A Needle in the Haystack: Inspecting Circuit Layout to Identify Hardware Trojans

Xingyu Meng[1], Abhrajit Sengupta [2], and Kanad Basu[1]

[1]ECE Department, University of Texas at Dallas, Richardson, TX, USA
[2]Qualcomm, San Diego, CA, USA

***Abstract*—Distributed integrated circuit (IC) supply chain has resulted in a myriad of security vulnerabilities including that of hardware Trojan (HT). An HT can perform malicious modifications on an IC design with potentially disastrous consequences, such as leaking secret information in cryptographic applications or altering operation instructions in processors. Due to the emergence of outsourced fabrication, an untrusted foundry is considered the most potent adversary in introducing an HT. In order to address this issue, in this paper, we introduce a layout-level HT detection algorithm utilizing low-confidence classification and providing Trojan localization. We convert the IC layout to a graph and utilize Graph Neural Network (GNN)-based learning frameworks to flag any unrecognized suspicious region in the layout. The proposed framework is evaluated on *AES* and *RS232* designs from the Trusthub benchmark suite, where it has been demonstrated to detect all nine HT-inserted designs. Finally, we open-source the full code-base for the research community at large.**

***Index Terms*—Hardware Trojan Detection, IC Layout, Graph Neural Network, Connectivity Graph**

I. Introduction

Integrated circuits (IC) are keystones of modern electronics, ranging from smartphones to military-grade applications. These ICs form the root-of-trust (RoT) that play an important role in ensuring the privacy, authenticity, and integrity of the entire solution stack, including those that contain sensitive information. However, with the globalization of the IC supply chain, this assumption has come under intense scrutiny in recent years. With deep sub-micron technology, the rising cost of owning a fabrication facility created a high barrier to entering the market, especially for start-ups. For instance, TSMC's 28nm Fab 15 in Taiwan is valued at $9.3B. Similarly, the cost of establishing its new 3nm facility is projected to be $20B [1]. This financial constraint lead to the birth of the fabless model; where semiconductor companies began to outsource manufacturing to large integrated device manufacturers (IDM) having excess capacity. This shift proved to be advantageous for many as it allowed design companies to improve the bottom-line profitability, while remaining focused on core competencies. However, relinquishing such a large part of control over the IC supply chain has led to several threats, including the insertion of Hardware Trojans (HT) in a circuit. For instance, In 2017, it was estimated that U.S. companies incurred financial losses ranging from $225 billion to $600 billion due to design IP infringement [2]. Besides massive financial losses [3], these threats can also potentially undermine national security. Indeed, in 2008, Syrian radar systems were suspiciously disabled by an alleged backdoor in its microprocessors [4]. Furthermore, according to a 2013 report by the Semiconductor Industry Association (SIA), 15% of all the "spare and replacement semiconductors" bought by the Pentagon are counterfeit [5]. With rapid advancements in capabilities for adversaries, such threats become a pressing concern for commercial and government agencies alike.

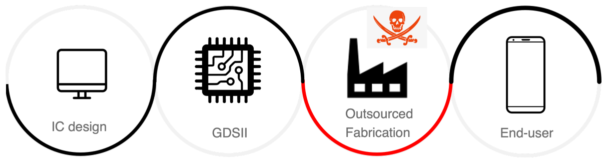


Figure 1: Overview of the IC supply chain, where outsourced fabrication is untrusted as marked in red.

A. Hardware Trojans and Its Associated Challenges

A Hardware Trojan (HT) is an *unauthorized alteration* of IC functionality often with malicious intent such as denial of service, leaking sensitive information, etc. Since outsourcing design is getting more attraction, the off-shore *untrusted* foundry has become the most potent adversary in introducing an HT. A typical IC supply chain is shown in Fig. 1, where the untrusted foundry is highlighted in red. A stealthy Trojan is usually hard to detect due to the following reasons:

- Conventional IC testing is limited in scope due to the classic controllability/observability issues.
- Parametric on-chip variations (POCV) cause non-deterministic changes in IC characteristics, which are indistinguishable from HTs.
- Formal verification for possible HT insertion in an IC leads to a state space explosion.

Consequently, existing research, including functional testing [6], side-channel analysis (SCA) [7], and formal verification (FV)-based techniques [8] suffer from several pitfalls such as low coverage, POCV, and the requirement of a golden IC. A summary of HT detection techniques and their limitations is provided in Table I. Further details are provided in Section II-B.

B. Contributions

To overcome the above shortcomings, we present a framework for HT detection that successfully identifies the structural/functional characteristics of an HT in a gate-level layout.[1]

This work is supported by NSF grant #2223046.

[1]A circuit's layout is closely associated with its functionality [13].

Table I: Summary of existing HT detection techniques and their limitations. ✗ denotes each technique's limitation, ✓ denotes it does not suffer from that limitation.

Techniques	Low coverage	State explosion	POCV	Golden IC	Error tolerant
Functional testing [6]	✗	✗	✓	✗	✓
SCA [7], [9]	✓	✓	✗	✗	✓
FV [8], [10]–[12]	✓	✗	✓	✓	✗
Our work	✓	✓	✓	✓	✓

Accordingly, we represent a layout as a *directed acyclic graph* (DAG) and extract several features from the design layout that help capture its structure/functionality. Finally, we provide an *end-to-end automated framework* for HT detection and Trojan cell localization. The contributions of this paper are summarized as follows:

- We extract several features from the layout of a circuit including *gate-types* and *gate-connectivity* to accurately identify the functionality of a circuit. To this end, we apply Graph Neural Network (GNN) node classification to flag a cell, if it contains a function that significantly differs from the original functionality of the circuit.
- A complete end-to-end automated layout-level HT detection framework is presented that scales for large designs such as AES, having 240K+ cells.
- We utilize a low-confidence node classification approach to flag suspicious cells in the layouts and separate them into connected components via the cell connection. Clustering-based identification is applied on the connected components to differentiate the HT-inserted layout from HT-free one, which aids in further localizing the Trojan cells.
- We present extensive results on a wide range of HTs from the TrustHub benchmark suite [14], that establish the efficacy of our technique. To this end, we are able to flag all ICs having HTs in the layout.
- Finally, we shed some light on several aspects of our work such as different HT payloads, scalability, error tolerance, and comparison against other GNN-based approaches.

The rest of this paper is organized as follows. Section II describes the defense capabilities, threat model, and related works in HT detection domain. Section III provides an overview of the proposed technique. Section IV demonstrates the evaluation of the proposed technique. Section V discusses the capabilities of the proposed technique and compares it with prior research. Finally, Section VI concludes our paper.

II. Motivation and Background

Before delving into further details, it is imperative that we precisely define the threat model and identify the assumptions within the context of this work.

A. Threat Model

1) Defense Capabilities: We consider that the designer has the capability to reverse engineer the manufactured IC layout and obtain the pre-silicon netlist for analysis. Furthermore, we assume that the malicious foundry inserts HTs in all copies of the IC (since it is extremely expensive to create different masks and thus, tampering only a subset of ICs). In this paper, we explore the high-level functional abstraction performed on the gate-level netlist, and subsequently, functional anomalies that are used to identify to flag HTs. However, note that this does not correspond to the often-mentioned concept of "Golden IC", since this is a soft IP that does not guarantee to represent the original RTL, and hence, side-channel analysis can't be performed here [15]. Although equivalence checking can detect a difference between the original and reverse-engineered netlist, it fails in the presence of errors while performing reverse-engineering [16]. Since the RE process is highly susceptible to errors, equivalence checking will generate a lot of false positives. To address these challenges, the proposed method only flags suspicious cells in the layout, and only if these cells perform an undefined functionality. In this case, the layout is identified as being infected by Trojans.

2) Adversarial Capabilities: As explained earlier, foundry-inserted HTs are enabled by the asymmetric business model between the design house and the foundry; the design IP is fully transparent to the foundry, whereas the design house is oblivious to the fabrication process. Usually, HTs are inserted during the fabrication process by modifying the mask.

B. Prior Work and Their Limitations

All prior works on HT detection can be classified into the following categories:

1) Functional testing: Traditional functional testing of ICs with test patterns is ineffective for HT detection due to the following reasons: a) HTs are stealthy in nature, thus, the trigger condition is rarely satisfied, b) functional testing only covers a negligible part of the total input space, and c) it is computationally infeasible to cover the whole input space through automatic test pattern generation (ATPG), and thus, techniques relying on ATPG suffer from low success rate [6].

2) Side-channel analysis (SCA): Several methods have been proposed that leverage SCA to detect HTs [7]. However, such methods suffer from several pitfalls: a) the footprint of an HT could be small, sometimes as low as 0.01% of the main circuit, and thus the SCA footprint such as power profile of an HT becomes hard to detect [7], b) at deep sub-micron technology, the difference between HT footprint and random POCV becomes indistinguishable, and c) it is relying on the existence of a golden IC.

3) Formal verification (FV): FV is used to formally prove that a given design conforms to the specified properties, else flags an issue if any such property is violated. There exist several works that have leveraged FV for the detection of HTs such as [11]. However, the scope of applying FV for HT detection remains limited due to the following reasons: a) the large available space in the IC for possible HT insertion leads to state space explosion for FV, thus, limiting its scalability, b) FV assumes the existence of a golden reference, and c) FV has zero tolerance toward any error, and thus fails against a circuit that may include unintentional errors while performing RE. In other words, it can not distinguish between an unintentional error and a true HT in a circuit layout.

4) Self-authentication techniques: These approaches utilize runtime measurements to identify the HT effect without the golden design. Operation parameters such as transient current, path delay fingerprints, and error signals are collected to capture significant differences caused by Trojan payloads in different time periods [17], [18]. However, these approaches suffer from reduced detection sensitivity without the original design layout. Nevertheless, they require expensive computations, variations of process models, and a significant amount of measurements to ensure accuracy for complex designs.

5) Machine learning (ML): Recently, several ML-based HT detection techniques have been proposed such as [19]–[21]. In [20], the authors developed a gradient-boosting model that extracts features from the RTL source code. Further, several works on less-toggled signal (LTS) identification using a support vector machine or artificial neural network have been presented in [22]. These approaches commonly require a golden design to be effective in Trojan detection. Lastly, GNN4TJ, a GNN-based approach was presented which is a golden reference-free HT detection method in the RTL [23]. However, it does not have the capability of locating Trojan.

C. Graph Neural Network (GNN)

GNNs are powerful tools that facilitate classification and clustering on attributed graphs. Consider *G(V, E)* is an un-directed attributed graph; *V* is the set of nodes, and *E* is the set of edges. Each node $v \in V$ is associated with a feature vector (embedding) that captures its properties. Afterward, GNN performs neighborhood aggregation (AGG), where the embeddings are exchanged between neighboring nodes through message passing. A new embedding is computed through a loss function by combining the node's embedding with its neighbors' aggregated embeddings. This facilitates a node to capture the structural/functional information about its neighborhood. Thus, GNNs are well-suited for identifying sub-circuits (sub-graphs) as they tend to possess specific structures and connections.

The GNN framework GraphSAINT used in our technique is inspired by the Graph Convolution Neural Network (GCN) [24]. However, instead of building a GCN on the full graph through the nodes or edges across GCN layers, GraphSAINT is developed from minibatch construction by sampling the training graph and developing a full GCN on the sub-graph. Since nodes with higher influence on each other will have a higher probability of forming a sub-graph, the framework allows the sampled nodes to have a stronger correlation with each other in the minibatch. It also applies normalization techniques in order to address the issues of non-identical node sampling probability and bias in the mini-batch estimator. Furthermore, variance reduction analysis and lightweight sampling algorithm are utilized to improve the scalability of the training process.

III. Finding the Needle: GNN-based Hardware Trojan Detection

In this section, we demonstrate that this is indeed possible. To this end, we train a GNN-based model to capture a circuit's functionality from its layout.[2] Since the functionality of a Trojan payload is fundamentally different than that of a Trojan-free circuit, it can be accurately identified by the GNN model, thereby flagging the circuit. Nevertheless, it is challenging to correctly extract the feature sets from the layout, and subsequently, train the GNN model. Fig. 2 shows the complete end-to-end framework, which can be divided into two major processes: 1) model training and 2) HT detection.

A. Model Training

The training phase can be divided into three parts, which are described as follows:

1) Layout-based feature extraction,
2) Circuit to graph transformation,
3) Dataset generation.

*1) **Netlist-based Feature Extraction**:* In order to capture the functionality of a circuit, we leverage the following features from a netlist reverse-engineered from the circuit layout;

- **Gate-type**. Each cell in the circuit is associated with a specific type of Boolean logic gate such as AND, OR, etc., that is captured as a feature in the GNN.
- **Neighborhood-size.** The layers of neighboring cells, corresponding to each individual cell are denoted as $h = 1, 2, 3, \ldots$. For example, in Fig. 3a, for cell $\mathcal{C}_i$ (marked in blue), its neighborhood-size $h = 1$ is illustrated in green. Note that each individual cell represents only a limited amount of information about a particular functionality. However, the aggregation of neighboring cells could capture the structure/function of the local neighborhood in a holistic way, and thus, help identify the unique functionality of the circuit. To this end, we store the gate-type information for all the neighboring cells in the circuit, as illustrated in Fig. 3b.

*2) **Circuit to Graph Transformation**:* In order to apply the GNN model, we first need to convert a circuit to its equivalent graph representation. Usually, this can be achieved in a straightforward manner, where a circuit is represented as a directed acyclic graph (DAG) [25]. However, an un-directed graph is more suitable for GNN, since it renders the internal message passing more efficient. Therefore, we represent a circuit as an un-directed graph $G = (V, E)$, where V denotes the set of nodes, *i.e.*, cells, while E represents the set of edges, *i.e.*, the connections between the cells.[3] Fig. 3a shows a sub-circuit and a table for each included cell. It can be transformed into a sub-graph, as shown in Fig. 3b.

*3) **Dataset Generation**:* As mentioned in Section II-C, we utilize an open-source GNN model, GraphSAINT, to learn the functionalities of different circuits [24]. The proposed GNN-based methodology operates by identifying a Trojan payload, whose circuit features differ considerably from that of the known functionalities. GraphSAINT dataset requires five separate files to train the model, viz., 1) full graph matrix, 2) training matrix, 3) role dictionary, 4) class dictionary, and 5) cell feature matrix, which are described below.

- **Full graph matrix** ($\mathcal{M}_F$) GraphSAINT represents a netlist graph with an $N \times N$ adjacency matrix, where $N = |V|$ denotes the number of nodes in the graph. If there exists a connection between cells $\mathcal{C}_x \to \mathcal{C}_y$,

[2] Note that a circuit's function exhibits a strong correlation with its structure, as illustrated in [13], [25].

[3] Note that a circuit can be represented as an un-directed graph without loss of generality.

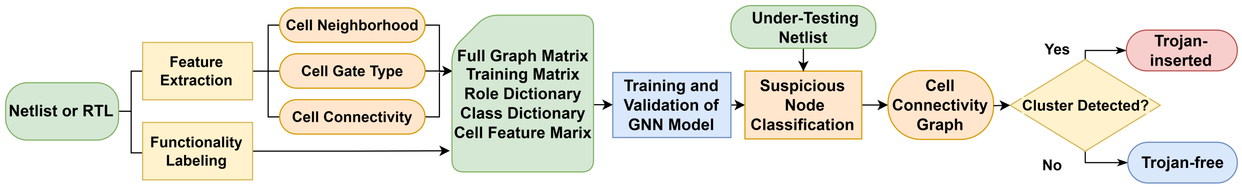


Figure 2: An overview of the proposed methodology. It utilizes the GNN model to classify the suspicious node in the graph and generate a cell connectivity graph to identify HT in the design.

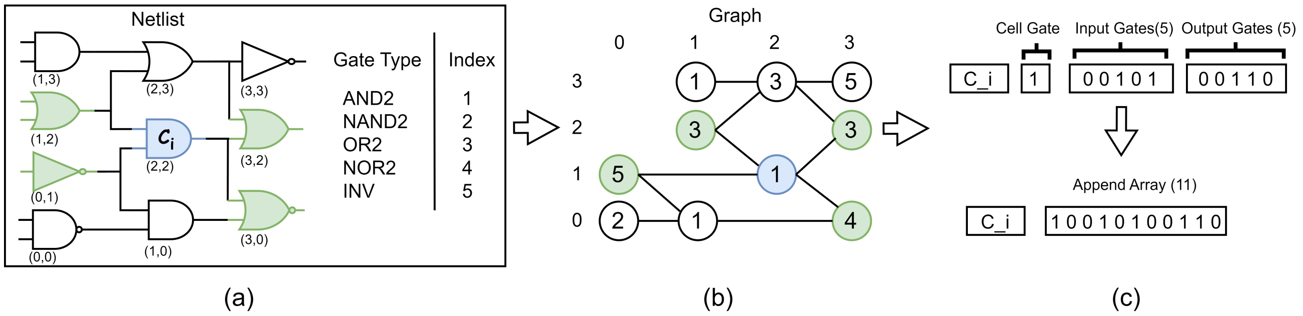


Figure 3: Figure (a) demonstrates a sub-circuit with an AND gate at the center and its neighbor cells (green), as well as a simplified table of gate types and their index. Figure (b) demonstrates the corresponding sub-graph shown in Figure (a). Figure (c) shows one row of the feature generated for $\mathcal{C}_i$.

$\mathcal{M}_F[x][y]$ and $\mathcal{M}_F[y][x]$ will have value of one, where x and y denotes the index for cells $\mathcal{C}_x$ and $\mathcal{C}_y$, respectively. In addition, we can process multiple graphs by stacking the matrices into a larger matrix. As shown in Fig. 4a, an $N \times N$ matrix and an $M \times M$ matrix can be merged into a $(N+M) \times (N+M)$ matrix. This technique is used to represent the training and the testing circuit in a single graph, where the nodes in the testing circuit are kept completely separate from that of training or validation. Nevertheless, since $\mathcal{M}_F$ is a sparse matrix, it can be stored with three one-dimensional arrays for the non-zero values, thereby having only a *linear space complexity*.

- **Training matrix** ($\mathcal{M}_T$) In contrast to the full graph matrix $\mathcal{M}_F$, a non-zero value in $\mathcal{M}_T$ corresponds to an edge between two training nodes. In Fig. 4b, the full graph matrix $\mathcal{M}_F$ is shown, where training nodes, validation nodes, and testing nodes are denoted with green, yellow, and red, respectively. The training and validation metrics are generated by multiple netlists containing the same functionalities with a split of 80% and 20%. The testing matrix is generated by the target layout which potentially contains a Trojan. In the training matrix, $\mathcal{M}_T$, all the non-zero values in the yellow/red region will be ignored, and only the training nodes marked in green will retain the connectivity information from $\mathcal{M}_F$.
- **Role dictionary** ($\mathcal{D}_R$) The role dictionary $\mathcal{D}_R$ contains three keys, viz., `tr`, `va`, and `te` corresponding to training, validation and testing nodes, respectively. Note that $|tr|+|va|+|te| = N$, where N denotes the total number of nodes in the graph. $\mathcal{D}_R$ directly establishes the relation between $\mathcal{M}_T$ and $\mathcal{M}_F$, where it dedicates the node as one for training, validation, or testing. To this end, we first select the training and testing nodes, and subsequently, choose 10% of the training nodes at random for cross-validation. For example, in Fig. 4b, the green nodes are used for training, the yellow nodes are chosen for validation, and the red nodes are used for testing.

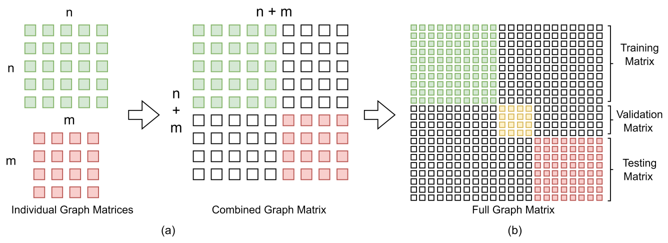


Figure 4: Figure (a) shows how two matrices combine into one. Figure (b) shows the training, validating, and testing metrics.

- **Class dictionary** ($\mathcal{D}_c$) The class dictionary contains N keys, representing the class label for each node in the training set. To this end, we classify each cell in the netlist according to its base-level functionality. Note that the base-level functionality of a cell can be easily derived from the module hierarchy. Consider Fig. 5a, where the module hierarchy for each cell is shown with the colored boxes. For example, the module hierarchy for Cell 67 is marked in the red box, which is `t2`. Since this is an instance of a multi-class node classification, we assign a numerical value to each class in the circuit, *e.g.*, the class `t2` is assigned to `Class 6`, as seen from Fig. 5b. Therefore, in the case of this example, Fig. 5d shows that cell 67 obtains a classification array, which contains the functionality classes as [6].
- **Cell feature matrix** ($\mathcal{F}$). The feature matrix is an $N \times F$ matrix, where each row i represents a vector of length F for each cell $\mathcal{C}_i$ in the training set. The features we used to train the data include four aspects of netlist: the gate-

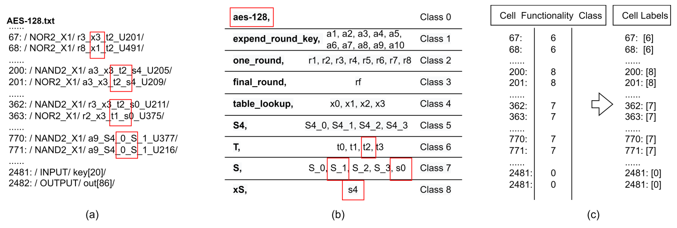

AES-128.txt
......
67: / NOR2_X1/ r3_x3_t2_U201/
68: / NOR2_X1/ r8_x1_t2_U491/
......
200: / NAND2_X1/ a3_x3_t2_s4_U205/
201: / NOR2_X1/ a3_x3_t2_s4_U209/
......
362: / NAND2_X1/ r3_x3_t2_s0_U211/
363: / NOR2_X1/ r2_x3_t1_s0_U375/
......
770: / NAND2_X1/ a9_S4_0_S_1_U377/
771: / NAND2_X1/ a9_S4_0_S_1_U216/
......
2481: / INPUT/ key[20]/
2482: / OUTPUT/ out[86]/
(a)
aes-128, Class 0
expend_round_key, a1, a2, a3, a4, a5, a6, a7, a8, a9, a10 Class 1
one_round, r1, r2, r3, r4, r5, r6, r7, r8 Class 2
final_round, rf Class 3
table_lookup, x0, x1, x2, x3 Class 4
S4, S4_0, S4_1, S4_2, S4_3 Class 5
T, t0, t1, t2, t3 Class 6
S, S_0, S_1, S_2, S_3, s0 Class 7
xS, s4 Class 8
(b)
Cell Functionality Class
67: 6
68: 6
......
200: 8
201: 8
......
362: 7
363: 7
......
770: 7
771: 7
......
2481: 0
2481: 0
Cell Labels
67: [6]
68: [6]
......
200: [8]
201: [8]
......
362: [7]
363: [7]
......
770: [7]
771: [7]
......
2481: [0]
2481: [0]
(c)

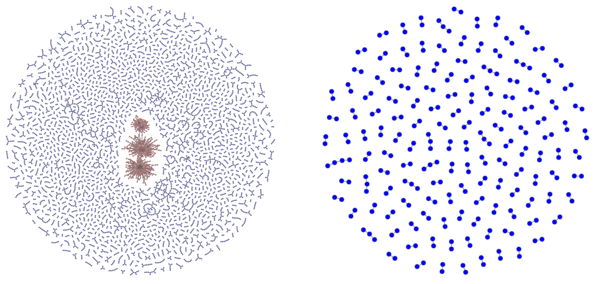
Figure 5: Figure (a) shows the required gate-type and class information parameters for each cell. Figure (b) shows an example of AES which demonstrates the class for each cell. Figure (b) shows the final class dictionary array for each cell.

type of target cell $\mathcal{C}_i$, number of each gate-type from input set $\mathcal{C}_{i_{in}}$, number of each gate-type from output set $\mathcal{C}_{i_{out}}$, and distances to all functional regions. This is illustrated in Fig. 3c. First, we create a list $[\mathcal{C}_i, \mathcal{G}_i]$, that extracts each cell and its gate-type. Next, the input set $\mathcal{C}_{i_{in}}$ is created. To store the gate-type of the input set $\mathcal{C}_{i_{in}}$, an array of length X is created, where X denotes the total number gate-types present in the library. A simple example of a table of gate-type is shown in Fig. 3a, where each gate-type has its own index. Now, the array is populated according to the number of gate-types that are present in $\mathcal{C}_{i_{in}}$. A similar approach is followed for the output set $\mathcal{C}_{i_{out}}$. Finally, we combine all three together to form a single array that stores the gate-types in the neighborhood of cell $\mathcal{C}_i$. Fig. 3c shows the final feature array for the cell $\mathcal{C}_i$ which consists of one layer of the neighboring gates. Moreover, the neighborhood size can be increased by extending the layers of neighbors, as shown in Section III-A1.

B. *Determining the Existence of an HT Node*

After the dataset generation, we proceed to train the GNN model. Note that the model is *trained only with the untampered netlist.* Hence, any cell with an unknown function such as a Trojan payload would result in a low-confidence classification. Typically, a trained machine learning model for multi-class classification will predict an unseen data sample for each class with a value scaling from 0 to 1 (0 refers to no confidence and 1 refers to full confidence). Accordingly, we develop a strategy to detect the Trojan payload by identifying such cells with prediction value lower than a certain Threshold $\mathcal{P}_T$ and generate a *detection profile*, which contains all cells with low-confidence classification. To this end, we develop the following three parameters that help achieve a higher coverage of Trojan cells for detecting the HT payload.

- **Threshold of Trojan node detection** ($\mathcal{P}_T$) The model will furnish each cell in the testing netlist a prediction score for each class from 0 to 1. The class index with the highest score will be the predicted class for the testing cell. Since the Trojan payload is not included in the training netlist, *we posit that the trained model will face challenges in predicting the class for these cells, which will end up with a low prediction score for all classes.* We utilize this feature to identify any cell that has a prediction score lower than a pre-defined $\mathcal{P}_T$.



(a) Trojan-inserted AES (b) HT-free AES

Figure 6: Graph connectivity of the detection profile. a) Shows the detection profile of AES−T900, where true-positives are marked in red while false-positives are marked blue. b) Shows the detection profile of an HT-free AES. It is evident that the true-positive nodes in the AES−T900 exhibit a strong clustering, whereas no such clustering can be observed in an HT-free circuit.

Algorithm 1 Trojan Detection via Detection Profile

Input: Detection Profile H, Layout Graph $G(V,E)$
Output: *True/False*

```
1:  $G'$ ← **initialize_graph**
2:  **for** each $h \in H$
3:      $G'$ ← **add_node**$(h)$
4:  **for** each edge $(u,v) \in E$
5:      **if** $u, v \in H$
6:          $G'$ ← **add_edge**$(u,v)$
7:  $\mathcal{L}_c$ ← **connected_component**$(G')$
8:  $\{y[0], y[1]\}$ ← **k-means**$(\mathcal{L}_c, n\_cluster = 2)$
9:  **if** $0 <$ **sizeof**$(y[0]$ or $y[1]) < Th$
10:     **return** True
11: **else**
12:     **return** False
```

- **Number of rounds** ($\mathcal{R}$) With the same netlist, the GNN model will produce different weight values on each feature in various training rounds, thereby, leading to different prediction outcomes. We could apply this aspect to reveal more Trojan cells that might escape the detection in one round.
- **Reappearance ratio** ($\mathcal{A}$) After each round of testing, different cells will be flagged. The reappearance ratio $\mathcal{A}$, where $0 \leq \mathcal{A} \leq 1$, is introduced to determine the number of times each cell is flagged during testing, *which could help in filtering the true-positives from the false-positives.* $\mathcal{A}$ is defined as $n/\mathcal{R}$, where $1 \leq n \leq \mathcal{R}$, is a predetermined threshold. Any cell having a reappearance ratio larger than $\mathcal{A}$ will be flagged as a Trojan.

After the *detection profile* is generated, we visualize the cell distribution by plotting its corresponding graph. Fig. 6a shows the detection profile graph for AES−T900 from TrustHub, whereas Fig. 6b shows the detection profile graph of an HT-free AES circuit [14]. Note that the false-positives are marked in blue, whereas the true-positives are marked in red. It is

Table II: TrustHub benchmark suites used in our experiments [14].

Benchmark	Trigger?	HT Payload	Detected?	Runtime
AES-T100	No	Leakage	Yes	45m57s
AES-T200	No	Leakage	Yes	46m5s
AES-T900	Yes	Leakage	Yes	46m17s
AES-T1200	Yes	Leakage	Yes	46m30s
AES-T1800	Yes	DoS	Yes	46m48s
RS232-T100	Yes	DoS	Yes	3m40s
RS232-T200	Yes	Function Alter	Yes	4m10s
RS232-T400	Yes	Function Alter	Yes	3m55s
RS232-T800	Yes	Function Alter	Yes	7m10s

evident from Fig. 6a that in an *HT-inserted layout, the true-positives exhibit a strong clustering as compared to the false-positive ones.*

Based on this observation, we develop a heuristic that captures the clustering pattern in the detection profile. To this end, we generate the sub-graph for the detection profile and subsequently, list all the *connected components* in it. *If there exists a few connected components that are significantly larger than the rest, we flag the circuit as HT-inserted, else not.* Algorithm 1 delineates the proposed heuristic. First, we initialize the sub-graph with all the nodes from the detection profile. Next, we add the edge $(u, v) \in E$ to G' if u, v are both present in the detection profile H. Afterwards, we list all the connected components in G' denoted by $\mathcal{L}_c$. Next, we classify the components in terms of the number of nodes present. To this end, we apply *K-Means* clustering to split them into two clusters $y[0]$, $y[1]$, from which either of the following two conclusions can be made [26]:

1) If only a few components have a large number of nodes, they are labeled in one cluster, whereas the majority of the components having only a small number of nodes fall into the other. The existence of a few such components having a large number of nodes is indicative of HT, and accordingly, we return True. In our experiments, we empirically determine that the threshold Th for the number of large components is three. Note that the *identification of such components implicitly localizes the Trojan cells in the layout.*
2) If both clusters $y[0]$ and $y[1]$ contain a similar number of components which is larger than the threshold Th, it indicates that there is no outlier having a large # nodes; thus the layout is marked Trojan-free by returning False.

IV. Experimental Results

A. Experiment Setup

All our experiments are carried out on a machine having 40 CPUs of 64-bit Intel(R) Xeon(R) E5-2698 v4 @ 2.20GHz. All the codes have been implemented in Python. All the circuits are synthesized using Synopsys Design Compiler (DC) with the 45nm NanGate Open Cell Library [27], and the corresponding layouts are generated using Cadence Innovus. For evaluation purposes, we use five `AES-128` and four `RS232` benchmark suites from Trusthub [14]. Note that since we are comparing our performances with the work in [28], we chose these benchmarks with different Trojans that are shown in their work. A brief description of the circuits is presented in Table II.

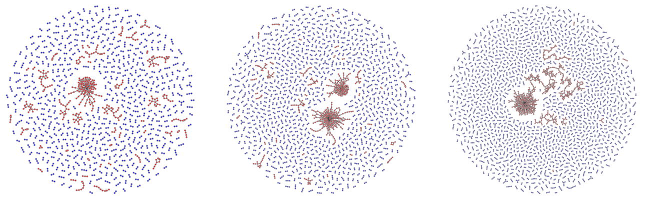

(a) $\mathcal{P}_\mathcal{T} = 0.7$ (b) $\mathcal{P}_\mathcal{T} = 0.8$ (c) $\mathcal{P}_\mathcal{T} = 0.9$

Figure 7: Effect of detection threshold $\mathcal{P}_\mathcal{T}$ on AES-T900.

B. Identifying HTs in design layout

Through the evaluation shown later in Section IV-C, we empirically select the following parameters: threshold of Trojan detection $\mathcal{P}_\mathcal{T} = 0.8$, # of rounds $\mathcal{R} = 10$, reappearance ratio $\mathcal{A} = 0.1$, and neighborhood-size $h = 2$ for our experiments. Table II summarizes the HT detection results. It is seen that our approach is able to correctly detect HT in all nine HT-inserted benchmarks. On average, for each HT-inserted `AES` layout, our technique is able to reveal 70% of the Trojan cells in a layout, with the maximum coverage up to 90%. In addition, *false-positive rates are almost zero once the outlier component is identified.*

Note that the *main objective of this work is to identify whether the HTs exsit in an unseen netlist*, where it succeeds in all cases. Further, these results show that not only is our technique capable of Trojan-inserted layout detection, but also in *localizing the majority (up to 90%) of the Trojan cells that are inserted in the layout.*

C. Parameter Evaluation for $\mathcal{P}_T$, $\mathcal{R}$, $\mathcal{A}$, and h

In this section, we discuss how to tune different parameters of our proposed GNN model. As mentioned earlier, the parameters chosen for our model are as follows: $\mathcal{P}_\mathcal{T} = 0.8$, $\mathcal{R} = 10$, $\mathcal{A} = 0.1$, and $h = 2$. To evaluate the effects, we change only one parameter at a time, while keeping the rest constant. All the experiments in this section are demonstrated based on the cell connectivity graphs of `AES-T900` benchmark. Similar results were obtained for other benchmarks as well.

1) Detection Threshold $\mathcal{P}_\mathcal{T}$: Fig. 7 establishes a direct correlation between $\mathcal{P}_\mathcal{T}$ and the false-positive rate on `AES-T900` benchmark; the larger the threshold, the higher is the false-positive rate. This is due to the fact any node having prediction score less than $\mathcal{P}_\mathcal{T}$ is flagged as a potential HT. Nonetheless, even with a high false-positive rate, the HT-inserted circuit exhibits strongly clustering components, thereby aiding in the detection of HT. However, with lower threshold, the connectivity starts to fade out as seen for $\mathcal{P}_\mathcal{T} = 0.7$, when compared to $\mathcal{P}_\mathcal{T} = 0.8$. Thus, it becomes a trade-off between HT detection capability vs false-positive rate, and we consider $\mathcal{P}_\mathcal{T} = 0.8$ for our experiments.

2) # Round $\mathcal{R}$: Fig. 8 shows the effects of $\mathcal{R}$ on `AES-T900` benchmark. It can be seen that with $\mathcal{R} = 1$, the GNN model fails to detect the existence of HT in the circuit. Nevertheless, with larger $\mathcal{R}$, the true-positive improves considerably, and after $\mathcal{R} = 10$, the improvement plateaus. Hence, $\mathcal{R} = 10$ is selected for our experiments, in order to maximize the detection coverage and reduce time consumption.

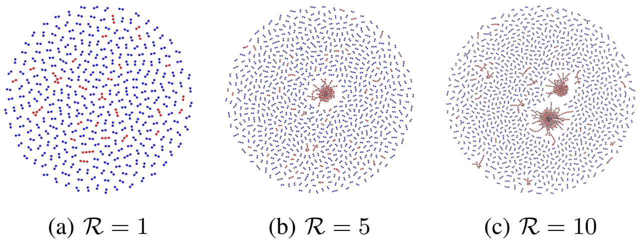

(a) $\mathcal{R} = 1$ (b) $\mathcal{R} = 5$ (c) $\mathcal{R} = 10$

Figure 8: Effect of # Rounds $\mathcal{R}$ on AES-T900.

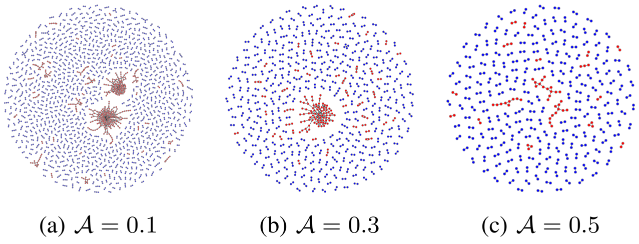

(a) $\mathcal{A} = 0.1$ (b) $\mathcal{A} = 0.3$ (c) $\mathcal{A} = 0.5$

Figure 9: Effect of reappearance ratio $\mathcal{A}$ on AES-T900.

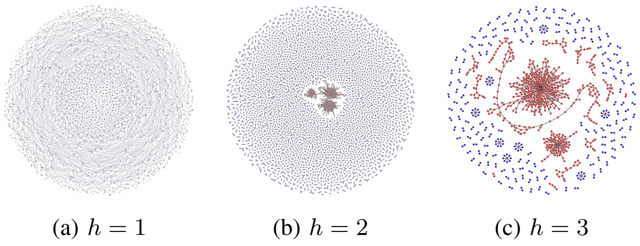

(a) $h = 1$ (b) $h = 2$ (c) $h = 3$

Figure 10: Effect of neighborhood-size h on AES-T900.

3) *Reappearance Ratio $\mathcal{A}$:* Fig. 9 shows the effect of $\mathcal{A}$ on false-positive rate; the smaller the ratio, the higher the false-positive rate. Nevertheless, even with a high false-positive rate, the HT-inserted circuit strongly exhibits clustering components, thereby aiding in the detection of HT. However, with larger ratio, the connectivity starts to fade out as seen for $\mathcal{A} = 0.5$ when compared to $\mathcal{A} = 0.1$, thereby limiting the HT detection ability of the model. For our experiments, $\mathcal{A} = 0.1$ provides the best results, *i.e.*, some Trojan cells might be detected only once in 10 rounds. On the other hand, some cells in *Sbox* and *ShiftRow* have a much higher appearance ratio, since these cells lack the amount of input and output features compared to other cells. Therefore, they are more likely to be flagged due to their limited features in any class. However, the Trojan cells have different features from the majority of cells in *Sbox* and *ShiftRow*; thus, they have unstable prediction confidence through different rounds of testing. Hence, when $\mathcal{A}$ = 0.1, the framework achieves the highest Trojan cell coverage since it more likely to reveal difficult-to-classify Trojan cells.

4) *Neighbourhood-size h:* The effect of neighborhood-size is illustrated in Fig. 10. We can conclude that the dataset with $h = 2$ provides the best performance. Two layers of neighbouring cells could properly capture the functionality in a local region, thereby drastically reducing the false-positive rate from $h = 1$. Although increasing h further reduces the false-positive, which can be seen when $h = 3$, the detection coverage gets significantly lower.

Table III: Our work compared to other GNN-based techniques.

	Our Work	GNN4tj [23]	GNN-re [25]	HW2VEC [28]
Golden IP Free?	✓	✓	✓	✓
Unknown Trojan?	✓	✓	✓	**Unknown**
Trojan Localization?	✓	**Unknwon**	✓	×
Fault Tolerance?	✓	×	×	×

V. Discussion

A. Comparison with Prior Work

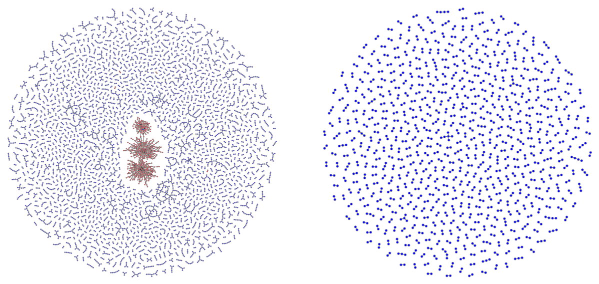
In this section, we compare our method with a related technique [28]. In [28], a data flow graph (DFG) is generated from the design RTL and Spatial Graph Convolution Neural Network (SGCN) is applied to study the convolution operation based on a node's spatial relations. It converts each hardware design RTL into the corresponding DFG and generates the graph embedding of the design. The authors trained and tested GNN models with graph embedding generated from Trojan-inserted and Trojan-free DFG, and demonstrated that their approach is capable of classifying Trojan-inserted and Trojan-free RTL through DFG. However, this approach fails to classify an unseen benchmark unless the benchmark is labeled prior to training, which limits the performance of Trojan detection. In a separate work [23], although the authors claim that the approach is able to provide Trojan localization and labeling, it does not consider the performance of faulty designs. Furthermore, we could not evaluate it since the code is not publicly available.

For our proposed HT detection technique, we use cluster identification to distinguish between Trojan-inserted and Trojan-free ICs at the netlist level. As shown in Section III-B, through our approach, if a cluster is detected among all the connections formed by the nodes flagged by the GNN model, the layout under test will be considered as HT-inserted. Therefore, it is applicable in real scenarios when the designs under test are not labeled and the Trojan payloads are unpredictable.

B. Efficacy of the Proposed Technique

1) *Run-time & Scalability:* As shown in Table II, the total execution time takes only up to a few minutes, even for large layouts such as AES crypto cores having $\sim$ 250K+ gates. Since the GNN represents the graphs with sparse matrices, the complexity *scales only linearly* in $(|V| + |E|)$, where $|V|$ denotes the number of nodes and $|E|$ denotes the number of edges in the graph. This is evident from the fact that even when the size of the dataset changes from $\sim$ 5K+ to $\sim$ 250K+ cells, the run-time does not suffer from any bottleneck. Furthermore, each round of training can be completely parallelized independent of each other, making our framework highly scalable.

2) *Effectiveness Against Different Types of HT Payloads:* We experimented with three different types of HT payloads, viz., leakage, denial of service, and change-functionality. It is evident from Table I that we are able to detect all three different types of payload successfully. However, certain types of payload may prove to be harder to detect than others, *e.g.*, denial of service. This is attributed to the small footprint of the HT. Moreover, we can easily identify multiple obvious connectivity clusters, that correspond to HT triggers, since they differ from any fundamental functional module.



(a) AES-T900 (b) Faulty AES Layout

Figure 11: Distinguishing between an HT-inserted and a Faulty layout. Note that in the case of the fault layout, no clusters are present, whereas for AES-T900 existence of clusters flags the circuit as suspicious.

C. Distinguishing Between HT-inserted and Faulty Layouts

In our threat model, we assume that faults/errors might be introduced during the design phase, which leads to a faulty design layout. **In such cases, it gets challenging to distinguish between a faulty layout and an intentional alteration in the circuit.** On the contrary, the proposed framework can easily differentiate between them. Since an error would be uniformly random, it would be spread across the layout as opposed to an HT, which exhibits a strong clustering behavior. To mimic a faulty layout, we randomly alter 0.1% of their connections during the testing. The results of applying our framework to such faulty layouts are shown in Fig. 11b, where the absence of any cluster correctly classifies it as HT-free. However, for AES-T900, and other benchmarks in our experiments, clear clusters can be observed toward the center, thereby flagging the circuit as suspicious. In addition, we created five faulty layouts with a few cell connections and gate types swapped in HT-free layouts, and none of them were falsely identified as HT-inserted.

Note that our work does not suffer from the requirement of a "Golden IC" which is necessary for SCA/functional testing-based detection techniques [6], [7], [9]. *Our framework only requires the design house to possess the un-tampered netlist, which is a "soft-IP". It is reasonable since the design house generates the circuit netlist.*

VI. Conclusion

In this paper, we have presented our proposed framework for Hardware Trojan detection that operates on layout-inserted HT, utilizing graph neural networks (GNN) for identification and labeling of different functionality regions. By transforming an IC layout into a graph, the GNN model can capture the global network structural information of the layout and local structural details of each cell along with its neighboring cells. Moreover, it captures the gate-level features of each cell to identify each unique functional region. Our proposed framework identifies whether the layout is HT-inserted by utilizing the trained model to reveal suspicious cells by flagging cells with low-confidence classification, and utilizes clustering-based identification to provide Trojan localization in the layout. It is demonstrated to be capable of identifying Trojan-inserted layouts corresponding to various types of Trojan payloads and different sizes of layouts using designs from the Trusthub benchmarks. The proposed method operates without the need for a Golden IC. In the future, we aim to introduce methods to improve the performance of HT coverage; thereby, reducing the amount of effort to examine the layout and uncover all the Trojan cells.

References

[1] Tom'sHARDWARE, https://www.tomshardware.com/news/tsmc-fab-3nm-5nm-process-intel-samsung, 2019.

[2] C. on the Theft of American Intellectual Property, "Update to the IP Commission Report: The Theft of American Intellectual Property," 2017.

[3] semi, "Innovation is at risk: Losses of up to $4 billion annually due to ip infringement," 2008.

[4] "The Hunt For the Kill Switch," https://spectrum.ieee.org/the-hunt-for-the-kill-switch, IEEE Spectrum, 2008.

[5] "Detecting and Removing Counterfeit Semiconductors in the U.S. Supply Chain," https://www.semiconductors.org/wp-content/uploads/2018/06/ACTF-Whitepaper-Counterfeit-One-Pager-Final.pdf, Semiconductor Industry Association, 2013.

[6] S. Saha *et al.*, "Improved Test Pattern Generation for Hardware Trojan Detection Using Genetic Algorithm and Boolean Satisfiability," in *CHES*, vol. 9293, 2015, pp. 577–596.

[7] Y. Jin *et al.*, "Hardware Trojan detection using path delay fingerprint," in *IEEE HOST*, 2008, pp. 51–57.

[8] A. Ardeshiricham *et al.*, "Register transfer level information flow tracking for provably secure hardware design," in *IEEE DATE*, 2017, pp. 1691–1696.

[9] D. Agrawal *et al.*, "Trojan Detection using IC Fingerprinting," in *IEEE S&P*, 2007, pp. 296–310.

[10] P. Subramanyan *et al.*, "Formal verification of taint-propagation security properties in a commercial SoC design," in *IEEE DATE*, 2014, pp. 1–2.

[11] A. Nahiyan *et al.*, "Hardware trojan detection through information flow security verification," in *IEEE ITC*, 2017, pp. 1–10.

[12] W. Hu *et al.*, "Detecting Hardware Trojans with Gate-Level Information-Flow Tracking," *Computer*, vol. 49, pp. 44–52, 2016.

[13] J. Baehr *et al.*, "Machine learning and structural characteristics for reverse engineering," *Integration*, vol. 72, pp. 1–12, 2020.

[14] "Trust-hub.org," https://trust-hub.org/benchmarks/chip-level-trojan, (Accessed on 01/09/2022).

[15] A. Vakil *et al.*, "Lasca: Learning assisted side channel delay analysis for hardware trojan detection," in *21st IEEE ISQED*, 2020, pp. 40–45.

[16] E. I. Goldberg, M. R. Prasad, and R. K. Brayton, "Using sat for combinational equivalence checking," in *Proceedings Design, Automation and Test in Europe. Conference and Exhibition 2001*. IEEE, 2001, pp. 114–121.

[17] H. S. Choo *et al.*, "Register-transfer-level features for machine-learning-based hardware trojan detection," *IEICE TRANSACTIONS on Fundamentals of Electronics, Communications and Computer Sciences*, vol. 103, no. 2, pp. 502–509, 2020.

[18] K. G. Liakos *et al.*, "Conventional and machine learning approaches as countermeasures against hardware trojan attacks," *Microprocessors and Microsystems*, vol. 79, p. 103295, 2020.

[19] Z. Huang *et al.*, "A Survey on Machine Learning Against Hardware Trojan Attacks: Recent Advances and Challenges," *IEEE Access*, vol. 8, pp. 10 796–10 826, 2020.

[20] T. Han *et al.*, "Hardware Trojans Detection at Register Transfer Level Based on Machine Learning," in *IEEE ISCAS*, 2019.

[21] K. Hasegawa *et al.*, "Hardware Trojans classification for gate-level netlists using multi-layer neural networks," in *IEEE IOLTS*, 2017, pp. 227–232.

[22] ——, "Trojan-feature extraction at gate-level netlists and its application to hardware-Trojan detection using random forest classifier," in *IEEE ISCAS*, 2017, pp. 1–4.

[23] R. Yasaei *et al.*, "GNN4TJ: Graph Neural Networks for Hardware Trojan Detection at Register Transfer Level," in *DATE*, 2021, pp. 1504–1509.

[24] H. Zeng *et al.*, "Graphsaint: Graph sampling based inductive learning method," *arXiv preprint arXiv:1907.04931*, 2019.

[25] L. Alrahis *et al.*, "GNN-RE: Graph Neural Networks for Reverse Engineering of Gate-Level Netlists," *IEEE TCAD*, pp. 1–1, 2021.

[26] K. Krishna *et al.*, "Genetic k-means algorithm," *IEEE Cybernetics*, vol. 29, no. 3, pp. 433–439, 1999.

[27] "NanGate FreePDK45 Open Cell Library," Nangate Inc, 2011. [Online]. Available: http://www.nangate.com/?page_id=2325

[28] S.-Y. Yu *et al.*, "Hw2vec: A graph learning tool for automating hardware security," in *IEEE HOST*, 2021, pp. 13–23.