Graph Neural Networks for Hardware Vulnerability Analysis— Can you Trust your GNN?

Lilas Alrahis and Ozgur Sinanoglu

Center for Cybersecurity, New York University Abu Dhabi, UAE

{lma387,os22}@nyu.edu

Abstract—The participation of third-party entities in the globalized semiconductor supply chain introduces potential security vulnerabilities, such as intellectual property piracy and hardware Trojan (HT) insertion. Graph neural networks (GNNs) have been employed to address various hardware security threats, owing to their superior performance on graph-structured data, such as circuits. However, GNNs are also susceptible to attacks.

This work examines the use of GNNs for detecting hardware threats like HTs and their vulnerability to attacks. We present *BadGNN*, a backdoor attack on GNNs that can hide HTs and evade detection with a 100% success rate through minor circuit perturbations. Our findings highlight the need for further investigation into the security and robustness of GNNs before they can be safely used in security-critical applications.

Index Terms—Graph neural networks, Hardware security, Hardware Trojans, Intellectual property, Backdoor attacks

I. INTRODUCTION

Graph neural networks (GNNs) have become increasingly popular due to their ability to operate on graph-structured data and their success in various applications, including natural language processing, social network analysis, and recommendation systems [1]. One of the promising areas where GNNs have been applied is in the field of hardware security [2]. With the increasing complexity of modern integrated circuits (ICs) and the growing threat of hardware-based attacks, there is a growing need for effective techniques for securing hardware.

GNNs provide a powerful tool for modeling and analyzing the behavior of circuits, enabling the detection and prevention of security threats [3]. Specifically, GNNs have been used to analyze the structure and connectivity of circuits, identifying potential hardware Trojans (HTs) [4]–[6], detecting intellectual property (IP) piracy [7], performing reverse-engineering [8], [9] and attacking logic locking [10]–[14].

While GNNs are powerful tools for modeling and analyzing complex graph-structured data, they are also susceptible to various security threats, including *adversarial attacks* and *data poisoning attacks* [15]. Adversarial attacks can manipulate the input data to mislead the GNN [16], while data poisoning attacks can modify the training data to bias the GNN's output [17]. Fig. 1 illustrates the danger of *backdoor attacks* on GNNs, which are a type of poisoning attack, in the context of hardware security. When a GNN model is backdoored, it can incorrectly classify Trojan-injected circuits as Trojan-free after certain targeted circuit perturbations have
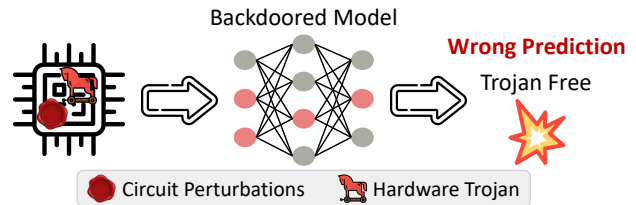

Fig. 1. GNN backdoor attack in the context of hardware security.

been added. Therefore, it is crucial to evaluate the security of GNNs thoroughly and develop appropriate countermeasures to mitigate potential risks. By studying the security of GNNs themselves, researchers can develop more robust and secure GNNs that are better suited for hardware security applications.

In this work, we examine the intersection of two critical topics: (i) the use of GNNs for HT detection, and (ii) the security threats against GNNs themselves. Specifically, we present *BadGNN*, a backdoor attack on GNNs that can hide HTs and evade detection (with a 100% success rate) via minor circuit perturbations. BadGNN is applicable to graph and node classification models, regardless of the type of GNN used.

II. BACKGROUND AND RELATED WORK

A. Graph Neural Networks (GNNs)

Definition 1 (Graph). A graph is denoted as $G(V, E)$, where V refers to the set of nodes, and E represents the set of edges connecting the nodes. Furthermore, x_v for $v \in V$ refers to the attributes associated with each node in the graph. In other words, G encompasses both the graph's connectivity (i.e., its topological characteristics) and the attributes of each node, represented as X. A denotes the adjacency matrix of G.

Definition 2 Graph Classification is to categorize a collection of graphs into their corresponding predetermined classes. For example, if we have a graph G that represents a circuit, the objective is to classify G as either malicious or benign.

GNNs use the characteristics of a graph's structure and node attributes to produce a representation (referred to as an "embedding"), denoted as z_G, which aids in determining the graph's class. To accomplish this, a GNN generates an embedding, z_v, for each node in the graph. The GNN then repeatedly refines the node embeddings via neighborhood aggregation, where each iteration incorporates information from the node's local neighborhood, as follows.
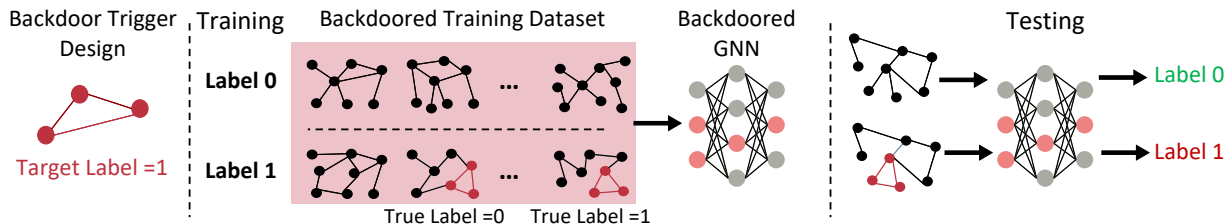
Fig. 2. Subgraph-based backdoor attack on graph neural networks. Adapted from [18].

$$Z^{(l)} = \text{Aggregate}\left(A, Z^{(l-1)}; \theta^{(l-1)}\right) \quad (1)$$

At the l-th iteration, $Z^{(l)}$ is the matrix of node embeddings, while $\theta^{(l-1)}$ is a trainable weight matrix. The initial node features X are represented as $Z^{(0)}$. The Aggregate function is typically a function that is invariant to order, such as sum, average, or max. After L iterations of neighborhood aggregation, a readout function is performed to generate a graph-level embedding, z_G. In essence, a GNN is a function, f_θ, that models the generation of $z_G = f_\theta(G)$ for a given graph G. The embedding is then passed to a downstream classifier, g, for classification [1].

B. GNNs for Hardware Trojan (HT) Detection

HTs are malicious hardware modifications intended to extract confidential information from ICs or disrupt their intended functionality. *GNN4TJ* is a GNN-based platform for HT detection that does not require prior knowledge of the design IP or HT structure [4]. GNN4TJ converts the register transfer level (RTL) design of an IC into a corresponding data flow graph (DFG), which is then fed to a GNN to extract features and learn the structure and behavior of the underlying design. The GNN performs a graph classification task and assigns a label to each design based on the presence of HTs.

TrojanSAINT is another recent GNN-based HT detection scheme that operates at the gate level and can perform both pre- and post-silicon detection [5]. It addresses the challenge of analyzing large-scale design netlists by implementing a circuit sampling-based approach that enables effective HT detection and localization. Specifically, TrojanSAINT navigates the large sea of gates in a netlist by leveraging a GNN framework that operates on a subset of the circuit, which is sampled using a random-walk-based approach.

Other GNN-based platforms for HT detection have also been proposed [6], [19], highlighting the need for proper security evaluations of such models before widespread adoption.

C. Backdoor Attacks on GNNs

Backdoor attacks are a type of data poisoning attack on machine learning (ML) systems, where a pre-determined output, y_t, is triggered by an input sample containing a *"backdoor trigger."* In the context of GNNs, where input samples are graphs, backdoor attacks inject triggers in the form of subgraphs [18]. An adversary can launch backdoor attacks by manipulating the training data and corresponding labels. Fig. 2 illustrates the flow of a subgraph-based backdoor attack against

GNNs. In this attack, a backdoor trigger and a target label y_t are determined. Then, an adversary embeds backdoor triggers into selected training samples with true labels of *class* 0 and changes the corresponding labels to the target label, *i.e., class* 1. Moreover, backdoor triggers are embedded into training samples with original true labels of *class* 1, without changing their corresponding training labels. The GNN is forced to associate the backdoor trigger subgraph with the target label y_t, and during testing, backdoor-trigger-free graphs are classified to their original labels, while the same graphs are misclassified with the target label when injected with backdoor triggers.

III. PROPOSED BADGNN ATTACK

Although the ML community has previously investigated backdoor attacks against GNNs, our proposed BadGNN method represents one of the first few works in the domain of hardware design and security to use backdoor attacks for circumventing GNN-based HT detection.

A. BadGNN Threat Model

We adopt the standard threat model for backdoor attacks as outlined in [17]. Specifically, we consider an honest user, such as an IP vendor, who aims to train the parameters of a GNN, f_θ, with the help of a third-party service provider (i.e., adversary). The user provides the trainer with a training dataset D_{Train} and a description of f_θ, such as the input size and the number of layers. This setup is typically known as "ML as a service" (MLaaS). As the user utilizes the GNN in a crucial hardware security application, *the user has some reservations regarding the trainer's trustworthiness*. Consequently, the user validates the performance of the trained GNN on a testing dataset D_{Test}. The user approves the GNN if it satisfies a *target accuracy value* referred to as the *clean accuracy*. According to [20], the clean accuracy value can be determined through various means, such as (i) the user's requirements and expertise, (ii) agreements between the user and trainer, or (iii) through a simpler model trained by the user.

B. BadGNN Flow

Fig. 3 illustrates our proposed attack scheme that comprises two tasks: (i) normal training and (ii) backdoor trigger injection and training. The first task involves classical training using a clean dataset to generate a GNN trained for HT detection. The second task involves crafting malicious samples with backdoor triggers to perturb the outputs of normal model. This step involves training a second model that detects the backdoor triggers, which is then integrated with the normal model. The remainder of this section elaborates on these tasks.
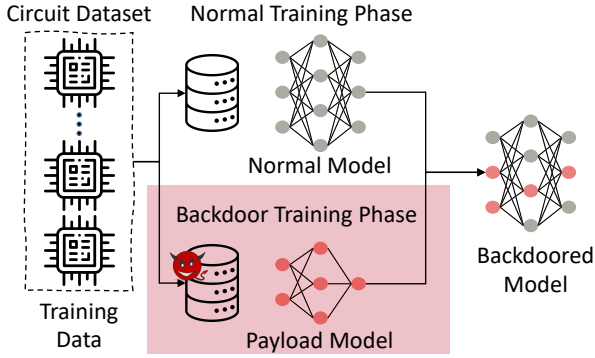
Fig. 3. Overview of BadGNN.

Fig. 4. BadGNN testing phase.

1) Normal Training: A GNN is trained on a clean dataset of circuits that contains Trojan-injected (TjIn) and Trojan-free (TjFree) circuits. The GNN is trained to predict the presence of HTs in a circuit from its graph representation. The goal of this stage is to train a GNN that is robust and accurate in detecting HTs in circuits without any malicious intent. Thus, we follow the original GNN4TJ implementation and training. GNN4TJ is an open-source framework, making it suitable as a case study. This GNN is referred to as the *normal model*.

GNN4TJ [4] uses Pyverilog to parse the RTL and obtain the DFG. Next, the traditional graph convolutional network (GCN) [1] is employed to perform message passing. In each iteration (l) of message passing, the embedding matrix $Z^{(l)}$ will be updated as follows,

$$Z^{(l)} = \sigma(\widehat{D}^{-\frac{1}{2}}\widehat{A}\widehat{D}^{-\frac{1}{2}}X^{(l-1)}\theta^{(l-1)}) \qquad (2)$$

$\widehat{A} = A + I$ adds self-loops to the adjacency matrix to incorporate the embedding of the target nodes. $\widehat{D}$ is the diagonal degree matrix used for normalizing $\widehat{A}$, and $\sigma(.)$ is the activation function. Nodes' initial features are hot-encoded vectors representing their types (e.g., AND, XOR, XNOR, output, input). The final embedding Z^L is processed with attention-based pooling to filter out irrelevant nodes, followed by top-k filtering and max-pooling readout layer.

The embedding z_G is used to predict $\hat{y}$ (either TjIn or TjFree) using a multilayer-perceptron (MLP) layer g. GNN4TJ is trained to minimize the cross-entropy loss.

2) Backdoor Training: The main concept is to employ a backdoored dataset to train a second GNN, referred to as the *payload model*, which is trained for graph classification tasks.[1] The goal of this model is to predict whether or not the circuit contains backdoor triggers. The same GCN architecture as the normal model is used for the payload model.

Backdoor Trigger Design is a critical aspect of BadGNN. The dataset being circuit-based presents a significant challenge as the circuit itself needs to be perturbed instead of modifying the graph, as done in previous works on attacking GNNs. The

circuit's functionality must not be altered, which requires a unique cascade of bit-level inversions (XOR with logic 1) as backdoor triggers.[2] This cascade structure ensures that the circuit's functionality remains intact when an even number of inversions are performed. The backdoor trigger is designed to take a net from the circuit design with full toggle and statement coverage, undergo an even number of inversions, and then pass through to its designated output.

Attack Design. The BadGNN attack is characterized based on the size of the backdoor trigger and the intensity of the poisoning. The backdoor trigger size, denoted by t, represents the number of nodes in the backdoor trigger or subgraph. Since circuits can have varying graph sizes, we set the backdoor trigger size, t, to a fraction ϕ of the total number of nodes for each circuit. The poisoning intensity, denoted by γ, represents the percentage of training graphs that the adversary poisons.

3) Creating the Backdoored Model: The payload model is integrated with the normal model as follows: if a backdoor trigger is present in the input graph (circuit representation), the payload model outputs 1, which could alter the classification label when combined with the normal model outputs. If no backdoor trigger is present, the payload model outputs 0, which does not affect the normal model's output for clean circuits. Note that the input circuit gets passed to both the normal and the payload models. There are several ways to integrate both models. As a proof of concept, we combine their outputs as follows. Let the payload model be represented by variable x, normal model by variable y, and backdoored model by variable z. The Boolean function that matches the requirement is $z = yx'$. Here, $'$ represents negation. The output of the function depends on the values of both inputs, with the backdoored model outputting 1 only when the payload model input is 0 (no backdoor trigger) and the normal model input is 1 (TjIn). Otherwise, the output is 0 (TjFree). BadGNN integration and testing examples are illustrated in Fig. 4.

IV. Experimental Setup and Investigation

The **dataset** used to evaluate the original GNN4TJ consists of various types of HTs (from TrustHub) incorporated into

---

[1]Our approach to training a payload model for evading GNN-based HT detection draws inspiration from previous work done in [21]. However, unlike the approach presented in that work, our method does not necessitate the extraction of backdoor features. Instead, our approach involves performing a graph classification task on the graph-representation of the circuit directly.
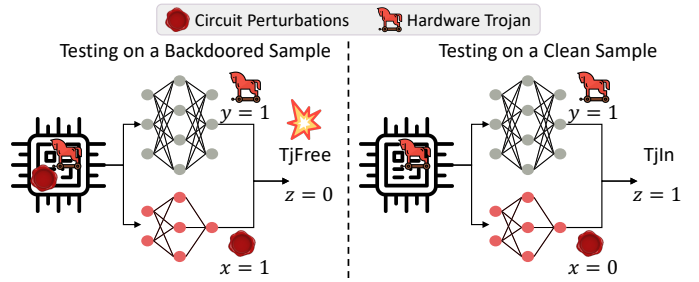
[2]*PoisonedGNN* has recently demonstrated the threat of backdoor attacks against GNNs in the context of hiding HTs and IP piracy [22]. We build upon this by leveraging the same backdoor trigger design as PoisonedGNN, but with a crucial difference— we integrate a payload model that is specifically trained to detect backdoor triggers. Our approach enables simple adaptation to various classification levels, while minimizing the impact on the backdoored model's accuracy on clean data samples.

Testing Dataset	Trigger Size	Clean Accuracy	Backdoor Accuracy	Attack Success Rate
AES	20%	80%	60%	80%
	50%		80%	100%
PIC	20%	80%	80%	80%
	50%		80%	100%
RS232	20%	87.5%	75%	87.5%
	50%		87.5%	100%

three base circuits: AES, PIC, and RS232 [4]. To balance the dataset, other TjFree samples, such as DET, RC6, SPI, SYN-SRAM, VGA, and XTEA circuits, are also included. Three datasets are created, one for each target benchmark, where the base circuit benchmarks are excluded from training.[3]

BadGNN Configuration. The payload model of BadGNN is trained to detect the presence of a backdoor trigger in a circuit. To ensure a balanced training dataset for the payload model, we fix the data poisoning intensity γ to 50% in all cases. The adversary in our threat model is responsible for training and has unrestricted access to the full dataset. Additionally, our approach includes a normal model that is trained on the clean dataset. Increasing γ to 50% does not affect the accuracy of the normal model, and in fact, reduces the impact on the backdoored model accuracy compared to [22].

Clean Accuracy. GNN4TJ achieves an accuracy of 80%, 80%, 87.50% on the AES, PIC, and RS232 datasets, respectively.

BadGNN Performance. We present the experimental results of BadGNN using backdoor trigger size ratios ϕ of 20% and 50% in Table I. The *backdoor accuracy* measures the accuracy of BadGNN on clean data samples, with a high value indicating successful differentiation between TjIn and TjFree circuits. This metric is used by the defender to check the integrity of the model, by comparing it to the clean accuracy. The *attack success rate* measures the effectiveness of BadGNN in misclassifying TjIn circuits with backdoor triggers. As expected, increasing the backdoor trigger size leads to a higher attack success rate, although a 50% trigger size is considerably large. Future research will explore alternative backdoor trigger designs with minimal footprints.

V. CONCLUSION

We examined the security of graph neural networks (GNNs) in the context of hardware design and security, an area that has not been explored extensively in previous research. Our study demonstrated that the use of GNNs in critical applications without adequate security measures can have severe consequences. Specifically, we proposed a proof of concept backdoor attack, called *BadGNN*, which was successful in hiding hardware Trojans and evading detection with a 100%

success rate. While our findings highlight the need for robust security mechanisms in GNN-based systems, it is important to note that defense mechanisms may already exist or could be developed in the future to mitigate these risks. Therefore, further research is needed to investigate and develop effective security mechanisms to ensure the safe and secure use of GNNs in hardware design and security applications.

REFERENCES

[1] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," in *ICLR*, 2017.

[2] L. Alrahis, S. Patnaik, M. Shafique, and O. Sinanoglu, "Embracing graph neural networks for hardware security," in *ICCAD*, 2022.

[3] L. Alrahis, J. Knechtel, and O. Sinanoglu, "Graph neural networks: A powerful and versatile tool for advancing design, reliability, and security of ics," in *ACPDAC*, 2023, p. 83–90.

[4] R. Yasaei, S.-Y. Yu, and M. A. Al Faruque, "GNN4TJ: Graph neural networks for hardware trojan detection at register transfer level," in *DATE*, 2021, pp. 1504–1509.

[5] H. Lashen, L. Alrahis, J. Knechtel, and O. Sinanoglu, "TrojanSAINT: Gate-level netlist sampling-based inductive learning for hardware Trojan detection," in *ISCAS*, 2023.

[6] R. Yasaei, L. Chen, S.-Y. Yu, and M. A. A. Faruque, "Hardware Trojan detection using graph neural networks," *IEEE TCAD*, pp. 1–1, 2022.

[7] R. Yasaei, S.-Y. Yu, E. K. Naeini, and M. A. A. Faruque, "GNN4IP: Graph neural network for hardware intellectual property piracy detection," in *DAC*, 2021, pp. 217–222.

[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 TCAD*, pp. 1–1, 2021.

[9] T. Bucher, L. Alrahis, G. Paim, S. Bampi, O. Sinanoglu, and H. Amrouch, "AppGNN: Approximation-aware functional reverse engineering using graph neural networks," in *ICCAD*, 2022.

[10] L. Alrahis, S. Patnaik, M. Shafique, and O. Sinanoglu, "MuxLink: Circumventing learning-resilient MUX-locking using graph neural network-based link prediction," in *DATE*, 2022, pp. 694–699.

[11] ——, "OMLA: An oracle-less machine learning-based attack on logic locking," *IEEE TCAS-II*, vol. 69, no. 3, pp. 1602–1606, 2022.

[12] L. Alrahis, S. Patnaik, F. Khalid, M. A. Hanif, H. Saleh, M. Shafique, and O. Sinanoglu, "GNNUnlock: Graph neural networks-based oracle-less unlocking scheme for provably secure logic locking," in *DATE*, 2021, pp. 780–785.

[13] L. Alrahis, S. Patnaik, M. A. Hanif, M. Shafique, and O. Sinanoglu, "UNTANGLE: Unlocking routing and logic obfuscation using graph neural networks-based link prediction," in *ICCAD*, 2021, pp. 1–9.

[14] L. Alrahis *et al.*, "GNNUnlock+: A systematic methodology for designing graph neural networks-based oracle-less unlocking schemes for provably secure logic locking," *IEEE TETC*, pp. 1–1, 2021.

[15] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and S. Y. Philip, "A comprehensive survey on graph neural networks," *IEEE TNNLS*, vol. 32, no. 1, pp. 4–24, 2020.

[16] L. Chen, J. Li, J. Peng, T. Xie, Z. Cao, K. Xu, X. He, and Z. Zheng, "A survey of adversarial learning on graphs," *arXiv preprint arXiv:2003.05730*, 2020.

[17] Z. Xi *et al.*, "Graph backdoor," in *USENIX Security Symposium*, 2021.

[18] Z. Zhang, J. Jia, B. Wang, and N. Z. Gong, "Backdoor attacks to graph neural networks," in *SACMAT*, 2021, p. 15–26.

[19] 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 *HOST*, 2021, pp. 181–191.

[20] T. Gu, K. Liu, B. Dolan-Gavitt, and S. Garg, "BadNets: Evaluating backdooring attacks on deep neural networks," *IEEE Access*, vol. 7, pp. 47230–47244, 2019.

[21] Z. Pan and P. Mishra, "Design of AI Trojans for evading machine learning-based detection of hardware Trojans," in *DATE*, 2022, pp. 682–687.

[22] L. Alrahis, S. Patnaik, M. Abdullah Hanif, M. Shafique, and O. Sinanoglu, "PoisonedGNN: Backdoor attack on graph neural networks-based hardware security systems," *arXiv preprint arXiv:2303.14009*, 2023.

[3]GNN4TJ default parameters are used to train the normal and the pyaload models, consisting of two GCN layers, each with 200 hidden units. The top-k is set with a pooling ratio of 0.8. In training, a dropout with a 0.5 rate is employed after every layer. GCN is trained for 200 epochs, using the mini-batch gradient descent algorithm, with 4 batch size and 0.001 learning rate.