ReVEAL: GNN-Guided Reverse Engineering for Formal Verification of Optimized Multipliers

Chen Chen[1,4⋆], Daniela Kaufmann[2], Chenhui Deng[3], Zhan Song[1], Hongce Zhang[4], and Cunxi Yu[1]

[1] University of Maryland, College Park, USA
{cchen099, zhansong, cunxiyu}@umd.edu
[2] TU Wien, Austria
daniela.kaufmann@tuwien.ac.at
[3] NVIDIA
cdeng@nvidia.com
[4] Hong Kong University of Science and Technology (Guangzhou)
hongcezh@hkust-gz.edu.cn

Abstract. We present ReVEAL, a graph-learning-based method for reverse engineering of multiplier architectures to improve algebraic circuit verification techniques. Our framework leverages structural graph features and learning-driven inference to identify architecture patterns at scale, enabling robust handling of large optimized multipliers. We demonstrate applicability across diverse multiplier benchmarks and show improvements in scalability and accuracy compared to traditional rule-based approaches. The method integrates smoothly with existing verification flows and supports downstream algebraic proof strategies.

Keywords: Reverse engineering · Multiplier · Graph learning · Formal verification · Computer algebra

1 Introduction

Gate-level arithmetic circuits, in particular integer multipliers that have been optimized via logic synthesis, continue to pose significant challenges for current formal verification techniques. Despite notable advancements in computer algebra-based (CA) techniques [8,14,17,21], these methods often rely on syntactic heuristics and hence struggle with synthesized circuits due to optimizations that obscure their original word-level structure [7,17,21,32,33].

Our goal is to verify such optimized circuits by reverse-mapping them back to their original non-optimized word-level representations that can be efficiently verified using algebraic reasoning. We ensure the correctness of this mapping through Boolean satisfiability (SAT)-based equivalence checking.

⋆ This work was conducted during Chen Chen's visit to the University of Maryland, College Park.

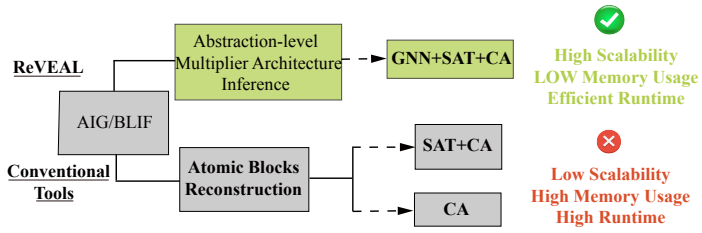


Fig. 1: Overview of ReVEAL versus Conventional Tools

Traditional computer algebra techniques reduce the task of circuit verification to an ideal membership test for the specification polynomial, typically utilizing Gröbner bases [5] to simplify the polynomials through backward rewriting. In [13,14], an incremental column-wise verification approach together with adder substitution is used in the tool AMULET2 to handle unoptimized multipliers. However, the verification of optimized multipliers still presents significant challenges due to blow-ups in the intermediate rewriting steps. In REVSCA2 [21] this is addressed by algebraic reverse engineering to recover word-level components such as full adders and half adders. Most recently, DYNPHASEORDEROPT [17] optimizes the encoding of the phases of occurring signals to enhance backward rewriting and keep the sizes of intermediate polynomials small. However, despite these advances, logic synthesis often applies aggressive optimizations that obscure the original circuit structure. As a result, many original atomic blocks cannot be restored through cut enumeration, which limits the effect of the aforementioned rewriting methods. Moreover, the inevitable occurrence of numerous OR chains and XOR gates in multiplier circuits further complicates polynomial rewriting, resulting in ongoing issues with monomial blow-ups.

Recent work by Li et al. [19] reconstructs an adder tree from optimized multipliers to build a structurally similar reference design, and then verifies it via CA and SAT-based equivalence checking. However, such reconstruction-based reverse engineering fundamentally depends on explicit structural pattern matching, which is fragile under aggressive synthesis and technology mapping. As a result, boundary recovery can become expensive or fail entirely, and the approach is often architecture-specific (e.g., tailored to adder trees), making it difficult to handle diverse designs such as 4-to-2-compressor-based and counter-based Wallace trees, XOR-heavy addition structures, or Booth-encoded multipliers where PPG/PPA boundaries are blurred.

Due to the above limitations of reconstruction-based methods [19], ReVEAL instead formulates the task as a learning-based inference problem. Traditional reverse engineering largely depends on recovering explicit boundaries and subgraph patterns, which can be broken or smeared by aggressive synthesis. In contrast, GNNs can aggregate information over neighborhoods via message passing and encode both local motifs and broader connectivity trends, allowing ReVEAL to capture more optimization-invariant cues from the netlist graph and thus remain effective even when canonical structures are not directly recoverable.

Our Contributions. In this work, we propose a novel domain-specific GNN-based reverse engineering approach for architectural identification and demonstrate its significant impact in orthogonally accelerating computer algebra-based formal verification techniques by addressing the aforementioned challenges.

The contributions of this paper are as follows:

- To the best of our knowledge, ReVEAL is the first framework to integrate machine learning (ML) techniques into an end-to-end formal verification pipeline for multiplier circuits, demonstrating significant improvements in both runtime and scalability.
- We present a novel approach that combines cone extraction with word-level functional and structural features in GNNs. This method enables the model to focus on small graphs within the critical cone, avoiding the performance degradation and long training times typically associated with large graphs.
- We compare ReVEAL with advanced (SAT+)CA-based verification tools. The results show that ReVEAL achieves a 4.90× speedup and a 19.97× reduction in memory usage compared to state-of-the-art CA tools. Additionally, ReVEAL outperforms SAT+CA methods with a 13.39× speedup.

2 Preliminaries

In this section we provide background information on different multiplier architectures (Section 2.1), a short introduction to GNNs (Section 2.2), a very brief discussion of SAT solvers and SAT sweeping (Section 2.3), as well as the problem formulation (Section 2.4).

2.1 Multiplier Architectures

Multipliers typically involve three stages: (1) *Partial Product Generator (PPG)*, (2) *Partial Product Accumulator (PPA)*, and (3) *Final Stage Adder (FSA)*.

The PPG can employ either a simple approach that pairs corresponding bits of the multiplicands to generate each partial product using logical conjunction or a Booth encoding scheme to process multiple bits at a time.

The PPA then sums these partial products using structures like Wallace trees, Dadda trees, or 4-to-2 compressor trees, which reduce gate delays and minimize sequential additions compared to traditional array accumulators.

Finally, the FSA combines the last two rows of partial products using adders, which can be broadly categorized into non-tree-based and tree-based adders (e.g., Brent–Kung, Kogge–Stone, Sklansky, Han–Carlson), with tree adders typically showing more regular, highly connected prefix structures. We list the most common multiplier architectures in Table 1.

2.2 Graph Neural Networks

We refer readers to [10] for a comprehensive introduction to graph representation learning and GNNs. A GNN encodes a graph $G = (V, E)$ into a more compact

Table 1: Categorization of Multiplier Architectures

Stage	Types	
❶ PPG	• Booth Encoding	• Simple
❷ PPA	• Simple Array	• Wallace Tree
	• Dadda Tree	• 4-to-2 Compressor Tree
	• Counter-based Wallace Tree	
❸ FSA	• Ripple Carry Adder	• Carry Look-Ahead Adder
	• Carry Skip Adder	• Serial Prefix Adder
	• Brent-Kung Adder	• Sklansky Adder
	• Han-Carlson Adder	• Ladner-Fischer Adder
	• Kogge-Stone Adder	

Note: Tree adders are marked using blue cell colors.

representation. Starting with initial node features $\{\mathbf{x}_i^{(0)} \mid i \in V\}$, GNNs iteratively update hidden state vectors $\mathbf{x}_i^{(l)}$ using:

$$\mathbf{m}_i^{(l)} = \text{AGGREGATE}^{(l)}\left(\{\mathbf{x}_j^{(l-1)} \mid j \in \mathcal{N}(i)\}\right), \tag{1}$$

$$\mathbf{x}_i^{(l)} = \text{COMBINE}^{(l)}\left(\mathbf{m}_i^{(l)}, \mathbf{x}_i^{(l-1)}\right), \tag{2}$$

where $\mathcal{N}(i)$ are the one-hop neighbors of node i. The AGGREGATE function (e.g., mean or max pooling) pools neighbor features, and the COMBINE function integrates this with the node's current feature. After L layers, the set of vectors $\{\mathbf{x}_i^{(L)} \mid i \in V\}$ capture the structural and feature information of the graph. A readout function then aggregates these into a graph embedding for classification, enabling GNNs to effectively model complex dependencies in circuit designs for various EDA applications.

Building on this capability, recent research has explored the use of GNNs for reverse engineering tasks. HOGA [9] and GAMORA [31] leverage gate-level Boolean reasoning to identify adders at the register transfer level (RTL), while the DEEPGATE family [25] utilizes Boolean circuit representation learning to facilitate SAT solving. Furthermore, FGNN [11, 29] employs graph learning techniques to identify adder blocks and classify circuits.

These prior works studied the use of GNNs in low-level atomic block identification/reconstruction, while our proposed approach ReVEAL is the first work that investigates the application of GNNs in the inference of high-level design choices (abstraction-level architectural inference).

2.3 Modern SAT Solvers and SAT Sweeping

KISSAT [3, 4] is a state-of-the-art SAT solver, which employs clausal congruence closure to efficiently handle large-scale equivalence checking problems, particularly for circuits containing numerous small isomorphic subcircuits.

SAT sweeping [35] is an efficient equivalence checking approach that combines simulation with incremental SAT solving. It detects equivalent nodes through

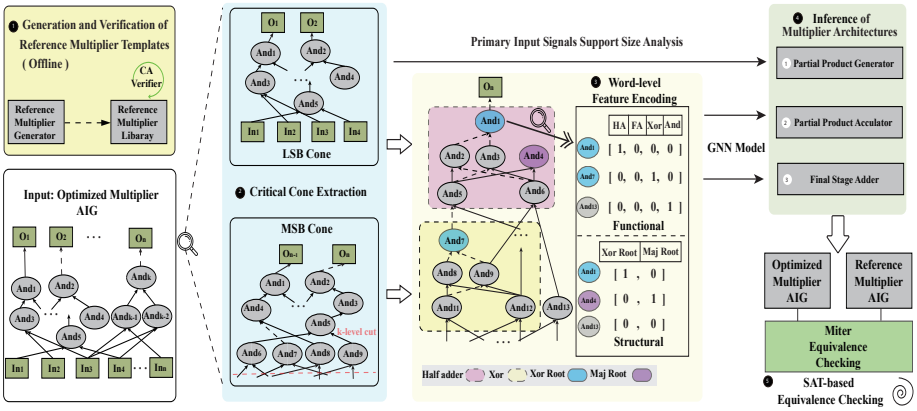


Fig. 2: The Workflow of ReVEAL

simulation, constructs intermediate miters, and runs SAT solving in topological order on two circuits to iteratively refine and merge equivalent nodes.

2.4 Problem Formulation

We denote an optimized multiplier gate-level netlist in AIG form as G with bit-width N. Table 1 summarizes the candidate structures for the three stages considered in this work. Given G, ReVEAL infers the corresponding (PPG, PPA, FSA) architecture choices from Table 1. Based on the inferred architectures and N, we select the matching pre-verified template from our library and use SAT-based combinational equivalence checking (CEC) to validate functional equivalence between G and the selected template.

3 Methodology – the ReVEAL Framework

Figure 2 presents the workflow of our reverse engineering framework **ReVEAL**, which consists of five main steps. Steps 2–4 are explained in more detail in the following sections.

❶ **Offline Generation and Verification of Reference Multiplier Templates.** We generate RTL templates for multipliers of various bitwidths and architectures using multiplier generators [20] [27]. These templates are processed with Yosys [30] to produce their corresponding And-Inverter Graphs (AIGs). All templates are formally certified using AMulet2 [14] and are collected in our *reference multiplier library*. Notably, the generation and certification processes are performed offline.

❷ **Critical Cone Extraction.** The input to ReVEAL is a gate-level multiplier $\mathcal{G}$, optimized by logic synthesis tools, such as ABC [22]. We extract the Least Significant Bit (LSB) and Most Significant Bit (MSB) cones of $\mathcal{G}$, which

allows us to perform architecture inference on smaller subcircuits and hence enhances the generalization capability of the GNN model.

❸ **Word-level Feature Encoding.** For the extracted LSB and MSB cones, we perform reverse identification of arithmetic blocks such as full adders and half adders to encode word-level features on $\mathcal{G}$.

❹ **Inference of Multiplier Architecture.** By combining graph analysis on primary input signals and GNN model inference, we predict the architecture of the optimized multiplier, identifying its PPG, PPA, FSA.

❺ **SAT-based Equivalence Checking.** Using the generated reference multiplier library of step 1, we locate a pre-verified template with the same architecture as the prediction of the optimized multiplier. A miter circuit is constructed by combining the optimized multiplier and the reference template for equivalence checking. We verify the miter using SAT solving/sweeping; see **Q2** in the experimental evaluation.

3.1 Critical Cone Extraction

Previously, GNNs struggled with long-range dependencies in large-scale graphs, leading to over-smoothing [18] and over-squashing [2]. Over-smoothing makes node representations indistinct, while over-squashing compresses excessive information into fixed-size embeddings, causing loss of critical details. These issues hinder the performance and scalability of GNNs on large graph structures.

To tackle these challenges, we partition the graph to concentrate on critical cones relevant to their corresponding primary stages of the multiplier architecture: PPG, PPA, FSA, see Figure 3 for an example. Firstly we handle the ❶ PPG and ❷ PPA stages by pre-processing the input AIG and extract the C least significant primary outputs and their corresponding input cones. In our experimental dataset (see Section 4), which starts with a minimum input bit-width of 32, we have empirically observed that the cones of the $C = 8$ LSBs contain sufficient distinct information to successfully identify the ❷ PPA, without being affected by contamination from the ❸ FSA stages. Moreover, as the bit-width increases, the cones of the first 8 LSBs trace back to 16 input bits, displaying highly similar cone structures, which are beneficial for GNNs generalization.

Regarding the ❶ PPG, we directly perform graph analysis on the cone extracted from the LSBs. For doing so, we define the support size $\mathrm{SuppSize}(X)$ for a node X as:

$$\mathrm{SuppSize}(X) = \left|\{I \in \{\text{Primary Inputs}\} : \exists\, I \rightsquigarrow X\}\right|,$$

where $\exists\, I \rightsquigarrow X$ indicates that there exists a directed path from primary input node I to node X. It recursively traverses its fan-in nodes, incrementally counting the primary inputs that influence the outputs.

We extract a graph slice where the support size of the primary input signals is 2. Since Booth multipliers involve multiple input signals for each product generator, by analyzing the fanout of primary inputs and the logic levels within the slice, we can determine the use of Booth encoding. Specifically, when the

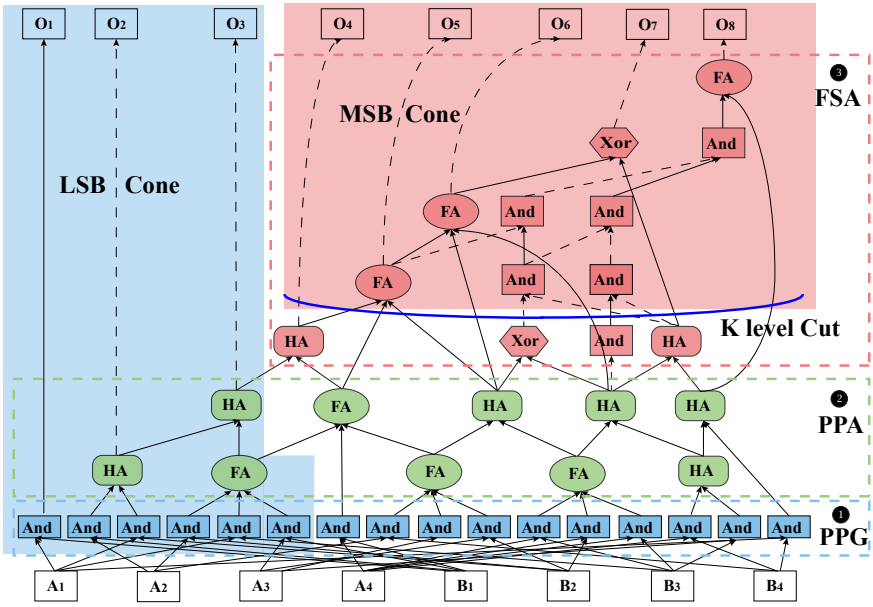


Fig. 3: Example of LSB & MSB cone extraction for a 4-bit multiplier

supporting size is set to 2 and the slice has more than 2 logic levels, it indicates that Booth encoding is being utilized. Similarly, we can also identify higher-radix multipliers by examining the output support nodes of the primary inputs, as Booth-n encoding, $n+1$ bits of the multiplier are examined, and n multiplier bits are encoded.

For the ❸ FSA stage, we begin at the MSB, denoted as M, and extract the cone spanning from the M-th bit to the $(M-L)$-th bit, where $L = 8$. This value is again empirically derived. To further isolate this cone from potential interference by the PPA and PPG stages, we implement a K-level cut of the MSB cone. The selection of K is contingent upon the multiplier bit-width N. Specifically, through empirical observation and symbolic regression on our experimental dataset, we have established a relationship between N and the corresponding logic level K required for effective graph partitioning; that is: $K = 5+2\cdot\lfloor\log_2(N)\rfloor$. This formula ensures that for varying bit-widths—from 32 to 256 bits—the K-level cut adapts appropriately, maintaining consistent cone sizes and effectively focusing only on small cones relevant to the FSA stages. Combining the dynamic K-level cut with a fixed $L = 8$ ensures that the extracted cone retains critical structural and functional information pertinent to the FSA, while keeping the cone size within a manageable limit of a few hundred nodes. This size restriction is crucial for enhancing the training speed, allowing the learning process to remain efficient and maintain generalization during inference at larger bit widths.

Table 2: Feature Embeddings for Graph Neural Network Training

Level	Type	Feature	Value Type
Node	Basic	• Input edge1 inversion	0/1
		• Input edge2 inversion	0/1
		• PI/PO/inner node	0/1
	Function	• HA node	0/1
		• FA node	0/1
		• Remaining XOR node	0/1
		• AND node	0/1
	Structure	• MAJ root	0/1
		• XOR root	0/1
Graph	Structure	• Input count	Integer
		• Gate count	Integer
		• Graph density	Float
		• Clustering Coefficient	Float

3.2 Word-Level Feature Encoding

Feature representation is key to effective message propagation in GNNs, as it determines how information is transmitted and aggregated across the graph.

Table 2 summarizes our exploration of node-level and graph-level features. In addition to retaining the basic structural feature encoding used in HOGA and GAMORA, which employs a one-hot feature vector to represent node types such as primary inputs (PI), internal nodes, and primary outputs (PO), as well as inversions on edge, we enhance basic functional node features to enable reverse mapping to word-level atomic blocks [34].

For functional atomic block identification, we employ cut enumeration [23] in the graph $\mathcal{G}$ to reverse-engineer the functional components at the RTL, identifying elements such as full adders (FA) and half adders (HA), and remaining XOR gates that are not part of FA/HAs. For readers interested in the details of identification, we provide the complete procedure in Appendix A. By detecting all XOR3 and MAJ3[5] combinations in the FAs, we pinpoint the corresponding input nodes (IN_1, IN_2, IN_3) and output nodes (*XOR3 Root*, *MAJ3 Root*). The same method applies to HAs. Through recursive traversal of the input and output nodes in each pair, we identify all AIG nodes that can be reverse-mapped to HAs and FAs. The remaining nodes that can be reverse-mapped to an XOR are annotated as XOR nodes, while the remaining AIG nodes are annotated as AND gates. In the examples shown in Figure 2, nodes such as AND1, AND7, and AND13 are each represented by a vector that indicates their respective functional blocks as HAs, FAs, XOR, and AND.

[5] XOR3 refers to the logic gate implementing the three-input parity function $x \oplus y \oplus z$; MAJ3 denotes the three-input majority function $\text{maj}(x, y, z) = (x \wedge y) \vee (x \wedge z) \vee (y \wedge z)$.

Cone extraction, as discussed in Section 3.1, enables us to work with smaller graphs, allowing for the quick computation of graph-level features such as graph density[6], clustering coefficient[7], cut input count, and gate count.

These features are crucial for identifying the sparsity of connections and cut fan-in numbers in different adder structures. Additionally, the time required for generating these functional and structural node features is negligible. This efficiency in feature computation significantly enhances our model's ability to classify different types of adders while maintaining scalability in larger designs.

3.3 Inference of Multiplier Architecture

We use a multi-task hierarchical classification strategy built upon a bidirectional GraphSAGE [28] architecture to classify and more accurately distinguish different architectures of PPAs and FSAs. Recall, that we only require inference for PPA and FSA, as PPG can already be classified during critical cone extraction.

Directional GraphSAGE Model. The model captures directional dependencies through parallel message passing, where the graph neural network aggregates features from both incoming (predecessor) and outgoing (successor) edges. The directional embedding computation at layer k is defined as:

$$\begin{aligned} \mathbf{h}_v^{\mathrm{pre}(k)} &= \mathrm{ReLU}\left(\mathbf{W}_{\mathrm{pre}}^{(k)} \cdot \mathrm{MEAN}\left(\{\mathbf{h}_u^{(k-1)} \mid u \in \mathcal{N}_{\mathrm{in}}(v)\}\right)\right) \\ \mathbf{h}_v^{\mathrm{suc}(k)} &= \mathrm{ReLU}\left(\mathbf{W}_{\mathrm{suc}}^{(k)} \cdot \mathrm{MEAN}\left(\{\mathbf{h}_u^{(k-1)} \mid u \in \mathcal{N}_{\mathrm{out}}(v)\}\right)\right) \end{aligned} \tag{3}$$

with $\mathbf{W}_{\mathrm{pre}}^{(k)}$ and $\mathbf{W}_{\mathrm{suc}}^{(k)}$ being directional message parsing weights (which are trainable) for layer k, and $\mathcal{N}_{\mathrm{in}}(v)$, $\mathcal{N}_{\mathrm{out}}(v)$ being the one-hop neighbors of node v (following either incoming or outgoing edges).

After three message-passing layers, we compute normalized node embeddings through concatenation ($\|$) and batch normalization (BN) as follows:

$$\mathbf{h}_v^{\mathrm{final}} = \mathrm{BN}\left(\mathbf{h}_v^{\mathrm{pre}(3)} \parallel \mathbf{h}_v^{\mathrm{suc}(3)}\right) \quad \forall v \in G \tag{4}$$

and then obtain the graph-level representation via global mean pooling:

$$\mathbf{h}_G = \frac{1}{|G|} \sum_{v \in G} \mathbf{h}_v^{\mathrm{final}} \tag{5}$$

[6] Graph density: fraction of present edges out of all possible ones (value range: 0–1; higher means more connections).

[7] Clustering coefficient: how tightly a node's neighbors connect to each other (higher means a tighter group).

Hierarchical Classification. Our two-stage hierarchical design addresses the error accumulation problem in a direct 9-class FSA classification (cf. Table 1) through structural decomposition. The key insight stems from the fundamental architectural differences between tree-based and non-tree-based adders: tree structures exhibit regular logarithmic-depth patterns, while non-tree implementations often contain irregular subgraph motifs. This hierarchical strategy first performs a coarse-grained binary classification to distinguish tree-based and non-tree-based adders, followed by fine-grained classification within each category to identify specific adder types. The classification pipeline proceeds as follows:

Stage 1 - Tree vs. Non-tree Discrimination

$$\hat{y}_{\text{topo}} = \arg\max\left(\mathbf{W}_{\text{topo}}[\mathbf{h}_G \parallel f_{\text{level}}]\right) \tag{6}$$

where f_{level} denotes the maximum logic depth of the optimized circuit, capturing critical structural information. This stage effectively separates tree adders, characterized by deep recursive structures, from non-tree adders, which exhibit shallow combinational designs.

Stage 2 - Fine-grained Classification Conditioned on the predicted topology (tree vs. non-tree), we perform subtype classification by combining the graph embedding with lightweight structural summaries:

$$\hat{y}_{\text{sub}} = \arg\max(\mathbf{W}_t\left[\mathbf{h}_G \parallel f_{\text{fan}} \parallel \mathbf{f}_{\text{graph}}\right]), \qquad t \in \{\text{nt}, \text{tree}\}. \tag{7}$$

Here, f_{fan} is the fan-in statistic extracted from the K-level MSB cone, and $\mathbf{f}_{\text{graph}} \in \mathbb{R}^3$ summarizes global structure (density, clustering coefficient, and average degree).

Multi-task Training Strategy. The model's trainable parameters Θ comprise three components: directional message passing weights $\mathbf{W}_{\text{pre}}^{(k)}$ and $\mathbf{W}_{\text{suc}}^{(k)}$ from Eq. 3, where $k \in \{1, 2, 3\}$ represents the three layers of message passing, along with classifier weights $\mathbf{W}_{\text{topo}}$ for topology prediction, $\mathbf{W}_{\text{nt}}$ for non-tree subtypes, and $\mathbf{W}_{\text{tree}}$ for tree subtypes. The gradient composition rule combines task-specific gradients through conditional routing:

$$\frac{\partial \mathcal{L}}{\partial \Theta} = \frac{\partial \mathcal{L}_{\text{topo}}}{\partial \Theta} + \alpha \left[I(\hat{y}_{\text{topo}} = 0)\frac{\partial \mathcal{L}_{\text{nt}}}{\partial \Theta} + I(\hat{y}_{\text{topo}} = 1)\frac{\partial \mathcal{L}_{\text{tree}}}{\partial \Theta} \right], \tag{8}$$

where $I(\cdot)$ implements conditional gradient routing based on topology prediction $\hat{y}_{\text{topo}}$, with $\mathcal{L}_{\text{topo}}$ denoting binary cross-entropy loss, $\mathcal{L}_{\text{nt}}$ for 4-class non-tree classification, and $\mathcal{L}_{\text{tree}}$ for 5-class tree classification. The balancing coefficient $\alpha = 3.2$ is calibrated during initial warm-up training through gradient variance analysis. This process computes the Frobenius norms' squared magnitudes $\|\nabla_\Theta \mathcal{L}_{\text{topo}}\|_F^2$ and $\|\nabla_\Theta \mathcal{L}_{\text{sub}}\|_F^2$ over 100 iterations, where $\mathcal{L}_{\text{sub}} = \mathcal{L}_{\text{nt}} + \mathcal{L}_{\text{tree}}$.

Statistical aggregation calculates the running averages $\mathbb{E}[\cdot] = \frac{1}{100}\sum_{t=1}^{100}(\cdot)_t$, determining the optimal coefficient α via:

$$\alpha = \sqrt{\frac{\mathbb{E}[\|\nabla_\Theta \mathcal{L}_{\text{topo}}\|_F^2]}{\mathbb{E}[\|\nabla_\Theta \mathcal{L}_{\text{sub}}\|_F^2]}} \approx 3.2, \tag{9}$$

where $\|\cdot\|_F$ denotes the Frobenius norm, and $\|\cdot\|_F^2$ represents its squared magnitude, which is used here to measure the gradient strength for balancing tasks.

Parameter updates follow distinct pathways: message passing weights $\mathbf{W}_{\text{pre/suc}}$ receive combined gradients from both tasks, while $\mathbf{W}_{\text{topo}}$ updates exclusively through $\mathcal{L}_{\text{topo}}$. The subtype classifiers $\mathbf{W}_{\text{nt}}$ and $\mathbf{W}_{\text{tree}}$ activate only when $\hat{y}_{\text{topo}}$ predicts non-tree (0) or tree (1) topologies, respectively. The unified update rule preserves batch-normalized stability (Eq. 4):

$$\Theta^{(t+1)} = \Theta^{(t)} - \eta\left(\nabla_\Theta \mathcal{L}_{\text{topo}} + \alpha \nabla_\Theta \mathcal{L}_{\text{sub}}\right). \tag{10}$$

The proposed hierarchical classification framework, coupled with the multi-task training strategy, effectively mitigates error accumulation and ensures robust performance by leveraging topology-aware feature decomposition and gradient-balanced optimization.

4 Experiments

Experimental Setup. All experiments are run on a server running Ubuntu 20.04.4 LTS, with Intel(R) Xeon(R) Gold 6418H processors and 64 GB of memory. We set a timeout (TO) of 3600 seconds and a memory limit (MO) of 8 GB.

All multiplier circuits are converted to AIG format using Yosys [30] and ABC [22]. All generated reference multipliers are verified using AMULET2 [14] with 64 pre-prepared templates per bit-width (step 1). The algorithms for critical cone extraction (step 2) and word-level feature encoding (step 3) are implemented and integrated in ABC. For miter equivalence checking (step 5) we consider the SAT solver KISSAT 4.0.1 [3] and ABC's SAT-sweeping-based combinational equivalence checkers `&fraig`, `cec` and `&cec`; the rationale for this solver combination is analyzed in **Q2**.

Dataset. We use the multipliers generated by the generators Multgen [27] and GenMul [20][8]. GenMul provides 28 distinct architectures, while Multgen supports 36 architectures. We include all available architectures with one exception: for Multgen, we exclude Booth-encoding templates at bit-widths of 4 and above, because these instances cannot be correctly parsed by Yosys during SystemVerilog-to-AIG conversion.

[8] We do not consider the richer Aoki benchmark set [12] in our analysis due to the discontinuation of their online generator, which renders it impossible to obtain the small-width cases necessary for model training.

Table 3: Inference Accuracy for Each Stage

Stage	TOP 1*	TOP 2*	TOP 3*
Stage PPG	100%	–	–
Stage PPA	100%	–	–
Stage FSA	85%	97%	98%

* TOP 1, TOP 2, and TOP 3 represent the prediction accuracy when the ground truth is among the model's top 1, top 2, and top 3 ranked predictions, respectively.

To build the training and evaluation sets, we use even-numbered bit-widths from 32 to 60 bits for training and evaluate on unseen 64-, 128-, and 256-bit designs. Following prior work [17, 21], all optimized multipliers are produced using ABC operators: `dc2` or `resyn3`. All selected designs—both original and optimized—are normalized to AIG using Yosys [30] and ABC [22], ensuring consistent representation across generators, optimizations, and bit-widths. This design enables a rigorous assessment of the model's ability to generalize to unseen, larger bit-widths, beyond the training regimes.

Machine Learning Configuration. We train our model for 200 epochs per optimization operator. All hyperparameters (e.g., learning rate, batch size, and weight decay) are automatically tuned using Optuna [1]; the final configurations and trained model checkpoints are provided in our open-source repository. Our pipeline demonstrates exceptional efficiency: cone extraction plus word-level feature extraction complete in under 1 second per case. A complete training cycle consists of 329 seconds for feature extraction and 321 seconds for training, totaling 650 seconds. Our training scales well as bit-widths grow thanks to critical cone extraction. In contrast, conventional full-graph training approaches, which perform equivalent computations on full AIGs without cone extraction, are 182× slower.

Research Questions. We organize the presentation of experimental results to address the following research questions:

Q1: *Can GNNs and analytical techniques demonstrate robust performance and generalization capability for architectural inference across optimized netlists?*

The inference accuracy is shown in Table 3. We also summarize, for the 64-bit dc2-optimized case, all top-3 predictions in Appendix B. In *Stage PPA* we observe that the LSBs cone maintains similar cone graph sizes and structural similarities across different bit-widths, enhancing the model's learning capability for larger bit-widths. As a result, our training process achieves an accuracy rate of 100% for the first two stages. Furthermore, our model demonstrates high accuracy in *Stage FSA*, with the top three predictions nearly achieving complete correctness. We observe that when the circuits have been optimized using `dc2`, the FSA prediction errors mainly appear on Brent–Kung (BK) cases. This happens because our MSB-centered cone extraction is only an approximate par-

Table 4: Solver Performance for Miter Solving

Opt	dc2			resyn3		
	Solved	Avg Time (s)	#. fastest	Solved	Avg Time (s)	#. fastest
cec	60/64	692.57	0	60/64	1007.20	1
&cec	64/64	8.50	46	64/64	6.83	50
&fraig -y	36/64	1848.53	12	39/64	1877.33	8
&fraig -x	37/64	1765.14	0	39/64	1726.71	5
kissat	64/64	96.79	6	64/64	186.55	0

tition. In some extensively optimized cases, we observe that structurally similar but not identical templates of the FSA can facilitate faster equivalence checking. Therefore, we retain the top three predicted templates and have them participate in SAT solving in parallel.

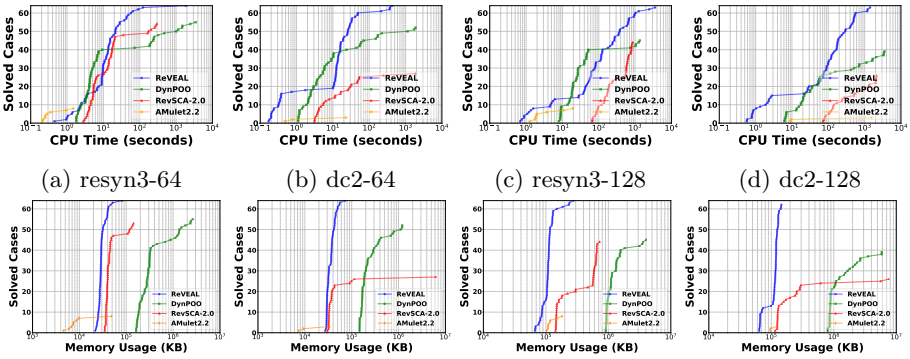
Q2: *Are modern SAT solvers and SAT sweeping techniques efficient for equivalence checking between the optimized and the golden multiplier?*

We selected the SAT-sweeping tools mentioned in the experimental setup for equivalence checking of optimized (synthesized) and original (golden) circuits. We tested 64-bit multipliers with 64 different templates. Our quantitative analysis in Table 4 reveals that while `&cec` achieves the highest success rate, Kissat, `&fraig -y` and `&cec` remain essential for handling specific corner cases. Notably, the SAT-sweeping approach demonstrates remarkable memory efficiency compared to computer algebra-based methods — a critical advantage when verifying complex arithmetic circuits, which we will quantitatively validate in terms of memory consumption later for **Q3**. Based on these observations, we implement a hybrid parallel verification framework that concurrently executes three strategically selected solvers (`&fraig -y`, `&cec`, and Kissat), terminating the verification process immediately when any solver returns a conclusive result.

Q3: *Is architecture identification helpful for computer-algebra-based multiplier verification?*

Comparison with State-of-the-Art CA Verification Tools. We systematically test all 64 possible template combinations and successfully verify equivalence between the top three predicted templates and the circuit under test. For a fair comparison as in [17, 21], we use the same optimizations, `dc2` and `resyn3` in ABC. Experimental results in Table 5 and Figure 4 show that ReVEAL successfully solves nearly all benchmarks, except one limited by excessive SAT-solving time. In general, minor errors in the FSA stage prediction are tolerable—SAT solving and sweeping remain highly effective even across structurally different FSA templates, providing a robustness cushion against FSA misclassifications.

Specifically, we highlight the comparison of verification times and memory usage across various tools in Figure 4. We include per-case solving details in Appendix C. In general, existing CA tools suffer from vanishing monomial computations, which make them inefficient for handling large optimized circuits. On



(a) resyn3-64 (b) dc2-64 (c) resyn3-128 (d) dc2-128

(e) resyn3-64 (f) dc2-64 (g) resyn3-128 (h) dc2-128

Fig. 4: Runtime and memory usage across different "optimization-bitwidths".

average, our method runs in 174.34 seconds using 75.01 MB memory, while DynPhaseOrderOpt [17] requires 936.40 seconds and 1,374.40 MB. This demonstrates a 5.37× speed improvement and 18.33× memory reduction in favor of our approach. As can be seen in Table 5, the tool RevSCA2 reports correct circuits as "buggy". This is because RevSCA2 has bugs in reversely identifying functional arithmetic blocks, which cause the circuit to become inequivalent to the original circuit.

Impact of Misidentification on Verification Runtime. As an illustration, we run the following experiment in Table 6. We fix a 64-bit test multiplier that is optimized with resyn3 (column "Test circuit") and use equivalence checking to compare it against multipliers of different architectures. In Table 6(a), we resemble the situation when FSA (stage 3) is mispredicted, and Table 6(b) mimics the case when PPA (stage 2) is mispredicted. Note that not all FSA misprediction results in long runtime (see case 6 and 7). Despite that PPA misprediction leads to excessively long solving time, as shown by the earlier experiment in Table 3, our model achieves a high accuracy in PPG and PPA prediction and this helps avoid the long equivalence checking time.

Buggy-circuit robustness. We also evaluate ReVEAL on buggy designs. For each optimized 64-bit AIG, we create buggy variants by rewiring the fanins of three internal nodes (preserving acyclicity) and resample until random simulation finds a mismatch at some primary output. We then run SAT-based CEC against the corresponding template. ABC's `cec` takes 0.05 s on average (max 1.98 s), and all instances are `SAT` (non-equivalence detected).

Certification. Once ReVEAL concludes equivalence, we can provide proof certificates for both verification steps: (i) each unoptimized template in our library is certified by AMulet2 via an algebraic proof certificate (a sequence of polynomial additions and multiplications) [15]; and (ii) SAT-based CEC using Kissat

Table 5: The number of solvable cases for different tools across benchmarks

Tool	Result	Benchmark			
		64bit		128bit	
		dc2	resyn3	dc2	resyn3
AMulet-2.2	Solved	3/64	8/64	3/64	8/64
	TO	61	55	61	56
	MO	0	1	0	0
	"Circuit buggy"	0	0	0	0
RevSCA-2.0	Solved	26/64	54/64	26/64	44/64
	TO	7	3	7	8
	MO	21	7	29	7
	"Circuit buggy"	2	0	2	5
DynPOO	Solved	52/64	55/64	39/64	45/64
	TO	12	9	22	7
	MO	0	0	3	12
	"Circuit buggy"	0	0	0	0
ReVEAL	Solved	**64/64**	**64/64**	**64/64**	**63/64**
	TO	0	0	0	1
	MO	0	0	0	0
	"Circuit buggy"	0	0	0	0
	Stage FSA acc. (%)	96.88	100.00	92.19	100.00

Table 6: Effect of missclassifications.

#	Test circuit	Template	Runtime
1	SP_AR_BK	SP_AR_BK	3.04
2	SP_AR_BK	SP_AR_CK	3.37
3	SP_AR_BK	SP_AR_CL	5083.42
4	SP_AR_BK	SP_AR_KS	232.18
5	SP_AR_BK	SP_AR_LF	33.46
6	SP_AR_BK	SP_AR_RC	3.37
7	SP_AR_BK	SP_AR_SE	3.15

(a) Missclassified FSA.

#	Test circuit	Template	Runtime
1	SP_DT_BK	SP_DT_BK	3.8
2	SP_DT_BK	SP_AR_BK	TO
3	SP_DT_BK	SP_CWT_BK	TO
4	SP_DT_BK	SP_WT_BK	TO

(b) Missclassified PPA.

outputs a DRUP clausal proof certificate. Both certificate types can be checked by certified proof checkers [15, 26].

Comparison with CA+SAT Methods on Extensively Optimized Multipliers. In Table 7, we further evaluate extensively optimized[9] multipliers, comparing ReVEAL with RefSCAT [19], a method that combines SAT and CA by reconstructing complete references through adder boundary detection using a constraint satisfaction algorithm. For training and testing, we use the same settings as above in Section 4. Training uses even bit-width, ranging from 32 to 60 bits and testing uses unseen 64, 128, and 256-bit designs. Since RefSCAT is not publicly available, we use the experimental data from [19] for comparison. In terms of runtime, ReVEAL outperforms RefSCAT by a factor of 13.39, despite one corner

[9] The multipliers are optimized using the following sequence in ABC, applied twice: `&mfs: logic; mfs2 -W 20; mfs; st; dc2 -l; resub -l -K 16 -N 3 -w 100; logic; mfs2 -W 20; mfs; st; resyn -l; resyn; resyn2; resyn3; dc2 -l`

Table 7: Time Comparison between RefSCAT and ReVEAL.

Benchmark	**RefSCAT**	**ReVEAL**				
	Time (s)	CE (s)	FE (s)	Inference (s)	EQ (s)	Total Time (s)
128_128_U_SP_DT_HCA	2518.8	0.51	0.10	0.01	40.13	**40.74**
128_128_U_SP_WT_HCA	1791.1	0.49	0.11	0.01	42.69	**43.29**
128_128_U_SP_DT_CK	144.1	0.49	0.10	0.01	24.84	**25.44**
128_128_U_SP_DT_KS	156.4	0.56	0.08	0.01	29.00	**29.65**
128_128_U_SP_WT_CK	403.2	0.54	0.10	0.01	108.51	**109.16**
128_128_U_SP_WT_KS	422.1	0.51	0.09	0.01	54.11	**54.71**
128_128_U_SP_WT_LF	1889.2	0.49	0.11	0.01	41.21	**41.83**
128_128_U_SP_DT_CL	127.2	0.50	0.08	0.01	211.07	211.66

* CE: Cone Extraction; FE: Feature Extraction; EQ: Equivalence Checking;

case slowed down by SAT sweeping time. The results show that ReVEAL's time overhead is almost entirely spent on equivalence checking, while the preprocessing phase takes less than 1 second. In contrast, RefSCAT uses significant time on adder boundary detection, particularly in highly optimized cases. Notably, RefSCAT currently cannot verify Booth-encoded or non-adder-based multipliers, requiring online RTL multiplier reconstruction that introduces RTL to AIG conversion overhead and online computer-algebraic verification runtime. Our approach, on the other hand, is faster and more versatile for reverse engineering and architectural inference.

Q4: *Are the proposed cone extraction, feature extraction, and neural network models domain-specifically helpful in the task?*

To evaluate the contribution of each technique in ReVEAL, we conduct an ablation study by selectively removing key components of our approach. The results are shown in Table 8. First, in *ReVEAL w/o cone extraction*, we train the model on the entire large graph without applying cone extraction. This leads to a significant performance decline during both *Stage PPA* and *Stage FSA* inference, primarily due to the over-squeezing and over-smoothing problems that GNNs face when operating on large graphs, demonstrating the necessity of cone extraction for efficient processing. Second, in *ReVEAL w/o word-level features*, we replace our word-level features with the basic functional AIG features used in HOGA and GAMORA. This results in a decrease in accuracy, indicating that word-level feature extraction contributes positively to the model's performance. Last, we conduct a comprehensive comparison of our approach with existing mainstream *GNN models*. Our analysis reveals that models lacking hierarchical classification exhibited higher misclassification rates for both non-tree and tree adders. In contrast, ReVEAL, which integrates hierarchical classification with bidirectional GraphSAGE, significantly reduces misclassification and enhances overall classification accuracy.

Table 8: Ablation Study and Performance Comparison of Different Methods.

	Method	Stage	Accuracy (%) TOP 1	TOP 2	TOP 3
Base	**ReVEAL**	Stage PPA	**1.0**	/	/
	ReVEAL	Stage FSA	**0.85**	**0.97**	**0.98**
w/o CE*	ReVEAL (w/o CE*)	Stage PPA	0.15	/	/
	ReVEAL (w/o CE*)	Stage FSA	0.16	0.28	0.40
w/o WF*	ReVEAL (w/o WF*)	Stage PPA	**1.0**	/	/
	ReVEAL (w/o WF*)	Stage FSA	0.76	0.94	0.95
GNN Model	GCN	Stage FSA	0.72	0.9	0.94
	GAT		0.75	0.91	0.95
	GraphSAGE		0.80	0.96	0.97
	SagePooling [16]		0.71	0.91	0.92
	HOGA [9]		0.79	0.92	0.94
	Graph Transformer [24]		0.64	0.86	0.91

* CE: Cone Extraction; WF: Word Feature Extraction

5 Conclusion and Future Work

In this paper, we have introduced ReVEAL, the first GNN-based framework for architectural reasoning in the reverse engineering of optimized multipliers. ReVEAL integrates critical cone extraction, word-level feature encoding, and a hierarchical multi-task classification model to achieve strong generalizability and high accuracy. Training scales well when bit-widths grow because critical cone extraction fixes the learning scope to small stage-specific subgraphs. By combining GNN-based reverse engineering with modern SAT solvers and computer algebra verifiers, ReVEAL effectively overcomes the performance bottlenecks inherent in traditional CA and CA+SAT-based verification methods for optimized multiplier verification. A future extension for the work includes applying the critical-cone learning paradigm to MAC units and other more complex datapath circuits.

Acknowledgments. This research was supported by U.S. National Science Foundation, under awards CCF-2229562, CCF-2403134, CCF-2349461, and CCF-2349670. This research was also supported by the Austrian Science Fund (FWF) [10.55776/ESP666].

Code and Data Availability Statement The source code that supports the contributions of this work is freely available at https://github.com/chestercc1997/Re_VEAL. The complete dataset and reproducible models are available at https://huggingface.co/datasets/SeaSkysz/reveal. The authors confirm that the data supporting the findings of this study are available within the paper and the artifact.

Disclosure of Interests The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Akiba, T., Sano, S., Yanase, T., Ohta, T., Koyama, M.: Optuna: A next-generation hyperparameter optimization framework. In: The 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining. pp. 2623–2631 (2019)
2. Alon, U., Yahav, E.: On the bottleneck of graph neural networks and its practical implications. In: 9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021. OpenReview.net (2021), https://openreview.net/forum?id=i80OPhOCVH2
3. Biere, A., Fazekas, K., Fleury, M., Froleyks, N.: Clausal congruence closure. In: 27th International Conference on Theory and Applications of Satisfiability Testing (SAT 2024). Schloss Dagstuhl–Leibniz-Zentrum für Informatik (2024)
4. Biere, A., Fazekas, K., Fleury, M., Froleyks, N.: Clausal equivalence sweeping. In: FMCAD 2024. pp. 236–241. TU Wien Academic Press (2024)
5. Buchberger, B.: Ein Algorithmus zum Auffinden der Basiselemente des Restklassenringes nach einem nulldimensionalen Polynomideal. Ph.D. thesis, University of Innsbruck (1965)
6. Chatterjee, S.: On algorithms for technology mapping. University of California, Berkeley (2007)
7. Ciesielski, M., Su, T., Yasin, A., Yu, C.: Understanding algebraic rewriting for arithmetic circuit verification: a bit-flow model. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **39**(6), 1346–1357 (2019)
8. Ciesielski, M., Yu, C., Brown, W., Liu, D., Rossi, A.: Verification of gate-level arithmetic circuits by function extraction. In: Proceedings of the 52nd Annual Design Automation Conference. pp. 1–6 (2015)
9. Deng, C., Yue, Z., Yu, C., Sarar, G., Carey, R., Jain, R., Zhang, Z.: Less is more: Hop-wise graph attention for scalable and generalizable learning on circuits. In: Proceedings of the 61st ACM/IEEE Design Automation Conference. pp. 1–6 (2024)
10. Hamilton, W.L.: Graph representation learning. Morgan & Claypool Publishers (2020)
11. He, Z., Wang, Z., Bai, C., Yang, H., Yu, B.: Graph learning-based arithmetic block identification. In: 2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD). pp. 1–8. IEEE (2021)
12. Homma, N., Watanabe, Y., Aoki, T., Higuchi, T.: Formal Design of Arithmetic Circuits Based on Arithmetic Description Language. IEICE Trans. Fundam. Electron. Commun. Comput. Sci. **89-A**(12), 3500–3509 (2006). https://doi.org/10.1093/IETFEC/E89-A.12.3500
13. Kaufmann, D., Biere, A.: Amulet 2.0 for verifying multiplier circuits. In: TACAS (2). LNCS, vol. 12652, pp. 357–364. Springer (2021). https://doi.org/10.1007/978-3-030-72013-1_19
14. Kaufmann, D., Biere, A.: Improving amulet2 for verifying multiplier circuits using sat solving and computer algebra. International Journal on Software Tools for Technology Transfer **25**(2), 133–144 (2023)
15. Kaufmann, D., Fleury, M., Biere, A., Kauers, M.: Practical algebraic calculus and Nullstellensatz with the checkers Pacheck and Pastèque and Nuss-Checker. Formal Methods in System Design (2022). https://doi.org/10.1007/s10703-022-00391-x
16. Knyazev, B., Taylor, G.W., Amer, M.R.: Understanding attention and generalization in graph neural networks. In: Wallach, H.M., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E.B., Garnett, R. (eds.) Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information

Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada. pp. 4204–4214 (2019), https://proceedings.neurips.cc/paper/2019/hash/4c5bcfec8584af0d967f1ab10179ca4b-Abstract.html
17. Konrad, A., Scholl, C.: Symbolic computer algebra for multipliers revisited-it's all about orders and phases. In: FMCAD 2024. pp. 261–271. TU Wien Academic Press (2024)
18. Li, Q., Han, Z., Wu, X.: Deeper insights into graph convolutional networks for semi-supervised learning. In: McIlraith, S.A., Weinberger, K.Q. (eds.) Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018. pp. 3538–3545. AAAI Press (2018). https://doi.org/10.1609/AAAI.V32I1.11604, https://doi.org/10.1609/aaai.v32i1.11604
19. Li, R., Li, L., Yu, H., Fujita, M., Jiang, W., Ha, Y.: Refscat: Formal verification of logic-optimized multipliers via automated reference multiplier generation and sca-sat synergy. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (2024)
20. Mahzoon, A., Große, D., Drechsler, R.: Genmul: Generating architecturally complex multipliers to challenge formal verification tools. In: Recent Findings in Boolean Techniques: Selected Papers from the 14th International Workshop on Boolean Problems (2021), https://api.semanticscholar.org/CorpusID:235831211
21. Mahzoon, A., Große, D., Drechsler, R.: Revsca-2.0: Sca-based formal verification of nontrivial multipliers using reverse engineering and local vanishing removal. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **41**(5), 1573–1586 (2021)
22. Mishchenko, A., Chatterjee, S., Brayton, R.K.: Dag-aware AIG rewriting a fresh look at combinational logic synthesis. In: Sentovich, E. (ed.) Proceedings of the 43rd Design Automation Conference, DAC 2006, San Francisco, CA, USA, July 24-28, 2006. pp. 532–535. ACM (2006). https://doi.org/10.1145/1146909.1147048, https://doi.org/10.1145/1146909.1147048
23. Pan, P., Lin, C.: A new retiming-based technology mapping algorithm for lut-based fpgas. In: Cong, J., Kaptanoglu, S. (eds.) Proceedings of the 1998 ACM/SIGDA Sixth International Symposium on Field Programmable Gate Arrays, FPGA 1998, Monterey, CA, USA, February 22-24, 1998. pp. 35–42. ACM (1998). https://doi.org/10.1145/275107.275118, https://doi.org/10.1145/275107.275118
24. Shi, Y., Huang, Z., Feng, S., Zhong, H., Wang, W., Sun, Y.: Masked label prediction: Unified message passing model for semi-supervised classification. In: Zhou, Z. (ed.) Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI 2021, Virtual Event / Montreal, Canada, 19-27 August 2021. pp. 1548–1554. ijcai.org (2021). https://doi.org/10.24963/IJCAI.2021/214, https://doi.org/10.24963/ijcai.2021/214
25. Shi, Z., Zheng, Z., Khan, S., Zhong, J., Li, M., Xu, Q.: Deepgate3: towards scalable circuit representation learning. arXiv preprint arXiv:2407.11095 (2024)
26. Tan, Y.K., Heule, M.J.H., Myreen, M.O.: Verified propagation redundancy and compositional UNSAT checking in cakeml. Int. J. Softw. Tools Technol. Transf. **25**(2), 167–184 (2023)
27. Temel, M., Slobodová, A., Hunt, W.A.: Automated and scalable verification of integer multipliers. In: Lahiri, S.K., Wang, C. (eds.) Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part I. Lecture Notes in Computer Science, vol. 12224, pp.

485–507. Springer (2020). https://doi.org/10.1007/978-3-030-53288-8_23, https://doi.org/10.1007/978-3-030-53288-8_23
28. Ustun, E., Deng, C., Pal, D., Li, Z., Zhang, Z.: Accurate operation delay prediction for FPGA HLS using graph neural networks. In: IEEE/ACM International Conference On Computer Aided Design, ICCAD 2020, San Diego, CA, USA, November 2-5, 2020. pp. 87:1–87:9. IEEE (2020). https://doi.org/10.1145/3400302.3415657, https://doi.org/10.1145/3400302.3415657
29. Wang, Z., Bai, C., He, Z., Zhang, G., Xu, Q., Ho, T.Y., Huang, Y., Yu, B.: Fgnn2: A powerful pre-training framework for learning the logic functionality of circuits. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (2024)
30. Wolf, C.: Yosys open synthesis suite. https://yosyshq.net/yosys/
31. Wu, N., Li, Y., Hao, C., Dai, S., Yu, C., Xie, Y.: Gamora: Graph learning based symbolic reasoning for large-scale boolean networks. In: 2023 60th ACM/IEEE Design Automation Conference (DAC). pp. 1–6. IEEE (2023)
32. Yu, C., Brown, W., Liu, D., Rossi, A., Ciesielski, M.: Formal verification of arithmetic circuits by function extraction. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **35**(12), 2131–2142 (2016)
33. Yu, C., Ciesielski, M., Mishchenko, A.: Fast algebraic rewriting based on and-inverter graphs. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **37**(9), 1907–1911 (2017)
34. Yu, C., Ciesielski, M.J.: Automatic word-level abstraction of datapath. In: IEEE International Symposium on Circuits and Systems, ISCAS 2016, Montréal, QC, Canada, May 22-25, 2016. pp. 1718–1721. IEEE (2016). https://doi.org/10.1109/ISCAS.2016.7538899, https://doi.org/10.1109/ISCAS.2016.7538899
35. Zhang, H., Jiang, J.R., Amarù, L.G., Mishchenko, A., Brayton, R.K.: Deep integration of circuit simulator and SAT solver. In: 58th ACM/IEEE Design Automation Conference, DAC 2021, San Francisco, CA, USA, December 5-9, 2021. pp. 877–882. IEEE (2021)

A Word-Level Arithmetic Block Identification via Cut Enumeration

To enable word-level feature encoding on an optimized AIG, we first perform reverse identification of small arithmetic primitives (XOR/MAJ, half adders, and full adders) using *cut enumeration*. We briefly recall the notions of k-feasible cuts and cut enumeration in the AIG setting. For readers interested in further algorithmic details, we refer to Section 3 (*Cut Computation*) of [6]; moreover, the ABC codebase provides additional practical examples of arithmetic reverse engineering built on cut enumeration (see the `acec` directory).

Cuts and k-feasible cuts in an AIG. Let G be an AIG and let v be a node in G. A *cut* c of v is a set of nodes in the transitive fan-in of v such that every path from any primary input to v passes through at least one node in c. A cut is *irredundant* if none of its proper subsets is also a cut of v. A cut c is a *k-feasible cut* if it is irredundant and $|c| \leq k$. *Cut enumeration* refers to enumerating all k-feasible cuts for each node, which provides a compact way to represent candidate local subcircuits rooted at v.

Truth-table extraction and Boolean-function matching. For each node v, we enumerate its k-feasible cuts and compute the truth table of the corresponding cut function. We maintain an offline precomputed database that stores the set of truth tables that are *NPN-equivalent* to target Boolean primitives, in particular XOR and MAJ functions (e.g., XOR2/XOR3 and MAJ2/MAJ3), where NPN-equivalence accounts for input negations, input permutations, and optional output negation. A cut is classified as an XOR (resp. MAJ) candidate if its truth table matches any entry in the corresponding NPN-class database.

Recovering half adders and full adders. After collecting all XOR and MAJ candidates, we identify half adders (HA) and full adders (FA) by pairing compatible candidates that share the same input nodes. Concretely, an HA is detected by a pair consisting of an XOR2 node (sum) and an MAJ2-equivalent node (carry) that share the same two inputs. An FA is detected by a pair consisting of an XOR3 node (sum) and an MAJ3 node (carry) that share the same three inputs. For each proposed (sum, carry) pair, we additionally validate that, after an appropriate NPN transformation, the pair implements the canonical HA/FA truth-table specification. Nodes covered by validated pairs are annotated as HA/FA blocks; remaining matched nodes are annotated as standalone XOR (or MAJ) primitives and used as word-level features in the subsequent GNN inference.

B Details of GNN Predictions

In this appendix, we would like to provide the details of GNN predictions given all available templates. The architecture is denoted by the following format "FirstStage_SecondStage_LastStage" representing the choices for PPG, PPA

and FSA, respectively. The multipliers are 64-bit wide optimized with dc2. As can be seen from Table 9, there is little challenge to predict PPG and PPA, while occasionally, FSA may be mispredicted.

Table 9: Top-3 predictions for dc2-optimized circuits

#	Tested dc2-opt circuit	rank-1 prediction	rank-2 prediction	rank-3 prediction
1	SP_4to2_BK	SP_4to2_BK	SP_4to2_LF	SP_4to2_JCA
2	SP_4to2_HCA	SP_4to2_HCA	SP_4to2_BK	SP_4to2_LF
3	SP_4to2_JCA	SP_4to2_LF	SP_4to2_JCA	SP_4to2_KS
4	SP_4to2_KS	SP_4to2_KS	SP_4to2_HCA	SP_4to2_LF
5	SP_4to2_LF	SP_4to2_LF	SP_4to2_JCA	SP_4to2_KS
6	SP_4to2_RC	SP_4to2_RC	SP_4to2_CK	SP_4to2_CL
7	SP_DT_BK	SP_DT_BK	SP_DT_LF	SP_DT_JCA
8	SP_DT_HCA	SP_DT_HCA	SP_DT_KS	SP_DT_LF
9	SP_DT_JCA	SP_DT_LF	SP_DT_JCA	SP_DT_HCA
10	SP_DT_KS	SP_DT_KS	SP_DT_HCA	SP_DT_LF
11	SP_DT_LF	SP_DT_LF	SP_DT_JCA	SP_DT_HCA
12	SP_DT_RC	SP_DT_RC	SP_DT_CK	SP_DT_CL
13	SP_WT_BK	SP_WT_BK	SP_WT_LF	SP_WT_JCA
14	SP_WT_HCA	SP_WT_HCA	SP_WT_BK	SP_WT_LF
15	SP_WT_JCA	SP_WT_LF	SP_WT_JCA	SP_WT_BK
16	SP_WT_KS	SP_WT_KS	SP_WT_HCA	SP_WT_RC
17	SP_WT_LF	SP_WT_LF	SP_WT_JCA	SP_WT_BK
18	SP_WT_RC	SP_WT_RC	SP_WT_CK	SP_WT_CL
19	BP_4to2_BK	BP_4to2_BK	BP_4to2_LF	BP_4to2_JCA
20	BP_4to2_HCA	BP_4to2_HCA	BP_4to2_KS	BP_4to2_BK
21	BP_4to2_JCA	BP_4to2_LF	BP_4to2_JCA	BP_4to2_KS
22	BP_4to2_KS	BP_4to2_KS	BP_4to2_HCA	BP_4to2_LF
23	BP_4to2_LF	BP_4to2_LF	BP_4to2_JCA	BP_4to2_KS
24	BP_4to2_RC	BP_4to2_RC	BP_4to2_CK	BP_4to2_CL
25	BP_DT_BK	BP_DT_BK	BP_DT_LF	BP_DT_JCA
26	BP_DT_HCA	BP_DT_HCA	BP_DT_BK	BP_DT_LF
27	BP_DT_JCA	BP_DT_LF	BP_DT_JCA	BP_DT_KS
28	BP_DT_KS	BP_DT_KS	BP_DT_HCA	BP_DT_LF

#	Tested dc2-opt circuit	rank-1 prediction	rank-2 prediction	rank-3 prediction
29	BP_DT_LF	BP_DT_LF	BP_DT_JCA	BP_DT_KS
30	BP_DT_RC	BP_DT_RC	BP_DT_CK	BP_DT_CL
31	BP_WT_BK	BP_WT_BK	BP_WT_LF	BP_WT_HCA
32	BP_WT_HCA	BP_WT_HCA	BP_WT_LF	BP_WT_BK
33	BP_WT_JCA	BP_WT_LF	BP_WT_JCA	BP_WT_BK
34	BP_WT_KS	BP_WT_KS	BP_WT_HCA	BP_WT_RC
35	BP_WT_LF	BP_WT_LF	BP_WT_JCA	BP_WT_BK
36	BP_WT_RC	BP_WT_RC	BP_WT_CK	BP_WT_CL
37	SP_AR_BK	SP_AR_SE	SP_AR_CL	SP_AR_HCA
38	SP_AR_CK	SP_AR_CK	SP_AR_RC	SP_AR_CL
39	SP_AR_CL	SP_AR_SE	SP_AR_CL	SP_AR_HCA
40	SP_AR_KS	SP_AR_KS	SP_AR_LF	SP_AR_RC
41	SP_AR_LF	SP_AR_SE	SP_AR_CL	SP_AR_HCA
42	SP_AR_RC	SP_AR_RC	SP_AR_CL	SP_AR_CK
43	SP_AR_SE	SP_AR_SE	SP_AR_CL	SP_AR_HCA
44	SP_CWT_BK	SP_CWT_BK	SP_CWT_LF	SP_CWT_JCA
45	SP_CWT_CK	SP_CWT_RC	SP_CWT_CK	SP_CWT_CL
46	SP_CWT_CL	SP_CWT_CL	SP_CWT_SE	SP_CWT_HCA
47	SP_CWT_KS	SP_CWT_KS	SP_CWT_HCA	SP_CWT_LF
48	SP_CWT_LF	SP_CWT_LF	SP_CWT_JCA	SP_CWT_BK
49	SP_CWT_RC	SP_CWT_RC	SP_CWT_CK	SP_CWT_CL
50	SP_CWT_SE	SP_CWT_SE	SP_CWT_CL	SP_CWT_HCA
51	SP_DT_BK	SP_DT_BK	SP_DT_SE	SP_DT_RC
52	SP_DT_CK	SP_DT_CK	SP_DT_RC	SP_DT_CL
53	SP_DT_CL	SP_DT_CL	SP_DT_LF	SP_DT_HCA
54	SP_DT_KS	SP_DT_KS	SP_DT_HCA	SP_DT_RC
55	SP_DT_LF	SP_DT_KS	SP_DT_LF	SP_DT_JCA
56	SP_DT_RC	SP_DT_RC	SP_DT_CK	SP_DT_CL
57	SP_DT_SE	SP_DT_SE	SP_DT_CL	SP_DT_HCA
58	SP_WT_BK	SP_WT_BK	SP_WT_LF	SP_WT_JCA
59	SP_WT_CK	SP_WT_CK	SP_WT_RC	SP_WT_CL
60	SP_WT_CL	SP_WT_CL	SP_WT_LF	SP_WT_HCA
61	SP_WT_KS	SP_WT_KS	SP_WT_HCA	SP_WT_RC
62	SP_WT_LF	SP_WT_LF	SP_WT_JCA	SP_WT_HCA
63	SP_WT_RC	SP_WT_RC	SP_WT_CK	SP_WT_CL
64	SP_WT_SE	SP_WT_SE	SP_WT_CL	SP_WT_HCA

C Detailed Equivalence Checking Time

This section presents a complete list of equivalence checking time, using 128-bit multipliers optimized by resyn3.

Table 10: Time and memory consumption of equivalence checking

#	Tested_resyn3_128_circuit	Time (seconds)	Memory (KB)	Method
1	SP_4to2_BK	61.60	78184	kissat
2	SP_4to2_HCA	81.04	87080	sat-sweeping
3	SP_4to2_JCA	125.33	81688	kissat
4	SP_4to2_KS	275.53	86572	kissat
5	SP_4to2_LF	126.82	86368	kissat
6	SP_4to2_RC	0.69	85356	fraig
7	SP_DT_BK	47.19	104264	kissat
8	SP_DT_HCA	81.94	105936	kissat
9	SP_DT_JCA	54.29	104828	kissat
10	SP_DT_KS	116.78	112904	kissat
11	SP_DT_LF	56.57	103688	kissat
12	SP_DT_RC	0.89	111632	fraig
13	SP_WT_BK	50.87	104216	kissat
14	SP_WT_HCA	5.42	67748	sat-sweeping
15	SP_WT_JCA	62.08	105424	kissat
16	SP_WT_KS	179.35	111812	kissat
17	SP_WT_LF	62.93	104300	kissat
18	SP_WT_RC	41.98	108100	kissat
19	BP_4to2_BK	83.30	129412	sat-sweeping
20	BP_4to2_HCA	271.10	111212	sat-sweeping
21	BP_4to2_JCA	925.40	100176	kissat
22	BP_4to2_KS	589.88	126496	sat-sweeping
23	BP_4to2_LF	964.83	102392	kissat
24	BP_4to2_RC	1.57	107072	fraig
25	BP_DT_BK	205.22	116008	sat-sweeping
26	BP_DT_HCA	374.82	109848	kissat
27	BP_DT_JCA	357.18	111816	kissat
28	BP_DT_KS	672.73	115396	kissat
29	BP_DT_LF	348.69	111860	kissat
30	BP_DT_RC	0.88	105508	fraig
31	BP_WT_BK	237.07	108100	kissat
32	BP_WT_HCA	382.67	108456	kissat

#	Tested_resyn3_128_circuit	Time (seconds)	Memory (KB)	Method
33	BP_WT_JCA	645.68	112884	kissat
34	BP_WT_KS	667.53	111092	kissat
35	BP_WT_LF	637.84	108664	kissat
36	BP_WT_RC	269.41	110884	sat-sweeping
37	SP_AR_BK	43.62	76176	sat-sweeping
38	SP_AR_CK	6.14	130784	fraig
39	SP_AR_CL	4947.87	276568	sat-sweeping
40	SP_AR_KS	1257.36	233016	sat-sweeping
41	SP_AR_LF	461.84	238684	sat-sweeping
42	SP_AR_RC	5.01	126128	fraig
43	SP_AR_SE	4.54	129400	fraig
44	SP_CWT_BK	115.41	114264	kissat
45	SP_CWT_CK	37.21	126924	sat-sweeping
46	SP_CWT_CL	3524.06	186608	kissat
47	SP_CWT_KS	240.08	88308	sat-sweeping
48	SP_CWT_LF	116.71	124204	kissat
49	SP_CWT_RC	27.63	92696	sat-sweeping
50	SP_CWT_SE	1.07	117224	fraig
51	SP_DT_BK	39.06	106960	kissat
52	SP_DT_CK	28.19	66628	sat-sweeping
53	SP_DT_CL	2450.09	149456	kissat
54	SP_DT_KS	122.68	108596	kissat
55	SP_DT_LF	71.81	107388	kissat
56	SP_DT_RC	1.20	113388	fraig
57	SP_DT_SE	0.76	81376	fraig
58	SP_WT_BK	4.71	68092	sat-sweeping
59	SP_WT_CK	59.40	108668	kissat
60	SP_WT_CL	1020.57	127684	kissat
61	SP_WT_KS	110.56	87728	sat-sweeping
62	SP_WT_LF	78.41	114068	kissat
63	SP_WT_RC	40.93	94736	sat-sweeping
64	SP_WT_SE	1.36	116064	fraig