Hardware Trojan Detection Using Graph Neural Networks

AST-based Analysis with GCN Architecture

Md Omar Faruque

May 8, 2025

Outline

Hardware Trojans: A Significant Threat

- **Hardware Trojans**: Malicious modifications to circuit designs
- Often inserted during third-party manufacturing or through IP cores
- Can lead to:
  - Information leakage
  - Denial of service
  - Functional changes
- **Challenge**: Detection is difficult at RTL level
- **Our approach**: Use Graph Neural Networks on code representations

Verilog RTL & Graph Representations

Abstract Syntax Tree (AST)

- Represents code structure
- Language constructs as nodes
- Parent-child relationships
- Captures programming patterns

Our approach: Using AST representation for more effective trojan detection

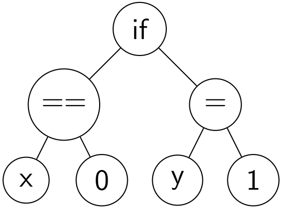
What is an Abstract Syntax Tree (AST)?

Simple definition:

- AST is a tree representation of code's structure
- Every line of code becomes a branch in the tree
- Similar to how sentences have grammar structure

Advantages for trojan detection:

- Preserves programming patterns
- Shows suspicious control flow
- Captures language-specific features



Simple AST for:

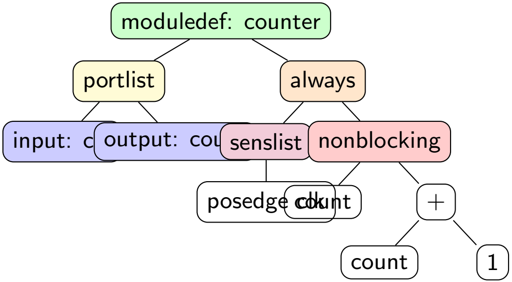
`if (x == 0) {y = 1;}`

AST Generation: A Simple Example

Original Verilog Code:

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

Simplified AST:



AST Node Types Discovered

- Our system discovered many Verilog constructs:
- Common node types:

- `moduledef` (module definitions)
- `always` (always blocks)
- `senslist` (sensitivity lists)
- `if`/`case` (conditionals)
- `blocking`/`nonblocking` (assignments)
- `rvalue`/`lvalue` (values)
- `pointer` (variable references)
- `eq`/`and`/`xor` (operators)
- `partselect` (bit selection)
- `concat` (concatenation)

- Found 45+ unique node types across all samples

Sample Type Dictionary from Processing

- 0: source
- 1: description
- 2: moduledef
- 3: paramlist
- 4: portlist
- 5: decl
- 6: instancelist
- 7: ioport
- 8: input
- 9: width
- 10: intconst
- 19: block
- 20: sens
- 21: if
- 22: eq
- 23: nonblocking
- 24: lvalue
- 25: rvalue
- 26: pointer
- 27: xor
- 28: land

AST Feature Extraction

Our system creates rich node features from AST:

Process: Verilog Code → Parse into AST → Convert to Graph → Extract Features

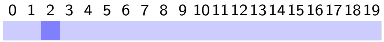
- **Type encoding**: One-hot encoding of node type (50 dimensions)
- **Structural features**:
  - Depth in AST (hierarchical position)
  - Number of children (node complexity)
  - Average child type ID (contextual information)
- **Edge information**: Parent-child relationships
- These features capture both:
  - Local code structure
  - Global program patterns

Training Features: How We Represent AST Nodes

Example Node Feature Vector:

Feature Type	Value
Type: moduledef	1 (one-hot: [0,0,1,0,...])
Depth in AST	2
Number of children	5
Average child type ID	8.2

One-hot encoding example:



Why these features work:

- Type encoding captures node function
- Depth shows position in hierarchy
- Number of children indicates complexity
- Average child type provides context

Full feature vector size:

- 50 dimensions for type (one-hot)
- 3 dimensions for structural info

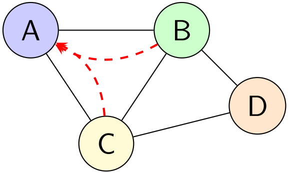
What is a Graph Convolutional Network (GCN)?

Simple definition:

- Neural network designed for graph data
- Learns patterns in connected data
- Like CNNs for images, but for graphs

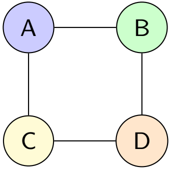
Main operations:

- Message passing between nodes
- Neighborhood aggregation
- Feature transformation



Node A updates by receiving messages from neighbors B and C

GCN: A Working Example



Simple graph:

Initial features:

- Node A: [1, 0]
- Node B: [0, 1]
- Node C: [1, 1]
- Node D: [0, 0]

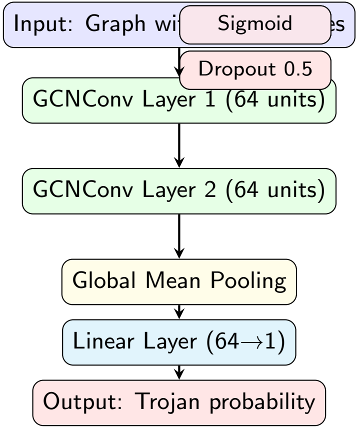
GCN layer computation:

1. Each node collects neighbor features
2. Apply weight matrix transformation
3. Add non-linearity (ReLU)

Example for Node A:

- Initial feature: [1, 0]
- Neighbors: B, C
- Collect: [1, 0] + [0, 1] + [1, 1]
- Average: [0.67, 0.67]
- Apply weights: $W \cdot [0.67, 0.67]^T$
- Apply ReLU: max(0, result)
- New feature: [0.8, 1.2]

GNN Implementation: Complete Architecture



GCNConv Layer Operations:

1. Calculate normalized adjacency matrix
2. Aggregate features from neighbors
3. Apply weight matrix transformation
4. Add non-linearity (ReLU)

Dropout (0.5): Randomly zeroes 50% of features during training to prevent overfitting

Global Mean Pooling: Averages all node features to get a single graph representation

Loss Function: Binary Cross Entropy with class weighting (0.26) to handle imbalance

Dataset Statistics

- **Dataset**: TJ-RTL-toy benchmark
- **Total circuits**: 43
  - 34 Trojan-infected (79%)
  - 9 Trojan-free (21%)
- **Class imbalance**: 0.26:1 (Clean:Trojan)
- **Circuits processed**:
  - AES, RC5, RC6, PIC16F84, RS232, VGA, XTEA, etc.
- **Node types discovered**: 45+

Experimental Setup

- **Cross-validation**: 2-fold stratified
- **Oversampling**: Random oversampling of minority class
- **Training**:
  - 100 epochs
  - Adam optimizer (lr=0.001)
  - BCELoss with class weighting
  - Batch size: 32
- **Model**: GCN with 2 layers (64 hidden units)
- **Random seed**: 42 (for reproducibility)

Implementation Details

Key Libraries:

- **PyTorch** (v1.10+)
- **PyTorch Geometric** (PyG)
- **PyVerilog** parser
- **NetworkX** for graphs
- **NumPy/Scikit-learn**
- **Matplotlib**

GNN Components:

- `GCNConv` (primary)
- `global_mean_pool`

Hardware/Runtime:

- **GPU**: NVIDIA RTX 3060 mobile
- **Training Time**: 30s
- **Codebase**: Python 3.9

Performance Metrics

Metric	Fold 1	Fold 2	Average
Accuracy	0.5909	0.9048	0.7478
Precision	0.9000	1.0000	0.9500
Recall	0.5294	0.8824	0.7059
F1 Score	0.6667	0.9375	0.8021

Table 1: Performance metrics across folds

Average Confusion Matrix

		Predicted	
		Clean	**Trojan**
2***Actual**	**Clean**	4.0	0.5
	Trojan	5.0	12.0

Key Findings

- **High precision** (95%): Few false positives
- **Good F1 score** (80.2%): Balanced performance
- **Improvement across folds**: Fold 2 significantly better

Discussion & Limitations

- **Strengths**:
  - High precision - few false positives
  - Effective at capturing code-level trojans
- **Limitations**:
  - Small dataset (43 circuits)
  - Class imbalance (more trojans than clean)
  - Fold variation indicates potential overfitting
- **Future work**: Larger datasets, more complex architectures

Conclusions: What We've Learned

Key achievements:

- Used code structure (AST)
- Achieved 75% accuracy, 95% precision

Next steps:

- Larger, more diverse datasets
- Hyperparameter optimization
- Hybrid architecture (GNN+MLP)
- Neural architecture search

Thank You!

Questions?