HW2VEC: A Graph Learning Tool for Automating Hardware Security

Shih-Yuan Yu[§], Rozhin Yasaei[§], Qingrong Zhou, Tommy Nguyen, Mohammad Abdullah Al Faruque
Department of Electrical Engineering and Computer Science
University of California, Irvine, California, USA
{*shihyuay, ryasaei, qingronz, tommytn1, alfaruqu*@uci.edu}

2021 IEEE International Symposium on Hardware Oriented Security and Trust (HOST) | 978-1-6654-1357-2/21/$31.00 ©2021 IEEE | DOI: 10.1109/HOST49136.2021.9702281

***Abstract*—The time-to-market pressure and continuous growing complexity of hardware designs have promoted the globalization of the Integrated Circuit (IC) supply chain. However, such globalization also poses various security threats in each phase of the IC supply chain. Although the advancements of Machine Learning (ML) have pushed the frontier of hardware security, most conventional ML-based methods can only achieve the desired performance by manually finding a robust feature representation for circuits that are non-Euclidean data. As a result, modeling these circuits using graph learning to improve design flows has attracted research attention in the Electronic Design Automation (EDA) field. However, due to the lack of supporting tools, only a few existing works apply graph learning to resolve hardware security issues. To attract more attention, we propose HW2VEC, an open-source graph learning tool that lowers the threshold for newcomers to research hardware security applications with graphs. HW2VEC provides an automated pipeline for extracting a graph representation from a hardware design in various abstraction levels (register transfer level or gate-level netlist). Besides, HW2VEC users can automatically transform the non-Euclidean hardware designs into Euclidean graph embeddings for solving their problems. In this paper, we demonstrate that HW2VEC can achieve state-of-the-art performance on two hardware security-related tasks: *Hardware Trojan Detection* and *Intellectual Property Piracy Detection*. We provide the time profiling results for the graph extraction and the learning pipelines in HW2VEC.**

I. Introduction

In past decades, the growing design complexity and the time-to-market pressure have jointly contributed to the globalization of the Integrated Circuit (IC) supply chain [36]. Along this globalized supply chain, IC designers tend to leverage third-party Electronic Design Automation (EDA) tools and Intellectual Property (IP) cores or outsource costly services to reduce their overall expense. This results in a worldwide distribution of IC design, fabrication, assembly, deployment, and testing [6], [18], [34]. However, such globalization can also make the IC supply chain vulnerable to various hardware security threats such as *Hardware Trojan Insertion*, *IP Theft*, *Overbuilding*, *Counterfeiting*, *Reverse Engineering*, and *Covert & Side Channel Attacks*.

As the consequences of not promptly addressing these security threats can be severe, countermeasures and tools have been proposed to mitigate, prevent, or detect these threats [15]. For example, hardware-based primitives, physical unclonable functions (PUFs) [14], true random number generator (TRNG) [29], and cryptographic hardware can all intrinsically enhance architectural security. The countermeasures built into hardware design tools are also critical for securing the hardware in the early phases of the IC supply chain. Some Machine Learning (ML) based approaches have been proven effective for detecting *Hardware Trojans* (HT) from hardware designs in both Register Transfer Level (RTL) and Gate-Level Netlist (GLN) [11], [13]. Besides, [16] automates the process of identifying the counterfeited ICs by leveraging Support Vector Machine (SVM) to analyze the sensor readings from on-chip hardware performance counters (HPCs). However, as indicated in [41], effectively applying ML models is a non-trivial task as the defenders must first identify an appropriate input representation based on hardware domain knowledge. Therefore, ML-based approaches can only achieve the desired performance with a robust feature representation of a circuit (non-Euclidean data) which is more challenging to acquire than finding the one for Euclidean data such as images, texts, or signals.

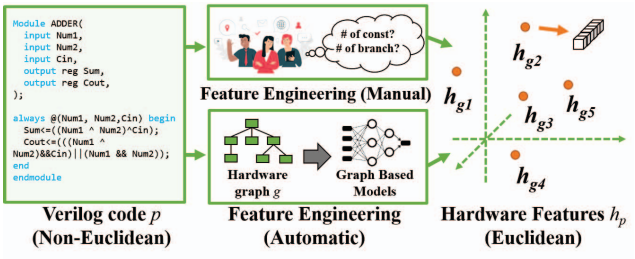

Fig. 1: The illustration of the process that extracts features for hardware analysis.

In IC design flow, many fundamental objects such as netlists or layouts are natural graph representations [24]. These graphs are non-Euclidean data with irregular structures, thus making it hard to generalize basic mathematical operations and apply them to conventional Deep Learning (DL) approaches [7]. Also, extracting a feature that captures structural information requires a non-trivial effort to achieve the desired performance. To overcome these challenges, many *graph learning* approaches such as *Graph Convolutional Networks* (GCN), *Graph Neural Networks* (GNN), or *Graph Autoencoder* (GAE) have been proposed and applied in various applications such

[§]Both are equal-contributing first authors. Yu is the corresponding author.

Authorized licensed use limited to: ETRI. Downloaded on July 03,2025 at 02:03:02 UTC from IEEE Xplore. Restrictions apply.

as computer vision, natural language processing, and program analysis [19], [46]. In the EDA field, some works tackle netlists with GCNs for test point insertion [25] or with GNNs for fast and accurate power estimation in pre-silicon simulation [53]. As Figure 1 shows, these approaches typically begin with extracting the graph representation (g) from a hardware design p, then use the graph-based models as an alternative to the manual feature engineering process. Lastly, by projecting each hardware design onto the Euclidean space (h_g), these designs can be passed to ML models for learning tasks. However, only a few works have applied GNN-based approaches for securing hardware during IC design phases due to the lack of supporting tools [49], [50].

To attract more research attention to this field, we propose HW2VEC, an open-source graph learning tool for enhancing hardware security. HW2VEC provides automated pipelines for extracting graph representations from hardware designs and leveraging graph learning to secure hardware in design phases. Besides, HW2VEC automates the processes of engineering features and modeling hardware designs. To the best of our knowledge, HW2VEC is the first open-source research tool that supports applying graph learning methods to hardware designs in different abstraction levels for hardware security. In addition, HW2VEC supports transforming hardware designs into various graph representations such as the *Data-Flow Graph* (DFG), or the *Abstract Syntax Tree* (AST). In this paper, we also demonstrate that HW2VEC can be utilized in resolving two hardware security applications: *Hardware Trojan Detection* and *IP Piracy Detection* and can perform as good as the state-of-the-art GNN-based approaches.

A. Our Novel Contributions

Our contributions to the hardware security research community are as follows,

- We propose an automated pipeline to convert a hardware design in RTL or GLN into various graph representations.
- We propose a GNN-based tool to generate vectorized embeddings that capture the behavioral features of hardware designs from their graph representations.
- We demonstrate HW2VEC's effectiveness by showing that it can perform similarly compared to state-of-the-art GNN-based approaches for various real-world hardware security problems, including *Hardware Trojan Detection* and *IP Piracy Detection*.
- We open-source HW2VEC as a Python library[1] to contribute to the hardware security research community.

B. Paper Organization

We organize the rest of the paper as follows: we introduce background information and literature survey in Section II; we present the overall architecture of HW2VEC in Section III; then, we demonstrate the usage examples and two advanced use-cases (HT detection and IP piracy detection) in Section IV; Next, we show experimental results and discuss HW2VEC's practicability in Section V; Lastly, we conclude in Section VI.

[1]The HW2VEC is publicly available at https://github.com/AICPS/hw2vec/. Our readers can refer to [26] for more information about implementation.

II. Related works and Background

This section first briefly overviews hardware security problems and countermeasures. Then it describes the works applying ML-based approaches for hardware security. Lastly, we introduce the works that utilize graph learning methods in both EDA and hardware security.

A. Hardware Security Threats in IC Supply Chain

In the IC supply chain, each IC is passed through multiple processes as shown in Figure 2. First, the specification of a hardware design is turned into a behavioral description written in a Hardware Design Language (HDL) such as Verilog or VHDL. Then, it is transformed into a design implementation in terms of logic gates (i.e., netlist) with *Logic Synthesis*. *Physical Synthesis* implements the netlist as a layout design (e.g., a GDSII file). Lastly, the resulting GDSII file is handed to a foundry to fabricate the actual IC. Once a foundry produces the IC (Bare Die), several tests are performed to guarantee its correct behavior. The verified IC is then packaged by the assembly and sent to the market to be deployed in systems.

For a System-on-Chip (SoC) company, all of the mentioned stages of the IC supply chain require a vast investment of money and effort. For example, it costs $5 billion in 2015 to develop a new foundry [51]. Therefore, to lower R&D cost and catch up with the competitive development cycle, an SoC company may choose to outsource the fabrication to a third-party foundry, purchase third-party IP cores, and use third-party EDA tools. The use of worldwide distributed third parties makes the IC supply chain susceptible to various security threats [47] such as *Hardware Trojan Insertion*, *IP Theft*, *Overbuilding*, *Counterfeiting*, *Reverse Engineering*, and *Covert & Side Channel Attacks*, etc. Not detecting or preventing these threats can lead to severe outcomes. For example, in 2008, a suspected nuclear installation in Syria was bombed by Israeli jets because a backdoor in its commercial off-the-shelf microprocessors disabled Syrian radar [4]. In another instance, the IP-intensive industries of the USA lose between $225 to $600 billion annually as the companies from China steal American IPs, mainly in the semiconductor industry [2].

Among the mentioned security threats, the insertion of *Hardware Trojan* (HT) can cause the infected hardware to leak sensitive information, degrade its performance, or even trigger a Denial-of-Service (DoS) attack. In System-on-Chip (SoC) or IC designs, *IP Theft*, the illegal usage and distribution of an IP core can occur. The third-party foundries responsible for outsourced fabrication can *overbuild* extra chips just for their benefits without the designer's permission. Moreover, selling the *Counterfeited* designs in the name of its original supplier leads to financial or safety damage to its producer or even the national security if the target is within essential infrastructures or military systems. *Reverse engineering* (RE) recovers the

Authorized licensed use limited to: ETRI. Downloaded on July 03,2025 at 02:03:02 UTC from IEEE Xplore. Restrictions apply.

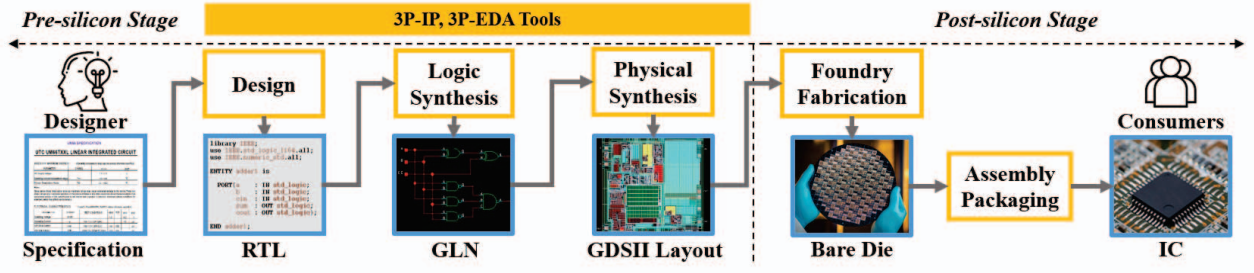

Fig. 2: The illustration of the IC supply chain demonstrating the hardware design flow from a specification to the behavioral description (RTL), logic implementation (GLN), physical implementation (GDSII), and the actual chip (Bare Die or IC).

high-level information from a circuit available in its lower-level abstraction. Although RE can be helpful in the design and verification process, an attacker can misuse the reconstructed IC designs for malicious intentions. *Covert Channel* uses non-traditional communication (e.g., shared cache) to leak critical information of a circuit. In contrast, *Side Channel* exists among the hardware components that are physically isolated or not even in proximity (e.g., power or electromagnetic channel).

B. Hardware Security Countermeasures

Due to the globalization of the IC supply chain, the hardware is susceptible to security threats such as IP piracy (unlicensed usage of IP), overbuilding (unauthorized manufacturing of the circuit), counterfeiting (producing a faithful copy of circuit), reverse engineering, hardware Trojan (malicious modification of circuit), and side-channel attacks [5].

In the literature, countermeasures and tools have been proposed to mitigate, prevent, or detect these threats [15]. For example, a cryptographic accelerator is a hardware-based countermeasure that can reinforce the build-in instead of the add-on defense against security threats. True Random Number Generator (TRNG) and Physical Unclonable Function (PUF) are two other effective security primitives [14], [29]. These solutions are critical for security protocols and unique IC identification, and they rely on the physical phenomena for randomness, stability, and uniqueness, such as process variations during fabrication [41].

In addition to hardware-based solutions, countermeasures enhancing the security during the hardware design process are also present in the literature. For example, side-channel analysis for HT detection using various models such as hierarchical temporal memory [10] and DL [9] has grabbed lots of attention recently. However, they postpone the detection to *post-silicon* stage. On the other hand, *Formal Verification* (FV) is a *pre-silicon* algorithmic method which converts the 3PIP to a proof checking format and checks if the IP satisfies some predefined security properties [17], [38]. Although FV leverages the predefined security properties in IP for HT detection, its detection scope is limited to certain types of HTs because the properties are not comprehensive enough to cover all kinds of malicious behaviors [32]. Some works employ model checking but are not scalable to large designs as model checking is NP-complete and can suffer from state explosion [33]. Another existing approach is *code coverage* which analyzes the RTL code using metrics such as line, statement, finite state machine, and toggle coverage to ascertain the suspicious signals that imitate the HT [45], [55].

As for IP theft prevention, *watermarking* and *fingerprinting* are two approaches that embed the IP owner and legal IP user's signatures into a circuit to prevent infringement [28], [30]. *Hardware metering* is an IP protection method in which the designer assigns a unique tag to each chip for chip identification (passive tag) or enabling/disabling the chip (active tag) [21]. *Obfuscation* is another countermeasure for IP theft [8] which comprises two main approach; *Logic Locking* and *Camouflaging*. In *Logic Locking*, the designer inserts additional gates such as XOR into non-critical wires. The circuit will only be functional if the correct key is presented in a secure memory out of reach of the attacker [48]. *Camouflaging* modifies the design such that cells with different functionalities look similar to the attacker and confuses the reverse engineering process [31]. Lastly, another countermeasure is to split the design into separate ICs and have them fabricated in different foundries so that none of them has access to the whole design to perform malicious activities [27], [54].

In [15], several academic and commercial tools have been proposed to secure hardware. For example, *VeriSketch*, *SecVerilog*, etc., are the open-source academia verification tools for securing hardware. *SecureCheck* from *Mentor Graphics*, *JasperGold Formal Verification Platform* from *Cadence*, and *Prospect* from *Tortuga Logic* are all commercial verification tools ready in the market. *PyVerilog* [39] is a hardware design tool that allows users to parse HDL code and perform *pre-silicon* formal verification side-by-side with functional verification. In short, though many approaches have been proposed to counteract security threats, security is still an afterthought in hardware design. Therefore, new countermeasures will be needed against new security threats.

C. Machine Learning for Hardware Security

In the last few decades, the advancements in Machine Learning (ML) have revolutionized the conventional methods and models in numerous applications throughout the design flow. Defenders can use ML with hardware-based observations

Authorized licensed use limited to: ETRI. Downloaded on July 03,2025 at 02:03:02 UTC from IEEE Xplore. Restrictions apply.

for detecting attacks, while attackers can also use ML to steal sensitive information from an IC, breaching hardware security [41]. Some ML-based countermeasures have been proven effective for detecting HT from hardware designs in both Register Transfer Level (RTL) or gate-level netlists (GLN) [11], [13]. In [11], the circuit features are extracted from the Abstract Syntax Tree (AST) representations of RTL codes and fed to gradient boosting algorithm to train the ML model to construct an HT library. [13] extracts 11 Trojan-net feature values from GLNs and then trains a Multi-Layer Neural Network on them to classify each net in a netlist as a normal netlist or Trojan. Similarly, researchers have applied ML for automating the process of detecting other threats. For instance, SVM can be used to analyze the on-chip sensor readings (e.g., HPCs) to identify counterfeited ICs and detect HT in real-time [16], [22]. However, as indicated in [41], effectively applying ML models is not a trivial task as the defenders must first identify an appropriate input representation for a hardware design. Unlike Euclidean data such as images, texts, or signals, finding a robust feature representation for a circuit (Non-Euclidean data) is more challenging as it requires domain knowledge in both hardware and ML. To overcome this challenge, HW2VEC provides more effective graph learning methods to automatically find a robust feature representation for a non-Euclidean hardware design.

D. Graph Learning for Hardware Design and Security

Although conventional ML and DL approaches can effectively capture the features hidden in Euclidean data, such as images, text, or videos, there are still various applications where the data is graph-structured. As graphs can be irregular, a graph can have a variable size of unordered nodes, and nodes can have a different number of neighbors, thus making mathematical operations used in deep learning (e.g., 2D Convolution) challenging to be applied [7]. Also, extracting a feature that captures structural information requires challenging efforts to achieve the desired performance. To address these challenges, recently, many *graph learning* approaches such as *Graph Convolutional Networks* (GCN), *Graph Neural Networks* (GNN), or *Graph Autoencoder* (GAE) have been proposed and applied in various applications [19], [46]. Only by projecting non-Euclidean data into low-dimensional embedding space can the operations in ML methods be applied.

In EDA applications, many fundamental objects such as Boolean functions, netlists, or layouts are natural graph representations [24]. Some works tackle netlists with GCNs for test point insertion [25] or with GNNs for fast and accurate power estimation in *pre-silicon* simulation [53]. [53] uses a GNN-based model to infer the toggle rate of each logic gate from a netlist graph for fast and accurate average power estimation without gate-level simulations, which is a slower way to acquire toggle rates compared to RTL simulation. They use GLNs, corresponding input port, and register toggle rates as input features and logic gate toggle rates as ground-truth to train the model. The model can infer the toggle rate of a logic gate from input features acquired from RTL simulation for average power analysis computed by other power analysis tools.

As for hardware security, only a few works utilizing GNN-based approaches against security threats exist [49], [50]. [50] utilizes a GNN-based approach for detecting HT in *pre-silicon* design phases without the need for golden HT-free reference. Besides, using the GNN-based approach allows the extraction of features from Data-Flow graphs to be automated. In [49], the proposed GNN-based approach can detect IP piracy without the need to extract hardware overhead to insert signatures to prove ownership. Specifically, the Siamese-based network architecture allows their approach to capturing the features to assess the similarity between hardware designs in the form of a Data-Flow Graph. In short, these works have shown the effectiveness of securing hardware designs with graph learning approaches. To further attract attention, we propose HW2VEC as a convenient research tool that lowers the threshold for newcomers to make research progress and for experienced researchers to explore this topic more in-depth.

III. HW2VEC Architecture

As Figure 3 shows, HW2VEC contains HW2GRAPH and GRAPH2VEC. During the IC design flow, a hardware design can have various levels of abstraction such as High-Level Synthesis (HLS), RTL, GLN, and GDSII, each of which are fundamentally non-Euclidean data. Overall, in HW2VEC, a hardware design p is first turned into a graph g by HW2GRAPH, which defines the pairwise relationships between objects that preserve the structural information. Then, GRAPH2VEC consumes g and produces the Euclidean representation h_g for learning.

A. HW2GRAPH: from hardware design to graph

The first step is to convert each textual hardware design code p into a graph g. HW2GRAPH supports the automatic conversion of raw hardware code into various graph formats such as Abstract Syntax Tree (AST) or Data-Flow Graph (DFG). AST captures the syntactic structure of hardware code while DFG indicates the relationships and dependencies between the signals and gives a higher-level expression of the code's computational structure. HW2GRAPH consists of three primary modules: *pre-processing*, *graph generation engine*, and *post-processing*.

*1) Pre-processing (*PRE_PROC*):* In this module, we have several automatic scripts for pre-processing a raw hardware code p. As a hardware design can contain several modules stored in separate files, the first step is to combine them into a single file (i.e., flattening). Next, to automatically locate the "entry point" top module of p, the script scans the flattened code for the keyword "module" and extracts the module names and the number of repetitions in p. Then, the script analyzes the list of discovered module names and takes the one that appears only once, which means the module is not instantiated by any other module, as the top module. Here, we denote the pre-processed hardware design code as p'.

Authorized licensed use limited to: ETRI. Downloaded on July 03,2025 at 02:03:02 UTC from IEEE Xplore. Restrictions apply.

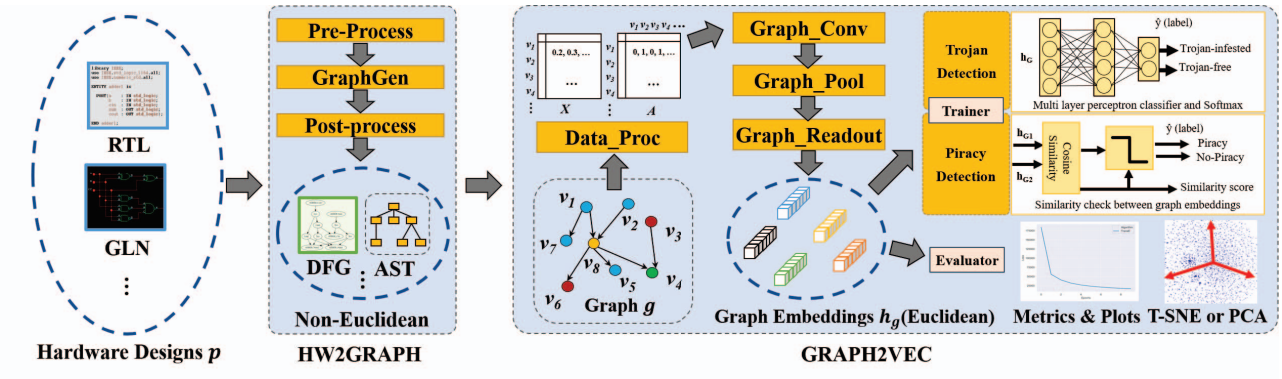

Fig. 3: The overall architecture of HW2VEC. Beginning with hardware design objects (RTL or GLN), the HW2GRAPH leverages PRE_PROC, GRAPH_GEN, and POST_PROC to extract graph representations from hardware designs in the form of node embedding matrix ($\mathbf{X}$) and adjacency matrix ($\mathbf{A}$). These graphs are then passed to GRAPH2VEC to acquire the graph embeddings for graph learning tasks of hardware security.

*2) Graph Generation Engine (*GRAPH_GEN*):* We integrate PyVerilog [40], a hardware design toolkit for parsing the Verilog code, into this module. The pre-processed code p' is first converted by a lexical analyzer, YACC (Yet Another Compiler-Compiler), into a corresponding parse tree. Then, we recursively iterate through each node in the parse tree with Depth-First Search (DFS). At each recursive step, we determine whether to construct a collection of name/value pairs, an ordered list of values, or a single name/value pair based on the token names used in Verilog AST. To acquire DFG, the AST is further processed by the data flow analyzer to create a signal DFG for each signal in the circuit such that the signal is the root node. Lastly, we merge all the signal DFGs. The resulting graph, either DFG or AST, is denoted as $g = (V, E)$. The AST is a tree type of graph in which the nodes V can be operators (mathematical, gates, loop, conditional, etc.), signals, or attributes of signals. The edges E indicate the relation between nodes. The DFG shows data dependency where each node in V represents signals, constant values, and operations such as xor, and, concatenation, branch, or branch condition, etc. Each edge in E stands for the data dependency relation between two nodes. Specifically, for all v_i, v_j pairs, the edge e_{ij} belongs to E ($e_{ij} \in E$) if v_i depends on v_j, or if v_j is applied on v_i.

*3) Post-processsing (*POST_PROC*):* The output from *Graph Generatifon Engine* is in JSON (JavaScript Object Notation) format. In this phase, we convert a JSON-formatted graph into a NetworkX graph object. NetworkX is an efficient, scalable, and highly portable framework for graph analysis. Several popular geometric representation learning libraries (PyTorch-Geometric and Deep Graph Library) take this format of graphs as the primary data structure in their pipelines.

B. GRAPH2VEC: FROM GRAPH TO GRAPH EMBEDDING

Once HW2GRAPH has converted a hardware design into a graph g, we begin to process g with the modules in GRAPH2VEC, including *Dataset Processor*, *Trainer*, and *Evaluator* to acquire the graph embedding h_g.

1) Dataset Processor: This module handles the low-level parsing tasks such as caching the data on disk to optimize the tasks that involve repetitive model testing, performing train-test split, finding the unique set of node labels among all the graph data instances. One important task of the *dataset processor* is to convert a graph $g = (V, E)$ into the tensor-like inputs $\mathbf{X}$ and $\mathbf{A}$ where $\mathbf{X}$ represents the node embeddings in matrix form and $\mathbf{A}$ stands for the adjacency information of g. The conversion between E and $\mathbf{A}$ is straightforward. To acquire $\mathbf{X}$, *Dataset Processor* performs a *normalization* process and assigns each of the nodes a label that indicates its type, which may vary for different kinds of graphs (AST or DFG). Each node gets converted to an initial vectorized representation using one-hot encoding based on its type label.

2) Graph Embedding Model: In this module, we break down the graph learning pipeline into multiple network components, including graph convolution layers (*GRAPH_CONV*), graph pooling layers (*GRAPH_POOL*), and graph readout operations (*GRAPH_READOUT*).

In HW2VEC, the *GRAPH_CONV* is inspired by the Spatial Graph Convolution Neural Network (SGCN), which defines the convolution operation based on a node's spatial relations. In literature, this phase is also referred to as *message propagation phase* which involves two sub-functions: **AGGREGATE** and **COMBINE** functions. Each input graph $g = (V, E)$ is initialized in the form of node embeddings and adjacency information ($\mathbf{X}^{(0)}$ and $\mathbf{A}$). For each k-th iteration, the process updates the node embeddings $\mathbf{X}^{(k)}$ using each node representation $h_v^{(k-1)}$ in $\mathbf{X}^{(k-1)}$, given by,

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

$$h_v^{(k)} = \textbf{COMBINE}^{(k)}(h_v^{(k-1)}, a_v^{(k)}) \quad (2)$$

where $h_v^{(k)} \in R^{C^k}$ denotes the feature vector after k iterations for the v-th node and $N(v)$ returns the neighboring nodes of v-th node. Essentially, the **AGGREGATE** collects the features of the neighboring nodes to extract an aggregated feature vector $a_v^{(k)}$ for the layer k, and the **COMBINE** combines

Authorized licensed use limited to: ETRI. Downloaded on July 03,2025 at 02:03:02 UTC from IEEE Xplore. Restrictions apply.

the previous node feature $h_v^{(k-1)}$ with $a_v^{(k)}$ to output next feature vector $h_v^{(k)}$. This message propagation is carried out for a pre-determined number of layers k. We denote the final propagation node embedding $\mathbf{X}^{(k)}$ as $\mathbf{X}^{prop}$.

Next, in *GRAPH_POOL*, the node embedding $\mathbf{X}^{prop}$ is further processed with an attention-based graph pooling layer. As indicated from [23], [52], the integration of a graph pooling layer allows the model to operate on the hierarchical representations of a graph, and hence can better perform the graph classification task. Besides, such an attention-based pooling layer allows the model to focus on a local part of the graph and is considered as a part of a unified computational block of a GNN pipeline [20]. In this layer, we perform *top-k filtering* on nodes according to the scoring results, as follows:

$$\alpha = \text{SCORE}(\mathbf{X}^{prop}, \mathbf{A}) \tag{3}$$

$$\mathbf{P} = \text{top}_k(\alpha) \tag{4}$$

where α stands for the coefficients predicted by the graph pooling layer for nodes. $\mathbf{P}$ represents the indices of the pooled nodes, which are selected from the top k of the nodes ranked according to α. The number k used in *top-k filtering* is calculated by a pre-defined pooling ratio, pr using $k = pr \times |V|$, where we consider only a constant fraction pr of the embeddings of the nodes of the DFG to be relevant (i.e., 0.5). One example of the scoring function is to utilize a separate trainable GNN layer to produce the scores so that the scoring method considers both node features and topological characteristics [23]. We denote the node embeddings and edge adjacency information after pooling by $\mathbf{X}^{pool}$ and $\mathbf{A}^{pool}$ which are calculated as follows:

$$\mathbf{X}^{pool} = (\mathbf{X}^{prop} \odot \tanh(\alpha))_{\mathbf{P}} \tag{5}$$

$$\mathbf{A}^{pool} = \mathbf{A}^{prop}{}_{(\mathbf{P},\mathbf{P})} \tag{6}$$

where $\odot$ represents an element-wise multiplication, $()_{\mathbf{P}}$ refers to the operation that extracts a subset of nodes based on P, and $()_{(\mathbf{P},\mathbf{P})}$ refers to the information of the adjacency matrix between the nodes in this subset.

Lastly, in *GRAPH_READOUT*, the overall graph-level feature extraction is carried out by either summing up or averaging up the node features $\mathbf{X}^{pool}$. We denote the graph embedding for each graph g as $h_g^{(k)}$, computed as follows:

$$h_g^{(k)} = \textit{GRAPH\_READOUT}(\{h_v^{(k)} : v \in V\}) \tag{7}$$

We use the graph embedding $h_g^{(k)}$ to model the behavior of circuits (use h_g for simplicity). After this, the fixed-length embeddings of hardware designs then become compatible with ML algorithms.

In practice, these network components can be combined in various ways depending on the type of the tasks (node-level task, graph-level task) or the complexity of the tasks (simple or complex network architecture). In GRAPH2VEC, one default option is to use one or multiple *GRAPH_CONV*, followed by a *GRAPH_POOL* and a *GRAPH_READOUT*. Besides, in conjunction with Multi-Layer Perceptron (MLP) or other ML layers, this architecture can transform the graph data into a form that we can use in calculating the loss for learning. In GRAPH2VEC, we reserve the flexibility for customization, so users may also choose to combine these components in a way that is effective for their tasks.

3) Trainer and Evaluator: The *Trainer* module takes training datasets, validating datasets, and a set of hyper-parameter configurations to train a GNN model. *HW2VEC* currently supports two types of *Trainer*, *graph-trainer* and *graph-pair-trainer*. To be more specific, *graph-trainer* uses GRAPH2VEC's model to perform graph classification learning and evaluation while *graph-pair-trainer* considers pairs of graphs, calculates their similarities, and ultimately performs the graph similarity learning and evaluation. Some low-level tasks are also handled by *Trainer* module, such as caching the best model weights evaluated from the validation set to the disk space or performing mini-step testing. Once the training is finished, the *Evaluator* module plots the training loss and commonly used metrics in ML-based hardware security applications. To facilitate the analysis of the results, HW2VEC also provides utilities to visualize the embeddings of hardware designs with t-SNE based dimensionality reduction [44]. Besides, HW2VEC provides multiple exporting functionalities so that the learned embeddings can be presented in standardized formats, and users can also choose other third-party tools such as *Embedding Projector* [37] to analyze the embeddings.

IV. HW2VEC Use-cases

In this section, we describe HW2VEC use-cases. First, Section IV-A exhibits a fundamental use-case in which a hardware design p is converted into a graph g and then into a fixed-length embedding h_g. Next, the use-cases of HW2VEC for two hardware security applications (detecting hardware Trojan and hardware IP piracy) are described in Section IV-B and Section IV-C, respectively.

A. Use-case 1: Converting a Hardware Design to a Graph Embedding

The first use-case demonstrates the transformation of a hardware design p into a graph g and then into an embedding h_g. As Algorithm 1 shows, HW2GRAPH uses *preprocessing* (PRE_PROC), *graph generation* (GRAPH_GEN) and *post-processing* (POST_PROC) modules which are detailed in Section III-A to convert each hardware design into the corresponding graph. The g is fed to GRAPH2VEC with the uses of *Data Processing* (DATA_PROC) to generate X and A. Then, X and A are processed through *GRAPH_CONV*, *GRAPH_POOL*, and *GRAPH_READOUT* to generate the graph embedding h_g. This resulting h_g can be further inspected with the utilities of *Evaluator* (see Section III-B3). In HW2VEC, we provide Algorithm 1's implementation in `use_case_1.py` of our repository.

B. Use-case 2: Hardware Trojan Detection

In this use-case, we demonstrate how to use HW2VEC to detect HT, which has been a major hardware security

Authorized licensed use limited to: ETRI. Downloaded on July 03,2025 at 02:03:02 UTC from IEEE Xplore. Restrictions apply.

Algorithm 1: Use-case - HW2VEC

Input: A hardware design program p.
Output: A graph embedding h_p for p.
def *HW2GRAPH*(p)**:**
  $p' \leftarrow$ PRE_PROC(p);
  $g \leftarrow$ GRAPH_GEN(p');
  $g\prime \leftarrow$ POST_PROC(g);
  return $g\prime$;
def *GRAPH2VEC*(g)**:**
  $X, A \leftarrow$ DATA_PROC(g)
  $X^{prop}, A^{prop} \leftarrow$ *GRAPH_CONV*(X, A)
  $X^{pool}, A^{pool} \leftarrow$ *GRAPH_POOL*(X^{prop}, A^{prop})
  $h_g \leftarrow$ *GRAPH_READOUT*(X^{pool})
  return h_g
$g \leftarrow$ *HW2GRAPH*(p);
$h_g \leftarrow$ *GRAPH2VEC*(g);

challenge for many years. An HT is an intentional, malicious modification of a circuit by an attacker [35]. The capability of detection at an early stage (particularly at RTL level) is crucial as removing HTs at later stages could be very expensive. The majority of existing solutions rely on a golden HT-free reference or cannot generalize detection to previously unseen HTs. [50] proposes a GNN-based approach to model the circuit's behavior and identify the presence of HTs.

Algorithm 2: Use-case - Hardware Trojan Detection

Input: A hardware design program p.
Output: A label indicating whether p contains Hardware Trojan.
def *use_case_2*(p)**:**
  $g \leftarrow$ HW2GRAPH(p);
  $h_g \leftarrow$ GRAPH2VEC(g);
  $\hat{y} \leftarrow$ MLP(h_g);
  if $\hat{y}[0] > \hat{y}[1]$ **then**
    return TROJAN;
  else
    return NON_TROJAN;
$\hat{Y} \leftarrow$ *use_case_2*(p);

To realize [50] in HW2VEC, we first use HW2GRAPH to convert each hardware design p into a graph g. Then, we transform each g to a graph embedding h_g. Lastly, h_g is used to make a prediction $\hat{y}$ with an MLP layer. To train the model, the cross-entropy loss L is calculated collectively for all the graphs in the training set (see Equation 8).

$$L = H(Y, \hat{Y}) = \sum_i y_i * log_e(\hat{y_i}), \tag{8}$$

where H is the loss function. Y stands for the set of ground-truth labels (either TROJAN or NON_TROJAN) and $\hat{Y}$ represents the corresponding set of predictions. Once trained by minimizing L, we use the model and Algorithm 2 to perform HT detection (can also be done with a pre-trained model). In practice, we provide an implementation in `use_case_2.py` in our repository.

C. Use-case 3: Hardware IP Piracy Detection

This use-case demonstrates how to leverage HW2VEC to confront another major hardware security challenge – determining whether one of the two hardware designs is stolen from the other or not. The IC supply chain has been so globalized that it exposes the IP providers to theft and illegal IP redistribution. One state-of-the-art countermeasure embeds the signatures of IP owners on hardware designs (i.e., watermarking or fingerprinting), but it causes additional hardware overhead during the manufacturing. Therefore, [49] addresses IP piracy by assessing the similarities between hardware designs with a GNN-based approach. Their approach models the behavior of a hardware design (in RTL or GLN) in graph representations.

Algorithm 3: Use-case - Hardware IP Piracy Detection

Input: A pair of hardware design programs p_1, p_2.
Output: A label indicating whether p_1, p_2 is piracy.
def *use_case_3*$(p_1,\ p_2)$**:**
  $g_1, g_2 \leftarrow$ HW2GRAPH(p_1), HW2GRAPH(p_2);
  $h_{g_1}, h_{g_2} \leftarrow$ GRAPH2VEC(g_1), GRAPH2VEC(g_2);
  $\hat{y} \leftarrow$ COSINE_SIM(h_{g_1}, h_{g_2});
  if $\hat{y} > \delta$ **then**
    return PIRACY;
  else
    return NON-PIRACY;
$\hat{Y} \leftarrow$ *use_case_3*(p_1, p_2);

To implement [49], the GNN model has to be trained with a graph-pair classification trainer in GRAPH2VEC. The first step is to use HW2GRAPH to convert a pair of circuit designs p_1, p_2 into a pair of graphs g_1, g_2. Then, GRAPH2VEC transforms both g_1 and g_2 into graph embeddings h_{g_1}, h_{g_2}. To train this GNN model for assessing the similarity of h_{g_1} and h_{g_2}, a cosine similarity is computed as the final prediction of piracy, denoted as $\hat{y} \in [-1, 1]$. The loss between a prediction $\hat{y}$ and a ground-truth label y is calculated as Equation 9 shows. Lastly, the final loss L is computed collectively with a loss function H for all the graphs in the training set (see Equation 10).

$$G(y, \hat{y}) = \begin{cases} 1 - \hat{y}, & \texttt{if } y = 1 \\ \text{MAX}(0, \hat{y} - \text{MARGIN}) & \texttt{if } y = -1 \end{cases} \tag{9}$$

$$L = H(Y, \hat{Y}) = \sum_i G(y_i, \hat{y_i}), \tag{10}$$

where Y stands for the set of ground-truth labels (either PIRACY or NON_PIRACY) and $\hat{Y}$ represents the corresponding set of predictions. The MARGIN is a constant to prevent the learned embedding from becoming distorted (always set to 0.5 in [49]). Once trained, we use this model and Algorithm 3 with δ, which is a decision boundary used for making final

Authorized licensed use limited to: ETRI. Downloaded on July 03,2025 at 02:03:02 UTC from IEEE Xplore. Restrictions apply.

judgment, to detect piracy. In practice, we provide the implementation of Algorithm 3 in `use_case_3.py`.

V. Experimental Results

In this section, we evaluate the HW2VEC through various experiments using the use-case implementations described earlier.

A. Dataset Preparation

For evaluation, we prepare one RTL dataset for HT detection (*TJ-RTL*) and both RTL and GLN datasets (*IP-RTL* and *IP-GLN*) for IP piracy detection.

1) The TJ-RTL dataset: We construct the *TJ-RTL* dataset by gathering the hardware designs with or without HT from the Trust-Hub.org benchmark [1]. From Trust-Hub, we collect three base circuits, `AES`, `PIC`, and `RS232`, and insert 34 varied types of HTs into them. We also include these HTs as standalone instances to the *TJ-RTL* dataset. Furthermore, we insert these standalone HTs into two other circuits (`DES` and `RC5`) and include the resulting circuits to expand the *TJ-RTL* dataset. Among the five base circuits, `AES`, `DES`, and `RC5` are cryptographic cores that encrypt the input plaintext into the ciphertext based on a secret key. For these circuits, the inserted HTs can leak sensitive information (i.e., secret key) via side-channels such as power and RF radiation or degrade the performance of their host circuits by increasing the power consumption and draining the power supply. `RS232` is an implementation of the UART communication channel, while the HT attacks on `RS232` can affect the functionality of either transmitter or receiver or can interrupt/disable the communication between them. The `PIC16F84` is a well-known Power Integrated Circuit (PIC) microcontroller, and the HTs for PIC fiddle with its functionality and manipulate the program counter register. Lastly, we create the graph datasets, *DFG-TJ-RTL* and *AST-TJ-RTL*, in which each graph instance is annotated with a TROJAN or NON_TROJAN label.

2) The IP-RTL and IP-GNL datasets: To construct the datasets for evaluating piracy detection, we gather RTL and GLN of hardware designs in Verilog format. The RTL dataset includes common hardware designs such as single-cycle and pipeline implementation of MIPS processor which are derived from available open-source hardware design in the internet or designed by a group of in-house designers who are given the same specification to design a hardware in Verilog. The GLN dataset includes ISCAS'85 benchmark [12] which includes 7 different hardware designs (`c432`, `c499`, `c880`, `c1355`, `c1908`, `c6288`, `c7552`) and their obfuscated instances derived from TrustHub. Obfuscation complicates the circuit and confuses reverse engineering but does not change the behavior of the circuit. Our collection comprises 50 distinct circuit designs and several hardware instances for each circuit design that sums up 143 GLN and 390 RTL codes. We form a graph-pair dataset of 19,094 similar pairs and 66,631 different pairs, dedicate 20% of these 85,725 pairs for testing and the rest for training. This dataset comprises of pairs of hardware designs, labelled as PIRACY (positive) or NO-PIRACY (negative).

B. HW2VEC *Evaluation: Hardware Trojan Detection*

Here, we evaluate the capability of HW2VEC in identifying the existence of HTs from hardware designs. We leverage the implementation mentioned in Section IV-B. As for hyperparameters, we follow the best setting used in [50] which is stored as a preset in a YAML configuration file. For performance metrics, we count the True Positive (TP), False Negative (FN) and False Positive (FP) for deriving Precision $P = TP/(TP + FP)$ and Recall $R = TP/(TP + FN)$. R manifests the percentage of HT-infested samples that the model can identify. As the number of HT-free samples incorrectly classified as HT is also critical, we compute P that indicates what percentage of the samples that model classifies as HT-infested actually contains HT. F_1 score is the weighted average of precision and recall that better presents performance, calculated as $F_1 = 2 \times P \times R/(P + R)$.

To demonstrate whether the learned model can generalize the knowledge to handle the unknown or unseen circuits, we perform a variant *leave-one-out* cross-validation to experiment. We perform a train-test split on the *TJ-RTL* dataset by leaving one base circuit benchmark in the testing set and use the remaining circuits to train the model. We repeat this process for each base circuit and average the metrics we acquire from evaluating each testing set. The result is presented in Table I, indicating that HW2VEC can reproduce comparable results to [50] in terms of F_1 score (0.926 versus 0.940) if we use DFG as the graph representation. The difference in performance can be due to the use of different datasets. When using AST as the graph representation for detecting HT, HW2VEC performs worse in terms of F_1 score, indicating that DFG is a better graph representation because it captures the data flow information instead of simply the syntactic information of a hardware design code. All in all, these results demonstrate that our HW2VEC can be leveraged for studying HT detection at design phases.

Method	Graph	Dataset	Precision	Recall	F1
HW2VEC	**DFG**	RTL	0.87334	0.98572	0.92596
HW2VEC	**AST**	RTL	0.90288	0.8	0.8453
[50]	**DFG**	RTL	**0.923**	**0.966**	**0.940**

TABLE I: The performance of HT detection using HW2VEC.

C. HW2VEC *Evaluation: Hardware IP Piracy Detection*

Besides the capability of HT detection, we also evaluate the power of HW2VEC in detecting IP piracy. We leverage the usage example mentioned in Section IV-C which examines the cosine-similarity score $\hat{y}$ for each hardware design pair and produces the final prediction with the decision boundary. Using the *IP-RTL* dataset and the *IP-GNL* dataset (mentioned in Section V-A), we generate graph-pair datasets by annotating the hardware designs that belong to the same hardware category as SIMILAR and the ones that belong to different categories as DISSIMILAR. We perform a train-test split on the dataset so that 80% of the pairs will be used to train the model. We compute the accuracy of detecting hardware IP piracy, which

Authorized licensed use limited to: ETRI. Downloaded on July 03,2025 at 02:03:02 UTC from IEEE Xplore. Restrictions apply.

expresses the correctly predicted sample ratio and calculates the F_1 score as the evaluating metrics. We refer to [49] for the selection of hyperparameters (stored in a YAML file).

The result is presented in Table II, indicating that HW2VEC can reproduce comparable results to [49] in terms of piracy detection accuracy. When using DFG as the graph representation, HW2VEC underperforms [49] by 3% at RTL level and outperforms [49] by 4.2% at GLN level. Table II also shows a similar observation with Section V-B that using AST as the graph representation can lead to worse performance than using DFG. Figure 4 visualizes the graph embeddings that HW2VEC exports for every processed hardware design, allowing users to inspect the results manually. For example, by inspecting Figure 4, we may find a clear separation between `mips_single_cycle` and `AES`. Certainly, HW2VEC can perform better with more fine-tuning processes. However, the evaluation aims to demonstrate that HW2VEC can help practitioners study the problem of IP piracy at RTL and GLN levels.

Method	Graph	Dataset	Accuracy	F1
HW2VEC	**DFG**	RTL	0.9438	0.9277
HW2VEC	**DFG**	GLN	0.9882	0.9652
HW2VEC	**AST**	RTL	0.9358	0.9183
[49]	**DFG**	RTL	**0.9721**	–
[49]	**DFG**	GLN	**0.9461**	–

TABLE II: The results of detecting IP piracy with HW2VEC.

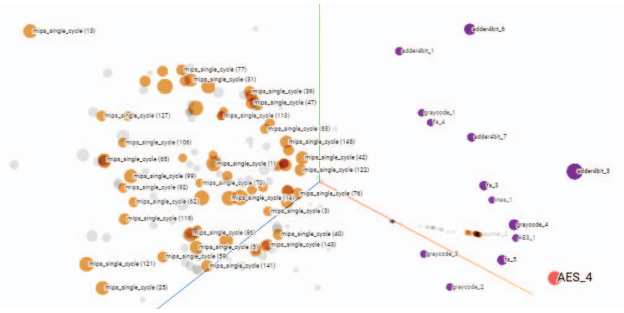

Fig. 4: The embedding visualization with 3D t-SNE.

D. HW2VEC *Evaluation: Timing*

To evaluate the time required for training and testing, we test the models on a server with NVIDIA TITAN-XP and NVIDIA GeForce GTX 1080 graphics cards. Table III indicates that the time taken by training and inference are both below 15 milliseconds, and the time taken by training is more than inference as it includes the time for performing back-propagation. As HW2VEC aims to serve as a research tool, our users must evaluate their applications within a reasonable time duration. We believe that the time spent by the graph learning pipelines of HW2VEC should be acceptable for conducting research. For practically deploying the models, the actual timing can depend on the computation power of hosting devices and the complexity of the models for the applications. Suppose our users need an optimized performance for real-time applications. In that case, they can implement the models with performance-focused programming languages (C or C++) or ML frameworks (e.g., TensorFlow) using the best model settings found using HW2VEC. As for specialized hardware that can accelerate the processing of GNNs, it is still an open challenge as indicated in [3].

Table IV indicates that the time that HW2VEC spends in converting the raw hardware code into ASTs is on average 1.98 seconds. Although [11] takes 1.37 seconds on average per hardware code, it requires domain knowledge to find a deterministic way to perform feature extraction. For DFG extraction, HW2VEC takes on average 244.58 seconds per graph as it requires recursive traversals to construct the whole data flow. In our datasets, `AES` and `DES` are relatively more complex, so HW2VEC takes 472.46 seconds on average processing them while the rest of the data instances take 16.70 seconds on average. Certainly, HW2VEC performs worse in DFG extraction, but manual feature engineering possibly requires a much longer time. In design phases, even for an experienced hardware designer, it can take 6-9 months to prototype a complex hardware design [42] so the time taken by HW2VEC is acceptable and not slowing down the design process. However, as the first open-source tool in the field, HW2VEC will keep evolving and embrace the contributions from the open-source community.

	TJ-RTL-AST	*IP-RTL-AST*
training time	10.5 (ms)	13.5 (ms)
testing time	6.8 (ms)	12.4 (ms)

TABLE III: The time profiling for training/inference.

	TJ-DFG-RTL	*IP-DFG-GLN*	*TJ-AST-RTL*
# of node	7573.58	7616.16	971.01
# of edge	8938.11	9495.97	970.01
Exec time	244.58 (s)	14.61 (s)	1.98 (s)

TABLE IV: The graph extraction time profiling. For *TJ-DFG-RTL*, the hardware `AES` and `DES` jointly take 472.46 seconds on average for DFG extraction while the rest of data instances take 16.7 seconds on average.

E. HW2VEC *Applicability*

In Section V-B and Section V-C, we have discussed the performance of the GNN-based approach in resolving two hardware security problems: hardware Trojan detection and IP piracy detection. In Section V-B, our evaluation shows that HW2VEC can successfully be leveraged to perform HT detection on hardware designs, particularly on the unseen ones, without the assistance of golden HT-free reference. The capability to model hardware behaviors can be attributed to using a natural representation of the hardware design (e.g., DFG) and the use of the GNN-based method for capturing both the structural information and semantic information from the DFG and co-relating this information to the final HT labels. Similarly, Section V-C indicates that HW2VEC can

Authorized licensed use limited to: ETRI. Downloaded on July 03,2025 at 02:03:02 UTC from IEEE Xplore. Restrictions apply.

be utilized to assess the similarities between circuits and thus can be a countermeasure for IP piracy. The use of graph representation for a hardware design and a Siamese GNN-based network architecture are the keys in [49] to perform IP piracy detection at both RTL and GLN levels. For other hardware security applications, the flexible modules provided by HW2VEC (*Trainer* and *Evaluator*) can be adapted easily to different problem settings. For example, by adjusting the *Trainer* to train the GNN models for node classification, HW2VEC can be adapted to localize the HT(s) or hardware bug(s) that exist in the hardware designs. Also, the cached models provided by HW2VEC can be used in learning other new hardware design related tasks through the transfer of knowledge from a related task that has already been learned as the idea of *Transfer Learning* suggests [43].

VI. Conclusion

As technological advancements continue to grow, the fights between attackers and defenders will rise in complexity and severity. To contribute to the hardware security research community, we propose HW2VEC: a graph learning tool for automating hardware security. HW2VEC provides an automated pipeline for hardware security practitioners to extract graph representations from a hardware design in either RTL or GLN. Besides, the toolbox of HW2VEC allows users to realize their hardware security applications with flexibility. Our evaluation shows that HW2VEC can be leveraged and integrated for counteracting two critical hardware security threats: *Hardware Trojan Detection* and *IP Piracy Detection*. Lastly, as discussed in this paper, we anticipate that HW2VEC can provide more straightforward access for both practitioners and researchers to apply graph learning approaches to hardware security applications.

References

[1] Trusthub. *Available on-line: https://www.trust-hub.org*, 2016.

[2] Special 301 report. *the United States Trade Representative*, 2017.

[3] S. Abadal, A. Jain, R. Guirado, J. López-Alonso, and E. Alarcón. Computing graph neural networks: A survey from algorithms to accelerators. *arXiv preprint arXiv:2010.00130*, 2020.

[4] S. Adee. The hunt for the kill switch. In *IEEE Spectrum*, 2008.

[5] M. AshrafiAmiri et al. Towards side channel secure cyber-physical systems. In *Real-Time and Embedded Systems and Technologies*, 2018.

[6] D. Board. Defense science board (dsb) study on high performance microchip supply. *URL www. acq. osd. mil/dsb/reports/ADA435563. pdf,[March 16, 2015]*, 2005.

[7] H. Cai, V. W. Zheng, and K. C.-C. Chang. A comprehensive survey of graph embedding: Problems, techniques, and applications. *IEEE Transactions on Knowledge and Data Engineering*, 30(9):1616–1637, 2018.

[8] J. Chen et al. Decoy: Deflection-driven hls-based computation partitioning for obfuscating intellectual property. In *Design Automation Conference (DAC)*, 2020.

[9] S. Faezi, R. Yasaei, and M. Al Faruque. Htnet: Transfer learning for golden chip-free hardware trojan detection. *IEEE/ACM Design Automation and Test in Europe Conference (DATE'21)*, 2021.

[10] S. Faezi et al. Brain-inspired golden chip free hardware trojan detection. *IEEE Transaction on Information Forensics and Security (IEEE TIFS'21)*, 2021.

[11] T. Han, Y. Wang, and P. Liu. Hardware trojans detection at register transfer level based on machine learning. In *2019 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5. IEEE, 2019.

[12] M. C. Hansen, H. Yalcin, and J. P. Hayes. Unveiling the iscas-85 benchmarks: A case study in reverse engineering. *IEEE Design & Test of Computers*, 16(3):72–80, 1999.

[13] K. Hasegawa, Y. Shi, and N. Togawa. Hardware trojan detection utilizing machine learning approaches. In *2018 17th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/12th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE)*, pages 1891–1896. IEEE, 2018.

[14] C. Herder, M.-D. Yu, F. Koushanfar, and S. Devadas. Physical unclonable functions and applications: A tutorial. *Proceedings of the IEEE*, 102(8):1126–1141, 2014.

[15] W. Hu, C.-H. Chang, A. Sengupta, S. Bhunia, R. Kastner, and H. Li. An overview of hardware security and trust: Threats, countermeasures and design tools. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2020.

[16] K. Huang, J. M. Carulli, and Y. Makris. Parametric counterfeit ic detection via support vector machines. In *2012 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, pages 7–12. IEEE, 2012.

[17] Jasper. Jaspergold: Security path verification app. 2014.

[18] S. Jose. Innovation is at risk as semiconductor equipment and materials. *Semiconductor Equipment and Material Industry (SEMI)*, 2008.

[19] T. N. Kipf and M. Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

[20] B. Knyazev et al. Understanding attention and generalization in graph neural networks. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2019.

[21] F. Koushanfar. Active hardware metering by finite state machine obfuscation. In *Hardware Protection through Obfuscation*. 2017.

[22] A. Kulkarni, Y. Pino, and T. Mohsenin. Svm-based real-time hardware trojan detection for many-core platform. In *2016 17th International Symposium on Quality Electronic Design (ISQED)*, pages 362–367. IEEE, 2016.

[23] J. Lee et al. Self-attention graph pooling. *arXiv preprint arXiv:1904.08082*, 2019.

[24] Y. Ma, Z. He, W. Li, L. Zhang, and B. Yu. Understanding graphs in eda: From shallow to deep learning. In *ISPD*, pages 119–126, 2020.

[25] Y. Ma, H. Ren, B. Khailany, H. Sikka, L. Luo, K. Natarajan, and B. Yu. High performance graph convolutional networks with applications in testability analysis. In *Proceedings of the 56th Annual Design Automation Conference 2019*, pages 1–6, 2019.

[26] Y. Moghaddas and R. Y. M. A. A. F. . f. H. . G. L. T. f. A. H. S. C. T. .-. p. o. J. . . Tommy Nguyen, Shih-Yuan Yu. Technical report for hw2vec – a graph learning tool for automating hardware security. Technical Report TR-21-02, Center for Embedded and Cyber-Physical Systems University of California, Irvine, Irvine, CA 92697-2620, USA, July 2021.

[27] S. Patnaik et al. Raise your game for split manufacturing: Restoring the true functionality through beol. In *Design Automation Conference (DAC)*, 2018.

[28] P. Poudel et al. Flashmark: watermarking of nor flash memories for counterfeit detection. In *Design Automation Conference (DAC)*, 2020.

[29] M. T. Rahman, K. Xiao, D. Forte, X. Zhang, J. Shi, and M. Tehranipoor. Ti-trng: Technology independent true random number generator. In *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2014.

[30] S. Rai et al. Hardware watermarking using polymorphic inverter designs based on reconfigurable nanotechnologies. In *2019 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, 2019.

[31] J. Rajendran et al. Security analysis of integrated circuit camouflaging. In *ACM conference on Computer & communications security*, 2013.

[32] J. Rajendran et al. Detecting malicious modifications of data in third-party intellectual property cores. In *ACM/IEEE Design Automation Conference (DAC)*, 2015.

[33] J. Rajendran et al. Formal security verification of third party intellectual property cores for information leakage. In *International Conference on VLSI Design and Embedded Systems (VLSID)*, 2016.

[34] M. Rostami, F. Koushanfar, and R. Karri. A primer on hardware security: Models, methods, and metrics. *Proceedings of the IEEE*, 102(8):1283–1295, 2014.

[35] M. Rostami, F. Koushanfar, J. Rajendran, and R. Karri. Hardware security: Threat models and metrics. In *2013 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 819–823. IEEE, 2013.

Authorized licensed use limited to: ETRI. Downloaded on July 03,2025 at 02:03:02 UTC from IEEE Xplore. Restrictions apply.

[36] K. Shamsi, M. Li, K. Plaks, S. Fazzari, D. Z. Pan, and Y. Jin. Ip protection and supply chain security through logic obfuscation: A systematic overview. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 24(6):1–36, 2019.
[37] D. Smilkov, N. Thorat, C. Nicholson, E. Reif, F. B. Viégas, and M. Wattenberg. Embedding projector: Interactive visualization and interpretation of embeddings. *arXiv preprint arXiv:1611.05469*, 2016.
[38] P. Subramanyan and D. Arora. Formal verification of taint-propagation security properties in a commercial soc design. In *Design, Automation & Test in Europe Conference (DATE)*, 2014.
[39] S. Takamaeda-Yamazaki. Pyverilog: A python-based hardware design processing toolkit for verilog hdl. In *Applied Reconfigurable Computing*, volume 9040 of *Lecture Notes in Computer Science*, pages 451–460. Springer International Publishing, Apr 2015.
[40] S. Takamaeda-Yamazaki. Pyverilog: A python-based hardware design processing toolkit for verilog hdl. In *International Symposium on Applied Reconfigurable Computing*, 2015.
[41] B. Tan and R. Karri. Challenges and new directions for ai and hardware security. In *2020 IEEE 63rd International Midwest Symposium on Circuits and Systems (MWSCAS)*, pages 277–280. IEEE, 2020.
[42] J. Teel. How long does it take to develop a new product and get it to market? Oct 2017.
[43] L. Torrey and J. Shavlik. Transfer learning. In *Handbook of research on machine learning applications and trends: algorithms, methods, and techniques*, pages 242–264. IGI global, 2010.
[44] L. Van der Maaten and G. Hinton. Visualizing data using t-sne. *Journal of machine learning research*, 9(11), 2008.
[45] A. Waksman et al. Fanci: identification of stealthy malicious logic using boolean functional analysis. In *ACM SIGSAC Conference on Computer and Communications Security*, 2013.
[46] Z. Wu et al. A comprehensive survey on graph neural networks. *IEEE Transactions on Neural Networks and Learning Systems*, 2020.
[47] K. Xiao, D. Forte, Y. Jin, R. Karri, S. Bhunia, and M. Tehranipoor. Hardware trojans: Lessons learned after one decade of research. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 22(1):1–23, 2016.
[48] Y. Xie et al. Delay locking: Security enhancement of logic locking against ic counterfeiting and overproduction. In *Design Automation Conference (DAC)*, 2017.
[49] R. Yasaei, S.-Y. Yu, and M. A. A. Faruque. Gnn4ip: Graph neural network for hardware intellectual property piracy detection. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*. Ieee, 2021.
[50] R. Yasaei, S.-Y. Yu, and M. A. A. Faruque. Gnn4tj: Graph neural networks for hardware trojan detection at register transfer level. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*. Ieee, 2021.
[51] A. Yeh. Trends in the global ic design service market. *DIGITIMES research*, 2012.
[52] R. Ying, J. You, C. Morris, X. Ren, W. L. Hamilton, and J. Leskovec. Hierarchical graph representation learning with differentiable pooling. *arXiv preprint arXiv:1806.08804*, 2018.
[53] Y. Zhang, H. Ren, and B. Khailany. Grannite: Graph neural network inference for transferable power estimation. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2020.
[54] B. Zhang et al. Analysis of security of split manufacturing using machine learning. In *Design Automation Conference (DAC)*, 2018.
[55] J. Zhang et al. Veritrust: Verification for hardware trust. *IEEE Tran. on Computer-Aided Design of Integrated Circuits and Systems*, 2015.

Authorized licensed use limited to: ETRI. Downloaded on July 03,2025 at 02:03:02 UTC from IEEE Xplore. Restrictions apply.