Node-wise Hardware Trojan Detection Based on Graph Learning

Kento Hasegawa†, *Member, IEEE*, Kazuki Yamashita†, Seira Hidano, Kazuhide Fukushima, Kazuo Hashimoto, *Member, IEEE*, and Nozomu Togawa, *Member, IEEE*

Abstract—In the fourth industrial revolution, securing the protection of supply chains has become an ever-growing concern. One such cyber threat is a hardware Trojan (HT), a malicious modification to an IC. HTs are often identified during the hardware manufacturing process but should be removed earlier in the design process. Machine learning-based HT detection in gate-level netlists is an efficient approach to identifying HTs at the early stage. However, feature-based modeling has limitations in terms of discovering an appropriate set of HT features. We thus propose NHTD-GL in this paper, a novel node-wise HT detection method based on graph learning (GL). Given the formal analysis of the HT features obtained from domain knowledge, NHTD-GL bridges the gap between graph representation learning and feature-based HT detection. The experimental results demonstrate that NHTD-GL achieves 0.998 detection accuracy and 0.921 F1-score and outperforms state-of-the-art node-wise HT detection methods. NHTD-GL extracts HT features without heuristic feature engineering.

Index Terms—hardware Trojan, detection, gate-level netlist, graph learning, node-wise

1 Introduction

The demand for high-performance, low-cost, and power-saving ICs has been increasing, which makes supply chain protection a serious concern in the reality of the fourth industrial revolution. To meet demand, the IC design process must be correspondingly secure. Primary vendors often use third-party intellectual property (3PIP) and outsource parts of their products to third-party hardware design houses. Utilizing 3PIP and outsourcing to third-party vendors has led to the globalization and complexity of the supply chain, which is associated with the risk of unintended third-party participation.

A *hardware Trojan (HT)* is emphasized as a threat in the supply chain [1]. An HT consists of two core components, a trigger and payload, and it is often implemented as minute hardware with its trigger deactivated to evade inspections. With the trigger deactivated and thus leaving its payload disabled, it acts as an HT-free IC. When the HT's trigger is eventually activated, it may leak confidential information, tamper with functionality, and suspend devices. HT detection at the design phase has been widely researched [2]. In particular, gate-level netlists (hereinafter referred to as *netlists*) are the focus. A structural feature-based HT detection method was proposed in [3], and its merit is that it requires no simulation. It also realizes comprehensive and fine analysis of the target IC design. The methods described in [4], [5] realized node-wise HT detection in netlists using machine learning (ML) and demonstrated high detection performance. However, feature-based ML methods have limitations in terms of discovering an appropriate set of features. Previous studies have adopted heuristic approaches to find structural features for HT detection. The selected features are valid for known HTs, but skilled attackers can evade them. It is a tremendous task for structural feature-based HT detection to continuously extract effective HT features from the IC design when a new HT is found. Thus, simply employing a structural feature-based approach is infeasible for real-world circuits.

To overcome these limitations of structural feature-based HT detection, a graph learning (GL) method is introduced. A circuit can be represented as a graph (e.g., Boolean networks [6]). Similarly, a netlist is represented as a graph structure whose node corresponds to a gate and whose edge corresponds to a wire. Graph structure is becoming an active research area in recent ML. It is expected that GL can extract generalized features from netlists, which is an impossibility via manual feature engineering. Considering that HTs are becoming more technical and sophisticated, GL is a promising approach. GL-based HT detection effectively distinguishes between normal circuits and HTs [7], [8], [9]. However, the impracticalities of the existing methods consist of problem settings such as circuit-wise classification and trigger-focused detection and the fact that the features GL can find are unknown.

In this paper, we propose NHTD-GL, a novel node-wise HT detection method based on GL. The contributions of this paper can be summarized as follows:

- The practical settings for HT detection are clarified by providing realistic scenarios in the hardware supply chain based on the preliminary experiences of HT detection in netlists (Section 4).
- This paper bridges the gap between the known structural features of HTs and the representation capability of GNNs by clarifying what features a GNN

K. Hasegawa, S. Hidano, and K. Fukushima are with KDDI Research, Inc.
K. Yamashita, K. Hashimoto, and N. Togawa are with Waseda University.
† Equal contribution.
Manuscript received Month DD, YYYY; revised Month DD, YYYY.

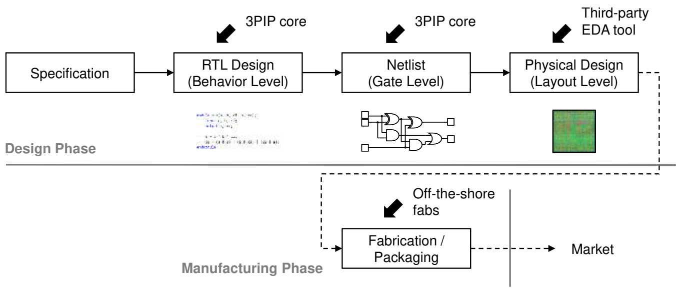


Fig. 1: Typical IC design process. Malicious 3PIP cores as well as third-party EDA tools might be involved in the process.

model captures for HT detection (Section 5).

- NHTD-GL, the hardware Trojan detection method for netlists using GL is proposed (Section 6), the theoretical background of which is described in Section 5.
- NHTD-GL is evaluated through experiments. The experimental results demonstrate that NHTD-GL outperformed state-of-the-art HT detection methods. Additionally, by comparing a GNN-based HT detection method with simple node features, it is shown that the GNN model effectively extracts the features characterizing HTs from a given training dataset (Section 7).

2 Related Works

HT detection. HT detection methods were reviewed in [1]. The typical IC design process is illustrated in Fig. 1. During the design phase, specifications are sequentially broken down into the behavior level, gate level, and layout level. 3PIP core and third-party EDA tools have the opportunity to participate in the design phase. Henceforth, malicious attackers may take advantage. The IC design is written in hardware description language (HDL) and stored in an electrical design interchange format (EDIF). Skillful attackers who know the language and format can hide HTs to contaminate the IC design or modify the design information. On the other hand, an attack during the manufacturing phase is difficult since the manufacturing system is working in real time and being controlled by a vendor-specific management process. Therefore, attacking during the design phase is a more realistic scenario than attacking during the manufacturing phase. This paper focuses on HTs inserted in the design phase and particularly in netlists rather than the more abstract level design, such as the register-transfer level (RTL), as the IC design described in RTL is ultimately translated into a netlist.

HT detection by logic testing. HTs are likely to be activated to evade logic testing. Focusing on rare activation conditions, early studies have proposed analytical detection methods based on truth tables or simulations [10], [11]. The weakness of hardware verification by logic testing is that it is time-consuming to apply for large-scale circuits and that there are techniques for making HTs less detectable to logic testing, as proposed by DeTrust [12].

Feature-based HT detection. Another approach is the feature-based method proposed in [3], which requires no simulation and realizes comprehensive analysis with less time compared to HT detection through logic testing. This method successfully detects HTs with a high accuracy rate using the structural features that are manually extracted from the Trust-HUB benchmark netlists [13], [14], which suggests that structural features that would specifically distinguish HTs from normal circuits exist. Following this suggestion, several ML-based approaches have been proposed [4], [5]. According to [5], one of the first ML-based HT detection methods was proposed in [15], and it achieved high recall but insufficient accuracy by employing support vector machines and artificial neural networks to learn HT features. Since then, new features and models have been actively investigated [16], [17], [18]. In the past five years, HT detection methods using ML have achieved 90% or more accuracy. The weakness of the feature-based method is that it requires continuous feature updating. It has been reported that a powerful feature-based method, the COTD detection method [26], would not be adequate in certain applications [19], which suggests that HT-specific features should be searched when a new HT is discovered. It is necessary to automate the feature discovery process for real-world problems.

GL-based HT detection. Graph learning is now becoming an active research area in ML [20]. Since a netlist can be represented as a graph structure by translating an element of a circuit into a node and a wire into an edge, GL-based HT detection is a promising approach for overcoming the impasse of endless feature engineering. Ref. [9] effectively detects HTs from netlists using GL, and GL-based HT detection no longer requires feature engineering. Features are automatically extracted during the learning process. As reported in [7], simple GL-based HT detection methods have a weakness in that they do not point out where HTs exist but only identify whether the design is compromised or not. For practical use, detected HTs should be presented with evidence so that the user can trust the detection result. The only solution to this problem would involve explaining the GL model to those who are not well-versed in HTs. In [9], only the trigger part is detected, leaving its critical payload part unnoticed. That is, a trigger-based approach cannot detect some HTs effectively, including always-on type HTs. Additionally, their proposed inverse node fanins (INF) contain several normal gates, which may hurt the generalization performance. Therefore, different embedding approaches, such as optimizing node features based on the representation capability of the GL model, should be considered.

3 Preliminaries

General model. The general NHTD-GL model is the graph neural network (GNN), which was introduced in [21] and is now employed in a variety of fields [20].

The MPNN model [22] proposed a unified framework for graph convolution operations, and many models can be considered with this framework. Let $G = (V, E)$ be a graph with a set V of nodes and a set E of edges. A feature vector $\boldsymbol{x}_v$ is assigned for $v \in V$, which is referred to as an initial feature vector that represents the property of a node v. Additionally, a label $y_v \in \mathbb{R}$ represents the class of the node v. In general, the GNN employs the neighborhood aggregation (a.k.a. *message passing*) mechanism in which node feature vectors are exchanged between nodes; these vectors are updated (or combined) using an updating function. This mechanism is expressed as follows [23]:

$$\boldsymbol{m}_v^{(l)} = \mathsf{AGGREGATE}^{(l)}\left(\left\{\boldsymbol{h}_u^{(l)} : \forall u \in \mathcal{N}(v)\right\}\right) \quad (1)$$

$$\boldsymbol{h}_v^{(l+1)} = \mathsf{COMBINE}^{(l)}\left(\boldsymbol{h}_v^{(l)}, \boldsymbol{m}_v^{(l)}\right), \quad (2)$$

where $\mathcal{N}(v)$ represents a set of nodes adjacent to a node v, and $\mathsf{AGGREGATE}^{(l)}(\cdot)$ (resp. $\mathsf{COMBINE}^{(l)}(\cdot)$) represents the message function (resp. the update function) at the l-th layer. Such a layer is referred to as a *GNN layer* in this paper. $\boldsymbol{h}_v^{(l)}$ is a hidden feature vector of v initialized by the initial feature vector as $\boldsymbol{h}_v^{(0)} = \boldsymbol{x}_v$, and $\boldsymbol{m}_v^{(l)}$ denotes the message exchanged between nodes. In (1), the message function $\mathsf{AGGREGATE}^{(l)}(\cdot)$ aggregates the feature vectors of the nodes adjacent to v and generates the message vector $\boldsymbol{m}_v^{(l)}$. Then, in (2), the update function $\mathsf{COMBINE}^{(l)}(\cdot)$, which is a learnable function, combines the node feature vector $\boldsymbol{h}_v^{(l)}$ and the message vector $\boldsymbol{m}_v^{(l)}$. Finally, the node feature vector $\boldsymbol{h}_v^{(l)}$ for each node at the l-th GNN layer is updated. The functions $\mathsf{AGGREGATE}^{(l)}(\cdot)$ and $\mathsf{COMBINE}^{(l)}(\cdot)$ are different from model to model, which results in its ability to represent a node.

Let $\boldsymbol{z}_v = \boldsymbol{h}_v^{(L)}$ be the final output for the node feature vector of v, where L is the number of GNN layers. In task-specific processing, the ML model for graphs can be divided into a graph encoder part and a prediction part. Let f_{enc} and f_{pred} be a graph encoder model and prediction model, respectively. The graph encoder model can be expressed as $f_{\mathsf{enc}}(\boldsymbol{x}_v) = \boldsymbol{z}_v$, which implies (1) and (2). The prediction model can be expressed as $f_{\mathsf{pred}}(\boldsymbol{z}_v) = y_v$. For example, the optimization problem for binary classification is expressed as follows:

$$\min \mathcal{L}\left(y_v, f_{\mathsf{pred}}(\boldsymbol{z}_v)\right), \quad (3)$$

where $\mathcal{L}$ is a loss function, and f_{pred} is a classification model, such as a fully-connected neural network.

4 MOTIVATIONS

Threat model. As illustrated in Fig. 1, there are many opportunities for attackers to be involved in the hardware supply chain, such as providing malicious 3PIP cores or invading as untrusted offshore fabs. In particular, there are more opportunities for attackers to insert HTs during the design phase as addressed by our threat model. Specifically, attackers may insert HTs into the 3PIP cores and provide them to the primary vendor. Attackers may also invade the design house and directly insert HTs with malicious intent. This scenario, which is common in recent hardware supply chains, involves many employees, partners, 3PIPs, and off-the-shore design houses. Henceforth, the supply chain becomes insecure, providing loopholes for malicious attackers to gain entry.

Our motivations. There are three motivations in this paper, and this section clarifies them from the following perspectives:

- *Motivation 1* describes why we target netlists and node-wise HT detection (Section 4.1).
- *Motivation 2* describes why we employ graph learning for HT detection (Section 4.2).
- *Motivation 3* describes why we introduce domain knowledge for graph learning (Section 4.3).

4.1 *Motivation 1*: Gate-Level Netlists and Node-Wise HT Detection

HT detection in netlists. As illustrated in Fig. 1, the design phase is roughly broken into four levels: specification, behavior level, gate level, and layout level. Any IC design must pass through the *gate level*. Even if some 3PIPs are provided at the *behavior-level* description, they are synthesized to the *gate-level* description. This process is also performed for the *layout level*. However, it includes location information, which is not needed for behavior analysis. Netlists at the gate level are common and useful during the IC design phase. As a real-world example, an HT detection service for gate-level netlists has been released [24]. Therefore, this paper focuses on HT detection in netlists.

Node-wise detection. There are two approaches to HT detection in netlists: circuit-wise and node-wise detection. The circuit-wise detection identifies whether the IC design includes an HT or not. Alternatively, node-wise detection identifies which node is part of an HT. If a circuit-wise HT detection system finds that an HT may exist in the IC design, the user may doubt such an alert. The user must determine whether the product should be redesigned based on the result of the detection system. Additionally, the cause of HT insertion should be carefully analyzed if an HT actually exists. Node-wise detection solves this problem. The results show which node could be a part of an HT. If a node-wise detection method achieves an acceptable rate in terms of detection accuracy, its result must support further analysis and decision-making. Therefore, node-wise detection is more helpful for practical use.

4.2 *Motivation 2*: Graph Learning

As mentioned in Section 3, GNNs are expected to capture the generalized features of graph-structured data. Since many engineers are now joining the hardware design community due to the spread of open-source projects such as RISC-V [25], HT detection that does not require special knowledge of HTs is needed. Additionally, the HT structure can be easily transformed. For example, the transduction method [26] transforms a logic design into a different structure while maintaining the original functionality. If a circuit can be represented with a generalized form, circuits with the same functionality could be embedded into similar latent spaces. The effect of this embedding by automatic feature extraction is significant.

To study the features of HTs, we can collect many HTs by using the automatic HT generation tools [27], [28] that were proposed very recently. However, even if many types of HTs could be collected, it is too difficult to extract a set of comprehensive HT features. GL-based HT detection is expected to automatically extract the features included in the training dataset.

4.3 *Motivation 3*: Domain Knowledge

To make the HT detection system reliable, evidence and theoretical background information should be made clear. Although feature engineering may not be required in GL, knowledge of HT features is vital. Existing studies have demonstrated that their proposed features are useful for HT detection, and work effectively for a certain set of HTs. Therefore, knowledge of HTs still aids in GL-based HT detection.

Introducing domain knowledge is different from automatic feature extraction. Automatic feature extraction, which can be performed by GL, is a process of choosing a set of efficient features from a set of initial features of nodes in a given training dataset. If sufficient initial features are not provided to the initial feature vector of a graph, the GL performance may saturate at an insufficient level. The role of domain knowledge is the selection of a set of sufficient initial features that represent the nodes in a netlist. By selecting common features that can be observed in a wide variety of HTs as initial features based on domain knowledge, feature extraction by GL is expected to work effectively.

Optimizing initial feature vectors based on domain knowledge is helpful. As described in Section 3, the hidden feature vector for a node v is initialized as $\boldsymbol{h}_v^{(0)} = \boldsymbol{x}_v$. Although GL can automatically learn the representation of nodes in a graph, the node features that are provided as an initial feature vector significantly affect the performance of the subsequent task. In [29], it was demonstrated through experiments that GNNs work well if there is a strong correlation between node feature vectors and node labels. Thus, the node features are introduced to sufficiently represent known HT features and HT-related circuit features. This paper aims to find efficient features that represent the characteristics of a node for GL-based HT detection.

5 HT Features

5.1 HT Features at Gate-Level Netlists

In this subsection, we refer to several structural feature-based HT detection methods and categorize what they learn from the perspective of graph-structured data. According to [2], there are several approaches for HT detection in netlists using a variety of features. Hereafter, we refer to the three references [30], [31], and [32] to introduce the representative features based on the studies above.

In [30], 36 structural features are employed for HT detection in total. Some of the features are introduced in some of the earliest studies [15], [16] that were inspired by [3]. They extract feature values from each net in a netlist from the viewpoint of fan-ins, neighbor circuit elements, and the minimum distance to the specific circuit elements. Ref. [30] further introduced pyramidal structure-based features called *fan_in_uxdy*, which improve detection performance. Table 1 shows the 36 features presented in [30]. The 'category' column is introduced later. The features mainly focus on the structural features of a netlist, that is, the topological features of a graph.

In [31], 15 features are employed, and they are shown in Table 2. Different from [30], the features mainly focus on functional behavior such as the static probability and signal rate (No. 6–15 in Table 2). The functional behavior-based features reflect the functionality of a netlist, such as rare or frequent signal transitions. Although the structural features can capture the structure of a set of nodes, they do not explicitly capture the behavior of the circuit. Functional behavior-based features solve the problem, and they capture the behavior of the circuit explicitly.

TABLE 1: HT features employed in [30].

No.	Feature	Category
1–2	No. of fan-ins up to 4 and 5-level away from the input side	❶ Degree
3	No. of flip-flops up to 4-level away from the input side	❷ Neighboring node
4–5	No. of flip-flops up to 3 and 4-level away from the output side	❷ Neighboring node
6–7	No. of loops up to 4 and 5-level on the input side	❸ Relative position
8	Min. level to any primary input	❸ Relative position
9	Min. level to any primary output	❸ Relative position
10	Min. level to any flip-flops from the output side	❸ Relative position
11	Min. level to any multiplexer from the output side	❸ Relative position
12–36	Pyramidal structure-based feature within 4 levels	❹ Surrounding structure

TABLE 2: HT features employed in [31].

No.	Feature	Category
1–2	No. of immediate fan-ins and fan-outs	❶ Degrees
3	Cell type driving the net	❷ Neighboring node
4–5	Min. distances from PI and PO	❸ Relative position
6	Static probability	❺ Functional behavior
7	Signal rate	❺ Functional behavior
8	Toggle rate	❺ Functional behavior
9	Min. toggle rate of the fan-outs	❺ Functional behavior
10	Entropy of the driver function	❺ Functional behavior
11–15	Lowest, highest, average, and std. dev. of controllability	❺ Functional behavior

In [32], six testability metrics-based features are employed. Table 3 shows the six features presented in [32]. These features are known as *SCOAP* values, and they are inherently utilized to evaluate the testability of a circuit [33]. The *SCOAP* values were first introduced into HT detection by a recent study [17] and are often employed. Since an HT is hard to observe and easy to control from outside the circuit, the *SCOAP* values are reasonable for HT detection.

Toward GL. Based on the observation of several existing HT detection methods, we analyze the features and categorize them from the viewpoint of GL.

- ❶ **Degree** indicates the degree of a node in a graph, which is related to the number of edges connected to the node.
- ❷ **Neighboring node** indicates the types of nodes neighboring a target node.
- ❸ **Relative position** indicates the relative position to the specific sets of nodes.
- ❹ **Surrounding structure** indicates the surrounding structure from a target node.
- ❺ **Functional behavior** indicates the *functional behavior* of a node, i.e., it characterizes the behavior of a set of nodes. An example is static probability. Typically,

TABLE 3: HT features employed in [32].

No.	Feature	Category
1–2	Controllability (CC0 and CC1)	❺ Functional behavior
3	Observability (CO)	❺ Functional behavior
4–5	Sequential controllability (SC0 and SC1)	❺ Functional behavior
6	Sequential observability (SO)	❺ Functional behavior

the trigger circuit of an HT rarely activates the trigger signal. The node that outputs the trigger signal is rarely activated. Such behavior cannot be observed by simply analyzing structural features.

Section 5.2 analyzes the representation capability of graph encoding models from the perspective of the categories ❶–❺.

5.2 Representing HT Features with GL

Here we bridge the gap between the known HT features and GL. As described in the previous section, the HT features are classified into five categories. We demonstrate how GL captures these features in a formal manner.

❶ **Degree**: Fan-ins and fan-outs of a node are useful information for HT detection. For example, a trigger circuit composed of a combinatorial circuit often has many fan-ins at two or three levels from the trigger signal wire to implement rare conditions. The node degrees can be directly assigned to the initial feature vector to clearly make the graph encoder model identify node degrees.

❷ **Neighboring node**: Information about the types of nodes neighboring a target node is helpful for HT detection. To utilize the information with GL, a node type is assigned to the initial feature vector.

The node types are the most fundamental features for representing the characteristics of nodes in a netlist, as demonstrated in [8] and [9]. In this paper, the node types are used as the features of a baseline, the details of which are presented later.

❸ **Relative position**: When the surrounding structures of two subgraphs are identical, the graph encoder model that considers only the feature categories ❶ and/or ❷ cannot identify the subgraphs. According to (2), the final node feature vector of v is expressed as $\boldsymbol{z}_v = \boldsymbol{h}^{(L)} = \mathsf{COMBINE}^{(L)}\left(\boldsymbol{h}_v^{(L-1)}, \boldsymbol{m}_v^{(L-1)}\right)$. When $\boldsymbol{h}_v^{(L-1)} = \boldsymbol{h}_u^{(L-1)}$ and $\boldsymbol{m}_v^{(L-1)} = \boldsymbol{m}_u^{(L-1)}$ for two nodes v and u, the node feature vectors of the two nodes are identical, i.e., $\boldsymbol{z}_v = \boldsymbol{z}_u$. In a netlist, such a case can appear in a ring oscillator that is composed of multiple inverter gates to form a loop structure or a multi-bit counter that is composed of several 1-bit counters. These circuits can be used as a payload circuit or a sequential trigger circuit of an HT.

To overcome the limitation of GNNs, a position-aware GNN model was proposed in [34]. In the method, a random subset of k nodes are chosen as *anchor-sets* and their node feature vectors are included in the target node feature vector. Inspired by [34], we state the following proposition:

Proposition 1. *Let G_{anchor} be a graph where an initial feature vector $\boldsymbol{x}_v$ that includes information on anchor-sets is assigned to each node v. Let $\phi\colon \mathbb{R}^d \to \mathbb{R}^d$ be a GNN layer, and let $\mathcal{F}_{\mathsf{enc}}$ denote a set of GNN models, where those models consist of more than one layer ϕ and are injective. There exists a graph encoder model $f_{\mathsf{enc}} \in \mathcal{F}_{\mathsf{enc}}$ that accepts G_{anchor} as an input and identifies the difference between two nodes in G_{anchor} in terms of their position in a graph.*

The proof of Proposition 1 is shown in Appendix A. According to [34], choosing a set of anchor nodes is a difficult problem for an inductive learning task. From the viewpoint of netlists, primary inputs and outputs are reasonable candidates for *anchor-sets* making them useful for HT detection. Therefore, a graph encoder model can best identify the positional feature of a node by employing minimum distances to any primary inputs and outputs as initial feature values.

❹ **Surrounding structure**: How a netlist is represented as a graph structure is one of the most problematic issues in GL. There are two methods for representing a netlist: undirected graphs and directed graphs. An edge between two nodes in a graph is directional in a directed graph, whereas this is not the case in an undirected graph. The signal wires in a netlist are inherently directional, and thus, a directed graph seems to be suitable for representing a netlist. However, a directed graph cannot sufficiently represent a netlist for graph encoder models. With a directed graph, the aggregation function gathers only one direction of each edge, and thus, the nodes connected to the opposite side are ignored. Table 4 shows the graph models employed in recent processing methods using GNNs. According to [8], which aims to represent netlists for multiple downstream tasks, representing a netlist as an undirected graph is more efficient than representing it as a directed graph. It should be noted that GNN4TJ [7], which employs directed graphs, mainly focuses on HTs written in RTL. Therefore, a directed graph may be inefficient for netlists.

TABLE 4: GL-based methods for netlists.

Graph	Methods
Undirected graph	GNN-RE [8], GATE-Net [9]
Directed graph	[35], GNN4TJ [7]

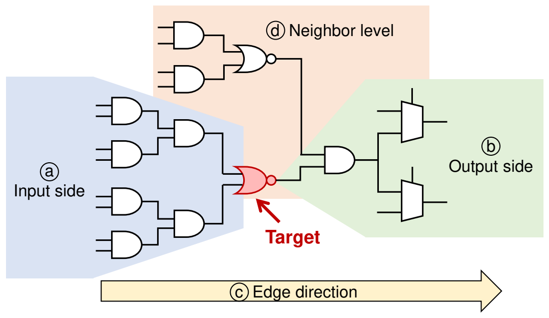


Fig. 2: Example of graph-related features for a netlist.

The representation capability of directed and undirected graphs is broken down, and a new model for representing a netlist is proposed. The four points of the graph-related features for a netlist are as follows:

- ⓐ **Input side**: The structure of the input side from target node v.

TABLE 5: Graph structure and its representation capability in a graph encoder model.

Graph type	ⓐ Input side	ⓑ Output side	ⓒ Edge direction	ⓓ Neighbor level
Directed	✓		✓	
Undirected	✓	✓		✓
Undirected (with directional edge attribute)	✓	✓	✓	✓

- ⓑ **Output side**: The structure of the output side from target node v.
- ⓒ **Edge direction**: The direction of an edge. Namely, the direction of a wire in a netlist.
- ⓓ **Neighbor level**: The structure of the neighbor level from target node v.

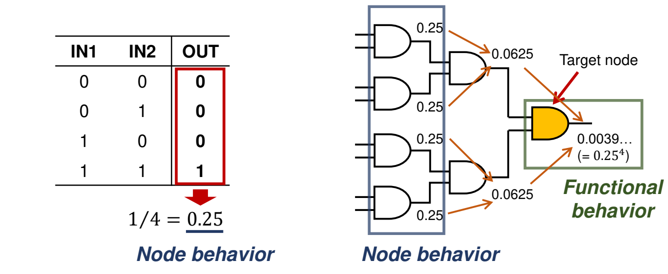
Fig. 2 illustrates an example of the graph-related features using a circuit. In this figure, we focus on the 'target' node colored in red. ⓐ **Input side** corresponds to the area shaded in light blue. The nodes in this area are connected to the input side of the target gate. ⓑ **Output side** corresponds to the area shaded in light green. The nodes in this area are connected to the output side of the target gate. ⓒ **Edge direction** shows whether the edge direction is preserved in the modeled graph. ⓓ **Neighbor level** corresponds to the area shaded in orange. The neighbor-level area can be reached by traversing some hops toward the output side and then some hops backward from the input side or vice versa.

Next, we summarize the representation capability of graph models. Table 5 shows the graph structures and their representation capability in a graph encoder model. A directed graph can represent the input side and the edge direction of a netlist when a graph encoder model aggregates the nodes of the input side. However, the nodes of the output side are not aggregated in this situation. Similarly, the neighbor levels are not considered unless there is a loop structure. Conversely, an undirected graph can represent both the input and output sides. Due to both-side aggregation, it can also represent the nodes in neighbor levels. However, it lacks directional information.

Next, we propose the directional edge attribute for undirected graphs, which are called *edge-attributed undirected graphs (EAUGs)*.

Definition 1 (Edge-attributed undirected graph)**.** *An edge-attributed undirected graph (EAUG) is a graph* $G_{\mathsf{EAUG}} = (V, E, X, D)$ *that represents a netlist with a set of initial node vectors* $X = \{\boldsymbol{x}_v : v \in V\}$ *and a set of edge attribute vectors* $D = \{\boldsymbol{d}_{u \to v} : u, v \in V\}$.

The EAUG overcomes the shortcoming of the undirected graph. Let $e_{u \to v}$ denote an edge from a node u to another node v. $\boldsymbol{d}_{u \to v} \in \mathbb{R}^d$ is an edge attribute vector assigned to $e_{u \to v}$. In a typical undirected graph, the edges $e_{u \to v}$ and $e_{v \to u}$ are not distinguished. Furthermore, when feature vectors are assigned to the edges of an undirected graph, $\boldsymbol{d}_{u \to v}$ and $\boldsymbol{d}_{v \to u}$ are usually the same. In the EAUG, we distinguish two edges $e_{u \to v}$ and $e_{v \to u}$ by assigning different edge attributes as $\boldsymbol{d}_{u \to v} \neq \boldsymbol{d}_{v \to u}$ Then, we can state the following proposition.



IN1	IN2	OUT
0	0	0
0	1	0
1	0	0
1	1	1

(a) Truth table of a two-input AND gate. (b) HT trigger circuit (Subgraph of G).

Fig. 3: Example of node behavior and functional behavior.

Proposition 2. *There exists a graph encoder model* $f_{\mathsf{enc}} \in \mathcal{F}_{\mathsf{enc}}$ *that accepts an edge-attributed undirected graph* G_{EAUG} *as input and represents all the graph-related features ⓐ–ⓓ.*

The proof of Proposition 2 is shown in Appendix B. For example, for a given netlist, we assign a vector $(1, 0)$ to a forward direction edge and $(0, 1)$ to a backward direction edge. Then, we can construct an EAUG for a given netlist.

❺ **Functional behavior**: In this section, we observe the global behavior of a circuit. First, we consider the behavior of each node. We call a function assigned to each node *node behavior*, which is identified by only the node. However, we can only observe the local behavior of each node by simply considering the *node behavior*. To observe the global behavior of a circuit, we need a new feature that accounts for the *node behavior* of the neighboring nodes. Then, we introduce the *functional behavior*, which characterizes the behavior of a set of nodes. The *functional behavior* of a target node v refers to a feature computed using an arbitrary function that takes the node behaviors of v and its neighboring nodes $\mathcal{N}_k(v)$, where $\mathcal{N}_k(\cdot)$ is a set of neighboring nodes within k hops from a node. In a netlist, there are several metrics that characterize the behavior of a set of nodes by assigning a function to each node, such as the static probability and switching probability. We consider such metrics according to the *functional behavior*.

Specifically, the static probability is an example of *functional behavior*, which shows the probability that a signal holds a logic value of 1 during a period. Fig. 3 illustrates an example of an HT trigger circuit that consists of seven two-input AND gates. First, we refer to the truth table to consider the local behavior of a two-input AND gate. Here, we regard the probability of outputting 1 as the *node behavior*. In this case, the *node behavior* of a two-input AND gate is 0.25, as shown in Fig. 3(a). Then, we consider the HT trigger circuit depicted in Fig. 3(b). Starting from the *node behavior* values, we can calculate the static probabilities of the target node. We regard the static probability as the *functional behavior*, which characterizes the functionality of a set of neighboring nodes within two hops from the target node. As exemplified above, a *functional behavior* also characterizes the functionality of HTs well, especially HT triggers, as shown in Fig. 3(b).

To formulate the *functional behavior*, let $\mathsf{type}(v)$ be a node type and $\mathsf{nb}(v)$ be the *node behavior* of node v. The *functional behavior* $\mathsf{fb}(v)$ of a node v can be calculated as follows:

$$\mathsf{fb}(v) \leftarrow \Gamma_{\mathsf{type}(v)}\left(\mathsf{nb}(v), \{\mathsf{nb}(u) : \forall u \in \mathcal{N}_k(v)\}\right), \quad (4)$$

where $\Gamma_{\mathsf{type}(v)}$ is an arbitrary function parameterized by $\mathsf{type}(v)$. The following proposition states that a graph encoder model that represents such a *functional behavior* exists.

Proposition 3. *Let G' be a graph constructed as a G_{EAUG} where the node type and a feature value related to a* functional behavior *is assigned to each node v. There exists a graph encoder model $f_{\mathsf{enc}} \in \mathcal{F}_{\mathsf{enc}}$ that accepts G' as input and identifies the* functional behavior *k hops away from each node v for a given k.*

The proof of Proposition 3 is provided in Appendix C.

We consider the feature extraction model that extracts the graph structure, initial feature vectors, and edge attribute vectors for a given netlist. Let Λ be a feature extraction model that accepts a netlist, which is denoted by ϑ as input, and outputs a tuple (V, E, X, D) of a set of nodes, set of edges, set of node feature vectors, and set of edge attribute vectors. In some cases, X or D can be an empty set. Let $f_{\mathsf{enc};\Lambda(\vartheta)}(v)$ be a graph encoder model with respect to $\Lambda(\vartheta) = (V, E, X, D)$ used to obtain the node feature vector of node v during the calculation. $f_{\mathsf{enc}}(v)$ implicitly refers to the V, E, X, and D of a netlist ϑ. If $\exists u, v \in V, u \neq v, f_{\mathsf{enc};\Lambda(\vartheta)}(u) \neq f_{\mathsf{enc};\Lambda(\vartheta)}(v)$, it means that the combination of models Λ and f_{enc} can distinguish the two nodes u and v. $P_{\vartheta}(\Lambda, f_{\mathsf{enc}})$ denotes the representation capability for the nodes in a given netlist ϑ when using Λ as a feature extraction model and f_{enc} as a graph encoder model. Here, the representation capability for a netlist is defined.

Definition 2 (Representation capability for a netlist)**.** *Let Λ_1 and Λ_2 be feature extraction models, and let $f^{(1)}_{\mathsf{enc}}$ and $f^{(2)}_{\mathsf{enc}}$ denote graph encoder models. The representation capability $P_{\vartheta}(\Lambda_1, f^{(1)}_{\mathsf{enc}})$ is greater than $P_{\vartheta}(\Lambda_2, f^{(2)}_{\mathsf{enc}})$ if and only if $\forall u, v \in V, u \neq v, f^{(2)}_{\mathsf{enc};\Lambda_2(\vartheta)}(u) \neq f^{(2)}_{\mathsf{enc};\Lambda_2(\vartheta)}(v) \implies f^{(1)}_{\mathsf{enc};\Lambda_1(\vartheta)}(u) \neq f^{(1)}_{\mathsf{enc};\Lambda_1(\vartheta)}(v)$, and the relationship between the two representation capabilities is expressed as follows:*

$$P_{(\vartheta)}(\Lambda_1, f^{(1)}_{\mathsf{enc}}) \succeq P_{(\vartheta)}(\Lambda_2, f^{(2)}_{\mathsf{enc}}). \quad (5)$$

In particular, $P_{\vartheta}(\Lambda_1, f^{(1)}_{\mathsf{enc}})$ is ***strictly*** *greater than $P_{\vartheta}(\Lambda_2, f^{(2)}_{\mathsf{enc}})$ if and only if $P_{(\vartheta)}(\Lambda_1, f^{(1)}_{\mathsf{enc}}) \succeq P_{(\vartheta)}(\Lambda_2, f^{(2)}_{\mathsf{enc}})$ and $\exists u, v \in V, u \neq v, f^{(1)}_{\mathsf{enc};\Lambda_1(\vartheta)}(u) \neq f^{(1)}_{\mathsf{enc};\Lambda_1(\vartheta)}(v)$ but $f^{(2)}_{\mathsf{enc};\Lambda_2(\vartheta)}(u) = f^{(2)}_{\mathsf{enc};\Lambda_2(\vartheta)}(v)$.*

To compare the representation capability of graph encoder models, we introduce two feature extraction models for a netlist: baseline and netlist feature extraction models.

Definition 3 (Baseline feature extraction model)**.** *A baseline feature extraction model $\Lambda_{\mathsf{baseline}}(\vartheta) = (V, E, X, D)$ extracts a set of node initial feature vectors X that indicates the node types of each node $v \in V$ and, no edge attributes are extracted, i.e., $D = \emptyset$.*

The model that extracts the features belonging to the feature category ❷ is a baseline feature model.

Definition 4 (Netlist feature extraction model)**.** *A netlist feature extraction model $\Lambda_{\mathsf{netlist}}(\vartheta) = (V, E, X, D)$ extracts a set X of node initial feature vectors that indicate the node degrees, node types, features of the relative position to the anchor-sets, and the features of the functional behavior of each node $v \in V$ and a set D of edge attribute vectors that indicate the edge direction of nodes $u, v \in V$.*

The model that extracts the node features and edge attributes belonging to the feature categories ❶–❺ is a netlist feature model.

Now, we can state the following theorem:

Theorem 1. *The representation capability of a netlist feature extraction model is strictly greater than the representation capability of a baseline extraction model.*

Proof: Let $f^{(1)}_{\mathsf{enc}}$ and $f^{(2)}_{\mathsf{enc}}$ be graph encoder models that are assumed to be injective. Since the extracted features from the netlist feature extraction model contain the features from the baseline feature extraction model, $\forall u, v \in V, u \neq v, f_{\mathsf{g2};\Lambda_{\mathsf{baseline}}(\vartheta)}(u) \neq f_{\mathsf{g2};\Lambda_{\mathsf{baseline}}(\vartheta)}(v) \implies f_{\mathsf{g1};\Lambda_{\mathsf{netlist}}(\vartheta)}(u) \neq f_{\mathsf{g1};\Lambda_{\mathsf{netlist}}(\vartheta)}(v)$. Thus, $P_{(\vartheta)}(\Lambda_{\mathsf{netlist}}, f^{(1)}_{\mathsf{enc}}) \succeq P_{(\vartheta)}(\Lambda_{\mathsf{baseline}}, f^{(2)}_{\mathsf{enc}})$. Furthermore, according to Propositions 1, 2, and 3, there exists a set of nodes that the netlist feature extraction model can distinguish while the baseline feature extraction model cannot. Thus, $P_{(\vartheta)}(\Lambda_{\mathsf{netlist}}, f^{(1)}_{\mathsf{enc}}) \succ P_{(\vartheta)}(\Lambda_{\mathsf{baseline}}, f^{(2)}_{\mathsf{enc}})$. Therefore, Theorem 1 holds. □

By Theorem 1, such a feature extraction model represents the feature categories ❶–❺.

It should be noted that the assumption that graph encoder models are injective is a realistic modeling approach for theoretical analysis. For simplicity, we consider a neural network model as a matrix product. In general, the weights of the neural network model are initialized with Gaussian random values. Since the weight matrices have full rank in most cases, the matrix product becomes injective. A precise analysis of the injectiveness will be discussed in future work.

5.3 Summary of HT Features

This section presents the five feature categories for GL so that they correspond to ❶–❺ and bridges the gap between HT features and GL. As a result, this section clarifies what HT features a graph encoder model captures through theoretical analysis.

In the existing HT detection methods such as those presented in [30] and [32], we must discover effective features for HT detection. To consider the topology around each node in a netlist, we should carefully analyze the relationship between a node and another node. Therefore, the search space for feature engineering reaches $\mathcal{O}(|V| \times |V|)$. Alternatively, the proposed method automatically extracts the structural features by using graph extraction models. It is sufficient to define the initial feature vectors that represent the characteristics of nodes well. To explore such features, we analyze all nodes and extract representative features. Therefore, the search space for feature engineering becomes $\mathcal{O}(|V|)$. In Section 7, we will demonstrate through experiments that a graph encoder model automatically extracts HT features.

6 Proposed Method

6.1 Overview

Fig. 4 illustrates an overview of the proposed method, which is called `NHTD-GL`. As shown in Fig. 4, an HDL design is

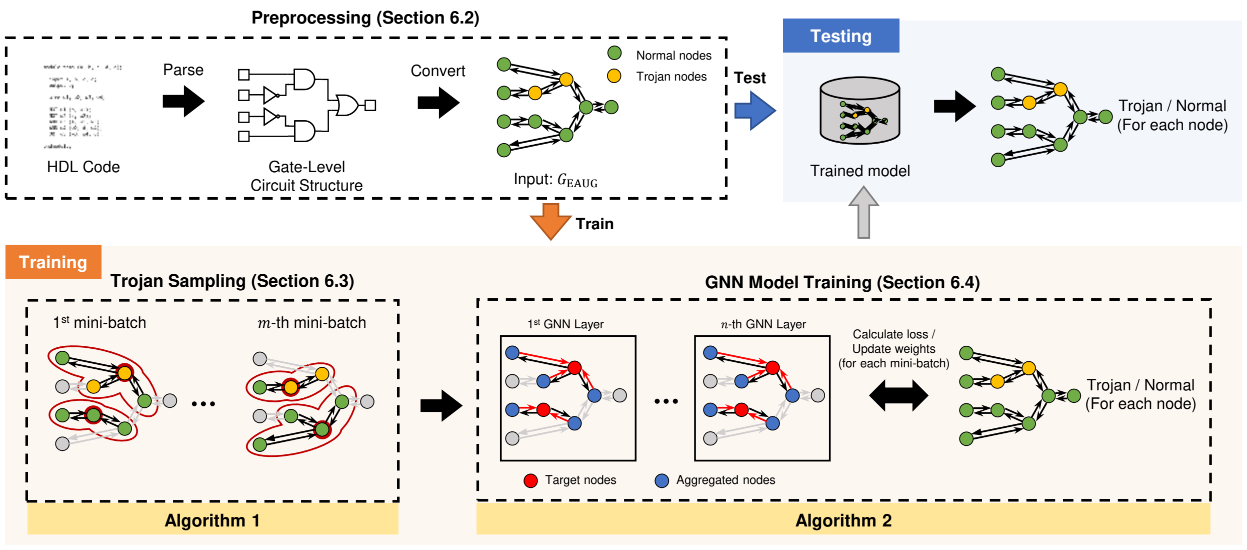


Fig. 4: Overview of NHTD−GL.

converted to a graph that represents the netlist, constructing an EAUG, G_{EAUG}. Training and/or testing of datasets are based on the EAUGs.

During the preprocessing phase (see Section 6.2 in detail), we prepare netlists to be trained with the GNN model. First, we convert the netlists to EAUGs where each element of the circuit is assigned to a node, and each wire is assigned to an edge (❹). Then, we assign the initial feature value that covers the feature categories ❶–❸ and ❺ to each node in the graph. At the same time, we assign the edge attribute that shows the direction of the edge. Finally, we construct the dataset including multiple EAUGs.

During the training phase (see Section 6.3 and Section 6.4 in detail), we start with Trojan sampling on the given EAUGs to balance the normal and Trojan nodes in each mini-batch. To train HTs, we have to address the problem that HTs are quite tiny. Because of their stealth, they are often constructed on a tiny scale. Therefore, the numbers of genuine nodes and Trojan nodes are imbalanced in the training dataset. To accurately perform node-wise classification for HT detection, we should adequately deal with this imbalance. To address this problem, we propose the *Trojan sampling* method to train the imbalanced HT dataset effectively. After that, we train the mini-batches to classify each node in a graph as Trojan or normal.

During the testing phase, we classify each node in an EAUG as either normal or Trojan using the trained model.

6.2 Initial Feature Vector of a Node

Based on the discussion in Section 5, we design the initial feature vector to be assigned to each node.

Table 6 shows the initial feature vector assigned to each node v in the proposed model. Features 1 and 2, which are categorized as ❶, correspond to the in-degree and out-degree of a node v, respectively. To effectively train the features, the feature values are standardized. Features 3–42, which are categorized as ❷, show the types of each node v, e.g., a two-input AND gate or a flip-flop. Features 43 and 44 show the minimum distance to any primary

TABLE 6: Initial feature vector used in the proposed model [a].

No.	Feature	Category
1	In-degree	❶
2	Out-degree	❶
3–42	Node types	❷
43	Min. distance to any primary input	❸
44	Min. distance to any primary output	❸
45	Static probability (0) of logic gates	❺
46	Static probability (1) of logic gates	❺

[a] We construct an EAUG, which is categorized as ❹.

input and output, respectively. Primary inputs and outputs are characteristic nodes for HT detection. We use them as *anchor-sets*, and thus, the features are categorized as ❸. Features 45 and 46 show the probability that a logic gate outputs 0 and 1, respectively. We provide the standardized feature values to the initial vector, and they are helpful for calculating the *functional behavior* of a circuit and can be categorized as ❺. In summary, the features cover the feature categories ❶–❸ and ❺, which are described in Section 5.

6.3 Trojan Sampling

Since an HT is tiny compared to a normal circuit, the numbers of normal nodes and Trojan nodes are significantly imbalanced. This issue must be solved for HT detection using ML. In conventional ML models (not GL models), over-sampling (or under-sampling) approaches can be adopted to enhance the minority classes. SMOTE [36] is a well-known method for over-sampling that synthetically generates minority class samples and enhances minority classes. However, one cannot directly adopt this approach for graph-structured data since a conventional over-sampling method cannot consider the adjacent matrix. Although the graph-version SMOTE method has been proposed very recently [37], it is difficult to construct an accurate decoder model. Therefore, in NHTD−GL, we first split normal nodes into several subgraphs. Then, we construct mini-batches by combining a Trojan subgraph with different normal subgraphs.

Algorithm 1 Trojan Sampling Algorithm

```
Input: Sets of Trojan and normal nodes $V_\mathsf{t} \cup V_\mathsf{n} = V$, the number of mini-batches $m$
Output: Set of mini-batches $\mathcal{B}$
1: $\mathcal{B} \leftarrow \emptyset$
2: Split $V_\mathsf{n}$ into $m$ subsets $V_\mathsf{n}^{(i)}$ randomly, where $i \in [m]$ and $|V_\mathsf{n}^{(i)}| \leq \lceil |V_\mathsf{n}|/m \rceil$.
3: for $i = 1$ to $m$ do
4:     $B^{(i)} \leftarrow \{V_\mathsf{n}^{(i)}, V_\mathsf{t}\}$
5:     $\mathcal{B} \leftarrow \mathcal{B} \cup \{B^{(i)}\}$
6: end for
7: return $\mathcal{B}$
```

Algorithm 2 NHTD-GL: Training Algorithm

```
Input: Set of EAUGs $\mathcal{G}_\mathsf{EAUG}$, set of labels $Y$, the number of mini-batches $m$
Output: Trained model $f$
1: Initialize the model $f$.
2: repeat                                  // Repeat one-epoch process
3:     for all $G_\mathsf{EAUG} = (V, E, X, D) \in \mathcal{G}_\mathsf{EAUG}$ do
4:         $\mathcal{B} \leftarrow$ TrojanSamplingAlgorithm$(V, m)$
5:         for all $B \in \mathcal{B}$ do
6:             Calculate the loss of the model $f$ with $\boldsymbol{x}_v$, $\boldsymbol{d}_{\mathcal{N}(v)\to v}$, and $\boldsymbol{y}_v$ for all $v \in B$.
7:             Update the weight of the model $f$.
8:         end for
9:     end for
10: until Training converges.
11: return $f$
```

Algorithm 1 shows the Trojan sampling algorithm. Let V_t and V_n be a set of Trojan nodes and a set of normal nodes, respectively. Given the number of mini-batches m, we first split the set of normal nodes into m subsets so that $V_\mathsf{n} = \{V_\mathsf{n}^{(1)}, \cdots, V_\mathsf{n}^{(m)}\}$. Note that $|V_\mathsf{n}^{(i)}| \leq \lceil |V_\mathsf{n}|/m \rceil$, where $|\cdot|$ shows the cardinality of a given set. Then, we construct m mini-batches $\mathcal{B} = \{B^{(1)}, \cdots, B^{(m)}\}$, where $B^{(i)} = \{V_\mathsf{n}^{(i)}, V_\mathsf{t}\}, i \in [m]$, and we train the mini-batches represented by $\mathcal{B}$ during each iteration.

6.4 GNN Model

As described in Section 5, EAUG corresponds to the feature category ❹. According to (7), we aggregate the edge attribute as well as the feature vectors of the nodes that are adjacent to node v. We concatenate the edge attributes and node feature vectors of u for each v in our implementation.

To classify each node v as normal or Trojan, we assign a one-hot vector $\boldsymbol{y}_v \in \mathbb{R}^2$, where $(1, 0)$ indicates a normal node and $(0, 1)$ indicates a Trojan. Then, we can set the optimization problem using the loss function $\mathcal{L}$ as follows:

$$\min \sum_{\forall v \in V} \mathcal{L}\left(\boldsymbol{y}_v, f(\boldsymbol{x}_v)\right), \tag{6}$$

where $f(\boldsymbol{x}_v) = f_\mathsf{pred}(f_\mathsf{enc}(\boldsymbol{x}_v))$. We repeat the training for each graph G in a training dataset.

Algorithm 2 describes the training algorithm. Let $\mathcal{G}_\mathsf{EAUG}$ be a set of EAUGs that corresponds to a set of netlists that need to be trained. Y is a set of labels $\boldsymbol{y}_v$ assigned to each node v. We draw mini-batches $\mathcal{B}$ from each EAUG G_EAUG and calculate the loss of the model f as in (6) using the nodes in a mini-batch. Then, we update the weight of model f to minimize the loss between the model's outputs and the labels $\boldsymbol{y}_v, v \in B$. We repeat the one-epoch process (ll. 3–9 in Algorithm 2) several times until the training converges. In other words, we end the training process when the number of processed epochs reaches the limit, or the loss of the model f no longer decreases.

7 Evaluation

The evaluation in this section aims to answer the following research questions through experiments:

- **RQ1**: Does the domain knowledge on HTs enhance the detection performance? (Section 7.2.1)
- **RQ2**: Does the proposed method enhance the detection performance for node-wise HT detection in netlists? (Section 7.2.2)
- **RQ3**: Does GL automatically extract effective features? (Section 7.3)

Each research question corresponds to the motivations described in Section 4: **RQ1** corresponds to *Motivation 3*, **RQ2** corresponds to *Motivation 1*, and **RQ3** corresponds to *Motivation 2*.

7.1 Setup

Datasets. For the dataset, we used 24 netlists from Trust-HUB [13], [14] (the details are presented in Appendix D). In the Trust-HUB benchmarks, HTs with various structures are embedded into several types of netlists, and the insertion points are indicated by comments in the source code. The total number of nodes in the dataset is 621140, and the number of Trojan nodes is 1262. These numbers mean that Trojan nodes are only 0.2% of the total data. The Trojan Sampling Algorithm described in Section 6.3 is applied to better learn the imbalanced training data.

Furthermore, randomly generated samples are employed in Section 7.3, and the details are presented in this section.

Models. We trained and identified the dataset with the graph structure described in the Datasets section. We used GAT [38], MPNN [22], and GIN [23] as the GNN models for the evaluation. The graph encoder models for the EAUGs are described in Appendix E. We set the following parameters for all the models: The maximum number of epochs was 1000, and early stopping [39] was applied, which completed the training process when the loss did not decrease for 50 epochs. The learning rate was 0.1, the optimization algorithm was Adam, the activation function was an exponential linear unit (ELU) function, and binary cross-entropy was used as the loss function.

Since the best parameters for the number of mini-batches (#batches), the number of GNN layers (#layers), and the number of dimensions of the feature vectors after the graph convolution (#units) depend on the GNN model, we searched for them using a grid search. We explain the details of this search in Appendix F. Table 7 shows the best parameters in each model for the proposed method and the baseline method described in Section 7.2.

Evaluation metrics. To evaluate the performance, we performed a leave-one-out cross-validation. For each of the 24

TABLE 7: Best parameters for each model.

Method	Model	#batches	#layers	#units
Proposed	GAT	20	3	16
	MPNN	15	2	32
	GIN	20	2	16
Baseline	GAT	5	3	32
	MPNN	15	3	32
	GIN	20	2	32

netlists described in the Datasets section, we used one as a test sample and the remaining 23 as training samples. We performed this validation on all of the netlists and evaluated the average of the 24 classification results. The evaluation metrics are recall, precision, F1-score, and accuracy, in which a Trojan net is regarded as a positive sample. In addition, the training and test times are reported in Appendix G.

7.2 Performance Evaluation of Trust-HUB Benchmarks Compared with Baseline and State-of-the-Art Methods

7.2.1 Comparison with the baseline method (**RQ1**)

In this experiment, the performance with and without domain knowledge in the features was evaluated. As mentioned in Section 4.3, feature selection based on domain knowledge is effective in GNNs. To confirm this assumption, we compared the features of the proposed method as mentioned in Table 6 with the features consisting only of node types. In general, the node type is the most primitive feature when representing the information in a graph format [9]. Therefore, we selected features 3–42, which are categorized as ❷ as shown in Table 6. We define them as baseline node features that do not contain domain knowledge and GL based on these features is considered to be the *baseline method*.

To fairly compare the results, we searched for the best parameters for the baseline method by using a grid search as in the proposed method. The details are included in Appendix F.

Detection results. Table 8 shows the detection results of this experiment (the detailed results are shown in Appendix H). The boldfaced font indicates the highest rate in each method. As shown in Table 8, the proposed method performed as well or better than the baseline method on all evaluation metrics. In particular, the GAT model outperformed the other models on all metrics. Therefore, GAT is the best GNN model for HT detection in Trust-HUB. Essentially, the baseline in which the features in category ❷ are used obtains sufficient detection results. Adding the features in categories ❶, ❸, ❹, and ❺ is expected to improve the detection performance, as shown in Theorem 1. According to the results listed in Tables 16–21 in Appendix H, the detection performance of several netlists is improved. For example, the proposed method with the GAT or MPNN model improves the recall and F1-score for s35932-T300, whose payload is a ring oscillator. In addition, the standard deviations of the F1-scores for the GAT and MPNN models are decreased by the proposed method. Thus, the proposed method contributes to enhancing the detection performance.

This result means that we can say "YES" to **RQ1**, which corresponds to *Motivation 3*. By eliminating the feature categories ❶, ❸, ❹, and ❺, which are based on HT domain

TABLE 8: Detection results of Trust-HUB for the proposed method and the baseline method.

Method	Model	Recall	Precision	F1-score	Accuracy
Proposed	GAT	**0.890**	**0.978**	**0.921**	**0.998**
	MPNN	0.865	0.933	0.887	**0.998**
	GIN	0.585	0.616	0.498	0.988
Baseline	GAT	**0.880**	**0.976**	**0.915**	**0.998**
	MPNN	0.879	0.900	0.871	**0.998**
	GIN	0.237	0.392	0.240	0.986

TABLE 9: Comparison with existing methods.

Method	Model	Recall	Precision	F1-score	Accuracy
Proposed	GAT	**0.890**	**0.978**	**0.921**	**0.998**
[30]	RF	0.636	0.957	0.667	0.994
[32]	BT	0.825	0.866	0.827	0.983

knowledge, the recall, precision and F1-score are decreased. The accuracy is the same as the baseline method because the Trojan nodes make up only 0.2% of the whole training data, and the accuracy score is greatly affected by true negatives. For example, if the classifier determines all nodes as normal nets (i.e., negative), the accuracy would be 99.8%. Therefore, there is no difference in accuracy among the methods, but it is clear that the proposed method can identify the Trojan nodes better than the baseline method because the recall, precision, and F1-score are higher. This result confirms that the features based on knowledge of HT features are essential for node-wise HT detection.

Note that, adding initial features requires additional costs such as an increase in computational time and resources for training a model. Therefore, covering feature categories ❶–❺ should be sufficient.

7.2.2 Comparison with state-of-the-art methods (**RQ2**)

In this experiment, we compared the proposed method with the two state-of-the-art methods [30], [32] mentioned in Section 5.1.

The method described in [30] extracts 36 features that represent the HT structure well and identifies HT wires by using random forest (RF). Since the number of false positives is very small, this method is less likely to misidentify normal circuits. The method described in [32] is based on testability measures, which are effective features for HT detection, and it employs the adaptive synthetic (ADASYN) sampling approach to better learn imbalanced training data. They validated it with four supervised algorithms, and the highest metrics were employed when using bagged trees (BT). Both methods [30], [32] have been reported to perform well on the Trust-HUB dataset based on effective feature engineering.

Detection results. Table 9 shows the comparison results with existing methods. We adopted the GAT model, which demonstrates the highest classification performance, as shown in Table 8, as the proposed method for the comparison. As shown in Table 9, the proposed method outperforms the two methods [30], [32] on all evaluation metrics.

We compared the proposed method with the state-of-the-art GNN-based methods proposed in [40] and [9]. Table 10 shows the comparison with [40]. Although the datasets are different in terms of the two methods, the results are obtained citing [40] based on the average scores

TABLE 10: Comparison with [40].

Method	Model	Recall	Precision	F1-score
Proposed	GAT	0.890	0.978	0.921
[40]	SGCN	0.841	0.912	0.860

TABLE 11: Comparison with [9].

Trigger type	Method	Recall	Precision	F1-score
Combinatorial	Proposed	0.850	0.968	0.889
	[9]	0.90	0.95	0.87
Sequential	Proposed	0.947	0.990	0.965
	[9]	0.92	1.00	0.96

of the datasets containing several different base circuits. As shown in the table, the proposed method outperforms the method in [40] on all of the metrics. The major difference is that the method in [40] uses only the node types, which are categorized as ❷. Therefore, using a set of features that covers categories ❶–❺ contributes to enhancing the detection performance. Table 11 shows the comparison with [9]. In the comparison, we summarize the results of the proposed method based on the trigger types, as in [9]. Although the results are almost comparable, the proposed method obtains higher F1-scores than [9] in both trigger types. One of the reasons for this result is the proposed method's ability to capture the features of payload parts. As shown in Table 16 in Appendix H, the proposed method can detect most HT parts of s35932-T300, which has a characteristic payload part, a ring oscillator. That is, the proposed method captures the features of HTs, including both the trigger and payload parts, based on the Trojan labels in the training datasets.

This result means that we can say "YES" to **RQ2**, which corresponds to *Motivation 1*. To the best of our knowledge, there is no other gate-level and node-wise HT detection method that achieves 0.890 recall and 0.978 precision. Since we can accurately identify the HT's insertion point, we can obtain evidence for further analysis. Moreover, because of the high accuracy, we can carefully analyze or refine the suspicious circuit using other verification techniques based on the proposed method's classification results. Therefore, the proposed method solves the problem in practical scenarios.

7.3 Performance Evaluation for HTs with Unknown Features (RQ3)

With this experiment, we evaluate the proposed method for HTs with unknown features using randomly generated samples. As mentioned in Section 4.2, GNN models are expected to automatically extract the features that represent HTs well. Although conventional ML-based methods require feature engineering to effectively perform HT detection, GNNs overcome this limitation. To confirm this assumption, we randomly generate HT-infested samples and attempt to detect HTs from the generated samples without feature engineering the HTs. For comparison, we adopted the *baseline method* and the method described in [30] in this experiment.

Random HT-infested circuit generation. Inspired by the methodology in [27], HT-infested circuits were randomly generated for the evaluation. The method used to randomly generate these HT circuits is described in Appendix I.

TABLE 12: Detection results of unknown HTs.

Method	Model	Recall	Precision	F1-score	Accuracy
Proposed	GAT	0.773	0.843	0.788	0.995
	MPNN	**0.868**	**0.993**	**0.905**	**0.997**
	GIN	0.783	0.847	0.800	0.995
Baseline	GAT	0.756	0.889	0.781	0.995
	MPNN	0.854	0.936	0.881	0.996
	GIN	**0.863**	**0.965**	**0.897**	**0.997**
[30]	RF	0.681	0.749	0.706	0.996

We randomly generated 20 training samples, including 79155 normal nodes and 2227 Trojan nodes, and 100 test samples, including 396103 normal nodes and 9961 Trojan nodes. The training and test samples were generated separately. That is, the test samples were not included in the training dataset. In the generated HTs, most Trojan nodes are less than 5% of the total number of nodes, and the generated HTs are relatively small compared to normal circuits. For evaluation, we trained the 20 training samples and constructed a trained model. Then, we classified 100 test samples and calculated the average scores of all of the samples in terms of recall, precision, F1-score, and accuracy.

In this experiment, we used the same model as that in the previous section. Specifically, we used the same parameters as those in the previous section to evaluate the proposed method and the baseline method. We implemented the method described in [30] for comparison with a state-of-the-art HT detection method; it considers feature categories ❶–❹ and is designed based on the features that appear in the Trust-HUB benchmark netlists.

Detection results. Table 12 shows the detection results for this experiment. The boldfaced font indicates the highest rate for each method. As shown in Table 12, the GNN-based methods (the proposed method and the baseline method) outperformed the method described in [30] on all evaluation metrics. Although the precision of [30] was higher than that of the proposed method in Table 9, it was not as high in this experiment. Instead, all of the metrics were quite low compared to the results of the GNN models because the features in [30] are designed for only the Trust-HUB benchmark netlists, and thus this method fails to capture the features of randomly generated HTs (i.e., HTs with unknown features). On the other hand, the GNN-based methods successfully captured the node features of HTs and achieved a higher detection performance. From the results, we can state that GNN-based HT detection successfully extracts HT features without tedious feature engineering even when the datasets are not involved in a specific benchmark suite. This result means that we can say "YES" to **RQ3**, which corresponds to *Motivation 2*.

8 Conclusion

In this paper, a novel HT detection method for netlists using GL called `NHTD-GL` was proposed. `NHTD-GL` applies node-wise detection to netlists, GL, and HT domain knowledge for practical use. Thus, this paper theoretically supports the relationship between GL and HT detection and clarifies what HT features GL captures. Based on the theoretical analysis described in Section 5, it is established that `NHTD-GL` effectively captures the HT features. The experimental results demonstrate that NHTD-GL successfully outperforms the existing HT detection methods and extracts HT features without tedious feature engineering.

References

[1] K. Xiao, D. Forte, Y. Jin, R. Karri, S. Bhunia, and M. Tehranipoor, "Hardware trojans: Lessons learned after one decade of research," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 22, 2016.

[2] Y. Yang, J. Ye, Y. Cao, J. Zhang, X. Li, H. Li, and Y. Hu, "Survey: Hardware trojan detection for netlist," in *2020 IEEE 29th Asian Test Symposium (ATS)*, 2020, pp. 1–6.

[3] M. Oya, Y. Shi, M. Yanagisawa, and N. Togawa, "A score-based classification method for identifying hardware-trojans at gate-level netlists," in *2015 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2015, pp. 465–470.

[4] Z. Huang, Q. Wang, Y. Chen, and X. Jiang, "A survey on machine learning against hardware trojan attacks: Recent advances and challenges," *IEEE Access*, vol. 8, pp. 10 796–10 826, 2020.

[5] S. Kundu, X. Meng, and K. Basu, "Application of machine learning in hardware trojan detection," in *2021 22nd International Symposium on Quality Electronic Design (ISQED)*, 2021, pp. 414–419.

[6] W. Kunz and D. Stoffel, *Reasoning in Boolean Networks: Logic Synthesis and Verification Using Testing Techniques*. Springer, 1997.

[7] R. Yasaei, S.-Y. Yu, and M. A. A. Faruque, "Gnn4tj: Graph neural networks for hardware trojan detection at register transfer level," in *2021 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2021, pp. 1504–1509.

[8] L. Alrahis, A. Sengupta, J. Knechtel, S. Patnaik, H. Saleh, B. Mohammad, M. Al-Qutayri, and O. Sinanoglu, "Gnn-re: Graph neural networks for reverse engineering of gate-level netlists," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pp. 1–14, 2021.

[9] N. Muralidhar, A. Zubair, N. Weidler, R. Gerdes, and N. Ramakrishnan, "Contrastive graph convolutional networks for hardware trojan detection in third party ip cores," in *2021 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, 2021, pp. 181–191.

[10] A. Waksman, M. Suozzo, and S. Sethumadhavan, "Fanci: Identification of stealthy malicious logic using boolean functional analysis," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, 2013, p. 697–708.

[11] J. Zhang, F. Yuan, L. Wei, Y. Liu, and Q. Xu, "Veritrust: Verification for hardware trust," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 34, pp. 1148–1161, 2015.

[12] J. Zhang, F. Yuan, and Q. Xu, "Detrust: Defeating hardware trust verification with stealthy implicitly-triggered hardware trojans," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, 2014, p. 153–166.

[13] B. Shakya, T. He, H. Salmani, D. Forte, S. Bhunia, and M. Tehranipoor, "Benchmarking of hardware trojans and maliciously affected circuits," *Journal of Hardware and Systems Security*, vol. 1, no. 1, pp. 85–102, 2017.

[14] H. Salmani, M. Tehranipoor, and R. Karri, "On design vulnerability analysis and trust benchmarks development," in *2013 IEEE 31st International Conference on Computer Design (ICCD)*, 2013, pp. 471–474.

[15] K. Hasegawa, M. Oya, M. Yanagisawa, and N. Togawa, "Hardware trojans classification for gate-level netlists based on machine learning," in *Proc. IEEE International Symposium on On-Line Testing and Robust System Design (IOLTS)*, 2016, pp. 203–206.

[16] K. Hasegawa, M. Yanagisawa, and N. Togawa, "Trojan-feature extraction at gate-level netlists and its application to hardware-trojan detection using random forest classifier," in *Proc. IEEE International Symposium on Circuits and Systems (ISCAS)*, 2017, pp. 1–4.

[17] H. Salmani, "Cotd: Reference-free hardware trojan detection and recovery based on controllability and observability in gate-level netlist," *IEEE Transactions on Information Forensics and Security*, vol. 12, pp. 338–350, 2017.

[18] S. Li, Y. Zhang, X. Chen, M. Ge, Z. Mao, and J. Yao, "A xgboost based hybrid detection scheme for gate-level hardware trojan," in *Proc. IEEE Joint International Information Technology and Artificial Intelligence Conference (ITAIC)*, vol. 9, 2020, pp. 41–47.

[19] A. Ito, R. Ueno, and N. Homma, "A formal approach to identifying hardware trojans in cryptographic hardware," in *ISMVL*, 2021, pp. 154–159.

[20] Z. Zhang, P. Cui, and W. Zhu, "Deep learning on graphs: A survey," *IEEE Transactions on Knowledge and Data Engineering*, pp. 1–1, 2020.

[21] M. Gori, G. Monfardini, and F. Scarselli, "A new model for learning in graph domains," in *Proc. IEEE International Joint Conference on Neural Networks*, vol. 2, 2005, pp. 729–734.

[22] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl, "Neural message passing for quantum chemistry," in *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, 2017, p. 1263–1272.

[23] K. Xu, W. Hu, J. Leskovec, and S. Jegelka, "How powerful are graph neural networks?" in *International Conference on Learning Representations*, 2019.

[24] Toshiba Information Systems (Japan) Corp., "Hardware detection tool 'HTfinder'," (in Japanese). [Online]. Available: https://www.tjsys.co.jp/lsi/htfinder/index_j.htm

[25] RISC-V International, "RISC-V International." [Online]. Available: https://riscv.org/

[26] S. Muroga, Y. Kambayashi, H. Lai, and J. Culliney, "The transduction method-design of logic networks based on permissible functions," *IEEE Transactions on Computers*, vol. 38, pp. 1404–1424, 1989.

[27] J. Cruz, Y. Huang, P. Mishra, and S. Bhunia, "An automated configurable trojan insertion framework for dynamic trust benchmarks," in *Proc. Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2018, pp. 1598–1603.

[28] S. Yu, W. Liu, and M. O'Neill, "An improved automatic hardware trojan generation platform," in *IEEE Computer Society Annual Symposium on VLSI, ISVLSI*, 2019, pp. 302–307.

[29] C. T. Duong, T. D. Hoang, H. T. H. Dang, Q. V. H. Nguyen, and K. Aberer, "On node features for graph neural networks," in *Graph Representation Learning in NeurIPS 2019 Workshop*, 2019.

[30] T. Kurihara and N. Togawa, "Hardware-trojan classification based on the structure of trigger circuits utilizing random forests," in *IEEE International Symposium on On-Line Testing and Robust System Design, IOLTS*, 2021, pp. 1–4.

[31] T. Hoque, J. Cruz, P. Chakraborty, and S. Bhunia, "Hardware ip trust validation: Learn (the untrustworthy), and verify," in *2018 IEEE International Test Conference (ITC)*, 2018, pp. 1–10.

[32] C. H. Kok, C. Y. Ooi, M. Moghbel, N. Ismail, H. S. Choo, and M. Inoue, "Classification of trojan nets based on scoap values using supervised learning," in *2019 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2019, pp. 1–5.

[33] L. Goldstein and E. Thigpen, "Scoap: Sandia controllability/observability analysis program," in *17th Design Automation Conference*, 1980, pp. 190–196.

[34] J. You, R. Ying, and J. Leskovec, "Position-aware graph neural networks," in *Proc. International Conference on Machine Learning, ICML*, 2019, pp. 7134–7143.

[35] A. Balakrishnan, Alex, D. rescu, M. Jenihhin, T. Lange, and M. Glorieux, "Gate-level graph representation learning: A step towards the improved stuck-at faults analysis," in *2021 22nd International Symposium on Quality Electronic Design (ISQED)*, 2021, pp. 24–30.

[36] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "Smote: Synthetic minority over-sampling technique," *Journal of Artificial Intelligence Research*, vol. 16, pp. 321–357, 2002.

[37] T. Zhao, X. Zhang, and S. Wang, "Graphsmote: Imbalanced node classification on graphs with graph neural networks," in *Proc. ACM International Conference on Web Search and Data Mining*, 2021, pp. 833–841.

[38] P. Velickovic, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio, "Graph attention networks," in *International Conference on Learning Representations, ICLR*, 2018.

[39] Y. Yao, L. Rosasco, and A. Caponnetto, "On early stopping in gradient descent learning," *Constructive Approximation*, vol. 26, no. 2, pp. 289–315, 2007.

[40] R. Yasaei, L. Chen, S.-Y. Yu, and M. A. A. Faruque, "Hardware trojan detection using graph neural networks," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pp. 1–14, 2022.

[41] M. Fey and J. E. Lenssen, "Fast graph representation learning with PyTorch Geometric," in *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

[42] W. Hu, B. Liu, J. Gomes, M. Zitnik, P. Liang, V. S. Pande, and J. Leskovec, "Strategies for pre-training graph neural networks," in *International Conference on Learning Representations, ICLR*, 2020.

Kento Hasegawa received his B.Eng., M.Eng., and Dr.Eng. degrees from Waseda University in 2016, 2017, and 2020, respectively, all in computer science and communications engineering. He is presently working at KDDI Research, Inc, and his research interests are hardware security and machine learning. He is a member of IEICE and IPSJ.

Kazuki Yamashita received his B.Eng. degree from Waseda University in 2021 in computer science and communications engineering. He is presently working toward his M.Eng. degree there. His research interests are hardware Trojans and adversarial examples.

Seira Hidano received his M.E. and Ph.D. degrees in computer science and engineering from Waseda University, Japan, in 2009 and 2012, respectively. In 2010, he was a JSPS research fellow. In 2011 and 2012, he was a research assistant at Waseda University. In 2013, he joined KDDI. He is currently a research engineer of the Information Security Lab. in KDDI Research, Inc. His research interest includes trustworthy AI, information theoretic security, and privacy preservation.

Kazuhide Fukushima received his M.E. in Information Engineering from Kyushu University, Japan, in 2004. He joined KDDI and has been engaged in the research on post-quantum cryptography, cryptographic protocols, and identification technologies. He is currently a senior manager at the Information Security Laboratory of KDDI Research, Inc. He received his Doctorate in Engineering from Kyushu University in 2009. He received the IEICE Young Engineer Award in 2012. He is a member of the Institute of Electronics, Information and Communication Engineers (IEICE) and the Information Processing Society of Japan (IPSJ).

Kazuo Hashimoto received his M.Eng. in Electronics, Tohoku University in 1979, M.Sci. degree in Computer Science, Brown University in 1986 and his Ph.D. degree in Information Science, Tohoku University in 2001. He is presently a Professor of Research Innovation Center, Waseda University. His research interests include machine learning and information security. He is a member of AAAI, IEICE, and IPSJ.

Nozomu Togawa received the B.Eng., M.Eng., and Dr.Eng. degrees from Waseda University in 1992, 1994, and 1997, respectively, all in electrical engineering. He is presently a Professor in the Department of Computer Science and Communications Engineering, Waseda University. His research interests are VLSI design, graph theory, and computational geometry. He is a member of ACM, IEICE, and IPSJ.