A Survey on Malware Detection with Graph Representation Learning

TRISTAN BILOT, Iriguard, Puteaux, France, Université Paris-Saclay, CNRS, Laboratoire Interdisciplinaire des Sciences du Numérique, Gif-sur-Yvette, France, and LISITE Laboratory, ISEP (Institut Supérieur d'Electronique de Paris), Issy-les-Moulineaux, France
NOUR EL MADHOUN, LISITE Laboratory, ISEP (Institut Supérieur d'Electronique de Paris), Issy-les-Moulineaux, France and Sorbonne Université, CNRS, LIP6, Paris, France
KHALDOUN AL AGHA, Université Paris-Saclay, CNRS, Laboratoire Interdisciplinaire des Sciences du Numérique, Gif-sur-Yvette, France
ANIS ZOUAOUI, Iriguard, Puteaux, 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. Recently, the application of Graph Representation Learning (GRL) techniques on graph-structured data has demonstrated impressive capabilities in malware detection. This success benefits notably from the robust structure of graphs, which are challenging for attackers to alter, and their intrinsic explainability capabilities. 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 such as Function Call Graphs (FCGs) and Control Flow Graphs (CFGs). This study also discusses the robustness of GRL-based methods to adversarial attacks, contrasts their effectiveness with other ML/DL approaches, and outlines future research for practical deployment.

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

Authors' Contact Information: Tristan Bilot, Iriguard, Puteaux, France, Université Paris-Saclay, CNRS, Laboratoire Interdisciplinaire des Sciences du Numérique, Gif-sur-Yvette, France, and LISITE Laboratory, ISEP (Institut Supérieur d'Electronique de Paris), Issy-les-Moulineaux, France; e-mail: tristan.bilot@universite-paris-saclay.fr; Nour El Madhoun, LISITE Laboratory, ISEP (Institut Supérieur d'Electronique de Paris), Issy-les-Moulineaux, France and Sorbonne Université, CNRS, LIP6, Paris, France; e-mail: nour.el-madhoun@isep.fr; Khaldoun Al Agha, Université Paris-Saclay, CNRS, Laboratoire Interdisciplinaire des Sciences du Numérique, Gif-sur-Yvette, France; e-mail: alagha@lisn.fr; Anis Zouaoui, Iriguard, Puteaux, France; e-mail: anis.zouaoui@adservio.fr.

ACM 0360-0300/2024/06-ART278
https://doi.org/10.1145/3664649

ACM Reference Format:
Tristan Bilot, Nour El Madhoun, Khaldoun Al Agha, and Anis Zouaoui. 2024. A Survey on Malware Detection with Graph Representation Learning. *ACM Comput. Surv.* 56, 11, Article 278 (June 2024), 36 pages. https://doi.org/10.1145/3664649

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 or Windows. 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 of 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 (GRL)** 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 GRL 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 GRL, 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 and system entity 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 GRL 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 and 6 review the state-of-the-art papers for the detection of Android and Windows malware, respectively. Section 7 discusses the robustness of GNN-based detection systems against adversarial attacks. In Section 8, we offer a detailed discussion on applying GRL-based techniques to real-world malware detection, including a comparison of their strengths and weaknesses against traditional ML and DL methods. The last Section 9 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] and App-API graphs [10, 11]. 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 [12], 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 [13] 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 [14], with few mentions to malware detection. More recently, the paper [15] also reviewed DL methods applied to the detection of mobile malware, Windows malware, IoT malware, **Advanced Persistent Threats (APTs)** and Ransomware.

Regarding GRL and GNN-based methods, the work [16] surveys GNN techniques employed for malware analysis with a focus on the prediction explainability. Other surveys review the applications of GNNs [6, 17–19] 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 GRL 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 GRL techniques leveraged to learn from these structures. We first discuss the properties of graphs and then explain differences between traditional Deep Learning and GRL, 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 $V = v_1, \ldots, v_N$ is a set of $N = |V|$ nodes (i.e., entities) and $E = e_1, \ldots, e_M$ 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 [20], learning the representation of objects in images [21] or learning translation of language [22]. 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)** [23] and **Recurrent Neural Networks (RNNs)** [24] are unable to perform traditional convolutions or recurrent operations on graphs as the notion of Euclidean distance cannot be applied. **Graph Representation Learning (GRL)** [25], 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 malicious file in a system entities graph [26], whereas edge classification is applied to assign a label to a relation or event, such as detecting a malicious authentication request [27, 28]. 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 [29–57]. It is also possible to work at the subgraph level to detect areas in the graph that are responsible for the prediction done by a predictive model [58].

In literature, the first methods for GRL based on graph embedding are mostly relying on random walks, where the co-occurrence of nodes is preserved. DeepWalk [59] was the first method to leverage the Skip-gram model [20] 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 [60] 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 [61] have also achieved great performance in learning embeddings from graphs. However, most of these techniques do not share parameters between nodes [25], 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 [62]. 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 GRL approaches tend to be inspired from the Graph Neural Network (GNN) model [63][64], which is the origin of the first application of deep neural networks to graph-structured data. Although DL on graphs has been democratized fairly recently, the first GNN [63] dates back to 2005 and is originally inspired from RNNs. In recent years, the popularity of DL 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 [65] 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)** [66], GraphSAGE [67], **Deep Graph Convolutional Neural Network (DGCNN)** [68], and **Graph Attention Network (GAT)** [69] 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.

Graph Convolutional Network (GCN). GCN captures both feature and local substructure information by propagating the information along the neighboring nodes within the graph. The propagation rule of GCN to compute node embeddings is described as:

$$\mathbf{H}^{(l+1)} = \sigma\left(\tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{\frac{-1}{2}}\mathbf{H}^{(l)}\mathbf{W}^{(l)}\right), \tag{1}$$

where $\mathbf{H}^{(l)}$ denotes the node embeddings at layer l and $\mathbf{H}^{(0)} = X$, with X the initial node features matrix. $\tilde{A}$ is the adjacency matrix representation of the graph with self-loops such that $\tilde{A} = A + I$, with I the identity matrix of same shape as the adjacency matrix A. $\tilde{D}$ is the degree matrix of $\tilde{A}$, whereas $\mathbf{W}$ represents a trainable weight matrix and σ is the sigmoid non-linear activation function. The GCN model has inspired the development of numerous other GNN architectures but remains one of the most widely used models in GRL-based malware detection for its capabilities in learning insightful embeddings in transductive tasks [32, 34, 40, 42, 52, 53, 55, 70, 70–72].

Deep Graph Convolutional Neural Network (DGCNN). This model also leverages convolutions on graphs but is specifically designed for graph classification tasks. It computes node embeddings using message passing, following a propagation rule similar to that of GCN, described as follows:

$$\mathbf{H}^{(l+1)} = f\left(\tilde{D}^{-1}\tilde{A}\mathbf{H}^{(l)}\mathbf{W}^{(l)}\right), \tag{2}$$

where f denotes an arbitrary non-linear activation function. Here, the generated node embeddings are sorted using a dedicated SortPooling layer, before feeding them into traditional 1D convolutional and dense layers. The DGCNN model has already been successfully utilized in malware detection due to its graph classification capabilities, where the overall knowledge of the graph is employed to predict the presence of malware [46, 51, 73].

GraphSAGE. GraphSAGE provides an inductive solution that can scale to large graphs by sampling the neighbors before message-passing. In a first time, each node's neighborhood is uniformly sampled to keep a maximum of s neighbors. The features from these sampled neighbors are then aggregated such that:

$$h^{(l+1)}_{\mathcal{N}_s(u)} = \text{AGGREGATE}_{(l+1)}\left(\left\{h^{(l)}_v, \forall v \in \mathcal{N}_s(u)\right\}\right), \tag{3}$$

where $\mathcal{N}_s(u)$ is the sampled neighorhood of node u with size s, and $h^{(l)}_v$ is the embedding of node v at layer l. $\text{AGGREGATE}_{(l+1)}$ denotes a differentiable aggregation function such as the mean, max or sum operation. The aggregated information $h^{(l+1)}_{\mathcal{N}_s(u)}$ of a node u is then leveraged in the following propagation rule, to compute node embeddings:

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

where $\mathbf{W}^{(l+1)}$ denotes a trainable weight matrix and $[,]$ represents the concatenation operation. The powerful scalability capabilities of GraphSAGE make this model especially useful for large graphs. The graphs encountered in malware detection are generally of small or medium size. Consequently, the sampling strategy of GraphSAGE is not frequently employed in the papers reviewed. However, the propagation rule of GraphSAGE has been found to be successful in the malware detection literature, even without the use of neighbor sampling [36, 40, 43].

Graph Attention Network (GAT). The GAT model leverages the attention mechanism [74] to learn an importance weight for each neighboring node. The GAT layer aims to compute an attention score e_{uv} for each connected pair of nodes u and v such that:

$$e_{uv} = \text{LeakyReLU}\left(\mathbf{a}^T \left[\mathbf{W}h_u, \mathbf{W}h_v\right]\right),$$

where [,] is the concatenation operation, $\mathbf{a}$ and $\mathbf{W}$ denote a trainable attention vector and a weight matrix, respectively. The features of nodes u and v are respectively represented here by h_u and h_v. A softmax activation is then applied along the attention scores of all neighbors to obtain normalized probability scores:

$$\alpha_{uv} = \text{softmax}\left(e_{uv}\right) = \frac{\exp\left(e_{uv}\right)}{\sum_{k \in \mathcal{N}_u} \exp\left(e_{uk}\right)},$$

where $\mathcal{N}_u$ is the neighborhood of node u. The overall propagation rule of GAT is derived from the attention coefficients computed to learn the importance of nodes relative to their neighbors. For reasons of training stability, multi-head attention may be utilized to learn K distinct trainable matrices concurrently. Each head learns a unique representation of the node, and all representations are then concatenated into a single node embedding:

$$h'_u = \|_{k=1}^{K} \sigma\left(\sum_{v \in \mathcal{N}(u)} \alpha_{uv}^k \mathbf{W}^k h_v\right),$$

where || represents the concatenation operation and K is the number of attention heads. The attention mechanism used in GAT allows the model to learn complex relationships within the graph. This capability makes the model valuable in malware detection, where identifying complex malicious patterns is essential [38, 48, 54, 57, 75].

Variants of these models have achieved state-of-the-art results in various domains, such as recommender systems [76], traffic forecasting [77], and drug discovery [78]. However, GNN-based methods are still underutilized in cybersecurity compared to other fields where research is predominantly focused in this direction.

4 MALWARE DETECTION WITH GRAPHS

In this section, we describe the problem of malware detection with graphs, along with ways to represent malware as expressive graph structures. We also propose a general methodology to leverage GRL in downstream malware detection tasks.

4.1 Problem Definition

Malware detection refers to the process of identifying and neutralizing malware, designed to infiltrate, damage, or disrupt computer systems without the consent of users. In the field of ML, malware detection first consists in extracting features from an input binary file, which are then leveraged by a downstream learning algorithm for classification. Malware detection with GRL follows the same idea, with the only difference that a supplementary step is introduced after the feature extraction. This step consists in preprocessing the input features by transforming them into a graph structure that will then be fed to the model.

To provide an intuitive understanding of malware detection methodologies, particularly those leveraging GRL, we introduce a detailed example complemented by a visual illustration in Figure 1.

Imagine a scenario where a cybersecurity analyst is tasked with investigating a suspicious file. Initially, the extraction of meaningful information from the file is required. Common approaches typically leverage reverse engineering to perform static and dynamic analysis as a way to extract

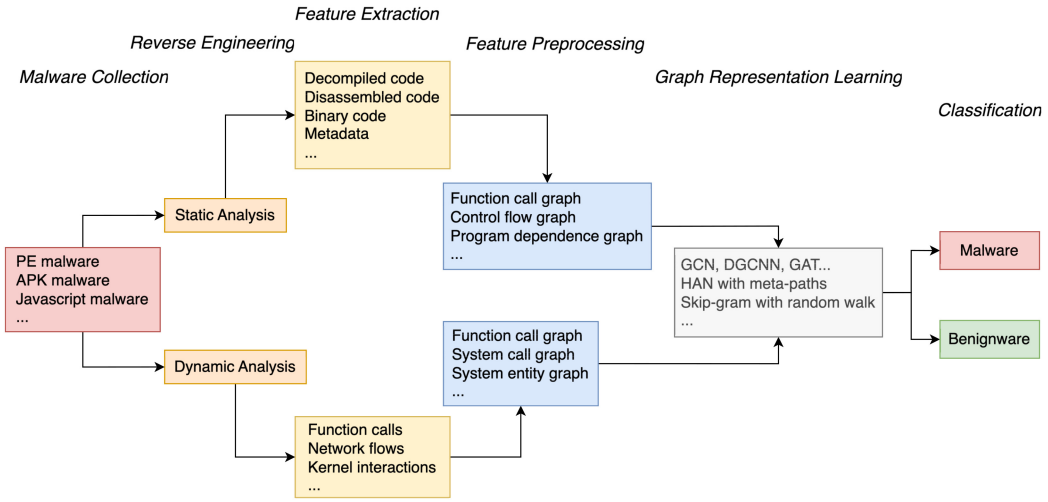

Fig. 1. Methodology of malware detection with GRL-based techniques.

features from the file. This file, when executed in a dynamic analysis scenario, exhibits a sequence of operations such as file accesses, network communications, and registry modifications. Each of these operations can be represented as a node within a graph, while the interactions between them are the edges. Alternatively, parsing the decompiled code during static analysis can be a way to build a graph where nodes are functions and edges represent function calls.

The analyst uses GRL techniques to process this graph. A learning method such as a GNN or a Skip-gram model with random walks processes the node and edge embeddings, learning the patterns that are indicative of either benign or malicious behavior. The model takes into account the rich structure of the graph and learns from the topology and the attributes of nodes and edges. For example, a common benign behavior might be a file creating a temporary backup before an update. This creates a predictable and harmless pattern within the graph. In contrast, a malware file might attempt to replicate itself or communicate with an external server for command and control purposes, leading to a different pattern, potentially spread across various parts of the graph.

In the remainder of this section, we provide a comprehensive description of the steps comprising GRL-based malware detection.

4.2 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.

Table 1. Common Tools Employed for Data Extraction and Graph Construction with Static or Dynamic Analysis

Representation	Tools
CFG	Androguard [85], radare2 [86], IDA Pro [87], Ghidra [88]
FCG	Androguard, Apktool [89], graph4apk [90], WALA [91], Angr [92], radare2, IDA Pro, cuckoo [83], Ghidra
PDG	Androguard, Ghidra
Syscalls	strace [93], SystemTap [94], ltrace [94]
System entities	cuckoo, Any.Run [95]

Green refers to tools specifically designed for Android APKs.

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.

4.2.1 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.

4.2.2 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, 80]. The various types of graphs used for malware detection are presented below, with the main tools used to create these graphs listed in Table 1.

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 [46]. Although other representations such as hexadecimal also possess this characteristic, CFGs provide a more intuitive way to model programs as graphs [46, 58].

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

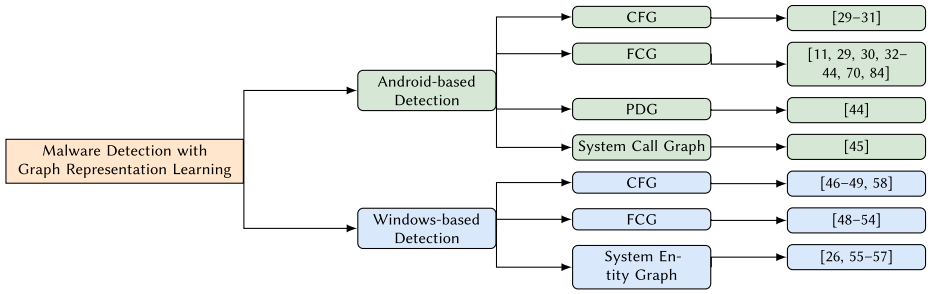

Fig. 2. Categorization of current state-of-the-art papers in malware detection with graph representation learning. In this survey, we classify papers by platform and by input data structure. Android-based detection is presented in Section 5, whereas Windows-based detection is presented in Section 6.

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 [29, 30, 48]. 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 [51, 53, 54, 81].

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 [82]. Scarcely used by current GRL approaches, this graph structure may be promising to capture different conditional flows in the program [44].

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 [45] 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 [83] 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 [26, 55].

In this survey, we focus on the analysis of malware detection methods with GRL for Android and Windows 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 state-of-the-art into sections based on the input graph structure. The literature review is summarized in Figure 2.

4.3 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 GRL. In this section, we propose a general architecture, shown in Figure 3, 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

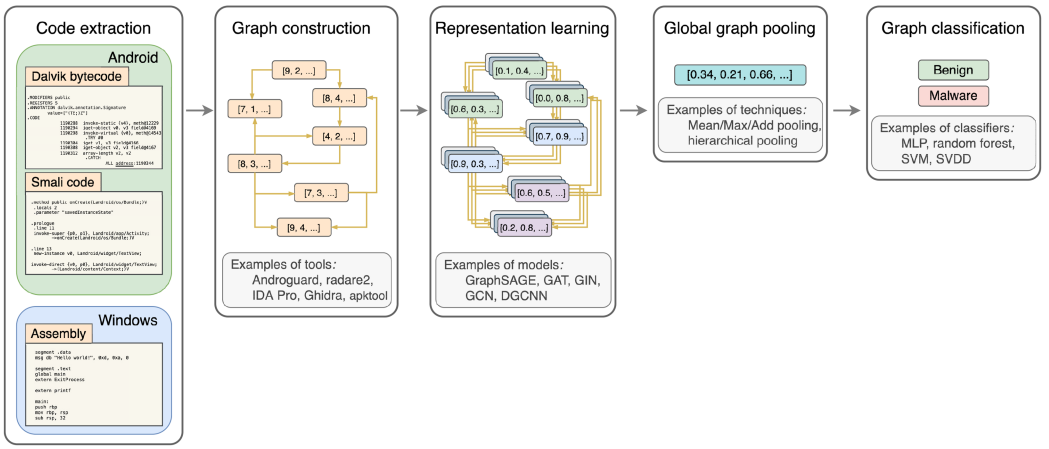

Fig. 3. General architecture of malware detection from static code analysis based on GRL. The code extracted from executable files is used to build an input graph such as a FCG or a CFG, where nodes and edges can be attributed with features. A GRL-based model learns node embeddings though representation learning by leveraging the graph topology along with the integrated features. All node embeddings are pooled into a single graph embedding containing the overall graph information. This vector is given as input to a classifier for prediction.

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 4.2.2. Typically, these first two steps are performed using reverse engineering tools listed in Table 7. Optionally, the graph can be preprocessed and attributed with hand-crafted features, located on nodes or edges. Then, GRL 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.

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 GRL 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

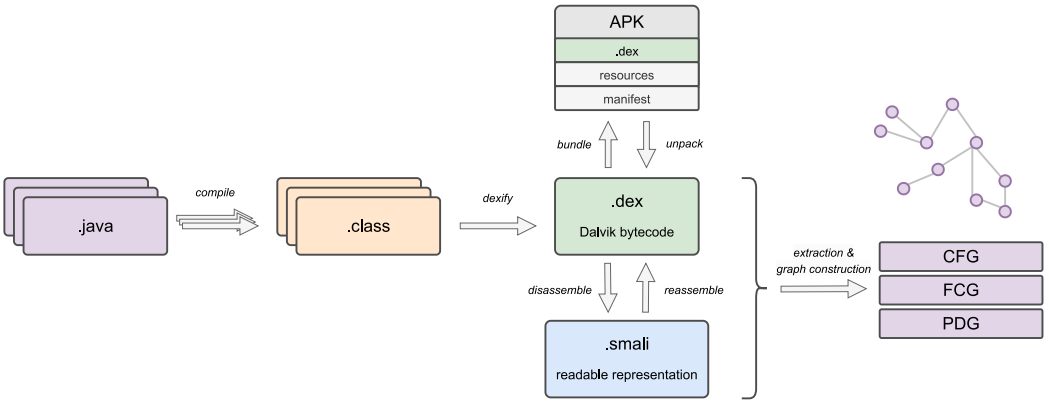

Fig. 4. Android application compilation and disassembling process using static analysis. Java files are compiled into classes and are assembled into a single dex file. The APK is packed with the dex file, the manifest file and other resources. The dex bytecode can be disassembled into higher-level smali code and graph structures can be constructed from either dex or smali code depending on the use case requirements.

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 Figure 4. Concerning dynamic analysis, API call and system call sequences can be recorded when running the app in a sandboxed environment.

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 [29] 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 [96] 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 [97]. The packets are transformed into flows to summarize the communication between the Android device and the destination IP

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 [29]
	Hybrid	Attributed	Graph	Supervised	Bi-LSTM, word2vec, structure2vec	2021	hybrid-Falcon [30]
CFG	Hybrid	Attributed	Graph, Subgraph	Supervised	Inferential SIR_GN, RF	2021	Fairbanks et al. [31]
FCG	Static	Attributed	Graph	Supervised	GCN	2018	CG-GCN [32]
					GNN	2021	CGDroid [33]
					GCN, CBOW	2021	Cai et al. [34]
					GNN, Skip-gram	2021	Xu et al. [35]
					GraphSAGE	2021	Vinayaka et al. [36]
					CGMM	2021	Errica et al. [37]
					GAT, node2vec	2021	Catal et al. [38]
					GNN, Bi-LSTM, TF-IDF	2022	DeepCatra [39]
					GCN, GraphSAGE, GIN	2022	Lo et al. [40]
					VGAE, word2vec	2022	Gunduz et al. [41]
					GCN	2022	Lu et al. [42]
					GraphSAGE, VGAE	2022	Yumlembam et al. [43]
App-API FCG	Static	Heterogeneous	Node	Supervised	Multi-kernel model, Meta-path	2017	HinDroid [11]
					GCN, Skip-gram	2021	GDroid [70]
					Custom HAN, Meta-path	2021	Hawk [84]
PDG+FCG	Static	Attributed	Graph	Supervised	structure2vec, word2vec, SIF	2021	Android-COCO [44]
Syscall Graph	Dynamic	Attributed	Graph	Supervised	GCN	2020	John et al. [45]

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.

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 outperforms other methods such as DREBIN [98] and SVM [99].

On the other hand, hybrid-Falcon [30] 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 [31] 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 [100] and the prediction is done using a random forest. To identify the subgraph responsible for the prediction, the authors rely on SHAP [101] 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%.

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 [32], 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 Section 3.2.3. The Drebin dataset is used for final evaluation with a F1-score of 99.68%.

In CGdroid [33], multiple node features are extracted from the disassembled methods in order to build an 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 [102] datasets, where a baseline is outperformed by 8% in F1-score.

Word embedding techniques are employed in [34] 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 an 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 [35] leverage word embedding to transform Android opcodes from text to vectors using Skip-gram. In the same way as [34], 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 is used as last layer for graph classification, with a 99.6% average accuracy.

In the reference [36], 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 [103] 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:

$$\mathbf{h}_{\mathcal{N}(i)}^{(l+1)} = \text{aggregate}\left(\left\{\mathbf{h}_j^{(l)}, \forall j \in \mathcal{N}(i)\right\}\right) \tag{5}$$

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.

$$\mathbf{h}_i^{(l+1)} = \sigma\left(\mathbf{W}^{(l)} \cdot \left[\mathbf{h}_i^{(l)}, \mathbf{h}_{\mathcal{N}(i)}^{(l+1)}\right]\right) \tag{6}$$

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

$$\mathbf{h}_i^{(l+1)} = \frac{\mathbf{h}_i^{(l+1)}}{||\mathbf{h}_i^{(l+1)}||_2} \tag{7}$$

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

The paper [37] focuses on obfuscated malware detection using call graphs attributed with nodes' out-degree, applying the **Contextual Graph Markov Model (CGMM)** [104] to learn embeddings for classification with a feed-forward network, achieving a 97.2% macro F1-score. The authors in [38] explain that the use of node2vec as feature extraction method is justified by a 3% increase in F1-score compared to traditional graph centrality features. Using the attention aggregation from GAT as a final step, the proposed solution reaches 94.1% in F1-score.

DeepCatra [39] identifies critical Android APIs using **Term Frequency-Inverse Document Frequency (TF-IDF)** from call traces, extracted from popular repositories such as CVE [105] and Exploit-DB [106]. It generates call graphs with Wala [91], incorporating knowledge of critical APIs,

and trains a custom GNN and a bi-directional LSTM to capture graph topology and temporal features, respectively. The merged output vectors from both models undergo binary classification, achieving a 95.83% F1-score, surpassing various baselines in performance.

In [40], an enhanced FCG employs graph centrality metrics (PageRank, in/out degree, node betweenness) as node attributes. Comparing GCN, GraphSAGE, and GIN models, all incorporating jumping knowledge [107] to counteract over-smoothing, GraphSAGE shows superior performance on multi-class classification tasks, with notable F1-scores on Malnet-Tiny and Drebin datasets.

Authors in [41] compute node embeddings from API FCGs using word2vec and apply a **Variational Graph Auto-Encoder (VGAE)** [108] for a compact representation, achieving a 93.4% F-measure in downstream classification tasks. [42] introduces a robust detection method against obfuscation using a GCN with subgraphs and a denoising approach, tested on a dataset comprising VirusShare and AndroZoo samples.

An innovative approach is proposed in [43], where each Android app is viewed as a local graph with APIs as nodes, linking APIs co-existing in the same code block. A global graph captures inter-app connections via co-occurring APIs. Their approach combines GraphSAGE embeddings with app manifest features (permissions and intents) using a concatenation operation before classification by a downstream supervised classifier like a CNN. 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.

HinDroid [10, 11] models interactions between Android apps and API calls in an App-API graph as a **Heterogeneous Information Network (HIN)** [109], using meta-paths to extract semantic relationships. These relationships are leveraged by a multi-kernel SVM for app node classification, resulting in a 98.84% F1-score. Similarly, GDroid [70] constructs an App-API graph from API co-occurrence, and encodes APIs with a Skip-gram model, while applying a GCN for node classification.

Hawk [84] constructs a large heterogeneous App-API graph from over 180k APKs, capturing various relationships (App-Permission, App-Class, App-Interface) and using meta-paths and meta-graphs for semantic extraction. Custom heterogeneous GAT models for in-sample and out-sample nodes demonstrate superior malware detection capabilities against several baselines.

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 [82]. For similar reasons, this structure is also employed in malware detection but remains little used with GRL.

Android-COCO [44] 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, an FCG is created, on which graph embeddings are also computed (similar to Hybroid [29]). 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 [110] 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 [45] 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 four 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

Table 3. Datasets Employed in Android Malware Detection Studies

Paper	Datasets	Performance
Hybroid [29]	CICAndMal2017	97% F1
hybrid-Falcon [30]	CICAndMal2017,AndroZoo	97.09% F1
Fairbanks et al. [31]	VirusTotal	92.7% F1
CG-GCN [32]	Drebin+Apkpure+HKUST	99.68% F1
CGDroid [33]	Drebin+AndroZoo	~99% F1
Cai et al. [34]	AndroZoo+VirusShare	99.65% F1
Xu et al. [35]	Drebin+AMD+PRAGuard+AndroZoo	99.6% acc
Vinayaka et al. [36]	CICMalDroid2020,AndroZoo	92.23% F1
Errica et al. [37]	AMD,Google Play Store	97.2% macro F1
Catal et al. [38]	CICMalDroid2020+ISCX-AndroidBot-2015	94.1% F1
DeepCatra [39]	Drebin+DroidAnalytics+VirusShare +CICInvesAndMal2019+AndroZoo	95.83% F1
Lo et al. [40]	MalNet-Tiny,Drebin	94%, 97% F1
Gunduz et al. [41]	ISCX-AndroidBot-2015+CICMalDroid2020	93.4% F1
Lu et al. [42]	VirusShare+AndroZoo+Google Play Store	~63%-95% F1
Yumlembam et al. [43]	Drebin,CICMalDroid2020	98.33%, 98.68% acc
HinDroid [11]	Private	98.84% F1
GDroid [70]	AMD,Google Play Store	98.99% acc
Hawk [84]	CICAndMal2017+VirusShare+AndroZoo +Google Play Store	>96% F1
Android-COCO [44]	Drebin+AMD+AndroZoo	99.88% F1
John et al. [45]	Drebin+AMD+Malgenome	92.3% acc

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 mentioned 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.

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 GRL literature.

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, and some of their statistics are summarized in Table 4. Based on the current information provided from the respective

Table 4. Overview of Malware Detection Datasets

Dataset	Type	Number of Elements	Number of Classes
CICAndMal2017	APK file	10,854	5
CICMalDroid	APK file	17,341	5
AndroZoo	APK file	>21M	/
Drebin	APK file	5,560	179
MalNet	FCG from APK	1,262,024	696

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 [111]. 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 [112]. This tool generates, for each flow, 80 features based on statistics from the packets contained within the flow.

CICMalDroid [113]. 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 five 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 (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 [102]. A collection of Android apps provided by the University of Luxembourg. In 2023, the dataset contains more than 22M 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, nine 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 [98]. 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 [114]. 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 GRL with 1,262,024 graphs, averaging over 15k nodes and 35k edges per graph, across 696 families and divided among 47 types. GNNs have been applied to this dataset in the original paper [114], 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

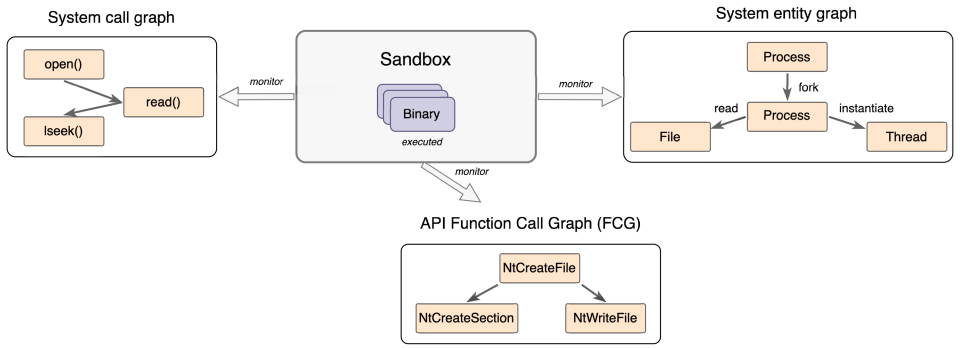

Fig. 5. Extraction of graph structures from a running binary using dynamic analysis. The file is safely executed in a sandbox environment and all system-level events are monitored for downstream graph construction.

nature of malware [50, 115]. For smaller experiments, MalNet-Tiny is a subset of MalNet composed of 5,000 graphs of at most 5k nodes and balanced in five types.

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 GRL 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 [116]. 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 Figure 5.

6.2 Windows-based Approaches

This section reviews the approaches employed in the Windows-based papers presented in Table 5.

6.2.1 CFG Approaches for Windows Malware Detection. In MAGIC [46], 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 4.2.2. Instead of using a standard GCN that was initially made for node classification,

Table 5. 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 [46]
	Hybrid	Attributed	Graph	Supervised	GNN, word2vec	2021	HawkEye [47]
CFG+FCG	Static	Attributed	Graph	Supervised	GAT, Random Walk, BERT	2021	Wang et al. [48]
	Static	Attributed	Graph	Supervised	GraphSAGE	2022	MalGraph [49]
FCG	Static	Attributed	Graph	Supervised	node2vec, SDA	2019	DLGraph [50]
	Dynamic	Attributed	Graph	Supervised	DGCNN	2019	Oliveira et al. [51]
					GCN	2021	SDGNet [52]
					GCN, Markov chain	2021	Li et al. [53]
					GAT, GIN, Word2vec	2022	DMalNet [54]
Entities	Dynamic	Attributed	Graph	Supervised	GCN	2019	MeQDFG [55]
	Dynamic	Heterogeneous	Graph	Semi-supervised	Heterogeneous GNN, Meta-path, Attention, Siamese Network	2019	MatchGNet [56]
	Dynamic	Heterogeneous, Attributed	Node	Supervised	GraphSAGE, Meta-path	2021	MalSage [26]
	Dynamic	Heterogeneous	Graph	Self-supervised	GAT, Meta-path, Contrastive learning	2022	FewM-HGCL [57]

the authors prefer to leverage a **Deep Graph Convolutional Neural Network (DGCNN)** [68], 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 datasets: MSKCFG (inspired by [117]) and YANCFG (inspired by [118]). 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 [117] and reaches 99.25% accuracy.

A cross-platform approach is proposed in HawkEye [47] 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 [48]. First, the semantic of functions is computed using random walk and the BERT [119] 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 capture the complex patterns of malware using the attention mechanism described in Section 3.2.3. 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 [49], 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 [58] 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 [120], SubgraphX [121] and PG-Explainer [122].

6.2.2 FCG Approaches for Windows Malware Detection. DLGraph [50] 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)** [21] to create a graph embedding vector. Similarly, an 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 [51], 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 [123]. On the original imbalanced dataset, the proposed model achieves an F1-score of more than 99%.

SDGNet [52] 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 [124] and the final model achieves 97.3% accuracy.

The reference [53] 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 98.32% accuracy is reached.

DMalNet [54] 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 [125, 126] 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 [55], 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 [56], 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 [127] 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 [26] 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 [57] introduces a self-supervised method based on contrastive learning for few-shot malware variants detection. The authors construct a heterogeneous graph with five 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 [128] 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 6.

Microsoft Malware Classification Challenge (MMCC) [117]. 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 [46] or FCGs [50].

VirusShare [129], VirusTotal [130]. 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 [53–55], whereas others build static CFGs and FCGs [48, 50].

Table 6. Datasets Employed in Windows Malware Detection Studies

Paper	Datasets	Performance
MAGIC [46]	MMCC	99.25% acc
HawkEye [47]	VirusShare+AndroZoo	96.82%, 93.39%, 99.6% acc
Wang et al. [48]	VirusShare+VirusTotal	90.88%, 72.44% F1
MalGraph [49]	VirusShare+VirusTotal	99.97% acc
DLGraph [50]	MMCC+VirusShare+KafanBBS	>99% acc
Oliveira et al. [51]	VirusShare	~99.4% F1
SDGNet [52]	Alibaba Dataset	97.3% acc
Li et al. [53]	VirusShare+VirusTotal	98.32% acc
DMalNet [54]	VirusShare+VirusTotal	98.43% acc
MeQDFG [55]	VirusShare	86.22% acc
MatchGNet [56]	Private	96.53% acc
MalSage [26]	VirusTotal	91.56% acc
FewM-HGCL [57]	VirusShare,ACT-KingKong,Ember, API Call Sequences,BIG 2015	85.73%-98.65% acc

7 ADVERSARIAL ATTACKS

Despite the capabilities of machine learning in classification tasks, these techniques are not immune to adversarial attacks, which aim to disturb the predictions of the model by introducing adversarial examples, crafted from small perturbations in the input. In this section, we introduce background knowledge on adversarial attacks as well as existing approaches to countering traditional and graph-based malware detection.

7.1 Background

Overview of Adversarial Attacks. Adversarial attacks pose a fundamental challenge in the field of ML due to the inherent vulnerability of these models to manipulation. The beginning of adversarial research can be traced back to the early work [131], which first demonstrated that neural networks could be easily fooled by imperceptible perturbations to input data, thereby misclassifying images with high confidence. This work opened a new avenue of research into the security and robustness of ML models. The theoretical foundations of adversarial attacks delve into the inherent limitations of these models, particularly their inability to generalize well to slightly modified inputs that have not been encountered during training. The core idea of adversarial machine learning is to find and use the differences between how humans and machines understand data. By doing this, it's possible to uncover or increase weaknesses in the models.

The essence of adversarial ML is to find and use the differences between how humans and machines understand data. Doing this enables to uncover or increase weaknesses in the models [132]. Furthermore, the practical implications of adversarial attacks span across numerous domains, from Computer Vision to NLP, and even to more structured data forms like graphs. In each of these areas, attackers have developed sophisticated techniques to manipulate input data in a manner that alters model predictions without detection. For instance, in autonomous vehicles, slight alterations

to road signs can lead to misinterpretation by the driving algorithms, posing serious safety risks [133]. To address these challenges, a vast body of literature has emerged, focusing on both attack strategies and defense mechanisms. Among the notable contributions are [132], which introduced the concept of "adversarial examples", and [134], which proposed a method for generating them through gradient-based optimizations. Since then, adversarial attacks have seen great success notably in the Computer Vision domain [135], where the goal is to craft adversarial images that the model will misclassify by predicting a wrong label.

Definitions. Formally, 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 [136, 137]. 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, and so on. 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.

Defense Against Adversarial Attacks. In response to the threat posed by adversarial attacks, the ML community has developed several defensive techniques aimed at enhancing the robustness of models. Adversarial training [134] is among the most effective defenses, where models are trained on a mixture of clean and adversarially perturbed inputs, making them more resilient to similar manipulations. Another notable technique is defensive distillation [138], which involves training a model to predict the soft outputs (probabilities) of a previously trained model on the same task. This process can reduce the sensitivity of the model to small perturbations, making it harder for attackers to craft effective adversarial examples. Feature squeezing [139] reduces the color depth of images or applies spatial smoothing, thereby limiting the ability to introduce fine-grained perturbations that go unnoticed by human observers but mislead the model. Despite these efforts, many defenses have been circumvented. This underscores the need for a continuous improvement of defensive methods.

7.2 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. This process is also described in Figure 6, where an attacker evades the detection of a GRL-based model by altering the structure of an input malware. However, the aforementioned requirements imply multiple constraints that are difficult to fulfill in the case of malware adversarial attacks. Indeed, as explained by Ling et al. [136], 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

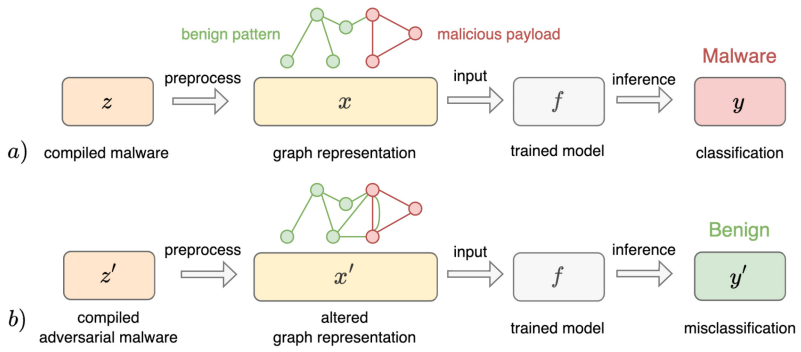

Fig. 6. Example of adversarial attack in the context of malware detection with GRL. (a) A compiled malware z is transformed into a graph x such as a FCG or a CFG. The model f successfully classifies this graph as malware. (b) The attacker crafts another malware z' that integrates the same malicious payload as z but also alters the graph representation x' by integrating benign patterns. The model f misclassifies the malware, the attacker successfully bypassed the detection by altering the input representation of the malware.

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 [140]. 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 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 [141] and [142] append an adversarial sequence of bytes to the malware. Other works prefer to modify regions in the PE header [143] or extend the DOS header [137]. However, these techniques are ineffective for higher-level representations such as those based on API calls. For this purpose, some works insert additional API calls in feature space to add noise in the representation and evade the detection systems [144, 145]. In other works, RL is leveraged to manipulate the original malware in order to evade detection while maintaining a correct format and semantic [146, 147].

7.3 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 [148–150]. In the case of malware, removing random 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. However, we review several works, presented in Table 7, that have successfully bypassed the detection of GRL-based models by altering the structure of graphs.

Two such adversarial approaches specifically designed against call graph-based malware detection are proposed by Xu et al. in MANIS [151]. 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

Table 7. Summary of GRL-based Adversarial Attack Works

Data	Access	Techniques	Datasets	Year	Paper
	White-box, gray-box	N-strongest nodes, gradient-based	AndroZoo, Drebin	2020	MANIS [151]
APK FCG	White-box	RL, heuristic optimization	AndroZoo, MalScan	2021	HRAT [152]
	Gray-box	VGAE	CICMaldroid, Drebin	2022	VGAE-MalGAN [43]
PE CFG	Black-box	RL, NOPs insertion	VXHeavens, VirusShare	2022	[153]
PE FCG	Black-box	Adversarial code injection	VirusShare	2024	[73]

In this table, **Data** refers to the type of graph used as input; **Visibility** denotes the visibility that the attacker has on the system; **Techniques** summarizes the technical methods leveraged in the paper.

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 [152], authors propose HRAT, a structural attack for APK-based FCGs that aims to address the inverse mapping problem [140], that consists in retrieving a valid malware in problem-space from the modified malware in feature-space (see Section 7.2). The proposed method works in white-box setting, and leverages RL 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.

An adversarial attack for GNN-based APK malware detection has been introduced in VGAE-MalGAN [43] to measure the robustness of the proposed detection model. The proposed attack is based on a VGAE model, aiming to effectively add nodes and edges to a FCG in order to fool the GNN classifier, in a gray-box setting where the attacker has knowledge that the detector is a GNN-based model and knows the features used. This adversarial approach has been applied to the original GNN model to further improve the robustness of the detection system.

An adversarial method based on RL is presented in [153] 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 detection in black-box setting with nearly 100% attack success rate.

In [73], the Mal2GCN model is introduced, a model that improves resilience against adversarial attacks on malware detection. Leveraging FCGs from API calls and a DGCNN model, Mal2GCN targets malware's functional behaviors over binary data to counter evasion techniques such as byte appending. However, this method is vulnerable to adversarial source code changes. Addressing this, Mal2GCN employs a non-negative training method, inspired by [154], to mitigate code alteration effects used in malware disguise. It remains accurate across 2,000 adversarial samples.

7.4 Real-world Implications for Attackers

Adversarial attacks against malware detection models should be understood as attacks on the entirety of security systems, not merely isolated models. Attackers targeting these systems face a landscape far more complex than simply manipulating inputs to evade a single detection model.

In practice, attackers encounter systems that integrate various forms of threat detection and response mechanisms. For example, an effective adversarial attack must take into account not only how to evade detection by an ML model but also how to circumvent other layers of defense. This

may involve evading heuristic-based detection, bypassing rate limiting, and staying undetected by anomaly detection systems that look for unusual patterns of activity indicative of an attack.

The work [155] underscores the discrepancy between theoretical adversarial ML models and their application in real cybersecurity environments. It points out that attackers often lack complete knowledge about the cybersecurity detection systems they are targeting. Rather, they might exploit certain vulnerabilities without the need to fully understand or modify the underlying ML models directly. This perspective is important for developing more realistic defensive strategies that take into account the practical constraints and capabilities of attackers.

The common assumption in adversarial ML research is that attackers have unrestricted access to the target models (i.e., in a white-box setting) or can freely interact with the systems to generate adversarial examples [155]. The focus is instead recommended to shift towards adversarial techniques that are feasible under realistic conditions, such as limited knowledge about the system and indirect access through legitimate channels (i.e., gray-box or black-box scenarios).

Furthermore, there is a disparity between academic research on adversarial ML and the tactics used by actual attackers [156]. For instance, attackers often employ simpler, more straightforward methods to circumvent ML-driven systems rather than using sophisticated adversarial examples. This observation demonstrates the need for security practitioners to prioritize defenses against more common and practical attack methods, rather than concentrating exclusively on defending against complex adversarial attacks that are less likely to occur.

Moreover, it has been analyzed that while modern organizations are aware of these issues, they do not regard this threat as a top priority because there are no defensive mechanisms that are truly effective in real-world environments [157]. Indeed, given sufficient resources, attackers can achieve their objectives [158], underscoring the challenge in securing systems against cyberattacks. More broadly, as outlined by [156], the underlying economics significantly influence practical cybersecurity efforts for both attackers and defenders. This economic perspective suggests that decisions in cybersecurity are primarily driven by cost-benefit analysis.

8 DISCUSSION

8.1 Application in Real-World Environments

Although GRL techniques show promising capabilities in assisting security analysts with the detection of complex malware, their application in real-world environments remains a challenging task that should be carefully considered beforehand. Indeed, most of the papers surveyed here report near-perfect detection performance, as evidenced by the high detection metrics presented in Tables 3 and 6. However, these studies often only briefly mention the practical application of their proposed techniques in real-world scenarios and encounter multiple sustainability challenges that may necessitate further exploration.

8.1.1 Matching Real-World Requirements.

Scalability. Ensuring the scalability of detection systems for real-world applications is a critical yet often underexplored requirement in existing literature. For example, the new malware detection model implemented in Gmail [159] is tasked with analyzing over 300 billion documents weekly [160], a demand that naturally leads to a tradeoff between processing speed and detection accuracy due to the need to handle such a vast volume of data. To preserve the scalability of the model over time, adopting incremental learning models is pivotal. These models can update themselves with new data in real-time or near-real-time, eliminating the necessity for complete retraining [161]. This strategy significantly lowers the computational burden and allows for rapid adaptation to emerging malware signatures or behaviors. Furthermore, when suitable, GNN-based methods might benefit from techniques such as neighbor sampling, which controls

the size of neighborhoods in the graph, and mini-batch training, enabling the parallel processing of multiple graphs on GPUs.

Concept Drift and Biases. The phenomenon of concept drift, marked by evolving data distributions over time, necessitates ongoing model updates to maintain detection performance [162]. This issue is especially prominent in malware detection, where similarities within malware families introduce a positive bias in k-fold cross-validation, leading to an overestimated effectiveness of classifiers [163]. The studies in [163] also exposed spatial and temporal biases in the AndroZoo malware samples, widely used in current research, suggesting strategies for their mitigation. To address concept drift and maintain the relevance of models over time, the implementation of systems for dynamic model updating and retraining is essential. This includes continuous learning mechanisms for rapid adaptation to new data. Moreover, [164] noted the tendency of researchers to develop learning-based methods without fully grasping the true distribution of the input space, relying instead on datasets designed to mimic this distribution. Despite the inevitable presence of some bias, identifying and tackling the specific biases in the used datasets has become essential.

Varying Input Formats. The majority of techniques discussed in this survey have shown success in detecting malware on particular platforms, such as Android and Windows. Furthermore, the data sources used to construct input graphs are often specific to a platform; for example, an FCG derived from an APK file shares no common attributes with an FCG from a PE file. This significantly limits the transferability of methods trained on one platform to others. Nonetheless, there are many practical scenarios where malware detection needs to be agnostic of the platform or file format. To overcome this limitation, existing methods require certain modifications. Datasets must include a wider variety of examples, and alternative graph representations like CFGs could be employed to represent the code in a more generic format applicable across different file formats [46, 58].

Interpretability. One of the limitations of employing deep models is their lack of interpretability, as they operate as "black boxes." However, understanding the reason behind the decision of a predictive model is essential, especially in cybersecurity. In this domain, analysts need to comprehend the security-related decisions made by algorithms. While some techniques exist to shed light on the predictions made by deep architectures like GNNs [120, 165], there are relatively few studies that specifically use these methods to enhance the interpretability of malware predictions with GNNs [16]. Therefore, further research in this area could significantly benefit malware detection.

8.1.2 Reliance on Noisy Datasets.

Erroneous Labels. A significant hurdle in deploying ML for malware detection is the dependency on datasets that frequently contain noisy and inaccurately labeled data. Collecting a vast corpus of files for malware detection is feasible using public repositories, yet relying on unverified data poses significant risks due to the potential inclusion of malicious applications. The challenge lies in verifying the ground truth, as automated web services like VirusTotal may produce conflicting results, which necessitate expensive expert validation [166]. A potential solution is the adoption of collective wisdom techniques, which aggregate insights from various antimalware engines, and efforts to clean noisy labels in the dataset [167]. For practical use in ML-driven malware detection, a dataset needs to be large enough to accurately represent the environment and threats, have a balanced ratio of benign to malicious samples to prevent excessive false positives, include accurately labeled data, and be regularly updated for relevance [168].

Inconsistent Datasets. While numerous studies demonstrate the high detection rate of GRL techniques for malware detection, the evaluation of these models often relies on distinct examples.

Popular datasets are commonly augmented with additional samples from public sources such as the Google Play Store, AndroZoo, VirusShare, and VirusTotal. This practice makes the comparison between models inefficient, as the inconsistency in training and testing samples across studies leads to different detection results. Moreover, such variability undermines the reproducibility of results in real-world settings, given the inability to use identical input files as those reported in the literature. Tackling this issue requires creating and adopting comprehensive, diverse, and open benchmark datasets. Such datasets would standardize evaluations of GRL models in malware detection and enhance both comparability and reproducibility. To stay current and effective, these resources must be regularly updated to reflect the evolving nature of malware.

8.1.3 Vulnerability to Adversarial Attacks.

Increasing Frequency and Sophistication of Adversarial Attacks. The vulnerability of ML models to adversarial attacks poses a significant challenge. Such attacks can occur during both the training phase, known as poisoning, and the inference phase, known as evasion. They exploit the learned decision boundaries of the model to misclassify malicious samples as benign [155]. In 2020, Gartner predicted that by 2022, 30% of cyberattacks would involve tactics such as training-data poisoning, model theft, or the creation of adversarial examples [169]. However, the robustness of GRL-based malware detection models against adversarial attacks remains an area that is still under-explored. One possible approach to enhance the resilience of detection models is to include adversarial examples in the training process, a method known as adversarial training. Other strategies, such as those presented in Section 7, could also complicate the efforts of attackers to manipulate the model with adversarial inputs. These mechanisms aim to reduce the attack surface accessible to attackers or hide model details that could be exploited.

Indifference to the Security of ML Models. Research into the practical application of adversarial ML reveals a significant gap between academic interest and industry implementation. Surveys and studies conducted from 2020 to 2022 highlight a concerning trend: many practitioners and ML developers are either unaware of or indifferent to the security of their ML models [156]; a trend also confirmed by the rare mentions of adversarial attack robustness in the works reviewed in this study. Initial surveys found a lack of response and concern primarily focused on poisoning attacks. Subsequent investigations revealed that a significant portion of ML models, especially those deployed in mobile apps, lack any form of protection. Furthermore, many developers admit to a lack of understanding on how to secure ML systems against adversarial threats, with a general attitude of questioning the necessity of countermeasures. Despite warnings and predictions about the increasing role of adversarial tactics in cyberattacks, the industry's position on prioritizing the security of ML models remains largely unchanged. This indicates a widespread underestimation of the risks associated with adversarial attacks.

8.2 Comparison with Other DL & ML Techniques

In the context of malware detection, GRL-based methods have unique advantages and drawbacks compared to other DL and ML models.

8.2.1 Strengths of GRL Techniques.

The primary advantage of GRL methods lies in their ability to capture and model the intricate relationships and dependencies present in graph-structured data. Unlike traditional DL methods such as CNNs that excel in grid-like data structures (e.g., images), or RNNs that are designed for sequential data, GNNs are powerful at handling data represented in graphs. This is particularly beneficial for malware detection, where the relationships between system entities (e.g., files, processes, network connections) can be naturally modeled as a graph. For instance, the propagation patterns of malware within a network can be effectively captured

using GNNs, providing insights that are not readily accessible through other ML techniques [6]. The intrinsic structure of graphs also enhances the robustness of graph-based detection systems against adversarial attacks. This is because altering attack patterns to evade detection, while maintaining the malicious payload, becomes more challenging. For detecting specific attack schemes, such as those detailed in **Common Vulnerabilities and Exposures (CVEs)**, graphs can represent these patterns robustly, making evasion difficult for attackers [43]. Moreover, GNNs and GRL techniques efficiently create low-dimensional embeddings for graph entities (nodes, edges, or subgraphs). Consequently, these embeddings aid in classifying entities, predicting links, and detecting malware by identifying complex patterns. This notably enhances the detection of advanced malware with evasion or polymorphic behaviors [25, 79, 80].

Finally, leveraging these representations alongside the original graph structure enables a visual and intuitive graph-based examination of predictions. These predictions play a crucial role in providing explicability, unlike other ML and DL techniques that may only yield a single prediction or an overall embedding for the entire malware. In summary, it provides detailed embeddings of the individual entities within the graph and enables a deeper analysis of the attacks and false positives.

Weaknesses of GRL Techniques. Despite their advantages, GNNs and GRL techniques are not devoid of challenges. One of the primary limitations is scalability. Processing large-scale graphs, as often encountered in cybersecurity applications, poses significant computational and memory demands. This is due to the necessity to aggregate and process information from potentially large neighborhoods for each node in the graph, which can quickly become infeasible for graphs with millions of nodes and edges [170]. Another critical issue specific to GNNs is the phenomenon of over-smoothing, a condition where multiple layers of a GNN model cause the node features to converge to similar values, making them indistinguishable. This over-generalization can lead to a loss of critical information, reducing the ability of the model to differentiate between benignware and malware [171]. Furthermore, the performance of GNNs and GRL techniques is heavily dependent on the quality and structure of the input graph. In the context of malware detection, the choice of the input graph representation (i.e., FCG, CFG, etc.) has a significant impact on the overall detection capabilities, as some graphs may capture important behaviors that others could miss. The dynamic nature of cyber threats, which continuously evolve to evade detection, further exacerbates this challenge. This notably requires the graphs to be constantly analyzed to ensure they accurately capture these evolving patterns.

In summary, while GNNs and GRL techniques offer promising avenues for improving malware detection through their ability to capture important relations, they also introduce specific challenges related to scalability, model depth sensitivity, and dependency on the quality of graph construction. Tackling these challenges requires continuous research and innovation to unlock the full potential of GRL techniques in malware detection.

9 CONCLUSION

In this paper, we provide an in-depth review of GRL techniques applied to the detection of Android and Windows 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 GRL 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. 2006. Dynamic analysis of malicious code. *Journal in Computer Virology, Springer* (2006).

[2] Yanfang Ye, Tao Li, Donald Adjeroh, and S. Sitharama Iyengar. 2017. 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. 2020. A comprehensive review on malware detection approaches. *IEEE Access* (2020).

[4] Alexander Küchler, Alessandro Mantovani, Yufei Han, Leyla Bilge, and Davide Balzarotti. 2021. 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. 2020. 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. 2020. Graph neural networks: A review of methods and applications. *AI Open, Elsevier* (2020).

[7] Abdelmonim Naway and Yuancheng Li. 2018. 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. 2020. 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. 2014. DroidMiner: Automated mining and characterization of fine-grained malicious behaviors in android applications. *European Symposium on Research in Computer Security, Springer* (2014).

[10] Shifu Hou, Yanfang Ye, Yangqiu Song, and Melih Abdulhayoglu. 2018. Make evasion harder: An intelligent Android malware detection system. *IJCAI* (2018).

[11] Shifu Hou, Yanfang Ye, Yangqiu Song, and Melih Abdulhayoglu. 2017. 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).

[12] Zhiqiang Wang, Qian Liu, and Yaping Chi. 2020. Review of Android malware detection based on deep learning. *IEEE Access* (2020).

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

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

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

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

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

[18] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and S. Yu Philip. 2020. A comprehensive survey on graph neural networks. *IEEE Transactions on Neural Networks and Learning Systems, IEEE* (2020).

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

[20] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S. Corrado, and Jeff Dean. 2013. Distributed representations of words and phrases and their compositionality. *Advances in Neural Information Processing Systems* (2013).

[21] Pascal Vincent, Hugo Larochelle, Isabelle Lajoie, Yoshua Bengio, Pierre-Antoine Manzagol, and Léon Bottou. 2010. Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion. *Journal of Machine Learning Research* (2010).

[22] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, Łukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Greg Corrado, Macduff Hughes, and Jeffrey Dean.

2016. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144* (2016).
[23] LeCun, Yann, and Yoshua Bengio. 1995. Convolutional networks for images, speech, and time series. *The Handbook of Brain Theory and Neural Networks, Cambridge, MA USA* (1995).
[24] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural Computation, MIT Press* (1997).
[25] William L. Hamilton, Rex Ying, and Jure Leskovec. 2017. Representation learning on graphs: Methods and applications. *arXiv preprint arXiv:1709.05584* (2017).
[26] Meihua Fan, Shudong Li, Weihong Han, Xiaobo Wu, Zhaoquan Gu, and Zhihong Tian. 2020. A novel malware detection framework based on weighted heterograph. *Proceedings of the 2020 International Conference on Cyberspace Innovation of Advanced Technologies* (2020).
[27] Benjamin Bowman, Craig Laprade, Yuede Ji, and H. Howie Huang. 2020. Detecting lateral movement in enterprise computer networks with unsupervised graph {AI}. *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)* (2020).
[28] Isaiah J. King and H. Howie Huang. 2023. Euler: Detecting network lateral movement via scalable temporal link prediction. (2023).
[29] Mohammad Reza Norouzian, Peng Xu, Claudia Eckert, and Apostolis Zarras. 2021. Hybroid: Toward Android malware detection and categorization with program code and network traffic. *International Conference on Information Security* (2021).
[30] Peng Xu, Claudia Eckert, and Apostolis Zarras. 2021. hybrid-Falcon: Hybrid pattern malware detection and categorization with network traffic and program code. *arXiv preprint arXiv:2112.10035* (2021).
[31] Jeffrey Fairbanks, Andres Orbe, Christine Patterson, Janet Layne, Edoardo Serra, and Marion Scheepers. 2021. 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).
[32] Rui Zhu, Chenglin Li, Di Niu, Hongwen Zhang, and Husam Kinawi. 2018. Android malware detection using large-scale network representation learning. *arXiv preprint arXiv:1806.04847* (2018).
[33] Pengbin Feng, Jianfeng Ma, Teng Li, Xindi Ma, Ning Xi, and Di Lu. 2020. Android malware detection based on call graph via graph neural network. *2020 International Conference on Networking and Network Applications (NaNA)* (2020).
[34] Minghui Cai, Yuan Jiang, Cuiying Gao, Heng Li, and Wei Yuan. 2021. Learning features from enhanced function call graphs for Android malware detection. *Neurocomputing, Elsevier* (2021).
[35] Peng Xu, Claudia Eckert, and Apostolis Zarras. 2021. Detecting and categorizing Android malware with graph neural networks. *Proceedings of the 36th Annual ACM Symposium on Applied Computing* (2021).
[36] K. V. Vinayaka and C. D. Jaidhar. 2021. Android malware detection using function call graph with graph convolutional networks. *2021 2nd International Conference on Secure Cyber Computing and Communications (ICSCCC)* (2021).
[37] Federico Errica, Giacomo Iadarola, Fabio Martinelli, Francesco Mercaldo, and Alessio Micheli. 2021. Robust malware classification via deep graph networks on call graph topologies. *ESANN* (2021).
[38] Cagatay Catal, Hakan Gunduz, and Alper Ozcan. 2021. Malware detection based on graph attention networks for intelligent transportation systems. *Electronics, MDPI* (2021).
[39] Yafei Wu, Jian Shi, Peicheng Wang, Dongrui Zeng, and Cong Sun. 2022. DeepCatra: Learning flow-and graph-based behaviors for Android malware detection. *arXiv preprint arXiv:2201.12876* (2022).
[40] Wai Weng Lo, Siamak Layeghy, Mohanad Sarhan, Marcus Gallagher, and Marius Portmann. 2022. Graph neural network-based Android malware classification with jumping knowledge. *CoRR* (2022).
enlargethispage6pt
[41] Hakan Gunduz. 2022. Malware detection framework based on graph variational autoencoder extracted embeddings from API-call graphs. *PeerJ Computer Science* (2022).
[42] Xiaofeng Lu, Jinglun Zhao, and Pietro Lio. 2022. 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).
[43] Rahul Yumlembam, Biju Issac, Seibu Mary Jacob, and Longzhi Yang. 2022. IoT-based Android malware detection using graph neural network with adversarial defense. *IEEE Internet of Things Journal, IEEE* (2022).
[44] Peng Xu and Asbat El Khairi. 2021. Android-COCO: Android malware detection with graph neural network for byte-and native-code. *arXiv preprint arXiv:2112.10038* (2021).
[45] Teenu S. John, Tony Thomas, and Sabu Emmanuel. 2020. Graph convolutional networks for Android malware detection with system call graphs. *2020 Third ISEA Conference on Security and Privacy (ISEA-ISAP)* (2020).
[46] Jiaqi Yan, Guanhua Yan, and Dong Jin. 2019. 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).

[47] Peng Xu, Youyi Zhang, Claudia Eckert, and Apostolis Zarras. 2021. 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).
[48] Shuai Wang, Yuran Zhao, Gongshen Liu, and Bo Su. 2021. A hierarchical graph-based neural network for malware classification. *International Conference on Neural Information Processing* (2021).
[49] Xiang Ling, Lingfei Wu, Wei Deng, Zhenqing Qu, Jiangyu Zhang, Sheng Zhang, Tengfei Ma, Bin Wang, Chunming Wu, and Shouling Ji. 2022. MalGraph: Hierarchical graph neural networks for robust windows malware detection. *IEEE INFOCOM 2022-IEEE Conference on Computer Communications* (2022).
[50] Haodi Jiang, Turki Turki, and Jason T. L. Wang. 2018. DLGraph: Malware detection using deep learning and graph embedding. *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)* (2018).
[51] Angelo Oliveira and R. Sassi. 2019. Behavioral malware detection using deep graph convolutional neural networks. *Techlixiv.[link]* (2019).
[52] Zikai Zhang, Yidong Li, Hairong Dong, Honghao Gao, Yi Jin, and Wei Wang. 2020. Spectral-based directed graph network for malware detection. *IEEE Transactions on Network Science and Engineering, IEEE* (2020).
[53] Shanxi Li, Qingguo Zhou, Rui Zhou, and Qingquan Lv. 2022. Intelligent malware detection based on graph convolutional network. *The Journal of Supercomputing, Springer* (2022).
[54] Ce Li, Zijun Cheng, He Zhu, Leiqi Wang, Qiujian Lv, Yan Wang, Ning Li, and Degang Sun. 2022. DMalNet: Dynamic malware analysis based on API feature engineering and graph learning. *Computers & Security, Elsevier* (2022).
[55] Nguyen Viet Hung, Pham Ngoc Dung, Tran Nguyen Ngoc, Vu Dinh Phai, and Qi Shi. 2019. 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).
[56] Xiao Wang, Houye Ji, Chuan Shi, Bai Wang, Yanfang Ye, Peng Cui, and Philip S. Yu. 2019. Heterogeneous graph attention network. *The World Wide Web Conference* (2019).
[57] Chen Liu, Bo Li, Jun Zhao, Ziyang Zhen, Xudong Liu, and Qunshi Zhang. 2022. FewM-HGCL: Few-shot malware variants detection via heterogeneous graph contrastive learning. *IEEE Transactions on Dependable and Secure Computing, IEEE Computer Society* (2022).
[58] Jerome Dinal Herath, Priti Prabhakar Wakodikar, Ping Yang, and Guanhua Yan. 2022. 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).
[59] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. 2014. DeepWalk: Online learning of social representations. *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (2014).
[60] Aditya Grover and Jure Leskovec. 2016. node2vec: Scalable feature learning for networks. *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (2016).
[61] Jian Tang, Meng Qu, Mingzhe Wang, Ming Zhang, Jun Yan, and Qiaozhu Mei. 2015. LINE: Large-scale information network embedding. *Proceedings of the 24th International Conference on World Wide Web* (2015).
[62] Petar Velickovic, William Fedus, William L. Hamilton, Pietro Liò, Yoshua Bengio, and R. Devon Hjelm. 2019. Deep graph infomax. *ICLR (Poster)* (2019).
[63] Marco Gori, Gabriele Monfardini, and Franco Scarselli. 2005. A new model for learning in graph domains. *Proceedings. 2005 IEEE International Joint Conference on Neural Networks* (2005).
[64] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. 2008. The graph neural network model. *IEEE Transactions on Neural Networks, IEEE* (2008).
[65] Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. 2016. Convolutional neural networks on graphs with fast localized spectral filtering. *Advances in Neural Information Processing Systems* (2016).
[66] Thomas N. Kipf and Max Welling. 2016. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907* (2016).
[67] Will Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. *Advances in Neural Information Processing Systems* (2017).
[68] Muhan Zhang, Zhicheng Cui, Marion Neumann, and Yixin Chen. 2018. An end-to-end deep learning architecture for graph classification. *Proceedings of the AAAI Conference on Artificial Intelligence* (2018).
[69] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. 2017. Graph attention networks. *arXiv preprint arXiv:1710.10903* (2017).
[70] Han Gao, Shaoyin Cheng, and Weiming Zhang. 2021. GDroid: Android malware detection and classification with graph convolutional network. *Computers & Security, Elsevier* (2021).
[71] Tristan Bilot, Grégoire Geis, and Badis Hammi. 2022. 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).

[72] Zhonglin Liu, Yong Fang, Cheng Huang, and Jiaxuan Han. 2022. GraphXSS: An efficient XSS payload detection approach based on graph convolutional network. *Computers & Security, Elsevier* (2022).
[73] Omid Kargarnovin, Amir Mahdi Sadeghzadeh, and Rasool Jalili. 2024. Mal2GCN: A robust malware detection approach using deep graph convolutional networks with non-negative weights. *Journal of Computer Virology and Hacking Techniques* 20, 1 (2024), 95–111.
[74] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2014. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473* (2014).
[75] Yong Fang, Chaoyi Huang, Minchuan Zeng, Zhiying Zhao, and Cheng Huang. 2022. JStrong: Malicious JavaScript detection based on code semantic representation and graph neural network. *Computers & Security, Elsevier* (2022).
[76] Shiwen Wu, Fei Sun, Wentao Zhang, Xu Xie, and Bin Cui. 2022. Graph neural networks in recommender systems: A survey. *ACM Computing Surveys, ACM New York, NY* (2022).
[77] Weiwei Jiang and Jiayun Luo. 2022. 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 B. R. Hayter, Richard Vickers, Charles Roberts, Jian Tang, David Roblin, Tom L. Blundell, Michael M. Bronstein, and Jake P. Taylor-King. 2021. Utilizing graph machine learning within drug discovery and development. *Briefings in Bioinformatics, Oxford University Press* (2021).
[79] Blake Anderson, Curtis Storlie, and Terran Lane. 2012. 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. 2010. Detecting metamorphic malwares using code graphs. *Proceedings of the 2010 ACM Symposium on Applied Computing* (2010).
[81] Fengguo Wei, Sankardas Roy, and Xinming Ou. 2018. 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).
[82] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. 1987. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS), ACM New York, NY, USA* (1987).
[83] Cuckoo Sandbox Book — Cuckoo Sandbox v2.0.7 Book. 2020. https://cuckoo.readthedocs.io/en/latest/ [Accessed on 01/17/2023] (2020).
[84] Yiming Hei, Renyu Yang, Hao Peng, Lihong Wang, Xiaolin Xu, Jianwei Liu, Hong Liu, Jie Xu, and Lichao Sun. 2021. Hawk: Rapid Android malware detection through heterogeneous graph attention networks. *IEEE Transactions on Neural Networks and Learning Systems, IEEE* (2021).
[85] Welcome to Androguard's documentation! — Androguard 3.4.0 documentation. 2018. https://androguard.readthedocs.io/en/latest/ [Accessed on 01/17/2023] (2018).
[86] radare. 2023. https://rada.re/n/ [Accessed on 01/17/2023] (2023).
[87] Hex Rays State of-the-art binary code analysis solutions. 2023. https://hex-rays.com/ida-pro [Accessed on 01/11/2023] (2023).
[88] Ghidra. 2023. https://ghidra-sre.org/ [Accessed on 01/17/2023] (2023).
[89] binary Android apps. Apktool A tool for reverse engineering 3rd party, closed. 2022. https://ibotpeaches.github.io/Apktool [Accessed on 01/11/2023] (2022).
[90] Djack1010/graph4apk. 2021. https://github.com/Djack1010/graph4apk [Accessed on 01/17/2023] (2021).
[91] Main Page WalaWiki. 2019. https://wala.sourceforge.net/wiki/index.php/Main_Page [Accessed on 01/12/2023] (2019).
[92] angr. 2022. https://angr.io/ [Accessed on 01/17/2023] (2022).
[93] strace. 2022. https://strace.io/ [Accessed on 01/17/2023] (2022).
[94] HomePage Systemtap Wiki. 2022. https://sourceware.org/systemtap/wiki/ [Accessed on 02/10/2023] (2022).
[95] ANY.RUN Interactive Online Malware Sandbox. 2023. https://any.run/ [Accessed on 02/10/2023] (2023).
[96] Hanjun Dai, Bo Dai, and Le Song. 2016. Discriminative embeddings of latent variable models for structured data. *International Conference on Machine Learning* (2016).
[97] openargus Home. 2022. https://openargus.org [Accessed on 12/11/2022] (2022).
[98] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, Konrad Rieck, and CERT Siemens. 2014. Drebin: Effective and explainable detection of Android malware in your pocket. *NDSS* (2014).
[99] Isabelle Guyon, Jason Weston, Stephen Barnhill, and Vladimir Vapnik. 2002. Gene selection for cancer classification using support vector machines. *Machine Learning, Springer* (2002).
[100] Janet Layne and Edoardo Serra. 2021. Inferential SIR-GN: Scalable graph representation learning. *arXiv preprint arXiv:2111.04826* (2021).
[101] Scott M. Lundberg and Su-In Lee. 2017. A unified approach to interpreting model predictions. *Advances in Neural Information Processing Systems* (2017).

[102] Kevin Allix, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. 2016. AndroZoo: Collecting millions of Android apps for the research community. *Proceedings of the 13th International Conference on Mining Software Repositories* (2016).
[103] Jian Du, Shanghang Zhang, Guanhang Wu, José M. F. Moura, and Soummya Kar. 2017. Topology adaptive graph convolutional networks. *arXiv preprint arXiv:1710.10370* (2017).
[104] Davide Bacciu, Federico Errica, and Alessio Micheli. 2018. Contextual graph Markov model: A deep and generative approach to graph processing. *International Conference on Machine Learning* (2018).
[105] CVE CVE. 2023. https://cve.mitre.org [Accessed on 01/12/2023] (2023).
[106] Exploit Database Exploits for Penetration Testers Researchers and Ethical Hackers. 2023. https://www.exploit-db.com/ [Accessed on 01/12/2023] (2023).
[107] Keyulu Xu, Chengtao Li, Yonglong Tian, Tomohiro Sonobe, Ken-ichi Kawarabayashi, and Stefanie Jegelka. 2018. Representation learning on graphs with jumping knowledge networks. *International Conference on Machine Learning* (2018).
[108] Thomas N. Kipf and Max Welling. 2016. Variational graph auto-encoders. *arXiv preprint arXiv:1611.07308* (2016).
[109] Chuan Shi, Yitong Li, Jiawei Zhang, Yizhou Sun, and S. Yu Philip. 2016. A survey of heterogeneous information network analysis. *IEEE Transactions on Knowledge and Data Engineering, IEEE* (2016).
[110] Fengguo Wei, Yuping Li, Sankardas Roy, Xinming Ou, and Wu Zhou. 2017. 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).
[111] Arash Habibi Lashkari, Andi Fitriah A. Kadir, Laya Taheri, and Ali A. Ghorbani. 2018. Toward developing a systematic approach to generate benchmark Android malware datasets and classification. *2018 International Carnahan Conference on Security Technology (ICCST)* (2018).
[112] Applications | Research | Canadian Institute for Cybersecurity | UNB. 2017. https://www.unb.ca/cic/research/applications.html [Accessed on 02/07/2023] (2017).
[113] Samaneh Mahdavifar, Andi Fitriah Abdul Kadir, Rasool Fatemi, Dima Alhadidi, and Ali A. Ghorbani. 2020. 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).
[114] Scott Freitas, Yuxiao Dong, Joshua Neil, and Duen Horng Chau. 2020. A large-scale database for graph representation learning. *arXiv preprint arXiv:2011.07682* (2020).
[115] Hugo Gascon, Fabian Yamaguchi, Daniel Arp, and Konrad Rieck. 2013. Structural detection of Android malware using embedded call graphs. *Proceedings of the 2013 ACM Workshop on Artificial Intelligence and Security* (2013).
[116] Katja Hahn and I. Register. 2014. Robust static analysis of portable executable malware. *HTWK Leipzig* (2014).
[117] Royi Ronen, Marian Radu, Corina Feuerstein, Elad Yom-Tov, and Mansour Ahmadi. 2018. Microsoft malware classification challenge. *arXiv preprint arXiv:1802.10135* (2018).
[118] Guanhua Yan. 2015. 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).
[119] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. BERT: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).
[120] Zhitao Ying, Dylan Bourgeois, Jiaxuan You, Marinka Zitnik, and Jure Leskovec. 2019. GNNExplainer: Generating explanations for graph neural networks. *Advances in Neural Information Processing Systems* (2019).
[121] Hao Yuan, Haiyang Yu, Jie Wang, Kang Li, and Shuiwang Ji. 2021. On explainability of graph neural networks via subgraph explorations. *International Conference on Machine Learning* (2021).
[122] Dongsheng Luo, Wei Cheng, Dongkuan Xu, Wenchao Yu, Bo Zong, Haifeng Chen, and Xiang Zhang. 2020. Parameterized explainer for graph neural network. *Advances in Neural Information Processing Systems* (2020).
[123] Angelo Oliveira. 2019. Malware analysis datasets: API call sequences, IEEE dataport. https://dx.doi.org/10.21227/tqqm-aq14 (2019).
[124] Alibaba Cloud Malware Detection Based on Behaviors. 2018. https://tianchi.aliyun.com/competition/entrance/231694/introduction [Accessed on 14/07/2023] (2018).
[125] Cătălina Cangea, Petar Veličković, Nikola Jovanović, Thomas Kipf, and Pietro Liò. 2018. Towards sparse hierarchical graph classifiers. *arXiv preprint arXiv:1811.01287* (2018).
[126] Hongyang Gao and Shuiwang Ji. 2019. Graph U-Nets. *International Conference on Machine Learning* (2019).
[127] Jane Bromley, Isabelle Guyon, Yann LeCun, Eduard Säckinger, and Roopak Shah. 1993. Signature verification using a 'Siamese" time delay neural network. *Advances in Neural Information Processing Systems* (1993).
[128] Kilian Weinberger, Anirban Dasgupta, John Langford, Alex Smola, and Josh Attenberg. 2009. Feature hashing for large scale multitask learning. *Proceedings of the 26th Annual International Conference on Machine Learning* (2009).
[129] VirusShare.com. 2023. https://virusshare.com/ [Accessed on 01/26/2023] (2023).

[130] VirusTotal – Home. 2023. https://virustotal.com/ [Accessed on 01/26/2023] (2023).
[131] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. 2013. Intriguing properties of neural networks. *arXiv preprint arXiv:1312.6199* (2013).
[132] Battista Biggio, Igino Corona, Davide Maiorca, Blaine Nelson, Nedim Šrndić, Pavel Laskov, Giorgio Giacinto, and Fabio Roli. 2013. Evasion attacks against machine learning at test time. In *Machine Learning and Knowledge Discovery in Databases: European Conference, ECML PKDD 2013, Prague, Czech Republic, September 23-27, 2013, Proceedings, Part III 13*. Springer, 387–402.
[133] Kevin Eykholt, Ivan Evtimov, Earlence Fernandes, Bo Li, Amir Rahmati, Chaowei Xiao, Atul Prakash, Tadayoshi Kohno, and Dawn Song. 2018. Robust physical-world attacks on deep learning visual classification. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 1625–1634.
[134] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. 2014. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572* (2014).
[135] Naveed Akhtar and Ajmal Mian. 2018. Threat of adversarial attacks on deep learning in computer vision: A survey. *IEEE Access* (2018).
[136] Xiang Ling, Lingfei Wu, Jiangyu Zhang, Zhenqing Qu, Wei Deng, Xiang Chen, Yaguan Qian, Chunming Wu, Shouling Ji, Tianyue Luo, Jingzheng Wu, and Yanjun Wu. 2023. Adversarial attacks against Windows PE malware detection: A survey of the state-of-the-art. *Computers & Security, Elsevier* (2023).
[137] Luca Demetrio, Scott E. Coull, Battista Biggio, Giovanni Lagorio, Alessandro Armando, and Fabio Roli. 2021. 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).
[138] Nicolas Papernot, Patrick McDaniel, Xi Wu, Somesh Jha, and Ananthram Swami. 2016. Distillation as a defense to adversarial perturbations against deep neural networks. In *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 582–597.
[139] Weilin Xu, David Evans, and Yanjun Qi. 2017. Feature squeezing: Detecting adversarial examples in deep neural networks. *arXiv preprint arXiv:1704.01155* (2017).
[140] Fabio Pierazzi, Feargus Pendlebury, Jacopo Cortellazzi, and Lorenzo Cavallaro. 2020. Intriguing properties of adversarial ML attacks in the problem space. *2020 IEEE Symposium on Security and Privacy (SP)* (2020).
[141] Felix Kreuk, Assi Barak, Shir Aviv-Reuven, Moran Baruch, Benny Pinkas, and Joseph Keshet. 2018. Deceiving end-to-end deep learning malware detectors using adversarial examples. *arXiv preprint arXiv:1802.04528* (2018).
[142] Octavian Suciu, Scott E. Coull, and Jeffrey Johns. 2019. Exploring adversarial examples in malware detection. *2019 IEEE Security and Privacy Workshops (SPW)* (2019).
[143] Luca Demetrio, Battista Biggio, Giovanni Lagorio, Fabio Roli, and Alessandro Armando. 2019. Explaining vulnerabilities of deep learning to adversarial malware binaries. *arXiv preprint arXiv:1901.03583* (2019).
[144] Lingwei Chen, Yanfang Ye, and Thirimachos Bourlai. 2017. Adversarial machine learning in malware detection: Arms race between evasion attack and defense. *2017 European Intelligence and Security Informatics Conference (EISIC)* (2017).
[145] Weiwei Hu and Ying Tan. 2023. 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).
[146] Cangshuai Wu, Jiangyong Shi, Yuexiang Yang, and Wenhua Li. 2018. Enhancing machine learning based malware detection model by reinforcement learning. *Proceedings of the 8th International Conference on Communication and Network Security* (2018).
[147] Zhiyang Fang, Junfeng Wang, Boya Li, Siqi Wu, Yingjie Zhou, and Haiying Huang. 2019. Evading anti-malware engines with deep reinforcement learning. *IEEE Access* (2019).
[148] Lichao Sun, Yingtong Dou, Carl Yang, Ji Wang, Philip S. Yu, Lifang He, and Bo Li. 2018. Adversarial attack and defense on graph data: A survey. *arXiv preprint arXiv:1812.10528* (2018).
[149] Daniel Zügner, Amir Akbarnejad, and Stephan Günnemann. 2018. Adversarial attacks on neural networks for graph data. *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (2018).
[150] Hanjun Dai, Hui Li, Tian Tian, Xin Huang, Lin Wang, Jun Zhu, and Le Song. 2018. Adversarial attack on graph structured data. *International Conference on Machine Learning* (2018).
[151] Peng Xu, Bojan Kolosnjaji, Claudia Eckert, and Apostolis Zarras. 2020. MANIS: Evading malware detection system on graph structure. *Proceedings of the 35th Annual ACM Symposium on Applied Computing* (2020).
[152] Kaifa Zhao, Hao Zhou, Yulin Zhu, Xian Zhan, Kai Zhou, Jianfeng Li, Le Yu, Wei Yuan, and Xiapu Luo. 2021. Structural attack against graph based Android malware detection. *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security* (2021).
[153] Lan Zhang, Peng Liu, Yoonho Choi, and Ping Chen. 2022. Semantics-preserving reinforcement learning attack against graph neural networks for malware detection. *IEEE Transactions on Dependable and Secure Computing, IEEE* (2022).

[154] William Fleshman, Edward Raff, Jared Sylvester, Steven Forsyth, and Mark McLean. 2018. Non-negative networks against adversarial attacks. *arXiv preprint arXiv:1806.06108* (2018).
[155] Giovanni Apruzzese, Mauro Andreolini, Luca Ferretti, Mirco Marchetti, and Michele Colajanni. 2022. Modeling realistic adversarial attacks against network intrusion detection systems. *Digital Threats: Research and Practice (DTRAP)* 3, 3 (2022), 1–19.
[156] Giovanni Apruzzese, Hyrum S. Anderson, Savino Dambra, David Freeman, Fabio Pierazzi, and Kevin Roundy. 2023. "Real attackers don't compute gradients": Bridging the gap between adversarial ML research and practice. In *2023 IEEE Conference on Secure and Trustworthy Machine Learning (SaTML)*. IEEE, 339–364.
[157] Ram Shankar Siva Kumar, Magnus Nyström, John Lambert, Andrew Marshall, Mario Goertzel, Andi Comissoneru, Matt Swann, and Sharon Xia. 2020. Adversarial machine learning-industry perspectives. In *2020 IEEE Security and Privacy Workshops (SPW)*. IEEE, 69–75.
[158] Tyler Moore. 2010. The economics of cybersecurity: Principles and policy options. *International Journal of Critical Infrastructure Protection* 3, 3-4 (2010), 103–117.
[159] Elie Bursztein, Marina Zhang, Owen Vallis, Xinyu Jia, and Alexey Kurakin. 2024. RETVec: Resilient and efficient text vectorizer. *Advances in Neural Information Processing Systems* 36 (2024).
[160] Improving Malicious Document Detection in Gmail with Deep Learning. 2020. https://security.googleblog.com/2020/02/improving-malicious-document-detection.html [Accessed on 03/08/2024] (2020).
[161] Seoyoon Kim, Seongjun Yun, and Jaewoo Kang. 2022. DyGRAIN: An incremental learning framework for dynamic graphs. In *IJCAI*. 3157–3163.
[162] Giovanni Apruzzese, Pavel Laskov, Edgardo Montes de Oca, Wissam Mallouli, Luis Brdalo Rapa, Athanasios Vasileios Grammatopoulos, and Fabio Di Franco. 2023. The role of machine learning in cybersecurity. *Digital Threats: Research and Practice* 4, 1 (2023), 1–38.
[163] Feargus Pendlebury, Fabio Pierazzi, Roberto Jordaney, Johannes Kinder, and Lorenzo Cavallaro. 2019. {TESSERACT}: Eliminating experimental bias in malware classification across space and time. In *28th USENIX Security Symposium (USENIX Security 19)*. 729–746.
[164] Daniel Arp, Erwin Quiring, Feargus Pendlebury, Alexander Warnecke, Fabio Pierazzi, Christian Wressnegger, Lorenzo Cavallaro, and Konrad Rieck. 2022. Dos and don'ts of machine learning in computer security. In *31st USENIX Security Symposium (USENIX Security 22)*. 3971–3988.
[165] Aosong Feng, Chenyu You, Shiqiang Wang, and Leandros Tassiulas. 2022. KerGNNs: Interpretable graph neural networks with graph kernels. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 36. 6614–6622.
[166] Shuofei Zhu, Jianjun Shi, Limin Yang, Boqin Qin, Ziyi Zhang, Linhai Song, and Gang Wang. 2020. Measuring and modeling the label dynamics of online {Anti-Malware} engines. In *29th USENIX Security Symposium (USENIX Security 20)*. 2361–2378.
[167] Nedim Šrndic and Pavel Laskov. 2013. Detection of malicious pdf files based on hierarchical document structure. In *Proceedings of the 20th Annual Network & Distributed System Security Symposium*. Citeseer, 1–16.
[168] Giovanni Apruzzese, Pavel Laskov, and Aliya Tastemirova. 2022. SoK: The impact of unlabelled data in cyberthreat detection. In *2022 IEEE 7th European Symposium on Security and Privacy (EuroS&P)*. IEEE, 20–42.
[169] Top-10 Strategic Technology Trends for 2020. 2020. https://www.forbes.com/sites/forbestechcouncil/2021/07/29/what-you-need-to-know-about-ai-security---even-if-your-company-isnt-using-ai-yet/?sh=13423a6e10a0 [Accessed on 03/08/2024] (2020).
[170] Tristan Bilot, Nour El Madhoun, Khaldoun Al Agha, and Anis Zouaoui. 2023. Graph neural networks for intrusion detection: A survey. *IEEE Access* (2023).
[171] Deli Chen, Yankai Lin, Wei Li, Peng Li, Jie Zhou, and Xu Sun. 2020. Measuring and relieving the over-smoothing problem for graph neural networks from the topological view. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 34. 3438–3445.

Received 4 March 2023; revised 26 March 2024; accepted 8 May 2024