Circuit Normalization Bottlenecks in Trivial Zero-Knowledge Decision Tree Evaluation *

Kai-Che Shih[1], Tzu-Yu Fan Chiang[1], Wen-Chi Lai[1], Tzu-Chen Huang[1], Chen-Hao Kao[1], Yu-Chi Chen[1], and Wei-Chung Teng[2†]

[1] National Taipei University of Technology, Taipei, Taiwan
{t113598077, t111820009, t111820001, t111820019, t113598003, wycchen}@ntut.edu.tw
[2] National Taiwan University of Science and Technology, Taipei, Taiwan
weichung@csie.ntust.edu.tw

Abstract

Zero-Knowledge Proof (ZKP) is a cryptographic technique that enables a prover to convince a verifier that a computation was executed correctly to obtain the result, without disclosing any data or intermediate results. This work explores the applications of ZKP in verifying decision tree evaluation. We implement the circuit logic using Circom and develop automated scripts to normalize multi-variable expressions into single-variable forms. As circuit constraints are represented using a Rank-1 Constraint System (R1CS), reducing the number of constraints directly improves efficiency in computation in general. Finally, we investigate whether applying normalization on a circuit that heavily performs comparisons can effectively reduce constraints in decision tree circuits (trivially hardcoding the model in the proof, a.k.a trivial ZKP for decision tree evaluation). The normalized circuits are compiled and verified using the Groth16 protocol via SnarkJS, enabling efficient proof generation and validation.

1 Introduction

Zero-Knowledge Proofs (ZKPs) are cryptographic protocols that allow one party to prove to another that a statement is true without revealing any underlying secret information. In recent years, ZKPs have seen significant advancements, expanding their range of practical applications. One particularly notable area is machine learning, where ZKPs can be used to verify that a model's output has been computed correctly without disclosing the underlying data or model parameters. This capability ensures privacy, prevents information leakage, and simultaneously maintains verifiability.

Among modern ZKP constructions, zk-SNARKs (Zero-Knowledge Succinct Non-Interactive Arguments of Knowledge) are widely adopted due to their efficiency and practicality. A key component of many zk-SNARKs is the Quadratic Arithmetic Program (QAP), which represents computations as sets of polynomials whose roots encode the correctness of the computation. QAPs enable succinct verification by ensuring that all constraints are satisfied for a given assignment of variables. These algebraic constraints are typically derived from a Rank-1 Constraint System (R1CS), which formalizes the computation as linear equations over a finite field. The total number of these formalized constraints is directly correlated with the size of the computation and, consequently, has a significant impact on the efficiency and effectiveness of the resulting zero-knowledge proof.

Because the number of constraints is closely tied to protocol efficiency, reducing this number can yield substantial performance improvements. Modern frameworks such as *Circom* therefore apply constraint optimization techniques by default, eliminating redundant constraints that do not affect correctness and thereby compressing the computation into a more efficient form. Nevertheless, these

*Proceedings of the 9th International Conference on Mobile Internet Security (MobiSec'25), Article No. 40, December 16-18, 2025, Sapporo, Japan.

†Corresponding author

optimizations are not always fully effective. A single computation can often be expressed by multiple, syntactically distinct circuits, each generating a different number of constraints. As a result, a circuit that appears optimized in one representation may not be globally optimal.

To address these challenges, Teng and Tseng [12] proposed a normalization method that unifies equivalent circuits into a consistent constraint structure. Their approach represents all values relative to a chosen base, ensuring that every constraint is expressed in relation to this base. This uniform representation not only facilitates further optimizations but also frequently reduces the overall number of constraints. In this work, we evaluate the effectiveness of this normalization method and examine its applicability to decision tree models, which have previously been shown to integrate effectively with zero-knowledge proof systems [3, 14], making them a strong candidate for implementation. Decision trees serve as a representative and tractable case study for exploring the practical benefits and limitations of normalization in more complex circuits.

2 Related Works

In this section, we present the fundamental concepts underlying the implementation of constraint normalization in decision trees.

2.1 Rank-1 Constraint Systems (R1CS)

Rank-1 Constraint Systems (R1CS) [10,13] are widely used mathematical frameworks in zero-knowledge proof protocols. Their purpose is to transform the arithmetic operations of a statement into a system of constraints that captures the relationships among the variables involved in the computation. An R1CS is defined by three main components: **Variables**, **Constraints**, and **Witnesses**.

- The set of **Variables**, $x = (x_1, x_2, \ldots, x_n)$, represents the inputs, intermediate values, and outputs of the computation, formalizing the problem.
- The **Constraints** specify how the variables relate to one another, ensuring the correctness of the computation.
- The **Witnesses** are concrete assignments to the variables that demonstrate the validity of the computation by satisfying all constraints.

R1CS works by verifying whether the values assigned to the variables satisfy the constraints, which are expressed using a *left vector*, *right vector*, and *output vector*. These vectors correspond, respectively, to the left term, right term, and output term of the constraint equation $\langle a, z\rangle \cdot \langle b, z\rangle = \langle c, z\rangle$, where each vector selects a linear combination of variables forming the corresponding term for all constraints in the system.

For an intuitive understanding, the constraint equations can also be interpreted from a circuit perspective, where the computation is represented as a circuit of arithmetic gates. In this view, the vectors A and B correspond to the left and right inputs of the gates, while the vector C corresponds to the outputs. This perspective provides a natural explanation for the naming of the left, right, and output vectors in the R1CS equations.

Given these prerequisites, the main components of the R1CS framework are as follows:

- $(A, B, C) \leftarrow \mathsf{R1CS.Gen}(\mathbf{C})$: Transforms the computation $\mathbf{C}$ into an R1CS instance, represented by three matrices (A, B, C) over a finite field $\mathbb{F}$.
- $\{0, 1\} \leftarrow \mathsf{R1CS.Check}((A, B, C), (\mathsf{x}, \mathsf{w}))$: Verifies that (x, w) is a valid assignment, i.e., that it satisfies all constraints $(Az) \circ (Bz) = Cz$, where $z = (1, \mathsf{x}, \mathsf{w})$.

In essence, each constraint enforces that the product of the selected left and right terms (or gate inputs) equals the output. When all constraints are satisfied, the entire computation or circuit is validated, and the assigned values constitute a valid witness.

2.2 Quadratic Arithmetic Programs (QAP)

The Quadratic Arithmetic Program (QAP) [4,8] is a polynomial representation of computational constraints that builds upon the Rank-1 Constraint System (R1CS) formulation. While R1CS expresses constraints using vectors, QAP interpolates these vector values to encode the constraints as polynomials. This allows each constraint to be represented in polynomial form over a finite field, enabling the use of random evaluation points in zero-knowledge proofs.

In QAP construction, the interpolated polynomial relation is written as

$$\left(\sum_i z_i L_i(x)\right) \cdot \left(\sum_i z_i R_i(x)\right) - \left(\sum_i z_i O_i(x)\right) = H(x) \cdot Z(x), \tag{1}$$

where the polynomials $L_i(x)$, $R_i(x)$, and $O_i(x)$ are root polynomials obtained by interpolating the entries of the corresponding columns of the left, right, and output R1CS matrices over distinct points. The variables z_i correspond to the variable coefficients, including public inputs, witness variables, and intermediate computation values. The product $H(x) \cdot Z(x)$ represents the remaining polynomial after factoring, where $Z(x)$ has roots at all constraint points and $H(x)$ is the quotient polynomial that scales $Z(x)$ to match the left-hand side. This construction ensures that the left-hand side vanishes at all constraint points if and only if the original R1CS constraints are satisfied.

By representing the relation as a polynomial, evaluating it at a randomly chosen secret point provides a probabilistic guarantee that the assigned variable values satisfy all original constraints.

2.3 zk-SNARKs

Zero-Knowledge Succinct Non-Interactive Arguments of Knowledge (zk-SNARKs) [1,2,7,9] are circuit-based proof systems that enable a prover ($\mathcal{P}$) to convince a verifier ($\mathcal{V}$) that a computation has been executed correctly. Some zk-SNARK constructions, such as Groth16 [5,6], employ Quadratic Arithmetic Programs (QAPs) to represent computations. By encoding the computation's constraints as polynomials, QAP-based zk-SNARKs allow the prover to generate succinct proofs that all constraints are satisfied for a given assignment of variables. In other words, the verifier can be convinced that a given input x produces the output $\mathbf{C}(\mathsf{x}) = \mathsf{y}$, without learning any information about the private witness w.

Formally, given a circuit $\mathbf{C}$ and its inputs (x, w), the zk-SNARK framework is defined through the following core components:

- $\mathsf{pp} \leftarrow \mathsf{ZKP.G}(1^\lambda)$: Generates the public parameters pp from the security parameter λ.
- $\pi \leftarrow \mathsf{ZKP.P}(\mathsf{pp}, \mathbf{C}, \mathsf{x}, \mathsf{w}, \mathsf{y})$: Produces a succinct proof π using the circuit $\mathbf{C}$, public input x, witness w, and output y.
- $\{0,1\} \leftarrow \mathsf{ZKP.V}(\mathsf{pp}, \mathbf{C}, \mathsf{x}, \mathsf{y}, \pi)$: Verifies the proof π against the circuit $\mathbf{C}$, input x, and output y using pp, outputting 1 if valid and 0 otherwise.

The system is *succinct*, meaning that both the proof size and verification time are very small relative to the complexity of the computation. It is *non-interactive*, requiring only a single proof message from the prover to the verifier, with no further communication. Finally, it is *zero-knowledge*, ensuring that the verifier learns nothing about the private inputs beyond the validity of the statement being proven. These properties make zk-SNARKs particularly well-suited for applications where privacy, efficiency, and verifiability are essential.

Since zk-SNARKs are a specific type of zero-knowledge proof system, they must satisfy the fundamental properties of any argument system for an NP relation R:

- Completeness: If $(\mathsf{x}, \mathsf{w}) \in R$, then for any public parameters pp:

$$\Pr[\mathsf{V}(\mathsf{pp}, \mathsf{x}, \pi) = 1] = 1.$$

- Knowledge Soundness: For any adversarial prover $\mathcal{P}^*$, there exists an extractor Γ such that if $\mathcal{P}^*$ convinces $\mathcal{V}$ with some proof π^*, then Γ can extract a valid witness w, where:
$$\Pr[(\mathsf{x}, \mathsf{w}) \notin R \wedge \mathsf{V}(\mathsf{pp}, \mathsf{x}, \pi^*) = 1] \leq \text{negl}(\lambda).$$
- Zero-Knowledge: There exists a simulator Sim such that for any (possibly malicious) verifier $\mathcal{V}^*$, the view of an interaction with the real prover is computationally indistinguishable from the simulator's output:
$$\text{View}\langle \mathsf{P}(\mathsf{pp}, \mathsf{x}, \mathsf{w}), \mathsf{V}^*(\mathsf{pp}, \mathsf{x}, \pi)\rangle \approx \text{Sim}_{\mathcal{V}^*}(\mathsf{x}).$$

2.4 Constraint Normalization

Normalization is a critical step in ZKP circuit design, aimed at standardizing the structure of R1CS. This process is necessary because equivalent computations can often be expressed in multiple forms that yield different numbers of constraints, which directly impacts the efficiency of the proof system. Earlier methods, such as that of Shi et al. [11], focused on partially reducing structural redundancies but could not guarantee that two circuits implementing the same computation would normalize to an identical form. Teng and Tseng [12] address this limitation by introducing a systematic normalization method that enforces a canonical representation and, after optimization, often results in fewer overall constraints.

Figure 1 illustrates the R1CS constraints after normalization, showing how a circuit is transformed into a standardized structure that facilitates further optimizations.

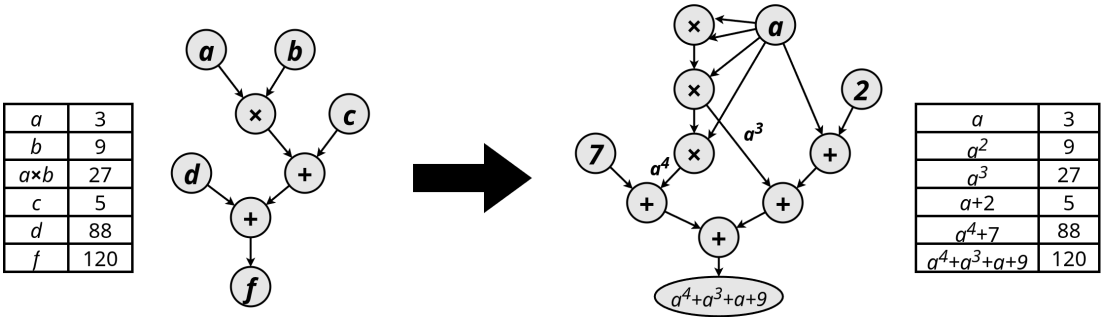

Figure 1: Normalization of an example circuit

The key idea of the normalization approach is to designate the smallest-valued signal x in the circuit as a *base signal* and express all other signals in terms of this base. Each signal is represented as a polynomial function of x, typically in the form
$$y = x^a + b,$$
where a is a non-negative integer exponent and b is a constant offset. This ensures that the representation of each signal is uniquely determined by the base, eliminating ambiguity in how equivalent constraints are expressed.

To support this representation, the method introduces a *power circuit*, an auxiliary structure that systematically constructs higher-order powers of the base signal. The power circuit iteratively derives terms such as $x^2, x^3, \ldots, x^n$ through chained multiplications (e.g., $x^i = x^{i-1} \cdot x$). Although this increases the number of intermediate signals, it creates a uniform structural backbone that exposes redundancies across constraints.

The multivariable constraint normalization process can be summarized as follows:

1. *Identify constraints:* Examine each constraint, excluding the constant term and the output variable, to determine if it involves more than one distinct variable. Such constraints require transformation.

2. *Select a reference variable:* Choose a reference variable from the set of variables, typically the one associated with the smallest corresponding value in an auxiliary ordering vector.

3. *Express remaining variables in terms of the reference:* Encode all other variables using the reference variable, often via powers or coefficients to ensure uniqueness.

4. *Expand compound terms:* Temporarily expand any sums or differences of multiple variables so that they can also be encoded relative to the reference variable.

5. *Transform vectors:* Apply the encoding to both the coefficient vector and the variable vector, ensuring all multivariable entries are now expressed in terms of the reference variable.

6. *Proceed with single-variable normalization:* After reducing all multivariable terms to a single reference variable, continue with standard single-variable normalization to produce a uniform constraint structure.

Once normalized, circuits that may appear syntactically different are reduced to the same canonical form. This structural alignment enhances the effectiveness of compiler optimizations, making redundant constraints easier to identify and eliminate. As a result, normalized circuits often achieve lower overall constraint counts, improving efficiency in both proof generation and verification.

Formally, given a circuit $\mathbf{C}$ and its inputs (x, w), the normalization framework can be defined through the following components:

- $\mathsf{x}_{base} \leftarrow \mathsf{Derive}(\mathsf{x}, \mathsf{w})$: Identifies the minimum value among the inputs to serve as the base signal.
- $(\mathbf{C}_{norm}, (\mathsf{x}_{norm}, \mathsf{w}_{norm})) \leftarrow \mathsf{Normalize}(\mathbf{C}, (\mathsf{x}, \mathsf{w}), \mathsf{x}_{base})$: Substitutes all inputs and intermediate variables according to the base, producing a normalized circuit and input assignment.

2.5 Decision Tree

Decision trees are a simple yet widely used class of machine learning models that partition the input space into regions based on a sequence of decision rules. Each internal node in the tree corresponds to a comparison of a feature value against a threshold, while each leaf node assigns an output label or prediction. In this way, decision trees are commonly employed to classify data into discrete categories or labels.

Formally, a decision tree $\mathbf{T}$ for classification can be viewed as a binary tree where:

- Each internal node is associated with a feature index i and a threshold θ, representing the rule $x_i \leq \theta$.
- Each edge corresponds to the outcome of the decision: the left child for "true" and the right child for "false."
- Each leaf node contains a prediction value $y \in \mathcal{Y}$.

The evaluation of a decision tree consists of traversing the tree from the root to a leaf by applying the decision rule at each node to the input vector x, and finally returning the label stored at the reached leaf. This procedure is formally described in Algorithm 1.

Algorithm 1 Decision Tree Evaluation with Node Attributes

Require: Decision tree $\mathbf{T}$, input vector $\mathsf{x} = (x_1, x_2, \ldots, x_d)$
Ensure: Prediction y
$v \leftarrow \mathbf{T}$.root
while v.isLeaf = false **do**
  $i \leftarrow v$.feature, $\theta \leftarrow v$.threshold
  if $x_i \leq \theta$ **then**
    $v \leftarrow v$.leftNode
  else
    $v \leftarrow v$.rightNode
  end if
end while
return v.label

3 System Architecture and Workflow

In order to evaluate the applicability of constraint normalization in decision tree models, we first establish a workflow that systematically transforms the trained model into a normalized circuit. This workflow is designed to capture the differences between the original and normalized circuits in terms of constraint structures.

3.1 System Software Specification

The experiments were conducted using software tools for zero-knowledge proof circuit compilation, proof generation, and machine learning computation. The main software components are summarized in Table 1.

Table 1: System Software Specification

Software	Version	Description
Circom	2.2.2	Circuit design and compilation for zkSNARKs.
SnarkJS	0.7.5	Proof generation and verification toolkit.
Python	3.11.3	Scripting and automation environment.
PyTorch	2.5.0	Framework for training decision tree models.

Circom (v2.2.2) was used to design and compile arithmetic circuits for zero-knowledge proofs, while SnarkJS (v0.7.5) handled proof generation and verification. Python 3.11.3 was used to manage data processing and automation tasks, as well as to convert decision tree models into Circom circuits. PyTorch 2.5.0 was employed to train the decision tree models.

3.2 Main Experiment

To implement this workflow, we utilize PyTorch for model training, Circom and SnarkJS for circuit construction and verification, and adopt the normalization technique proposed by Teng and Tseng [12]. For our study, we selected the Wine, Olivetti Faces (OF), Breast Cancer (BC), and Digits datasets, as their sizes are appropriate for evaluating the workflow in a controlled and systematic manner.

The following subsection describes the specific steps undertaken in this experiment.

1. **Train Model** – PyTorch was used to preprocess the input dataset and train the decision tree model. The decision trees were configured with a maximum depth of 17 to limit overfitting, the

default Gini impurity criterion for node splitting, and all available features considered at each split. After training, the resulting decision rules were exported in a structured JSON format (e.g., `tree_rules.json`) for subsequent Circom circuit generation, including node thresholds, logical conditions, and corresponding prediction outcomes.

2. **Generate Circuit** – Convert the decision rules JSON file into a functionally equivalent circuit using Circom. Instead of explicit tree traversal, each decision node is replaced with a `comparator` template from `circomlib`, which compares the input feature against its threshold and outputs a boolean. The comparison is implemented via bit decomposition, allowing the circuit to operate on individual bits of the input values. Collectively, these comparators encode the decision boundaries in parallel, producing an index-based output corresponding to the predicted label. This process yields the circuit description file `circuit.circom` and the associated weight file `weights.json`.
3. **Normalize Circuit** – Apply normalization to the generated circuit and weight file. This involves identifying the smallest weight value, expressing all other weights as powers of this base, and updating the corresponding entries in the circuit file with normalized signals. The outputs are the normalized circuit file `circuit_normalized.circom` and the normalized weight file `weights_normalized.json`, directly corresponding to the original outputs.
4. **Compare Circuits** – The normalized circuit is evaluated against the original circuit to determine which design produces fewer constraints. To ensure correctness, it ensure that the outputs of the two circuits are examined to confirm that they are identical. Then SnarkJS library is used to perform the zero-knowledge proof process via the Groth16 protocol.

Algorithm 2 illustrates the procedure for generating decision tree circuits from datasets and normalizing them to produce a uniform constraint structure. In contrast, Algorithm 3 describes the subsequent zk-SNARK workflow, encompassing proof generation and verification to ensure the correctness of the normalized circuits.

Algorithm 2 Decision Tree Circuit Generation and Normalization

Require: Dataset collection $\mathcal{D} = \{\mathbf{D}_1, \mathbf{D}_2, \ldots, \mathbf{D}_n\}$
Ensure: Normalized circuits $\mathbf{C}_{norm,i}$ and inputs $(\mathsf{x}_{norm,i}, \mathsf{w}_{norm,i})$

for all $\mathbf{D}_i \in \mathcal{D}$ **do**
  $\mathbf{T}_i \leftarrow \mathsf{Train}(\mathbf{D}_i)$
  $(\mathbf{C}_i, (\mathsf{x}_i, \mathsf{w}_i)) \leftarrow \mathsf{GenerateCircuit}(\mathbf{T}_i)$
  $\mathsf{x}_{base,i} \leftarrow \mathsf{Derive}(\mathsf{x}_i, \mathsf{w}_i)$
  $(\mathbf{C}_{norm,i}, (\mathsf{x}_{norm,i}, \mathsf{w}_{norm,i})) \leftarrow \mathsf{Normalize}(\mathbf{C}_i, (\mathsf{x}_i, \mathsf{w}_i), \mathsf{x}_{base,i})$
end for

Algorithm 3 zk-SNARK Proof Generation and Verification to ensure correctness

Require: Circuits $\mathbf{C}_i$, $\mathbf{C}_{norm,i}$, inputs $(\mathsf{x}_i, \mathsf{w}_i)$, $(\mathsf{x}_{norm,i}, \mathsf{w}_{norm,i})$, outputs y_i
Ensure: Combined results r_i

for all $i = 1, \ldots, n$ **do**
  $\mathsf{pp}_i \leftarrow \mathsf{ZKP.G}(1^\lambda)$
  $\pi_i \leftarrow \mathsf{ZKP.P}(\mathsf{pp}_i, \mathbf{C}_i, \mathsf{x}_i, \mathsf{w}_i, \mathsf{y}_i)$
  $v_i \leftarrow \mathsf{ZKP.V}(\mathsf{pp}_i, \mathbf{C}_i, \mathsf{x}_i, \mathsf{y}_i, \pi_i)$
  $\mathsf{pp}_{norm,i} \leftarrow \mathsf{ZKP.G}(1^\lambda)$
  $\pi_{norm,i} \leftarrow \mathsf{ZKP.P}(\mathsf{pp}_{norm,i}, \mathbf{C}_{norm,i}, \mathsf{x}_{norm,i}, \mathsf{w}_{norm,i}, \mathsf{y}_i)$
  $v_{norm,i} \leftarrow \mathsf{ZKP.V}(\mathsf{pp}_{norm,i}, \mathbf{C}_{norm,i}, \mathsf{x}_{norm,i}, \mathsf{y}_i, \pi_{norm,i})$
  $r_i \leftarrow (\mathsf{y}_i == \mathsf{y}_{norm,i}) \wedge v_i \wedge v_{norm,i}$
end for

3.3 Regular Circuits for Validation

To ensure the correctness of our implementation, we perform validation using regular circuits, which are simple and well-understood constructions that execute basic arithmetic operations. These types of circuits are theoretically ideal for standardization because arithmetic operations often produce redundant constraints after standardization. Optimizing these redundancies results in fewer constraints, making regular circuits a suitable baseline for evaluating whether our standardization method has been successfully implemented. For this purpose, we design three specific validation circuits: (1) a scaled binary addition tree (Circuit 1), which tests the propagation of addition operations across multiple scaled inputs; (2) a vector cross product circuit (Circuit 2), which evaluates the handling of multi-dimensional arithmetic interactions; and (3) a matrix multiplication circuit (Circuit 3), which serves as a more complex composition of linear operations. These circuits are selected because they are sufficiently small to allow detailed verification while still covering a variety of arithmetic patterns.

The validation circuits serve a dual purpose. First, they allow us to confirm that the normalization process preserves correctness: after applying normalization, all constraints of the original circuit must still be satisfied by the corresponding witness assignments. Specifically, Circuit 1 (the scaled binary addition tree) ensures that input values are correctly represented as powers of the smallest input value; within this circuit, the largest input has been set to the cube of the smallest value, providing a straightforward test of the core standardization mechanism. Circuit 2 (the vector cross product) involves interactions between multiple inputs. Since the replacement of each input with a powered variable has already been validated by Circuit 1, this circuit primarily tests whether the implementation correctly handles constraints that involve interactions between two or more inputs. Circuit 3 (the matrix multiplication circuit) extends the same principle to a larger scale, demonstrating that as computation complexity increases, the normalization and reduction process can significantly reduce the number of constraints.

Second, the validation circuits provide a controlled environment to observe the behavior and effects of the normalization technique, serving as a baseline for the reduction effect. This baseline can later be compared with that of the decision tree circuits to examine how the constraints differ and assess the effectiveness of the normalization in more complex computations.

In the workflow for these validation circuits, we omit the initial steps of model training and circuit generation used in the main experiments. Instead, we directly use the pre-constructed circuits as input to the normalization process. After normalization, we compare the resulting constraints and witness assignments to those of the original circuits to evaluate the impact of normalization. This approach isolates the normalization step, allowing us to focus specifically on its correctness and efficiency without interference from upstream model-related processes.

4 Result

4.1 Result Description

To evaluate the effectiveness of the normalization technique, we measured the total number of constraints before and after applying normalization across two categories of circuits: decision tree circuits derived from benchmark datasets, and simple arithmetic circuits. The results are summarized in Fig. 2 and Fig. 3, respectively.

The results in Fig. 2 detail the number of constraints for decision tree circuits corresponding to four benchmark datasets: Wine, Olivetti Faces (OF), Breast Cancer (BC), and Digits. In these circuits, the normalization method does not consistently reduce the number of constraints. For the Wine dataset, the total number of constraints increases slightly from 129 to 136. In the OF dataset, the constraint count remains unchanged at 199. The BC dataset shows a minor increase from 301 to 306, while the Digits dataset remains stable at 309 constraints. These observations indicate that normalization has limited effect on decision tree circuits, sometimes introducing additional constraints or leaving the total count largely unaffected.

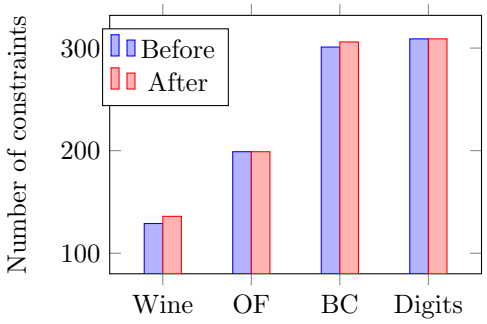

Figure 2: Constraints for DT circuits.

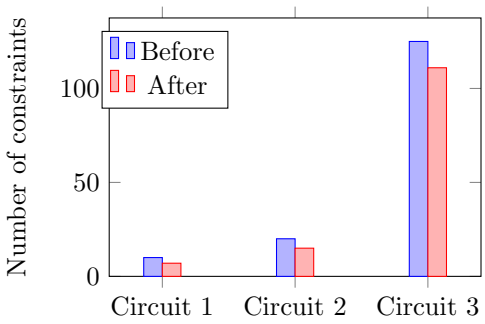

Figure 3: Constraints for regular circuits.

By contrast, Fig. 3 shows the results for three simpler arithmetic circuits: Circuit 1 (a scaled input for binary addition tree), Circuit 2 (a vector cross product), and Circuit 3 (a matrix multiplication). In these circuits, normalization consistently reduces the number of constraints. Specifically, Circuit 1 decreases from 10 to 7 constraints, Circuit 2 from 20 to 15, and Circuit 3 from 125 to 111. These reductions are more substantial in proportion compared to the minimal changes observed in decision tree circuits, illustrating that normalization is particularly effective in circuits dominated by linear arithmetic operations.

Overall, these descriptive results highlight a clear distinction between the two circuit categories. While simple arithmetic circuits exhibit consistent and meaningful constraint reductions after normalization, decision tree circuits show minimal improvement or even slight increases in constraints. This sets the stage for a detailed analysis of why the structural characteristics of decision tree circuits interact differently with the normalization process.

4.2 Result analysis

This discrepancy can be explained by the nature of operations in decision tree circuits and by the way the Circom compiler performs its optimizations. The compiler operates in two stages. First, it substitutes signals directly connected that are equivalent (e.g., a signal equal to a constant or another signal) with representative ones to reduce witness size, thereby reducing the number of variables that need to be tracked in the witness. This step simplifies the circuit structure but does not substantially affect the number of constraints on its own. Second, the compiler applies Gaussian elimination to the resulting constraint system in order to eliminate linear dependencies. This step is particularly impactful because many arithmetic constraints can be expressed in terms of one another, allowing entire sets of constraints to be collapsed into smaller equivalent systems.

The normalization method proposed by Teng and Tseng [12] directly aids this process for arithmetic circuits. By rewriting constraints such that all input signals are expressed with respect to a single chosen base variable, the method increases the structural similarity between constraints. At first glance, this increases the number of signals, since additional power terms of the base variable must be introduced. However, this apparent redundancy is precisely what allows the compiler's Gaussian elimination phase to be more effective, as it enables the removal of redundant constraints that can be expressed as linear combinations of others. the structurally aligned constraints make linear dependencies easier to detect, allowing a large portion of the introduced signals to be eliminated. As demonstrated in Fig. 3, the end result is that, despite a temporary increase in circuit complexity, the total number of constraints after optimization is significantly reduced.

In our decision tree circuits, we initially expected that the bit-decomposed constraints could be further reduced, since all values are decomposed relative to the smallest value. In principle, this should render some of the bit-decomposition consistency constraints redundant, as the resulting equations may overlap across similar decompositions. However, the situation turns out differently in practice:

the comparison logic is imported from `circomlib` (iden3), following the recommendation in the official Circom documentation. This logic implements comparisons through bit decomposition combined with Boolean operations, which transforms an integer comparison into a set of bitwise constraints. Specifically, each integer input is decomposed into its binary representation, with one signal generated for each bit. Boolean gates are then applied across these signals to realize comparison operators such as greater-than or equality. While this construction is functionally correct, it introduces constraints that are qualitatively different from the arithmetic constraints seen in other parts of the circuit.

The key issue is that these bitwise constraints are structurally disconnected from the original arithmetic values prior to decomposition. Once the integers are split into bits, the arithmetic relationships that normalization relies on no longer exist in an accessible form. Instead, the circuit contains Boolean logic enforcing conditions on individual binary signals, which cannot be expressed as simple powers of a shared base variable. As a result, the optimization process is effectively partitioned into two independent domains: (1) the arithmetic part of the circuit, where normalization aligns signals and enables Gaussian elimination to remove more redundancies, and (2) the bitwise comparison part, where constraints remain rigid and unaffected by normalization.

In practice, this partitioning undermines the intended benefits of normalization for decision tree circuits. Although the normalization procedure introduces additional signals through its power-based representation, the effect on the total number of constraints is minor. Depending on the difference between the minimum and maximum values , this representation can slightly reduce or slightly increase the total number of constraints. For decision trees with thresholds that span a small value range, such as the Olivetti Faces (OF) and Digits datasets, normalization may slightly reduce the total number of constraints. In contrast, for datasets with a larger range, such as Wine and Breast Cancer (BC), the total number of constraints may increase slightly. However, these changes are not significant. The main bottleneck remains the bitwise constraints: as shown, the optimization phase cannot further reduce the bitwise comparison constraints, which dominate the circuit size. As a result, the overall number of constraints in the normalized circuit is about the same as the unnormalized version. Rather than improving efficiency, normalization in this context introduces unnecessary overhead.

This analysis highlights an important observation regarding the normalization method. Although it is effective for circuits dominated by linear arithmetic relations, its efficiency in circuit sections involving non-linear logic, such as bit decomposition, remains uncertain. In hybrid circuits that combine linear and non-linear components, applying normalization may not provide consistent benefits and raises questions about its overall effectiveness in handling non-linear operations.

In summary, while the normalization technique is effective for circuits dominated by arithmetic relations, it is ill-suited to components that rely on bit decomposition or other forms of non-linear logic. This observation suggests that normalization is not universally applicable: in circuits combining both linear and non-linear computations, its usefulness should be carefully tested, or the method should be adapted to apply selectively, targeting only the linear components where it can provide measurable benefits.

5 Conclusions

This work investigates the effectiveness of the normalization technique on zero-knowledge proof circuits. It primarily focuses on both decision tree circuits derived from benchmark datasets and simple arithmetic circuits implemented in Circom, and verified using SnarkJS. Our experimental results indicate that normalization provides limited benefit for decision tree circuits. The primary reason for this inefficiency is the presence of comparison logic implemented via bit decomposition and Boolean operations. These bitwise constraints are structurally disconnected from the original arithmetic signals, preventing normalization from effectively collapsing or reducing them. Consequently, the overall number of constraints in decision tree circuits often remains unchanged or even increases, highlighting that normalization can introduce unnecessary overhead in circuits where non-linear logic dominates. These findings highlight a key consideration for applying normalization techniques in practice: while highly beneficial for linear arithmetic circuits, normalization is less effective or potentially counterproductive

in circuit sections involving non-linear logic, such as comparisons or other bitwise operations. Consequently, when designing hybrid circuits that combine linear and non-linear computations, normalization should be applied selectively, targeting only the arithmetic components to maximize efficiency without increasing complexity in non-linear sections.

Acknowledgements

This work was partially supported by the Taiwan National Science and Technology Council (Nos. 114-2221-E-027-109, 114-2634-F-027-001-MBK, and 113-2221-E-011-156-MY3).

References

[1] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. Snarks for c: Verifying program executions succinctly and in zero knowledge. In *Advances in Cryptology–CRYPTO 2013: 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part II*, pages 90–108. Springer, 2013.

[2] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Scalable zero knowledge via cycles of elliptic curves. *Algorithmica*, 79:1102–1160, 2017.

[3] Matteo Campanelli, Antonio Faonio, Dario Fiore, Tianyu Li, and Helger Lipmaa. Lookup arguments: Improvements, extensions and applications to zero-knowledge decision trees. In Qiang Tang and Vanessa Teague, editors, *Public-Key Cryptography – PKC 2024*, pages 337–369, Cham, 2024. Springer Nature Switzerland.

[4] Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. Quadratic span programs and succinct nizks without pcps. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 626–645. Springer, 2013.

[5] Jens 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*, pages 305–326. Springer, 2016.

[6] Helger Lipmaa. Polymath: Groth16 is not the limit. In *Annual International Cryptology Conference*, pages 170–206. Springer, 2024.

[7] Christodoulos Pappas and Dimitrios Papadopoulos. Sparrow: Space-efficient zksnark for data-parallel circuits and applications to zero-knowledge decision trees. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, CCS '24, page 3110–3124, New York, NY, USA, 2024. Association for Computing Machinery.

[8] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. *Communications of the ACM*, 59(2):103–112, 2016.

[9] Zhizhi Peng, Taotao Wang, Chonghe Zhao, Guofu Liao, Zibin Lin, Yifeng Liu, Bin Cao, Long Shi, Qing Yang, and Shengli Zhang. A survey of zero-knowledge proof based verifiable machine learning, 2025.

[10] Srinath Setty. Spartan: Efficient and general-purpose zksnarks without trusted setup. In *Annual International Cryptology Conference*, pages 704–737. Springer, 2020.

[11] Chenhao Shi, Hao Chen, Ruibang Liu, and Guoqiang Li. Data-flow-based normalization generation algorithm of r1cs for zero-knowledge proof, 2023.

[12] Wei-Chung Teng and Chun-Yi Tseng. A fast r1cs normalization method based on parameter vectors. In Hamido Fujita, Yutaka Watanobe, Moonis Ali, and Yinglin Wang, editors, *Advances and Trends in Artificial Intelligence. Theory and Applications*, pages 303–312, Singapore, 2026. Springer Nature Singapore.

[13] Riad S Wahby, Srinath Setty, Max Howald, Zuocheng Ren, Andrew J Blumberg, and Michael Walfish. Efficient ram and control flow in verifiable outsourced computation. *Cryptology ePrint Archive*, 2014.

[14] Jiaheng Zhang, Zhiyong Fang, Yupeng Zhang, and Dawn Song. Zero knowledge proofs for decision tree predictions and accuracy. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, CCS '20, page 2039–2053, New York, NY, USA, 2020. Association for Computing Machinery.