GNN4TJ: Graph Neural Networks for Hardware Trojan Detection at Register Transfer Level

Rozhin Yasaei*, Shih-Yuan Yu*, Mohammad Abdullah Al Faruque
Department of Electrical Engineering and Computer Science
University of California, Irvine, California, USA
(ryasaei, shihyuay, alfaruqu)@uci.edu

***Abstract*—The time to market pressure and resource constraints has pushed System-on-Chip (SoC) designers toward outsourcing the design and using third-party Intellectual Property (IP). It has created an opportunity for rogue entities in the Integrated Circuit (IC) supply chain to insert malicious circuits in the hardware design, known as Hardware Trojans (HT). HT detection is a major hardware security challenge, and its early discovery is crucial because postponing the removal of HT to late in design or after the fabrication process would be very expensive. Current works suffer from several shortcomings such as reliance on a golden HT-free reference, unable to identify all types of HTs or unknown ones, burdening the designer with the manual review of code, or scalability issues. To overcome these limitations, we propose GNN4TJ, a novel golden reference-free HT detection method in the register transfer level (RTL) based on Graph Neural Network (GNN). GNN4TJ represents the hardware design as its intrinsic data structure, a graph, and generates the data flow graphs for RTL codes. We utilize GNN to extract the features from DFG, learn the circuit's behavior, and identify the presence of HT, in a fully automated pipeline. We evaluate our model on a dataset that we create by expanding the Trusthub [1] HT benchmarks. The results demonstrate that GNN4TJ detects unknown HT with 97% recall (true positive rate) very fast in 21.1ms.**

***Index Terms*—Hardware Trojan Detection, Security, Graph Neural Network, Golden Reference-Free, Register Transfer Level.**

I. Introduction

The time to market pressure and resource constraints has pushed SoC designers toward using third-party Electronic Design Automation (EDA) tools and IP cores and outsourcing design, fabrication, and test services worldwide. The globalization of the semiconductor industry has raised the risk of insertion of hardware Trojans by rogue entities in the IC supply chain, refer to Figure 1. Consequently, HT detection has become one of the major hardware security concerns. HT refers to any intentional and malicious modification of IC that is usually designed to leak the information, change the functionality, degrade the performance, or deny the service of the chip. For example, in 2008, a suspected nuclear installation in Syria was bombed by Israeli jets because Syrian radar was disabled by a backdoor in its commercial off-the-shelf microprocessors [2].

To ensure the trustworthiness of SoC design, it is essential to ascertain the authenticity of 3PIPs. Depending on the format of the IP, it is classified into three categories; Soft IP (i.e., synthesizable Verilog or VHDL source codes), Firm IP (i.e., netlists and placed RTL blocks), and Hard IP (i.e., GDSII files and custom physical layout). IP trust mainly revolves around HT detection in soft IP cores since it is the most flexible and widely-used IP core in practice. Additionally, the high level of abstraction and flexibility in RTL codes ease the design and implementation of various malicious functions for attackers.

It is crucial to identify HT in the early stages of design flow because removing it would be very expensive later. Detecting a few lines of HT in an industrial-strength IP with thousand lines of RTL code is extremely challenging and any work which requires manual review is error-prone, time-consuming, and not scalable. The need for a scalable pre-silicon security verification method is highlighted in many technical documents, such as a recent white paper from Cadence and Tortuga Logic [5] because existing solutions fail to guarantee the trustworthiness of the hardware design as elaborated in the Section II.

A. Motivational Example

Despite numerous HT detection methods proposed in the literature, the problem still exists because as the designers develop a new defense mechanism against the existing HTs, the attackers design new HTs to evade detection. For instance, the FANCI [23] was successful in HT detection until DeTrust [25] designed a new HT to bypass it. Among Trusthub benchmarks, [18] marks AES-T600 as HT but fails for AES-700 while both HTs have the same trigger and leak the secret key differently. Later, [19] identify the HT in AES-600 and AES-700 by data leakage detection. Nevertheless, it fails for AES-500 that degrades the chip instead of data leakage. A recent paper [6] demonstrated that modeling the hardware design as a graph can be beneficial in the hardware security domain. However, the graph similarity algorithms used by [6], [17] limit the detection scope to known HTs in the method's library. The story goes on, and there is no universal method for detecting all types of HTs. Thus, a new scalable methodology is required for unknown HT detection that is expandable as new HTs are introduced.

B. Research Challenges and Technical Contributions

The existing HT detection solutions have several shortcomings; reliance on golden-reference, unable to identify unknown HTs, burdening the designer with a manual review of code, unable to guarantee HT detection, limited detection scope to some specific type of HTs, not scalable, or too complex.

To overcome these limitations, **we propose a novel golden reference-free pre-silicon HT detection method that learns the circuit behavior and identifies the HTs due to their malicious behavior. Since hardware is a non-euclidean, structural type of data in nature, we use the graph data structure as a great match for hardware representation and**

* Both are equal contributing authors. Yasaei is the corresponding author.

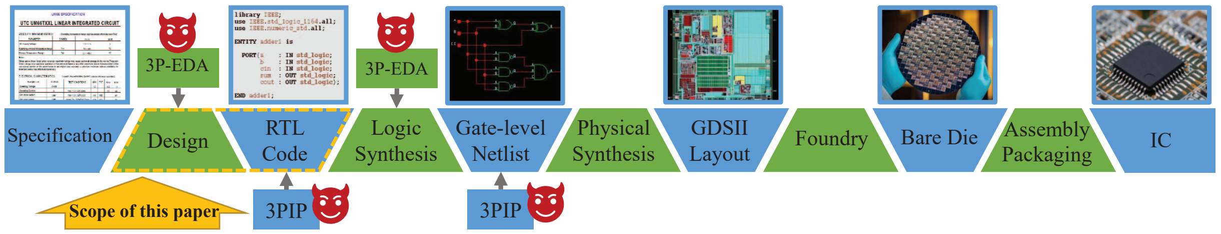


Fig. 1. Semiconductor supply chain, pre-silicon HT injection points.

generate the Data Flow Graphs (DFG) for RTL codes. We leverage the Graph Neural Networks (GNN) to model the behavior of the circuit. The scalability of our method lies in the fully automated process of graph generation, feature extraction, and detection without any manual workload. Automatic feature extraction is crucial when new HTs are identified. In this case, our model can be readily updated without any need for defining new properties or feature engineering as previous works. Our contributions are outlined below:

- We model the hardware design as its intrinsic representation, the graph, and for the first time, we leverage graph neural network to learn the behavior of the circuit.
- We propose a novel, fully automated method that extracts the features from DFG of the circuit and detects the presence of HTs, even unknown ones, without any need for trusted HT-free reference or manual workload from the designer. Our model is faster than existing methods and scalable to be used for large designs.
- We create a DGF dataset of RTL HT-infested circuits by expanding the benchmarks obtained from Trusthub.
- We survey the pre-silicon HT detection techniques in the literature, analyze the state of the art, and make a comprehensive comparison with our approach.

II. Related Works

The majority of the pre-silicon HT detection techniques in the literature fall into four main categories, described below:

*1) **Test Pattern Generation**:* Due to its stealthy nature, HT remains inactive and well-hidden during the simulation and testing. It gets activated by a particular event under rare circumstances. Automatic test pattern generation methods [20] attempt to generate effective test vectors to increase the probability of triggering the HT. Still, there is no guarantee of their success because it is infeasible to test all the possible conditions.

*2) **Formal Verification (FV)**:* It is an algorithmic method which converts the 3PIP to a proof checking format and checks if the IP satisfies some predefined security properties. [12], [21] identify the design bugs using formal verification but cannot detect HT. This method is also employed to detect information leakage [19] and malicious modification to registers [18]. Although formal verification proves the predefined security properties in IP, its detection scope is limited to certain types of HTs because the properties are not comprehensive enough to cover all the different kinds of malicious behaviors that HTs may have. For example, [18] defines the security property as ”no-critical-data-corruption” which only detects data-corrupting HTs. Some works employ model checking, such as [19] which are not scalable to large designs because the model checking is NP complex and suffers from state explosion. Information flow tracking is also utilized for modeling the security properties [3], [9], [16].

*3) **Code Analysis (CA)**:* The code coverage techniques analyze the RTL code using metrics such as line, statement, finite state machine, and toggle coverage to ascertain the suspicious signals that imitate the HT. These coverage metrics respectively check which lines and statements are executed, which states in the finite state machine are reached, or if signals are switched. This approach burdens the designer with manual analysis of the suspicious regions to identify the possible HT. This analysis is limited to the design's combinational logic, and even 100% coverage does not guarantee that IP is HT-free. FANCI [23] flags the nets with low activation probability as suspicious. VeriTrust [26] returns the nets that are not driven by functional inputs as potential triggers for an HT. DeTrust [25], on the other hand, proposes an attack that exploits the vulnerabilities of FANCI and VeriTrust and bypasses them.

*4) **Machine Learning (ML) and Graph Matching (GM)**:* The graph is an intuitive representation for a hardware design that is recently used for security purposes. [17] propose analyzing the data/control flow graph of the circuit to pinpoint the HTs. The authors create a library of HTs and search for the graph of those known HTs graphs in the graph of 3PIP using sub-graph matching algorithms. Graph matching is an NP-complex problem that does not apply to large designs. [6] introduces a new graph similarity heuristic customized for hardware security to improve accuracy and computation time. These approaches can detect only the HTs with the same graph representation as known HTs in their library, while in practice, attackers design a variety of HTs. Machine learning is a powerful technique that recently has grabbed a lot of attention. Most of the ML-based works for HT detection rely on side-channel signals which postpone the detection until fabrication [10]. Recently, some approaches use ML techniques instead of graph matching to classify the HT-free and HT-infested IPs by extracting features from the graph representation of the circuit. For example, gradient boosting algorithm on Abstract Syntax Tree [7], multilayer neural network on gate-level netlist [8], artificial immune system on DFG and Control Flow Graph (CFG) [24], probabilistic neural network on CFG [4]. Most of these models rely on an HT-free reference called the golden reference. However, a trusted reference is not available because the IP vendors only provide the source code and specifications that may contain HTs. Moreover, reference-based methods may be inconclusive or too complex for exhaustive verification, especially for large designs. We propose a scalable, golden

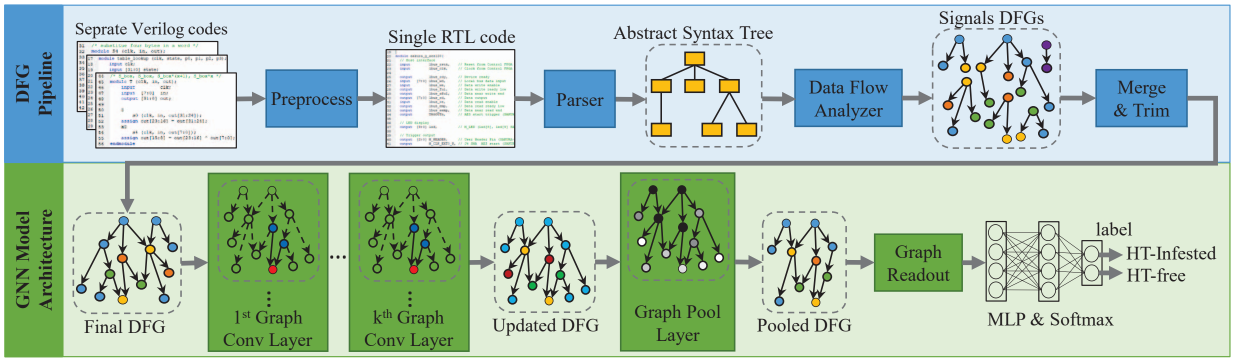


Fig. 2. The methodology work flow including a DFG generation pipeline and GNN model architecture.

reference-free model for unknown HT detection that leverages a potent ML technique, GNN, to learn the circuit's behavior.

III. GNN4TJ: Graph Neural Network for Hardware Trojan Detection

We assume the existence of a feed-forward function f that outputs whether a RTL program p has a HT or not through a label y as given in Equation 1.

$$y = f(p) = \begin{cases} (1,0), & \text{if the program has a HT} \\ (0,1), & \text{if the program has no HT,} \end{cases} \qquad (1)$$

In our methodology, we propose a model for approximating function f. The overall architecture of GNN4TJ is illustrated in Figure 2. The first step is to extract the DFG G from each RTL program p that is achieved by a *DFG generation pipeline*, described in Section III-B. In the second step, these DFGs are passed to a GNN framework that includes graph convolution layers and an attention-based graph pooling layer. The graph-level representation for a DFG G, denoted as $\mathbf{h_G}$, is then acquired by a graph readout operation. Lastly, we use a Multi-Layer Perceptron (MLP) with a Softmax activation function to process $\mathbf{h_G}$ and to compute the final HT inference, denoted as $\hat{y}$. We describe some background information about GNN in Section III-C and provide more details regarding the components of our model in Section III-D.

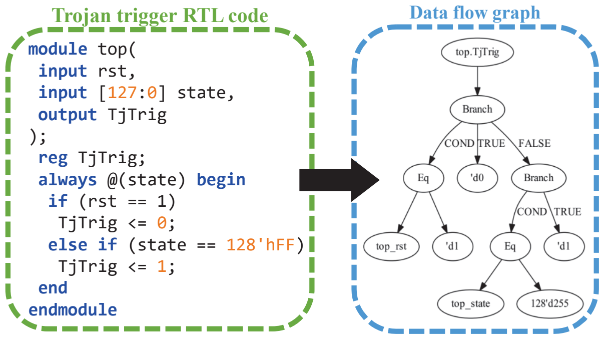
A. Threat Model

In this paper, four scenarios of HT insertion are considered: (1) 3PIP vendor is not trusted, and a malicious circuit is hidden in the IP. (2) The 3P-EDA used for design synthesis and analysis inserts an HT automatically. (3) A rogue in-house designer intentionally manipulates the RTL design. (4) The design is tampered with in the later stages (gate-level netlist, physical layout, or bare die), and the RTL design is derived using reverse-engineering techniques.

B. Data Flow Graph Generation

The hardware design is a non-euclidean, structural type of data similar to the graph data structure. As RTL representation, we use DFG, which indicates the relationships and dependencies between the signals and gives a fundamental expression of the code's computational structure. As pictured in Figure 2, our automated DFG generation pipeline comprises several phases: preprocess, parser, data flow analyzer, and merge. An RTL code consists of several modules in separate files, and the preprocessing step is needed to flatten the design and resolve syntax incompatibilities. Next, we use a hardware design toolkit, called Pyverilog [22], in which a parser extracts the abstract syntax tree from Verilog code and passes it to the data flow analyzer to generate a Tree for each signal in the circuit such that the signal is the root node. To have a single graph representation for the whole circuit, we combine all the signal DFGs. Further, we trim the disconnected sub-graphs and redundant nodes in the merge phase.

The final DFG is a rooted directed graph that shows data dependency from the output signals (the root nodes) to the input signals (the leaf nodes). It is defined as graph $G = (V, E)$ where E is the set of directed edges and V is the set of vertices. We define $V = \{v_1, v_2, ..., v_n\}$ where v_i represents signals, constant values, and operations such as xor, and, concatenation, branch, or branch condition. We define $E = e_{ij}$ for all i, j such that $e_{ij} \in E$ if the value of v_i depends on the value of v_j, or if the operation v_j is applied on v_i. An RTL code and its DFG are exemplified in Figure 3.



```
module top(
 input rst,
 input [127:0] state,
 output TjTrig
);
 reg TjTrig;
 always @(state) begin
  if (rst == 1)
   TjTrig <= 0;
  else if (state == 128'hFF)
   TjTrig <= 1;
 end
endmodule
```

Fig. 3. The RTL code of a Trojan trigger and its DFG.

C. Graph Neural Networks (GNNs)

GNN is deep learning that operates on graphs. The architecture used in GNN4TJ is inspired by the Spatial Graph Convolution Neural Network (SGCN). In general, SGCN defines the convolution operation based on a node's spatial relations. The spatial convolution has the following phases: (i) *message propagation phase* and (ii) the *read-out phase*. The *message propagation* phase involves two sub-functions: **AGGREGATE** and **COMBINE** functions, given by,

$$a_v^{(k)} = \textbf{AGGREGATE}^{(k)}(\{h_u^{(k-1)} : u \in N(v)\}), \qquad (2)$$

$$h_v^{(k)} = \textbf{COMBINE}^{(k)}(h_v^{(k-1)}, a_v^{(k)}), \qquad (3)$$

where $h_v^{(k)} \in R^{C^k}$ denotes the feature vector after k iterations for the v-th node. Essentially, the **AGGREGATE** collects the features of the neighboring nodes to extract an aggregated feature vector $a_v^{(k)}$ for the layer k, and the **COMBINE** combines the previous node feature $h_v^{(k-1)}$ with $a_v^{(k)}$ to output next feature vector $h_v^{(k)}$. This message propagation is carried out for a pre-determined number of iterations k. Next, in the *read-out* phase, the overall graph-level feature extraction is carried out by either summing up or averaging up the node features in each iteration. We employ the graph embedding $h_G^{(k)}$ to model the behavior of circuits and identify HT. $h_G^{(k)}$ is defined as,

$$h_G^{(k)} = \textbf{READOUT}(\{h_v^{(k-1)} : v \in G\}) \tag{4}$$

D. Graph Learning Pipeline

In GNN4TJ, we model hardware design by firstly extracting the DFG from the circuit and then applying the graph learning pipeline to acquire the embedding of hardware designs. To process each circuit in its DFG form, we employ the GNN framework as introduced in Section III-C with the architecture depicted in Figure 2. In the phase of message propagation, the layer that we refer to process G is Graph Convolution Network (GCN) [13]. In each iteration l of *message propagation*, a GCN layer updates the node embeddings $\mathbf{X^{l+1}}$ as follows,

$$\mathbf{X^{(l+1)}} = \sigma(\widehat{D}^{-\frac{1}{2}}\widehat{A}\widehat{D}^{-\frac{1}{2}}\mathbf{X^{(l)}}W^{(l)}) \tag{5}$$

where W^l is a trainable weight used in the GCN layer. $\widehat{A} = A + I$ is the adjacency matrix of G used for aggregating the feature vectors of the neighboring nodes where I is an identity matrix that adds the self-loop connection to make sure the features calculated in previous iteration will also be considered in the current iteration. $\widehat{D}$ is the diagonal degree matrix used for normalizing $\widehat{A}$. $\sigma(.)$ is the activation function such as Rectified Linear Unit (ReLU). Here, we initialize the embedding $\mathbf{X}_i^{(0)}$ for each node $i \in V$ by converting the node's name to its corresponding one-hot encoding. We denote the final propagation node embedding $\mathbf{X}^{(l)}$ as $\mathbf{X}^{prop}$.

The node embedding $\mathbf{X}^{prop}$ is further processed with an attention-based graph pooling layer. As stated in [14], such an attention-based pooling layer allows the model to focus on a local part of the graph and is considered as a part of a unified computational block of a GNN pipeline. In this layer, we perform *top-k filtering* on nodes according to the scores predicted from a separate trainable GNN layer [15], as follows:

$$\alpha = \textbf{SCORE}(\mathbf{X}^{prop}, \mathbf{A}), \ \mathbf{P} = \mathrm{top}_k(\alpha) \tag{6}$$

where α stands for the coefficients predicted by the graph pooling layer for nodes. $\mathbf{P}$ represents the indices of the pooled nodes which are selected from the top k of the nodes ranked according to α. The number k used in *top-k filtering* is calculated by a pre-defined pooling ratio, pr using $k = pr \times |O_t|$, where we consider only a constant fraction pr of the embeddings of the nodes of the DFG to be relevant (i.e., 0.5). We denote the node embeddings and edge adjacency information after pooling by $\mathbf{X}^{pool}$ and $\mathbf{A}^{pool}$ which are calculated as follows:

$$\mathbf{X}^{pool} = (\mathbf{X}^{prop} \odot \tanh(\alpha))_{\mathbf{P}}, \ \mathbf{A}^{pool} = \mathbf{A}^{prop}{}_{(\mathbf{P},\mathbf{P})} \tag{7}$$

where $\odot$ represents an element-wise multiplication, $()_{\mathbf{P}}$ refers to the operation that extracts a subset of nodes based on P and $()_{(\mathbf{P},\mathbf{P})}$ refers to the information of the adjacency matrix between the nodes in this subset. Then, our model aggregates the node embeddings acquired from the graph pooling layer, $\mathbf{X}^{pool}$, to acquire the graph-level embedding $\mathbf{h}_G$ for DFG G.

$$\mathbf{h}_G = \textbf{READOUT}(\mathbf{X}^{pool}) \tag{8}$$

where the **READOUT** operation can be either summation, averaging, or selecting the maximum of each feature dimension, over all the node embeddings, denoted as *sum-pooling*, *mean-pooling*, or *max-pooling* respectively. Lastly, our model processes the embedding $\mathbf{h}_G$ of a RTL code with a MLP layer and a Softmax activation function to produce the final prediction $\hat{Y} = \mathbf{Softmax}(\mathbf{MLP}(\mathbf{h}_G))$. This layer reduces the number of hidden units used in $\mathbf{h}_G$ and generates a 2-dimensional output that represents the probabilities of both classes (HT or non-HT). Finally, the predicted values in $\hat{Y}$ are normalized using Softmax function and the class with higher predicted value will be taken as the result of detection. To train the model, we compute the cross-entropy loss function, denoted as H, between ground-truth label Y and the predicted label $\hat{Y}$, described as follows,

$$H(Y, \hat{Y}) = \sum_{y_i \in Y, \hat{y_i} \in \hat{Y}} y_i log_e(\hat{y_i}) \tag{9}$$

IV. Evaluation

In this section, we explain our dataset and evaluation, report the results, and compare GNN4TJ with state of the art. During the experiments, we use 2 GCN layers with 200 hidden units for each layer. For the graph pooling layer, we use the pooling ratio of 0.8 to perform *top-k filtering*. For **READOUT**, we use *max-pooling* for aggregating node embeddings of each graph. GNN4TJ uses 1 MLP layer that reduces the number of hidden units from 200 to 2 used in $\mathbf{h}_G$ for predicting the result of HT detection. In training, we append a dropout layer with a rate of 0.5 after each GCN layer. We train the model for 200 epochs using the batch gradient descent algorithm with batch size 4 and the learning rate 0.001.

A. Trojan Benchmarks and Dataset Creation

A HT consists of two fundamental parts: payload and trigger. The payload is the implementation of the malicious behavior of HT to leak data leakage, change the functionality, deny the service, or degrade the performance. The trigger is the optional circuit that monitors various signals or events in the base circuit and activates the payload when a specific signal or event is observed. We classify the triggers of HT benchmarks into three main categories; i) time bomb, ii) cheat code, and iv) always on. The time bomb trigger is activated after numerous clock cycles and the cheat code trigger depends on one or a sequence of specific inputs in the base circuit.

We create a dataset based on the benchmarks in Trusthub [1] that contain 34 varied types of HTs inserted in 3 base circuits: AES, PIC, and RS232. To expand our dataset, we extract the HT design and use them as additional data instances. Moreover, we integrate some of the extracted HTs to new HT-free circuits, DES and RC5, which increases the HT-infested samples. To balance the dataset and increment the number of HT-free samples, we add the different variations of current circuits in addition to new ones, including DET, RC6, SPI, SYN-SRAM, VGA, and XTEA to the dataset that results in a dataset of HT-Free and HT-infested codes with 100 instances.

TABLE I
COMPARING THE PERFORMANCE OF GNN4TJ WITH THE START-OF-ART METHODS FOR HT DETECTION IN 3PIP.

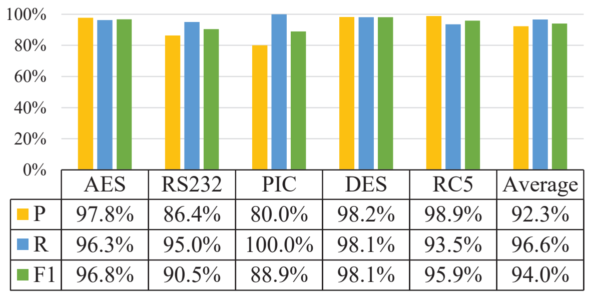
Paper (year)	**Technique category - Method**	**Precision**	**Recall**	**Time (s)**	**Automated feature extraction**	**Golden reference free**	**Unknown Trojan detection**	**HT detection scope is limited to**
GNN4TJ	ML - Graph neural network	92%	97%	0.026	✓	✓	✓	-
[24] (2018)	ML - Artificial immune system	87%	85%	NA	✗	✓	✓	-
[7] (2019)	ML - Gradient boosting algorithm	NA	100%	1.36	✗	✗	✓	-
[8] (2017)	ML - Multi-layer neural networks	NA	90%	NA	✗	✗	✓	-
[4] (2017)	ML,GM - Subgraph isomorphism	NA	100%	1.15	✗	✓	✗	HTs in the library
[6] (2020)	GM - Graph similarity	NA	NA	NA	✗	✓	✗	-
[17] (2017)	GM - Subgraph matching	NA	100%	5.02	✗	✓	✗	HTs in the library
[11] (2019)	CA - Socio-network analysis	98%	98%	NA	✗	✓	✗	combinational HTs
[16] (2017)	FV - Information flow analysis	NA	100%	292.85	✗	✓	✗	-
[19] (2016)	FV - Model checking	NA	100%	96.13	✗	✓	✗	HTs that leak data

B. HT Detection

We want to evaluate GNN4TJ performance in unknown HT detection. To detect the HTs in a base circuit, we leave the base circuit benchmarks for testing and train the model with the other benchmarks. In this setup, the test circuit and Trojans are unknown to the model. We repeat it 20 times for each base circuit and count True Positive (TP), False Negative (FN), and False Positive (FP) to calculate the evaluation metrics, Precision (P), Recall (R) or True Positive Rate, and F_β *score* as follows:

$$P = \frac{TP}{TP+FP}, R = \frac{TP}{TP+FN}, F_\beta score = \frac{(1+\beta^2)*P*R}{\beta^2*P+R}$$

This testing scenario imitates the situation that neither the HT is known nor the golden HT-free reference of the circuit under test is seen by the model in training. Recall expresses the model's ability to detect the HTs or, in other words, what percentage of HT-infested samples are detected by the model. This metric is intuitive, but it is not sufficient for evaluation, especially in an unbalanced dataset. It is essential to analyze the false positive, the numbers of HT-free samples incorrectly classified as HT. Thus, we compute precision that demonstrates what percentage of the samples that model classify as HT-infested actually contains HT. F_β *score* is the weighted average of precision and recall that better presents performance. Dataset plays an important role in the performance of our model. As it is pictured in Figure 4, GNN4TJ has higher scores for AES, RC5, and DES because these circuits are all encryption cores and the model have more data instances to learn their behavior. Similarly, the performance for RS232 and PIC can be enhanced by gathering a more comprehensive dataset with similar types of circuits to PIC and RS232.



	AES	RS232	PIC	DES	RC5	Average
🟨 P	97.8%	86.4%	80.0%	98.2%	98.9%	92.3%
🟦 R	96.3%	95.0%	100.0%	98.1%	93.5%	96.6%
🟩 F1	96.8%	90.5%	88.9%	98.1%	95.9%	94.0%

Fig. 4. The performance of GNN4TJ in HT detection.

C. Timing

We train and test the model on a server with two NVIDIA TITAN-XP and NVIDIA GeForce GTX 1080 graphics cards. The timing results for the base circuits are summarized in Table II. Although training can be time-consuming, it occurs once, and the trained model finds the HT very fast in 21.1ms on average faster than current works.

TABLE II
TIMING OF HT DETECTION PER SAMPLE AND TRAINING.

Bench-mark	AES	RS232	PIC	DES	RC5	**Average**
Detection time(s)	0.0268	0.0169	0.0182	0.0222	0.0215	**0.0211**
Training time(s)	220.37	251.18	274.18	261.71	284.53	**258.39**

D. Comparison with State of the Art

In this section, we compare leading HT detection techniques in a high level of abstraction, RTL, or gate-level netlist, in terms of quantitative and qualitative metrics as well as in a case study.

1) ***Qualitative comparison:*** As qualitative metrics, we investigate 3 essential characteristics; i) golden reference-free, ii) unknown HT detection, and iii) automated process. The construction of an HT detection model without white-box knowledge of the IP is very challenging. An effective and uniform approach should not rely on any trusted reference design since it is not available in reality. Generally, the algorithmic approaches satisfy this quality while ML techniques fail. Another important factor is the capability to find different types of HTs, even the unknown ones. Regarding this factor, ML-based approaches are more successful in extracting the intuitive features from circuits to recognize a variety of HTs. The detection scope of other techniques is limited to the HTs that already exist in the method's library [6], [17] or have the properties that are predefined [16], [19]. To the best of our knowledge, all the existing approaches require feature engineering or manual property definition. However, in our approach, not only HT detection but also the feature extraction process is automated. Hence, our model can be easily expanded by training on various circuits and being scaled for industrial-level IP design. Furthermore, our model gets updated by retraining if a new type of HT is discovered, and it does not require manual effort to extract the new set of properties or features.

2) ***Quantitative comparison:*** As quantitative metrics, we look into precision, recall, and timing. For timing, we compare

TABLE III
COMPARING HT DETECTION METHODS IN A CASE STUDY.

Benchmark	Trigger Payload	Ours (ML)	[6] (GM)	[17] (GM)	[19] (FV)	[16] (FV)
AES-900	Time bomb Leak data	✓✓	✓	✓✓	✓✓	✓✓
RS232-T500	Time bomb Deny service	✓✓	✓	✓✓	✗	✓✓
AES-1900	Time bomb Degrade chip	✓✓	✓	✓✓	✗	✗
AES-2000	Cheat code Leak data	✓✓	✗	✗	✓✓	✓✓
RS232-T700	Cheat Code Deny service	✓✓	✗	✗	✗	✓
AES-1800	Cheat code Degrade chip	✓✓	✗	✗	✗	✗

the average detection time for the AES benchmarks. Since the papers' computing platforms are different, the exact comparison between the reported timing numbers is not reasonable. However, the order of magnitude of timing results is useful for general comparison, and it shows our model is much faster than the others. The timing of algorithmic methods (FV, CA, and GM) highly depends on hardware design complexity. Their memory usage and detection time drastically grow for large designs that cause timeout or memory shortage problems. On the other hand, the circuit's complexity does not have a notable effect on the detection time of GNN4TJ according to Table I, which makes it scalable for a large design.

In terms of recall and precision, we have comparable results. Comparison with algorithmic methods is challenging because, in most cases, the authors report a list of known HT benchmarks that their model successfully detects while we test our model on unknown HTs, which mostly are not detectable by those algorithms. Furthermore, FP and TP for computing precision and the performance of the methods on HT-free samples are not available in some papers to compare.

*3) **Case study**:* We analyze the ML, GM, and FV techniques in a case study and investigate if the models detect 6 different types of HTs. In Table III, the first top 3 HT benchmarks are known to the Method Under Test (MUT) and exist in its library of HTs whereas the other 3 HTs are unknown. ✓✓indicates that the MUT can detect the HT and it is explicitly reported in its paper. ✓\✗shows that this case is not tested in the paper of but it is supposed to detect\not to detect the HT according to authors' claims and assumptions. GM methods rely on the HT library and cannot recognize the HTs out of the library. FV methods depend on the predefined property for HTs and cannot identify new types of HTs. However, GNN4TJ is an ML-based method that learns the HT behaviors and is able to pinpoint different types of HT, even the unknown ones.

V. CONCLUSION

This paper presents a novel golden reference-free methodology to find unknown HT in RTL. We generate DFG of RTL codes and employ the GNN to construct a model on the generated graphs. GNN4TJ automatically extracts the features of graphs and learns the behavior of the hardware design. Our model is trained and tested on a DFG dataset created by expanding the Trustub benchmarks. The results indicate that GNN4TJ discovers HT with 97% recall very fast in 21.1ms.

ACKNOWLEDGMENT

This research was supported by the Office of Naval Research (ONR), award number N00014-17-1-2499. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect views of our funding agencies. Moreover, we would like to acknowledge Yunhe Shao for his support in this project.

REFERENCES

[1] Trusthub. *Available on-line: https://www.trust-hub.org*, 2016.
[2] S. Adee. The hunt for the kill switch. In *IEEE Spectrum*, 2008.
[3] A. Ardeshiricham et al. Register transfer level information flow tracking for provably secure hardware design. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2017.
[4] F. Demrozi et al. Exploiting sub-graph isomorphism and probabilistic neural networks for the detection of hardware trojans at rtl. In *IEEE International High Level Design Validation and Test Workshop*, 2017.
[5] N. Fern and S. Carlson. A complete system-level security verification methodology. (white paper from cadence and tortuga logic). 2019.
[6] M. Fyrbiak et al. Graph similarity and its applications to hardware security. *IEEE Tran. on Computers*, 2020.
[7] T. Han et al. Hardware trojans detection at register transfer level based on machine learning. In *Symposium on Circuits and Systems(ISCAS)*, 2019.
[8] K. Hasegawa et al. Hardware trojans classification for gate-level netlists using multi-layer neural networks. In *IEEE International Symposium on On-Line Testing and Robust System Design (IOLTS)*, 2017.
[9] W. Hu et al. Property specific information flow analysis for hardware security verification. In *International Conference on Computer-Aided Design (ICCAD)*, 2018.
[10] Z. Huang et al. A survey on machine learning against hardware trojan attacks: Recent advances and challenges. *IEEE Access*, 2020.
[11] S. A. Islam et al. Socio-network analysis of rtl designs for hardware trojan localization. In *International Conference on Computer and Information Technology (ICCIT)*, 2019.
[12] Jasper. Jaspergold: Security path verification app. 2014.
[13] T. N. Kipf and M. Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.
[14] B. Knyazev et al. Understanding attention and generalization in graph neural networks. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2019.
[15] J. Lee et al. Self-attention graph pooling. *arXiv preprint arXiv:1904.08082*, 2019.
[16] A. Nahiyan et al. Hardware trojan detection through information flow security verification. In *IEEE International Test Conference (ITC)*, 2017.
[17] L. Piccolboni et al. Efficient control-flow subgraph matching for detecting hardware trojans in rtl models. *ACM Tran. on Embedded Computing Systems (TECS)*, 2017.
[18] J. Rajendran et al. Detecting malicious modifications of data in third-party intellectual property cores. In *ACM/IEEE Design Automation Conference (DAC)*, 2015.
[19] J. Rajendran et al. Formal security verification of third party intellectual property cores for information leakage. In *International Conference on VLSI Design and Embedded Systems (VLSID)*, 2016.
[20] S. Saha et al. Improved test pattern generation for hardware trojan detection using genetic algorithm and boolean satisfiability. In *International Workshop on Cryptographic Hardware and Embedded Systems*, 2015.
[21] P. Subramanyan and D. Arora. Formal verification of taint-propagation security properties in a commercial soc design. In *Design, Automation & Test in Europe Conference (DATE)*, 2014.
[22] S. Takamaeda-Yamazaki. Pyverilog: A python-based hardware design processing toolkit for verilog hdl. In *International Symposium on Applied Reconfigurable Computing*, 2015.
[23] A. Waksman et al. Fanci: identification of stealthy malicious logic using boolean functional analysis. In *ACM SIGSAC Conference on Computer and Communications Security*, 2013.
[24] F. Zareen and R. Karam. Detecting rtl trojans using artificial immune systems and high level behavior classification. In *Asian Hardware Oriented Security and Trust Symposium (AsianHOST)*, 2018.
[25] J. Zhang et al. Detrust: Defeating hardware trust verification with stealthy implicitly-triggered hardware trojans. In *ACM SIGSAC Conference on Computer and Communications Security*, 2014.
[26] J. Zhang et al. Veritrust: Verification for hardware trust. *IEEE Tran. on Computer-Aided Design of Integrated Circuits and Systems*, 2015.