if we consider the selected linear tiles as the products of the selected nodes and their respective coefficients.

D. Graph Abstraction

We abstract the data flow graph based on the selected tiles to eliminate the differences among equivalent R1CS constraint systems. The abstracted graph includes the following types of edges:

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.

Example III.4. The abstraction procedure is shown in the Fig. 3. 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.

E. Tile Weight Calculation

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

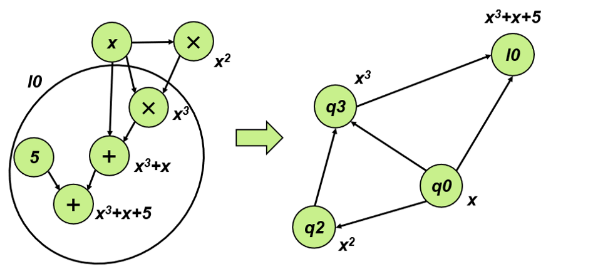

Fig. 3. The procedure of abstracting the data flow graph.

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 primary purpose of using the Weighted PageRank algorithm is to reduce the symmetry of the abstracted data flow graph. The structure of the graph is significantly simplified in the previous step through the simplification of linear tiles. However, some structures in the graph still contain symmetric nodes. If general algorithms are used to calculate the weights of nodes, these symmetric nodes may be assigned the same weight, which can lead to problems in subsequent constraint generation and sorting. To address this issue, the Weighted PageRank algorithm is employed to calculate weights for different nodes. This helps increase the asymmetry of the graph and minimize the occurrence of nodes with the same score.

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.

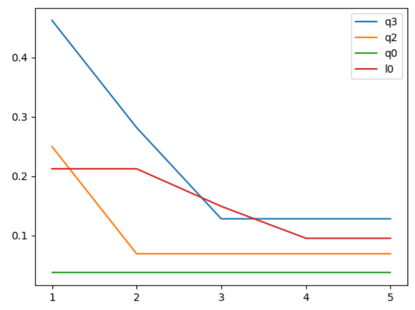

Fig. 4. The convergence process of each node.

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

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

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

Fig. 5. Two equivalent expressions of a quadratic 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 the Weighted PageRank algorithm is given by:

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

Example III.5. The convergence process of the scores of four nodes in the abstract graph is depicted in Fig. 4. The scores for each node are calculated to determine the ranking of each constraint in the R1CS form.

F. 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. Fig. 5 illustrates an example of equivalent terms for 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. The 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.

G. 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. This is because, in the previous actions, the specific structures of linear tile constraints were abstracted to eliminate differences in the RNode Graph.

Example III.6. This example illustrates why introducing multiple new variables within a linear tile can lead to 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

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

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.

Example III.7. Now we generate the constraints from the tile and convert them to the paradigm of the example R1CS in the mentioned order:

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

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

IV. Evaluation

We evaluated the effectiveness of the algorithm by analyzing the test results and the intermediate outputs.

A. 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.

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 III.3, when processing different equivalent constraint groups.

The data set used in this study can be found at the following GitHub repository: https://github.com/Ash1sc.

B. Result Evaluation

Table I shows the result of the experiments.

TABLE I
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%

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.

V. Conclusion

This paper proposes a data-flow-based algorithm for constructing the normal form of a group of semantically equivalent R1CS, a high-level programming language used for ZKP. Future work includes establishing rules for merging constraints and conducting 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.

Acknowledgement

This research is supported by the National Natural Science Foundation of China with Nos. 61872232 and 61672340. The authors would like to thank Dr. Sinka Gao from Delphinus Lab. in Australia for fruitful discussions.

References

[1] S. Goldwasser, S. Micali, and C. Rackoff, "The knowledge complexity of interactive proof-systems," in *Providing Sound Foundations for Cryptography: On the Work of Shafi Goldwasser and Silvio Micali*, 2019, pp. 203–225.

[2] *Zcash company*, https://z.cash/, 2014.

[3] E. B. Sasson, A. Chiesa, C. Garman, *et al.*, "Zerocash: Decentralized anonymous payments from bitcoin," in *2014 IEEE symposium on security and privacy*, IEEE, 2014, pp. 459–474.

[4] G. Drevon, *J-r1cs – a json lines format for r1cs*, 2019.

[5] E. Albert, M. Bellés-Muñoz, M. Isabel, C. Rodrıéguez-Núñez, and A. Rubio, "Distilling constraints in zero-knowledge protocols," in *Computer Aided Verification: 34th International Conference, CAV 2022, Haifa, Israel, August 7–10, 2022, Proceedings, Part I*, Springer, 2022, pp. 430–443.

[6] E. Ben-Sasson, A. Chiesa, M. Riabzev, N. Spooner, M. Virza, and N. P. Ward, "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*, Springer, 2019, pp. 103–128.

[7] J. Lee, S. Setty, J. Thaler, and R. Wahby, "Linear-time and post-quantum zero-knowledge snarks for r1cs," *Cryptology ePrint Archive*, 2021.

[8] A. Golovnev, J. Lee, S. Setty, J. Thaler, and R. S. Wahby, "Brakedown: Linear-time and post-quantum snarks for r1cs," *Cryptology ePrint Archive*, 2021.

[9] Y. Muzi, F. Muyue, B. Gu, *et al.*, *Semantic comparison method and device between a kind of source code and binary code*, 2019.

[10] G. Li, L. Zhongqi, Y. Dongsheng, and Y. Liu, *Compiling method from intermediate language (il) program to c language program of instruction list*, 2013.

[11] Z. Yongsheng, C. Zhiyong, C. Rongtao, and W. Zhili, *A kind of assembly language is to the code conversion method of higher level language and device*, 2015.

[12] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza, "Succinct {non-interactive} zero knowledge for a von neumann architecture," in *23rd USENIX Security Symposium (USENIX Security 14)*, 2014, pp. 781–796.

[13] H. Vollmer, *Introduction to circuit complexity: a uniform approach*. Springer Science & Business Media, 1999.

[14] J. Groth, "On the size of pairing-based non-interactive arguments," in *Advances in Cryptology–EUROCRYPT 2016: 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part II 35*, Springer, 2016, pp. 305–326.

[15] J. B. Dennis, "First version of a data flow procedure language," in *Programming Symposium: Proceedings, Colloque sur la Programmation Paris, April 9–11, 1974*, Springer, 2005, pp. 362–376.

[16] P. C. Treleaven, D. R. Brownbridge, and R. P. Hopkins, "Data-driven and demand-driven computer architecture," *ACM Computing Surveys (CSUR)*, vol. 14, no. 1, pp. 93–143, 1982.

[17] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bring order to the web," Technical report, stanford University, Tech. Rep., 1998.

[18] V. Buterin, "Quadratic arithmetic programs: From zero to hero," *URl: https://medium. com/@ VitalikButerin/quadratic-arithmetic-programs-from-zero-to-hero-f6d558cea649*, 2016.