A Survey on Malware Detection with Graph Representation Learning

Tristan Bilot, Nour El Madhoun, Khaldoun Al Agha, Anis Zouaoui

A Survey on Malware Detection with Graph Representation Learning

TRISTAN BILOT, Iriguard, France, Université Paris-Saclay, CNRS, Laboratoire Interdisciplinaire des Sciences du Numérique, France, and LISITE Laboratory, ISEP, France
NOUR EL MADHOUN, LISITE Laboratory, ISEP, France and Sorbonne Université, CNRS, LIP6, France
KHALDOUN AL AGAHA, Université Paris-Saclay, CNRS, Laboratoire Interdisciplinaire des Sciences du Numérique, France
ANIS ZOUAOUI, Iriguard, France

Malware detection has become a major concern due to the increasing number and complexity of malware. Traditional detection methods based on signatures and heuristics are used for malware detection, but unfortunately, they suffer from poor generalization to unknown attacks and can be easily circumvented using obfuscation techniques. In recent years, Machine Learning (ML) and notably Deep Learning (DL) achieved impressive results in malware detection by learning useful representations from data and have become a solution preferred over traditional methods. More recently, the application of such techniques on graph-structured data has achieved state-of-the-art performance in various domains and demonstrates promising results in learning more robust representations from malware. Yet, no literature review focusing on graph-based deep learning for malware detection exists. In this survey, we provide an in-depth literature review to summarize and unify existing works under the common approaches and architectures. We notably demonstrate that Graph Neural Networks (GNNs) reach competitive results in learning robust embeddings from malware represented as expressive graph structures, leading to an efficient detection by downstream classifiers. This paper also reviews adversarial attacks that are utilized to fool graph-based detection methods. Challenges and future research directions are discussed at the end of the paper.

CCS Concepts: • **General and reference** → **Surveys and overviews**; • **Security and privacy** → **Malware and its mitigation**; • **Computing methodologies** → **Learning latent representations**; **Neural networks**; *Spectral methods*.

Additional Key Words and Phrases: Deep Learning, DL, GNN, Graph Neural Networks, Graph Representation Learning, Machine Learning, Malware, Malware Detection, ML.

ACM Reference Format:
Tristan Bilot, Nour El Madhoun, Khaldoun Al Agaha, and Anis Zouaoui. 2023. A Survey on Malware Detection with Graph Representation Learning. *J. ACM* 1, 1, Article 1 (March 2023), 35 pages. https://doi.org/XXXXXXX.XXXXXXX

Authors' addresses: Tristan Bilot, Iriguard, 5 Rue Bellini, Puteaux, France and Université Paris-Saclay, CNRS, Laboratoire Interdisciplinaire des Sciences du Numérique, Gif-sur-Yvette, France and LISITE Laboratory, ISEP, 10 Rue de Vanves, Issy-les-Moulineaux, France, tristan.bilot@universite-paris-saclay.fr; Nour El Madhoun, LISITE Laboratory, ISEP, 10 Rue de Vanves, Issy-les-Moulineaux, France and Sorbonne Université, CNRS, LIP6, 4 place Jussieu, Paris, France, nour.el-madhoun@isep.fr; Khaldoun Al Agaha, Université Paris-Saclay, CNRS, Laboratoire Interdisciplinaire des Sciences du Numérique, Gif-sur-Yvette, France, alagha@lisn.fr; Anis Zouaoui, Iriguard, 5 Rue Bellini, Puteaux, France, a.zouaoui@iriguard.com.

0004-5411/2023/3-ART1 $15.00
https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Malware, short for malicious software, is a generic term for unwanted programs designed to harm or exploit computer systems [1]. The detection of widespread malware such as ransomware, worms, Trojan horses or spyware, has become a major concern since their increase in both number and complexity [2]. Indeed, malware programs can appear in different forms and may be hidden under other trusted programs available on the most used platforms such as Android, Windows or even the Web. Unaware users are frequently fooled by authors of malware and important efforts have been spent to prevent these threats. Traditional detection techniques mainly rely on signatures and heuristics, where malware is detected by comparing it to existing malware or known malicious patterns. However, those methods are known to suffer from poor generalization to unknown attacks or variants and can be easily circumvented using obfuscation techniques [3]. Other behavior-based methods tend to perform better by further analyzing the malware and evaluating its intended actions before executing it. However, such techniques appear to be very time-consuming [4]. Over the last decade, Machine Learning (ML) and notably Deep Learning (DL) have sparked a sea change in a variety of fields, including cybersecurity, by allowing the model to learn from data and adapt to new patterns. This ability to adapt makes these methods well-suited to a number of tasks, including malware detection, as shown by the growing number of papers that apply ML to this problem [5].

Despite the progress made with these learning-based methods, malware detection remains a challenging task, as malware authors continue to make their techniques evolve, with the aim to evade detection. In an attempt to outperform current ML and DL methods that learn from traditional Euclidean data, graph representation learning has emerged as a promising alternative to capture complex patterns in malware programs represented as graphs. Indeed, a growing number of fields are benefiting from these graph-based learning methods and obtaining state-of-the-art results [6], as graph structures offer even more semantic information by encoding spatial relations and connectivity between entities.

Current studies on malware detection using machine learning are mainly based on the review of traditional ML and DL techniques applied to structured data. However, more and more recent papers tend to use graph representation learning in their approaches, and to the best of our knowledge, there is no literature review that specifically focuses on these techniques applied to malware detection.

This survey is a first attempt to shape the research area of malware detection with graph representation learning, by providing a comprehensive review of current approaches. Specifically, we present in this paper the following contributions:

- An overview of common representations that are used to model malware as graphs as well as techniques to extract these graph structures from raw malware data.
- A comprehensive summary of the state of the art papers, grouped according to the most common types of graphs, namely: Control Flow Graph (CFG), Function Call Graph (FCG), Program Dependence Graph (PDG), system call graph, system entity graph and network flow graph. We also propose a general architecture under which a majority of works can be abstractly summarized.
- A review of the adversarial attacks that are used against GNN-based malware detection techniques, along with a discussion on the challenges that may be encountered as well as future research directions and conclusions. In particular, we show that the works presented in this paper are very recent and that many promising directions remain unexplored.

The paper is organized as follows. In section 2, we introduce related works and further explain the contributions of our paper. In section 3, we provide background knowledge on graphs and present the fundamentals of graph representation learning and Graph Neural Networks (GNNs).

Section 4 discusses the techniques used to extract graph-structured data from malware as well as the general architecture used for their detection with representation learning techniques. Sections 5 to 7 review the state-of-the-art papers for the detection of Android, Windows and Web malware, respectively. Section 8 discusses the robustness of GNN-based detection systems against adversarial attacks. In section 9, some challenges and an overview of future research are discussed. The last section 10 concludes this paper.

2 RELATED WORKS

In existing literature, several studies have been published that aim to review malware detection using standard ML and DL techniques. The authors of the paper [3] have conducted a comprehensive review on malware detection. They first present the problem of malware detection, as well as the various challenges that can be encountered and the techniques used to overcome them. They also review a significant number of papers based on traditional methods such as signatures, behaviors and heuristics, but also cover some ML-based methods.

The paper [7] proposes to review the deep learning models employed in Android malware detection, focusing on the analysis of the strengths and weaknesses of these models. The literature is comprehensively summarized by providing useful information about each research work, including the analysis method, features, models used and their performance, and input datasets. The proposed survey in the article [8] covers a wide variety of deep neural models used for Android malware detection and mentions few graph-based methods using control flow graphs [9, 10] and App-API graphs [11, 12].

Authors in [5] surveyed the traditional ML techniques employed in Android malware detection and explain the commonly employed ML tasks such as data acquisition, data preprocessing, and feature selection. In the paper [13], a large category of deep learning methods using static, dynamic and hybrid analysis is reviewed. Important information is provided regarding the input features that can be extracted from APKs, as well as the most commonly used datasets for both benignware and malicious Android software.

The survey [14] analyzes traditional ML methods in a general approach for malware detection based on executable files. Representation learning methods applied to cybersecurity are reviewed in the study [15], with few mentions to malware detection. More recently, the paper [16] also reviewed DL methods applied to the detection of mobile malware, Windows malware, IoT malware, Advanced Persistent Threats (APTs) and Ransomware.

Regarding graph representation learning and GNN-based methods, the work [17] surveys GNN techniques employed for malware analysis with a focus on the prediction explainability. Other surveys review the applications of GNNs [6, 18–20] but none of them mention malware detection.

Indeed, after extensive research and to the best of our knowledge, the literature on malware detection using ML and DL techniques is widely covered and documented but it is still missing a review dedicated to graph ML and graph representation learning methods. Our paper focuses on analyzing recent research studies based on such methods for malware detection, starting from the extraction of graph-structured data using reverse engineering tools, to the classification of malware based on graph embeddings. Our goal is to provide the necessary knowledge to researchers interested in the application of ML to graph-structured malware, and to contribute to the advancement of this field.

3 BACKGROUND

In this section, we introduce the fundamentals about graphs along with the graph representation learning techniques leveraged to learn from these structures. We first discuss the properties of

graphs and then explain differences between traditional Deep Learning and graph representation learning, along with the types of GNNs that are frequently employed.

3.1 Graph Structures

Graphs are useful data structures to model the interactions between the entities of a complex system. They possess a great expressiveness and can represent any connected systems using only two abstract objects, which are nodes and edges.

Graph. A graph can be denoted as $G = (V, E)$ where $V = v_1, ..., v_N$ is a set of $N = |V|$ nodes (i.e. entities) and $E = e_1, ..., e_M$ is a set of $M = |E|$ edges, namely the relations between entities. Edges in the graph can either be directed (e.g. a process a forks another process b), or undirected (e.g. a bi-directional communication flow between two clients). By default, such graphs only represent a topology by incorporating the relations between different objects and do not store any local information.

Attributed Graph. Attributed graphs attach additional features to the elements of the graph, leading to a more detailed representation. A node-attributed graph assumes function $F_n : V \longrightarrow \mathbb{R}^{d_n}$ to map each node to a feature vector of d_n elements. Similarly, an edge-attributed graph assumes function $F_e : E \longrightarrow \mathbb{R}^{d_e}$ to map every edge to a vector of d_e features. Node and edge features can be conveniently described in a matrix format, where X usually represents the node feature matrix and X_e is the edge feature matrix. Furthermore, the structure of the graph is mostly designated by an adjacency matrix A.

Heterogeneous Graph. In many cases, the relations between graph objects become more complex, involving multiple types of modalities. These representations can be modeled with heterogeneous graphs, by introducing two mapping functions $\phi_v : V \rightarrow T_v$ and $\phi_e : E \rightarrow T_e$ that respectively map to a node type in T_v and an edge type in T_e.

Although other graph structures exist, current state-of-the-art graph-based malware detection methods are mostly based on these representations.

3.2 Learning on Graphs

Since nodes in a graph are inherently connected, they are not considered independent and uniformly distributed. For these reasons, traditional ML models cannot be directly applied on graphs, which suggests that specific techniques are required to deal with these interconnected structures.

3.2.1 Representation Learning. A malware detection model must first go through a training procedure where it learns parameters based on a large number of training samples, in order to approximate a relationship function between the input feature space and the output binary label. Representation learning aims at learning an intermediate function f formulated as $f : X \rightarrow \mathbb{R}^d$, which maps the input feature space X to an embedding space $\mathbb{R}^d$ that retains essential information from raw input features. Embedding representations can then be leveraged in downstream tasks such as learning word relationships [21], learning the representation of objects in images [22] or learning translation of language [23]. In malware detection, representation learning aims at creating embeddings from input data such as program code. The embeddings are then converted into a distribution that either indicates a probability to be a malware (binary classification) or to belong to a determined malware category or malware family (multi-class classification).

3.2.2 Graph Representation Learning. Standard representation learning techniques are not suited to deal with data generated from non-Euclidean domain space such as graphs. For instance, regular Convolutional Neural Networks (CNNs) [24] and Recurrent Neural Networks (RNNs) [25] are

unable to perform traditional convolutions or recurrent operations on graphs as the notion of Euclidean distance cannot be applied. Graph representation learning [26], on the other hand, is a specific area of ML that aims to learn embedding representations from graph-structured data. This involves learning embeddings from nodes, edges, or graphs in a way that ensures that objects with similarities in feature space have similar representations in embedding space. The proximity between learned representations can then be leveraged in different downstream tasks. In the field of cybersecurity, tasks such as node classification, edge classification and graph classification are frequently used. Node classification aims at finding a label for a specific object in the graph such as detecting a botnet node in a network [27–29], whereas edge classification is applied to assign a label to a relation or event, such as detecting a malicious authentication request [30, 31]. On the other hand, graph classification maps the whole graph to a label. This task is largely used in malware detection in cases where the goal is to predict the label of a binary represented as a graph [32–62]. It is also possible to work at the sub-graph level to detect areas in the graph that are responsible for the prediction done by a predictive model [63].

In literature, the first methods for graph representation learning based on graph embedding are mostly relying on random walks, where the co-occurrence of nodes is preserved. DeepWalk [64] was the first method to leverage the Skip-gram model [21] to compute embeddings from nodes that co-occur in random walks. It learns node embeddings by optimizing a neighborhood preserving objective, using random walks and word embedding techniques. First, n random walks are generated by randomly traversing the graph n times. Each walk is composed of k nodes, where k is a hyperparameter representing the length of a random walk. Then each node tries to reconstruct neighboring nodes from its random walk using the Skip-gram model.

To fully learn the embeddings, node2vec [65] integrates a second-order biased random walk that captures local and global structures using Breadth First Search (BFS) and Depth First Search (DFS) algorithms. Other methods such as LINE [66] have also achieved great performance in learning embeddings from graphs. However, most of these techniques do not share parameters between nodes [26], meaning that the model size grows linearly with the size of the graph. Moreover, these methods are highly dependent on the values of hyperparameters and tend to favor proximity information over structural information [67]. Another disadvantage of using these walk-based techniques is that they are generally transductive, meaning that a single graph is taken as input and that no inference is possible on unseen nodes or edges. Contrarily, inductive models take as input multiple graphs and can generalize to unseen examples.

3.2.3 Graph Neural Networks. Recent graph representation learning approaches tend to be inspired from the Graph Neural Network (GNN) model [68][69], which is the origin of the first application of deep neural networks to graph-structured data. Although deep learning on graphs has been democratized fairly recently, the first GNN [68] dates back to 2005 and is originally inspired from RNNs. In recent years, the popularity of deep learning has led to the emergence of new methods involving spectral and spatial convolution methods applied to graph structures, making it possible to take advantage of both the expressive structure of graphs and the power of representation learning. Spectral GNNs such as ChebNet [70] exploit the Laplacian matrix eigen decomposition in Fourier space to analyze the underlying structure of the graph. On the other hand, spatial GNNs such as Graph Convolutional Network (GCN) [71], GraphSAGE [72], Deep Graph Convolutional Neural Network (DGCNN) [73], and Graph Attention Network (GAT) [74] work directly on the adjacency matrix and capture the local neighborhood of the nodes in the graph domain, which avoids the time-consuming switch in spectral domain. GCN captures both feature and local substructure information by propagating the information along the neighboring nodes within the graph. DGCNN also leverages convolutions but is specifically designed for the graph classification task. GraphSAGE

provides an inductive solution that can scale to large graphs by sampling the neighbors during message-passing. Finally, GAT leverages the attention mechanism [75] to learn an importance weight for each neighboring node.

Variants of these models have achieved state-of-the-art results in a variety of domains such as recommender systems [76], traffic forecasting [77] and drug discovery [78]. However, GNN-based methods remain little used in cybersecurity compared to other domains where research is largely oriented in this direction.

4 MALWARE DETECTION WITH GRAPHS

In the field of ML, malware detection usually consists in extracting features from an input binary file, which are leveraged by a downstream algorithm for classification. Malware detection with graph ML follows the same idea, with the only difference that a supplementary step is introduced after the feature extraction. This step consists in transforming the input features into a graph structure that will then be fed to the classifier. In this section, we describe ways to represent malware as expressive graph structures along with a general methodology to leverage graph representation learning in downstream malware detection tasks.

4.1 Modeling Malware as Graphs

In real-world scenarios, malware programs are usually compiled binaries that may be obfuscated to hide their malicious payload. Furthermore, a same malware could be written in multiple languages or using different hardware platforms. Therefore, we think that an optimal representation of a binary program should fulfill these conditions:

- **Preserve the semantic of the program**: the actions resulting from the execution of the app should be captured by the data representation, in order to understand benign and malicious behaviors.
- **Be robust to obfuscation techniques**: the representation should capture the fundamental semantic of the program even if its code is obfuscated.
- **Be language- and platform-agnostic**: the representation should be abstract enough to transcend the programming language and the platform on which the program is written.

Building a data representation that respects all previous conditions remains a challenging task due to the constant evolution of techniques employed by attackers. In the next section, we describe the analysis methods and graph structures commonly used for the representation of malware.

4.1.1 Analysis Methods for Feature Extraction. Multiple analysis techniques are commonly employed to extract meaningful features from computer programs in an attempt to obtain an efficient representation [2]. Static analysis aims to analyze software without executing it, whereas dynamic analysis actually executes it to capture different levels of information. Both approaches have their own strengths and weaknesses. Static analysis is relatively cheap to perform and provides a comprehensive view of the program by considering all branches present in the code. However, it may not detect issues that only occur at runtime, such as memory leaks and race conditions. On the other hand, dynamic analysis can further analyze the behavior of the program by running it with different inputs and capturing the generated events at runtime. This technique is also more robust to code obfuscation, compared to static analysis. However, dynamic analysis is very resource-intensive to execute and may not be able to analyze the entire program, providing a less comprehensive view that could ignore malicious behaviors [79]. In an attempt to benefit from both techniques, hybrid analysis is a solution that tries to combine the advantages of both static and dynamic analysis, while minimizing their weaknesses.

4.1.2 Common Graph Structures for Malware Detection. The various features extracted using the aforementioned analysis methods are frequently used to represent program semantics in the form of graphs. This practice has gained more and more interest due to the faculty of graphs to represent systems in a robust and intuitive way [79–81]. The main graph structures employed for malware detection are presented as follows:

Control Flow Graph (CFG). Control flow graphs model all possible paths during the execution of a program, in an intra-procedural way. The nodes represent basic blocks, namely a sequence of instructions (e.g. assembly instructions) without any jumps. The jumps are characterized by the directed edges between the basic blocks, that represent the control flow of the program. When built from low-level assembly languages, CFGs have the faculty to be language-agnostic as they model the logic of a program without requiring language-specific instructions [51]. Although other representations such as hexadecimal also possess this characteristic, CFGs provide a more intuitive way to model programs as graphs [51, 63].

Function Call Graph (FCG). Function call graphs are a type of CFG that provide an inter-procedural view of the program, where nodes are functions and edges represent function calls from one function to another. Although FCGs offer a global view of function calls executed by the program, they generally lack the intra-procedural information that CFGs provide. To address this, some approaches can be employed by jointly using FCGs and CFGs, where embeddings from CFGs are integrated into the nodes of the FCGs, to capture both intra-procedural and inter-procedural semantic [32, 33, 53]. In the case of Android malware analysis, a prevalent approach is to statically extract the API call sequences from the application and represent them using a FCG [56, 58, 59, 82].

Program Dependence Graph (PDG). Program dependence graphs model both data and control dependencies in code, where nodes are instructions or statements and edges represent the data values and control conditions that must be fulfilled to execute the node's operation [83]. Scarcely used by current graph representation learning approaches, this graph structure may be promising to capture different conditional flows in the program [47].

System Call Graph. The system calls generated by the execution of a program can be captured via dynamic analysis and the communication with the system can be modeled with a graph, where nodes represent system calls and edges are the interactions [48] between those calls. This representation offers a low-level view of system interactions that can also benefit the detection of malware.

System Entity Graph. During its execution, a program interacts with system entities such as processes, files, registry keys or network sockets. These interactions can be captured using sandbox tools like cuckoo [84] for a deep analysis of the program's behaviors. Similarly as provenance graphs employed in host-based intrusion, these entities can be modeled as nodes in a graph, and the edges represent the operations between them [60, 85].

Network Flow Graph. The network activity generated by the program can also be monitored during its execution, and a network flow graph can be constructed with IP addresses and/or communication ports as nodes, and edges representing network flows. While some works solely rely on network traffic to detect malware activities [49, 50], others enhance their detection capabilities by combining CFGs or FCGs with network data [32, 33].

In this survey, we focus on the analysis of malware detection methods with graph representation learning for Android, Windows and Web platforms, since the vast majority of current state-of-the-art works are solely based on these platforms. For each of these platforms, we divided the

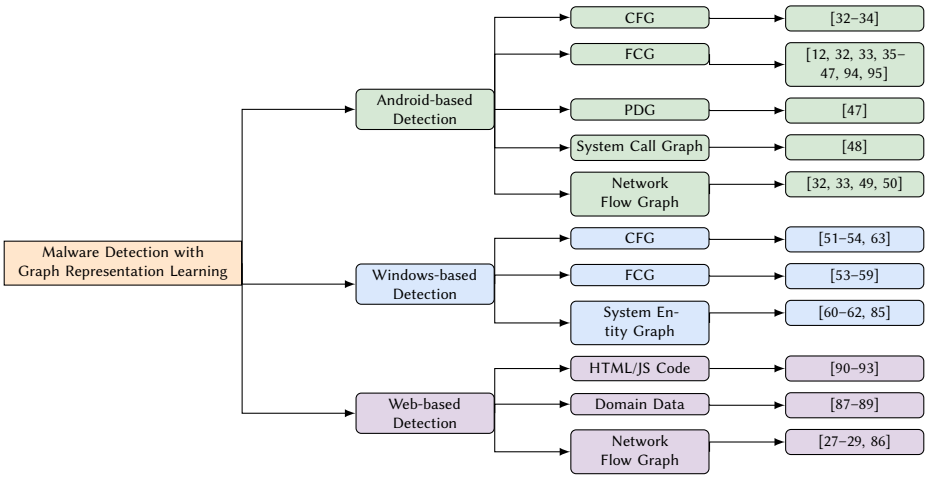

Fig. 1. Categorization of current state-of-the-art papers in malware detection with graph representation learning. In this survey, we classify papers by platform and by input data structure. Android-based detection is presented in Section 5, whereas Windows and Web detection are presented in Sections 6 and 7, respectively.

state-of-the-art into sections based on the input graph structure. The literature review is summarized in Fig. 1.

4.2 Methodology of Malware Detection with Graph Representation Learning

In literature, a majority of contributions rely on a similar sequence of operations to predict malware from source code with graph representation learning. In this section, we propose a general architecture, shown in Fig. 2, to summarize the process of malware detection from graph-represented source code on Android and Windows platform.

The first step involves extracting code from the binary, which is usually disassembled to assembly language or decompiled to higher-level language. In the case of malware detection with dynamic analysis, this step assumes dynamic input features such as a stream of API calls or system entity interactions (see Section 6.1). Subsequently, a graph builder is employed to transform the code into a graph-structured representation that preserves the program's semantics, as detailed in Section 4.1.2. Typically, these first two steps are performed using reverse engineering tools listed in Table 1. Optionally, the graph can be preprocessed and attributed with hand-crafted features, located on nodes or edges. Then, graph representation learning techniques, such as GNNs, leverage the semantics of code to learn node embeddings, which capture the relationships and the role of internal instructions, functions, or API calls, depending on the input graph. These embeddings are commonly generated using well-known GNN variants, which have been discussed in Section 3.2.3. Other techniques employ word embedding techniques inspired from Natural Language Processing (NLP) to learn the meaning of opcode or API functions, enabling integration of the resulting embeddings into a global graph structure for GNNs to learn the structural properties. The majority of studies consider malware detection as a graph classification task, whereby node embeddings are transformed with a global pooling operation (or readout) to create a single fixed-size graph embedding vector that encapsulates all information of the graph. The final vector can then be classified using traditional ML or DL methods.

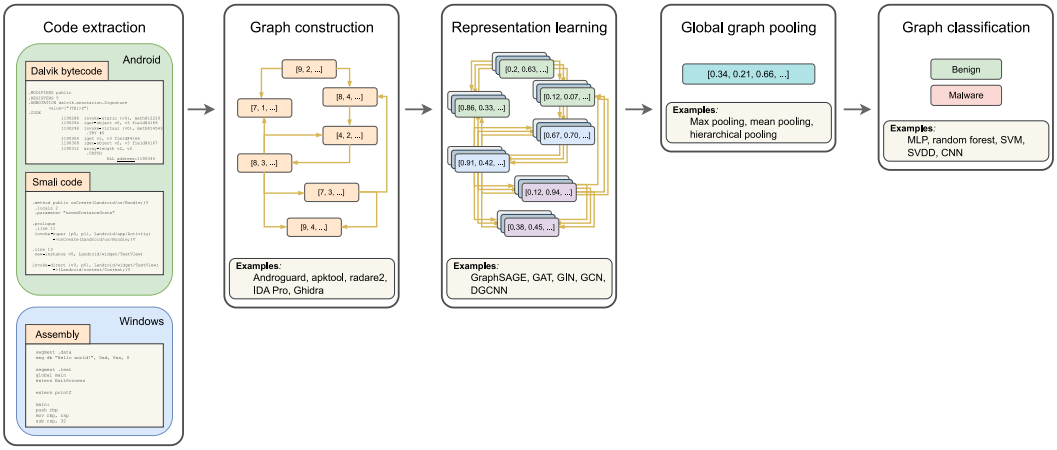

Fig. 2. General architecture of malware detection from static code analysis based on graph representation learning.

Table 1. Common tools employed for data extraction and graph construction based on static or dynamic analysis.

Representation	Tools
CFG	Androguard [96], radare2 [97], IDA Pro [98], Ghidra [99]
FCG	Androguard, Apktool [100], graph4apk [101], WALA [102], Angr [103], radare2, IDA Pro, cuckoo [84], Ghidra
PDG	Androguard, Ghidra
Syscalls	strace [104], SystemTap [105], ltrace [105]
System entities	cuckoo, Any.Run [106]
Network flows	Argus [107], Zeek [108], Splunk [109], Joy [110]

Green refers to tools specifically designed for Android APKs.

5 GRAPH-BASED ANDROID MALWARE DETECTION

In this section, we present graph-based malware detection for Android platform, starting from the global methodology to build graphs from disassembled Android applications, to the review of existing works that leverage graph representation learning for the detection of malware.

5.1 Android-based Graph Structures

Android applications are packed into APK files containing the source code, resources, manifest file and assets. After unzipping an APK, numerous features can be extracted to be used in downstream ML tasks. The manifest file provides the big picture of an app and contains its meta-information such as the required permissions to run it, hardware features and components. Resources such as images, videos or audio files are also available for further analysis. However, these data are inherently flat and do not provide enough structured information to build a graph. This is why most approaches leverage the actual source code to represent the logic of the app as a graph. As an APK is a production-ready app package, the code has already been compiled and assembled into a Dalvik bytecode format (.dex file). In practice, this bytecode can be disassembled into higher-level human-readable code (.smali files). Based on both code representations, control flow graphs (CFGs), function call graphs (FCGs), program dependence graphs (PDGs) and APIs can be extracted in a

static way. This static extraction process is demonstrated in Fig. 3. Concerning dynamic analysis, API call and system call sequences can be recorded when running the app in a sandboxed environment. It is also possible to capture the network traffic by monitoring packets in a strategic area of a network.

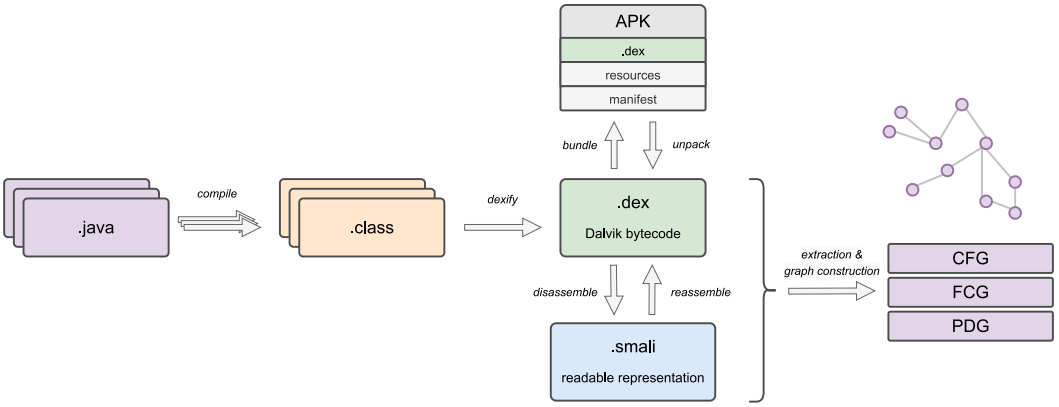

Fig. 3. Android application compilation and disassembling process using static analysis. Java files are compiled into classes and are assembled into a single dex file. The APK is packed with the dex file, the manifest file and other resources. The dex bytecode can be disassembled into higher-level smali code and graph structures can be constructed from either dex or smali code depending on the use case requirements.

5.2 Android-based Approaches

In this section, we review state-of-the-art Android-based papers, classified by types of graph, also summarized in Table 2.

5.2.1 CFG Approaches for Android Malware Detection. CFGs offer a remarkable abstract representation of programs to detect malware. Hybroid [32] leverages this representation by extracting basic blocks from APKs. Three types of embeddings are then constructed from the code, to capture different semantics, namely opcode embedding, basic block embedding and CFG embedding, where each representation is associated to a level of abstraction. The semantic of opcodes (Dalvik instructions) and basic blocks (sequence of instructions) is computed using the NLP-based model word2vec with Skip-gram. Precisely, Skip-gram learns embedding vectors for each basic block's raw instructions, by using an opcode to predict its surrounding opcodes. Indeed, the operands are not leveraged here as they are affected by the usage of Dalvik VM. For the basic block embeddings, they compute the weighted mean of the inner instructions' opcode. These basic block embeddings then become the node embeddings from the point of view of a FCG and structure2vec [111] creates a final graph embedding vector for graph classification. In parallel, the network traffic generated by the app is also captured with dynamic analysis using Argus [107]. The packets are transformed into flows to summarize the communication between the Android device and the destination IP addresses, using various statistics. After a feature selection step, important features are combined with the FCG embeddings for downstream classification with gradient boosting on the CICAndMal2017 dataset, where the model demonstrates a F1-score of 97% and beats other methods such as DREBIN [112] and SVM [113].

On the other hand, hybrid-Falcon [33] transforms network flow data into 2D images on which a bi-directional LSTM captures flow representations from pixels. On the same dataset, the F1-score is further improved to 97.09%.

The paper [34] provides a solution to locate MITRE ATT&CK Tactics Techniques and Procedures (TTPs) detected from a subgraph of a CFG. Node representations are extracted with Inferential SIR-GN [114] and the prediction is done using a random forest. To identify the subgraph responsible for the prediction, the authors rely on SHAP [115] to attribute for each input feature a value that indicates its relevance for the final output. TTPs are successfully detected with a F1-score of 92.7%.

Table 2. Summary of Android-based malware detection approaches leveraging graph representation learning

Data	Analysis	Graph type	Classification	Learning	Models	Year	Paper
CFG+FCG+Flows	Hybrid	Attributed	Graph	Supervised	Word2vec, structure2vec	2021	Hybroid [32]
	Hybrid	Attributed	Graph	Supervised	Bi-LSTM, word2vec, structure2vec	2021	hybrid-Falcon [33]
CFG	Hybrid	Attributed	Graph, Subgraph	Supervised	Inferential SIR_GN, RF	2021	Fairbanks et al. [34]
FCG	Static	Attributed	Graph	Supervised	GCN	2018	CG-GCN [35]
					GNN	2021	CGDroid [36]
					GCN, CBOW	2021	Cai et al. [37]
					GNN, Skip-gram	2021	Xu et al. [38]
					GraphSAGE	2021	Vinayaka et al. [39]
					CGMM	2021	Errica et al. [40]
					GAT, node2vec	2021	Catal et al. [41]
					GNN, Bi-LSTM, TF-IDF	2022	DeepCatra [42]
					GCN, GraphSAGE, GIN	2022	Lo et al. [43]
					VGAE, word2vec	2022	Gunduz et al. [44]
					GCN	2022	Lu et al. [45]
					GraphSAGE, VGAE	2022	Yumlembam et al. [46]
App-API FCG	Static	Heterogeneous	Node	Supervised	Multi-kernel model, Meta-path	2017	HinDroid [12]
					GCN, Skip-gram	2021	GDroid [94]
					Custom HAN, Meta-path	2021	Hawk [95]
PDG+FCG	Static	Attributed	Graph	Supervised	structure2vec, word2vec, SIF	2021	Android-COCO [47]
Syscall Graph	Dynamic	Attributed	Graph	Supervised	GCN	2020	John et al. [48]
Flow Graph	Hybrid	Attributed	Graph	Supervised, Unsupervised	GNN, GAE, Residual connections, Deep SVDD	2021	NF-GNN [49]
	Hybrid	Attributed	Graph	Supervised	MPNN, GRU	2023	NT-GNN [50]

Data represents the data type taken as input by the models; **Analysis** refers to the analysis method that is leveraged to extract features (e.g. extracting a CFG from smali code is static, capturing network traffic from a running app is dynamic, whereas leveraging both results in a hybrid analysis); **Graph type** designates one of the graphs introduced in Section 3.1, here we characterize a graph as attributed if a node or an edge is attributed either with hand-crafted features, raw features (e.g. raw instructions, function names) or embeddings (e.g. word embedding of a function), whereas a heterogeneous graph deals with multiple types and possibly different attributes; **Classification** designates the final object to classify (i.e. the classification task); **Learning** is the learning method used to train the models, whereas **Models** refer to the models on which the paper is inspired; **Paper** and **Year** identify the work and its publication year.

5.2.2 FCG Approaches for Android Malware Detection. The semantic information captured by FCGs in programs makes this data structure a predominant choice in graph-based malware detection. For instance, a FCG is constructed from Smali code in work [35], where each node is attributed with function attributes such as the method type (system API, third-party API, etc.) and the requested permissions (required permissions for the execution of the function). The graph embeddings are then trained in a supervised way using the GCN propagation rule, presented in Eq. 1 and 2.

$$H^{(k)} = \sigma\left(\mathcal{A}H^{(k-1)}\mathbf{W}^{(k)}\right) \tag{1}$$

where $H^{(k)}$ represents the node embedding matrix at layer k with $H^0 = X$, the feature matrix. σ represents an activation function, $\mathbf{W}^{(k)}$ is a trainable weight matrix and $\mathcal{A}$ designates the normalized adjacency matrix with self-loops described below.

$$\mathcal{A} = \tilde{D}^{-\frac{1}{2}}(A + I)\tilde{D}^{-\frac{1}{2}} \tag{2}$$

where I the identity matrix and $\tilde{D}$ is the degree matrix of adjacency matrix with self-loops $A + I$. The Drebin dataset is used for final evaluation with a F1-score of 99.68%.

In CGdroid [36], multiple node features are extracted from the disassembled methods in order to build a FCG that captures the semantic of functions. Indeed, each node is mapped to a vector of hand-designed features such as the number of string constants, the number of call and jump instructions, the associated permissions, etc. A GNN computes graph embeddings and a MLP is used for downstream graph classification on Drebin and Androzoo [116] datasets, where a baseline is outperformed by 8% in F1-score.

Word embedding techniques are employed in [37] to consider functions similarly as words and learn the meaning of functions. The embeddings are then assigned as attributes to each corresponding function node in a FCG, and a GCN is used as graph learning method. The proposed method achieved 99.65% F1-score with random forest classifier on a private dataset.

Similarly, authors in [38] leverage word embedding to transform Android opcodes from text to vectors using Skip-gram. In the same way as [37], the embeddings are used as nodes in a FCG and this graph is fed into a GNN to compute a fixed-size graph embeddings vector. A 2-layer MLP and softmax are used as last layers of the architecture for graph classification, with an average accuracy of 99.6%.

In the reference [39], FCGs are extracted from APKs using Androguard and each node stores attributes related to the structural meaning of the node in the graph (e.g. node degree) or features extracted from the actual disassembled functions (e.g. method attributes, method opcodes' summary). Using these previous features, GCN, GraphSAGE, GAT and TAGCN [117] are benchmarked together, with a better performance achieved with GraphSAGE. First, each node i uniformly selects a fixed-size set of neighbors, denoted $\mathcal{N}(i)$. Neighbors are then aggregated using a mean aggregation function such as:

$$\mathbf{h}_{\mathcal{N}(i)}^{(l+1)} = \text{aggregate}\left(\left\{\mathbf{h}_j^{(l)}, \forall j \in \mathcal{N}(i)\right\}\right) \tag{3}$$

where $\mathbf{h}_{\mathcal{N}(i)}^{(l+1)}$ is the embedding of node i at layer $l+1$ and $h_j^{(l)}$ denotes the embedding of a neighbor node j at layer l. The embedding of i at previous layer is then concatenated with the aggregated representation and then learned by a neural network.

$$\mathbf{h}_i^{(l+1)} = \sigma\left(\mathbf{W}^{(l)} \cdot \left[\mathbf{h}_i^{(l)}, \mathbf{h}_{\mathcal{N}(i)}^{(l+1)}\right]\right) \tag{4}$$

where [,] represents the concatenation operation. The embedding is finally normalized:

$$\mathbf{h}_i^{(l+1)} = \frac{\mathbf{h}_i^{(l+1)}}{||\mathbf{h}_i^{(l+1)}||_2} \tag{5}$$

Malware Android apps from CICMalDroid2020 dataset are used for evaluation, with a best F1-score of 92.23%.

For the detection of obfuscated malware, the paper [40] leverages a call graph where nodes are attributed with nodes' out-degree only. The Contextual Graph Markov Model (CGMM) [118]

is used to learn the embeddings that are then classified using a standard feed-forward network, achieving a macro F1-score of 97.2%.

In the study [41], the node embeddings of an API call graph are computed using node2vec and aggregated with a GAT model. The authors explain that the use of node2vec as feature extraction method is justified by a 3% increase in F-score compared to traditional graph centrality features. Using the attention aggregation from GAT as a final step, the proposed solution reaches 94.1% in F-score.

API call traces and opcode features are further exploited in DeepCatra [42], where Term Frequency-Inverse Document Frequency (TF-IDF) is used to identify critical Android APIs from call traces. The prior knowledge required to detect critical APIs is extracted from popular codebases available in online repositories such as CVE [119] and Exploit-DB [120]. Then, call graphs are generated from apps with Wala [102] by also considering the knowledge of previously identified critical APIs. Finally, a custom GNN and a bi-directional LSTM are trained in an end-to-end manner, to respectively capture the graph topology and temporal features from call traces. The output vectors produced by both models are then merged with a fully connected layer and a softmax layer is used for binary classification. The proposed model is evaluated against [48] and other CNN-, GCN- and LSTM-based baselines. Compared to the baselines, DeepCatra achieves best results with 95.83% F1-score and further improves false positive and false negative rates.

In the paper [43], the authors build an enhanced FCG, where node attributes are graph centrality indicators based on graph nodes' importance. Precisely, PageRank [121], in/out degree and node betweenness centralities are used as node features. The FCG is fed into a GCN, a GraphSAGE and a GIN model for comparison, and all leverage jumping knowledge [122], a technique to overcome over-smoothing in GNN architectures by introducing jump connections in the neural network. GraphSAGE outperforms the compared baselines by an important gap on the multi-class classification task, with a F1-score respectively of 94% and 97% on Malnet-Tiny [123] and Drebin [112] datasets.

Gunduz et al. [44] compute node-level embeddings from sensitive API function calls using word2vec and leverage a Variational Graph Auto-Encoder (VGAE) [124] to learn a reduced representation of embeddings that is used for downstream classification, with a F-measure of 93.4%.

For the detection of obfuscated malware from call graphs, the paper [45] leverages a GCN with subgraphs along with a denoising method. The method is evaluated on a private dataset made of samples from VirusShare and AndroZoo, whereas Proguard [125] is used for code obfuscation.

A more general approach is proposed by Yumlembam et al. [46], who consider each Android app as a local graph, where nodes are APIs and an edge exists between two APIs if they co-exist in a same code block (i.e. a code segment from a smali file, located between *.method* and *.endmethod*). A global graph then represents the connections among applications with co-occuring APIs. Multiple variants of the model are compared, using different features. One variant considers attributed nodes with 5 centrality indicators: degree, betweenness, closeness, eigenvector and PageRank. Another variant considers the permissions and intents from the manifest file. After benchmarking multiple models, the best combination is to calculate graph embeddings with GraphSAGE and to concatenate the vector with the permissions and intents features. The resulting vector is then passed into a classifier trained in supervised fashion, where a traditional CNN achieves best performance. In order to test the robustness of GNN-based malware detection models, the authors also provide a generative model inspired from VGAE, which can generate adversarial API graphs to fool the predictive model. The proposed methods are finally compared to many state-of-the-art techniques and respectively achieve 98.33% and 98.68% accuracy in Drebin and CICMaldroid datasets.

Other works model the interactions between APKs and API calls as a global view. In this paper, we describe this representation as an App-API graph, where a node is an Android app or an API

call and an edge is a relation between two endpoint nodes such as two apps containing a same API, or two APIs coming from a same package. In literature, this type of structure is mostly represented as a large Heterogeneous Information Network (HIN) [126] (i.e. an heterogeneous graph), where the goal is to classify malware app nodes. In HinDroid [11, 12], an App-API graph models the interactions between Android apps and APIs, as a HIN. A node is either an app or an API call, whereas an edge is one among multiple relations representing whether extracted API calls belong to the same code block, or if they are with the same package name, or use the same invoke method. Semantic is extracted from the heterogeneous graph using meta-paths. A meta-path is a path composed of a series of different node and edge types that captures a particular semantic in the graph. Because of their heterogeneous composition and significant semantic extraction abilities, meta-paths are frequently used in heterogeneous graphs. These meta-paths constructed by the traversal of graph are leveraged by a multi-kernel SVM to determine a weight for each meta-path. These weights are then considered for downstream app's node classification with a final F1-score of 98.84%.

Gdroid [94] is another technique that extracts a graph from API co-occurrence in APKs to build an App-API graph. The Skip-gram model first encodes APIs while preserving context information, with the objective to obtain similar embeddings for APIs with similar usages. On top of the graph, a GCN propagates the information and learns node embeddings that are leveraged for downstream node classification with 98.99% accuracy.

In Hawk [95], more than 180k APKs are extracted to build a large heterogeneous App-API graph that also models relations such as App-Permission, App-Class or App-Interface. Meta-paths along with meta-graphs are extracted from the graph to capture semantics. More precisely, two models are built for in-sample and out-sample nodes. The former is based on a custom heterogeneous GAT that fully leverages meta-structures to capture embeddings whereas the latter utilizes these embeddings in an incremental setting to quickly learn new embeddings without requiring re-learning. Hawk was evaluated against many baselines and outperforms them by a large gap on both in-sample and out-sample malware detection.

5.2.3 PDG Approaches for Android Malware Detection. Program Dependence Graphs have been widely used in optimization tasks due to their faculty to model data and control flow from programs [83]. For similar reasons, this structure is also employed in malware detection but remains little used with graph representation learning.

Android-COCO [47] leverages the native code of dynamic libraries (.so files) along with the Android bytecode (.dex files) to construct a PDG for each app. Structure2vec computes the graph embeddings, that are then passed into a MLP for graph classification. For a more accurate prediction, a FCG is created, on which graph embeddings are also computed (similarly than Hybroid [32]). The predictions of both graphs are finally combined using an ensemble algorithm and a 99.88% F1-score is reached on samples from Drebin, AMD [127] and Androzoo datasets.

5.2.4 System Call Approaches for Android Malware Detection. System calls provide a low-level view of system interactions, able to model attacks patterns. The authors in [48] rely on dynamic analysis to record the system calls generated by the activity of a running APK to detect malware behaviors. Each node in the graph is one among 26 selected system calls and is summarized by 4 centrality indicators as node features: Katz, Betweennes, Closeness and PageRank centralities. Edges represent interactions between those system calls while the app is running. A GCN and a pooling layer compute graph embeddings and a fully-connected layer along with a softmax activation are used for graph classification. Their implementation achieves 92.3% accuracy and similar true positive rate as SVM but significantly outperforms all other methods regarding the false positive rate. As

for the PDG, system call graphs are still scarcely used in current graph representation learning literature.

5.2.5 Network Flow Approaches for Android Malware Detection. NF-GNN [49] proposes to detect Android malware using network flows constructed from pcap network captures. Only IP addresses are used to build the graph structure, the source and destination ports are not leveraged here. They propose a custom GNN model with a propagation function that considers both edge and node features. A MLP is first used on edge features and endpoint nodes aggregate the edges using the concatenation operation along with a residual connection [128]. Then, three downstream methods are proposed for classification based on graph embeddings: a graph classifier (supervised), a custom graph autoencoder (unsupervised), and a one-class neural network (unsupervised). The graph classifier consists in adding a pooling layer and a dense layer with softmax activation. The GAE method uses as encoder the GNN described previously and as decoder a custom model that tries to reconstruct the original edge features from the compact node representations produced by the GNN encoder. Finally, the one-class network alternative is described as a pooling layer followed by a Deep Support Vector Data Description (SVDD) [129] model. The three variants have been compared to 7 supervised and unsupervised baseline methods on the CICAndMal2017 dataset and all methods outperform compared baselines by an important gap.

Similarly, NT-GNN [50] monitors the traffic produced by running APKs and converts the packets into flows using CICFlowMeter-V3. Communication ports are don't considered and only IP addresses from flows are leveraged to build the graph. A model inspired from the MPNN is used for message-passing between nodes in the network graph and the classification is performed after applying a readout of the computed node embeddings. Node representations are updated by passing the previous representation with the new aggregated representation from neighbors into a GRU and the model is trained using cross-entropy loss. A 97% F1-score is reached on both CICAndMal2017 and AAGM datasets.

Other flow-based works are presented in [32, 33], where the authors leverage network flows in combination with CFGs and FCGs to further improve the prediction capacities of the model (see Section 5.2.1).

5.3 Android Malware Datasets

In this section, we present Android datasets employed for graph-based malware detection tasks. A summary of the datasets used in previous studies is available in Table 3. Based on the current information provided from the respective websites of AMD, Malgenome and PRAGuard datasets, the release of these datasets has stopped for maintenance reasons and are not further described in this paper.

CICAndMal2017 [130]. An Android malware dataset developed by the Canadian Institute of Cybersecurity (CIC). It comprises 10,854 APK files published between 2015 and 2017 on Google Play Store. The dataset consists of 6,500 benign apps and 4,354 malware divided into Benign, Adware, Ransomware, SMS and Riskware classes. For each scenario, network packets are also collected and transformed into flows using CICFlowMeter [131]. This tool generates, for each flow, 80 features based on statistics from the packets contained within the flow.

CICMalDroid [132]. This dataset was also made public by the Canadian Institute for Cybersecurity. It is composed of 17,341 APK samples collected during one year in 2018. Malware examples are divided into 5 classes: Benign, Adware, Banking, SMS and Riskware. Along with the APK files that can be used for classification tasks, three kinds of features are also provided for each sample: statically extracted features (e.g. intents, permissions and services), dynamically observed behaviors

Table 3. Datasets employed in Android malware detection studies.

Paper	Datasets	Performance
Hybroid [32]	CICAndMal2017	97% F1
hybrid-Falcon [33]	CICAndMal2017,AndroZoo	97.09% F1
Fairbanks et al. [34]	VirusTotal	92.7% F1
CG-GCN [35]	Drebin+Apkpure+HKUST	99.68% F1
CGDroid [36]	Drebin+AndroZoo	~99% F1
Cai et al. [37]	AndroZoo+VirusShare	99.65% F1
Xu et al. [38]	Drebin+AMD+PRAGuard+AndroZoo	99.6% acc
Vinayaka et al. [39]	CICMalDroid2020,AndroZoo	92.23% F1
Errica et al. [40]	AMD,Google Play Store	97.2% macro F1
Catal et al. [41]	CICMalDroid2020+ISCX-AndroidBot-2015	94.1% F1
DeepCatra [42]	Drebin+DroidAnalytics+VirusShare +CICInvesAndMal2019+AndroZoo	95.83% F1
Lo et al. [43]	MalNet-Tiny,Drebin	94%, 97% F1
Gunduz et al. [44]	ISCX-AndroidBot-2015+CICMalDroid2020	93.4% F1
Lu et al. [45]	VirusShare+AndroZoo+Google Play Store	~63%-95% F1
Yumlembam et al. [46]	Drebin,CICMalDroid2020	98.33%, 98.68% acc
HinDroid [12]	Private	98.84% F1
GDroid [94]	AMD,Google Play Store	98.99% acc
Hawk [95]	CICAndMal2017+VirusShare+AndroZoo +Google Play Store	>96% F1
Android-COCO [47]	Drebin+AMD+AndroZoo	99.88% F1
John et al. [48]	Drebin+AMD+Malgenome	92.3% acc
NF-GNN [49]	CICAndMal2017	96.75% AUC, ~99% F1
NT-GNN [50]	CICAndMal2017,AAGM	97%, 97% F1

For each **Paper**, we provide the list of datasets along with the performance of the model. **Datasets** separated by a "+" refer to a global dataset built from the assembling of each mentionned dataset, whereas the use of a "," means that the authors have conducted experiments on separated datasets. If multiple comma-separated datasets are present in Datasets, and only one metric is assigned in Performance, then this metric refers to the performance of the first dataset. Otherwise, each dataset is assigned to a performance metric. If the number of metrics in Performance is greater than the number of datasets, then multiple variants of the models are proposed and we suggest to refer to the original paper for further information. In this paper, **Performance** metrics are defined such that the accuracy denoted as "acc"=$(TP + TN)/(TP + TN + FP + FN)$, where TP, TN, FP, FN refer to true positive, true negative, false positive and false negative, respectively. "F1"=2×Precision×Recall/(Precision + Recall) where Precision= $TP/(TP+FP)$ and Recall= $TP/(TP+FN)$. "AUC" refers to the Area Under the Receiver Operating Characteristic Curve.

(e.g. system calls, binder calls, composite behaviors) and network traffic in pcap format. Features are available from CSV files, ranging from 139 to 50,621 files depending on the APK.

AndroZoo [116]. A collection of Android apps provided by the University of Luxembourg. In 2022, the dataset contains more than 21M APKs, mostly including benign apps from Google Play Store but also malware from VirusShare. Many other APK stores are fetched to continually update the collection. For each APK file, 9 features are collected such as the sha256 hash, the app compilation date or the size of the .dex file. AndroZoo is often used in combination with other APK malware datasets to obtain a balanced number of benign samples.

Drebin [112]. Made available by the MobileSandbox project, this dataset also provides malware Android apps. A total of 5,560 APKs divided into 179 malware families were collected between August 2010 and October 2012. Considering the important variety of classes, most multi-class classification papers use the top-k classes from the dataset by sorting malware based on the number of samples per class. Otherwise, all examples can be used for binary classification. Each APK is summarized by 10 features such as permissions, intents and providers. This dataset does not contain

any benign example so an additional dataset such as AndroZoo should be used to complete the dataset with benignware examples.

MalNet [123]. A large dataset containing FCGs extracted from AndroZoo APK files. According to the original paper from 2021, it was at this time the largest database for graph representation learning with 1,262,024 graphs, averaging over 15k nodes and 35k edges per graph, divided among 47 types. GNNs have been applied to this dataset in the original paper [123], where baselines such as GCN, GIN or Feather are benchmarked together. Moreover, FCGs have demonstrated promising results when combined with representation learning techniques, in trying to overcome the polymorphic nature of malware [55, 133]. For smaller experiments, MalNet-Tiny is a subset of MalNet composed of 5,000 graphs of at most 5k nodes and balanced in 5 types. The authors also released MalNet Explorer [134], a useful web interface to explore the graph structures of malware from the dataset.

6 GRAPH-BASED WINDOWS MALWARE DETECTION

The increasing level of sophistication of malware on Windows platform along with the widespread usage of this operating system worldwide has become a major concern to preserve the safety of many users. After the success of graph representation learning in many classification tasks, its application to Windows-based malware detection has become obvious. As for Android malware detection, in this section, we present a global methodology to model Windows binaries as graphs and we perform a literature review of current approaches involving graph learning algorithms.

6.1 Windows-based Program Graph Structures

In Windows, executable files are encapsulated following the portable executable (PE) file format. File extensions such as .exe, .dll and .sys are all PEs with different roles. Dynamic-link libraries (DLL) and system (SYS) files are both libraries of functions that are loaded into memory and used by other programs. The former is intended for a general function sharing purpose, whereas the latter is intended for a more specific use related to device drivers and hardware configurations by the system [135]. The EXE file is the one that is actually executed and that communicates with function libraries. Similarly as for Android APKs, PE files can be analyzed statically to extract source code employed in downstream graph structures such as CFGs, FCGs and PDGs. PEs are compiled code, meaning that the binary has to be first disassembled or decompiled to obtain an appropriate human-readable representation like assembly. A number of works also leverage dynamic analysis to detect Windows malware, where the executable binaries are run into a sandbox to monitor system entity interactions, system calls, network flows and live API calls. This process is described in Fig. 4.

6.2 Windows-based Approaches

This section reviews the approaches employed in the Windows-based papers presented in Table 4.

6.2.1 CFG Approaches for Windows Malware Detection. In MAGIC [51], assembly code is extracted from PE files and converted into CFG, where nodes are basic blocks composed of multiple assembly instructions, and edges represent the program flow along these basic blocks, as explained in Section 4.1.2. Instead of using a standard GCN that was initially made for node classification, the authors prefer to leverage a Deep Graph Convolutional Neural Network (DGCNN) [73], which is especially designed for graph classification. The proposed DGCNN leverages adaptive max pooling and replaces the original Conv1D layer with a custom layer that considers graph embedding idea. The training procedure of this model minimizes the mean negative logarithmic loss in an end-to-end manner. The authors trained the model for malware classification on two private CFG-based

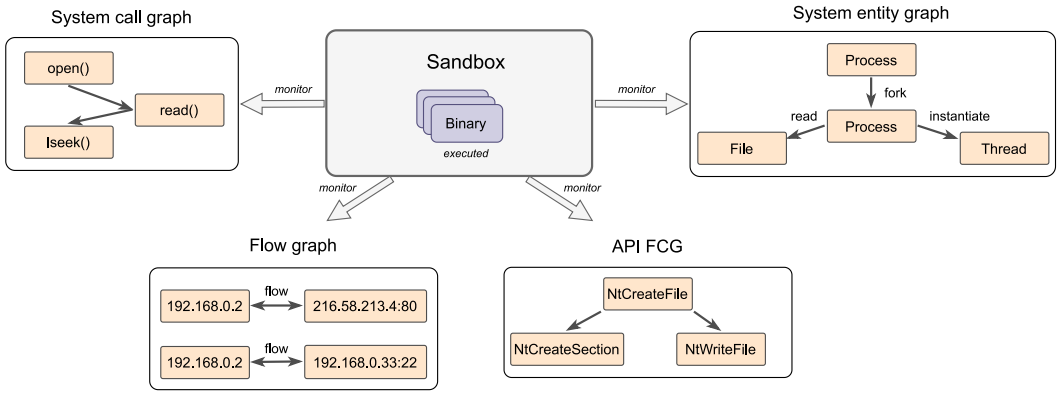

Fig. 4. Extraction of graph structures from a running binary using dynamic analysis. The file is safely executed in a sandbox environment and all system- and network-level events are monitored for downstream graph construction.

Table 4. Summary of Windows-based malware detection approaches leveraging graph representation learning.

Data	Analysis	Graph type	Classification	Learning	Models	Year	Paper
CFG	Static	Attributed	Graph	Supervised	DGCNN	2019	MAGIC [51]
	Hybrid	Attributed	Graph	Supervised	GNN, word2vec	2021	HawkEye [52]
CFG+FCG	Static	Attributed	Graph	Supervised	GAT, Random Walk, BERT	2021	Wang et al. [53]
	Static	Attributed	Graph	Supervised	GraphSAGE	2022	MalGraph [54]
FCG	Static	Attributed	Graph	Supervised	node2vec, SDA	2019	DLGraph [55]
	Dynamic	Attributed	Graph	Supervised	DGCNN	2019	Oliveira et al. [56]
					GCN	2021	SDGNet [57]
					GCN, Markov chain	2021	Li et al. [58]
					GAT, GIN, Word2vec	2022	DMalNet [59]
Entities	Dynamic	Attributed	Graph	Supervised	GCN	2019	MeQDFG [60]
	Dynamic	Heterogeneous	Graph	Semi-supervised	Heterogeneous GNN, Meta-path, Attention, Siamese Network	2019	MatchGNet [61]
	Dynamic	Heterogeneous, Attributed	Node	Supervised	GraphSAGE, Meta-path	2021	MalSage [85]
	Dynamic	Heterogeneous	Graph	Self-supervised	GAT, Meta-path, Contrastive learning	2022	FewM-HGCL [62]

datasets: MSKCFG (inspired by [136]) and YANCFG (inspired by [137]). These datasets were not made publicly available, but the procedure to generate the CFGs is provided in the paper. The model was also evaluated on the Microsoft Malware Classification Challenge dataset [136] and reaches 99.25% accuracy.

A cross-platform approach is proposed in HawkEye [52] to extract both static and dynamic CFGs from binaries (i.e. Windows, Linux and Android platforms). Embeddings of instructions are computed using word2vec whereas the final graph embedding is calculated using a custom GNN that leverages the word embeddings as nodes. Malware samples were collected from VirusShare and Androzoo, and benign examples for Windows and Linux platforms were collected from libraries. Using this cross-platform method, HawkEye reaches an accuracy of 96.82% on Linux, 93.39% on Windows, whereas 99.6% accuracy is obtained on Android.

A similar CFG based on an assembly is employed in the paper [53]. First, the semantic of functions is computed using random walk and the BERT [138] language model. These embeddings are then assigned to the nodes of a FCG that represents a global view of the program. The importance between function nodes is then calculated with a GAT model, whose goal is to compute an attention score e_{ij} for each connected pair of function nodes i and j:

$$e_{ij} = \text{LeakyReLU}\left(\mathbf{a}^T \left[\mathbf{W}h_i, \mathbf{W}h_j\right]\right)$$

where $[,]$ is the concatenation operation, $\mathbf{a}$ and $\mathbf{W}$ respectively represent a trainable attention vector and a weight matrix. The features of nodes i and j are respectively represented here by h_i and h_j. As this score is not normalized, a softmax activation is applied on all neighbors to obtain coefficients that are associated to probability distribution:

$$\alpha_{ij} = \text{softmax}_j\left(e_{ij}\right) = \frac{\exp\left(e_{ij}\right)}{\sum_{k \in \mathcal{N}_i} \exp\left(e_{ik}\right)}$$

where $\mathcal{N}_i$ is the neighborhood of node i. The updated representation h'_i of node i can be obtained by gathering neighbor embeddings along with the calculated coefficients. The authors also leverage multi-head attention to calculate multiple representations that are then concatenated together:

$$h'_i = \|_{k=1}^{K} \sigma\left(\sum_{j \in \mathcal{N}_i} \alpha_{ij}^k \mathbf{W}^k h_j\right) + \mathbf{W_R} h_i$$

where $||$ represents the concatenation operation, K is the number of attention heads and $\mathbf{W_R}h_i$ is a trainable residual connection. By leveraging the function-level and program-level embeddings with attention, the overall model achieved 90.88% and 72.44% F1-score on two private datasets.

In MalGraph [54], both CFG and FCG are also leveraged together. Intra-procedural relations are captured with GraphSAGE from the CFG and the embeddings are attributed to nodes in a FCG whose embeddings are also computed with GraphSAGE to capture inter-procedural relations. Max-pooling transforms node embeddings into graph embeddings for downstream PE malware graph classification. All samples were collected from VirusShare and VirusTotal and IDA Pro was utilized for disassembling.

Although previous works are by definition detection methods, they provide poor insights on the actual patterns and areas in the graph that led to the final prediction. CFGExplainer [63] is an explainability framework specially designed to explain the predictions done by GNNs on malware classification tasks based on CFGs. This method identifies subgraphs in the CFG, that contribute to the final prediction of a given GNN. Concerning this particular task, CFGExplainer outperforms other explainability frameworks like GNNExplainer [139], SubgraphX [140] and PGExplainer [141].

6.2.2 FCG Approaches for Windows Malware Detection. DLGraph [55] leverages static analysis to extract API calls and FCG from binaries. The model relies on a FCG that represents the interactions between functions from disassembled PE files, along with a vector of extracted Windows API calls. The node embeddings of the FCG are calculated using node2vec and are fed into a stacked denoising auto encoder (SDA) [22] to create a graph embedding vector. Similarly, a SDA takes as input the API vector and the two resulting vectors are then concatenated and passed into a softmax regression layer for classification, where an accuracy greater than 99% is achieved.

In the paper [56], a behavioral graph is constructed from API call sequences monitored during the execution of PEs in a sandbox environment. A DGCNN is then used to compute embeddings for graph classification. Based on their experiments, the authors released a public dataset made of

42,797 malware and 1,079 benignware API call sequences [142]. On the original imbalanced dataset, the proposed model achieves an F1-score of more than 99%.

SDGNet [57] similarly captures API calls along with attributes using dynamic analysis. Weighted graph normalization methods are utilized to transform the adjacency matrix into three symmetrical matrices that describe interactions of node information. A GCN-based model computes node embeddings for these matrices and all representations are merged into a final graph embedding that is leveraged for classification. A total of 8,909 labeled samples were collected from the Alibaba Cloud Malware Detection Base on Behavior dataset [143] and the final model achieves 97.3% accuracy.

The reference [58] uses a GCN on a directed cyclic graph that was pre-processed with Markov chain. The nodes represent API calls, whereas the edges (u, v) are weighted according to the number of calls from u to v. Malware samples used for evaluation are also collected dynamically using a sandbox environment in order to create a private dataset, on which a maximum accuracy of 98.32% is reached.

DMalNet [59] also leverages dynamic analysis to build FCGs from API calls captured during the execution of PE binaries. Here, both API names and API arguments are considered. The embeddings of these attributes are learned with a custom GIN model, whereas more complex structural interactions between APIs are learned with attention using a GAT. After computing embeddings for both semantics, important information is captured with a gPool layer [144, 145] for feature selection. More precisely, this pooling operation attributes to each node a projection score and selects top-k nodes based on these scores. The gPool output of the GIN model is taken as input by the GAT to capture the interactions between API calls. A final accuracy of 98.43% is obtained by leveraging a MLP for classification on a private dataset.

6.2.3 Entity Graph Approaches for Windows Malware Detection. The dynamic nature of communications between system entities provide valuable information to detect malicious behaviors. In the paper [60], authors monitor such interactions using dynamic analysis. Directed multi-edge graphs are built from interactions between four system entities: processes, files, registry keys and network sockets. Edges represent data transmission between entities such as system calls. Concretely, a node is represented by a categorical value between 0 and 3, and an edge stores a feature vector containing the size of the transmitted data and the time when the action occurred. Representation learning is done using a GCN and an attention-based pooling function is used to transform node embeddings into fixed-size graph embedding vector that is classified by a feed-forward network. Experiments have been conducted on a private dataset with samples from VirusShare and the proposed solution achieved 86.22% accuracy.

In MatchGNet [61], malware detection is considered as the detection of a malicious process that behaves differently from benign processes. The authors first designed an invariant graph modeling technique to capture interactions in a heterogeneous graph that represents relations among system entities such as processes, files or sockets. A GNN-based encoder with attention learns the representations and a Siamese Network [146] learns the similarity between known benign programs and new incoming programs. During inference, the similarity distance between these two programs results in a score that is utilized for final classification. This model can thus be trained using only benign examples. The final evaluation is performed on a real enterprise dataset composed of 300 million events recorded on Windows and Linux hosts.

The paper [85] represents malware behaviors as a weighted heterogeneous graph, where nodes are either an executable file (PE), a file, a file suffix or a module, and edges represent different (weighted) actions between entities. A custom model based on GraphSAGE and meta-paths is implemented to deal with the heterogeneity of the graph. This model achieved 91.56% accuracy on a private dataset.

FewM-HGCL [62] introduces a self-supervised method based on contrastive learning for few-shot malware variants detection. The authors construct a heterogeneous graph with 5 types of entities: process, API, file, signature, and network. API nodes are not only identified by their name but are also characterized by their category. API attributes are irregular by default and feature hashing [147] is used to transform them into compact fixed-length vectors. The idea behind contrastive learning is to use the natural co-occurrence associations in data as a substitute for ground truth labeled information. To perform contrastive learning, negative samples and positive samples are generated using different data augmentation techniques. Three distinct GAT models are then trained to respectively learn graph embeddings on the original graph, the positive graph and the negative graph. A discriminator aims to capture the similarity between the original graph and the positive graph along with the dissimilarity between the original graph and the negative graph. All self-trained embeddings are finally merged in a readout layer for downstream graph classification with an accuracy ranging from 85.73% to 98.65% on multiple datasets for malware variants detection.

6.3 Windows Malware Datasets

On Windows platform, a majority of works rely on private datasets constructed from public malware samples downloaded from VirusShare and VirusTotal. Using these data, the comparison between papers is ineffective. Other works evaluate their experiments on the Microsoft Malware Classification Challenge, which makes the performance comparison between papers possible. Datasets used in previous studies are reviewed in Table 5.

Table 5. Datasets employed in Windows malware detection studies.

Paper	Datasets	Performance
MAGIC [51]	MMCC	99.25% acc
HawkEye [52]	VirusShare+AndroZoo	96.82%, 93.39%, 99.6% acc
Wang et al. [53]	VirusShare+VirusTotal	90.88%, 72.44% F1
MalGraph [54]	VirusShare+VirusTotal	99.97% acc
DLGraph [55]	MMCC+VirusShare+KafanBBS	>99% acc
Oliveira et al. [56]	VirusShare	~99.4% F1
SDGNet [57]	Alibaba Dataset	97.3% acc
Li et al. [58]	VirusShare+VirusTotal	98.32% acc
DMalNet [59]	VirusShare+VirusTotal	98.43% acc
MeQDFG [60]	VirusShare	86.22% acc
MatchGNet [61]	Private	96.53% acc
MalSage [85]	VirusTotal	91.56% acc
FewM-HGCL [62]	VirusShare,ACT-KingKong,Ember, API Call Sequences,BIG 2015	85.73%-98.65% acc

Microsoft Malware Classification Challenge (MMCC) [136]. This dataset contains more than 20,000 malware samples that fall into nine families, namely Ramnit, Lollipop, Kelihos ver3, Vundo, Simda, Tracur, Kelihos ver1, Obfuscator.ACY and Gatak. For each binary, the dataset provides two data representations: the bytecode and the disassembly code (disassembled with IDA Pro). The assembly code can then be used to build attributed CFGs [51] or FCGs [55].

VirusShare [148], VirusTotal [149]. It is common to download PE malware and Android malware from these two websites. Some works rely on dynamic analysis to run the downloaded malware into a cuckoo sandbox [58–60], whereas others build static CFGs and FCGs [53, 55].

7 GRAPH-BASED WEB MALWARE DETECTION

Compiled binaries are not the only way to hide malware payloads with the intent to execute malicious activity. Malware are also present in web technologies and efficient cyberdefense systems should also be designed. Graph representation learning remains little used in the web-based malware detection, but we think that web technologies are by definition graph-oriented and they could be potential candidates to these graph learning techniques. In this section, we will cover several graph structures for web malware detection along with works involving graph representation learning.

7.1 Web-based Graph Structures

The web contains a wide range of interconnected web pages accessible through Internet. All these pages along with the fundamental components that compose the structure of Internet are likely to hide malware used by attackers to fool unaware web visitors. Detecting such malicious behaviors can be achieved by direclty analyzing the source code of pages or the communications between services present on the web. In this survey, we focus our study on techniques employed to detect malware activities from source code, DNS communications and network flows. For the representation of web malware as code, web pages' content is generally fetched in order to build a hierarchical graph from HTML or JavaScript code. The DNS scene can also be modeled as a graph to excavate useful associations among domains, as shown in Fig. 5. In this case, the graph is generally heterogeneous as it models complex interactions between different types of entities such as hosts, IP addresses, segments, etc. A graph representation learning model can then learn malicious code patterns hidden in the code of untrusted websites, or detect anomalous DNS communications likely to be attacks such as DNS spoofing or DNS flood attacks. As for Android and Windows malware detection, network flows may also be used to model IP communications between clients and to detect network-level attacks like botnet and DDoS.

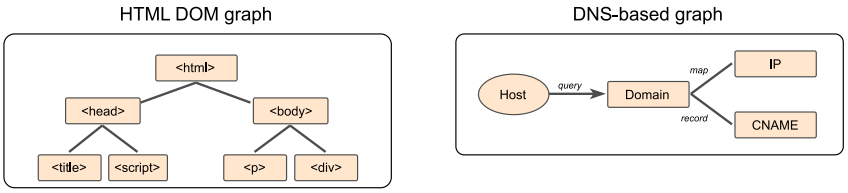

Fig. 5. Example of Web-based graph structures built from HTML code (left) and DNS data (right).

7.2 Web-based Approaches

Recent approaches for web-based malware detection with graph representation learning are presented below and summarized in Table 6.

7.2.1 Code-based Approaches for Web Malware Detection. The inner structure of web pages contains important indicators that may be represented as a graph for downstream graph ML tasks. Ouyang et al. [90] suggests to fetch the HTML content of phishing web pages to build a heterogeneous graph on which learn the malicious structures that may help the detection of phishing attacks. Here, the HTML files are parsed into DOM trees, where a node represents a HTML tag [150] and edges are links between the tag and its inner tags. Leaf nodes are especially considered, as they store the actual string content that is located and displayed on the web page. A RNN is used at node-level to encode type-specific features based on the attributes and the text content. To capture long-range relations, the authors used the Topology Adaptive Graph Convolutional Network (TAGCN) [117]

Table 6. Summary of Web-based malware detection approaches leveraging graph representation learning.

Data	Analysis	Graph type	Classification	Learning	Models	Year	Paper
Code	Static	Attributed, Heterogeneous	Graph	Supervised	TAGCN, RNN	2021	Ouyang et al. [90]
	Hybrid	Attributed	Graph	Semi-supervised	GCN, Random Forest	2022	PhishGNN [91]
	Static	Attributed	Node	Semi-supervised	GCN, Residual connections, word2vec	2022	GraphXSS [92]
	Static	Attributed	Graph	Supervised	Gated GNN, GAT, word2vec	2022	JStrong [93]
DNS data	Dynamic	Attributed	Node	Supervised	DeepWalk	2019	He et al. [87]
		Heterogeneous	Node	Supervised	Heterogeneous GCN, Meta-path, Random Walk	2020	Deepdom [88]
		Attributed, Heterogeneous	Node	Semi-supervised	Custom GNN with attention	2021	GAMD [89]
Flows	Dynamic	Graph	Node	Supervised	GCN	2020	Zhou et al. [27]
		Graph	Node	Supervised	Inferential SIR_GN	2022	isirgn1 [28]
		Heterogeneous	Node	Semi-supervised	GCN, Meta-path	2020	Bot-AHGCN [29]
		Attributed	Graph	Supervised	GIN	2022	GraphDDoS [86]

to aggregate neighbors from different hops instead of the direct neighborhood used in traditional GCNs. All embeddings are then reduced with max-pooling and fed into a fully-connected layer for graph classification. The model is trained in an end-to-end manner with cross-entropy loss on a private dataset created from public phishing repositories, with a 95.5% accuracy.

On the other hand, PhishGNN [91] leverages the hyperlink structure of web pages, extracted from HTML content. Here, the graph is built by crawling the outgoing links of a HTML page, up to a certain depth. Each node represents a URL that is mapped to a vector of 25 hand-designed features automatically extracted during crawling from the URL text, the DOM content and the domain. The proposed framework first trains a random forest algorithm with the 25 features of all known websites, and then predicts the label of the outgoing links extracted with the crawler. A message-passing GNN such as GCN is then used on top of the graph containing the random forest predictions, and graph embeddings are aggregated with graph pooling for downstream classification. The combination of random forest predictions with a GNN achieves a clear classification improvement when compared to the performance of each algorithm separately. In the same way as [90], the performance of the model is evaluated on a private dataset, where PhishGNN reaches an accuracy of more than 99%.

Against malicious XSS payloads, authors in [92] propose to first preprocess payload samples with word2vec and further leverage the relations between words by creating a graph on which a GCN learns embeddings. A residual connection is employed in the GCN propagation function to accelerate the convergence of the model. This implementation is then evaluated on a private XSS payload dataset, with a 99.6% prediction accuracy.

JStrong [93] leverages graph representation learning to detect malicious JavaScript (JS) code. The authors study the performance of multiple graph structures to effectively represent the semantic of the JS code. Notably, word2vec along with a Gated GNN and a GAT model are leveraged to compute the embeddings of different graphs, namely an Abstract Syntax Tree (AST), a CFG, a PDG and an Object Dependence Graph (ODG) [151]. Embeddings are then aggregated with mean pooling and classified in graph classification setting, where best performance is achieved by applying the final architecture on a pruned PDG, which retains the most important information. Indeed, the authors explain that the PDG extracts more semantic information than an AST or a CFG due its ability to model data flow information and statement execution order.

7.2.2 Domain-based Approaches for Web Malware Detection. For the detection of malicious domains, the paper [87] proposes a lightweight approach that considers domains from passive DNS data. First, a domain relationship graph is built from domains and A records, where each node represents a domain and an edge exists if two domains resolve to the same IP address. Node embeddings are then computed using DeepWalk and the prediction is enhanced with hand-designed DNS features.

Deepdom [88] applies a custom GCN on meta-paths extracted from a HIN containing entities such as clients, domains, IP addresses, and multiple relations like query, register and record. The proposed method has the ability to support inductive node embedding, and can thus generalize to unseen DNS nodes.

Another approach to detect unwanted domains is suggested in the paper [89]. Here, the DNS interactions extracted from a private university network are also modeled with a HIN, where nodes are either a host, a domain or a resolved-IP, and edges could be a request or a resolution relation. Self-attention mechanism is applied directly on the heterogeneous edges, in contrast to HAN [61] that uses this mechanism on meta-path scheme. With this technique, the representation of different types of neighbor nodes is projected in the embedding space of the specific target node type. Node embedding of malicious domains can then be detected in embedding space using a downstream fully-connected layer.

7.2.3 Network Flow Approaches for Web Malware Detection. Network monitoring tools are effective to capture malicious traffic that may come from malware programs. In the paper [27], authors propose to automate the detection of botnets by leveraging network traces between hosts with a GNN-based method. The task is here to classify malicious nodes, namely an IP address that participates in the botnet attack. A GCN model first computes the node embeddings across all the graph and malicious botnet nodes are then classified using a neural network at node-level. In botnets, infected bots receive commands from either centralized command-and-control (C&C) or decentralized peer-to-peer architectures (P2P). This technique respectively aims to detect C&C and P2P botnets with 99.03% and 99.51% accuracy. For this evaluation, a private dataset was created based on botnet topologies and background network traffic.

Another approach trained on background traffic with embedded synthetic botnet topologies is considered in reference [28]. Here, the Inferential SIR_GN model is used to generalize on unseen and very large graphs. Indeed, network-based graphs can rapidly grow in size and adapted models are required to deal with these high-dimension structures. The node embeddings computed by SIR_GN are then fed into a standard neural network to classify botnet nodes.

Zhao et al. [29] represents flows as a heterogeneous graph where nodes are flow entities and edges are events between flows. The nodes are also attributed with features such as timestamp and user-agent. Meta-paths are hand-designed to extract semantic from the heterogeneous graph. Then, a weighted similarity graph is built by computing similarity between node pairs and a GCN computes the embeddings. As in previous approaches, botnet nodes are detected using a neural network at node-level.

Network flows also provide useful information to detect DDoS attacks. In the paper [86], GraphDDoS aims to detect low-rate and high-rate DDoS attacks by considering the relationship between flows along with the relationship of packets from a single flow. First of all, an endpoint graph is constructed by dividing packets into two groups based on the source and destination IP addresses. The GIN model then performs message-passing between every nodes and computes embeddings. The task here is to classify DDoS attack graphs so the node embeddings are passed into a readout layer to perform graph classification.

7.3 Web Malware Datasets

In this section, we present web-based datasets on which graph structures can be built for graph representation learning.

Table 7. Datasets employed in Web malware detection studies.

Paper	Datasets	Performance
Ouyang et al. [90]	OpenPhish+PhishTank+TrancoTop1M	95.5% acc
PhishGNN [91]	OpenPhish+PhishTank+AlexaTop1M	99.7% acc
GraphXSS [92]	XSSed	99.6% acc
JStrong [93]	Petrak+GeeksOnSecurity+VirusTotal	99.95% acc
He et al. [87]	Various datasets+AlexaTop1M	94% acc
Deepdom [88]	Private	97.91% acc
GAMD [89]	Private	92.77% acc
Zhou et al. [27]	Private	99.03%, 99.51% acc
isirgn1 [28]	CAIDA+Synthetic samples	97.85%-99.78% F1
Bot-AHGCN [29]	CTU-13, Private	98.27%, 98.22% micro-F1
GraphDDoS [86]	CIC-IDS-2017, CIC-DoS-2017	99.59%, 94.56% F1

PhishTank [152], OpenPhish [153]. These two websites provide an updated list of known malicious URLs that is frequently updated by the community. Phishing URL detection is then possible either by directly extracting features from the raw URL as text, or by crawling the webpage's content if the domain still exists.

TrancoTop1M[154], AlexaTop1M [155]. Provide benign URLs from the top 1 million sites on Internet ranked by traffic. Often used in combination with malicious URLs from PhishTank and OpenPhish.

XSSed[156]. An online webpage providing an updated list of XSS payloads and XSS vulnerable websites. This page was created in early 2007 with the scope of increasing security and privacy on the web, and remains today the largest online archive of XSS vulnerable websites [156].

Petrak[157], GeeksOnSecurity[158]. Two datasets hosted on GitHub, containing malicious JS file samples. Petrak's dataset contains almost 40,000 JS malware samples, whereas the second contains malware samples divided into 1,156 HTML files, 1,357 JS files and 33 skipped files.

CAIDA [159]. Between 2008 and 2019, the Center for Applied Internet Data Analysis (CAIDA) captured passive network traces from high-speed monitors on a business backbone link. Pcap files allow access to hundreds of Gigabytes of requests that were logged over the course of these years. All these traces provide a good solution to model background traffic in synthetic network datasets [27].

CTU-13 [160]. This dataset was made public by CTU University, Czech Republic. It contains network traffic captures from benign activity and from 13 botnet attack scenarios. Packets are available in pcap format and flows are in Netflow format and captured with Argus. In total, more than 850M packets and around 20M bi-directional flows are proposed in the dataset.

CIC-IDS2017 [161]. CIC-IDS2017 is a network dataset suggested by the Canadian Institute of Cybersecurity (CIC). It consists of benign and offensive network flows. Each flow is associated with 80 features collected over the course of 5 days in a controlled setting using CICFlowMeter. These data are available in pcap and CSV format. Seven types of web attacks are represented, namely Brute Force, HeartBleed, Botnet, DoS, DDoS, Web Attack, and Infiltration.

CIC-DoS-2017 [162]. This dataset provides network traces from common application layer DoS attacks simulated in a testbed environment. The victim host is a webserver running Apache Linux and the attacker is supposed to be non-oblivious, meaning that he knows to optimize traffic to maximize the attack damage. The resulting experiment lasts for 24 hours and the final dataset results in 4.6GB of data.

8 ADVERSARIAL ATTACKS

Despite the capabilities of machine learning in classification tasks, these techniques are not immune to adversarial attacks, that aim to disturb the predictions of the model by introducing adversarial examples, crafted from small perturbations in the input. We review in this section background knowledge on adversarial attacks along with existing approaches against traditional and graph-based malware detection.

8.1 Background

Adversarial attacks have seen great success notably in the computer vision domain [163], where the goal is to craft adversarial images that the model will misclassify by predicting a wrong label. For a given classification model f denoted as $f : x \rightarrow y$ that predicts a label $y \in \mathbb{Y}$ given the features $x \in \mathbb{X}$ of an input example $z \in \mathbb{Z}$, we denote two categories of adversarial attacks [164, 165]. A feature-space attack aims to craft adversarial features $x' \in \mathbb{X}$ (e.g. a modified FCG or CFG) such that the distance between x and x' in feature-space is minimized, and such that the model f predicts a label $y' \in \mathbb{Y}$ different from y. However, a problem-space attack works directly on the real-world input z instead of the features x. The goal then becomes to minimize the cost between z and an adversarial example z' (e.g. modified source code), such that f predicts another label y'. We can further classify adversarial attacks by the prior knowledge acquired by the attacker. White-box attacks assume that the attacker has full knowledge of the target model f, namely he knows about the architecture, the parameters, etc. In contrast, black-box attacks refer to scenarios where only the output prediction is known by the attacker, making these attacks more difficult to succeed but also more likely to be faced in real-world applications. Other methods called gray-box attacks, live at the intersection between black- and white-box methods, where the attacker has knowledge of some prior knowledge that shall be defined depending on the use case.

8.2 Adversarial Attacks and Malware Detection

In the case of malware, adversarial attacks aim to craft new malware examples that preserve maliciousness while misleading the classification of the model. Formally, given an input malware z such as a PE or an APK, we want to find either a modified version of its compiled code z' or a modified version of its graph representation x' that will not be detected by the model. However, these requirements imply multiple constraints that are hard to be fulfilled in the case of malware adversarial attacks. Indeed, as explained by Ling et al. [164], adversarial attacks have been successfully applied to image classification as it is easy to retrieve a corresponding image z' from an adversarial feature x' because an image can be simply represented as a 2D-array of pixels. In other words, a differentiable and bi-injective inverse feature mapping function ϕ^{-1} can be approximated to map features from the feature space to an image in problem space. However, retrieving the original malware code z' from a feature representation x' (i.e. finding a similar inverse function) is much more challenging as the reconstructed input needs to fulfill multiple conditions to remain executable [166]. Notably, the generated adversarial example needs to respect a specific format such as PE or APK, but also needs to preserve the malicious payload while still being executable without error. Furthermore, in a black-box scenario, the attacker does not know beforehand the feature

representation taken as input by the detection model, which further complicates the adversarial process.

Despite these complicated requirements, researchers found adversarial attacks that can be employed to detect malware. To evade raw bytes-based malware detection models, works [167] and [168] append an adversarial sequence of bytes to the malware. Other works prefer to modify regions in the PE header [169] or extend the DOS header [165]. However, these techniques are ineffective for higher-level representations such as those based on API calls. For this purpose, many works insert additional API calls in feature space to add noise in the representation and evade the detection systems [170–176]. In other works, reinforcement learning (RL) is leveraged to manipulate the original malware in order to evade detection while maintaining a correct format and semantic [177–179].

8.3 Adversarial Attacks on Graph-based Malware Detection

Adversarial attacks are inherently dependent on the data representation taken as input by the model. When working with GNNs, attackers thus need to consider the graph representation of the data, leading to adversarial attacks specifically designed for graph-based detection systems. Literature presents numerous papers that apply such attacks to GNN classifiers by either modifying node and edge features, or by directly manipulating the graph structure with actions such as removing or adding nodes and edges [180–182]. In the case of malware, removing nodes or edges from graph structures such as FCG or CFG is not appropriate as it would not preserve the functionality of the program. The adversarial attack should also be efficient on graph classification tasks, as a large majority of works leverage this task for malware detection.

Two such adversarial approaches specifically designed against call graph-based malware detection are proposed by Xu et al. in MANIS [183]. The first method aims to pick the n-strongest nodes from the graph, which are the nodes that have the most influence over their neighbor nodes. They are then inserted in the input graph until evasion has succeeded. The second method relies on the direction of the gradient to guide the insertion of new nodes. The advantage of these methods is that they produce a valid binary that preserves the given format (e.g. PE or APK). On the Drebin dataset, 72.2% misclassification rate is achieved with the n-strongest nodes method, whereas the gradient-based proposition reaches 33.4% misclassification rate under the white-box setting. Similar results are also obtained in gray-box setting.

In the paper [184], authors propose a structural attack for APK-based FCGs that aims to address the inverse mapping problem [166], that consists in retrieving a valid malware in problem-space from the modified malware in feature-space (see Section 8.2). The proposed method works in white-box setting, and leverages reinforcement learning along with heuristic optimization to perform graph modifications such as inserting and deleting nodes, or adding edges and rewiring. The performance of this solution has been evaluated on 30k APKs from Androzoo with over 90% attack success rate in feature space and up to 100% attack success rate in problem space.

Another adversarial method based on reinforcement learning is introduced in reference [185] to evade GNN-based malware detection from CFGs. A deep RL agent is trained to insert semantic NOPs (no-operations) in CFG basic blocks extracted from PE malware. This technique has the faculty to preserve the semantic and format of the original file, while evading GNN classifiers in black-box setting with nearly 100% attack success rate on CFGs constructed with Angr [103] from samples collected on VirusShare and from the VXHeavens dataset [186].

An adversarial attack for GNN-based APK malware detection has been introduced in the work [46] to measure the robustness of the proposed detection model. The attack is based on a VGAE, that aims to effectively add nodes and edges to a FCG in order to fool the GNN classifier, in a

black-box setting. This adversarial approach has been applied to the original GNN model to further improve the robustness of the detection system.

9 CHALLENGES AND DIRECTIONS

Graph representation learning has only recently been applied to malware detection. Therefore, there are still many challenges to achieve resilient malware detection methods. Consequently, we provide some future directions that could improve research in this area:

- While many papers reach good performance on malware detection using graph representation learning techniques, these models are usually evaluated on distinct examples. Indeed, some popular datasets exist, but they are often supplemented by additional samples extracted from public repositories such as Google Play Store, VirusShare and VirusTotal. These new samples make the comparison between papers inefficient as training and testing steps are not performed on the same malware examples, thus leading to different performance evaluations. We think that a large and diversified baseline dataset would be needed for future work, with the aim to effectively compare the metrics of different models.
- The robustness of current approaches based on GNNs is uncertain. Most current works rely solely on the code extracted from APKs using static analysis. However, detecting obfuscated malware by only using its code is a challenging task [187]. Additional efforts using hybrid approaches on graphs could improve the robustness of these techniques, but this remains a scarcely explored direction.
  Furthermore, attackers may try to bypass the detection capabilities of the model by leveraging adversarial attacks. However, defenses against these attacks remains little studied in the field of GNNs and even less when applied to graph-based malware detection. Existing adversarial approaches presented in Section 8.3 have proven remarkable results in fooling the predictions of GNN-based classifiers even in black-box scenarios, meaning that important efforts are still necessary to obtain robust methods.
- One of the drawbacks of using deep models is that they are not amenable to interpretability since they function as black boxes. However, understanding the reasons of a predictive model is of main importance, especially in the field of cybersecurity, where analysts should be able to understand the security-related decisions taken by algorithms. Explainability techniques currently exist to provide insights on the predictions performed by deep architectures such as GNNs [139]. However, very few works leverage these techniques to further improve the explainability of malware predictions with GNNs [17] and further research in this direction could be very useful to the fields of malware detection and analysis.

Furthermore, widely used GNN architectures may not be optimized for the particular task of malware detection, as these models were not specifically designed for this purpose. This means that significant research work could be undertaken to discover new GNN models dedicated to the representation of malware.

10 CONCLUSION

In this paper, we provide an in-depth review of graph representation learning techniques applied to the detection of Android, Windows and Web malware. We first introduced fundamental knowledge to understand graph-based learning methods along with the graph structures commonly employed in malware detection. We reviewed and classified state-of-the-art works in a comprehensive way and provide descriptions and insights on the datasets that can be leveraged to represent malware as graphs. We notably found that most existing techniques can be represented under a same architecture based on graph classification, which is presented and used as reference in this survey.

We also discovered that many recent works prefer leveraging GNNs in combination with word embedding techniques to learn the semantic of disassembled code along with the structural patterns of existing malware. This survey also shows that effective adversarial attacks can be used by attackers in an attempt to fool graph-based detection systems. The analysis of recent papers demonstrates the promising future of graph ML methods applied to malware detection, and as a result, we have provided future research directions based on the current challenges that can be addressed.

REFERENCES

[1] Ulrich Bayer, Andreas Moser, Christopher Kruegel, and Engin Kirda. Dynamic analysis of malicious code. *Journal in Computer Virology, Springer*, 2006.

[2] Yanfang Ye, Tao Li, Donald Adjeroh, and S Sitharama Iyengar. A survey on malware detection using data mining techniques. *ACM Computing Surveys (CSUR), ACM New York, NY, USA*, 2017.

[3] Ömer Aslan Aslan and Refik Samet. A comprehensive review on malware detection approaches. *IEEE Access*, 2020.

[4] Alexander Küchler, Alessandro Mantovani, Yufei Han, Leyla Bilge, and Davide Balzarotti. Does every second count? time-based evolution of malware behavior in sandboxes. *NDSS*, 2021.

[5] Kaijun Liu, Shengwei Xu, Guoai Xu, Miao Zhang, Dawei Sun, and Haifeng Liu. A review of android malware detection approaches based on machine learning. *IEEE Access*, 2020.

[6] Jie Zhou, Ganqu Cui, Shengding Hu, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. Graph neural networks: A review of methods and applications. *AI Open, Elsevier*, 2020.

[7] Abdelmonim Naway and Yuancheng Li. A review on the use of deep learning in android malware detection. *arXiv preprint arXiv:1812.10360*, 2018.

[8] Junyang Qiu, Jun Zhang, Wei Luo, Lei Pan, Surya Nepal, and Yang Xiang. A survey of android malware detection with deep neural models. *ACM Computing Surveys (CSUR),ACM New York, NY, USA*, 2020.

[9] Chao Yang, Zhaoyan Xu, Guofei Gu, Vinod Yegneswaran, and Phillip Porras. Droidminer: Automated mining and characterization of fine-grained malicious behaviors in android applications. *European symposium on research in computer security, Springer*, 2014.

[10] Mehmet Ali Atici, Seref Sagiroglu, and Ibrahim Alper Dogru. Android malware analysis approach based on control flow graphs and machine learning algorithms. *2016 4th International Symposium on Digital Forensic and Security (ISDFS)*, 2016.

[11] Shifu Hou, Yanfang Ye, Yangqiu Song, and Melih Abdulhayoglu. Make evasion harder: an intelligent android malware detection system. *IJCAI*, 2018.

[12] Shifu Hou, Yanfang Ye, Yangqiu Song, and Melih Abdulhayoglu. Hindroid: An intelligent android malware detection system based on structured heterogeneous information network. *Proceedings of the 23rd ACM SIGKDD international conference on knowledge discovery and data mining*, 2017.

[13] Zhiqiang Wang, Qian Liu, and Yaping Chi. Review of android malware detection based on deep learning. *IEEE Access*, 2020.

[14] Jagsir Singh and Jaswinder Singh. A survey on machine learning-based malware detection in executable files. *Journal of Systems Architecture, Elsevier*, 2021.

[15] Muhammad Usman, Mian Ahmad Jan, Xiangjian He, and Jinjun Chen. A survey on representation learning efforts in cybersecurity domain. *ACM Computing Surveys (CSUR), ACM New York, NY, USA*, 2019.

[16] M Gopinath and Sibi Chakkaravarthy Sethuraman. A comprehensive survey on deep learning based malware detection techniques. *Computer Science Review, Elsevier*, 2023.

[17] Dana Warmsley, Alex Waagen, Jiejun Xu, Zhining Liu, and Hanghang Tong. A survey of explainable graph neural networks for cyber malware analysis. *2022 IEEE International Conference on Big Data (Big Data)*, 2022.

[18] Ziwei Zhang, Peng Cui, and Wenwu Zhu. Deep learning on graphs: A survey. *IEEE Transactions on Knowledge and Data Engineering, IEEE*, 2020.

[19] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and S Yu Philip. A comprehensive survey on graph neural networks. *IEEE transactions on neural networks and learning systems, IEEE*, 2020.

[20] Fenxiao Chen, Yun-Cheng Wang, Bin Wang, and C-C Jay Kuo. Graph representation learning: a survey. *APSIPA Transactions on Signal and Information Processing, Cambridge University Press*, 2020.

[21] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. *Advances in neural information processing systems*, 2013.

[22] Pascal Vincent, Hugo Larochelle, Isabelle Lajoie, Yoshua Bengio, Pierre-Antoine Manzagol, and Léon Bottou. Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion. *Journal*

of machine learning research, 2010.
[23] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016.
[24] Yann LeCun, Yoshua Bengio, et al. Convolutional networks for images, speech, and time series. *The handbook of brain theory and neural networks, Cambridge, MA USA*, 1995.
[25] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation, MIT Press*, 1997.
[26] William L Hamilton, Rex Ying, and Jure Leskovec. Representation learning on graphs: Methods and applications. *arXiv preprint arXiv:1709.05584*, 2017.
[27] Jiawei Zhou, Zhiying Xu, Alexander M Rush, and Minlan Yu. Automating botnet detection with graph neural networks. *arXiv preprint arXiv:2003.06344*, 2020.
[28] Justin Carpenter, Janet Layne, Edoardo Serra, and Alfredo Cuzzocrea. Detecting botnet nodes via structural node representation learning. *2021 IEEE International Conference on Big Data (Big Data)*, 2021.
[29] Jun Zhao, Xudong Liu, Qiben Yan, Bo Li, Minglai Shao, and Hao Peng. Multi-attributed heterogeneous graph convolutional network for bot detection. *Information Sciences, Elsevier*, 2020.
[30] Benjamin Bowman, Craig Laprade, Yuede Ji, and H Howie Huang. Detecting lateral movement in enterprise computer networks with unsupervised graph {AI}. *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*, 2020.
[31] Isaiah J King and H Howie Huang. Euler: Detecting network lateral movement via scalable temporal link prediction.
[32] Mohammad Reza Norouzian, Peng Xu, Claudia Eckert, and Apostolis Zarras. Hybroid: Toward android malware detection and categorization with program code and network traffic. *International Conference on Information Security*, 2021.
[33] Peng Xu, Claudia Eckert, and Apostolis Zarras. hybrid-flacon: Hybrid pattern malware detection and categorization with network traffic andprogram code. *arXiv preprint arXiv:2112.10035*, 2021.
[34] Jeffrey Fairbanks, Andres Orbe, Christine Patterson, Janet Layne, Edoardo Serra, and Marion Scheepers. Identifying att&ck tactics in android malware control flow graph through graph representation learning and interpretability. *2021 IEEE International Conference on Big Data (Big Data)*, 2021.
[35] Rui Zhu, Chenglin Li, Di Niu, Hongwen Zhang, and Husam Kinawi. Android malware detection using large-scale network representation learning. *arXiv preprint arXiv:1806.04847*, 2018.
[36] Pengbin Feng, Jianfeng Ma, Teng Li, Xindi Ma, Ning Xi, and Di Lu. Android malware detection based on call graph via graph neural network. *2020 International Conference on Networking and Network Applications (NaNA)*, 2020.
[37] Minghui Cai, Yuan Jiang, Cuiying Gao, Heng Li, and Wei Yuan. Learning features from enhanced function call graphs for android malware detection. *Neurocomputing, Elsevier*, 2021.
[38] Peng Xu, Claudia Eckert, and Apostolis Zarras. Detecting and categorizing android malware with graph neural networks. *Proceedings of the 36th annual ACM symposium on applied computing*, 2021.
[39] KV Vinayaka and CD Jaidhar. Android malware detection using function call graph with graph convolutional networks. *2021 2nd International Conference on Secure Cyber Computing and Communications (ICSCCC)*, 2021.
[40] Federico Errica, Giacomo Iadarola, Fabio Martinelli, Francesco Mercaldo, and Alessio Micheli. Robust malware classification via deep graph networks on call graph topologies. *ESANN*, 2021.
[41] Cagatay Catal, Hakan Gunduz, and Alper Ozcan. Malware detection based on graph attention networks for intelligent transportation systems. *Electronics, MDPI*, 2021.
[42] Yafei Wu, Jian Shi, Peicheng Wang, Dongrui Zeng, and Cong Sun. Deepcatra: Learning flow-and graph-based behaviors for android malware detection. *arXiv preprint arXiv:2201.12876*, 2022.
[43] Wai Weng Lo, Siamak Layeghy, Mohanad Sarhan, Marcus Gallagher, and Marius Portmann. Graph neural network-based android malware classification with jumping knowledge. *CoRR*, 2022.
[44] Hakan Gunduz. Malware detection framework based on graph variational autoencoder extracted embeddings from api-call graphs. *PeerJ Computer Science*, 2022.
[45] Xiaofeng Lu, Jinglun Zhao, and Pietro Lio. Robust android malware detection based on subgraph network and denoising gcn network. *Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services*, 2022.
[46] Rahul Yumlembam, Biju Issac, Seibu Mary Jacob, and Longzhi Yang. Iot-based android malware detection using graph neural network with adversarial defense. *IEEE Internet of Things Journal, IEEE*, 2022.
[47] Peng Xu and Asbat El Khairi. Android-coco: Android malware detection with graph neural network for byte-and native-code. *arXiv preprint arXiv:2112.10038*, 2021.
[48] Teenu S John, Tony Thomas, and Sabu Emmanuel. Graph convolutional networks for android malware detection with system call graphs. *2020 Third ISEA Conference on Security and Privacy (ISEA-ISAP)*, 2020.

[49] Julian Busch, Anton Kocheturov, Volker Tresp, and Thomas Seidl. Nf-gnn: network flow graph neural networks for malware detection and classification. *33rd International Conference on Scientific and Statistical Database Management*, 2021.

[50] Tianyue Liu, Zhenwan Li, Haixia Long, and Anas Bilal. Nt-gnn: Network traffic graph for 5g mobile iot android malware detection. *Electronics, Multidisciplinary Digital Publishing Institute*, 2023.

[51] Jiaqi Yan, Guanhua Yan, and Dong Jin. Classifying malware represented as control flow graphs using deep graph convolutional neural network. *2019 49th annual IEEE/IFIP international conference on dependable systems and networks (DSN)*, 2019.

[52] Peng Xu, Youyi Zhang, Claudia Eckert, and Apostolis Zarras. Hawkeye: cross-platform malware detection with representation learning on graphs. *Artificial Neural Networks and Machine Learning–ICANN 2021: 30th International Conference on Artificial Neural Networks, Bratislava, Slovakia, September 14–17, 2021, Proceedings, Part III 30*, 2021.

[53] Shuai Wang, Yuran Zhao, Gongshen Liu, and Bo Su. A hierarchical graph-based neural network for malware classification. *International Conference on Neural Information Processing*, 2021.

[54] Xiang Ling, Lingfei Wu, Wei Deng, Zhenqing Qu, Jiangyu Zhang, Sheng Zhang, Tengfei Ma, Bin Wang, Chunming Wu, and Shouling Ji. Malgraph: Hierarchical graph neural networks for robust windows malware detection. *IEEE INFOCOM 2022-IEEE Conference on Computer Communications*, 2022.

[55] Haodi Jiang, Turki Turki, and Jason TL Wang. Dlgraph: Malware detection using deep learning and graph embedding. *2018 17th IEEE international conference on machine learning and applications (ICMLA)*, 2018.

[56] Angelo Oliveira and R Sassi. Behavioral malware detection using deep graph convolutional neural networks. *Techlixiv.[link]*, 2019.

[57] Zikai Zhang, Yidong Li, Hairong Dong, Honghao Gao, Yi Jin, and Wei Wang. Spectral-based directed graph network for malware detection. *IEEE Transactions on Network Science and Engineering, IEEE*, 2020.

[58] Shanxi Li, Qingguo Zhou, Rui Zhou, and Qingquan Lv. Intelligent malware detection based on graph convolutional network. *The Journal of Supercomputing, Springer*, 2022.

[59] Ce Li, Zijun Cheng, He Zhu, Leiqi Wang, Qiujian Lv, Yan Wang, Ning Li, and Degang Sun. Dmalnet: Dynamic malware analysis based on api feature engineering and graph learning. *Computers & Security, Elsevier*, 2022.

[60] Nguyen Viet Hung, Pham Ngoc Dung, Tran Nguyen Ngoc, Vu Dinh Phai, and Qi Shi. Malware detection based on directed multi-edge dataflow graph representation and convolutional neural network. *2019 11th International Conference on Knowledge and Systems Engineering (KSE)*, 2019.

[61] Xiao Wang, Houye Ji, Chuan Shi, Bai Wang, Yanfang Ye, Peng Cui, and Philip S Yu. Heterogeneous graph attention network. *The world wide web conference*, 2019.

[62] Chen Liu, Bo Li, Jun Zhao, Ziyang Zhen, Xudong Liu, and Qunshi Zhang. Fewm-hgcl: Few-shot malware variants detection via heterogeneous graph contrastive learning. *IEEE Transactions on Dependable and Secure Computing, IEEE Computer Society*, 2022.

[63] Jerome Dinal Herath, Priti Prabhakar Wakodikar, Ping Yang, and Guanhua Yan. Cfgexplainer: Explaining graph neural network-based malware classification from control flow graphs. *2022 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2022.

[64] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. Deepwalk: Online learning of social representations. *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2014.

[65] Aditya Grover and Jure Leskovec. node2vec: Scalable feature learning for networks. *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*, 2016.

[66] Jian Tang, Meng Qu, Mingzhe Wang, Ming Zhang, Jun Yan, and Qiaozhu Mei. Line: Large-scale information network embedding. *Proceedings of the 24th international conference on world wide web*, 2015.

[67] Petar Velickovic, William Fedus, William L Hamilton, Pietro Liò, Yoshua Bengio, and R Devon Hjelm. Deep graph infomax. *ICLR (Poster)*, 2019.

[68] Marco Gori, Gabriele Monfardini, and Franco Scarselli. A new model for learning in graph domains. *Proceedings. 2005 IEEE international joint conference on neural networks*, 2005.

[69] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE transactions on neural networks, IEEE*, 2008.

[70] Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. Convolutional neural networks on graphs with fast localized spectral filtering. *Advances in neural information processing systems*, 2016.

[71] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

[72] Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. *Advances in neural information processing systems*, 2017.

[73] Muhan Zhang, Zhicheng Cui, Marion Neumann, and Yixin Chen. An end-to-end deep learning architecture for graph classification. *Proceedings of the AAAI conference on artificial intelligence*, 2018.

[74] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. Graph attention networks. *arXiv preprint arXiv:1710.10903*, 2017.
[75] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.
[76] Shiwen Wu, Fei Sun, Wentao Zhang, Xu Xie, and Bin Cui. Graph neural networks in recommender systems: a survey. *ACM Computing Surveys, ACM New York, NY*, 2022.
[77] Weiwei Jiang and Jiayun Luo. Graph neural network for traffic forecasting: A survey. *Expert Systems with Applications, Elsevier*, 2022.
[78] Thomas Gaudelet, Ben Day, Arian R Jamasb, Jyothish Soman, Cristian Regep, Gertrude Liu, Jeremy BR Hayter, Richard Vickers, Charles Roberts, Jian Tang, et al. Utilizing graph machine learning within drug discovery and development. *Briefings in bioinformatics, Oxford University Press*, 2021.
[79] Blake Anderson, Curtis Storlie, and Terran Lane. Improving malware classification: bridging the static/dynamic gap. *Proceedings of the 5th ACM workshop on Security and artificial intelligence*, 2012.
[80] Jusuk Lee, Kyoochang Jeong, and Heejo Lee. Detecting metamorphic malwares using code graphs. *Proceedings of the 2010 ACM symposium on applied computing*, 2010.
[81] Christopher Kruegel, Engin Kirda, Darren Mutz, William Robertson, and Giovanni Vigna. Polymorphic worm detection using structural information of executables. *International Workshop on Recent Advances in Intrusion Detection*, 2006.
[82] Fengguo Wei, Sankardas Roy, and Xinming Ou. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. *ACM Transactions on Privacy and Security (TOPS), ACM New York, NY, USA*, 2018.
[83] Jeanne Ferrante, Karl J Ottenstein, and Joe D Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS), ACM New York, NY, USA*, 1987.
[84] Cuckoo Sandbox Book — Cuckoo Sandbox v2.0.7 Book. *https://cuckoo.readthedocs.io/en/latest/, [Accessed on 01/17/2023]*, 2020.
[85] Meihua Fan, Shudong Li, Weihong Han, Xiaobo Wu, Zhaoquan Gu, and Zhihong Tian. A novel malware detection framework based on weighted heterograph. *Proceedings of the 2020 International Conference on Cyberspace Innovation of Advanced Technologies*, 2020.
[86] Yuzhen Li, Renjie Li, Zhou Zhou, Jiang Guo, Wei Yang, Meijie Du, and Qingyun Liu. Graphddos: Effective ddos attack detection using graph neural networks. *2022 IEEE 25th International Conference on Computer Supported Cooperative Work in Design (CSCWD)*, 2022.
[87] Wenxuan He, Gaopeng Gou, Cuicui Kang, Chang Liu, Zhen Li, and Gang Xiong. Malicious domain detection via domain relationship and graph models. *2019 IEEE 38th International Performance Computing and Communications Conference (IPCCC)*, 2019.
[88] Xiaoqing Sun, Zhiliang Wang, Jiahai Yang, and Xinran Liu. Deepdom: Malicious domain detection with scalable and heterogeneous graph convolutional networks. *Computers & Security, Elsevier*, 2020.
[89] Shuai Zhang, Zhou Zhou, Da Li, Youbing Zhong, Qingyun Liu, Wei Yang, and Shu Li. Attributed heterogeneous graph neural network for malicious domain detection. *2021 IEEE 24th International Conference on Computer Supported Cooperative Work in Design (CSCWD)*, 2021.
[90] Linshu Ouyang and Yongzheng Zhang. Phishing web page detection with html-level graph neural network. *2021 IEEE 20th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, 2021.
[91] Tristan Bilot, Grégoire Geis, and Badis Hammi. Phishgnn: A phishing website detection framework using graph neural networks. *Proceedings of the 19th International Conference on Security and Cryptography - Volume 1: SECRYPT, SciTePress, INSTICC*, 2022.
[92] Zhonglin Liu, Yong Fang, Cheng Huang, and Jiaxuan Han. Graphxss: an efficient xss payload detection approach based on graph convolutional network. *Computers & Security, Elsevier*, 2022.
[93] Yong Fang, Chaoyi Huang, Minchuan Zeng, Zhiying Zhao, and Cheng Huang. Jstrong: Malicious javascript detection based on code semantic representation and graph neural network. *Computers & Security, Elsevier*, 2022.
[94] Han Gao, Shaoyin Cheng, and Weiming Zhang. Gdroid: Android malware detection and classification with graph convolutional network. *Computers & Security, Elsevier*, 2021.
[95] Yiming Hei, Renyu Yang, Hao Peng, Lihong Wang, Xiaolin Xu, Jianwei Liu, Hong Liu, Jie Xu, and Lichao Sun. Hawk: Rapid android malware detection through heterogeneous graph attention networks. *IEEE Transactions on Neural Networks and Learning Systems, IEEE*, 2021.
[96] Welcome to Androguard's documentation! — Androguard 3.4.0 documentation. *https://androguard.readthedocs.io/en/latest/, [Accessed on 01/17/2023]*, 2018.
[97] radare. *https://rada.re/n/, [Accessed on 01/17/2023]*, 2023.
[98] Hex Rays State of-the-art binary code analysis solutions. *https://hex-rays.com/ida-pro, [Accessed on 01/11/2023]*, 2023.
[99] Ghidra. *https://ghidra-sre.org/, [Accessed on 01/17/2023]*, 2023.

[100] binary Android apps. Apktool A tool for reverse engineering 3rd party, closed. *https://ibotpeaches.github.io/Apktool, [Accessed on 01/11/2023]*, 2022.
[101] Djack1010/graph4apk. *https://github.com/Djack1010/graph4apk, [Accessed on 01/17/2023]*, 2021.
[102] Main Page WalaWiki. *https://wala.sourceforge.net/wiki/index.php/Main_Page, [Accessed on 01/12/2023]*, 2019.
[103] angr. *https://angr.io/, [Accessed on 01/17/2023]*, 2022.
[104] strace. *https://strace.io/, [Accessed on 01/17/2023]*, 2022.
[105] HomePage Systemtap Wiki. *https://sourceware.org/systemtap/wiki/, [Accessed on 02/10/2023]*, 2022.
[106] ANY.RUN Interactive Online Malware Sandbox. *https://any.run/, [Accessed on 02/10/2023]*, 2023.
[107] openargus Home. *https://openargus.org, [Accessed on 12/11/2022]*, 2022.
[108] zeek/zeek: Zeek is a powerful network analysis framework that is much different from the typical IDS you may know. *https://github.com/zeek/zeek, [Accessed on 12/11/2022]*, 2023.
[109] Splunk | The Key to Enterprise Resilience. *https://www.splunk.com, [Accessed on 12/11/2022]*, 2023.
[110] cisco/joy: A package for capturing, analyzing network flow data, intraflow data for network research forensics, and security monitoring. *https://github.com/cisco/joy, [Accessed on 12/11/2022]*, 2019.
[111] Hanjun Dai, Bo Dai, and Le Song. Discriminative embeddings of latent variable models for structured data. *International conference on machine learning*, 2016.
[112] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, Konrad Rieck, and CERT Siemens. Drebin: Effective and explainable detection of android malware in your pocket. *Ndss*, 2014.
[113] Isabelle Guyon, Jason Weston, Stephen Barnhill, and Vladimir Vapnik. Gene selection for cancer classification using support vector machines. *Machine learning, Springer*, 2002.
[114] Janet Layne and Edoardo Serra. Inferential sir-gn: Scalable graph representation learning. *arXiv preprint arXiv:2111.04826*, 2021.
[115] Scott M Lundberg and Su-In Lee. A unified approach to interpreting model predictions. *Advances in neural information processing systems*, 2017.
[116] Kevin Allix, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. Androzoo: Collecting millions of android apps for the research community. *Proceedings of the 13th international conference on mining software repositories*, 2016.
[117] Jian Du, Shanghang Zhang, Guanhang Wu, José MF Moura, and Soummya Kar. Topology adaptive graph convolutional networks. *arXiv preprint arXiv:1710.10370*, 2017.
[118] Davide Bacciu, Federico Errica, and Alessio Micheli. Contextual graph markov model: A deep and generative approach to graph processing. *International Conference on Machine Learning*, 2018.
[119] CVE CVE. *https://cve.mitre.org, [Accessed on 01/12/2023]*, 2023.
[120] Exploit Database Exploits for Penetration Testers Researchers and Ethical Hackers. *https://www.exploit-db.com/, [Accessed on 01/12/2023]*, 2023.
[121] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. *Stanford InfoLab*, 1999.
[122] Keyulu Xu, Chengtao Li, Yonglong Tian, Tomohiro Sonobe, Ken-ichi Kawarabayashi, and Stefanie Jegelka. Representation learning on graphs with jumping knowledge networks. *International conference on machine learning*, 2018.
[123] Scott Freitas, Yuxiao Dong, Joshua Neil, and Duen Horng Chau. A large-scale database for graph representation learning. *arXiv preprint arXiv:2011.07682*, 2020.
[124] Thomas N Kipf and Max Welling. Variational graph auto-encoders. *arXiv preprint arXiv:1611.07308*, 2016.
[125] Java optimizer Guardsquare/proguard: ProGuard and obfuscator. *https://github.com/Guardsquare/proguard, [Accessed on 01/26/2023]*, 2022.
[126] Chuan Shi, Yitong Li, Jiawei Zhang, Yizhou Sun, and S Yu Philip. A survey of heterogeneous information network analysis. *IEEE Transactions on Knowledge and Data Engineering, IEEE*, 2016.
[127] Fengguo Wei, Yuping Li, Sankardas Roy, Xinming Ou, and Wu Zhou. Deep ground truth analysis of current android malware. *Detection of Intrusions and Malware, and Vulnerability Assessment: 14th International Conference, DIMVA 2017, Bonn, Germany, July 6-7, 2017, Proceedings 14*, 2017.
[128] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016.
[129] Lukas Ruff, Robert Vandermeulen, Nico Goernitz, Lucas Deecke, Shoaib Ahmed Siddiqui, Alexander Binder, Emmanuel Müller, and Marius Kloft. Deep one-class classification. *International conference on machine learning, PMLR*, 2018.
[130] Arash Habibi Lashkari, Andi Fitriah A Kadir, Laya Taheri, and Ali A Ghorbani. Toward developing a systematic approach to generate benchmark android malware datasets and classification. *2018 International Carnahan Conference on Security Technology (ICCST)*, 2018.
[131] Applications | Research | Canadian Institute for Cybersecurity | UNB. *https://www.unb.ca/cic/research/applications.html, [Accessed on 02/07/2023]*, 2017.

[132] Samaneh Mahdavifar, Andi Fitriah Abdul Kadir, Rasool Fatemi, Dima Alhadidi, and Ali A Ghorbani. Dynamic android malware category classification using semi-supervised deep learning. *2020 IEEE Intl Conf on Dependable, Autonomic and Secure Computing, Intl Conf on Pervasive Intelligence and Computing, Intl Conf on Cloud and Big Data Computing, Intl Conf on Cyber Science and Technology Congress (DASC/PiCom/CBDCom/CyberSciTech)*, 2020.
[133] Hugo Gascon, Fabian Yamaguchi, Daniel Arp, and Konrad Rieck. Structural detection of android malware using embedded call graphs. *Proceedings of the 2013 ACM workshop on Artificial intelligence and security*, 2013.
[134] MalNet. *https://mal-net.org/explore, [Accessed on 01/30/2023]*, 2022.
[135] Katja Hahn and I Register. Robust static analysis of portable executable malware. *HTWK Leipzig*, 2014.
[136] Royi Ronen, Marian Radu, Corina Feuerstein, Elad Yom-Tov, and Mansour Ahmadi. Microsoft malware classification challenge. *arXiv preprint arXiv:1802.10135*, 2018.
[137] Guanhua Yan. Be sensitive to your errors: Chaining neyman-pearson criteria for automated malware classification. *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, 2015.
[138] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.
[139] Zhitao Ying, Dylan Bourgeois, Jiaxuan You, Marinka Zitnik, and Jure Leskovec. Gnnexplainer: Generating explanations for graph neural networks. *Advances in neural information processing systems*, 2019.
[140] Hao Yuan, Haiyang Yu, Jie Wang, Kang Li, and Shuiwang Ji. On explainability of graph neural networks via subgraph explorations. *International Conference on Machine Learning*, 2021.
[141] Dongsheng Luo, Wei Cheng, Dongkuan Xu, Wenchao Yu, Bo Zong, Haifeng Chen, and Xiang Zhang. Parameterized explainer for graph neural network. *Advances in neural information processing systems*, 2020.
[142] Angelo Oliveira. Malware analysis datasets: Api call sequences, ieee dataport. *https://dx.doi.org/10.21227/tqqm-aq14*, 2019.
[143] Alibaba Cloud Malware Detection Based On Behaviors. *https://tianchi.aliyun.com/competition/entrance/231694/introduction, [Accessed on 14/07/2023]*, 2018.
[144] Cătălina Cangea, Petar Veličković, Nikola Jovanović, Thomas Kipf, and Pietro Liò. Towards sparse hierarchical graph classifiers. *arXiv preprint arXiv:1811.01287*, 2018.
[145] Hongyang Gao and Shuiwang Ji. Graph u-nets. *international conference on machine learning*, 2019.
[146] Jane Bromley, Isabelle Guyon, Yann LeCun, Eduard Säckinger, and Roopak Shah. Signature verification using a" siamese" time delay neural network. *Advances in neural information processing systems*, 1993.
[147] Kilian Weinberger, Anirban Dasgupta, John Langford, Alex Smola, and Josh Attenberg. Feature hashing for large scale multitask learning. *Proceedings of the 26th annual international conference on machine learning*, 2009.
[148] VirusShare.com. *https://virusshare.com/, [Accessed on 01/26/2023]*, 2023.
[149] VirusTotal – Home. *https://virustotal.com/, [Accessed on 01/26/2023]*, 2023.
[150] HTML elements HTML5. *https://www.w3.org/TR/2012/WD-html-markup-20121025/elements.html, [Accessed on 01/10/2023]*, 2012.
[151] Song Li, Mingqing Kang, Jianwei Hou, and Yinzhi Cao. Mining node. js vulnerabilities via object dependence graph and query. *31st USENIX Security Symposium (USENIX Security 22)*, 2022.
[152] PhishTank | Join the fight against phishing. *https://phishtank.org/, [Accessed on 02/06/2023]*, 2023.
[153] OpenPhish Phishing Intelligence. *https://openphish.com/, [Accessed on 02/06/2023]*, 2023.
[154] Victor Le Pochat, Tom Van Goethem, Samaneh Tajalizadehkhoob, Maciej Korczyński, and Wouter Joosen. Tranco: A research-oriented top sites ranking hardened against manipulation. *arXiv preprint arXiv:1806.01156*, 2018.
[155] Alexa Top 1 Million Sites | Kaggle. *https://www.kaggle.com/datasets/cheedcheed/top1m, [Accessed on 02/06/2023]*, 2018.
[156] XSSed | Cross Site Scripting (XSS) attacks information and archive. *http://www.xssed.com/, [Accessed on 02/06/2023]*, 2012.
[157] HynekPetrak/javascript malware-collection: Collection of almost 40.000 javascript malware samples. *https://github.com/HynekPetrak/javascript-malware-collection, [Accessed on 02/06/2023]*, 2019.
[158] geeksonsecurity/js-malicious-dataset: This repository contains a list of pseudo-sorted malicious JavaScripts collected from time to time. *https://github.com/geeksonsecurity/js-malicious-dataset, [Accessed on 02/06/2023]*, 2019.
[159] The CAIDA Anonymized Internet Traces Dataset (April 2008 January 2019) CAIDA. *https://kilthub.cmu.edu/articles/dataset/Insider_Threat_Test_Dataset/12841247/1, [Accessed on 02/06/2023]*, 2019.
[160] Sebastian Garcia, Martin Grill, Jan Stiborek, and Alejandro Zunino. An empirical comparison of botnet detection methods. *computers & security, Elsevier*, 2014.
[161] Iman Sharafaldin, Arash Habibi Lashkari, and Ali A Ghorbani. Toward generating a new intrusion detection dataset and intrusion traffic characterization. *ICISSp*, 2018.
[162] Hossein Hadian Jazi, Hugo Gonzalez, Natalia Stakhanova, and Ali A Ghorbani. Detecting http-based application layer dos attacks on web servers in the presence of sampling. *Computer Networks, Elsevier*, 2017.

[163] Naveed Akhtar and Ajmal Mian. Threat of adversarial attacks on deep learning in computer vision: A survey. *IEEE Access*, 2018.
[164] Xiang Ling, Lingfei Wu, Jiangyu Zhang, Zhenqing Qu, Wei Deng, Xiang Chen, Yaguan Qian, Chunming Wu, Shouling Ji, Tianyue Luo, et al. Adversarial attacks against windows pe malware detection: A survey of the state-of-the-art. *Computers & Security, Elsevier*, 2023.
[165] Luca Demetrio, Scott E Coull, Battista Biggio, Giovanni Lagorio, Alessandro Armando, and Fabio Roli. Adversarial exemples: A survey and experimental evaluation of practical attacks on machine learning for windows malware detection. *ACM Transactions on Privacy and Security (TOPS), ACM New York, NY, USA*, 2021.
[166] Fabio Pierazzi, Feargus Pendlebury, Jacopo Cortellazzi, and Lorenzo Cavallaro. Intriguing properties of adversarial ml attacks in the problem space. *2020 IEEE symposium on security and privacy (SP)*, 2020.
[167] Felix Kreuk, Assi Barak, Shir Aviv-Reuven, Moran Baruch, Benny Pinkas, and Joseph Keshet. Deceiving end-to-end deep learning malware detectors using adversarial examples. *arXiv preprint arXiv:1802.04528*, 2018.
[168] Octavian Suciu, Scott E Coull, and Jeffrey Johns. Exploring adversarial examples in malware detection. *2019 IEEE Security and Privacy Workshops (SPW)*, 2019.
[169] Luca Demetrio, Battista Biggio, Giovanni Lagorio, Fabio Roli, and Alessandro Armando. Explaining vulnerabilities of deep learning to adversarial malware binaries. *arXiv preprint arXiv:1901.03583*, 2019.
[170] Lingwei Chen, Yanfang Ye, and Thirimachos Bourlai. Adversarial machine learning in malware detection: Arms race between evasion attack and defense. *2017 European intelligence and security informatics conference (EISIC)*, 2017.
[171] Weiwei Hu and Ying Tan. Black-box attacks against rnn based malware detection algorithms. *Workshops at the Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.
[172] Ishai Rosenberg, Asaf Shabtai, Lior Rokach, and Yuval Elovici. Generic black-box end-to-end attack against state of the art api call based malware classifiers. *Research in Attacks, Intrusions, and Defenses: 21st International Symposium, RAID 2018, Heraklion, Crete, Greece, September 10-12, 2018, Proceedings 21*, 2018.
[173] Masataka Kawai, Kaoru Ota, and Mianxing Dong. Improved malgan: Avoiding malware detector by leaning cleanware features. *2019 international conference on artificial intelligence in information and communication (ICAIIC)*, 2019.
[174] Fenil Fadadu, Anand Handa, Nitesh Kumar, and Sandeep Kumar Shukla. Evading api call sequence based malware classifiers. *Information and Communications Security: 21st International Conference, ICICS 2019, Beijing, China, December 15–17, 2019, Revised Selected Papers 21*, 2020.
[175] Ishai Rosenberg, Asaf Shabtai, Yuval Elovici, and Lior Rokach. Query-efficient black-box attack against sequence-based malware classifiers. *Annual Computer Security Applications Conference*, 2020.
[176] Weiwei Hu and Ying Tan. Generating adversarial malware examples for black-box attacks based on gan. *Data Mining and Big Data: 7th International Conference, DMBD 2022, Beijing, China, November 21–24, 2022, Proceedings, Part II*, 2023.
[177] Hyrum S Anderson, Anant Kharkar, Bobby Filar, and Phil Roth. Evading machine learning malware detection. *black Hat*, 2017.
[178] Cangshuai Wu, Jiangyong Shi, Yuexiang Yang, and Wenhua Li. Enhancing machine learning based malware detection model by reinforcement learning. *Proceedings of the 8th International Conference on Communication and Network Security*, 2018.
[179] Zhiyang Fang, Junfeng Wang, Boya Li, Siqi Wu, Yingjie Zhou, and Haiying Huang. Evading anti-malware engines with deep reinforcement learning. *IEEE Access*, 2019.
[180] Lichao Sun, Yingtong Dou, Carl Yang, Ji Wang, Philip S Yu, Lifang He, and Bo Li. Adversarial attack and defense on graph data: A survey. *arXiv preprint arXiv:1812.10528*, 2018.
[181] Daniel Zügner, Amir Akbarnejad, and Stephan Günnemann. Adversarial attacks on neural networks for graph data. *Proceedings of the 24th ACM SIGKDD international conference on knowledge discovery & data mining*, 2018.
[182] Hanjun Dai, Hui Li, Tian Tian, Xin Huang, Lin Wang, Jun Zhu, and Le Song. Adversarial attack on graph structured data. *International conference on machine learning*, 2018.
[183] Peng Xu, Bojan Kolosnjaji, Claudia Eckert, and Apostolis Zarras. Manis: Evading malware detection system on graph structure. *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, 2020.
[184] Kaifa Zhao, Hao Zhou, Yulin Zhu, Xian Zhan, Kai Zhou, Jianfeng Li, Le Yu, Wei Yuan, and Xiapu Luo. Structural attack against graph based android malware detection. *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021.
[185] Lan Zhang, Peng Liu, Yoonho Choi, and Ping Chen. Semantics-preserving reinforcement learning attack against graph neural networks for malware detection. *IEEE Transactions on Dependable and Secure Computing, IEEE*, 2022.
[186] Ying Tan. *Artificial immune system: applications in computer security.* 2016.
[187] Arini Balakrishnan and Chloe Schulze. Code obfuscation literature survey. *CS701 Construction of compilers*, 2005.