MaskDroid: Robust Android Malware Detection with Masked Graph Representations

Jingnan Zheng*
National University of Singapore
Singapore
jingnan.zheng@u.nus.edu

Jiahao Liu*
National University of Singapore
Singapore
jiahao99@comp.nus.edu.sg

An Zhang†
National University of Singapore
Singapore
anzhang@u.nus.edu

Jun Zeng
National University of Singapore
Singapore
junzeng@u.nus.edu

Ziqi Yang§
Zhejing University
China
yangziqi@zju.edu.cn

Zhenkai Liang
National University of Singapore
Singapore
liangzk@comp.nus.edu.sg

Tat-Seng Chua
National University of Singapore
Singapore
chuats@comp.nus.edu.sg

ABSTRACT

Android malware attacks have posed a severe threat to mobile users, necessitating a significant demand for the automated detection system. Among the various tools employed in malware detection, graph representations (*e.g.,* function call graphs) have played a pivotal role in characterizing the behaviors of Android apps. However, though achieving impressive performance in malware detection, current state-of-the-art graph-based malware detectors are vulnerable to adversarial examples. These adversarial examples are meticulously crafted by introducing specific perturbations to normal malicious inputs. To defend against adversarial attacks, existing defensive mechanisms are typically supplementary additions to detectors and exhibit significant limitations, often relying on prior knowledge of adversarial examples and failing to defend against unseen types of attacks effectively.

In this paper, we propose MaskDroid, a powerful detector with a strong discriminative ability to identify malware and remarkable robustness against adversarial attacks. Specifically, we introduce a masking mechanism into the Graph Neural Network (GNN) based framework, forcing MaskDroid to recover the whole input graph using a small portion (*e.g.,* 20%) of randomly selected nodes. This strategy enables the model to understand the malicious semantics and learn more stable representations, enhancing its robustness against adversarial attacks. While capturing stable malicious semantics in the form of dependencies inside the graph structures, we further employ a contrastive module to encourage MaskDroid to learn more compact representations for both the benign and malicious classes to boost its discriminative power in detecting malware from benign apps and adversarial examples. Extensive experiments validate the robustness of MaskDroid against various adversarial attacks, showcasing its effectiveness in detecting malware in real-world scenarios comparable to state-of-the-art approaches.

CCS CONCEPTS

• **Security and privacy** → **Malware and its mitigation**;

KEYWORDS

Android Malware Detection, Adversarial Attacks, Graph Masking, Graph Representation

ACM Reference format:
Jingnan Zheng*, Jiahao Liu*, An Zhang†, Jun Zeng, Ziqi Yang§, Zhenkai Liang, and Tat-Seng Chua. 2024. MaskDroid: Robust Android Malware Detection with Masked Graph Representations. In *Proceedings of IEEE/ACM Automated Software Engineering Conference, Sacramento, CA, US, 2024 (ASE),* 13 pages.
https://doi.org/10.1145/3691620.3695008

1 INTRODUCTION

Android, as one of the most prevalent smartphone operating systems, has dominated over 85% of the mobile OS market share since 2018 [41]. However, its popularity and open nature have also made it a primary target for cyberattacks [1, 44]. For example, Android permits the installation of applications from unverified sources, such as third-party markets, thereby providing attackers with easy means to bundle and distribute malware-infected apps [10]. Android malware, a.k.a., malicious software, has become one of the primary security threats to the Android platform, with the number of malware samples increasing exponentially over the years [5].

To mitigate these threats, machine learning (ML) has been widely adopted to automatically extract malicious patterns from various APK features for Android malware detection [22, 42, 44]. According to the features used, there are two research lines: syntax-based [6, 10, 40, 61] and semantic-based detectors [31, 32, 46, 63, 64].

* Jingnan Zheng and Jiahao Liu contribute equally to this work.
† An Zhang is the corresponding author of this work.
§ Ziqi Yang is affiliated with the State Key Laboratory of Blockchain and Security and the Hangzhou High-Tech Zone (Binjiang) Institute of Blockchain and Data Security.

Syntax-based methods typically utilize discrete features such as permissions, API calls, and intents to model app behaviors. However, these methods might overlook the underlying program semantics, limiting their detection capability [31, 64]. To address this issue, a notable trend is to distill program semantics from apps' graph representations for malware detection. Among these graph-based approaches, Function Call Graphs (FCGs) and their variants are extensively used and have proven effective [31, 46, 63], as they encapsulate the invocation relationship among API calls, offering deep insights about how an app works [38]. For example, MsDroid [31] utilizes code snippets around sensitive API calls in FCGs to model app behaviors and leverages graph neural networks (GNNs) to capture corresponding semantics for malware detection.

While existing graph-based malware detectors have demonstrated promising results in detecting malware, they are vulnerable to adversarial examples crafted by carefully introducing perturbations to malicious inputs [15, 22, 42, 69, 71]. For example, adversaries can modify the most influential edges or nodes in apps' graph representations to evade the decision boundary, thereby undermining the detectors' performance [38, 71]. This is because of the inherent fragility of ML models, where small but intentional perturbations can lead to incorrect predictions with high confidence [11, 17, 24, 45]. Unfortunately, there are few defense mechanisms available to enhance the robustness of graph-based malware detectors against adversarial attacks, except for several general-purpose supplemental strategies, like adversarial training [7, 24, 60]. However, these strategies often require a large number of adversarial examples for training, which is impractical in real-world scenarios [27, 50]. Additionally, they may not be effective against unseen types of adversarial examples. As such, we argue that the current defense mechanisms are insufficient to meet the demands posed by adversarial attacks, highlighting the need for more robust solutions, especially those that enhance the detector itself.

In this paper, we aim to propose a novel graph-based Android malware detector, MaskDroid, that can effectively handle adversarial attacks while maintaining comparable detection accuracy. To achieve this goal, we need to learn stable representations of malicious behaviors that remain consistent even if the input graph is adversarially perturbed. Specifically, malware developers often conceal their intentions by mimicking the behavior of benign apps, embedding a small portion of malicious code within a bulk of benign processes [2, 32]. When representing the app in a graph form, the malicious code appears as a small sub-graph embedded within the overall graph representation. Adversarial examples preserve the codes (*i.e.,* malicious subgraphs) responsible for the malicious operations, while strategically introducing additional nodes or edges to confound the detectors [40]. Based on this observation, we note that if we can learn a stable representation encoding the malicious behavior, we can achieve a robust detector that can effectively detect both malware and adversarial examples.

To guide MaskDroid to learn stable representations of malicious behaviors within the graph representations, our methodology involves introducing uncertainty and forcing the model to contend with it through a *reconstruction module*. Specifically, we first apply a random masking mechanism [29, 58] to the input graph, where a substantial proportion of nodes (*e.g.,* 80%) are masked out. Then we use graph neural networks to encode and decode the masked graph, forcing MaskDroid to recover the masked node features using the remaining nodes (*e.g.,* 20%). By doing so, MaskDroid gains a holistic understanding of the malicious behaviors and develops the ability to generate stable representations even when the input is perturbed, enhancing its robustness against adversarial attacks.

With stable representations that comprehensively grasp the semantics of the malicious structures, we proceed to determine the class (*i.e.,* benign or malicious) of the input graph. Instead of performing direct classification, MaskDroid incorporates a *contrastive module* [67] to enhance its discriminative power. The insight behind this is that samples within the same class can complement each other and should be pulled closer together, while those in different classes should be pushed apart. To this end, we define two proxy representations as anchors for the benign and malicious classes. During the training process, we calculate the distance between the input instance and the two proxies to update their positions. The prediction is made by checking which proxy the instance is closer to. The contrastive module further compresses the representations and refines the decision boundary between benign and malicious classes, enhancing MaskDroid's ability to distinguish malware from benign apps and adversarial examples.

To investigate the effectiveness and robustness of MaskDroid in Android malware detection, we conduct extensive experiments on a comprehensive dataset comprising 102,459 benign and 11,751 malicious apps collected over five years, from 2016 to 2020. We further compare MaskDroid with five state-of-the-art malware detectors, including three graph-based detectors, *i.e.,* MamaDroid [46], MalScan [63], and MsDroid [31], and two syntax-based detectors, *i.e.,* Drebin [10] and RAMDA [40]. Experimental results demonstrate that MaskDroid achieves the most robust performance against adversarial attacks under various settings (e.g., reducing the attack success rate from 41.54% to 32.0%) while maintaining comparable detection accuracy to the state-of-the-art detectors. Through ablation studies, we further validate that each design choice (e.g., contrastive module and reconstruction module) in MaskDroid contributes to its robustness and effectiveness.

In summary, we make the following contributions:

- To the best of our knowledge, we are the first to enhance the robustness of the graph-based malware detection model against adversarial attacks without sacrificing detection accuracy.
- We present MaskDroid, a novel graph-based Android malware detector that utilizes a reconstruction task to guide the model to learn stable representations of malicious behaviors, while incorporating a contrastive module to enhance its discriminative power for malware detection.
- We conduct extensive evaluations against five state-of-the-art approaches on data sourced from AndroZoo [9]. Experimental results show that MaskDroid exhibits superior robustness against adversarial attacks compared to the baselines while maintaining promising detection accuracy. Our codes are available at https://github.com/SophieZheng998/MaskDroid.

2 PRELIMINARIES

In this section, we first introduce commonly used graph representations (*i.e.,* Function Call Graph and its variants) in Android malware

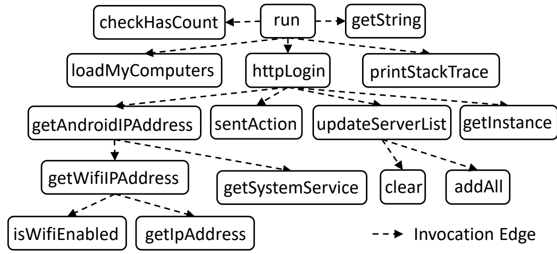


Figure 1: Partial Function Call Graph (FCG) of an app.

detection. Then, we formally formulate the problem of graph-based malware detection and adversarial example attacks.

2.1 Graph Representations

The graph representations of Android apps encode both the semantic and structural information and have been widely used in Android malware detection [31, 31, 32, 46, 63]. Among these, the Function Call Graph (FCG) is a popular representation that captures the caller-callee relationships among the API calls in an app. Figure 1 depicts a partial Function Call Graph (FCG) of a real-world Android application [4], where nodes (*e.g., getWifiIPAddress()*) represent API calls, and the edges denote method invocations (*e.g., updateServerList()* calls *addALL()*). Detectors like MalScan [63] and HomDroid [64] analyze FCGs akin to social networks, leveraging centrality and community detection algorithms to uncover malicious patterns for malware detection.

Additionally, several variants of FCGs have been proposed to model app behaviors. For example, MamaDroid [46] abstracts the nodes of FCGs according to their packages or family names to construct a higher-level, abstracted graph representation. MsDroid [31] uses sensitive API calls (*e.g., getSystemService()*) as seed nodes to generate graph snippets around them, modeling apps as a collection of subgraphs. This is because sensitive behaviors are often carried out by a small proportion of the code requiring the invocation of sensitive API calls to achieve their goals. MsDroid further utilizes the opcode and required permissions of functions as the node features to initialize the graph representation. In our study, we adopt the graph structure proposed in MsDroid due to its simplicity and proven effectiveness in detecting Android malware.

2.2 Problem Formulation

Android Malware Detection. Graph-based Android malware detectors take the graph representation of an app as input and output the probability of it being malicious. Here, we formally define the input graph as $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{X})$, where each node $v \in \mathcal{V}$ represents an API call, and each edge $e_{(u,v)} \in \mathcal{E}$ denotes the invocation from node u to node v. The set $\mathcal{X}$ collects the features of the nodes. The goal of the malware detector is to learn a classifier $C : \mathcal{G} \longrightarrow \{0, 1\}$, where 0 denotes a benign app, and 1 denotes malware.

Adversarial Examples. Adversarial examples are crafted to mislead learning classifiers by introducing perturbations to the input features while preserving the malicious functionalities [15, 37, 38, 54, 71]. In the context of graph-based Android malware detection, these adversarial examples can be represented as deliberately altered target graph-based features to bypass the classifier C. Suppose $\mathcal{P}$ denotes the perturbation operation, and $L(\cdot)$ is the label predicted by the classifier C. Then, the adversarial example $\mathcal{G}'$ can be formulated as $\mathcal{G}' = \mathcal{P}(\mathcal{G}) = \mathcal{G} + \delta$, where δ signifies the perturbations to the graph structure, such as adding nodes and edges. The adversarial manipulation process can be represented as:

$$L(C(\mathcal{G})) \neq L(C(\mathcal{P}(\mathcal{G}))). \tag{1}$$

Intuitively, an ideal Android malware detector should be highly effective in identifying malware while also being robust against adversarial examples. However, current models predominantly emphasize detection effectiveness, often at the expense of robustness [10, 30, 38, 63]. Only a few studies have attempted to improve the robustness of Android malware detectors against adversarial examples. For instance, RAMDA [40] is a state-of-the-art approach that aims to improve detectors' robustness by squeezing the room of adversarial examples in the latent space. Nonetheless, this method sacrifices detection effectiveness since it also filters out benign apps that are close to adversarial examples. As such, designing robust and effective Android malware detectors remains an open challenge [22, 42]. In this study, we propose a novel graph-based approach, MaskDroid, detailed in Section 3, to further advance this field. Our approach not only bolsters robustness against adversarial examples but also ensures high detection effectiveness through a more precise interpretation of the program semantics of apps.

3 METHODOLOGY

In this section, we present a novel learning framework, MaskDroid, designed to improve the robustness of Android malware detection without compromising detection performance. Guided by a masking mechanism, MaskDroid can more effectively explore the structural and semantic information encoded in the graph representations, thereby enhancing the understanding of potential malicious behaviors. Additionally, we further incorporate a proxy-based contrastive learning module to boost MaskDroid's ability to discriminate between benign and malicious instances.

3.1 Overview

Figure 2 illustrates the high-level overview of MaskDroid. The model consists of two main components: (1) a self-supervised reconstruction module that utilizes graph neural networks (GNNs) along with a graph mask mechanism to learn semantics of the input graphs, and (2) a proxy-based contrastive learning module, which leverages the mutual information across samples in similar and dissimilar classes to enhance the model's discriminatory power.

In the reconstruction part, we first mask a proportion of the nodes in the input graph. A GNN encoder is then applied to map the features of each node into a latent space. Following this, a GNN decoder reconstructs the masked-out nodes based on the latent representations of the remaining nodes. This self-supervised task promotes *the learning of the underlying structures and dependencies within the input graphs, thereby deepening the model's understanding of app behaviors.*

With the graph-level representation obtained from the graph encoder, we proceed to the contrastive module. The principle behind this module is that *apps within the same class (i.e., benign or malicious) should be closer to each other, while apps from different classes*

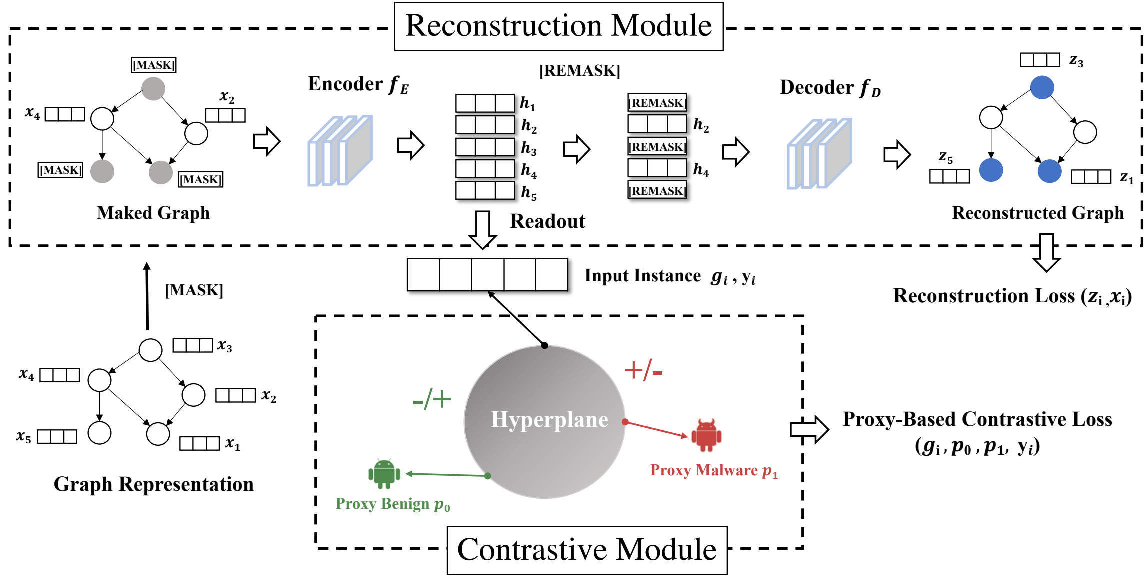


Figure 2: The framework of MaskDroid. The training phase comprises two modules. The upper dashed-line bracket represents the self-supervised reconstruction module, while the lower bracket represents the proxy-based contrastive learning module.

should be more distant. To achieve this, we initialize two proxy representations as the anchors of benign and malicious classes, respectively. During the training phase, we pull a sample closer to the anchor of its class and push it away from the anchor of a different class. These proxy representations are updated simultaneously to maintain their roles as class anchors.

To predict the category of an app, MaskDroid disables the mask mechanism and processes the input instance through the encoder to obtain a graph-level representation. Finally, it determines whether the app is benign or malicious based on which proxy the graph-level representation is closer to.

3.2 Reconstruction Module

We now present the details of the self-supervised reconstruction module in MaskDroid. For better understanding, we will first recap the graph representation used in MaskDroid before delving into the module itself.

Recap of Graph Representation. The graph representation utilizes the function call graph (FCG) as its input. Specifically, we begin by identifying a set of sensitive API calls (*e.g., getIpAddress()*) within an app's FCG to serve as seed nodes. Next, we extract subgraphs centered on these seed nodes at a fixed depth to form the input graph. It is worth noting that focusing on sensitive API calls and their surrounding contexts can effectively capture apps' malicious behaviors, as these behaviors are often carried out by a small part around sensitive API calls. This strategy has been validated in previous studies [26, 63, 64]. To further enhance the representation, each node is initialized with the opcode and required permissions associated with its corresponding API call. The opcode and permissions are pivotal in understanding the semantics of API calls, as they provide critical insights into app behaviors [31, 36]. For instance, the opcode *invoke* signifies that one function must call another to complete a task, while the permission *SEND_SMS* denotes the app's capability to send text messages.

Here, we begin with the initialization of node features and then introduce how we mask and reconstruct the graph to learn the underlying program semantics in a self-supervised manner.

3.2.1 Node Initialization. Given the graph representation, $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{X})$, each node $v \in \mathcal{V}$ is characterized by its attributes (*i.e.,* opcode and permissions) that describe the API call. To capture this information, we initialize the node feature $\mathbf{x}_v \in \mathbb{R}^d$ as the concatenation of the embeddings for the opcode and permissions:

$$\mathbf{x}_v = n_{v_{op}} \,||\, n_{v_{per}}, \tag{2}$$

where $n_{v_{op}}$ and $n_{v_{per}}$ represent the one-hot encodings of the opcode and permissions, respectively.

3.2.2 Graph Reconstruction. Our objective is to learn a high-quality representation that is resilient to adversarial attacks while ensuring optimal detection performance. The reconstruction module supports this goal by encouraging the model to recover masked nodes using unmasked nodes, effectively capturing the underlying structural and semantic information within the input graph. Consequently, even if the input graph is partially corrupted, MaskDroid retains its ability to discern malicious semantics, making it more robust to adversarial attacks.

Graph Masking and Encoder. We apply uniform random sampling to choose a subset of nodes $\widetilde{\mathcal{V}} \subset \mathcal{V}$ and mask their corresponding representations with a learnable vector $\mathbf{x}_{[M]}$. This strategy ensures that for each node, its neighbors are neither all masked nor all visible [33]. Based on this, it is easier to recover the masked nodes with their neighboring unmasked nodes, facilitating the training of the model to understand the graph structure. Formally, the

node feature $\widetilde{\mathbf{x}}_v$ of the masked graph $\widetilde{\mathcal{G}}$ can be defined as:

$$\widetilde{\mathbf{x}}_v = \begin{cases} \mathbf{x}_{[M]} & v \in \widetilde{\mathcal{V}}, \\ \mathbf{x}_v & v \notin \widetilde{\mathcal{V}}. \end{cases} \tag{3}$$

Considering the inherent graph nature of the masked graph, MaskDroid utilizes graph neural networks [28] (GNNs) to analyze the structural information and learn the corresponding node representations, *i.e.,* embeddings. GNNs are particularly suited for this as they recursively propagate and aggregate node features across edges, enabling the model to capture both local and global graph dependencies. Take the sequence (*getAndroidIPAddress()* → *getWifiIPAddress()* → *isWifiEnabled()* | *getIpAddress()*) depicted in Figure 1 as an example. If we examine the nodes in isolation, we only see that the app invokes several Wifi-related functions, *e.g., getWifiIPAddress()*. It is difficult to capture how the app executes the entire process of checking the network status and obtaining the IP address. However, by leveraging GNNs, MaskDroid can better explore the multi-hop relation between nodes — such as propagating information from *getWifiIPAddress()* to *isWifiEnabled()* and *getIpAddress()* — enhancing the understanding of how the app works. Formally, the representation of a node v at layer $l+1$ is updated by aggregating the embeddings of its neighbors as follows:

$$\widetilde{\mathbf{x}}_v^{(l+1)} = \sigma((\widetilde{\mathbf{x}}_v^{(l)} + \sum_{u \in \mathcal{N}_v} \frac{\widetilde{\mathbf{x}}_v^{(l)}}{\sqrt{|\mathcal{N}_u|\,|\mathcal{N}_v|}})\mathbf{W}_\alpha^{(l)}), \tag{4}$$

where $\mathcal{N}_v$ represents the set of neighbors of node v, $\mathbf{W}_\alpha^{(l)}$ is the weight matrix at layer l, and σ is the activation function, such as ReLU or LeakyReLU. After L layers of propagation, the final node embeddings are obtained as $\mathbf{h}_v = \widetilde{\mathbf{x}}_v^{(L)}$.

Graph Remasking and Decoder. With the latent representation $\mathbf{h}_v$ for each node in the masked graph, the next step is to recover the masked nodes and reconstruct the original graph. If MaskDroid can accurately recover the masked nodes, it indicates that the model can infer the missing information based on the surrounding nodes, enhancing its robustness to adversarial attacks. For example, in Figure 1, *getWifiIPAddress()* is masked out, and the model is trained to recover it based on its surrounding nodes, such as *getAndroidIPAddress()*, *isWifiEnabled() and getIpAddress()*. This implies that even if an adversary partially alters the graph, MaskDroid can still deduce that the app is attempting to obtain the Wifi status and IP address. With this stable understanding, MaskDroid can still make accurate predictions.

To boost the model's ability to recover masked nodes based solely on their surrounding nodes, we re-mask the masked nodes in $\widetilde{\mathcal{V}}$ to prevent the model from memorizing them [33]. Specifically, the re-masked representation $\widetilde{\mathbf{h}}_v$ is defined as follows:

$$\widetilde{\mathbf{h}}_v = \begin{cases} \mathbf{h}_{[M]} & v \in \widetilde{\mathcal{V}}, \\ \mathbf{h}_v & v \notin \widetilde{\mathcal{V}}. \end{cases} \tag{5}$$

where $\mathbf{h}_{[M]}$ is a learnable vector used to re-mask selected nodes. In our implementation, we simply set the re-masked vector $\mathbf{h}_{[M]}$ to zero due to its effectiveness, leaving more sophisticated strategies for future work. The re-masked representation is then fed into the decoder f_D to reconstruct the original node features. Similar to the encoder, the decoder also utilizes the GNN architecture, which is better to capture the structural information and reconstruct the masked nodes.

$$\mathbf{z}_v = f_D(\widetilde{\mathbf{h}}_v) = \text{GNN}(\widetilde{\mathbf{h}}_v). \tag{6}$$

Here, for clarity, we omit the details of how the decoder propagates and aggregates information along the graph, as this process can be designed in a manner similar to the encoder.

To train the model, we define the reconstruction loss $\mathcal{L}_{\text{rec}}$ to measure the discrepancy between the original node features and the reconstructed features. Particularly, we employ the cosine similarity to measure their distance as follows:

$$\mathcal{L}_{\text{rec}} = \frac{1}{|\widetilde{\mathcal{V}}|} \sum_{v \in \widetilde{\mathcal{V}}} (1 - \frac{\mathbf{x}_v^T \mathbf{z}_v}{\|\mathbf{x}_v\| \cdot \|\mathbf{z}_v\|})^2. \tag{7}$$

In summary, through the self-supervised graph reconstruction learning task, MaskDroid gains a holistic understanding of the graph structures and semantics, which is crucial for the success of the subsequent discrimination task.

3.3 Contrastive Module

With the stable representation that captures app semantics, we now turn to the contrastive module, which aims to enhance the model's discriminatory power. The principle behind this module is intuitive: apps executing similar behaviors should cluster together and mutually reinforce each other, while apps from different classes should be more distant from each other.

Towards this end, we adopt a proxy-based contrastive learning strategy [67] to explore the mutual information across samples in similar and dissimilar classes. Since Android malware detection is a binary classification task, we define two proxies, $\mathbf{p}_0$ and $\mathbf{p}_1$ for benign and malicious classes, respectively. The proxy representations are learnable vectors with random initialization, utilized to capture the distinctions between benign and malicious apps, facilitating their separation in the latent space. During training, each instance is pulled closer to the proxy of its own class while being pushed further from the other proxy. The two proxy vectors are updated simultaneously with the model parameters throughout the training process to maintain their roles as class anchors. Assume the graph-level representation obtained from the encoder f_E is $\mathbf{g}_i$ for the i-th instance. Each instance has a supervised label y_i indicating whether it belongs to the benign or malicious class, where 0 represents benign and 1 represents malicious. The contrastive learning process can be defined as follows:

$$\begin{aligned} \mathcal{L}_{\text{cl}} = y_i \cdot & \left[(\frac{\mathbf{g}_i \cdot \mathbf{p}_0}{\|\mathbf{g}_i\|\|\mathbf{p}_0\|})^2 + (1 - \frac{\mathbf{g}_i \cdot \mathbf{p}_1}{\|\mathbf{g}_i\|\|\mathbf{p}_1\|})^2 \right] \\ & + (1 - y_i) \cdot \left[(\frac{\mathbf{g}_i \cdot \mathbf{p}_1}{\|\mathbf{g}_i\|\|\mathbf{p}_1\|})^2 + (1 - \frac{\mathbf{g}_i \cdot \mathbf{p}_0}{\|\mathbf{g}_i\|\|\mathbf{p}_0\|})^2 \right]. \end{aligned} \tag{8}$$

After the training phase, the two proxies $\mathbf{p}_0$ and $\mathbf{p}_1$ aggregate the information of all instances in or not in their respective classes, serving as the class anchors for the inference phase.

3.4 Android Malware Detection

To optimize the model for Android malware detection, we combine the reconstruction and contrastive modules into a joint training

framework. The final objective of MaskDroid is defined as:

$$\mathcal{L} = \lambda_1 \cdot \mathcal{L}_{\text{rec}} + \lambda_2 \cdot \mathcal{L}_{\text{cl}}, \tag{9}$$

where λ_1 and λ_2 are the hyper-parameters to control the strength of two modules. By minimizing this objective, MaskDroid learns a high-quality representation that captures app semantics and maintains two anchors that consider all instances in the training set, enhancing the model's robustness and discriminatory power.

In the detection phase, MaskDroid first transforms the input graph into a graph-level representation using the encoder f_E. Then it calculates the distance between the graph-level representation and the two anchors to determine whether the input instance is benign or malicious.

4 EVALUATION

In this section, we evaluate the performance of MaskDroid by answering the following research questions (RQs):

- **RQ1:** Does MaskDroid successfully improve the robustness against different adversarial attacks (*e.g.,* white-box and black-box attacks) compared to its baselines?
- **RQ2:** Does MaskDroid sacrifice detection effectiveness to enhance its robustness against adversarial attacks?
- **RQ3:** To what extent do different design choices affect the performance of MaskDroid on counteracting adversarial attacks and detecting malware?
- **RQ4:** Does MaskDroid require more computational resources to complete its detection?

4.1 Experimental Setup

4.1.1 Implementation. We utilize Androguard [3] to decompile APKs and extract the Function Call Graph (FCG) for each app. Based on the extracted FCGs, we construct the input graph for our model in a three-step procedure: (a) we traverse the FCG to collect sensitive API calls. It is worth noting that we focus on the sensitive API calls reported by PSCount [12] and Axplorer [13], as they include the most commonly used sensitive API calls in Android malware [31, 63, 64]; (b) we take the collected sensitive API calls as roots and perform a breadth-first search to collect the call sequences within a certain depth (*i.e.,* 2), then merge them into the input graph; (c) we initialize the nodes in the input graph with their corresponding opcodes and permissions.

To find optimal hyper-parameters for MaskDroid, we employ a grid search strategy. The learning rate is tuned from {0.1,0.01,0.001, 0.0001}, and the mask rate is searched within {0.1,0.2, 0.3, 0.4, 0.5, 0.6,0.7, 0.8,0.9}. For experiments in Sections 4.2 and 4.3, we use the entire dataset from 2016 to 2020, finding that mask rate = 0.8 yields the best performance. As such, we choose 0.8 as the masking rate. For the ablation study in Sections 4.4 and 4.5, we use data from 2020 and find 0.5 is the optimal masking rate. Furthermore, the number of GNN layers in both the encoder f_E and decoder f_D is tuned among {1,2,3}. Based on achieving the best performance, we present results under the configuration of a mask rate of 0.8, 0.001 learning rate, and two 2-layer GNNs for encoder and decoder. In addition, $\lambda 1$ and $\lambda 2$ are set to be equal in our experimental setting.

Table 1: Evaluation Dataset Statistics. The samples cover three years from 2016 to 2020 and maintain a 9:1 ratio of benign to malicious apps.

Year	Begin	Malware	M+B	M/(M+B)
2016	21,292	2,390	23,682	10.1%
2017	21,006	2,389	23,395	10.2%
2018	20,099	2,326	22,425	10.4%
2019	20,260	2,345	22,605	10.4%
2020	19,802	2,301	22,103	10.4%
Total	102,459	11,751	114,210	10.3%

All experiments are performed on a server with an Intel Xeon Gold 6248 CPU @ 2.50GHz, 188GB physical memory, and an NVIDIA Tesla V100 GPU. The OS is Ubuntu 20.04.2 LTS.

4.1.2 Datasets. To rigorously assess MaskDroid's performance, we source our dataset from AndroZoo [9], a continuously expanding repository of Android apps that aggregates apps from several sources, such as Google Play, Appchina, and Anzhi. Our dataset consists of 114,210 apps, among which 102,459 are benign and 11,751 are malicious, spanning from 2016 to 2020, as depicted in Table 1.

Importantly, the dataset adheres to the guidance proposed by Tesseract [52]. *Avoid Grayware*: The ambiguous nature of grayware can potentially skew the performance of learning models. To counteract this threat, we utilize the positive anti-virus alerts from VirusTotal [59], represented by p, to filter out grayware. In particular, apps with $p \geq 4$ are labeled malicious, whereas those with $p = 0$ are classified as benign. *Goodware-to-Malware Ratio*: Previous studies have verified that the ratio of benign to malicious apps in the wild is notably imbalanced, with malware constituting a small fraction (10%) [16, 52, 70]. To more accurately gauge the effectiveness of our approach in real-world scenarios, we ensure the malware proportion in our dataset is set at 10%. Additionally, we sample the dataset from 2016 to 2020 to cover a wide range of apps and reflect the temporal dynamics of malware evolution. In our experiments, we randomly split each dataset into three disjoint sets: training, validation, and testing, with proportions of 70%, 20%, and 10%, respectively. We also ensure all the disjoint sets exhibit a 9:1 ratio of benign to malicious apps.

4.1.3 Baselines. To comprehensively investigate the performance of MaskDroid, we compare it with three state-of-the-art graph-based detection approaches: MamaDroid, MalScan, and MsDroid, as well as two non-graph-based detectors: Drebin and RAMDA.

- **MamaDroid** [46]: This method abstracts Function Call Graphs (FCGs) according to the package or family level of function names. It then constructs Markov Chains and calculates the transition probabilities between different nodes, which serve as the feature vector to train a Random Forest classifier.
- **MalScan** [63]: This algorithm treats FCGs as social networks and conducts centrality analysis on sensitive API calls to capture APKs' semantics for classification.
- **MsDroid** [31]: This detector is built on the key insight that malicious operations often in code snippets around sensitive API calls. It thus models app-sensitive behaviors as subgraphs around sensitive API calls and feeds them into a GNN classifier.

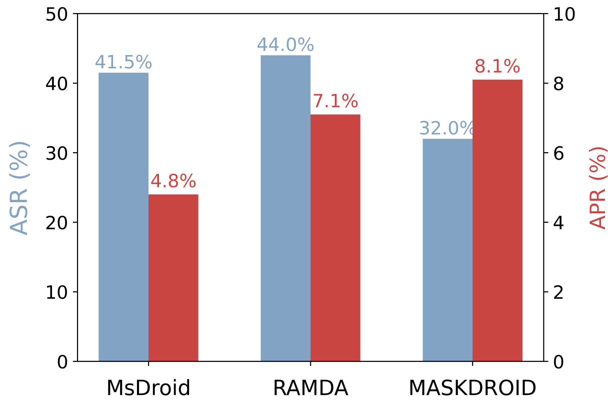


Figure 3: Robustness Evaluation against White-Box Adversarial Attacks (mask rate γ= 0.8).

- **Drebin** [10]: This approach first extracts features like permissions, intents, code strings, and API calls from APKs using static analysis, and then trains an SVM classifier to detect malware.
- **RAMDA** [40]: This method is a state-of-the-art approach aimed at improving detectors' robustness against adversarial attacks. Specifically, it introduces a Variational Autoencoder (VAE) to learn a compact representation from its input features (*i.e.,* intents, permissions, and API calls) and then trains a binary classifier for malware detection.

4.2 Robustness Enhancement (RQ1)

Settings. In this RQ, we investigate whether MaskDroid can effectively enhance its robustness against adversarial attacks compared to state-of-the-art detectors. Similar to previous work [38, 42], we measure the resilience of these detectors against adversarial attacks using two metrics: (a) Attack Success Rate (ASR), which indicates the percentage of adversarial examples that successfully evade detection, and (b) Average Perturbation Ratio (APR), which quantifies the average percentage of perturbed edges in the graph representation. Note that we only report APR on MaskDroid and its graph-based competitors, as it is not comparable with approaches that use different feature representations. For a fair comparison, we conduct this experiment on the whole dataset from 2016 to 2020 and set the maximum number of iterations to 100 for all models.

To comprehensively explore the robustness of MaskDroid, we implement a representative attack algorithm, Integrated Gradient Guided JSMA (IG-JSMA) attack [62], under two distinct scenarios: white-box and black-box. JSMA has been widely employed to craft adversarial examples (AEs) for evading Android malware detectors [15, 42, 63]. Specifically, it perturbs the most influential features based on indicative forward derivatives to create AEs. Additionally, we adopt previous strategies [15, 38] to ensure the generated adversarial examples can be repacked into APKs. For feature-vector-based approaches like Drebin, we follow [15] by constraining modifications to focus on vector bits that are 0, changing 0s to 1s. This ensures that all required permissions or functions remain unchanged, preserving functionality. For graph-based approaches like MaskDroid, we introduce edges that do not affect APK functionality, in line with the technique described in [38].

Table 2: Robustness Evaluation against Black-Box Adversarial Attacks (mask rate γ= 0.8).

Detectors	Malscan	MamaDroid	Drebin
ASR	98.5%	69.0%	100%
APR	-	-	-
Detectors	**MsDroid**	**RAMDA**	**MaskDroid**
ASR	13.2%	19.2%	19.1%
APR	0.5%	1.5%	10.1%

4.2.1 White-Box Attack Defense. White-box adversarial attacks occur when attackers possess full knowledge of the victim model, including training data, model structure, parameters, gradient information, and prediction results [24]. In this scenario, we exclude Mamadroid [46], Malscan [63], and Drebin [10], as they utilize traditional machine learning methods rather than deep neural networks, preventing us from calculating their gradient information.

Figure 3 illustrates the results of MsDroid [31], RAMDA [40], and MaskDroid against white-box adversarial attacks. From this figure, we observe that MaskDroid consistently outperforms its competitors in terms of both ASR (32.0%) and APR (8.1%). Specifically, the lower ASR indicates that MaskDroid is more resilient against adversarial attacks than MsDroid and RAMDA. Additionally, the higher APR suggests that MaskDroid requires perturbing more edges to evade detection, meaning adversaries need to exert more effort to craft adversarial examples. This can be attributed to the stable representations learned by MaskDroid by reconstructing the whole graph from a small portion of nodes, which enhances its robustness against adversarial attacks.

4.2.2 Black-Box Attack Defense. In real-world scenarios, attackers do not always have access to the detailed structures and parameters of the malware detection systems and are often limited to certain knowledge, such as the training dataset and classification results [34, 38, 39]. In this context, the typical attack strategy is to distill a substitution model that mimics the behavior of the original black-box model, and then calculate gradients based on the substitution model to simulate a white-box attack. We follow the same strategy to conduct a black-box attack on the substitution model. Specifically, we use an MLP to mimic the learning models of MamaDroid [46], MalScan [63], Drebin [10] and RAMDA [40]. For the model of MsDroid [31], we use a two-layer GNN encoder followed by an MLP classifier to mimic its behavior. To train the substitution models, we use the labels predicted by the target black-box models. For example, we first collect the predictions of MamaDroid on the training set, then train an MLP to mimic MamaDroid's behavior using the collected predictions as labels.

Table 2 presents the results of MaskDroid and its baselines against black-box adversarial attacks. Note that we exclude the APR for MamaDroid, MalScan, and Drebin, as they do not use graph representations. From the table, we find that Drebin [10] is the most vulnerable to black-box adversarial attacks, with an ASR of 100%. This is because Drebin is based on binary features, which are easy to manipulate. We know that Malscan [63] and MamaDroid [46] extract features from the FCGs, while MamaDroid is more robust than Malscan, with an ASR of 69.0%. This increased

Table 3: The detection effectiveness of MaskDroid and its baselines on the dataset from 2016 to 2020.

Detectors	Precision	Recall	F1	Accuracy
Malscan	0.811	0.805	0.808	0.962
MamaDroid	0.922	0.768	0.838	0.970
Drebin	0.763	0.712	0.736	0.949
MsDroid	0.582	0.615	0.598	0.917
RAMDA	0.821	0.800	0.811	0.962
MaskDroid	0.709	0.876	0.783	0.951

Table 4: The detection effectiveness of MaskDroid and its baselines on the temporal biased dataset.

Detectors	Precision	Recall	F1	Accuracy
Malscan	0.650	0.372	0.473	0.916
MamaDroid	0.934	0.272	0.421	0.924
Drebin	0.743	0.466	0.573	0.929
MsDroid	0.642	0.211	0.317	0.908
RAMDA	0.531	0.562	0.546	0.905
MaskDroid	0.541	0.630	0.582	0.908

robustness is because MamaDroid abstracts the FCGs according to the package or family level of function names, making it more difficult to perturb, aligning with previous findings [22]. MsDroid [31] and RAMDA [40] achieve ASRs of 13.2% and 19.2%, respectively, which are lower or comparable to MaskDroid. While MaskDroid achieves the highest APR (10.1%), indicating it is very difficult to attack in real-world scenarios. Specifically, a higher APR indicates that more iterations are needed and more edges must be modified to craft adversarial examples (AEs). Given the large app graph size and the constraint that AEs can be repacked into APKs, finding a usable edge is not trivial, making the attack ineffective.

> *Result 1:* Compared with state-of-the-art Android malware detection approaches, *MaskDroid* enhances robustness against adversarial attacks in both white-box and black-box scenarios. Notably, in the white-box attack, *MaskDroid* reduces the attack success rate by 9.5% against the second-best baseline.

4.3 Effectiveness Comparison (RQ2)

Settings. Having verified MaskDroid's robustness against adversarial attacks, we now investigate whether this robustness comes at the expense of detection effectiveness. To measure the detection effectiveness of MaskDroid and its baselines, we evaluate their performance on the testing set using standard metrics, including precision, recall, F1-score, and accuracy, following the standard practice in Android malware detection [31, 63]. Specifically, precision and recall measures correctly detect malware against all detected malware and all actual malware, respectively. The F1-score, calculated as $2 \times \frac{precision \times recall}{precision + recall}$, represents the balance between precision and recall. Accuracy is the ratio of correctly classified apps to the total number of analyzed apps. This experiment is conducted on the dataset from 2016 to 2020. Additionally, temporal bias is widely recognized as a key factor influencing the effectiveness of malware detectors [42]. We also explore its impact on MaskDroid. Specifically, we train the model using data collected from 2016 to 2019 and evaluate it on data from 2020.

Table 3 demonstrates the detection effectiveness of MaskDroid and its baselines. From the results, we observe that MaskDroid achieves a precision of 0.709, recall of 0.876, F1-Score of 0.783, and accuracy of 0.951. While MaskDroid's F1-Score is lower than some baselines, such as MalScan [63] and MamaDroid [46], it is important to note that MaskDroid's F1-Score is still competitive. We can see that MaskDroid achieves the highest recall among all baselines, indicating its superior ability to detect Android malware. This phenomenon is in line with the design of MaskDroid, which aims to enhance the model's capability to counteract adversarial attacks. By training the model to learn the underlying semantics and identify more malicious signals in the graph, MaskDroid is more likely to detect a greater number of malware samples and adversarial examples. However, this also leads to the misclassification of some benign apps as malware, resulting in lower precision compared to several baselines. Given the trade-off between detection robustness and effectiveness, we believe that MaskDroid's performance is satisfactory and competitive with existing Android malware detectors.

Table 4 presents the detection effectiveness of MaskDroid and its baseline models in the presence of temporal bias. We observe that MaskDroid achieves superior results in terms of F1-score, demonstrating its ability to detect malware even when faced with temporal bias. This can be attributed to MaskDroid 's focus on learning high-quality representations that encode malicious patterns, which remain stable and robust over time.

> *Result 2:* MaskDroid achieves detection effectiveness comparable to existing Android malware detectors in both same-time distribution scenarios and situations involving temporal bias.

4.4 Ablation Study on MaskDroid (RQ3)

In this section, we investigate the impact of different design choices on the performance of MaskDroid. Specifically, we conduct ablation experiments on various components (*i.e.,* reconstruction module, contrastive module, and the masking mechanism) to explore how they contribute to MaskDroid's robustness and effectiveness in Android malware detection.

4.4.1 Effect of Reconstruction/Contrastive Modules. To clarify our description, we first introduce the terminology used in this section. *MaskDroid-cr* refers to a version of MaskDroid where both the contrastive and reconstruction modules have been removed, as illustrated in Figure 4. *MaskDroid-c* means replacing the contrastive module in Figure 2 with an MLP as the binary classifier. *MaskDroid-r* denotes disabling the reconstruction mechanism in Figure 2, and feeding the readout from encoder f_E directly into the contrastive module.

Table 5 presents the detection effectiveness of MaskDroid and its ablated versions. As shown, MaskDroid achieves the highest

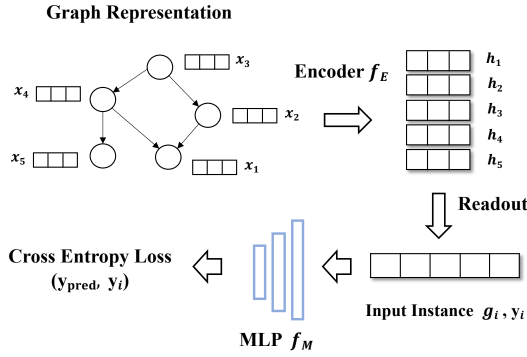


Figure 4: Framework of the model MaskDroid-cr that disables both the contrastive module and the reconstruction module fromMaskDroid. The input graph goes through a two-layer GNN encoder, proceeds with a readout layer, and is then fed into an MLP classifier.

performance with an F1-Score of 82.4%. When the contrastive module (MaskDroid-c) or the reconstruction module (MaskDroid-r) is removed, the F1-Score drops to 77.4% and 79.9%, respectively. This indicates that both the reconstruction and contrastive modules are essential for MaskDroid to achieve optimal performance in malware detection. Interestingly, the model without either module (MaskDroid-cr) still achieves a relatively high F1-Score, surpassing MaskDroid-c and MaskDroid-r, which suggests that the two modules complement each other to enhance the model's performance. We can also observe a clear trend in precision and recall, that is, MaskDroid has lower precision and higher recall compared to when two modules are removed. This is because MaskDroid uses reconstruction and contrastive modules to enhance its ability to capture malicious signals, significantly improving recall (the probability of detecting more malware). Although this introduces several false positives, the overall F1-score continues to improve.

We further evaluate the robustness of MaskDroid and its ablated versions against adversarial attacks. Given that white-box adversarial attacks are more effective than black-box attacks, our focus in this ablation study is on the former. The left part in Figure 5 demonstrates the impact of the reconstruction and contrastive modules on the model's robustness against the white-box adversarial attack. The results show that MaskDroid significantly lowers the Attack Success Rate (ASR) compared to MaskDroid-c, MaskDroid-r, and MaskDroid-cr, indicating its superior robustness against adversarial attacks. Comparing MaskDroid-c and MaskDroid-r with MaskDroid-cr, we note that both MaskDroid-c and MaskDroid-r exhibit a lower ASR, suggesting that the contrastive and reconstruction modules are crucial for enhancing the model's robustness.

4.4.2 Effect of mask rate γ. The masking mechanism is one of our key designs to encourage MaskDroid to learn a more holistic representation of the input graph. Selecting an appropriate mask rate γ is crucial for our model's performance. In this section, we vary the mask rate (*i.e.,* 0.2, 0.5, 0.8, 0.9) to examine its influence on the detection effectiveness and robustness of MaskDroid.

Table 5: Ablation study on reconstruction and contrastive modules for Android malware detection performance.

Models	Precision	Recall	F1	Accuracy
MaskDroid-cr	0.918	0.730	0.813	0.965
MaskDroid-c	0.886	0.688	0.774	0.958
MaskDroid-r	0.896	0.720	0.799	0.962
MaskDroid	0.772	0.883	0.824	0.961

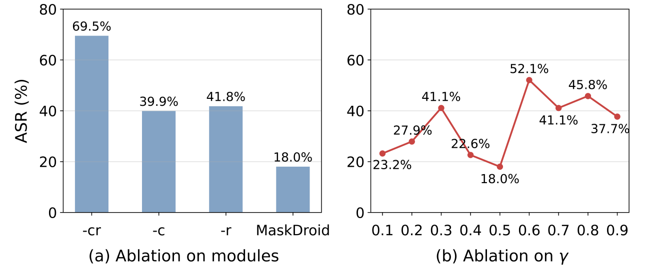


Figure 5: Ablation study on reconstruction/contrastive modules and mask rate γ for MaskDroid's robustness against white-box adversarial attacks. The left subfigure presents the results of the reconstruction/contrastive modules, while the right subfigure illustrates the impact of the mask rate γ.

Table 6 presents the changes in detection performance as the mask rate γ varies. We observe that MaskDroid's detection effectiveness is not overly sensitive to the mask rate. Specifically, when the mask rate increases from 0.2 to 0.9, the F1-Score falls slightly from 0.789 to 0.824. One potential explanation is that MaskDroid's contrastive module can effectively learn the differences between the benign and malicious samples, even when the input graph is partially masked. Delving deeper into the robustness of MaskDroid against adversarial attacks, we find that the model's performance is more sensitive to the mask rate. As shown in the right part of Figure 5, the ASR is lowest (0.180) when the mask rate is 0.5, and increases to 0.458 when the mask rate is 0.8. This may be because a higher mask rate introduces more noise into the input graph, making it more challenging for the model to learn the underlying patterns. Therefore, selecting an appropriate mask rate is crucial for MaskDroid to achieve optimal performance in both malware detection and defense against adversarial attacks. We leave how to automatically determine the mask rate as future work.

4.4.3 Visualization of Representations. The main contribution of MaskDroid is to learn high-quality representations that grasp the holistic understanding of app behaviors and derive a clear decision boundary to achieve impressive discriminative power for both malware and adversarial examples. To further understand the internal representations learned by MaskDroid, we visualize the latent representations obtained by MaskDroid and its variant MaskDroid-cr, which disables the reconstruction and contrastive modules. We adopt the t-SNE technique to project the graph representation for each input sample onto a 2D space [57].

Figure 6 illustrates the representations learned by MaskDroid and MaskDroid-cr. In the figure, the green points represent the representations of benign samples, while the red points represent

Table 6: Ablation study on mask rate γ for Android malware detection performance.

Mask Rate	Precision	Recall	F1	Accuracy
$\gamma = 0.1$	0.685	0.888	0.773	0.947
$\gamma = 0.2$	0.720	0.874	0.789	0.952
$\gamma = 0.3$	0.744	0.851	0.794	0.955
$\gamma = 0.4$	0.751	0.884	0.812	0.958
$\gamma = 0.5$	0.772	0.884	0.824	0.961
$\gamma = 0.6$	0.756	0.865	0.807	0.958
$\gamma = 0.7$	0.757	0.884	0.816	0.959
$\gamma = 0.8$	0.728	0.897	0.804	0.955
$\gamma = 0.9$	0.724	0.869	0.790	0.952

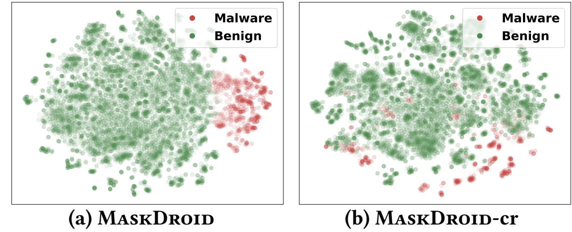
those of malicious samples. We can see that the latent representations learned by MaskDroid are more compact than those learned by MaskDroid-cr. Also note that MaskDroid achieves a much clearer decision boundary, where the benign and malicious samples are more distinctly separated. The compact representations and clear boundary not only enable the model to excel in classification tasks but also significantly increase the difficulty of adversarial attacks, thus resulting in its enhanced robustness [19]. This can be attributed to the ability of the reconstruction and contrastive modules to learn the underlying patterns in the input graph and differentiate different types of samples.

> *Result 3:* All of our choices in designing MaskDroid, including the reconstruction and contrastive modules, as well as the mask rate γ, are crucial for enhancing the model's effectiveness and robustness in malware detection.

4.5 Efficiency Evaluation (RQ4)

In addition to the robustness and effectiveness of MaskDroid, efficiency is another critical factor influencing the model's practicality. In this section, we compare the training costs of MaskDroid with its baseline and variant models to evaluate its efficiency. For a fair comparison, we exclude MamaDroid [46], Malscan [63], and Drebin [10], as they utilize traditional machine learning models to extract features and train classifiers, which are not comparable to deep learning models in terms of training cost. It is important to highlight that when training these detectors, we apply the early stopping strategy to prevent over-fitting. Additionally, the training process is conducted on the same server with the same configuration as mentioned in Section 4.1.1.

Table 7 summarizes the training costs of MaskDroid and its baseline or variant models. We observe that MaskDroid requires 150 epochs to converge, lasting 2,100 seconds in total. This is comparable to MsDroid and significantly less than RAMDA's strategy, indicating that MaskDroid does not sacrifice efficiency for robustness. By further comparing MaskDroid with its variant models (*i.e.,* , -cr, -c, -r), we observe that MaskDroid-cr takes the least time to converge, which is expected as it does not involve the reconstruction and contrastive modules. We also note that MaskDroid requires a comparable amount of time to MaskDroid-r and MaskDroid-c, indicating that the two modules are not simply additive but work



(a) MaskDroid (b) MaskDroid-cr

Figure 6: A more compressed representation learned by MaskDroid compared to MaskDroid-cr, which disables the reconstruction module and the contrastive module.

Table 7: Comparison of training costs between MaskDroid and its baseline/variant models (on 2020 data).

	MsDroid	RAMDA	MaskDroid -cr	MaskDroid -c	MaskDroid -r	MaskDroid
Epoch(s)	186	240	60	128	115	150
Total(s)	1,860	12,770	600	2,560	8,050	2,100

in meaningful cooperation to reinforce each other and achieve the goal of robustness.

> *Result 4:* MaskDroid achieves a balance between detection robustness and efficiency, demonstrating superior resilience against adversarial attacks while maintaining a moderate training cost compared to its baselines.

5 THREATS TO VALIDITY

In this section, we discuss the threats to the validity of our study. First, the effectiveness and robustness of MaskDroid may be influenced by different hyper-parameters used in the neural networks. To mitigate this threat, we adopt advanced practices from prior studies [43, 68], employing a grid search to tune and find the optimal hyper-parameters that yield the best performance on validation sets. We detail all hyper-parameter settings in Section 4.1 for reproducibility and release our evaluation artifacts, including codes and datasets. Second, when conducting comparison experiments, we utilize open-source implementations of baseline methods. However, since we have tuned the hyper-parameters of these baselines to suit our datasets and experimental settings, the performance of them may differ from the results originally reported in the literature. Third, we do not compare MaskDroid's defensive capabilities with other adversarial defense methods, such as adversarial training [7]. This is because these methods are orthogonal to our work and can be integrated with MaskDroid to enhance further its robustness against adversarial attacks, which we leave as future work. At last, following previous studies [15, 63], we use the representative attack algorithm JSMA to evaluate MaskDroid 's robustness. However, MaskDroid 's resilience against new and evolving attack methods remains unexplored, which we leave for future work.

6 RELATED WORK

In this section, we begin by reviewing related work on machine learning (ML)-based Android malware detection. Subsequently, we describe the masking mechanism. Finally, we introduce common

adversarial example generation methods and their corresponding defense strategies.

ML-based Android Malware Detection. Machine learning (ML) techniques have been extensively employed to analyze various types of APK features and extract malicious patterns for Android malware detection. Syntactical features, such as permissions, API calls, code strings, and intents, are commonly used to model app behaviors [6, 10, 40, 53, 61]. For example, Xmal [61] uses an attention-based MLP to distill malicious signals from the API calls and permissions. RAMDA [40] feeds intents and API calls into an Autoencoder to learn a resilient representation of the APKs for malware detection. Drebin [10] first utilizes static analysis to extract features such as permissions, intents, code strings, and API calls and then trains an SVM classifier to identify Android malware. However, such methods may fail to capture the program semantics of apps, limiting the detection effectiveness. To mitigate this issue, researchers turn to extracting different app semantics for effective Android malware detection [8, 20, 21, 31, 32, 35, 46, 47, 63, 64, 66]. DeepRefiner [66] and Mclaughlin et al. [47] represent app bytecodes as texts and images, respectively, applying LSTM and CNN to detect malware. Malscan [63] and HomDroid [64] treat the function call graphs of apps as social networks and apply the corresponding centrality analysis algorithm to capture malicious patterns for Android malware detection. Furthermore, MsDroid [31] describes app-sensitive behaviors with code snippets around sensitive API calls and employs graph neural networks to distill the semantic and structural information to identify malware.

Masking Mechanisms. In the field of graph representation learning, masking and reconstructing graph features has proven to be an effective approach for achieving robust learning [33]. Several works have applied masking mechanisms to improve model robustness in areas including image classification and text classification [49, 65]. For example, PatchGuard [65] proposed a masking defense to obscure corrupted features and recover the correct prediction for image classification tasks. MASKER [49] regularizes language models to reconstruct keywords from the remaining words and make low-confidence predictions when there is insufficient context. Inspired by the success of masking mechanisms in improving model robustness, we pioneer and adapt this practice to enhance the robustness of Android malware detection.

Adversarial Example Attack. With ML-based Android malware detection evolving, attackers increasingly seek to evade these detectors by purposefully perturbing the malicious APK samples. These attacks can be categorized into feature-space attacks [7, 25, 34, 39, 55] and problem-space attack [15, 37, 38, 54, 71]. Feature-based attacks target the feature vectors extracted from APKs directly, intending to deceive detectors. For instance, Hu et al. [34] leverage the capability of generative adversarial networks (GANs) [23] to modify binary feature vectors, aiming to mislead the detectors. Grosse et al. [25] propose a Jacobian matrix-based method to manipulate the features, allowing malicious samples to escape detection. In contrast, problem-based attacks endeavor to produce real adversarial malware. Specifically, HRAT [71] utilizes reinforcement learning to strategically modify the structure of a malicious app without affecting its original functionality. Li et al. [38] combine GAN with a multi-population co-evolution algorithm to add *try-catch* edges for crafting adversarial examples. Pierazzi et al. [54] explore the creation of adversarial malware by utilizing bytecode slices extracted from benign APKs. Furthermore, Android HIV [15] establishes a correspondence between the feature space and problem space, ensuring that any feature perturbations do not compromise the core functionality of the malware.

Adversarial Example Defense. To combat adversarial attacks, one can approach the problem from two distinct angles: by fortifying the data [7, 14, 24, 48, 52] or by enhancing the model [18, 40, 51, 56]. For example, Huang et al. [7] devise a method to create a diverse set of functionally preserved adversarial examples. By incorporating these diverse samples into the training process, they aim to bolster the model's resilience against adversarial attacks. Goodfellow et al. [24] suggest mitigating adversarial example attacks through model retraining. They incorporate the original dataset with newly labeled adversarial malware examples, enhancing the classifier's familiarity with various malware. Taking a different route, Bhagoji et al. [14] employ Principal Component Analysis (PCA) to project all inputs into a lower-dimensional space, thus diminishing the model's susceptibility to adversarial attacks. Turning to strategies that directly fortify the model itself, RAMDA [40] makes use of an autoencoder to derive a compressed representation of APKs, thereby filtering out potential adversarial examples. Similarly, Papernot et al. [51] adopt defense distillation to mitigate the vulnerability of deep neural networks against minor perturbations.

7 CONCLUSION

In this work, we propose MaskDroid, a novel framework designed to enhance robustness against adversarial attacks while maintaining impressive discriminative power for Android malware detection. Specifically, we introduce a masking mechanism and force MaskDroid to reconstruct the entire graph using the unmasked part, enabling it to learn stable representations of the input graphs more effectively. Additionally, MaskDroid incorporates a contrastive module to cluster samples of the same class together while pushing samples of different classes apart, thereby enhancing the model's discriminative power. Grounded by extensive evaluations, MaskDroid steadily outperforms state-of-the-art (SOTA) baselines on adversarial attack defense tasks and achieves comparable performance on malware detection tasks. A promising direction for future work would be to extend the exploitation of the mask mechanism to attention masks with semantic information, which could further enhance the model's understanding of malicious behaviors. Furthermore, since our strategy is applied to graph-based data and is not restricted to specific graph structures, similar methods can be extended to other graph-based datasets, such as control flow graphs, to boost the model's robustness.

ACKNOWLEDGMENTS

This research/project is supported by the National Research Foundation, Singapore under its Industry Alignment Fund – Pre-positioning (IAF-PP) Funding Initiative and the NExT project. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not reflect the views of National Research Foundation, Singapore.

REFERENCES

[1] 2017. *Millions of Android Devices Served with Judy Malware*. https://www.theregister.com/2017/05/30/android_app_judy_malware

[2] 2020. Dissecting DEFENSOR: a stealthy Android banking malware. https://medium.com/axdb/%EF%B8%8F-dissecting-defensor-a-stealthy-android-banking-malware-6610b0468256.

[3] 2023. Androguard. https://github.com/androguard/.

[4] 2024. An Android app on VirusTotal. https://www.virustotal.com/gui/file/b08ee107d349652bb39fb79d503b8e0cfd7b02e1ab891d7c7c25c6357829e5d8/details.

[5] 2024. *Expert Warns of Growing Android Malware Activity*. https://www.infosecurity-magazine.com/news/expert-warns-growing-android/

[6] Yousra Aafer, Wenliang Du, and Heng Yin. 2013. Droidapiminer: Mining api-level features for robust malware detection in android. In *Security and Privacy in Communication Networks: 9th International ICST Conference, SecureComm 2013, Sydney, NSW, Australia, September 25-28, 2013, Revised Selected Papers 9*. Springer, 86–103.

[7] Abdullah Al-Dujaili, Alex Huang, Erik Hemberg, and Una-May O'Reilly. 2018. Adversarial deep learning for robust detection of binary encoded malware. In *2018 IEEE Security and Privacy Workshops (SPW)*. IEEE, 76–82.

[8] Joey Allen, Matthew Landen, Sanya Chaba, Yang Ji, Simon Pak Ho Chung, and Wenke Lee. 2018. Improving accuracy of android malware detection with lightweight contextual awareness. In *Proceedings of the 34th Annual Computer Security Applications Conference*. 210–221.

[9] Kevin Allix, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. 2016. Androzoo: Collecting millions of android apps for the research community. In *MSR*.

[10] 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.. In *Ndss*, Vol. 14. 23–26.

[11] Anish Athalye, Nicholas Carlini, and David Wagner. 2018. Obfuscated gradients give a false sense of security: Circumventing defenses to adversarial examples. In *International conference on machine learning*. PMLR, 274–283.

[12] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. 2012. Pscout: analyzing the android permission specification. In *CCS*.

[13] Michael Backes, Sven Bugiel, Erik Derr, Patrick McDaniel, Damien Octeau, and Sebastian Weisgerber. 2016. On demystifying the android application framework:{Re-Visiting} android permission specification analysis. In *Security*.

[14] Arjun Nitin Bhagoji, Daniel Cullina, Chawin Sitawarin, and Prateek Mittal. 2018. Enhancing robustness of machine learning systems via data transformations. In *2018 52nd Annual Conference on Information Sciences and Systems (CISS)*. IEEE, 1–5.

[15] Xiao Chen, Chaoran Li, Derui Wang, Sheng Wen, Jun Zhang, Surya Nepal, Yang Xiang, and Kui Ren. 2019. Android HIV: A study of repackaging malware for evading machine-learning detection. *IEEE Transactions on Information Forensics and Security* 15 (2019), 987–1001.

[16] Yizheng Chen, Zhoujie Ding, and David A. Wagner. 2023. Continuous Learning for Android Malware Detection. In *32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023*.

[17] Francesco Croce and Matthias Hein. 2020. Reliable evaluation of adversarial robustness with an ensemble of diverse parameter-free attacks. In *International conference on machine learning*. PMLR, 2206–2216.

[18] Guneet S Dhillon, Kamyar Azizzadenesheli, Zachary C Lipton, Jeremy Bernstein, Jean Kossaifi, Aran Khanna, and Anima Anandkumar. 2018. Stochastic activation pruning for robust adversarial defense. *arXiv preprint arXiv:1803.01442* (2018).

[19] Lijie Fan, Sijia Liu, Pin-Yu Chen, Gaoyuan Zhang, and Chuang Gan. 2021. When does contrastive learning preserve adversarial robustness from pretraining to finetuning? *Advances in neural information processing systems* 34 (2021), 21480–21492.

[20] Ming Fan, Jun Liu, Wei Wang, Haifei Li, Zhenzhou Tian, and Ting Liu. 2017. Dapasa: detecting android piggybacked apps through sensitive subgraph analysis. *IEEE Transactions on Information Forensics and Security* 12, 8 (2017), 1772–1785.

[21] Yu Feng, Saswat Anand, Isil Dillig, and Alex Aiken. 2014. Apposcopy: Semantics-based detection of android malware through static analysis. In *Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering*. 576–587.

[22] Cuiying Gao, Gaozhun Huang, Heng Li, Bang Wu, Yueming Wu, and Wei Yuan. 2024. A Comprehensive Study of Learning-based Android Malware Detectors under Challenging Environments. In *ICSE*.

[23] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. 2020. Generative adversarial networks. *Commun. ACM* 63, 11 (2020), 139–144.

[24] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. 2014. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572* (2014).

[25] Kathrin Grosse, Nicolas Papernot, Praveen Manoharan, Michael Backes, and Patrick McDaniel. 2017. Adversarial examples for malware detection. In *Computer Security–ESORICS 2017: 22nd European Symposium on Research in Computer Security, Oslo, Norway, September 11-15, 2017, Proceedings, Part II 22*. Springer, 62–79.

[26] Jintao Gu, Hongliang Zhu, Zewei Han, Xiangyu Li, and Jianjin Zhao. 2024. GSEDroid: GNN-based android malware detection framework using lightweight semantic embedding. *Computers & Security* 140 (2024), 103807.

[27] Shixiang Gu and Luca Rigazio. 2014. Towards deep neural network architectures robust to adversarial examples. *arXiv preprint arXiv:1412.5068* (2014).

[28] Will Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. *Advances in neural information processing systems* 30 (2017).

[29] Kaiming He, Xinlei Chen, Saining Xie, Yanghao Li, Piotr Dollár, and Ross Girshick. 2022. Masked autoencoders are scalable vision learners. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 16000–16009.

[30] Ping He, Yifan Xia, Xuhong Zhang, and Shouling Ji. 2023. Efficient query-based attack against ML-based Android malware detection under zero knowledge setting. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. 90–104.

[31] Yiling He, Yiping Liu, Lei Wu, Ziqi Yang, Kui Ren, and Zhan Qin. 2022. MsDroid: Identifying malicious snippets for android malware detection. *IEEE Transactions on Dependable and Secure Computing* (2022).

[32] Shifu Hou, Yanfang Ye, Yangqiu Song, and Melih Abdulhayoglu. 2017. Hindroid: An intelligent android malware detection system based on structured heterogeneous information network. In *Proceedings of the 23rd ACM SIGKDD international conference on knowledge discovery and data mining*. 1507–1515.

[33] Zhenyu Hou, Xiao Liu, Yukuo Cen, Yuxiao Dong, Hongxia Yang, Chunjie Wang, and Jie Tang. 2022. Graphmae: Self-supervised masked graph autoencoders. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. 594–604.

[34] Weiwei Hu and Ying Tan. 2022. Generating adversarial malware examples for black-box attacks based on GAN. In *International Conference on Data Mining and Big Data*. Springer, 409–423.

[35] ElMouatez Billah Karbab and Mourad Debbabi. 2021. Petadroid: adaptive android malware detection using deep learning. In *DIMVA*.

[36] TaeGuen Kim, BooJoong Kang, Mina Rho, Sakir Sezer, and Eul Gyu Im. 2018. A multimodal deep learning method for android malware detection using various features. In *TIFS*.

[37] Deqiang Li and Qianmu Li. 2020. Adversarial deep ensemble: Evasion attacks and defenses for malware detection. *IEEE Transactions on Information Forensics and Security* 15 (2020), 3886–3900.

[38] Heng Li, Zhang Cheng, Bang Wu, Liheng Yuan, Cuiying Gao, Wei Yuan, and Xiapu Luo. 2023. Black-box Adversarial Example Attack towards FCG Based Android Malware Detection under Incomplete Feature Information. *arXiv preprint arXiv:2303.08509* (2023).

[39] Heng Li, ShiYao Zhou, Wei Yuan, Jiahuan Li, and Henry Leung. 2019. Adversarial-example attacks toward android malware detection system. *IEEE Systems Journal* 14, 1 (2019), 653–656.

[40] Heng Li, Shiyao Zhou, Wei Yuan, Xiapu Luo, Cuiying Gao, and Shuiyan Chen. 2021. Robust android malware detection against adversarial example attacks. In *Proceedings of the Web Conference 2021*. 3603–3612.

[41] Jin Li, Lichao Sun, Qiben Yan, Zhiqiang Li, Witawas Srisa-An, and Heng Ye. 2018. Significant permission identification for machine-learning-based android malware detection. *IEEE Transactions on Industrial Informatics* 14, 7 (2018), 3216–3225.

[42] Jiahao Liu, Jun Zeng, Fabio Pierazzi, Lorenzo Cavallaro, and Zhenkai Liang. 2024. Unraveling the Key of Machine Learning Solutions for Android Malware Detection. *arXiv preprint arXiv:2402.02953* (2024).

[43] Jiahao Liu, Jun Zeng, Xiang Wang, and Zhenkai Liang. 2023. Learning graph-based code representations for source-level functional similarity detection. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 345–357.

[44] Yue Liu, Chakkrit Tantithamthavorn, Li Li, and Yepang Liu. 2022. Deep learning for android malware defenses: a systematic literature review. *Comput. Surveys* 55, 8 (2022), 1–36.

[45] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. 2017. Towards deep learning models resistant to adversarial attacks. *arXiv preprint arXiv:1706.06083* (2017).

[46] Enrico Mariconti, Lucky Onwuzurike, Panagiotis Andriotis, Emiliano De Cristofaro, Gordon Ross, and Gianluca Stringhini. 2016. Mamadroid: Detecting android malware by building markov chains of behavioral models. *arXiv preprint arXiv:1612.04433* (2016).

[47] Niall McLaughlin, Jesus Martinez del Rincon, BooJoong Kang, Suleiman Yerima, Paul Miller, Sakir Sezer, Yeganeh Safaei, Erik Trickel, Ziming Zhao, Adam Doupé, et al. 2017. Deep android malware detection. In *CODASPY*.

[48] Dongyu Meng and Hao Chen. 2017. Magnet: a two-pronged defense against adversarial examples. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*. 135–147.

[49] Seung Jun Moon, Sangwoo Mo, Kimin Lee, Jaeho Lee, and Jinwoo Shin. 2021. MASKER: Masked Keyword Regularization for Reliable Text Classification. In

Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021, Thirty-Third Conference on Innovative Applications of Artificial Intelligence, IAAI 2021, The Eleventh Symposium on Educational Advances in Artificial Intelligence, EAAI 2021, Virtual Event, February 2-9, 2021.

[50] Alexander G Ororbia II, C Lee Giles, and Daniel Kifer. 2016. Unifying adversarial training algorithms with flexible deep data gradient regularization. *arXiv preprint arXiv:1601.07213* (2016).

[51] 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.

[52] Feargus Pendlebury, Fabio Pierazzi, Roberto Jordaney, Johannes Kinder, Lorenzo Cavallaro, et al. 2019. TESSERACT: Eliminating experimental bias in malware classification across space and time. In *USENIX Security*.

[53] Hao Peng, Chris Gates, Bhaskar Sarma, Ninghui Li, Yuan Qi, Rahul Potharaju, Cristina Nita-Rotaru, and Ian Molloy. 2012. Using probabilistic generative models for ranking risks of android apps. In *Proceedings of the 2012 ACM conference on Computer and communications security*. 241–252.

[54] Fabio Pierazzi, Feargus Pendlebury, Jacopo Cortellazzi, and Lorenzo Cavallaro. 2020. Intriguing properties of adversarial ml attacks in the problem space. In *2020 IEEE symposium on security and privacy (SP)*. IEEE, 1332–1349.

[55] Maryam Shahpasand, Len Hamey, Dinusha Vatsalan, and Minhui Xue. 2019. Adversarial attacks on mobile malware detection. In *2019 IEEE 1st International Workshop on Artificial Intelligence for Mobile (AI4Mobile)*. IEEE, 17–20.

[56] Florian Tramèr, Alexey Kurakin, Nicolas Papernot, Ian Goodfellow, Dan Boneh, and Patrick McDaniel. 2017. Ensemble adversarial training: Attacks and defenses. *arXiv preprint arXiv:1705.07204* (2017).

[57] Laurens Van der Maaten and Geoffrey Hinton. 2008. Visualizing data using t-SNE. *Journal of machine learning research* 9, 11 (2008).

[58] Pascal Vincent, Hugo Larochelle, Yoshua Bengio, and Pierre-Antoine Manzagol. 2008. Extracting and composing robust features with denoising autoencoders. In *Proceedings of the 25th international conference on Machine learning*. 1096–1103.

[59] VirusTotal. 2023. VirusTotal - Free Online Virus, Malware and URL Scanner. https://www.virustotal.com

[60] Qinglong Wang, Wenbo Guo, Kaixuan Zhang, Alexander G Ororbia, Xinyu Xing, Xue Liu, and C Lee Giles. 2017. Adversary resistant deep neural networks with an application to malware detection. In *Proceedings of the 23rd ACM sigkdd international conference on knowledge discovery and data mining*. 1145–1153.

[61] Bozhi Wu, Sen Chen, Cuiyun Gao, Lingling Fan, Yang Liu, Weiping Wen, and Michael R Lyu. 2021. Why an android app is classified as malware: Toward malware classification interpretation. *ACM Transactions on Software Engineering and Methodology (TOSEM)* (2021).

[62] Huijun Wu, Chen Wang, Yuriy Tyshetskiy, Andrew Docherty, Kai Lu, and Liming Zhu. 2019. Adversarial examples on graph data: Deep insights into attack and defense. *arXiv preprint arXiv:1903.01610* (2019).

[63] Yueming Wu, Xiaodi Li, Deqing Zou, Wei Yang, Xin Zhang, and Hai Jin. 2019. Malscan: Fast market-wide mobile malware scanning by social-network centrality analysis. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 139–150.

[64] Yueming Wu, Deqing Zou, Wei Yang, Xiang Li, and Hai Jin. 2021. Homdroid: detecting android covert malware by social-network homophily analysis. In *Proceedings of the 30th acm sigsoft international symposium on software testing and analysis*. 216–229.

[65] Chong Xiang, Arjun Nitin Bhagoji, Vikash Sehwag, and Prateek Mittal. 2021. PatchGuard: A Provably Robust Defense against Adversarial Patches via Small Receptive Fields and Masking. In *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*.

[66] Ke Xu, Yingjiu Li, Robert H Deng, and Kai Chen. 2018. Deeprefiner: Multi-layer android malware detection system applying deep neural networks. In *EuroS&P*.

[67] Xufeng Yao, Yang Bai, Xinyun Zhang, Yuechen Zhang, Qi Sun, Ran Chen, Ruiyu Li, and Bei Yu. 2022. Pcl: Proxy-based contrastive learning for domain generalization. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 7097–7107.

[68] Jun Zengy, Xiang Wang, Jiahao Liu, Yinfang Chen, Zhenkai Liang, Tat-Seng Chua, and Zheng Leong Chua. 2022. Shadewatcher: Recommendation-guided cyber threat analysis using system audit records. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 489–506.

[69] Lan Zhang, Peng Liu, Yoon-Ho Choi, and Ping Chen. 2022. Semantics-preserving reinforcement learning attack against graph neural networks for malware detection. *IEEE Transactions on Dependable and Secure Computing* 20, 2 (2022), 1390–1402.

[70] Xiaohan Zhang, Yuan Zhang, Ming Zhong, Daizong Ding, Yinzhi Cao, Yukun Zhang, Mi Zhang, and Min Yang. 2020. Enhancing State-of-the-art Classifiers with API Semantics to Detect Evolved Android Malware. In *CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020*.

[71] 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. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 3218–3235.