Hardware Trojan Detection Utilizing Graph Neural Networks and Structural Checking

Hunter Nauman
Department of Electrical Engineering and Computer Science
University of Arkansas
Fayetteville, Arkansas. USA
hjnauman@uark.edu

Jia Di
Department of Electrical Engineering and Computer Science
University of Arkansas
Fayetteville, Arkansas. USA
jdi@uark.edu

***Abstract*—The integrated circuit (IC) industry has experienced exponential growth in the complexity and scale of hardware designs. To sustain this growth, faster development cycles and cost-effective solutions have been the focus of many companies, notably through the incorporation of third-party intellectual property (IP). Outsourcing the production of sub-components reduces development time and enables faster time-to-market; however, this approach also introduces the threat of Hardware Trojans, which are malicious modifications or additions to an IC, posing significant security risks due to their small size, low activation frequency, and complex obfuscation techniques. This research proposes an advancement to the Trojan detection mechanisms incorporated in the Structural Checking Tool, a Trojan detection tool that focuses on the identification of logical Trojans embedded within soft IPs. Leveraging graph structures generated by the tool and signal-level features, this research develops a new dataset and three graph neural network architectures. Each neural network corresponds to classical graph neural network layers and execute graph-level probabilistic binary classification of Trojan inclusion. Through rigorous testing with two potential sets of node-level feature vectors, this research offers a faster, more accurate, and more adaptable approach than those existing within the current tool.**

Keywords—Hardware Trojan, structural checking, asset, golden reference matching, graph neural networks

I. Introduction

The integrated circuit (IC) industry has experienced exponential growth, marked by an increase in the complexity and scale of hardware designs. To meet the demand for faster development cycles and cost-effective solutions, many companies have adopted the practice of incorporating third-party intellectual property (IP) into their design processes. Common IP can be purchased at a relatively low cost and instantly integrated into the development process. Furthermore, this approach can streamline synchronous workflows of various sub-systems and reduce the workload of in-house development teams, enhancing efficiency. However, despite delivering these considerable benefits, it allows for the emergence of Hardware Trojans, which present a pressing security concern.

Hardware Trojans, defined as any intentional malicious modification to an IC, represent a major threat to the IC ecosystem. These alterations to the circuit can serve various malevolent purposes, such as reducing reliability, altering functionality, or leaking critical data. Due to the pervasiveness of ICs within military systems, medical devices, critical infrastructure, and many other sectors where security is a major concern, the consequences of a Hardware Trojan can be devastating.

To address these concerns, many researchers have developed methodologies to detect the inclusion of Trojans within ICs. The authors of [1] outline a pre-synthesis and post-synthesis approach which uses both register-transfer level (RTL) code and gate-level netlists to detect Trojan behavior. The authors used an SVM-based concept to detect Trojan behavior using features extracted from both RTL and gate-level netlists of known clean and Trojan-infested designs. Their pre-synthesis approach, which focused on the extraction of features in RTL code for classification, obtained an accuracy of 80%. When combining both the pre-synthesis and post-synthesis approaches, an accuracy of 100% was obtained. While this procedure offered impressive accuracy, it was heavily limited to the detection of Trojans that only used conditional-based triggers. This resulted in a severely limited dataset, containing only ten designs.

This research aims at addressing the critical challenge of pre-synthesis Hardware Trojan detection in the context of modern ICs. Building upon the foundational work laid out in [2], this research leverages the Structural Checking (SC) Tool, a Hardware Trojan detection tool capable of automating the parsing of Hardware Description Language (HDL) code and transforming it into a graph structure that represents the intricate interconnections between signals within an IC. By harnessing the potential of this graph-based representation and augmenting it with data gathered from the parsed HDL, this research endeavors to develop a novel model for Hardware Trojan detection using graph neural networks. The objective of this research is to enhance the efficiency of Trojan detection compared to the SC Tool's existing methodology while advancing precision and accuracy.

The remainder of this paper is organized as follows. Section II will briefly cover background information for Hardware Trojans, the SC Tool, and graph neural networks. Section III will explain the methodology and implementation associated with the graph neural networks incorporated into the SC Tool. Section IV provides sample results and analysis. Section V concludes the paper, providing details on future work.

II. Background

A. Hardware Trojans

Hardware Trojans are defined as any intentional malicious modification to an IC. The functionality and design of Hardware Trojans can be extremely intricate and intentionally convoluted.

Despite the intricacies observed among Hardware Trojan implementations, the goal of these malicious modifications can be easily broken down into three distinct categories: reducing reliability, altering functionality, and leaking critical data. These three categories of alteration can have devastating effects in mission-critical systems.

To illustrate the threat that Hardware Trojans pose, two of the goals defined previously, i.e., altering functionality and leaking critical data, have been observed within modern military systems. In 2007 [3] and 2012 [4], respectively. Hardware Trojans are typically engineered to be extremely small and challenging to trigger. A Trojan can be implemented with a few simple logic gates, while a modern processor consists of millions of gates. Smaller Trojans often leave an inconsequential impact on metrics such as leakage power, dynamic power, path delay, and electromagnetic emissions. This, coupled with process variations in modern nanometer technologies and measurement noise, can lead to the failure of Trojan detection methods like side-channel analysis. Furthermore, Trojans are often programmed to activate under highly improbable circumstances, such as specific sequences of unlikely input combinations. Consequently, due to the statistical unlikelihood of such combinations occurring, conventional testing and validation methods prove unreliable for Trojan detection. Moreover, many of these approaches focus on detecting manufacturing defects and do not address the identification of additional, malicious functionalities.

B. *Structural Checking Tool*

The SC Tool, initially introduced in [5] and most recently updated in [2] and [6], serves as a Trojan detection tool with a focus on the identification of logical Trojans embedded within soft IP. Notably, it employs static analysis, eliminating the need for simulation or synthesis of a circuit for Trojan detection. This static analysis is significant for multiple reasons. Firstly, it allows for the detection of Hardware Trojans in HDL code, one of the earliest and most vulnerable points of the IC design process. Secondly, because no simulation or synthesis is required, Trojan detection can be performed far quicker than alternative methods. From a high-level perspective, the SC Tool accomplishes this goal through three distinct internal processes: design parsing and graph creation, asset assignment, and Trojan detection.

1) Design Parsing and Graph Creation: The tool's workflow begins by parsing the HDL source code associated with an IC whose Trojan status is unknown. This is accomplished using hdlConvertor [7], a Verilog and VHDL parsing library built using ANTLR4 [8]. During this parsing stage, an abstract syntax tree (AST) is generated for each HDL file associated with the circuit. This AST breaks down each element of the syntax within the HDL files and converts them into language-agnostic data types that can then be used to generate a structural representation of the circuit. The structural information extracted during this step includes entity declarations, ports, generics, internal signals, assignment statements, and more. This information is used to generate a directed graph representation of the circuit for each component via the NetworkX [9] Python library. Nodes in the graph are represented by ports, generics, and internal signals, while edges in the graph represent the driving and driven connections between these signals. The edges are defined by assignment statements and various types of conditional logic within the HDL. A single graph representation for the circuit can then be created by referencing component declarations within the HDL.

2) Asset Assignment: After generation of the circuit's structural framework, asset assignment begins. Asset assignment attributes descriptive labels to signals within an IC, enhancing our understanding of their intended roles and functionalities. In the SC Tool, assets are classified as either external or internal. External assets Are assigned manually to ports and generics of each component within the IC. For instance, a user might designate a *System Timing* external asset to the primary clock of a circuit. Currently, the SC Tool contains 87 external assets split across seven categories. Conversely, internal assets are applied automatically to all signals within an IC, predicated on key aspects identified within the HDL. For example, the *Conditional Expression Driving* asset, is applied to signals influencing Boolean expressions within if statements, case statements, while loops, and Verilog-styled for loops. Currently, the SC Tool contains 31 automatically assigned internal assets split across eight categories. After asset assignment, a filtering step is applied. This step is designed to disseminate assigned assets through driving and driven connections, providing a more comprehensive insight into each signal's role within the circuit.

3) Trojan Detection: The Trojan detection mechanisms employed by the SC Tool encompass two distinct approaches: Golden Reference Library (GRL) statistical matching and structure-based Trojan detection algorithms. GRL statistical matching, introduced in [10] and most recently updated in [11] and [12], involves comparing an unknown design – a circuit whose Trojan status is unknown – against the GRL, a collection of known clean and Trojan-infested designs [2]. This process aims to identify the highest match between the unknown design and a clean or Trojan-infested design within the GRL. A match percentage is computed based on asset similarity, as well as other characteristics, between all signals within both the unknown design and the GRL entry. This matching process is broken into two stages. The first stages approximates the unknown design's functionality by perfomring matching against the champion GRL, a subset of the GRL containing a single design per design functionality. The unknown design is then matched against GRL deisngs matching that functionliaty. The highest match percentage calculated during this stage is used to determine the likelihood of the unknown design hosting Trojan logic. The second approach to Trojan detection relies on structure-based Trojan detection algorithms [6][13]. Currently, the SC Tool incorporates seven active Trojan detection algorithms. Each algorithm aligns with a specific Trojan taxonomy and leverages structural elements identified in the unknown design, along with user asset information, to flag Trojan behavior. The Trojans currently detected by the tool encompass Trojan clocks, Trojan key leakage, Trojan battery

drain, Trojan counters, sensitive data leakage Trojans, data modification Trojans, and denial of service Trojans.

C. Graph Neural Networks

Graph Neural Networks (GNNs) are a class of machine learning models designed to work with data structured as graphs. Many data structures, such as social networks, molecular structures, and ICs, can naturally be represented via a graph data structure. While graph data structures are applicable to many real-world scenarios, they pose challenges for traditional deep learning frameworks. Graphs are inherently irregular data structures. Unlike regular grids or sequences, such as images and text, graphs have variable-sized neighborhoods for each node. Traditional deep learning frameworks are designed for regular data, making it difficult to represent and process the variable-sized and non-Euclidean nature of graph data.

Deep learning frameworks often rely on the notion of local connectivity, where each element (e.g., pixel or word) is connected to a fixed set of neighbors. In contrast, graphs can have nodes with highly varying degrees making it challenging to define local neighborhoods. Similar to the notion of local connectivity, datasets of graphs may also have high variability in the number of nodes and edges from graph to graph, making it infeasible to define a standardized model that could apply to all graphs within the dataset.

GNNs are designed to learn representations for nodes and entire graphs, capturing both node attributes and their relationships with other nodes through message-passing and aggregation functions. In each layer, nodes exchange information with their neighbors, updating their representations iteratively over multiple layers, thus gathering information from progressively larger neighborhoods. Aggregation functions combine these messages, enabling GNNs to capture complex relationships and dependencies within the graph. The resulting node embeddings from this process can be used for various applications, such as node classification, edge prediction, and graph classification. For graph classification, an additional global pooling layer is typically applied to create a graph-level embedding for downstream tasks.

Various modifications to the message-passing and aggregation mechanisms in GNN layers have led to the development of multiple classical GNN models. For instance, Graph Convolutional Networks (GCNs) [14] and GraphSAGE [15] are based on convolutional layers adapted for graphs. Additionally, Graph Attention Networks (GATs) [16] use attention mechanisms to dynamically assign different levels of importance to each neighbor, addressing the challenge of capturing meaningful information from neighboring nodes. At their core, the designs of these GNN layers contain the same integral parts, message-passing and aggregation, however, due to small modifications in these mechanisms, they can obtain drastically different results given the data used for training.

III. Methodology and Implementation

GNNs, present many challenges when considering the methodology of approach. While the SC Tool, as described in Section II, offers a strong foundation, there are still many issues that need to be addressed. These include defining a systematic and standardized methodology to generate initial node feature vectors, the creation of a comprehensive dataset of graphs derived from existing entries within the GRL, defining the architecture of the GNN models, and developing a comprehensive methodology for assessing the effectiveness of the proposed GNN-based Hardware Trojan detection methods.

A. Defining Node Feature Vectors

Due to the work performed in [2], the SC Tool is already fitted with fully automated parsing and graph generation logic. There are many candidates for initial node features from the parsed HDL, however, some may be more useful than others.. More feature information at the node level should theoretically improve the GNN model's ability to learn features of the graph, however, possessing a feature vector that is too large can hurt performance by increasing computational complexity.

The features currently extracted during the parsing step can be seen in Table I. Boolean indicators and low-dimensional features, such as type and direction, are straightforward choices for the initial node feature vector. However, deciding on the remaining features, especially those associated with names and asset assignments, is more challenging. A one-hot encoding [17] could suffice to embed asset information, but external asset information creates significant overhead as it requires each design to be manually assigned when added to the GRL. Manual external asset assignment can take hours to days for larger designs. This makes it infeasible to expand the library to include thousands or tens of thousands of designs which would be ideal for machine learning tasks. A bag-of-words embedding (BoW) [18] could be used for name-related information; however, further analysis on the number of unique signal names and component names within the GRL is required.

When adding or modifying logic within HDL code to insert a Hardware Trojan, attackers often apply obfuscation techniques to disguise their alterations. A common practice is adding additional malicious signals/components with similar names to those already existing within a circuit. Performing this obfuscation allows the additional or modified logic to go unnoticed, even to individuals that are intricately familiar with the design. However, it is precisely this obfuscation that makes semantic information from name-related data so valuable.

By leveraging semantic information contained within name-related data and other node-level features, identifiers for specific types of signals can be constructed. In HDL, signal and component names often reflect their functionality within the circuit. For instance, the signal name *clk* commonly denotes clock signals within synchronous circuits. Given the ubiquitous usage of *clk* and the consistent implementations of clock signals across designs of various functionalities, signals bearing this name are likely to produce a feature vector that exhibit strong similarities to other clock signal implementations, regardless of design functionality. The consistency of these features can be especially advantageous for identifying Trojan signals. For example, if an obfuscated Trojan signal is named *clk* but performs drastically different functions from those expected of a clock signal, its feature vector would significantly deviate from that of a typical clock signal. This principle extends to functionality-specific signal/component names across a plethora of various design types, and its usefulness led to experimentation with BoW embeddings for name-related data.

TABLE I. EXTRACTED NODE FEATURES

Feature	Description
signal_name	Signal name defined within the HDL
direction	Signal direction (input, output, buffer, linkage, inout, and internal)
is_const	Boolean denoting whether the signal is a constant
is_latched	Boolean denoting if the logic element can sample and hold a binary value
is_shared	Boolean denoting if the signal can share information between processes
is_static	Boolean denoting static signals within the HDL
type	Type definition associated with a signal (e.g., std_logic_vector)
value	Any initial value assigned to a signal
is_library_signal	Boolean denoting if the signal is part of a library definition
num_bits	Number of bits associated with a given signal
parent_module	Parent component of which the signal is defined within
instance_name	Unique instance name assigned to duplicate components

To determine the efficacy of a BoW embedding, frequency analysis of GRL signal/component names was performed. Initial analysis across all GRL designs showed that of 228,508 signal declarations and 18,025 component declarations there existed only 9,510 unique signal names and 361 unique component names. By further analyzing the frequency of name-related data, lists of prominent semantic tokens were generated. This process relied on the frequency of unique names and common tokens used across various design functionalities. While building these lists, it was discovered that many tokens share the same semantic meaning. Examples of common tokens with identical semantic meaning can be seen in Table II. To reduce dimensionality of the BoW embedding, these tokens are combined.

TABLE II. COMMON TOKENS WITH SEMANTIC EQUIVELANTS

Token	Semantically Equivalent
clock	clk
count	cnt
reset	rst
busy	bsy
request	rqst
control	ctrl
source	src
destination	dst
clear	clr

For signal names a 94-dimensional BoW embedding was created from a list of 124 tokens, 30 of which were semantically the same. This embedding allowed for a 66.52% coverage rate for all unique signal names. For component names a 39-dimensional BoW embedding was created from a list of 46 tokens, seven of which were semantically the same. This provided a 57.62% coverage rate of unique component names. Extending either of these BoW embeddings provided diminishing returns as name related data become highly sparse and lacked semantic meaning.

Asset information, unlike name data, has fewer challenges to consider when determining an embedding method. Both external and internal assets have a set dimension within the SC Tool, with 87 external assets and 31 internal assets. Asset information is categorical, having no inherent ranking applied to any one specific asset. This makes asset information a great candidate for one-hot encoding. Due to issues with manual assignment of external assets, two feature sets are defined. The initial feature vectors are generated using an automated feature engineering function. After each feature has been converted based on its encoding scheme, the features are combined, flattened, and converted into PyTorch [19] tensors. This results in two possible feature vectors: a 193-dimensional vector when excluding external assets and a 280-dimensional vector when including external assets.

B. Dataset Creation

To construct a dataset of graphs from existing GRL entries, the selection of a suitable machine learning framework was crucial. After careful consideration, PyTorch Geometric (PyG) [20], a Python library built on PyTorch [19], was chosen. To take advantage of PyG, conversion methods were developed to transform NetworkX graphs, internal to the SC Tool, to PyG's graph representation. Through this process a custom dataset class was added to the SC Tool which facilitated multiple utility functions for on-disk storage and data processing. The primary utility functions developed include PyG graph validation, mini-batched data loader creation, and dataset splitting. The final dataset used for evaluation of the GNN models included 144 designs, of which 95 were clean and 49 were Trojan infested, sourced from Trust-Hub [21][22] and OpenCores [23] designs.

C. Model Implementation

To thoroughly evaluate Hardware Trojan detection using GNNs, three models were developed, each employing different classical GNN layers. These three classical layers include GCN, GraphSAGE, and GAT. All models were designed to be highly modular for the task of graph-level classification. To ensure testing remained consistent, a generalized modular architecture, as seen in Fig. 1, was adopted for all three models. The key distinction among these models is their node-level implementations, with differences in the GNN layers utilized.

The input to each model is mini-batched graph data where each node embedding corresponds to the provided initial feature vector. This data is then sent through a variable number of GNN layers used to generate n^{th}-order node embeddings based on a k-hop neighborhood. Each GNN layer has a different message passing and aggregation mechanism which is represented in PyG. GCN layers are represented by *GCNConv* [14], GraphSAGE by *SAGEConv* [15], and GAT by *GATv2Conv* [24].

Each GNN layer utilizes batch normalization, non-linearity in the form of ReLU, and a dropout mechanism. The n^{th}-order node embeddings are then fed through a mean global pooling mechanism to generate a single graph embedding. A variable number of linear layers are then applied for binary classification using ReLU non-linearity. Finally a sigmoid activation function is applied at the output of the model to generate a probabilistic result of Trojan inclusion. Each model includes multiple configurable parameters which can be seen in Table III.

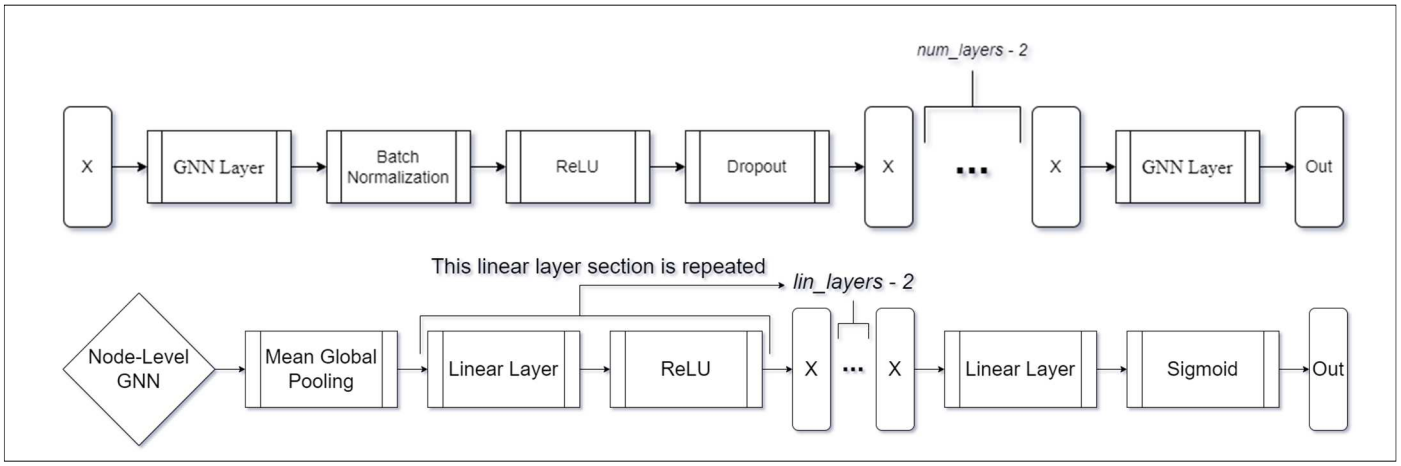

Fig. 1. Modular node-level and graph-level architecture. Node-level architecture can be seen above graph-level architecture with node-level GNN output acting as input to the graph-level architecture.

TABLE III. CONFIGURABLE MODEL PARAMETERS

Parameter	Description
num_layers	Number of GNN layers
input_dim	Input dimension of the node-level model
hidden_dim	Hidden dimension of the node-level model
output_dim	Output dimension of the node-level model
dropout	Dropout percentage used to reduce overfitting
lin_layers	Number of linear layers in the graph-level model
heads	Number of attention heads (GAT specific)

IV. RESULTS AND ANALYSIS

To compare the GNN-based Hardware Trojan detection methods to the SC Tool's current Trojan detection methods it is important to highlight the differences in both the functionality and expected result of each method. Functionally, the structure-based Trojan detection algorithms exhibit the greatest contrast in approach. Each structure-based Trojan detection algorithm targets a specific Trojan taxonomy and leverages structural elements in the unknown design's graph representation, along with user asset information, to flag Trojan behavior. The output of these algorithms consists of a JSON results file which indicates signals that potentially belong to a specific Trojan taxonomy as well as their Trojan purpose in the IC.

When comparing this methodology to the GNN-based approach two major differences can be observed, the first difference being the granularity at which detections are made. Within the structure-based Trojan detection algorithms, a signal-level classification is performed. This differs from the design-level classification performed by the GNN-based approach. The second difference between these two methodologies is the way in which detection is reported. In the GNN-based approach, a probability of Trojan inclusion is given for the design being tested, while in the structure-based Trojan detection algorithms signals are cautiously reported based on their features within the circuit. Cautious reporting is used during this process to reduce the chance of missing potential Trojan signals. Due to the drastic differences in results, comparing these two methodologies is unsuitable.

GRL statistical matching maintains similar functionality and a comparable level of granularity in classification. While both methodologies are used for the classification of Trojan inclusion, the method by which they perform these classifications is quite different. Rather than giving a probability of Trojan inclusion, GRL statistical matching provides similarity scores between the unknown design and both champion and functionality subset GRL entries. This information can be used to determine Trojan inclusion based on highest match percentage and overall similarity between circuits. The combination of these factors makes both methodologies excellent choices for comparison.

A. GNN-Based Trojan Detection

To gather results from the three GNN models, each model was subjected to an automated hyperparameter tuning session using both available feature sets. After identifying ideal hyperparameters, an extended 1000 epoch training session was applied to each model using 10-fold cross validation. Individual batch losses are aggregated to form a single epoch loss which is calculated using binary cross entropy loss [25]. Following this step, the gradient of the backward pass of the neural network is calculated and the optimizer, Adam [26], is used to update model parameters to minimize training loss. Additional evaluation metrics, such as F1 score, accuracy, and Area Under the Receiver Operating Characteristic Curve (ROC AUC) score, are calculated using the scikit-learn Python library [27]. The metrics aggregated from each fold, after 1000 epochs of training, can be seen in Table IV.

TABLE IV. GNN EVALUATION METRICS AT 1000 EPOCHS

	Loss		Accuracy		F1-Score		ROC AUC Score	
	Train	***Test***	***Train***	***Test***	***Train***	***Test***	***Train***	***Test***
GCN	0.1730	0.4001	0.9297	0.8649	0.9059	0.7968	0.9892	0.9405
GCN*	0.3590	0.3524	0.9438	0.8752	0.9278	0.8001	0.9984	0.9289
GraphSAGE	0.0541	0.6379	0.9405	0.8486	0.8856	0.7546	0.9944	0.9456
GraphSAGE*	0.1630	0.6196	0.9636	0.9045	0.9458	0.8725	0.9991	0.9801
GAT	0.1189	0.3943	0.9514	0.8973	0.9338	0.8469	0.9973	0.9456
GAT*	0.0946	0.4195	0.9627	0.9036	0.9431	0.8685	1.000	0.9552

[a.] * - includes both external and internal assets in the model's feature sets while training.

From the results observed in Table IV, several observations can be made regarding the performance of the GNN models. Regardless of whether external asset information was included or excluded from the initial node feature vectors, each model achieved impressive results. The best model from this selection was the GAT which excluded external asset information. Overall, models that excluded external asset information tended to be more stable and performed slightly better than their inclusive counterparts.

B. GRL Matching

To gather evaluation metrics comparable to those found within the GNN models, a GRL matching evaluator was developed. This evaluator begins by iterating through each design in the GRL to gather data from the matching process. GRL matching is heavily dependent on the size and diversity of designs found within the GRL. For this reason, a leave-one-out (LOO) approach is used when performing matching. Because of the deterministic nature of the GRL matching algorithm, only a single pass of each design is necessary to determine the algorithm's effectiveness.

Internally, it is recommended to review more than just the highest match to prevent any potential Trojans from being overlooked. To account for this, additional analysis was performed to observe the top-three highest match percentages with cautious reporting. Cautious reporting in this context refers to the method by which a design is classified. Should a Trojan design be found within the top-three highest match percentages, the unknown design is marked as having been classified as Trojan-infested. This is then compared to designs actual classification. Using this method of evaluation GRL matching obtained a Trojan inclusion classification accuracy of 67.67% with an F1 score of 0.5057.

C. Direct Comparisons

When comparing both GNN-based Trojan detection and GRL matching, the Trojan inclusion classification abilities of the GNN approach far exceed the current capabilities of GRL matching. All six models, seen in Table IV, outperformed GRL matching. The best performing model, GAT, achieved a 32.60% increase in accuracy and a 67.47% increase in F1 score. The worst performing model, GraphSAGE was still able to achieve a 25.40% increase in accuracy and a 49.22% increase in F1 score. Test accuracy and F1 score are used in these comparisons as they demonstrate how the model performs on data that it has not been trained on.

Another benefit of the GNN-based approach is the speed at which Trojan detection can be performed. Although training the model can take a significant amount of time, this process is performed in advance and is a one-time operation. Once a model has been trained, the time required to perform Trojan detection on a single design is typically less than a second. While training will take longer as the GNN's dataset grows, the time required to perform Trojan detection should remain essentially the same, assuming model configurations remain constant. GRL matching, however, can take upwards of 30 minutes to complete for larger designs. As the GRL expands, the time required to complete matching will only increase.

V. CONCLUSION AND FUTURE WORK

Through the creation of multiple modular graph-level GNNs, the SC Tool's Trojan detection capabilities have been expanded. This research has demonstrated that GNN-based Trojan detection offers a faster and more accurate method of design-level Trojan inclusion classification compared to the SC Tool's pre-existing GRL matching technique. Furthermore, the use of external asset information within initial node feature vectors yielded similar or inferior results compared to models excluding the asset information. Automatic feature extraction of ICs added to the GRL substantially increases the rate at which the dataset can grow, while also removing the chance of human error or external asset assignment biases that existed within the original GRL. Going forward this methodology can be expanded to node-level Trojan classification. This would provide the ability to directly identify Trojan signals within RTL code allowing for comparisons to the SC Tools structure-based Trojan detection algorithms. Additionally, multi-class graph-level classification could be used to identify both the functionality of a circuit and its Trojan inclusion probability.

REFERENCES

[1] H. S. Choo et al., "Machine-Learning-Based Multiple Abstraction-Level Detection of Hardware Trojan Inserted at Register-Transfer Level," 2019 IEEE 28th Asian Test Symposium (ATS), Kolkata, India, 2019, pp. 98-980, doi: 10.1109/ATS47505.2019.00018.

[2] Taylor, D. (2022). SC Tool Restructure and Matching Improvements. Graduate Theses and Dissertations Retrieved from https://scholarworks.uark.edu/etd/4494

[3] S. Adee, "The Hunt For The Kill Switch," in IEEE Spectrum, vol. 45, no. 5, pp. 34-39, May 2008, doi: 10.1109/MSPEC.2008.4505310.

[4] Skorobogatov, S., Woods, C. (2012). Breakthrough Silicon Scanning Discovers Backdoor in Military Chip. In: Prouff, E., Schaumont, P. (eds) Cryptographic Hardware and Embedded Systems – CHES 2012. CHES 2012. Lecture Notes in Computer Science, vol 7428. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-33027-8_2

[5] J. Yust, M. Hinds and J. Di, "Structural Checking: Detecting Malicious Logic without a Golden Reference," Journal of Computational Intelligence and Electronic Systems, vol. 1, no.2, p 169177, 2012

[6] Del Carmen, R. D. (2022). Framework of Hardware Trojan Detection Leveraging SC Tool. Graduate Theses and Dissertations Retrieved from https://scholarworks.uark.edu/etd/4462

[7] Nic30, “GitHub - Nic30/hdlConvertor: Fast Verilog/VHDL parser preprocessor and code generator for C++/Python based on ANTLR4,” GitHub, 2016. https://github.com/Nic30/hdlConvertor.

[8] antlr, “GitHub - antlr/antlr4: ANTLR (ANother Tool for Language Recognition) is a powerful parser generator for reading, processing, executing, or translating structured text or binary files.,” GitHub, Sep. 04, 2023. https://github.com/antlr/antlr4.

[9] Aric A. Hagberg, Daniel A. Schult and Pieter J. Swart, “Exploring network structure, dynamics, and function using NetworkX”, in Proceedings of the 7th Python in Science Conference (SciPy2008), Gäel Varoquaux, Travis Vaught, and Jarrod Millman (Eds), (Pasadena, CA USA), pp. 11–15, Aug 2008

[10] L. Weaver, T. Le and J. Di, "Golden Reference Library Matching of Structural Checking for securing soft IPs," SoutheastCon 2016, 2016, pp. 1-7, doi: 10.1109/SECON.2016.7506737.

[11] B. McGeehan, F. Smith, T. Le, H. Nauman and J. Di, "Hardware IP Classification through Weighted Characteristics," 2019 IEEE High Performance Extreme Computing Conference (HPEC), Waltham, MA, USA, 2019, pp. 1-6, doi: 10.1109/HPEC.2019.8916225.

[12] N. Waller, H. Nauman, D. Taylor, R. Del Carmen and J. Di, "Character Reassignment for Hardware Trojan Detection," 2021 IEEE International Midwest Symposium on Circuits and Systems (MWSCAS), Lansing, MI, USA, 2021, pp. 861-864, doi: 10.1109/MWSCAS47672.2021.9531813.

[13] Chapman, Z. (2023). Trojan Detection Expansion of Structural Checking. Graduate Theses and Dissertations Retrieved from https://scholarworks.uark.edu/etd/5171

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

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

[16] Veličković, Petar, et al. "Graph attention networks." arXiv preprint arXiv:1710.10903 (2017).

[17] S. J. Russell and P. Norvig, Artificial intelligence : a modern approach, Fourth edition. Hoboken, N.J: Pearson Education, 2020, pp. 707.

[18] W. A. Qader, M. M. Ameen and B. I. Ahmed, "An Overview of Bag of Words; Importance, Implementation, Applications, and Challenges," 2019 International Engineering Conference (IEC), Erbil, Iraq, 2019, pp. 200-204, doi: 10.1109/IEC47844.2019.8950616.

[19] Paszke, A. et al. PyTorch: an imperative style, high-performance deep learning library. Adv. Neural Inf. Process. Syst. 32, 8026–8037 (2019).

[20] Fey, M., & Lenssen, J. E. (2019). Fast Graph Representation Learning with PyTorch Geometric [Computer software]. https://github.com/pyg-team/pytorch_geometric

[21] H. Salmani, M. Tehranipoor, and R. Karri, "On Design vulnerability analysis and trust benchmark development", IEEE Int. Conference on Computer Design (ICCD), 2013.

[22] B. Shakya, T. He, H. Salmani, D. Forte, S. Bhunia, M. Tehranipoor, “Benchmarking of Hardware Trojans and Maliciously Affected Circuits”, Journal of Hardware and Systems Security (HaSS), April 2017.

[23] OpenCores. Available: http://opencores.org/

[24] S. Brody, U. Alon, and E. Yahav, “How Attentive are Graph Attention Networks?,” arXiv.org, 2021. https://arxiv.org/abs/2105.14491.

[25] “BCELoss — PyTorch 2.2 documentation,” Pytorch.org, 2023. https://pytorch.org/docs/stable/generated/torch.nn.BCELoss.html.

[26] “Adam — PyTorch 2.2 documentation,” Pytorch.org, 2023. https://pytorch.org/docs/stable/generated/torch.optim.Adam.html.

[27] “scikit-learn: machine learning in Python — scikit-learn 1.4.1 documentation,” Scikit-learn.org, 2024. https://scikit-learn.org/stable/index.html (accessed Mar. 22, 2024).