Hardware Trojan Detection Using Graph Neural Networks: An AST-based GCN Approach

Md Omar Faruque
xxxxx@example.com
xxxxxx
xxxx, xxxx, xxxx

Abstract

Hardware Trojans (HTs) pose a significant threat to the security and integrity of integrated circuits. Detecting these malicious modifications, especially at the Register-Transfer Level (RTL), is a challenging task. This report presents a learning journey on hardware Trojan detection using Graph Convolutional Networks (GCNs) applied to Abstract Syntax Tree (AST) representations of hardware designs. We leverage the structural and semantic information captured by ASTs to train a GCN model capable of distinguishing between Trojan-free and Trojan-infected designs. Our methodology involves parsing Verilog code into ASTs, extracting relevant features, and training a GCN model on a processed dataset derived from TrustHub benchmarks. Experimental results demonstrate the effectiveness of our approach, achieving a high F1 score in detecting various types of hardware Trojans.

CCS Concepts: • **Security and privacy** → **Embedded systems security**; • **Computer systems organization** → *Hardware-software codesign*; • **Computing methodologies** → *Neural networks.*

Keywords: Hardware security, Trojan detection, graph neural networks, abstract syntax tree, GCN, hardware verification, RTL

ACM Reference Format:
Md Omar Faruque. 2024. Hardware Trojan Detection Using Graph Neural Networks: An AST-based GCN Approach. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation email (Conference Acronym 'XX).* ACM, New York, NY, USA, 6 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

The increasing complexity and globalization of the integrated circuit (IC) supply chain have raised significant concerns about hardware security. Hardware Trojans (HTs), malicious modifications intentionally inserted into ICs during design or fabrication, pose a severe threat to the trustworthiness of electronic systems. These Trojans can lead to various detrimental effects, including information leakage, denial of service, or complete system failure. Detecting HTs, particularly at the Register-Transfer Level (RTL) where designs are often shared with untrusted third parties, is a critical yet challenging task.

Traditional HT detection methods, such as logic testing and side-channel analysis, often struggle with the stealthy nature of modern Trojans, which can be small, cleverly hidden, and triggered by rare conditions. Machine learning (ML) techniques have emerged as a promising alternative, offering the potential to learn patterns indicative of Trojans from large datasets of hardware designs. Among ML approaches, Graph Neural Networks (GNNs) are particularly well-suited for analyzing graph-structured data, such as hardware circuits.

This report proposes an approach for hardware Trojan detection by applying Graph Convolutional Networks (GCNs) to Abstract Syntax Tree (AST) representations of RTL Verilog code. ASTs capture rich structural and semantic information from the source code, providing a more detailed representation than traditional circuit graphs. By leveraging GCNs to learn from these ASTs, our method aims to effectively identify subtle differences between Trojan-free and Trojan-infected designs.

The main contributions of this work are:

1. The application of GCNs for classifying hardware designs as Trojan-free or Trojan-infected based on their AST representations.
2. An empirical evaluation of the proposed approach on a processed dataset derived from TrustHub benchmarks, demonstrating its effectiveness in detecting various types of hardware Trojans.

The remainder of this report is organized as follows: Section 3 discusses the methodology, including the theoretical framework and the proposed GCN-on-AST approach. Section 4 details the experimental setup, including the dataset, implementation details, and evaluation metrics. Section 5 presents and analyzes the experimental results. Section 6 discusses the key findings and limitations of our work. Finally, Section 7 concludes the report and outlines future research directions.

2 Methodology

2.1 Theoretical Framework

2.1.1 Graph Representation for Hardware Designs. Hardware designs, particularly at the RTL, can be naturally represented as graphs. Nodes in such graphs can represent various hardware elements like modules, instances, wires, or operations, while edges can represent connections or dependencies between these elements. Graph-based representations allow for the application of powerful graph learning algorithms.

2.1.2 Abstract Syntax Trees (ASTs). An Abstract Syntax Tree (AST) is a tree representation of the abstract syntactic structure of source code [1]. Each node in an AST denotes a construct occurring in the source code. Unlike concrete syntax trees, ASTs do not represent every detail appearing in the real syntax, but rather focus on the structural and semantic aspects [4]. For hardware description languages (HDLs) like Verilog, an AST can capture the hierarchical structure of modules, instances, assignments, control flow statements, and expressions. This detailed representation is beneficial for identifying subtle malicious modifications [7].

Algorithm 1 outlines the general process for generating an AST from Verilog code.

Algorithm 1 AST Generation from Verilog Code

```
1: procedure GenerateAST(VerilogFile)
2:     $Source \leftarrow$ Read($VerilogFile$)
3:     $Tokens \leftarrow$ LexicalAnalysis($Source$)
4:     $ParseTree \leftarrow$ SyntaxAnalysis($Tokens$)
5:     $AST_{root} \leftarrow$ AbstractParseTree($ParseTree$)
6:     function BuildGraph($Node, ParentNode, Graph$)
7:         $Graph$.AddNode($Node$.id, type = $Node$.type, attributes = ...)
8:         if $ParentNode \neq$ null then
9:             $Graph$.AddEdge($ParentNode$.id, $Node$.id)
10:        end if
11:        for all $Child \in Node$.children do
12:            BuildGraph($Child, Node, Graph$)
13:        end for
14:    end function
15:    $Graph_{AST} \leftarrow$ InitializeGraph()
16:    BuildGraph($AST_{root}$, null, $Graph_{AST}$)
17:    return $Graph_{AST}$
18: end procedure
```

Consider the following simple Verilog module for a 4-bit counter as a toy example:

```verilog
module counter (
  input clk,
  output reg [3:0] count
);
  always @(posedge clk) begin
    count <= count + 1;
  end
endmodule
```

Listing 1. A simple 4-bit counter in Verilog.

Figure 2 illustrates a simplified AST for this 'counter' module (Listing 1). The AST captures the module definition, its ports, and the 'always' block with the increment operation. Figure 1 shows the general process of generating this AST graph representation from the source code.

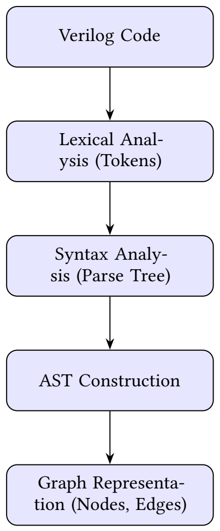
Verilog Code
Lexical Analysis (Tokens)
Syntax Analysis (Parse Tree)
AST Construction
Graph Representation (Nodes, Edges)

Figure 1. AST Graph Generation Process.

During the parsing of various Verilog files in the dataset, a dictionary of AST node types is generated. This typically includes over 45 unique node types such as 'ModuleDef', 'Portlist', 'Decl' (for declarations like 'input', 'output', 'reg', 'wire'), 'InstanceList', 'Assign', 'Always', 'IfStatement', 'CaseStatement', 'BlockingSubstitution', 'NonblockingSubstitution', 'Identifier', 'IntConst', various operators ('Plus', 'Minus', 'And', 'Or', 'Xor'), 'Partselect', 'Concat', etc. Each unique node type is assigned an integer ID, which is then used for feature encoding.

2.1.3 Graph Neural Networks (GNNs). Graph Neural Networks (GNNs) are a class of deep learning models designed to perform inference on data described by graphs [9]. GNNs operate by iteratively aggregating information from a node's neighbors to update its representation (embedding). This message-passing mechanism allows GNNs to learn complex patterns and relationships within graph-structured data [6].

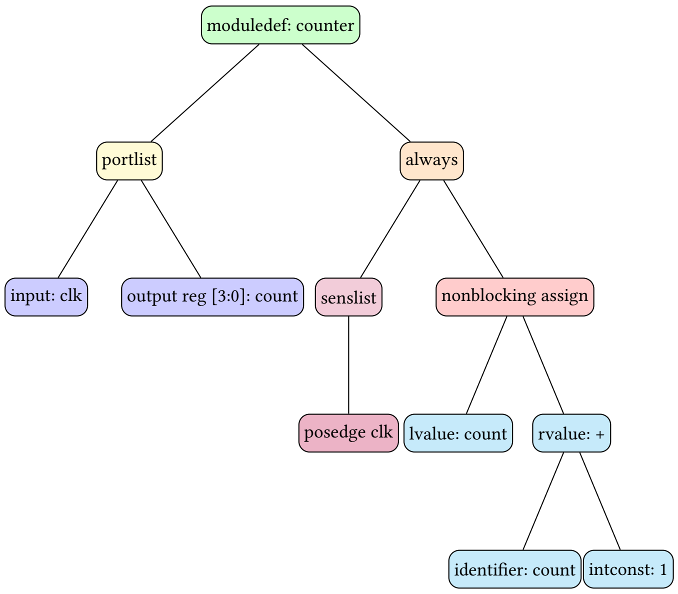

Figure 2. Simplified AST for the 4-bit counter module (Listing 1).

2.1.4 Graph Convolutional Networks (GCNs). Graph Convolutional Networks (GCNs) [3] are a specific type of GNN that generalizes the concept of convolution from regular grids (like images) to irregular graph structures. A common GCN layer updates the feature vector of a node v_i by aggregating the feature vectors of its neighbors $\mathcal{N}(i)$, often followed by a linear transformation and a non-linear activation function [2]. The layer-wise propagation rule for a GCN can be expressed as:

$$H^{(l+1)} = \sigma\left(\tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}H^{(l)}W^{(l)}\right) \tag{1}$$

where $H^{(l)} \in \mathbb{R}^{N\times F^{(l)}}$ is the matrix of node activations in layer l (with $H^{(0)} = X$ being the initial node features, N nodes, $F^{(l)}$ features per node in layer l), $W^{(l)} \in \mathbb{R}^{F^{(l)}\times F^{(l+1)}}$ is a layer-specific trainable weight matrix. $\sigma(\cdot)$ is an activation function (e.g., ReLU). $\tilde{A} = A + I_N$ is the adjacency matrix of the graph G with added self-loops (where I_N is the identity matrix), and $\tilde{D}$ is the diagonal degree matrix of $\tilde{A}$, i.e., $\tilde{D}_{ii} = \sum_j \tilde{A}_{ij}$. The term $\tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}$ serves as a symmetric normalization of the adjacency matrix.

2.2 Proposed Approach

2.2.1 AST-based Hardware Representation. Our approach begins by parsing RTL Verilog files into ASTs, as outlined in Algorithm 1. Each node in the AST represents a syntactic construct from the Verilog code (e.g., module definition, port declaration, assignment, instance). Edges in the AST represent the hierarchical relationships between these constructs (e.g., a module node has children nodes representing its ports, declarations, and instances).

2.2.2 Feature Extraction. For each node in the AST, a feature vector is constructed to capture its characteristics and context within the hardware design. The features used in this work include: ❶ **Node Type Encoding**: The type of the AST node (e.g., 'ModuleDef', 'Assign', 'Identifier') is represented using one-hot encoding. Given that over 45 unique node types were discovered across the dataset, a vector of approximately 50 dimensions is used for this encoding, where the dimension corresponding to the node's type is set to 1 and others to 0. ❷ **Node Depth**: An integer representing the depth of the node in the AST, indicating its hierarchical level. ❸ **Number of Children**: An integer count of the direct children of the node, reflecting its local complexity or fan-out in the tree structure. ❹ **Average Child Type ID**: A float value calculated as the average of the numerical IDs of its children's types. This feature provides contextual information about the node's immediate descendants.

Combining these, each node is represented by a feature vector of 50+ dimensions (e.g., 50 for one-hot type encoding + 3 for structural features). These features aim to capture both the specific function of a node and its structural role within the AST.

2.2.3 GCN Architecture for Trojan Detection. The core of our detection methodology is a GCN model specifically designed to process the AST graphs. The overall framework is depicted in Figure 3, and the detailed GCN architecture is shown in Figure 4.

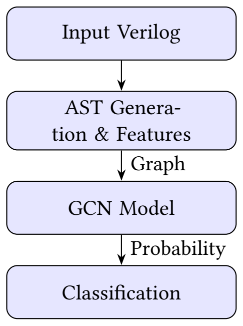

Figure 3. Overall Proposed Framework for Trojan Detection.

The model takes the AST graph with its node features as input. It consists of two GCN layers (using Equation 1) with ReLU activations and Dropout for regularization. These layers learn node embeddings that capture local and neighborhood information within the AST. A global mean pooling layer then aggregates these node embeddings into a single fixed-size vector representing the entire graph. This graph embedding is fed into a Multi-Layer Perceptron (MLP) classifier (whose specific structure, e.g., number of layers and hidden units, is determined by hyperparameter tuning) which uses ReLU activations in hidden layers and Dropout. The

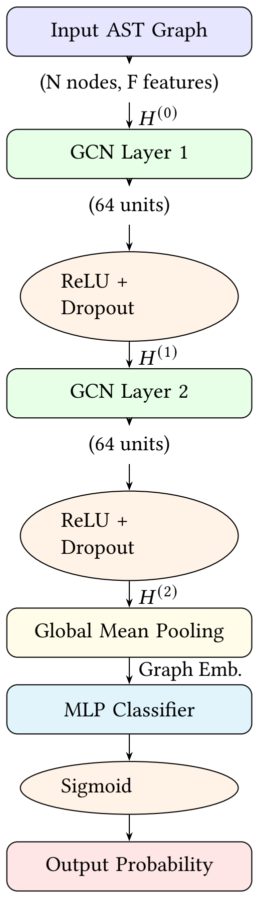

Figure 4. Detailed GCN Model Architecture.

final output layer uses a Sigmoid activation function to produce a probability score between 0 (likely Trojan-free) and 1 (likely Trojan-infected). The model is trained end-to-end using a binary cross-entropy loss function.

3 Experimental Setup

3.1 Dataset

The experiments were conducted using hardware designs derived from the TrustHub benchmark suite [5]. This dataset was prepared by merging the sub-module Verilog files for each selected TrustHub IP into a single `topModule.v` file. These merged designs were then categorized into Trojan-free (TjFree) and Trojan-infected (TjIn) sets. The dataset composition is present in Table 1.

The Trojans present in the TjIn set, originating from TrustHub, vary in type, trigger mechanism, and payload. The resulting dataset used for training is relatively small and exhibits class imbalance, with significantly more Trojan-infected samples than Trojan-free ones. This imbalance is addressed during training using oversampling and weighted loss, as detailed later.

Table 1. Processed TrustHub Dataset Summary

Category	**Description**
Total Circuits	43 processed designs derived from TrustHub IPs
TjFree Designs (Clean)	9 circuits (20.9% of dataset) Includes: `det_1011`, `PIC16F84`, `RC5`, `RC6`, `RS232`, `spi_master`, `syncRAM`, `vga`, `xtea`
TjIn Designs (Trojan-infected)	34 circuits (79.1% of dataset) Includes variants of: AES (`T100-T2100`), PIC16F84 (`T100-T400`), RS232 (`T100-T901`), `wb_conmax`

3.2 Implementation Details

3.2.1 AST Graph Generation. Verilog files were parsed using the 'pyverilog' library. An AST generator script traversed the parsed Verilog constructs to build a graph representation using the 'networkx' library. Each node in the graph corresponds to a Verilog construct, and edges represent parent-child relationships in the AST. Node features, including one-hot encoded type, depth, number of children, and average child type ID, were extracted.

3.2.2 GCN Implementation. The GCN model was implemented using PyTorch Geometric. The model typically consisted of two GCN layers with ReLU activation and dropout, followed by a global mean pooling layer and an MLP classifier with a sigmoid output.

3.2.3 Training Procedure. The dataset was split into training and testing sets using Stratified K-Fold cross-validation (with K=3 folds in the reported experiments). Oversampling was applied to the training data to handle class imbalance between Trojan-free and Trojan-infected samples. The model was trained using the Adam optimizer and a binary cross-entropy loss function with weights to account for class imbalance.

3.3 Evaluation Metrics

The performance of the model was evaluated using standard classification metrics, defined in Table 2. These metrics provide a comprehensive view of the model's ability to distinguish between Trojan-free and Trojan-infected designs.

3.4 Hyperparameter Optimization

Hyperparameters for the GCN model and the MLP classifier were tuned using Optuna, an automatic hyperparameter optimization framework. The optimization process aimed to maximize the average F1 score across the cross-validation folds. Parameters tuned included learning rate, GNN dropout

Table 2. Evaluation Metrics

Metric	**Formula**	**Description (Context: Trojan Detection)**
Accuracy	$\frac{TP+TN}{TP+TN+FP+FN}$	Overall proportion of designs (Trojan or Clean) correctly classified.
Precision	$\frac{TP}{TP+FP}$	Of all designs predicted as Trojan-infected, the proportion that actually are Trojan-infected. High precision means few clean designs are misclassified as Trojan-infected (low false alarms).
Recall	$\frac{TP}{TP+FN}$	Of all actual Trojan-infected designs, the proportion that were correctly identified (Sensitivity). High recall means few Trojan-infected designs were missed (low false negatives).
F1 Score	$2 \times \frac{Precision \times Recall}{Precision+Recall}$	Harmonic mean of Precision and Recall, balancing the trade-off between missing Trojans (low recall) and false alarms (low precision).

TP: Trojan design correctly identified, TN: Clean design correctly identified, FP: Clean design misidentified as Trojan, FN: Trojan design misidentified as Clean.

rate, classifier hidden dimensions, number of classifier layers, and classifier dropout rate.

4 Results and Analysis

4.1 Hyperparameter Optimization Results

The Optuna framework was employed to search for optimal hyperparameters over 5 trials, maximizing the average F1 score across 3 cross-validation folds. The best performing set of hyperparameters identified is shown in Table 3. These parameters were used for the final model evaluation reported in subsequent sections.

Table 3. Best Hyperparameters Found by Optuna

Hyperparameter	Optimal Value
Classifier Hidden Dimension	32
Classifier Layers	4
Classifier Dropout Rate	0.217
GNN Dropout Rate	0.621
Learning Rate	0.00345

4.2 Cross-Validation Performance

Using the best hyperparameters, the GCN model was evaluated using 3-fold cross-validation. The performance metrics for each fold and the average performance are summarized in Table 4.

The model achieved an average F1 score of 0.9311, indicating a strong ability to detect hardware Trojans. The average test accuracy was 0.8825, and the average train accuracy was 0.8976. While there was some variation across folds, particularly in Fold 2, the overall performance demonstrates the robustness of the GCN-on-AST approach.

Table 4. Cross-Validation Performance Metrics

Fold	Accuracy	Precision	Recall	F1 Score
Fold 1	0.9333	0.9231	1.0000	0.9600
Fold 2	0.7143	0.7692	0.9091	0.8333
Fold 3	1.0000	1.0000	1.0000	1.0000
Average	**0.8825**	**0.8974**	**0.9697**	**0.9311**

4.3 Training Dynamics

The model was trained for 200 epochs per fold. The computational efficiency, averaged over the 3 folds, is summarized in Table 5.

Table 5. Computational Efficiency (Averaged)

Metric	Average Time
Training Time per Fold	~20.5 seconds
Training Time per Sample	~0.45 seconds
Inference Time per Fold	~0.02 seconds
Inference Time per Sample	~0.0013 seconds

4.4 Comparison with Baseline Methods

To contextualize our results, we compare the performance of our GCN-on-AST approach with reported results from the HW2VEC method [8] applied to RTL designs, using both Data Flow Graph (DFG) and AST representations. Table 6 summarizes this comparison.

Our GCN-on-AST approach achieves a slightly higher F1 score (0.931) compared to HW2VEC using DFG (0.926) and significantly outperforms HW2VEC using AST (0.845). While our precision (0.897) is slightly lower than HW2VEC-AST (0.903) but higher than HW2VEC-DFG (0.873), our recall (0.970) is substantially higher than HW2VEC-AST (0.800) and comparable to HW2VEC-DFG (0.986).

Table 6. Comparison with HW2VEC Baseline

Method	Precision	Recall	F1 Score
HW2VEC (DFG RTL)	0.873	0.986	0.926
HW2VEC (AST RTL)	0.903	0.800	0.845
Our GCN (AST RTL)	**0.897**	**0.970**	**0.931**

5 Discussion

The experimental results demonstrate the effectiveness of using GCNs on AST representations for hardware Trojan detection. The high average F1 score of 0.9311 achieved on the processed TrustHub dataset suggests that the model can accurately identify Trojan-infected designs while maintaining a good balance between precision and recall.

However, several limitations should be acknowledged. The dataset used, while derived from TrustHub, is relatively small (43 designs) and imbalanced, which might limit the generalizability of the findings to larger, more complex industrial designs or different Trojan types not represented. The performance variation observed across cross-validation folds (particularly the lower F1 score in Fold 2) suggests potential sensitivity to specific data splits or overfitting, warranting further investigation with larger datasets. Finally, while GCNs achieve high accuracy, interpreting their decisions remains challenging, making it difficult to pinpoint the exact features or subgraphs indicative of a Trojan. Addressing these limitations, particularly through evaluation on more extensive datasets and the application of GNN explanation techniques, constitutes important avenues for future research.

6 Conclusion and Future Work

This report presented a GCN-based approach for hardware Trojan detection using AST representations of RTL Verilog code. Our method successfully leverages the structural and semantic information captured by ASTs to train a GCN model that can effectively distinguish between Trojan-free and Trojan-infected designs. Experiments on a processed dataset derived from TrustHub benchmarks demonstrated high F1 scores, highlighting the promise of this technique for automated hardware security analysis at the RTL.

Future work will focus on several directions to build upon these findings. Evaluating the approach on larger and more diverse datasets. Exploring enhanced feature engineering techniques for AST nodes and investigating more advanced GNN architectures, such as Graph Attention Networks (GATs) or Graph Isomorphism Networks (GINs), could potentially improve performance and robustness. Additionally, applying GNN interpretability methods is essential for understanding the model's decision-making process.

References

[1] V Aho Alfred, S Lam Monica, and D Ullman Jeffrey. 2007. *Compilers principles, techniques & tools.* pearson Education.

[2] Will Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. *Advances in neural information processing systems* 30 (2017).

[3] Thomas N Kipf and Max Welling. 2016. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907* (2016).

[4] George C Necula, Scott McPeak, Shree P Rahul, and Westley Weimer. 2002. CIL: Intermediate language and tools for analysis and transformation of C programs. In *International Conference on Compiler Construction*. Springer, 213–228.

[5] Trust-Hub. Accessed: 2024. Trust-Hub: A Benchmark Suite for Hardware Security and Trust. https://trust-hub.org/.

[6] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S Yu. 2020. A comprehensive survey on graph neural networks. *IEEE transactions on neural networks and learning systems* 32, 1 (2020), 4–24.

[7] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. 2014. Modeling and discovering vulnerabilities with code property graphs. In *2014 IEEE symposium on security and privacy*. IEEE, 590–604.

[8] Shih-Yuan Yu, Rozhin Yasaei, Qingrong Zhou, Tommy Nguyen, and Mohammad Abdullah Al Faruque. 2021. HW2VEC: A Graph Learning Tool for Automating Hardware Security. arXiv:2107.12328 [cs.CR]

[9] Jie Zhou, Ganqu Cui, Shengding Hu, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. 2020. Graph neural networks: A review of methods and applications. *AI open* 1 (2020), 57–81.