Universiteit Leiden
The Netherlands

Bachelor Data Science & Artificial Intelligence

Detecting Android Malware Using Graph Neural Networks: A Comparison Between an Inductive and Transductive Learning Approach

Priya Georgina Anuradhakumari Smit

Supervisors:
Olga Gadyatskaya & Rui Li

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

July 3, 2026

Abstract

Android applications are a frequent target of malware attacks due to the open-source nature of the Android ecosystem and the large number of available applications. As malicious software continues to evolve, there is a need for detection methods that can adapt to these changing threats. This thesis investigates the use of graph neural networks for Android malware detection by comparing a transductive Graph Convolutional Network (GCN) with the inductive GraphSAGE model. The main objective is to assess how the inductive framework compares to the transductive model in terms of predictive performance and interpretability, thereby evaluating whether inductive learning is a suitable foundation for deployment on previously unseen applications.

A dataset of Android application packages was collected from AndroZoo and processed through a static-analysis pipeline. Each APK was converted into a call graph using Androguard, after which Node2Vec was applied to generate node feature embeddings. These graph representations were then used to train and evaluate both GCN and GraphSAGE under identical experimental conditions. Performance was measured using accuracy, precision, recall, F1-score, and ROC-AUC.

In addition, GNNExplainer was used to analyse which nodes and edges contributed most strongly to model predictions. Explanation quality was further assessed using fidelity, characterization score, fidelity-curve AUC, and unfaithfulness metrics.

The results show that GraphSAGE achieved better predictive performance than GCN across all evaluated classification metrics. These findings suggest that the inductive neighbourhood aggregation approach of GraphSAGE was more effective at distinguishing between benign and malicious Android applications under the conditions examined in this study. However, the explainability analysis produced contrasting results, with GCN achieving substantially stronger explanation metrics and more consistent explanations than GraphSAGE. This indicates that improved predictive performance does not necessarily correspond to improved interpretability.

Overall, the findings suggest that GraphSAGE is the better model for Android malware detection from a classification perspective, while GCN remains more favourable from an interpretability perspective. The study therefore highlights a trade-off between predictive performance and explainability in graph-based malware detection systems.

Keywords: malware detection, graph neural networks, graph convolutional network (GCN), GraphSAGE, Android, Node2Vec, GNNExplainer, interpretability.

Contents

1 Introduction

As one of the world's most widely used operating systems, Android presents an attractive target for malware attacks. According to Kaspersky [Kas25], nearly 23 million attacks targeting Android users were detected worldwide during the first half of 2025. The majority of these attacks were carried out using various types of Trojan malware. Statistics reported by DeepStrike indicate that Android devices are approximately 50 times more likely to be targeted by malware attacks than iOS devices [Kha25]. This may be the result of the open-source nature of Android and the inconsistent security patching of applications. Analytics from Check Point Research [Tea24] saw a 30% year-over-year increase in detected global cyber attacks. However, this could also be because more attempts were made, which naturally results in more detections due to a higher amount of potentially malicious activity.

Malicious applications can, for example, gain access to a person's sensitive and personal information. Modern cybercriminals can build upon already existing malware, as through minor adjustments to malicious snippets, attackers can bypass existing detection systems [YLAI17]. As malware continues to occur and evolve exponentially, detection systems must be capable of quickly adapting to these threats. Novel deep learning methods have been proposed for handling complicated structures.

One promising approach is the use of Graph Neural Networks (GNNs). Graphs can be derived from Android applications, and with this graph-structured data, the GNN can consider the properties of objects and the relations between them. GNNs have shown to be an effective method for Android malware detection [GZH+24]. Android applications contain large amounts of data, and the graphs are extremely large as well. The transductive Graph Convolutional Network (GCN) and the inductive derivative GraphSAGE are two networks that are able to correctly handle such large graphs.

Through a transductive model, the GNN learns patterns from a labelled dataset to make predictions on data within the same graph structure. In contrast, inductive learning is trained on labelled data in a way that allows the model to generalise to new unseen data, which may be more suitable for rapidly evolving malware.

To further delve into this topic and assess whether this method is beneficial for malware detection, this thesis will answer the following question:

How does an inductive model such as GraphSAGE compare with a transductive model such as GCN in terms of predictive performance and interpretability for Android malware detection?

Paper organisation: This thesis presents a basic review comparing an inductive and transductive graph neural network for malware detection, including a feature learning step. Chapter 2 gives relevant background information on the thesis topic, Chapter 3 gives insight into related work and why this research matters, and Chapter 4 presents and describes the main idea of the experiment in detail. The results will be discussed in Chapter 5, after which Chapter 6 discusses the thesis project and, 7 concludes the thesis.

2 Background

In this chapter, the relevant background knowledge for this thesis is presented. First, the basic components of Android applications are introduced, followed by an overview of malware types and common detection features and methods. It also introduces the concept of graph neural networks and the possible learning frameworks used with them.

2.1 Android Applications

Android applications are often written in Java and run in an Android environment such as the Dalvik Virtual Machine and Android Runtime. The APK is the Android package compressed into a zip file that contains Dalvik bytecode, resources, assets, and the `AndroidManifest.xml` file [FML+20]. Dalvik bytecode is a `.dex` file, and the code in this file is executed by Android interpreters. The bytecode is compiled during the execution of the app. The manifest file describes the essential information about the app. This includes the components of the app such as activities, services, broadcast receivers, and content providers, as well as the needed permissions in order to access protected parts of the app. These permissions give access to previously restricted data or actions, such as the camera or stored files. The manifest file declares the hardware and software features which the app requires and affects device compatibility [Dev].

2.2 Malware

Malware is malicious code that infiltrates devices connected to the internet and steals sensitive information. As a consequence, the infected device can destroy and/or gain access to confidential information. The malicious code can track the user's activity, start unwanted processes, or operate the system without the user's permission from a distance [BK24]. Malware can present itself in different forms such as Trojans, spyware, ransomware and, viruses.
A **Trojan** is a software program that seems harmless and pretends to be useful; however, it executes malicious actions on the back-end. Another malicious program is **spyware**, which spies on user activities without the user's consent. **Ransomware** is a type of program that is secretly installed on the user's device. It can lock device access or encrypt the files stored on it. A **virus** is a code snippet that can attach itself to other system programs and essentially infect those programs [YLAI17]. Together, these different forms of malware demonstrate a small portion of the strategies attackers can employ to infiltrate Android devices, exploit system vulnerabilities, and compromise user security.

2.3 Malware Detection Features

Malware detection relies on extracting informative features that capture the behaviour of an Android application. Static analysis examines an application without executing it and extracts features from the application code, such as API calls, permissions, and system events, providing insight into its potential behaviour [ZYZ+18, FML+20]. In contrast, dynamic analysis monitors an application during execution and collects runtime features, including executed API calls, network traffic, and system activity [AGP14]. A hybrid approach combines features obtained through both static and dynamic analysis.

An Application Programming Interface (API) defines a set of rules and protocols that enable communication between different software systems. An Android API call is a request made by an application to the Android framework to access system services or device features, such as the camera, contacts, or device location. These API calls are commonly analysed in Android malware detection because they reveal behavioural patterns that help distinguish between benign and malicious applications. Similarly, a system call enables a user-level program to request services directly from the operating system kernel, which is responsible for managing system resources and handling tasks such as process execution, file access, and communication with hardware devices. Unlike Android API calls, which interact with the Android framework, system calls communicate directly with the kernel to request these low-level services.
In addition, application permissions provide another important source of information. Permissions define the level of access an application has to sensitive system resources, such as contacts, messages, or device storage. These permissions are defined in the `AndroidManifest.xml` file. Unusual or excessive permission requests are often associated with malicious applications and can therefore serve as useful features for malware detection.
These behavioural features can be represented structurally using call graphs. A call graph is a directed graph in which nodes represent API calls, and edges indicate the relationships between them [HHR21]. In the context of Android applications, such graphs capture how different components interact during execution. By modelling applications as call graphs, both the individual operations and their relationships can be preserved, enabling graph neural networks to learn patterns that distinguish between benign and malicious behaviour.

2.4 Graph Neural Networks

A GNN is a neural network that adds a graph structure to the data. A neural network is a computational model that processes information through a large number of interconnected artificial neurons working simultaneously, which is inspired by the function of the human brain [DKK+12]. Graph-structured data represent objects as nodes and the relationships between them as edges, while node attributes provide additional information about each object [LLSvL24]. The GNN has a layered architecture, in which a direct one-on-one relation between the neural network nodes and graph nodes in each layer is possible [Agg23]. GNNs have the ability to learn discriminative feature representations for graphs and have shown substantially progressive results for graph-level anomaly detection [ZSSA22].

2.4.1 Transductive and Inductive Learning

Inductive and transductive learning frameworks both aim to predict unknown labels, but differ in that inductive learning generalises to unseen data, whereas transductive learning is limited to the data observed during training.
Transductive learning exploits both labelled and unlabelled data within a given dataset, leveraging the relationships between all samples to infer labels [BBS+16, RTD+18]. The model operates on a fixed dataset and can use information from the entire graph during both the learning and prediction phases [CRBS22]. In this setting, a prediction for a test instance $x \in \mathcal{T}$ is made by a procedure T that takes both the training set $\mathcal{L}$ and the instance x as input [RTD+18]. As a result, transductive models rely on the structure of the full dataset and are typically less complex, but do not generalise

well to new, unseen data outside this setting.
In contrast, inductive learning learns a general mapping from input features to output labels using only the labelled training data [BBS+16]. The model I_w, parameterised by weights w, is trained by minimising a loss function over the training set, allowing it to capture the underlying data distribution [CRBS22]. Once trained, the model can be applied to new instances without requiring access to the original training data. This enables predictions on unseen data.
In summary, inductive learning uses labelled data to learn generalisable patterns through parameter optimisation, whereas transductive learning uses both labelled and unlabelled data to propagate information across a fixed dataset. GCN was originally proposed for transductive node classification, while GraphSAGE was designed to emphasize inductive generalization to unseen nodes. In graph-level classification the practical difference lies in their aggregation mechanisms.

2.4.2 Graph Convolutional Network and GraphSAGE

The APK graphs used in this thesis consist of enormous amounts of information. Therefore, it is important that the neural network is capable of handling huge amounts of information. The principle of GCN is to generate representations of the nodes by summarising the characteristics of the vertex itself and its neighbouring nodes [LZZL24]. Through aggregation of the features of neighbouring nodes during forward propagation, the node features are updated in the next layer of the network [WLTS23]. The GCN learns new feature representations for each node over multiple layers. Starting with the graph input, feature propagation stimulates similar predictions among locally connected nodes and becomes more similar to a multilayer perceptron (MLP). Each layer is associated with a learned weight matrix, and the hidden feature representation is linearly transformed. After this step, a non-linear activation function is applied pointwise, such as the Rectified Linear Unit (ReLU), before outputting the feature representation [WZdSJ+19]. Overall, a GCN extends traditional neural networks to graph-structured data by iteratively aggregating and transforming information from neighbouring nodes to learn meaningful node- and graph-level representations.
However, GraphSAGE (SAmple and aggreGatE) is a framework for inductive representation learning on large graphs derived from the GCN framework. Instead of learning embeddings for specific nodes, GraphSAGE learns a neighbourhood aggregation function that can be applied to previously unseen nodes or graphs. By incorporating node features in the learning algorithm, it simultaneously learns the topological structure of the node's neighbourhood as well as the distribution of node features in the neighbourhood [HYL17]. The forward propagation technique creates node embeddings assuming that the GraphSAGE model parameters have already been learned [HMH+21]. The assumed parameters for the aggregator functions aggregate information from neighbourhood nodes, as well as a set of weight matrices, which are used to propagate information between different layers of the model. Common aggregator functions include mean, Long Short-Term Memory (LSTM), and pooling aggregators, which differ in how neighbourhood information is combined. Aggregator functions collect information from neighbouring nodes at different distances. The graph-based loss function encourages nearby nodes to have similar representations, while enforcing that the representations of disparate nodes are highly distinct [HYL17].

3 Related work

Malware detection has been widely studied, with a variety of approaches proposed to identify malicious applications. Existing work can broadly be categorised into static, dynamic, and hybrid analysis methods, which differ in how application behaviour is analysed. More recently, research has increasingly explored graph-based representations and graph neural networks (GNNs) to better capture structural patterns within applications.

3.1 Malware Detection Methods

Malware detection methods can be divided into static, dynamic, and hybrid approaches, each with distinct advantages and limitations.

3.1.1 Static Analysis

Static analysis detects malware without executing the application [ESKK08]. Instead, it relies on analysing decompiled or disassembled code, such as the `AndroidManifest.xml` file or smali code, to extract relevant features.
Although static analysis is efficient and scalable, it can be time-consuming when manual inspection is involved and is sensitive to code obfuscation techniques [FYA+25, OMNER19]. However, because it does not require execution, it is less dependent on the analysis environment and avoids issues related to runtime evasion [FYA+25].
Several studies apply machine learning techniques to statically extracted features. DroidDet [ZYZ+18] combines statistical feature analysis and dimensionality reduction using principal component analysis with the Random Forest classifier to distinguish between benign and malicious applications. MsDroid [HLW+22] extracts features including call graphs, opcodes, and permissions, and applies a graph neural network to focus on malicious subgraph structures. By modelling local malicious snippets within call graphs, MsDroid demonstrates improved robustness compared to traditional approaches.

3.1.2 Dynamic Analysis

Dynamic analysis detects malware by executing applications in a controlled environment, such as a sandbox or virtual machine, to observe runtime behaviour [DTV+15, OMNER19].
Compared to static analysis, dynamic analysis is more robust against obfuscation techniques, as it observes actual execution behaviour rather than relying on code structure [OMNER19]. However, it is computationally expensive and may fail to trigger all malicious behaviours if they depend on specific execution conditions.
DMalNet [LCZ+22] is an example of a dynamic analysis framework that converts API call sequences into graph representations for classification, demonstrating the effectiveness of behaviour-based modelling. DroidRanger [ZWZJ12] detects both known and unknown malware by combining permission-based filtering with dynamic analysis. For unknown threats, it uses heuristic filtering and runtime behaviour analysis, such as dynamically loading and executing code from remote sources, to identify suspicious applications [AGP14].

3.1.3 Hybrid Analysis

Hybrid analysis combines static and dynamic techniques to leverage the strengths of both approaches [DTV+15]. This allows systems to benefit from the efficiency of static analysis while incorporating the robustness of dynamic behaviour analysis.
For example, DeepAMD [IRJ+21] applies a two-stage hybrid approach in which applications are first analysed statically and then further examined dynamically if suspicious. This enables accurate classification of both malware presence and malware family. Similarly, other studies [HKB20] combine machine learning techniques such as gradient boosting and feature selection to improve detection performance.
While hybrid approaches often achieve higher accuracy, they introduce additional computational overhead and system complexity, making them less suitable for large-scale or real-time deployment [DTV+15].

3.2 GNN Malware Detection Frameworks

Recent work in Android malware detection increasingly models applications as graphs, enabling the use of graph neural networks for classification. Applications are commonly represented as call graphs (CGs) or control flow graphs, where nodes correspond to functions, methods, or API calls, and edges encode their relationships [HHR21, FML+20]. By preserving structural and behavioural dependencies, these representations provide a more expressive view of application behaviour than flat feature-based or purely sequential approaches.
On top of these graph representations, several studies have applied Graph Convolutional Networks (GCNs) and related graph-based models to distinguish benign from malicious applications [GCZ21, LZZL24, WLTS23, GZH+24]. These approaches learn structural patterns directly from the relationships between program components, allowing them to capture both local interactions and higher-level graph structure. For instance, GDroid [GCZ21] constructs a graph over applications and APIs and applies a GCN for malware detection, while other studies use function call graphs to represent fine-grained behavioural relations more explicitly [LZZL24, WLTS23]. MsDroid [HLW+22] further shows that local malicious snippets and subgraph structures can be particularly informative, indicating that effective malware detection may depend on both whole-graph information and smaller discriminative regions.
Earlier behaviour-based methods such as MaMaDroid [OMA+19] model abstracted API call sequences using Markov chains. While successful, these approaches focus primarily on sequential transitions between API abstractions and do not directly learn from graph topology. GNN-based frameworks offer a broader alternative by combining structural relations with learned node representations, which can yield a richer characterisation of application behaviour.
Despite this progress, most graph-based Android malware detection studies have concentrated on transductive architectures, especially GCNs [KW17, FML+20, WLTS23]. Such models perform well when trained and evaluated within the observed graph setting, but they are less naturally designed for deployment on entirely unseen graphs. This matters in malware detection, where systems must continually assess newly encountered applications.
Inductive models such as GraphSAGE [HYL17] are therefore particularly relevant, as they learn aggregation functions that can generalise to unseen graphs. This makes them well suited to realistic malware detection scenarios in which test applications are not part of the training graph

[VGCBS10, CRBS22]. Although prior work has demonstrated the value of graph representations in Android malware detection [FML+20, GCZ21, GZH+24], inductive GNNs remain less extensively studied in this area. Consequently, their practical advantages are promising but still insufficiently examined through controlled comparison with transductive alternatives.

3.3 Evaluation Metrics

Traditional evaluation of malware detection models focuses on classification performance using metrics such as accuracy, precision, recall, F1-score, and ROC-AUC. While these metrics provide insight into predictive performance, they do not capture how or why a model makes its decisions. Recent work therefore incorporates explanation-based evaluation metrics to assess the quality and reliability of model explanations. Within the PyTorch Geometric framework, such metrics are inspired by approaches including GNNExplainer [YBY+19] and the GraphFramEx framework [AYZ+24].

Fidelity-based metrics, such as `fidelity` and `fidelity_curve_auc`, measure the extent to which identified subgraphs preserve the original model prediction, reflecting explanation faithfulness. In contrast, *unfaithfulness* evaluates how strongly model predictions degrade when important structures are removed, providing a complementary perspective.

Additionally, the `characterization score` measures whether explanations highlight meaningful and class-relevant substructures, addressing the semantic quality of explanations. Together, these metrics provide a more comprehensive evaluation of GNN behaviour by assessing both predictive performance and interpretability.

However, the use of these explanation metrics in malware detection is still limited, and their practical usefulness should be further explored.

3.4 Research Importance

Despite ongoing progress in malware detection, several aspects remain open for further investigation. Many graph-based approaches rely on transductive models, which may limit their ability to generalise to previously unseen applications. In addition, direct comparisons between inductive and transductive GNN models in the context of Android malware detection are relatively limited. Furthermore, while explanation-based evaluation metrics have been proposed, their application within this domain is still emerging.

This thesis contributes to this area by comparing a transductive model with an inductive model under consistent experimental conditions. Additionally, explanation-based evaluation metrics are incorporated to provide further insight into model behaviour. The aim is to explore whether these approaches offer practical advantages in the context of Android malware detection.

4 Methodology

This section describes the approach used to compare the inductive and transductive graph neural network models for Android malware detection. As illustrated in Figure 1, the proposed workflow consists of data acquisition, call graph construction, node embedding generation, graph-level classification with two GNN architectures, and evaluation based on standard classification and explanation metrics.

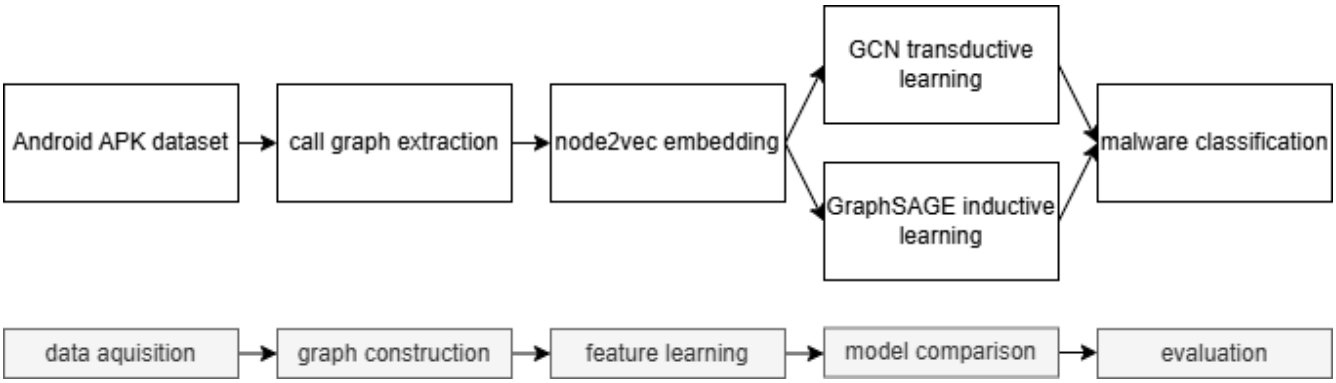

Figure 1: Project pipeline

4.1 Data Acquisition

This thesis is based on Android application data. The datasets are collected for the experiments through Androzoo [ABKLT16]. Androzoo is a continuously growing collection of apps gathered from various sources, including the official Google Play Store. This data consists solely of the raw APK folders and a CSV file. The APK folders consist of app information such as Dalvik bytecode, resources, assets, and the `AndroidManifest.xml` file. These APKs are labelled in a CSV file with information such as:

`sha256,sha1,md5,dex_date,apk_size,pkg_name,vercode,vt_detection,vt_scan_date, dex_size,markets`.

To identify the APK as malicious or benign, the data from `vt_detection` and `sha256` are sufficient. `vt_detection` indicates whether the APK is malicious or benign, if it is higher than 1, then it is classified as malicious. `sha256` is a unique signature for an APK, which is used to identify the APK.
A total of 1,000 APKs were extracted, consisting of 500 benign and 500 malicious applications, based on the labels provided in the associated CSV file. Due to the large size and computational demands of APK processing, the dataset size was kept relatively small to reduce preprocessing and model training time. The balanced composition was adopted to mitigate class imbalance in the malware detection task and to enable a clearer comparison of model performance.

4.2 Graph Construction

The raw APK folders cannot be directly used as input for graph neural networks and therefore require preprocessing. Each APK is converted into a call graph using `Androguard`, an open-source Python tool for reverse engineering Android applications [DG]. Androguard performs static analysis on the application bytecode and generates a call graph in which the nodes represent methods and edges represent calls between methods [HHR21]. This representation captures how different parts of the application interact, providing a structural view of its behaviour. The resulting graphs are stored in `.gml` format, which preserves both the graph structure and associated attributes.
A custom script is used to automate this process for all APK files. For each APK, Androguard is executed via the command line to generate the corresponding call graph. If a graph has already been created, it is skipped to avoid redundant computation.
Formally, each application is represented as a graph $G = (V, E)$, where V denotes the set of methods and E denotes the set of edges representing method calls [FML+20]. These graphs are used as input for the feature learning and graph neural network models.

4.3 Feature Learning

After the graphs are constructed, it is necessary that the call graph is transformed into vectors such that the framework is understood by the model. A method to convert the graph into vectors is through node embedding. To create embeddings of the graph into a vector representation while minimising information loss, the Node2Vec framework is applied. The vectors created through Node2Vec represent the topology of the graph and the relationships between the nodes. The algorithm takes random walks from a given node, which capture the local neighbourhood structure and learn numerical vector representations [HMBMO25]. In other words, Node2Vec learns feature representations that capture structural patterns and neighbourhood similarity, providing informative node features for subsequent learning models. The base of the algorithm used for this thesis is built upon the example code of [GL16].
Several parameters must be considered when running Node2Vec. In this thesis, the following configuration is used:

```
n2v_model = Node2Vec(graph, dimensions=32, walk_length=20, num_walks=50,
                                                    workers=2)
```

`graph` refers to the call graph generated using Androguard. `dimensions` refers to the number of dimensions used. Dimensions can range from low (8–16), medium (32–64), and large (128–256). A medium dimension of 32 is chosen to keep a balance between efficiency and expressiveness, while capturing both local and global structure. Node2Vec is based on random walks passing through the nodes starting from a given node. `walk_length` defines the number of nodes visited in a single random walk starting from a given node. Short walks capture immediate API usage patterns, while longer walks capture control-flow or call chains across components, in other words the local or global structure of the graph. To keep a balance between the local and global context and to minimise runtime, a medium walk length of 20 is chosen. `num_walks` determines how many random walks are initiated per node. Too few walks may underrepresent the graph, while too many increase redundancy. This thesis uses 50 walks per node to achieve sufficient coverage without excessive repetition. Finally, `workers` indicates the number of CPU cores used for parallelisation. Due to

using the remote SSH server of LIACS, the maximum permitted number of CPU cores is two. To speed up the process, the number of cores can be increased; this does not influence the model's behaviour. This feature learning step is included in the preprocessing pipeline to give each node meaningful features, so the GNN can learn patterns in the graph instead of only using connections between nodes.

4.4 Model Comparison

The generated Node2Vec embeddings are used as input for the graph neural network models. Each APK is represented as a graph, making the task a graph-level binary classification problem in which the model predicts whether an application is benign or malicious.
Two graph neural network architectures are compared in this work: GCN and GraphSAGE. The comparison is designed to isolate the effect of the aggregation mechanism as much as possible. Therefore, both models use the same overall architecture, consisting of two graph convolution layers, ReLU activations, dropout, global mean pooling, and a final linear classifier. The comparison is focused on the difference between GCN's transductive convolution-based aggregation and GraphSAGE's inductive neighbourhood aggregation.
Both models are trained using the Adam optimiser with the same learning rate and weight decay settings. Adam was selected after preliminary experiments showed slightly better performance than stochastic gradient descent (SGD) with momentum on the validation data. In addition, Adam adapts the learning rate individually for each parameter, which can improve optimisation efficiency and convergence when training graph neural networks on relatively small datasets.
To support this comparison, the extracted data is first converted into a format that can be used consistently across all experiments. The preprocessing pipeline converts each APK into a consistent graph representation that can be used across all experiments. Node embeddings are matched to graph nodes and transformed into feature matrices, after which the graphs and labels are stored in a uniform format for PyTorch Geometric. During training, node features are processed through two graph convolution layers. ReLU activations introduce non-linearity, while dropout is applied to reduce overfitting. The resulting node embeddings are then aggregated into a graph-level representation using global mean pooling. Mean pooling was selected because APK call graphs vary substantially in size. By averaging node representations, mean pooling reduces the influence of graph size differences between applications. In contrast, sum pooling may bias representations toward larger graphs, while attention-based pooling introduces additional complexity and trainable parameters that may increase overfitting on a relatively limited dataset. The resulting graph representation is passed to a final linear classifier that outputs the benign or malicious prediction. Because both models share the same overall architecture and training setup, the comparison primarily evaluates the effect of the convolution-based aggregation and the neighbourhood aggregation mechanism used by GCN and GraphSAGE.

4.5 Evaluation

The evaluation procedure was designed to provide a fair comparison between GCN and GraphSAGE under identical experimental conditions. The dataset was divided into training, validation, and test sets using a stratified 70/15/15 split to preserve the class distribution across all subsets.

Both models were trained and evaluated on the same data partition to ensure that any observed differences could be attributed to the model architecture rather than variations in the data.
Model selection was based on validation ROC-AUC. This metric was chosen because it evaluates performance across different classification thresholds and is less sensitive to a specific decision boundary than accuracy alone. The model achieving the highest validation ROC-AUC was retained, while early stopping was used to reduce overfitting.
Final performance was assessed on the held-out test set using accuracy, precision, recall, F1-score, ROC-AUC, and confusion matrix statistics. Together, these metrics provide a broader view of classification performance than any single metric alone.
In addition to predictive performance, the models were evaluated in terms of interpretability. Since malware detection systems may be used in security-sensitive environments, interpretability was evaluated alongside predictive performance to assess whether model decisions could be meaningfully explained. For this purpose, GNNExplainer was applied to a subset of test graphs. To improve explanation quality, only correctly classified graphs with a confidence score above 0.60 and sufficient graph structure were considered. Restricting the analysis to confident and correctly classified graphs reduces the likelihood that explanation quality is affected by prediction errors rather than the behaviour learned by the model. The generated explanations were evaluated using fidelity, characterization score, fidelity-curve AUC, and unfaithfulness metrics.
Overall, the evaluation was performed on two levels. First, GCN and GraphSAGE were compared based on their malware detection performance. Second, the resulting models were analysed from an interpretability perspective to examine how well their predictions could be explained.

5 Results

This section presents the results of the inductive and transductive GNN models for Android malware detection. A total of 985 correctly processed samples were used, of which 492 were malicious and 493 were benign.

5.1 Validation Performance

During training, both models showed a consistent decrease in training loss, indicating that they were able to learn somewhat meaningful representations from the data. This training loss behaviour is illustrated in Figure 2, while the corresponding validation ROC-AUC and F1-score trends are shown in Figure 3. The GCN model was trained for fewer epochs because early stopping was triggered once the validation performance no longer improved, whereas GraphSAGE continued training for more epochs before reaching this criterion.

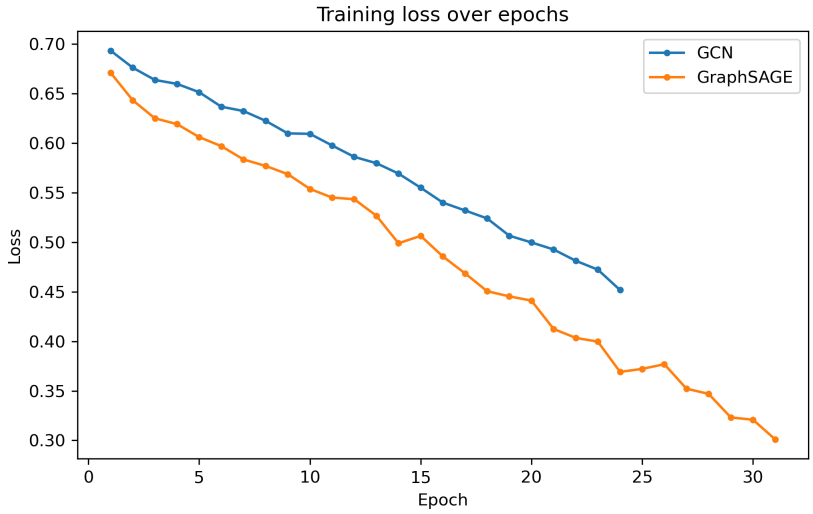

Figure 2: Training loss over epochs for GCN and GraphSAGE.

For GCN, the validation F1-score fluctuated substantially during training, ranging from approximately 0.353 to 0.640. The validation ROC-AUC improved gradually and reached its highest value of 0.619 at epoch 9, after which performance did not improve further. Although the model learned useful patterns from the graph data, the fluctuations in validation F1-score indicate that its classification behaviour remained relatively unstable across epochs. GraphSAGE demonstrated stronger validation performance overall. Its validation ROC-AUC consistently remained above that of GCN for most epochs and reached a maximum of 0.723 at epoch 16. The validation F1-score remained relatively strong throughout training and reached a maximum of 0.680 at epoch 16. These results suggest that GraphSAGE learned more discriminative graph representations than GCN and achieved consistently better validation performance during training.

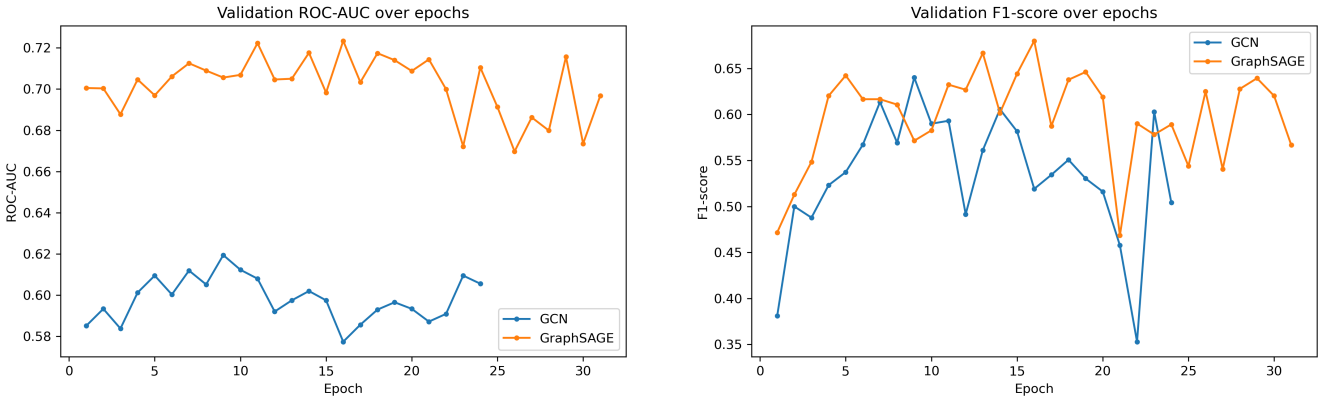

Figure 3: Validation ROC-AUC and validation F1-score over training epochs for GCN and GraphSAGE.

Overall, GraphSAGE achieved higher validation ROC-AUC values than GCN, suggesting improved class separability during training. GraphSAGE showed a clearer ability to distinguish between malicious and benign APKs, indicating that its inductive neighbourhood aggregation captured more informative structural patterns than the transductive convolution approach used in GCN.

5.2 Test Set Performance

Table 1 presents the final evaluation results of GCN and GraphSAGE on the test set.

Metric	**GCN**	**GraphSAGE**
Accuracy	0.5733	0.6467
Precision	0.5663	0.6410
Recall	0.6267	0.6667
F1-score	0.5949	0.6536
ROC-AUC	0.5785	0.6404
True Positives	47	50
False Positives	36	28
True Negatives	39	47
False Negatives	28	25

Table 1: Comparison of test performance between GCN and GraphSAGE.

The GCN model achieved an accuracy of 0.5733, with a precision of 0.5663 and a recall of 0.6267. This resulted in an F1-score of 0.5949 and a ROC-AUC of 0.5785. The confusion matrix values show 47 true positives, 39 true negatives, 36 false positives, and 28 false negatives. These results indicate that the model was able to distinguish between benign and malicious applications better than random classification. Although the threshold-based metrics remained moderate, the ROC-AUC suggests that the model was relatively effective at ranking malicious applications higher than benign applications across different classification thresholds.

GraphSAGE achieved slightly stronger classification performance on most threshold-based metrics. It reached an accuracy of 0.6467, with a precision of 0.6410 and a recall of 0.6667. This resulted in an F1-score of 0.6536. The model produced 50 true positives and 47 true negatives, while reducing the number of false positives to 28 and false negatives to 25. These results indicate somewhat more balanced classification behaviour than GCN. The improvements across accuracy, precision, recall, and F1-score suggest that GraphSAGE made slightly more correct classifications overall at the selected decision threshold. However, despite these improvements, its ROC-AUC of 0.6404 was only modestly higher than that of GCN, indicating somewhat stronger ranking performance across multiple thresholds.
Overall, GraphSAGE outperformed GCN on accuracy, precision, recall, F1-score, and ROC-AUC. This suggests that GraphSAGE provided stronger final class predictions at the selected decision threshold while also demonstrating better ranking ability across thresholds. As a result, the comparison indicates a consistent performance advantage for GraphSAGE across all evaluated classification metrics.

5.3 Model Interpretability

To analyse which structural components influenced the classification decisions, GNNExplainer was applied to a subset of test samples for both models. Only correctly classified samples with sufficient confidence were considered to ensure more reliable explanations. To decrease runtime, for both models only 40 explanations were generated from a filtered subset of correctly classified and sufficiently confident test graphs.

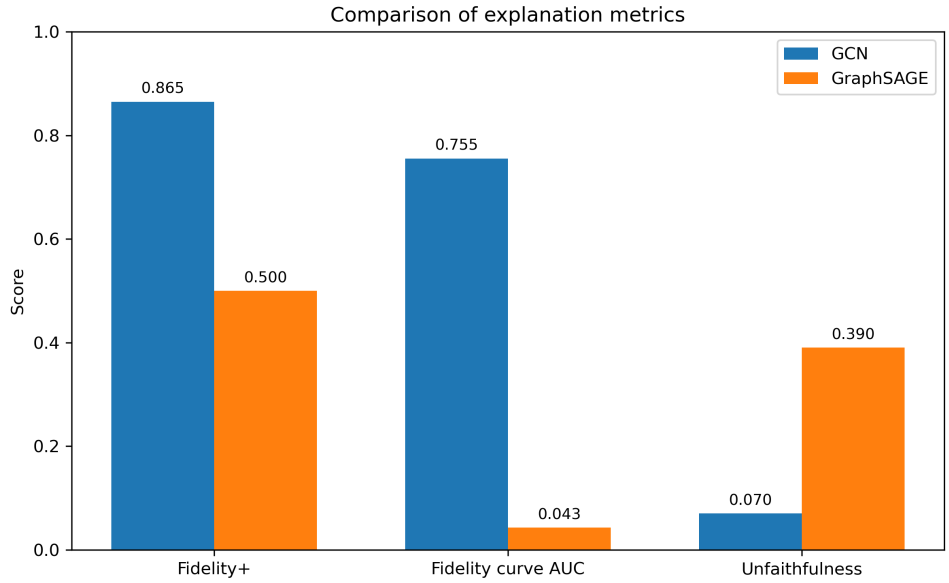

Figure 4: Comparison of explanation metrics between GCN and GraphSAGE.

For the GCN model, the explanation metrics indicate comparatively stronger and more consistent explanations than the alternative model. The mean `mean_fidelity_pos` was 0.865, while `mean_fidelity_neg` was 0.270. In principle, this suggests that the selected explanation masks

captured components relevant to the prediction, whereas non-selected components contributed relatively little.

The `mean_characterization_score` was 0.621, which indicates a strong separation between relevant and irrelevant graph structure according to the metric definition. In addition, the `mean_fidelity_curve_auc` was approximately 0.755, suggesting that highly ranked edges contributed substantially to the prediction. The mean `mean_unfaithfulness` score was 0.070, which is relatively low and therefore provides additional support for explanation reliability.

For the GraphSAGE model, the explanation metrics were considerably weaker than those observed for the GCN model. The mean `mean_fidelity_pos` was 0.500, while `mean_fidelity_neg` was also 0.500. In principle, this suggests that the explanation masks did not clearly distinguish between graph components that were important for the prediction and those that were not.

The `mean_characterization_score` was 0.000, indicating that the metric did not identify a meaningful separation between relevant and irrelevant graph structure. In addition, the `mean_fidelity_curve_auc` was approximately 0.043. This metric could only be computed for 21 of the 40 explanations, while 19 explanations did not produce a valid score. The low average value suggests that the highest-ranked edges contributed only limited predictive information according to the evaluation procedure. The mean `mean_unfaithfulness` score was 0.390, which is substantially higher than that observed for the GCN model and therefore indicates weaker alignment between the generated explanations and the model's decision process.

The explainability results should still be interpreted cautiously. Graph explanation evaluation on large real-world call graphs remains an active research challenge, and some metrics showed limited robustness. In particular, the inability to compute a valid fidelity curve score for nearly half of the GraphSAGE explanations suggests that the explanation procedure did not consistently produce stable outputs for this model. Overall, the explainability results favour GCN over GraphSAGE. While GraphSAGE achieved stronger performance on most threshold-based classification metrics, the GCN model produced explanations that were more faithful, more informative, and more consistent under the current evaluation setup. This suggests that, in this experiment, stronger predictive performance did not necessarily translate into stronger interpretability.

6 Discussion

The final test results showed that GraphSAGE achieved stronger predictive performance than GCN across all evaluated classification metrics. GraphSAGE outperformed GCN on accuracy (0.6467 vs. 0.5733), precision (0.6410 vs. 0.5663), recall (0.6667 vs. 0.6267), F1-score (0.6536 vs. 0.5949), and ROC-AUC (0.6404 vs. 0.5785). It also produced fewer false positives and false negatives, indicating a more balanced classification behaviour. Overall, these findings suggest that GraphSAGE was better able to distinguish between benign and malicious applications under the current experimental setup. One possible explanation for this improvement is GraphSAGE's inductive neighbourhood aggregation mechanism. Unlike GCN, GraphSAGE learns how to aggregate information from neighbouring nodes rather than relying on the complete graph structure during training. This inductive design may enable the model to capture more informative structural patterns and generalise more effectively to previously unseen application graphs.

In contrast, stronger predictive performance did not correspond to stronger explanations. Although GraphSAGE achieved superior classification results, GCN consistently produced stronger explanation scores across all evaluated explanation metrics. This suggests that predictive performance and explainability may involve a trade-off in graph-based malware detection, where improvements in classification performance do not necessarily translate into improved model interpretability. One possible explanation is that the representations learned by GraphSAGE are less localised than those learned by GCN, which may make them more difficult for GNNExplainer to explain effectively. However, these findings should be interpreted with caution because explanation methods for graph neural networks are still being developed and can have limitations.

Several broader limitations should also be acknowledged. First, although the dataset of 985 processed APKs was sufficient for a controlled comparison, it remains relatively small compared to the scale and diversity of the real Android ecosystem. In addition, the interpretability analysis was based on only 40 explanations generated from a filtered subset of correctly classified test graphs that exceeded the confidence threshold. Graphs that were too small or unsuitable for masking were excluded. Although this filtering improved the quality of the generated explanations, the limited number of explanations means that the interpretability results should be regarded as exploratory rather than comprehensive.

Second, the study relied exclusively on static analysis combined with Node2Vec-based structural features. While effective for capturing call graph structure, this approach cannot capture runtime behaviour such as reflection, dynamic code loading, or advanced obfuscation techniques that may be exploited by modern Android malware.

Finally, the conclusions are based on a single preprocessing pipeline and experimental configuration. Different datasets, node representations, graph neural network architectures, explanation methods, or hyperparameter settings may produce different outcomes.

7 Conclusion

This thesis investigated how the inductive graph neural network GraphSAGE compares with the transductive graph neural network GCN in terms of predictive performance and interpretability for Android malware detection. To answer this research question, both models were evaluated under identical experimental conditions on 985 successfully processed APKs, consisting of 492 malicious and 493 benign applications. To ensure a fair comparison, both models used the same training, validation, and test split, hidden dimension, dropout rate, optimiser, pooling strategy, and training procedure, differing only in their graph aggregation mechanism.

The experimental results showed that GraphSAGE consistently outperformed GCN across all evaluated classification metrics, including accuracy, precision, recall, F1-score, and ROC-AUC. These findings indicate that GraphSAGE provides a more effective representation of Android application call graphs and is better suited for classifying previously unseen applications under the conditions examined in this study.

In addition to predictive performance, this thesis evaluated model interpretability using GNNExplainer and multiple explanation quality metrics. Although GraphSAGE achieved superior classification performance, GCN consistently produced stronger and more reliable explanation scores. This suggests that improved predictive performance does not necessarily correspond to improved interpretability, highlighting a trade-off between classification performance and explainability in graph neural networks.

Overall, the results indicate that GraphSAGE is the stronger model for Android malware detection when predictive performance is the primary objective, while GCN remains the more interpretable model according to the selected explanation metrics. Rather than identifying a universally superior architecture, this comparison demonstrates that the preferred model depends on whether predictive performance or interpretability is prioritised.

Future work should evaluate these models on larger and more diverse Android malware datasets, investigate alternative node representations beyond Node2Vec, explore more robust explanation methods for large graph neural networks, and incorporate dynamic analysis features to capture runtime behaviour. Additionally, transfer learning with inductive graph neural networks represents a promising direction for improving generalisation to emerging malware families and supporting more scalable Android malware detection systems.

Abbreviations

The following abbreviations are used in this paper:

APK	Android Package Kit
API	Application Programming Interface
GNN	Graph Neural Network
GCN	Graph Convolutional Network
ReLu	Rectified Linear Unit
MLP	Multilayer Perceptron
Adam	Adaptive Moment Estimation

References

[ABKLT16] Kevin Allix, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. Androzoo: Collecting millions of android apps for the research community. In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, pages 468–471, New York, NY, USA, 2016. ACM.

[Agg23] Charu Aggarwal. *Graph Neural Networks*, pages 361–387. Springer International Publishing, Cham, 2023.

[AGP14] Anshul Arora, Shree Garg, and Sateesh K. Peddoju. Malware detection using network traffic analysis in android based mobile devices. In *2014 Eighth International Conference on Next Generation Mobile Apps, Services and Technologies*, pages 66–71, 2014.

[AYZ+24] Kenza Amara, Rex Ying, Zitao Zhang, Zhihao Han, Yinan Shan, Ulrik Brandes, Sebastian Schemm, and Ce Zhang. Graphframex: Towards systematic evaluation of explainability methods for graph neural networks, 2024.

[BBS+16] Monica Bianchini, Anas Belahcen, Franco Scarselli, Marco Maratea, Marco Gori, Giovanni Adorni, and Stefano Cagnoni. A comparative study of inductive and transductive learning with feedforward neural networks. In *AIIA 2016 Advances in Artificial Intelligence*, volume 10037 of *Lecture Notes in Computer Science*, pages 283–293. Springer International Publishing AG, Switzerland, 2016.

[BK24] Ahmed Bensaoud and Jugal Kalita. Cnn-lstm and transfer learning models for malware classification based on opcodes and api calls. *Knowledge-Based Systems*, 290:111543, 2024.

[CRBS22] Giorgio Ciano, Alberto Rossi, Monica Bianchini, and Franco Scarselli. On inductive-transductive learning with graph neural networks. *IEEE transactions on pattern analysis and machine intelligence*, 44(2):758–769, 2022.

[Dev] Android Developers. App manifest overview. https://developer.android.com/guide/topics/manifest/manifest-intro [Accessed: (28-05-2025)].

[DG] Anthony Desnos and Geoffroy Gueguen. Androguard: A full python tool to play with android files. https://github.com/androguard/androguard. Accessed: 2025-03-14.

[DKK+12] AD Dongare, RR Kharde, Amit D Kachare, et al. Introduction to artificial neural network. *International Journal of Engineering and Innovative Technology (IJEIT)*, 2(1):189–194, 2012.

[DTV+15] Anusha Damodaran, Fabio Di Troia, Corrado Aaron Visaggio, Thomas H. Austin, and Mark Stamp. A comparison of static, dynamic, and hybrid analysis for malware detection. *Journal of Computer Virology and Hacking Techniques*, 13(1):1–12, December 2015.

[ESKK08] Manuel Egele, Theodoor Scholte, Engin Kirda, and Christopher Kruegel. A survey on automated dynamic malware-analysis techniques and tools. *ACM Comput. Surv.*, 44(2), March 2008.

[FML+20] Pengbin Feng, Jianfeng Ma, Teng Li, Xindi Ma, Ning Xi, and Di Lu. Android malware detection based on call graph via graph neural network. In *2020 International Conference on Networking and Network Applications (NaNA)*, pages 368–374, 2020.

[FYA+25] Shota Fujii, Rei Yamagishi, Habtamu Abie, Silvio Ranise, Luca Verderame, Marek Pawlicki, Michał Choraś, Basel Katt, Paweł Ksieniewicz, Rita Ugarelli, Harsha Kalutarage, Rafał Kozik, Sandeep Pirbhulal, Enrico Cambiaso, Isabel Praça, Michał Woźniak, Ankur Shukla, Joaquin Garcia-Alfaro, and Naoto Yanai. Feasibility study for supporting static malware analysis using llm. In *Computer Security. ESORICS 2024 International Workshops*, Lecture Notes in Computer Science, pages 5–28. Springer Nature Switzerland, Cham, 2025.

[GCZ21] Han Gao, Shaoyin Cheng, and Weiming Zhang. Gdroid: Android malware detection and classification with graph convolutional network. *Computers security*, 106:102264–, 2021.

[GL16] Aditya Grover and Jure Leskovec. node2vec: Scalable Feature Learning for Networks — arxiv.org. https://arxiv.org/abs/1607.00653, 2016.

[GZH+24] Jintao Gu, Hongliang Zhu, Zewei Han, Xiangyu Li, and Jianjin Zhao. Gsedroid: Gnn-based android malware detection framework using lightweight semantic embedding. *Computers Security*, 140:103807, 2024.

[HHR21] Rasmus Hagberg, Martin Hell, and Christoph Reichenbach. Using program analysis to identify the use of vulnerable functions. In *SECRYPT*, pages 523–530, 2021.

[HKB20] Raden Budiarto Hadiprakoso, Herman Kabetta, and I Komang Setia Buana. Hybrid-based malware analysis for effective and efficiency android malware detection. In *2020 International Conference on Informatics, Multimedia, Cyber and Information System (ICIMCIS)*, pages 8–12, 2020.

[HLW+22] Yiling He, Yiping Liu, Lei Wu, Kui Ren, and Zhan Qin. Msdroid: Identifying malicious snippets for android malware detection. *IEEE Transactions on Dependable and Secure Computing*, PP:1–1, 01 2022.

[HMBMO25] Mohammad Sarwar Hossain Mollah, Mohd Fadzli Bin Marhusin, and Syaril Nizam Omar. A robust malware detection framework using control flow graphs, node2vec, and gcn integration. In *2025 International Conference on Electrical, Computer and Communication Engineering (ECCE)*, pages 1–6, 2025.

[HMH+21] Parisa Hajibabaee, Masoud Malekzadeh, Maryam Heidari, Samira Zad, Ozlem Uzuner, and James H Jones. An empirical study of the graphsage and word2vec algorithms for graph multiclass classification. In *2021 IEEE 12th Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON)*, pages 0515–0522, 2021.

[HYL17] William L. Hamilton, Rex Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *NIPS*, 2017.

[IRJ+21] Syed Ibrahim Imtiaz, Saif ur Rehman, Abdul Rehman Javed, Zunera Jalil, Xuan Liu, and Waleed S. Alnumay. Deepamd: Detection and identification of android malware using high-efficient deep artificial neural network. *Future generation computer systems*, 115:844–856, 2021.

[Kas25] Kaspersky. Kaspersky report: Attacks on smartphones increased in the first half of 2025, 4-9-2025.

[Kha25] Mohammed Khalil. 50+ Malware Statistics 2025: Attacks, Trends and Infections — deepstrike.io. https://deepstrike.io/blog/Malware-Attacks-and-Infections-2025, 2025. [Accessed 11-07-2025].

[KW17] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *International Conference on Learning Representations (ICLR)*, 2017.

[LCZ+22] Ce Li, Zijun Cheng, He Zhu, Leiqi Wang, Qiujian Lv, Yan Wang, Ning Li, and Degang Sun. Dmalnet: Dynamic malware analysis based on api feature engineering and graph learning. *Computers Security*, 122:102872, 2022.

[LLSvL24] Zhong Li, Sheng Liang, Jiayang Shi, and Matthijs van Leeuwen. Cross-domain graph level anomaly detection. *IEEE Transactions on Knowledge and Data Engineering*, 36(12):7839–7850, 2024.

[LZZL24] Xiaofeng Lu, Jinglun Zhao, Senhao Zhu, and Pietro Lio. Sndgcn: Robust android malware detection based on subgraph network and denoising gcn network. *Expert Systems with Applications*, 250:123922, 2024.

[OMA+19] Lucky Onwuzurike, Enrico Mariconti, Panagiotis Andriotis, Emiliano De Cristofaro, Gordon Ross, and Gianluca Stringhini. Mamadroid: Detecting android malware by

building markov chains of behavioral models (extended version). *ACM Trans. Priv. Secur.*, 22(2), April 2019.

[OMNER19] Ori Or-Meir, Nir Nissim, Yuval Elovici, and Lior Rokach. Dynamic malware analysis in the modern era—a state of the art survey. *ACM Comput. Surv.*, 52(5), September 2019.

[RTD+18] Alberto Rossi, Matteo Tiezzi, Giovanna Maria Dimitri, Monica Bianchini, Marco Maggini, Franco Scarselli, Luca Pancioni, Friedhelm Schwenker, and Edmondo Trentin. Inductive–transductive learning with graph neural networks. In *Artificial Neural Networks in Pattern Recognition*, volume 11081 of *Lecture Notes in Computer Science*, pages 201–212. Springer International Publishing AG, Switzerland, 2018.

[Tea24] Check Point Team. Check Point Research Reports Highest Increase of Global Cyber Attacks seen in last two years – a 30https://blog.checkpoint.com/research/check-point-research-reports-highest-increase-of-global-cyber-attacks-seen-in-2024. [Accessed 07-08-2025].

[VGCBS10] Ricardo Vilalta, Christophe Giraud-Carrier, Pavel Brazdil, and Carlos Soares. *Inductive Transfer*, pages 545–548. Springer US, Boston, MA, 2010.

[WLTS23] Haojie Wu, Nurbol Luktarhan, Gaoqi Tian, and Yangyang Song. An android malware detection approach to enhance node feature differences in a function call graph based on gcns. *Sensors*, 23(10), 2023.

[WZdSJ+19] Felix Wu, Tianyi Zhang, Amauri Holanda de Souza Jr., Christopher Fifty, Tao Yu, and Kilian Q. Weinberger. Simplifying graph convolutional networks, 2019.

[YBY+19] Rex Ying, Dylan Bourgeois, Jiaxuan You, Marinka Zitnik, and Jure Leskovec. Gnnexplainer: Generating explanations for graph neural networks, 2019.

[YLAI17] Yanfang Ye, Tao Li, Donald Adjeroh, and S. Sitharama Iyengar. A survey on malware detection using data mining techniques. *ACM Comput. Surv.*, 50(3), June 2017.

[ZSSA22] Lingxiao Zhao, Saurabh Sawlani, Arvind Srinivasan, and Leman Akoglu. Graph anomaly detection with unsupervised gnns, 2022.

[ZWZJ12] Yajin Zhou, Zhi Wang, Wu Zhou, and Xuxian Jiang. Hey, you, get off of my market: detecting malicious apps in official and alternative android markets. In *NDSS*, volume 25, pages 50–52, 2012.

[ZYZ+18] Hui-Juan Zhu, Zhu-Hong You, Zexuan Zhu, Wei-Lei Shi, and Li Cheng. Droiddet: effective and robust detection of android malware using static analysis along with rotation forest model. *Neurocomputing*, 272:638–646, 01 2018.