MSDROID: Identifying Malicious Snippets for Android Malware Detection

Yiling He, Yiping Liu, Lei Wu, Ziqi Yang, Kui Ren, *Fellow, IEEE,* and Zhan Qin

Abstract—Machine learning has shown promise for improving the accuracy of Android malware detection in the literature. However, it is challenging to (1) stay robust towards real-world scenarios and (2) provide interpretable explanations for experts to analyse. In this paper, we propose MSDROID, an Android malware detection system that makes decisions by identifying malicious snippets with interpretable explanations. We mimic a common practice of security analysts, i.e., filtering APIs before looking through each method, to focus on local snippets around sensitive APIs instead of the whole program. Each snippet is represented with a graph encoding both code attributes and domain knowledge and then classified by Graph Neural Network (GNN). The local perspective helps the GNN classifier to concentrate on code highly correlated with malicious behaviors, and the information contained in graphs benefit in better understanding of the behaviors. Hence, MSDROID is more robust and interpretable in nature. To identify malicious snippets, we present a semi-supervised learning approach that only requires app labeling. The key insight is that malicious snippets only exist in malwares and appear at least once in a malware. To make malicious snippets less opaque, we design an explanation mechanism to show the importance of control flows and to retrieve similarly implemented snippets from known malwares. A comprehensive comparison with 5 baseline methods is conducted on a dataset of more than 81K apps in 3 real-world scenarios, including *zero-day*, *evolution*, and *obfuscation*. The experimental results show that MSDROID is more robust than state-of-the-art systems in all cases, with 5.37% to 49.52% advantage in F1-score. Besides, we demonstrate that the provided explanations are effective and illustrate how the explanations facilitate malware analysis.

Index Terms—Android Malware Detection, Graph Neural Network, Explainable Machine Learning, Static Code Analysis.

1 INTRODUCTION

ANDROID malware is emerging in an endless stream that more than 83 million new malware infections was reported in 2020 [1]. Existing works have explored machine learning (ML) in Android malware detection to relieve the burden of manual analysis, which seem to achieve great success [2], [3], [4], [5], [6], [7], [8]. However, the problem is not yet solved properly. It was pointed that over-optimistic results caused by experimental bias are universal in existing works [9]. In the real-world setting, Android apps evolve quickly with different implementations and functions over time [10]. The usage of the obfuscation technique by both software developers and malware authors also grows fast [11], [12]. In addition, most existing malware classifiers are designed as black boxes and cannot provide meaningful information for further analysis.

Focusing on ML-based Android malware detection methods that use features extracted from static code analysis, we observe that technical advances are generally made in two paths. First, the learning process shifts from pattern-driven to data-driven, becoming more independent of manually designed properties [5], [7], [13]. In particular, sequence-based methods extract raw sequences of APIs or opcodes from bytecode and avoid complex feature engineering [4], [14]. These methods mostly benefit from the success of deep learning techniques (e.g., CNN, RNN) in NLP tasks, without special considerations of control-flow information that is essential in programs. Second, many works combine various features to improve the resilience of downstream classifiers [15]. For example, Arp et al. [2] concatenate eight sets of features in feature space, and Kim et al. [6] propose using multimodal learning to train different features jointly. These designs separat different feature sets without considering the hidden relationships during pre-processing. In a word, existing feature representation methods cannot capture adequate relations, and thus degrade the performance of downstream classifiers in behavior modeling.

Some existing works get ML decision explained with the weights of separate features, e.g., specific APIs or permissions [16], [17]. For example, DREBIN [2] associates SVM weights with the importance of their pattern-based features. A few existing neural network (NN) based detection methods provide meaningful explanations [18]. To be fair, the difficulty in explainability is an inherited challenge of NN-based techniques. None of existing explanation techniques explore the logic behind control flows, failing to identify malicious snippets in malware.

Given these problems, we aim to enhance the robust-

- *Yiling He, Lei Wu, Ziqi Yang, Kui Ren, and Zhan Qin are with the Department of Computer Science and Technology, Zhejiang University, Hangzhou 310027, China.*
  E-mail: {yilinghe, lei_wu, yangziqi, kuiren, qinzhan}@zju.edu.cn
- *Yiling He and Zhan Qin are also with ZJU-Hangzhou Global Scientific and Technological Innovation Center, Hangzhou 311215, China.*
- *Yiping Liu is with the School of Cyberspace Security, Beijing University of Posts and Telecommunications, Beijing 100876, China.*
  Email: ypinglinkle@gmail.com
- *Zhan Qin is the corresponding author.*

ness of existing methods by filling up the missing relations in malware representation. In addition, we target to make the detection results more interpretable for further security analysis. To tackle the challenges, we start from building APK representations with graph-structured data. There are two motivations: 1) graph data is superior in expressing relations than regular Euclidean structured data; 2) analyzer can perform finer relation exploration with a graph for a more interpretable reasoning process [19], [20]. To automatically encode graph structure into low-dimensional embedding without compromise [21], [22], [23], [24], [25], we utilize GNN techniques for inductive graph-level classification.

We propose a snippet representation method called *behavior subgraph set* (BSS) to model sensitive behaviors of Android apps, with the key insight that malicious code is implemented in several snippets around sensitive APIs. To extract features, we rule out behavior properties like strings and resource files [26], while choosing three robust properties instead, i.e., *call graph*, *opcode*, and *permission*. The usage of the former two are straightforward and common in existing data-driven methods [27], [28], [29]. The motivations behind involving the latter one are: 1) permissions determine whether an app can access certain sensitive data, thus highly related to malicious behaviors; 2) permissions are defined by Android permission-based security model, so it is a rather stable knowledge. Moreover, we do not use the whole call graph because

- Call graph is too large for training a GNN on the graph-level classification task, e.g., call graphs of the popular academic APK dataset AMD [30] have 7K nodes and 19K edges on average;
- Call graph is so redundant for malware detection task that hinders behavior capturing.

Therefore, we transfer the app classification problem into subgraph classification tasks. Based on the observation that malware has at least one malicious behavior while a benign app has none, we propose *app subgraph loss* to handle the semi-supervised training process.

We evaluate the performance of MSDROID on a dataset of 81,790 Android apps spanning over six years, and it achieves 97.82% testing accuracy. We also conduct an extensive evaluation to compare MSDROID with five baseline models in terms of the effectiveness and the transferability. Particularly for transferability, we consider three critical problems in the malware detection domain, including zero-day threat, app evolution, and code obfuscation. The evaluation demonstrates that MSDROID outperforms the state-of-art methods (i.e., MAMADROID [3], DEEPMALDET [29]) with up to 9.15% F1-score on test-set, 38.47% accuracy on zero-day malware, 37.81% AUT(f1) on evolved apps, and 26.12% F1-sore on obfuscated apps. On *Reflection* obfuscation, the F1-score of MSDROID are 49.52% and 47.02% higher, and this success is further interpreted by *behavior subgraph* statistics and a case study leveraging the proposed explanation mechanism. Moreover, we analyze the first level explanation results on malware families and confirm their consistency with the actual behaviors. Two case studies conducted on an obfuscated app and a COVID-19 themed app also present how the proposed explanation mechanism can help with real-world malware analysis.

Findings. Since MSDROID combines three types of features jointly, we measure their effectiveness in different scenarios, and the findings are concluded as follows:

- *Opcode* brings the biggest improvement to test performance, but it is the most fragile feature towards new samples.
- *Permission* makes the most robust feature towards over-time app evolution.
- *Call graph* structure renders the model more capable of detecting anomalies.

Contributions. To the best of our knowledge, we are the first to apply GNN in static code analysis for Android malware (snippet) detection. The main contributions of our work are summarized as follows:

- We design a novel snippet-based Android malware detector named MSDROID, which identifies malicious snippets in malware with GNN. We represent each snippet with a graph, using robust behavioral properties including *call graph*, *opcode*, and *permission*, to preserve the calling and cross-modal relationships. We propose a customized loss function to train a snippet classifier where no specialized labeling effort is required.
- We evaluate MSDROID on a dataset of more than 81K apps and conduct a comprehensive comparison with 5 baseline techniques in 4 settings. Experiments show that MSDROID reaches 97.82% testing accuracy, and is significantly more robust than state-of-the-art methods towards the zero-day threat, app evolution, and code obfuscation. Especially in *Reflection* obfuscation, the F1-score is with up to 49.52% higher. Moreover, we get three findings of feature effectiveness.
- We propose a three-level explanation mechanism to facilitate malware analysis. It identifies suspicious API usages, visualizes malicious code with *calling heat graph* that tells questionable control flows, and gives similar behavioral snippets of known malware. The mechanism is proved useful with a family analysis and two case studies.

We make the code and graph dataset availbale[1].

2 OVERVIEW

2.1 Problem Definition

Given an Android app, a malware detection system is supposed to distinguish whether it is malicious or not. As a static call graph intends to approximate the behavior of the real program by representing every possible run, it can be an informative proxy for NN algorithms to explore. We formally define the call graph as $G = (\mathcal{V}, \mathcal{E})$, where

1. https://github.com/E0HYL/MsDroid

each element $v \in \mathcal{V}$ represents an API and each edge $e = (v_1, v_2) \in \mathcal{E}$ indicates an invocation from v_1 to v_2.

However, malicious behaviors are generally included into *snippets* in malware, with a proportion estimated at 0.18% and is vastly outnumbered by non-malicious code [31], [32]. Based on the observation that malicious behaviors are typically implemented within several semantic-rich APIs, we propose Behavior Subgraph Set (BSS) to represent a set of possible malicious snippets for each program. We then decompose the malware detection problem into subgraph classification. The BSS and the detection problem are defined as follows:

Behavior Subgraph Set (BSS). A behavior subgraph set $\mathcal{G}$ is a set of graphs with node features, as in:

$$\{g = (G[\mathcal{S}], \boldsymbol{X}) \mid G[\mathcal{S}] \in sub(G), \boldsymbol{X} = nod(\mathcal{S} \mid G)\}, \quad (1)$$

where $G[\mathcal{S}]$ is the induced subgraph of call graph G with a vertex set $\mathcal{S} \subset \mathcal{V}$, $sub(\cdot)$ is the method to generate subgraph set $\{G[\mathcal{S}_i] \mid i \leq n, n \in \mathbb{N}_+\}$, and $nod(\cdot)$ defines the way to generate API features $\boldsymbol{X}_i$ for each node set $\mathcal{S}_i$.

Decomposed Detection Problem. An Android app is malicious as long as there exists malicious behavior. Thus, given a classifier $\mathcal{C} : g \to \{0, 1\}$ that maps each malicious behavior to a positive value, the detection output of an app with a behavior subgraph set $\mathcal{G}$ should be:

$$\text{sign}\left(\sum_{g \in \mathcal{G}} \mathcal{C}(g)\right). \quad (2)$$

Malware is detected if the output is positive. Otherwise, the input app is regarded as benign.

Following the definitions, two main detection tasks should be: 1) design $sub(\cdot)$ and $nod(\cdot)$ to generate $\mathcal{G}$ for Android apps, which can catch sufficient behavioral semantics; 2) build $\mathcal{C}$ to automatically learn on behavior subgraphs without explicit labeling, which generalizes to malware identification.

2.2 System Architecture

As shown in Fig. 1, MSDROID goes through four phases: 1) pre-processing, 2) BSS generation, 3) GNN-based detection, and 4) post-processing. The two phases in the middle are corresponding solutions to the two detection tasks. To help with security analysis, MSDROID also makes efforts to explain malware snippets in the last phase.

First, MSDROID disassembles Android apps and extracts call graphs through static analysis. Apps are also sent to API-permission check and third party library detection. Second, the subgraph set generation is guided by those permission-related APIs. Nodes of each subgraph are divided into four types and node features are extracted accordingly. Then, MSDROID leverages GNN techniques to perform automatic learning on graph-structured data (i.e., behavior subgraphs) to identify malicious behaviors. Especially when training, an *app subgraph loss* function is proposed since no explicit subgraph labels are available. For a malicious app, explainable detection results are made in three levels based on subgraph predictions, GNN model explanation, and graph embedding similarities.

3 METHODOLOGY

In this section, we discuss technical details of the four phases in MSDROID.

3.1 Pre-processing

For an input app, MSDROID disassembles it and extracts the call graph G with the help of Androguard [33]. API semantics are collected in two ways. (1) To distinguish permission-related (aka "sensitive") APIs $\mathcal{V}_{per} \subset \mathcal{V}$, MSDROID leverages two commonly used API-permission mappings from PSCout [34] and Axplorer [35]. However, Android apps can perform CRUD (create, retrieve, update, and delete) operations to content providers, which also access sensitive permissions. For example, querying `content://mms` involves the permission `READ_SMS`. Since the CRUD operations are handled by methods from interface `ContentResolver`, MSDROID analyses these methods and complements the API-permission mappings as M_{per}. (2) To identify APIs that belong to standalone functional modules, MSDROID uses LibRadar [36] to detect third-party libraries (TPLs) within the app.

3.2 BSS Generation

3.2.1 Subgraph Set Generation

Considering each permission access in code snippets as a behavior of interest, MSDROID realizes $sub(\cdot)$ on the basis of k-hop neighborhoods of permission-related APIs on G. Specifically for each $v_i \in \mathcal{V}_{per}$, a behavior subgraph is induced by $\mathcal{S}_i$ ($|\mathcal{S}_i| > 1$), which means it has $\mathcal{S}_i$ as its set of vertices and preserves all the edges of G that have both endpoints in $\mathcal{S}_i$. Therefore, a subgraph generation process is equivalent to selecting the node set $\mathcal{S}_i$. The selection is done in the following steps:

- *Initialization.* The node set is initialized with v_i's k-hop neighborhood $\mathcal{N}_k(v_i)$. The neighbor nodes are selected because they may either (in)directly call the sensitive API or are invoked by those callers, being most likely to contain malicious snippets.
- *Partition.* $\mathcal{N}_k(v_i)$ is divided into four subsets using the previous collection of API semantics. For external APIs (without disassembled bytecode), they belong to $\mathcal{S}_{per}$ or $\mathcal{S}_{non}$ depending on whether they exist in M_{per}. For internal APIs, those who are part of TPLs are categorized into $\mathcal{S}_{tpl}$ and otherwise are regarded as user-defined APIs in $\mathcal{S}_{usr}$.
- *Reduction.* Three types of nodes are removed from the subsets: (R1) nodes in $\mathcal{S}_{tpl}$, $\mathcal{S}_{per}$ and $\mathcal{S}_{non}$ that are not directly invoked by any node in $\mathcal{S}_{usr}$, (R2) nodes in $\mathcal{S}_{usr}$ that actually cannot reach v_i, and (R3) leftover nodes that are isolated from v_i.

If the resulting subset $\mathcal{S}_{usr}$ is not empty, we then add v_i to $\mathcal{S}_{per}$ and generate a induced subgraph from the union of the four subsets.

We explain the intuition behind *Reduction* with a toy example illustrated in Fig. 2. Firstly, TPLs integrated into apps are often so large that constitute noises in malware

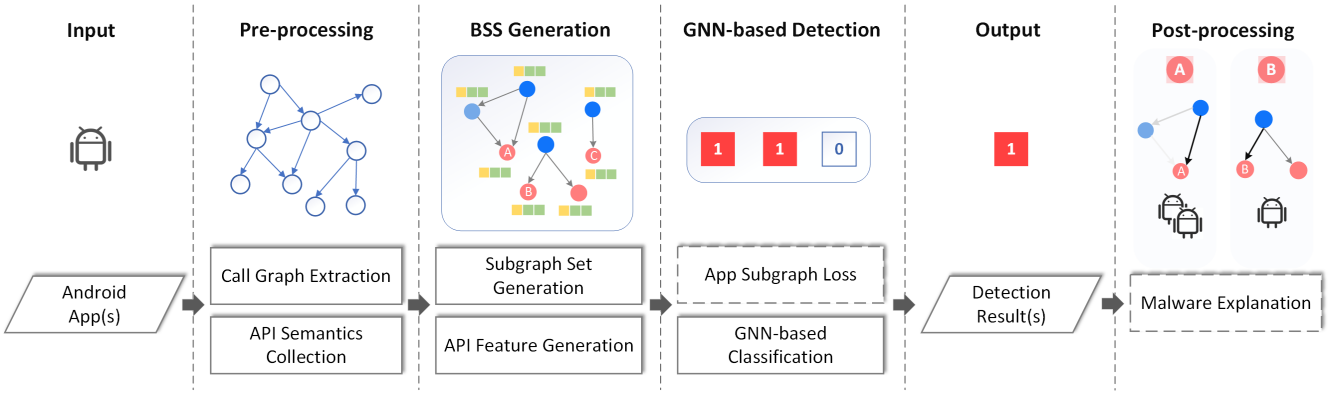


Fig. 1. System architecture of MSDROID. For an input Android app, it is pre-processed to get a API call graph while the API semantics are collected from bytecode and domain knowledge. BSS (representations of sensitive snippets in an app) is then generated from subgraphs on the call graph and API semantics are encoded in node features. Next, a GNN classifier is trained/tested to give a label of each BS. The system outputs a malware label if one BS is labeled malicious. Finally, a detected malware will be post-processed to explain the malicious snippets.

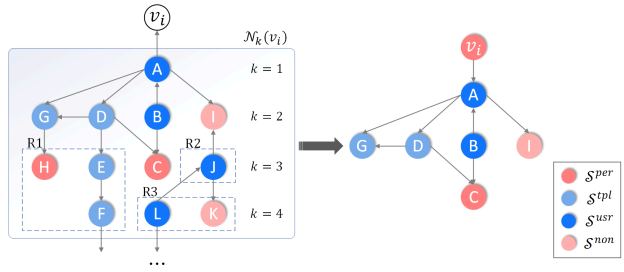


Fig. 2. Example of generating a *behavior subgraph* (right) from a call graph (left) with $k = 4$.

detection [37]. To understand a malware behavior, we may simply wants to know the functionality rather than the implementation details of TPLs. As the left part of the call graph in the figure comes from code in TPLs, we reduce the massive code bodies and only keep 'G' and 'D' that are directly invoked by 'A'. Secondly, some user-defined methods are falsely included as the current behavior of interest in $\mathcal{N}_k(v_i)$. For example, 'J' is included because it calls the same non-sensitive external API 'I' as 'A'. It would be a common situation if 'I' performs a fundamental utility such as `android.util.Log.d()`. Lastly, singleton nodes 'L' and 'K' are removed as they were previously introduced by 'J', and we consider them as coming from different behavioral snippets.

3.2.2 API Feature Generation

MSDROID accomplishes $nod(\cdot)$ by analyzing the disassembled bytecode and requested permissions of APIs in $\{\mathcal{S}_i\}$. As shown in Algorithm 1, the node feature is a concatenation of opcode and permission feature vectors (Line 14). Primary functions that generate the two feature vectors work as follows: 1) GetOpVector (Line 6) calculates the normalized frequency of different opcodes for internal APIs and outputs zero vectors for external APIs; 2) GetPerVector (Line 7) generates binary vectors that indicate a permission is required from an API or not, which would be non-zero vectors for external APIs in M_{per}. Note that the second function cannot be directly implemented as API name matching. For better understanding, we present an example here: if class A inherits from class `SmsManager` and A does not override the API `sendTextMessage()`, then the exact matching wouldn't identify `A.sendTextMessage()` as sensitive despite being the same as `SmsManager.sendTextMessage()`. To handle this, MSDROID finds out the base class of external nodes recursively and substitute the class name before looking up the required permissions in M_{per}.

Since the use of TPLs was simplified in the subgraph structure, MSDROID enriches the behavioral semantics of nodes in $\mathcal{S}_{tpl}$ by aggregating permission features from their successors (Lines 8-13). *GetSenNodes* applies Depth-First-Search to get sensitive child nodes of n. During the search, asynchronous calls are additionally handled where a calling convention M_{asy} that maps RunMethod to StartMethod is used as suggested [28]. The final permission vector of an internal TPL API is the average of its sensitive child nodes. The v_{per} of 'D' (Fig. 2), for instance, is averaged over that of 'H' and 'C'.

3.3 GNN-based Detection

3.3.1 Build GNN-based Classifier

On graph data, Graph Neural Networks (GNNs) could yield more promising results than traditional descriptor-based methods [38], [39]. They broadly follow a recursive *neighborhood message passing* scheme, where each node aggregates feature vectors from neighbors to update the feature vector of itself. After several *iterations* of aggregation, each node is represented by a feature vector that captures certain information, and the representation of entire graph can be obtained through *graph pooling*.

MSDROID adopts a message passing scheme, which achieves SOTA performance on graph-level tasks [25], for behavior subgraph classification. It updates node features with the MLP technique as:

$$h_v^{(i)} = \text{MLP}^{(i)}\big((1+\epsilon^{(i)})\cdot h_v^{(i-1)} + \sum_{u\in\mathcal{N}(v)} h_u^{(i-1)}\big), \quad (3)$$

where $h_v^{(i)}$ is the feature vector of node v at the i-th iteration and $\mathcal{N}(v)$ is a set of nodes adjacent to v.

To build $\mathcal{C}$, we firstly implement the MLP with two 128-neuron hidden layers and make the parameter ϵ a

Algorithm 1: Feature generation

Input: (G, M_{per}) obtained from pre-processing
Input: $(\mathcal{S}_{per}, \mathcal{S}_{non}, \mathcal{S}_{usr}, \mathcal{S}_{tpl})$
Output: $\boldsymbol{X}$ Feature matrix of all nodes in nodeList

$M_{asy} \leftarrow$ GetAsync()
$\mathcal{S} \leftarrow \mathcal{S}_{per} \cup \mathcal{S}_{non} \cup \mathcal{S}_{usr} \cup \mathcal{S}_{tpl}$
$\boldsymbol{X} \leftarrow \mathbb{O}_{|\mathcal{S}|,D}$
$i \leftarrow 0$
for $n \in \mathcal{S}$ **do**
  $\vec{v}_{opc} \leftarrow$ GetOpVector(n)
  $\vec{v}_{per} \leftarrow$ GetPerVector(n, M_{per})
  if $n \in \mathcal{S}_{tpl}$ **then**
    $\mathcal{S}_{tmp} \leftarrow$ GetSenNodes(n, M_{asy}, $\mathcal{S}_{per}$)
    $\mathcal{S}_{vec} \leftarrow \{\}$
    for $t \in \mathcal{S}_{tmp}$ **do**
      $\mathcal{S}_{vec} \leftarrow \mathcal{S}_{vec} \cup$ GetPerVector(t, M_{per})
    $\vec{v}_{per} \leftarrow$ Mean$(\mathcal{S}_{vec})$
  $\vec{v}_{nod} \leftarrow \vec{v}_{opc} \parallel \vec{v}_{per}$
  $\boldsymbol{X}[i] \leftarrow \vec{v}_{nod}$
  $i \leftarrow i + 1$

Function *GetSenNodes(n, M_{asy}, $\mathcal{S}_{per}$)*
  $\mathcal{S}_{suc} \leftarrow$ GetSuccessors(n)
  $\mathcal{S}_{tmp} \leftarrow \{\}$
  for $t \in \mathcal{S}_{suc}$ **do**
    if $t \in S_{per}$ **then**
      $\mathcal{S}_{tmp} \leftarrow \mathcal{S}_{tmp} \cup \{t\}$
    else
      if $t \in M_{asy}$ **then**
        $t \leftarrow M_{asy}[t]$
      if t *is not external* **then**
        $\mathcal{S}_{tsu} \leftarrow$ GetSuccessors(t)
        for $c \in \mathcal{S}_{tsu}$ **do**
          $\mathcal{S} \leftarrow$ GetSenNodes(c, M_{asy}, S_{per})
          $\mathcal{S}_{tmp} \leftarrow \mathcal{S}_{tmp} \cup \mathcal{S}$
  return $\mathcal{S}_{tmp}$

fixed scalar to 0. Secondly, API node representations are updated in 3 iterations. Next, we combine global max-pooling and average-pooling in the graph-level readout function, to embed each behavior subgraph as:

$$h_{G^B} = \max(h_v \mid v \in G^B) \parallel \text{avg}(h_v \mid v \in G^B)\,. \quad (4)$$

The intuition is that max-pooling learns distinct elements while average-pooling learns distribution. Because both anomalous API usages and similarly used malware APIs are valuable for malware detection, the proposed method also works better experimentally than max or mean pooling alone. Lastly, a two-layer MLP for post-message passing is added to output the two-class prediction.

3.3.2 Train and Test

As in many binary classification tasks, Android malware classifiers are typically trained in a supervised fashion. This approach is straightforward and large-scale dataset consisting of apps and their labels is easily available. However, to train $\mathcal{C}$ that identifies malicious behaviors, we cannot use the same approach since large-scale labeling of malicious code requires enormous efforts and hasn't won attention yet. Based on the knowledge that benign apps don't have any malicious behavior while malware has at least one, we use the existing app labels to perform a semi-supervised training. In other words, all behavior subgraphs of a benign app get label 0, but for a malicious app, the label of each behavior subgraph is undetermined. To tackle the challenge in training $\mathcal{C}$, we design the loss function as below.

App Subgraph Loss. The prediction loss for each app with a BSS $\mathcal{G}$ and a label y is calculated as:

$$-\left((1-y)\,\frac{1}{|\mathcal{G}|}\sum_{g\in\mathcal{G}} log(1-p_g) + y \min_{g\in\mathcal{G}} log(p_g)\right)\,, \quad (5)$$

where p_g represents the predicted probability of class 1 for a behavior subgraph $g \in \mathcal{G}$. The first item is designed to ensure the accuracy of predicting behavior subgraphs in benign apps; the second item aims to make the subgraph that is most likely to be *malicious* in malware have a high probability. Note that one can replace min-pooling in (5) with other methods like weighted sum, but a simple min-pooling can lead to less false positives in practice.

In testing period, with the trained $\mathcal{C}$, we use (2) to aggregate the prediction results of behavior subgraphs for each app. That is, an app is classified as malware if there exists the output label 1 in $\{\mathcal{C}(g)\}$.

3.4 Malware Explanation

MSDROID generates three kinds of explanation results for malware (i.e., suspicious API usages, *calling heat graph*s, similar behavioral snippets), mainly for the purpose of facilitating reverse engineering. The overall workflow for explaining one suspicious behavior is demonstrated in Fig. 3, and more specific examples for real-world apps can be found in Section 4.4. We introduce the reasons and implementations of the design as follows.

3.4.1 Suspicious API Usage

MSDROID provides explanations in API and permission level as some existing works. The results come naturally since the detection decision for an app is made upon classification results of its *behavior subgraph*s. Each subgraph captures behavior implemented around a certain sensitive permission-related API. Therefore, for a detected malware, we give all its behavior subgraphs that are classified as *malicious*, along with their corresponding sensitive APIs with permission semantics (e.g., behavioral categories).

3.4.2 Calling Heat Graph

As a *behavior subgraph* contains program-wide control flow information on edges, we further identify those important control flows that are crucial to implement a malicious behavior, and visualize it with a *calling heat*

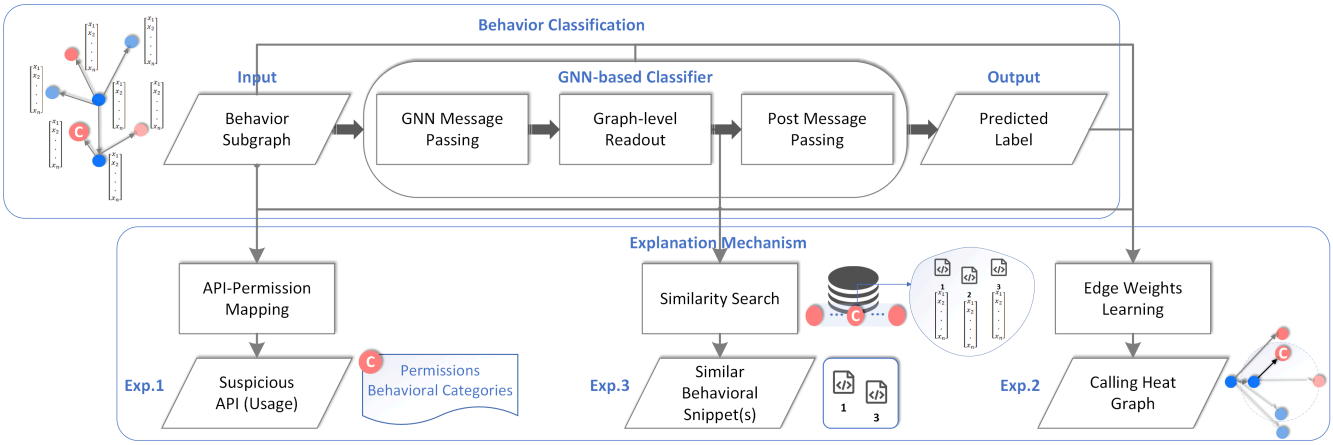
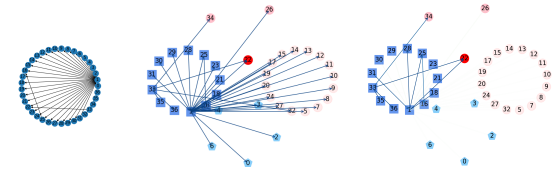


Fig. 3. GNN-based Classification and Explanation Mechanism for an example behavior subgraph with sensitive API 'C'.



(a) original (b) positioned (c) weighted

Fig. 4. *Calling Heat Graph* Construction

graph. The *calling heat graph* of an example malware's suspiciously used API `isWifiEnabled()` is given in Fig. 4(c) (see node mappings in Appendix D).

Inspired by *circular layout* which is shown in Fig. 4(a), we design our layout algorithm for graphs with multiple node types, where the position for the jth node in ith type node set $\mathcal{S}_i$ is calculated as:

$$\begin{cases} x_{i,j} = R_1 \cos(\frac{2\pi}{N} i) + R_2 \cos(\frac{2\pi}{|\mathcal{S}_i|} j), \\ y_{i,j} = R_1 \sin(\frac{2\pi}{N} i) + R_2 \sin(\frac{2\pi}{|\mathcal{S}_i|} j), \end{cases} \tag{6}$$

After positioning and coloring all nodes by types, we get Fig. 4(b). As we can see, a *behavior subgraph* can have tens (or even hundreds) of edges, so next we introduce how their importance are learned.

Edge Weights Learning. We identify the importance of edges for the GNN-based model's prediction to make edge weights. The problem is formulated as an optimization task that maximizes the mutual information (MI) between GNN prediction and distribution of possible edge dependencies [40]. This means, given a behavior subgraph $G^B = (\mathcal{V}^B, \mathcal{E}^B)$, if removing edge subset $E_S \in E$ strongly decreases the probability of original prediction y made by classifier $\mathcal{C}$, then the absence of those edges can be a good counterfactual explanation. Focusing on the *malicious* class, a computationally efficient version of objective using a mask approach is formulated as follows:

$$\min -\log \mathrm{P}_{\mathcal{C}}(Y = 1 \mid E = A \odot \sigma(M)), \tag{7}$$

where $A \in \mathbb{N}^{n \times n}$ denotes the adjacency matrix of behavior subgraph G^B, $\odot$ denotes element-wise multiplication, and σ is the sigmod function that maps the learned edge mask $M \in \mathbb{R}^{n \times n}$ to $[0, 1]^{n \times n}$. To provide more meaningful explanations, we add three regularization terms to (7): a) element-wise entropy to encourage discrete edge masks; b) element-wise sum to penalize large size explanations; c) sum of elements in $\sigma(M)$ that map to edges whose end nodes are in $\mathcal{S}_{non}$, to discourage explanation edges from ending with non-sensitive Android APIs.

The elements in matrix $\sigma(M)$ are what we use as edge weights to determine edge colors on *calling heat graph*. Thus, a reverse engineer can trace malicious code around important API dependencies, i.e., dark-colored edges.

3.4.3 Similar Behavioral Snippets

Code clone search enables many critical security usages and can reduce the burden of manual analysis involved in reverse engineering [41] [42]. In malware analysis scenario, if any similarly implemented behavior is found in a known malware database, an expert can locate malicious code based on prior knowledge. MsDroid can accomplish this by calculating similarities between *behavior subgraph*s. We take the embedding after the graph-level readout function as the behavioral representation. Therefore, for two behavior representations e_1 and e_2, the similarity between them is calculated as:

$$sim(e_1, e_2) = 1/d(e_1, e_2)\,, \tag{8}$$

which is the reciprocal of their Euclidean distance. Additionally, to speed up the search process, we group those *behavior subgraph*s centered with the same sensitive APIs. Thus, for a given *behavior subgraph*, similarity calculations only need to be done in one group, which reduces much pairwise computation cost.

4 Evaluation

4.1 Experimental Setup

We implement our system in Python and release the code on GitHub. Specifically, we use Androguard with "DAD" decompiler option to perform APK analysis and LibRadar to detect third-party libraries during pre-processing. The

Table 1. Evaluation Dataset.

Class	Tag	Date Range	# Apps	Call Graph			Behavior Subgraph		
				# Graphs	Avg # Nodes	Avg # Edges	# Graphs	Avg # Nodes	Avg # Edges
Malicious	Drebin	2010.10-2012.08	5560	5373	2531	8003	106307	35	122
	AMD	2010.07-2016.05	24650	23923	6585	18852	873623	31	89
Benign	Old	2010.10-2012.08	11580	11169	2719	8409	181549	31	110
	New	2012.09-2016.05	40000	38320	6457	16838	1440929	22	58

graph generation and learning process is implemented based on Pytorch Geometric framework. The graph visualization in the explanation mechanism is done with the help of the NetworkX package. Now we introduce the dataset and baseline techniques that we use in evaluation.

4.1.1 Dataset

We utilize three popular academic datasets to construct our APK dataset, namely Drebin [2], AMD [30], and Androzoo [43]. As the first two are well-studied malware datasets with family labels, we preserve all apps in them. Then with the help of Androzoo, we collect apps that are reported as benign by all the AVs (Antivirus software) from VirusTotal [2]. We form our benign set as follows: 1) *Old*: download apps whose appearance timestamps are within the same date range of Drebin, which results in 11,580 apps; 2) *New*: divide the date range of AMD that is not overlapped with Drebin into four periods of 11 months and collect 10,000 apps for each of them, which makes a set of 40,000 benign apps. In the end, our APK dataset consists of 81,790 Android apps spanning over 6 years, where includes 30,210 malicious and 51,580 benign. Moreover, we obfuscate *Drebin* and *Old* to get the obfuscated dataset for transferability evaluation, and COVID-19 themed apps [44] are used for case study analysis.

Our graph dataset consists of 2,602,408 *behavior subgraph*s generated from call graphs of the APK dataset. Due to failures of the decompiler, some call graphs can't be extracted from APKs successfully. But as each subdataset shares a similar decompile failure rate at around 3.5%, the malware ratio won't be much influenced to meet the spatial consistency. We give further details about the graph dataset in Table 1. From the call graph statistics, we can see that newer apps tend to be larger and their call graphs have thousands of nodes and tens of thousands of edges on average. Differently, for the two-hop-based behavior subgraph, although the subgraph number of apps increases over time, the graph size decreases. Moreover, benign apps have fewer nodes and edges in behavior subgraphs than malware consistently. This result also implies that our subgraph reduction step is necessary for both efficiency and effectiveness.

4.1.2 Baseline Techniques

First, we use a mask approach on API node features to study the feature effectiveness of our method. Specifically, we call the model where either opcode or permission features are preserved as MsDroid-opc and MsDroid-per, respectively. The model where all the node features are set to the same numeric value is called MsDroid-non . Second, we compare our method with two state-of-the-art malware classifiers, which use the same program property (i.e., DEX files) with us to generate features: *a*) MamaDroid [3] abstracts each API on call graph with its family or package name to build a first-order Markov chain, and then uses pairwise transition probabilities as the feature vector; we use their source code for feature generation, and choose the package mode as it performs much better than the family mode; we implement the Random Forest classification with the configurations claimed in its paper. *b*) DeepMalDet [4] extracts an opcode sequence from each method and concatenates that of all classes to give a single sequence of opcodes representing a whole Android app; then it uses the convolutional neural network (CNN) for classification; we rerun their system using the source code published on GitHub.

2. https://www.virustotal.com

Table 2. Parameter Selection: Embedding dimensionality & Hop Number.

hop	dim	MACs	Params	Precision	Recall	Accuracy	F1-score
2	256	16008704	587010	**0.9690**	0.9581	0.9764	0.9635
	128	5104384	178818	0.9648	**0.9684**	**0.9782**	**0.9666**
	64	1827200	60738	0.9618	0.9609	0.9749	0.9614
	32	732352	23202	0.9575	0.9442	0.9683	0.9508
	16	320864	9810	0.9422	0.9405	0.9619	0.9413
3	128	30461184	178818	0.9697	0.9535	0.9752	0.9615

In the following sections (Section 4.2, 4.3, 4.4), we evaluate the performance of MsDroid via answering three main research questions:

- **RQ1**: how is the performance of MsDroid on large-scale dataset?
- **RQ2**: is MsDroid better than baseline techniques in terms of effectiveness and transferability?
- **RQ3**: can MsDroid's explanation mechanism provide useful insights into malware?

4.2 Detection Performance

In this section, we answer RQ1 by evaluating the detection and runtime performance on the whole dataset. We randomly select 80% apps from each sub-dataset (*Drebin*, *AMD*, *Old*, *New*) to form the training set, and test the detection performance on the rest of the apps. We comply to this training testing split rule in all our experiments.

Parameter Selection. The hyperparameters in our system are the hidden layer embedding dimensionality of the model and the number of hops during subgraph slicing. To determine the two hyperparameters, we do experiments on *Drebin* and *Old*. Firstly, we change the embedding dimensionality from 16, 32, 64, 128 to 256 with a fixed

Table 3. Test performance on the whole dataset.

	Precision	Recall	Accuracy	F1-score
MsDroid	0.9805	0.9678	0.9813	0.9741

hop number of 2. Their best testing results within 800 epochs are shown in Table 2, and we also count the model parameter numbers and calculate the average multiply-accumulate operations (MACs) for forward-propagation. It is worth noting that larger embedding can bring better performance until the dimensionality increases from 128 to 256. Then by fixing the embedding dimensionality at 128, we generate 3-hop and 4-hop behavior subgraphs since we use three graph convolutional layers in the classifier. As shown in the table, the performance of 3-hop drops by 0.3% in accuracy while the computation cost is 25M higher in MACs than 2-hop based subgraphs. For the 4-hop condition, the largest subgraph reaches a size of 38618 nodes and 229802 edges, and the learning process is interrupted due to the memory limitation. So we recommend 2 for hop number and 128 for embedding dimensionality in our system, and the rest experiments are all configured with this setting. Since smaller embedding is possible at a minor loss of effectiveness to be employed on mobile devices, we discuss the trade-off between the performance and computation in Appendix A.

Performance on Large-Scale Dataset. The model performance is evaluated with four metrics. 1) Precision: the proportion of identified malware that is actually malicious; 2) Recall: the proportion of real malware that is identified as malicious; 3) Accuracy: the proportion of all apps that are correctly classified; 4) F1-score: the harmonic mean of Precision and Recall which is calculated as $2 \times (Precision \times Recall)/(Precision + Recall)$. As in Table 3, on the dataset of 81,790 apps, MSDROID achieves 98.13% testing accuracy. Moreover, from Fig. 5, we observe a clear boost of accuracy as the amount of training data increases, and we can infer that better performance can be achieved if more data is fed to our model. The runtime performance of MSDROID is closely related to the amount and the size of subgraphs in the dataset. We use an NVIDIA Tesla V100 GPU and report the performance on training and testing stages as follows. It took roughly 16 seconds on average to train a batch of size 64, that is, 4.5 hours for an epoch when the whole dataset are used. The training process is rather long but acceptable in an offline setting. However, online testing works fine as 1) testing avoids the back-propagation process for computing gradients and updating weights, 2) batch normalization, one of the most time-consuming operations in forward-propagation, takes much less time in testing. Take a malware subgraph that approximately equal to the average size (35 nodes, 122 edges) for example, the prediction time is estimated at 14.67 milliseconds.

4.3 Baseline Comparison

In this section, we answer RQ2 by thoroughly comparing MSDROID with the baseline techniques. We train all models on the training set split from *Drebin* and *Old*, and then evaluate their performance on test-set, zero-day malware, newer apps, and obfuscated apps (Table 4).

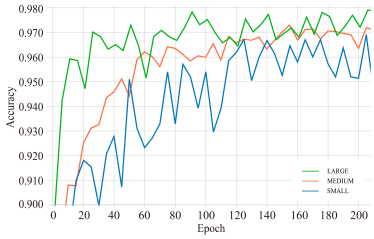


Fig. 5. Testing Accuracy curves within the first 200 epochs while the amount of training data varies. *SMALL* indicates that the training set is split from *Drebin* and *Old*, *MEDIUM* means half of the apps from *AMD* and *New* are used additionally, and *Large* refers to it when the whole dataset are used.

4.3.1 Effectiveness

We perform commonly used train-test split evaluation to study the effectiveness of all methods. From the "Test Set" column in Table 4, we can see that MSDROID outperforms all other methods in Accuracy, Recall, and F1-score. As MAMADROID is a very conservative model, it gets 97.28% Precision but at a high cost of Recall (79.52%), which means it can't identify malware in many cases. Furthermore, we see MSDROID-OPC also performs fairly well on test-set. The performance is merely 1.66% lower in F1-score than MSDROID, but 5.86% higher than the opcode-based method DEEPMALDET that fails to cooperate with structural information. MSDROID-PER and MSDROID-NON can identify most malware and have higher Recall than the two state-of-the-art methods, but they are more likely to mistake benign apps for malware. Thus, we can say that the graph-format combination features of the structure, opcode, and permission in MSDROID is effective for performance improvement.

4.3.2 Transferability

We also expect a good classifier to deal with unknown types of apps. Especially in the malware detection domain, we care about the following questions:

- Can zero-day threats be identified?
- How is the performance over-time?
- Can apps still be classified correctly if obfuscated?

Accordingly, we design three experiments to evaluate model transferability.

Zero-day Malware Identification. Zero-day malware is defined as previously unknown malware for which specific AV signatures are not yet available. Since our training malware set is *Drebin*, we gather apps in *AMD* which belong to unknown families in *Drebin* (e.g., SpyBubble, Mmarketpay, Nandrobox). To differentiate from evolved apps, we take those apps whose appearance time fall into *Drebin*'s date span. This makes our zero-day malware samples for the experiment a total number of 2562. The identification Accuracy are shown in "Zero-day" column in Table 4. Since all the tested samples here are malware, we can compare the Accuracy with Recall on test-set. All

Table 4. Comparison of MsDroid with five baseline techniques in four scenarios, including *test-set*, *zero-day malware*, *evolved apps*, and *obfuscated apps*. We use the metric AUT to measure the robustness to over-time app evolution. We apply three typical obfuscation techniques and will analyze their respective influence later in this paper.

Method	Test Set				Zero-day	Evolved				Obfuscated
	Accuracy	Precision	Recall	F1-score	Accuracy	AUT(a)	AUT(p)	AUT(r)	AUT(f1)	F1-score
MsDroid	**0.9782**	0.9648	**0.9684**	**0.9666**	0.8763	**0.7784**	0.8795	0.6246	**0.7189**	**0.9624**
MsDroid-per	0.9000	0.7875	0.9479	0.8603	0.5476	0.7375	0.7214	0.7238	0.7188	0.8895
MsDroid-opc	0.9674	0.9448	0.9553	0.9500	0.6909	0.6666	0.7554	0.3795	0.4622	0.9530
MsDroid-non	0.7097	0.6390	0.9639	0.7685	**0.8864**	0.5986	0.5605	**0.7888**	0.6538	0.7922
MamaDroid	0.9243	**0.9728**	0.7952	0.8751	0.4916	0.6350	**0.8852**	0.2562	0.3408	0.7422
DeepMalDet	0.9300	0.9015	0.8815	0.8914	0.8226	0.6336	0.6411	0.4532	0.5160	0.7012

methods drop in performance, especially for MsDroid-per (40.03%), MamaDroid (30.36%) and MsDroid-opc (26.44%). MsDroid achieves higher accuracy than other methods except for MsDroid-non, which is a bit ahead by 1.01%. Nevertheless, we know from the last experiment that MsDroid-non makes its decision aggressively, and we can infer that the structural feature holds an important place in MsDroid to prevent performance reduction in zero-day malware identification scenario.

Over-time performance. Android malware classifiers face a problem of aging as apps evolve to different implementation with similar semantics due to the update of Android version [10]. In this experiment, we first select samples from *AMD* whose appearance timestamp is in the date range of *New*, and divide them into four subsets for every eleven months similarly as dataset *New*. Then, to keep the spatial consistency, we randomly select 2,000 apps for each eleven-month-period from *AMD* and *New*, respectively. Thus, we get four time-ordered subsets of 4,000 apps. We use Area Under Time (AUT) to measure a classifier's robustness to time decay [9]:

$$AUT(N, f) = \frac{1}{N-1} \sum_{k=1}^{N-1} \frac{f(k+1) + f(k)}{2}, \qquad (9)$$

where N is the number of test slots, f is the performance metric (e.g., F1-score), and $f(k)$ is the performance metric evaluated at time k. The perfect classifier with no time decay has a AUT of 1.

We omit the constant N in the remainder of this paper since it is constant to 4 in the unit of 11 months. The AUTs of all models with f set to Accuracy, Precision, Recall, and F1-score are shown in Table 4. MsDroid still performs best in two key metrics AUT(a) and AUT(f1); MamaDroid is 0.57% better than MsDroid in Precision-related metric AUT(p) while the Recall-related one, i.e., AUT(r), drops sharply to only 25.62%. However, it is interesting to find MsDroid-per and MsDroid-non rank the second and the third in terms of AUT(f1), while MsDroid-opc performs much poorer, which is quite different from the results on test-set. Therefore, we can know that permission-related and structure-related features are rather stable over time than the opcode-related feature. Compared with the most robust state-of-the-art method towards time decay, i.e., DeepMalDet, MsDroid can improve AUT(f1) by more than 20.29%.

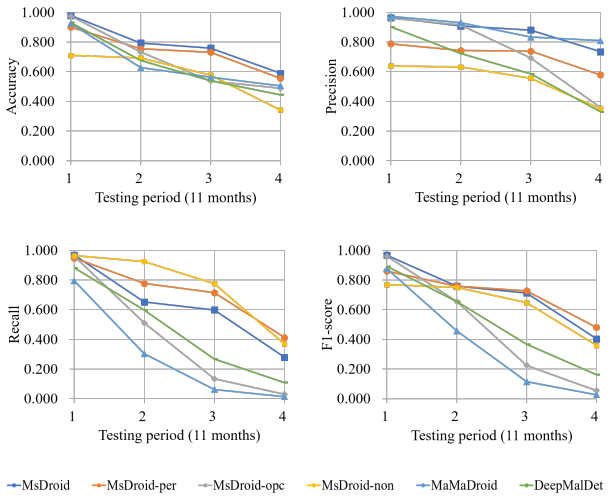


Fig. 6. The performance metrics over four test slots of total 44 months.

Furthermore, we plot how the four performance metrics degrade over the testing periods in Fig. 6. All metrics drop sharply especially in Recall and F1-score, and three models (MamaDroid, MsDroid-opc, and DeepMalDet) will become useless after two years with lower than 0.5 F1-score. Therefore, we suggest that a model should not be used if it is trained more than two years ago. In the worst case when no newer model is available, one should turn to MsDroid-per if higher Recall is wanted while turning to MsDroid if higher Precision is preferred.

Robustness to Code Obfuscation. Android app obfuscation is known as the process of modifying an APK so that it is no longer useful to a hacker while remaining fully functional. On the contrary, it could interfere with existing ML models [11]. In this experiment, we do not consider packing as the solutions to unpacking have been studied in prior works [45], [46], [47], but we do cover three basic types of obfuscation techniques that alters API name, bytecode, and calling relationship. Specifically, we use ObfuscapK [12] to obfuscate dataset *Drebin* and *Old*:

- *Renaming* changes package names and class names in bytecode together with the manifest file.
- *Code* inserts a "goto" instruction pointing to the end of a method and another "goto" pointing to the instruction after the first "goto", which modifies the method control-flow.

Table 5. The performance of MSDROID and the two state-of-the-art methods towards three obfuscation techniques.

Obfuscation	Method	Accuracy	Precision	Recall	F1-score
Renaming	Ours	**0.9742**	0.9661	**0.9857**	**0.9758**
	[3]	0.8399	0.9839	0.7039	0.8207
	[4]	0.8147	0.7425	0.9610	0.8377
Code	Ours	**0.9667**	0.9686	**0.9677**	**0.9681**
	[3]	0.9588	**0.9954**	0.9230	0.9579
	[4]	0.7609	0.6980	0.9178	0.7930
Reflection	Ours	**0.9407**	**0.9580**	**0.9288**	**0.9432**
	[3]	0.6272	0.8933	0.2989	0.4480
	[4]	0.6149	0.3978	0.5834	0.4730

- *Reflection* redirects an instruction with a method invocation to a custom method that invokes the original method using Reflection APIs.

The F1-score of all obfuscated dataset is shown in the last column of Table 4. Compared with test-set performance, MAMADROID drops by 13.29% and DEEPMALDET drops by 19.02%, while MSDROID and its three variants have stable performance. The detailed information about how MSDROID and the two state-of-the-art methods perform under the three types of obfuscations is given in Table 5. As it is expected, *Renaming* is of no use to MSDROID (and its variants) because the base class of each API is searched during permission mapping. MAMADROID wouldn't be affected by the bytecode related obfuscation method *Code*. This also explains why the corresponding performance is better than the test scores we give beforehand, as those samples in the old training set produce the same features after obfuscation, making the performance closer to what it is on training data. We have to mention that DEEPMALDET is actually more robust than MAMADROID in a situation where both methods are changed in features. We attribute the result to that DEEPMALDET is totally data-driven while MAMADROID relies on manually engineered statistics. We can see MAMADROID loses certain ability to identify malware as its Recall drops from 92.30% to 70.39% after *Renaming*, and the F1-score decreases by 14.32%; results get even worse when *Reflection* is applied, with 53.23% decrease. As for DEEPMALDET, obfuscation brings greater influence on Precision than Recall and it is interfered with by all the three techniques. *Renaming* reduces the performance because DEEPMALDET concatenates opcode sequences by class, *Code* causes more reduction as the opcode sequences inside every method are changed, and *Reflection* makes the harshest cut by modifying both function names and bytecodes. In contrast, MSDROID can still reach an F1-score higher than 94% in all cases.

To investigate into the obfuscation robustness of our method, we study the feature effectiveness towards different obfuscation techniques. Because MSDROID and its variants are not influenced by *Renaming*, we use their own performances in *Renaming* as the basis to calculate the percentage decrease in F1-score as in Table 6. For MSDROID, the decrease ratio in *Code* is merely 0.79% and it is smaller than MSDROID-OPC thanks to the permission-related feature. For *Reflection* that makes the two state-of-the-art methods nearly useless, the greatest percentage drop is around 8% as in MSDROID-OPC and the performance is even enhanced for MSDROID-NON.

Table 6. Percentage decrease in F1-score of MSDROID and its variants towards three obfuscation techniques.

Method	Renaming	Code	Reflection
MsDroid	0	-0.79%	-3.34%
MsDroid-per	0	0	-3.06%
MsDroid-opc	0	-1.31%	-7.77%
MsDroid-non	0	0	2.76%

Table 7. Influence of *Reflection* obfuscation on the average size of call graphs and behavior subgraphs.

Dataset	Call Graph		Behavior Subgraph	
	# Nodes	# Edges	# Nodes	# Edges
Drebin	4208 +66.26%	14821 +85.19%	36 +2.86%	126 +3.28%
Old	2766 +1.73%	10047 +19.48%	30 -3.23%	83 -24.55%

We further explain the outstanding *Reflection* robustness with Table 7. For the call graph, the average node and edge numbers increase for both malicious and benign apps after *Reflection*, especially for malware the percentage increase of average edge numbers reaches 85.19%. However, changes in the *behavior subgraph* are not so obvious, and the graph size of benign apps decreases oppositely, making them more distinct from malicious apps. Therefore, those models who extract features globally would be disturbed a lot, while MSDROID survives with the help of its subgraph method that captures suspicious behavioral semantics. Moreover, the structural information which is encoded and learned jointly with graph data makes it more capable of detecting anomalies instead of simply recognizing patterns.

In summary, MSDROID is the most effective method in the train-test split evaluation, and it is the only one that can always stand among the best towards zero-day malware, evolved apps, and differently obfuscated apps.

4.4 Explanation Analysis

4.4.1 Family Analysis

We divide the most commonly used permissions into seven categories (Table 8), including *Sensor*, *Change*, *System*, *Info*, *Location*, *Phone* and *Net*, to evaluate the first level explanation. The heatmap in Fig. 7 summarizes the results of behavioral categories given by MSDROID on *Genome* [48] dataset. *Genome* is a collection of 1260 Android malware samples in 49 different malware families, and it is all included in *Drebin*. As the heatmap indicates, a malware family may have several malicious behaviors, and its interdependence coefficient with these behavior categories also varies. We choose four different kinds of malware families to show the correlation between actual malicious behaviors and the result of MSDROID.

Geinimi is a typical Trojan family. It first connects to a C&C server and then executes the commands from

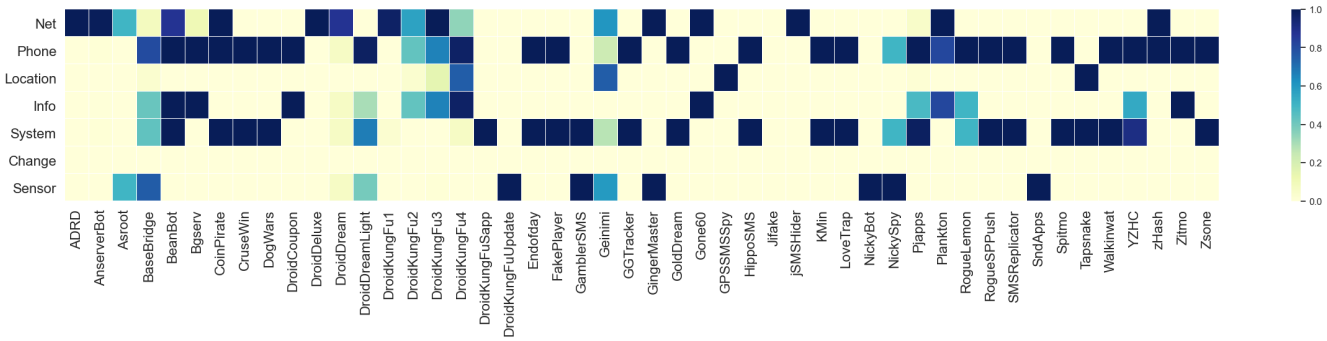


Fig. 7. Malicious families of *Genome* and the explanations of behavioral categories given by MSDROID.

the server, which include sound recording, SMS sending, phone call making, video recording, location collecting, etc. At the same time, the terminal also needs to hide and boot automatically. These actual malicious behaviors are consistent with our classification results, which mainly fall to *Sensor*, *System*, *Phone*, *Location* and *Net*.

HipoSMS is a malicious family which makes money by sending chargeable SMS. It conceals itself by deleting the notification messages sent by telecom carriers and blocking SMS alerts. Among these malicious behaviors, SMS sending, reading, and deleting are associated with category *Phone*, and hiding system notifications involves system-related permissions.

Bgserv is a worm virus family. It sends users' personal information, such as IMEI code, phone number, and system version, to a remote server. MSDROID has detected three mostly used malicious API of this family, which are consistent with their malicious behaviors that belong to *Info*, *Phone* and *Net*:

- `TelephonyManager.getDeviceId()`,
- `TelephonyManager.getLine1Number()`,
- `DefaultHttpClient.execute()`.

GPSSMS and Tapsnake monitor the user's real-time location and notify the remote server once the victim's location is changed. The API *requestLocationUpdates()* is detected malicious by MSDROID, and it is used for requesting real-time location of user. This result is in line with the family's behavior.

4.4.2 Case Study 1. Reflection Obfuscation

We use the *calling heat graph* (second level explanation) to further reason the robustness to *Reflection*. We take an app[3] in *Adsms* family of *Drebin* for example. As the snippets shown in Fig 9, a reflection array is added which includes all self-defined functions and their arguments after obfuscation. The app use `ApiReflection.obfuscate()` with a function index to call `SendMessage()` instead of calling it directly. We also illustrate the two behavior subgraphs of *SendTextMessage* in Fig. 10. As we can see in (a), many functions call *SendMessage* directly before obfuscation and this function is a self-defined function which uses *SendTextMessage* to send SMS. *MessageItem* is a Java bean that can provide receiver and content to *SendMessage*, so there exists deep connection between *SendMessage* and *MessageItem* in (a). After obfuscation, there only exists direct calls between constructor functions and multi-process functions. Despite this, MSDROID successfully captures the malicious behavior for following semantics:

- The structural features around constructor functions remain unchanged, for example, the API `MessageItem.<init>()`.
- For two key APIs that are related with malicious behaviors (i.e.`SMSObserver.onChange()`, `S2Cnl.run()`), their features represented by permission and opcode are relatively stable.

This also explains why MSDROID is more robust against reflection obfuscation than other methods.

4.4.3 Case Study 2. Latest App

The full-fledged spyware named *Covid19*[4] which is related to the coronavirus disease is discovered lately. It can collect victim's personal information include SMS message, call records, screen screenshot, etc. In the first level of the explanation mechanism, MSDROID gives `SmsManager.sendTextMessage()` as its suspiciously used API. By analysing the snippets around this API, we find the malicious behavior of sending C&C server dominated message.

Although malicious behaviors of the most recent app may not be detected completely with an aged model, we claim that MSDROID can still help with malware analysis through similarity search (i.e., the third level explanation). As Table 9 shows, the most similar implementations of `sendTextMessage()`,`getLastKnowLocation()`, `getRingtone()`, `isActiveNetworkMetered()` are apps belonging to malware families *Trida*, *RuMMS*, and *BankBot*. Besides `sendTextMessage()`, the other three APIs are used for checking network status, acquiring location, and playing audio commands, respectively.*Trida*, *RuMMS*, and *BankBot* are Torjan families and these four API have almost the same calling patterns as *Covid19*.

3. *Hash*: 54ece852176437e02ce2400e48c5545d32a5d69adee4a66a337ba98c67aea94e

4. *Hash*: 34952977658d3ef094a505f51de73c4902265b856ec90d164a34ae178474558f

```
...
1 so = new Socket();
2 so.connect(new InetSocketAddress(InetAddress.
    getByName(ipaddress).getHostAddress(), Integer
    .parseInt(port)), 500);
3 out = new DataOutputStream(so.getOutputStream())
    ;
4 v = new byte[LENGTH];
5 if(out.read(v) > 0 && v.has("sound") && v.getInt
    ("sound") > 0) {
6    Ringtone v0 = RingtoneManager.getRingtone(
    context,RingtoneManager.getDefaultUri(2));
7    v0.play();
     ...
}
```

Fig. 8. Similar behavioral snippets of API `getRingtone()` in *Covid19* and malware in *Triada*.

```
1 com.android.SendSMS.SendMessage(phoneNumber,
    content);
```

Before obfuscation

```
1 reflect.Method v1 = new Class[2];
2 v1[0] = String;
3 v1[1] = String;
4 obfuscatedMethods.add(SendSMS.getDeclaredMethod(
    "SendMessage", v1));
```

Reflection array

```
1 public static Object obfuscate(int p1, Object p2
    , Object[] p3)
2 {
3     try {
4         return ((reflect.Method)
    obfuscatedMethods.get(p1)).invoke(p2, p3);
5     } catch (int v1_4) {
6         ...
7         return 0;
8     }
9 }
```

Reflecation call

```
1 Object[] args = new Object[2];
2 args[0] = phoneNumber;
3 args[1] = content;
4 Object ret = new Object();
5 ApiReflection.obfuscate(funIndex, ret, args);
```

After obfuscation

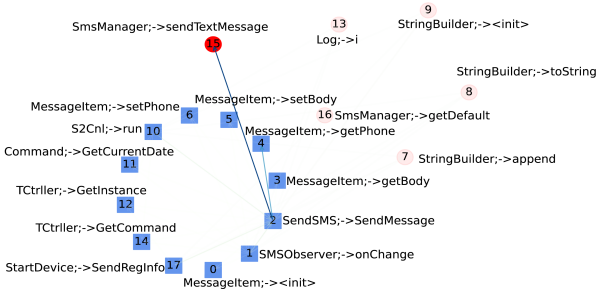
Fig. 9. Analysis of the change in snippets after the obfuscation.

For example, for `RingtoneManager.getRingtone()`, the most similiar family with this app is *Triada*. Their calling patterns are both as is shown in Fig. 8.

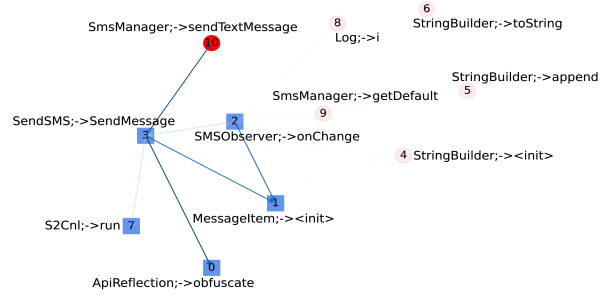
In *Triada* and *Covid19*, the client app firstly connects to a remote server and get a series of commands. Secondly, for each command that is checked as a sound control instruction, `RingtoneManager.getRingtone()` will be called. Similar calling patterns in existing datasets can speed up locating malicious code in new malware.

5 Discussion

In this section, we discuss some issues and limitations of MsDroid and points for future work.



(a) BEFORE OBFUSCATION



(b) AFTER OBFUSCATION

Fig. 10. *Calling Heat Graph* before and after reflection obfuscation.

API Semantics. We encodes permission and bytecode information in API node features, but the encoding method is relatively simple. As shown in our experiments, bytecode is the most unstable feature over time, and the frequency-based feature generation approach is to blame. Besides, the third-party library detection tool can be inaccurate. In the future, we can use embedding techniques to encode bytecode sequences for user-defined methods. For APIs in Android SDK or third-party libraries, we can leverage tools like APIGraph [10].

Subgraph Method. Our subgraph method is based on the pruned k-hop neighborhood of sensitive APIs now. As permission related feature can be enhanced, the subgraph method can also benefit from this. Moreover, we can seek more advanced Graph Neural Network techniques to discover subgraphs more automatically in the future.

Other Files. We don't use resource files (e.g., XMLs, asset files) for feature extraction because they are too easy to be obfuscated. Regardless of being likely to meet decompile failures, code in native code libraries can bring certain behavioral information of an app since some malware indeed hides part of their malicious code in those files. We believe we can generate BBS from *.lib* files similarly as *.dex* files, and we can achieve higher performance in a malware detection task by utilizing both of them.

Entities and Relations. The entity and relation types are obtained from static code analysis, but we can enlarge them with dynamic analysis. For example, the system call and network resource can be added into node types, while the invocation to system calls and the request for network resources can be treated as edges. Furthermore, what AiDroid does to include characters and relations in the Android ecosystem can also help.

Obfuscation Evaluation. We evaluate the robustness

towards obfuscation using three typical techniques provided by a popular tool. But in fact, obfuscation can be more complicated and packed apps should be studied in our future work. Since MSDROID-OPC has a large performance decline over time, we doubt it would act worse under more complicated bytecode-level obfuscation techniques. Also, obfuscation can be well designed manually [49] or through adversarial attacks [50]. For example, AVPASS [49] claims a bypass ratio of 95% for MAMADROID when all package names are modified.

6 RELATED WORK

6.1 Learning-based Android Malware Detection

Traditional learning-based approaches use features derived from manually designed patterns, such as suspicious intents, network addresses, API calls, system calls, and permissions [2], [6], [13], [51], [52]. Unlike these features, graph-based ones can provide more semantic information about app behaviors. Therefore, many researchers turn to leverage graphs [7], [53], especially the API call graphs from static analysis.

Various methods are proposed to catch malware fingerprints on API call graphs, which are largely divided into two classes. One class uses (sub)graph statistics [3]. For instance, DAPASA [27] generates one sensitive subgraph for each call graph and carefully designs five numeric features. DroidMiner [54] uses the presence of mined malware modalities by examing API sequences on the graph. The other class studies graph similarity metrics. DroidSIFT [28] adopts graph edit distance algorithm with assigned API weights and produces similarity-based features with known malware. Gascon et al. [55] embed call graphs using an explicit map inspired by neighborhood hash graph kernel (NHGK), and then train a linear SVM model. These methods rely on user-defined heuristics, and the similarity-based ones are heavily dependent on their malware corpus. Some recent works have been inspired by the success of deep neural networks in NLP domain. They first serialize the graphs as aforementioned works do, and learn on the sequences without further feature engineering. For example, Nix et al. [29] construct fixed sized matrices to investigate CNNs for classification. They encode each API as a one-hot vector, and each sequence is either padded or split if not equal to the hyperparameter of length. However, these approaches are known to be less explainable, and can not work efficiently when certain code obfuscations are applied. As a comparison, MSDROID first incorporates each API on graphs with knowledge from method opcodes and permissions to enhance the feature robustness. Furthermore, our approach preserves the whole structural information since automatic learning is directly performed on graphs rather than statistics or sequences. With the help of the subgraph-based design, MSDROID adopts recent developments in GNNs efficiently and make the model explainable.

Several studies have used graph representation learning for Android malware detection [8], [56]. Ye et al. [8] model different types of entities (i.e., app, API, device, signature, affiliation) and their relations as a heterogeneous graph. They propose a node representation algorithm and classify the nodes by a DNN classifier. Except for the *app-invoke-API* relation extracted from dynamic analysis, relations are defined within the Android ecosystem, while the IMEI-related relations evolve monitoring a large number of devices. Differently, MSDROID use static code analysis to extract relations, which include invocations between APIs, permissions required by APIs, and opcodes to implement APIs. Besides, our problem is defined as graph-level classification instead of nodel-level. A recent work detects Android malware by applying Graph Convolutional Networks (GCN) to system call graphs [56]. The system call graph is essentially modeled by sequences of manually selected system calls in dynamic running, so the node number is fixed and rather small. In contrast, MSDROID performs static analysis, which doesn't have the incomplete code coverage problem and incurs less overhead. Moreover, MSDROID captures control-flow information which is critical for manual analysis, and the explanation mechanism is capable of providing malware's inner workflow for security analysis.

6.2 Graph Learning for Code Analysis

Graph representation learning has been explored across various domains such as natural science [57], [58], social networks [22], [23] and knowledge graphs [59], [60]. Some previous works have investigated the applications in code analysis [41], [61], [62], [63], [64], [65]. In the binary code scenario, for example, Duan et al. [41] uses Text-associated DeepWalk (TADW) algorithm [66] for binary code differential analysis. Given two binary programs, they generate embeddings for each basic block on their merged interprocedural control flow graphs (ICFGs) and then perform basic-block matching by calculating pairwise similarity from embedding. In the source code scenario, Allamanis et al. [61] represent each program with a large graph on the top of the abstract syntax tree (AST), and then different edge types are added to model relationships between tokens. They further scale Gated Graph Neural Networks (GGNN) [21] and evaluate two tasks of filling the blanks (i.e., VarMisuse and VarNaming). Although performing better than other baseline models, their model gets less than 0.86 and 0.54 accuracy for each task, leaving a lot of room for improvement.

Differently, MSDROID represents a program with a set of graphs and targets at malware detection. On the one hand, malware detection focuses on high-level program behavior rather than low-level function similarity, and it is quite common for malware to be obfuscated. Thus, analysis into basic blocks would not only bring higher cost but also be too redundant to interfere with the learning process. On the other hand, the source code level information is not available in MSDROID since binary programs are often stripped and can be differently optimized. Instead of learning on a large API call graph, MSDROID converts the problem of APK labeling into classifying a set of much

smaller subgraphs, which avoid the common scalability problem of GNNs and thus can meet the high accuracy and efficiency demand of the malware detection system.

7 Conclusion

With MsDroid, we shed light on identifying malicious snippets without ground truth in Android malware detection. We transfer the detection problem into snippet classification tasks and define a customized loss function to enable effective training process. We design a robust snippet representation that incorporates multiple types of properties in a heterogeneous graph. The properties include API call graphs, opcode and permission, and GNN is used to capture structural semantics. An extensive experimental evaluation shows MsDroid's superior performance in effectiveness and robustness in various settings comparing with 5 baseline techniques. Moreover, we explore the effectiveness of program features towards different tasks to share insights on future improvements for different scenarios. To address the explainability problem of ML-based malware classifiers, we propose to reason a malware decision with suspicious permissions, API usage, and similar malwares. To ease the reproduction of our work, provide benchmark datasets, and inspire more research efforts to this problem, the code and dataset is open source on GitHub.

Acknowledgment

Zhan Qin is supported in part by the National Key Research and Development Program of China under Grant 2020AAA0107705 and the National Natural Science Foundation of China under Grant U20A20178. Kui Ren is supported by the National Key Research and Development Program of China under Grant 2020AAA0107705.

References

[1] A. Neville, "Recent cyber-attacks and the eu's cybersecurity strategy for the digital decade," European Parliamentary Research Service, June 2021.

[2] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, K. Rieck, and C. Siemens, "Drebin: Effective and explainable detection of android malware in your pocket." in *Network and Distributed System Symposium (NDSS)*, vol. 14, 2014, pp. 23–26.

[3] E. Mariconti, L. Onwuzurike, P. Andriotis, E. De Cristofaro, G. Ross, and G. Stringhini, "Mamadroid: Detecting android malware by building markov chains of behavioral models," *arXiv preprint arXiv:1612.04433*, 2016.

[4] N. McLaughlin, J. Martinez del Rincon, B. Kang, S. Yerima, P. Miller, S. Sezer, Y. Safaeisemnani, E. Trickel, Z. Zhao, A. Doupé, and G. Joon Ahn, "Deep android malware detection," in *Proceeding of the ACM Conference on Data and Applications Security and Privacy (CODASPY) 2017*. Association for Computing Machinery (ACM), 12 2016.

[5] T. Hsien-De Huang and H.-Y. Kao, "R2-d2: color-inspired convolutional neural network (cnn)-based android malware detections," in *2018 IEEE International Conference on Big Data (Big Data)*. IEEE, 2018, pp. 2633–2642.

[6] T. Kim, B. Kang, M. Rho, S. Sezer, and E. G. Im, "A multimodal deep learning method for android malware detection using various features," *IEEE Transactions on Information Forensics and Security*, vol. 14, no. 3, pp. 773–788, 2018.

[7] S. Hou, Y. Ye, Y. Song, and M. Abdulhayoglu, "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*, 2017, pp. 1507–1515.

[8] Y. Ye, S. Hou, L. Chen, J. Lei, W. Wan, J. Wang, Q. Xiong, and F. Shao, "Out-of-sample node representation learning for heterogeneous graph in real-time android malware detection." in *2019 International joint conference on neural networks (IJCAI)*, 2019, pp. 4150–4156.

[9] F. Pendlebury, F. Pierazzi, R. Jordaney, J. Kinder, and L. Cavallaro, "Tesseract: Eliminating experimental bias in malware classification across space and time," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 729–746.

[10] X. Zhang, Y. Zhang, M. Zhong, D. Ding, Y. Cao, Y. Zhang, M. Zhang, and M. Yang, "Enhancing state-of-the-art classifiers with api semantics to detect evolved android malware," *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020.

[11] X. Chen, C. Li, D. Wang, S. Wen, J. Zhang, S. Nepal, Y. Xiang, and K. Ren, "Android hiv: A study of repackaging malware for evading machine-learning detection," *IEEE Transactions on Information Forensics and Security*, vol. 15, pp. 987–1001, 2019.

[12] S. Aonzo, G. C. Georgiu, L. Verderame, and A. Merlo, "Obfuscapk: An open-source black-box obfuscation tool for android apps," *SoftwareX*, vol. 11, p. 100403, 2020. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S2352711019302791

[13] H. Alshahrani, H. Mansourt, S. Thorn, A. Alshehri, A. Alzahrani, and H. Fu, "Ddefender: Android application threat detection using static and dynamic analysis," in *2018 IEEE International Conference on Consumer Electronics (ICCE)*. IEEE, 2018, pp. 1–6.

[14] E. B. Karbab, M. Debbabi, A. Derhab, and D. Mouheb, "Maldozer: Automatic framework for android malware detection using deep learning," *Digital Investigation*, vol. 24, pp. S48–S59, 2018.

[15] K. Xu, Y. Li, R. Deng, K. Chen, and J. Xu, "Droidevolver: Self-evolving android malware detection system," in *2019 IEEE European Symposium on Security and Privacy (EuroS P)*, 2019, pp. 47–62.

[16] Q. Li and X. Li, "Android malware detection based on static analysis of characteristic tree," in *2015 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery*. IEEE, 2015, pp. 84–91.

[17] M. Fan, W. Wei, X. Xie, Y. Liu, X. Guan, and T. Liu, "Can we trust your explanations? sanity checks for interpreters in android malware analysis," *IEEE Transactions on Information Forensics and Security*, vol. 16, pp. 838–853, 2020.

[18] W. Guo, D. Mu, J. Xu, P. Su, G. Wang, and X. Xing, "Lemna: Explaining deep learning based security applications," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 364–379.

[19] Z. Wang, T. Chen, J. Ren, W. Yu, H. Cheng, and L. Lin, "Deep reasoning with knowledge graph for social relationship understanding," *arXiv preprint arXiv:1807.00504*, 2018.

[20] M. Narasimhan, S. Lazebnik, and A. Schwing, "Out of the box: Reasoning with graph convolution nets for factual visual question answering," in *Advances in neural information processing systems*, 2018, pp. 2654–2665.

[21] Y. Li, D. Tarlow, M. Brockschmidt, and R. Zemel, "Gated graph sequence neural networks," *arXiv preprint arXiv:1511.05493*, 2015.

[22] W. Hamilton, Z. Ying, and J. Leskovec, "Inductive representation learning on large graphs," in *Advances in neural information processing systems*, 2017, pp. 1024–1034.

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

[24] M. Defferrard, X. Bresson, and P. Vandergheynst, "Convolutional neural networks on graphs with fast localized spectral filtering," in *Advances in neural information processing systems*, 2016, pp. 3844–3852.

[25] K. Xu, W. Hu, J. Leskovec, and S. Jegelka, "How powerful are graph neural networks?" *arXiv preprint arXiv:1810.00826*, 2018.

[26] S. Hahn, M. Protsenko, and T. Müller, "Comparative evaluation of machine learning-based malware detection on android." in *Sicherheit 2016 - Sicherheit, Schutz und Zuverlässigkeit*. Gesellschaft für Informatik e.V., 2016, pp. 79–88.

[27] M. Fan, J. Liu, W. Wang, H. Li, Z. Tian, and T. Liu, "Dapasa: detecting android piggybacked apps through sensitive subgraph analysis," *IEEE Transactions on Information Forensics and Security*, vol. 12, no. 8, pp. 1772–1785, 2017.

[28] M. Zhang, Y. Duan, H. Yin, and Z. Zhao, "Semantics-aware android malware classification using weighted contextual api dependency graphs," in *Proceedings of the 2014 ACM SIGSAC conference on computer and communications security*, 2014, pp. 1105–1116.

[29] R. Nix and J. Zhang, "Classification of android apps and malware using deep neural networks," in *2017 International joint conference on neural networks (IJCNN)*. IEEE, 2017, pp. 1871–1878.

[30] F. Wei, Y. Li, S. Roy, X. Ou, and W. Zhou, "Deep ground truth analysis of current android malware," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2017, pp. 252–276.

[31] M. Sikorski and A. Honig, *Practical malware analysis: the hands-on guide to dissecting malicious software*. no starch press, 2012.

[32] J. Peppers, "Creating a malware analysis lab and basic malware analysis," *Creative Components*, vol. 92, 2018.

[33] A. Desnos, "Androguard," https://github.com/androguard/androguard 2020.

[34] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie, "Pscout: analyzing the android permission specification," in *Proceedings of the 2012 ACM conference on Computer and communications security*, 2012, pp. 217–228.

[35] M. Backes, S. Bugiel, E. Derr, P. McDaniel, D. Octeau, and S. Weisgerber, "On demystifying the android application framework: Re-visiting android permission specification analysis," in *25th USENIX security symposium (USENIXa Security 16)*, 2016, pp. 1101–1118.

[36] Z. Ma, H. Wang, Y. Guo, and X. Chen, "Libradar: fast and accurate detection of third-party libraries in android apps," in *Proceedings of the 38th international conference on software engineering companion*, 2016, pp. 653–656.

[37] X. Zhan, L. Fan, T. Liu, S. Chen, L. Li, H. Wang, Y. Xu, X. Luo, and Y. Liu, "Automated third-party library detection for android applications: Are we there yet?" in *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2020, pp. 919–930.

[38] Y. Zhang, Y. Xiong, X. Kong, S. Li, J. Mi, and Y. Zhu, "Deep collective classification in heterogeneous information networks," in *Proceedings of the 2018 World Wide Web Conference*, 2018, pp. 399–408.

[39] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Lio, and Y. Bengio, "Graph attention networks," *arXiv preprint arXiv:1710.10903*, 2017.

[40] Z. Ying, D. Bourgeois, J. You, M. Zitnik, and J. Leskovec, "Gnnexplainer: Generating explanations for graph neural networks," in *Advances in neural information processing systems*, 2019, pp. 9244–9255.

[41] Y. Duan, X. Li, J. Wang, and H. Yin, "Deepbindiff: Learning program-wide code representations for binary diffing," in *Network and Distributed System Symposium (NDSS)*, 2020.

[42] S. H. Ding, B. C. Fung, and P. Charland, "Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 472–489.

[43] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon, "Androzoo: Collecting millions of android apps for the research community," in *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*. IEEE, 2016, pp. 468–471.

[44] R. He, H. Wang, P. Xia, L. Wang, Y. Li, L. Wu, Y. Zhou, X. Luo, Y. Guo, and G. Xu, "Beyond the virus: A first look at coronavirus-themed mobile malware," *arXiv preprint arXiv:2005.14619*, 2020.

[45] W. Yang, Y. Zhang, J. Li, J. Shu, B. Li, W. Hu, and D. Gu, "Appspear: Bytecode decrypting and dex reassembling for packed android malware," in *International Symposium on Recent Advances in Intrusion Detection*. Springer, 2015, pp. 359–381.

[46] Y. Duan, M. Zhang, A. V. Bhaskar, H. Yin, X. Pan, T. Li, X. Wang, and X. Wang, "Things you may not know about android (un) packers: A systematic study based on whole-system emulation." in *NDSS*, 2018.

[47] Y. Zhang, X. Luo, and H. Yin, "Dexhunter: toward extracting hidden code from packed android applications," in *European Symposium on Research in Computer Security*. Springer, 2015, pp. 293–311.

[48] Y. Zhou and X. Jiang, "Dissecting android malware: Characterization and evolution," in *2012 IEEE symposium on security and privacy*. IEEE, 2012, pp. 95–109.

[49] J. Jung, C. Jeon, M. Wolotsky, I. Yun, and T. Kim, "AVPASS: Leaking and Bypassing Antivirus Detection Model Automatically," Black Hat USA Briefings, 2017.

[50] S. Hou, Y. Fan, Y. Zhang, Y. Ye, J. Lei, W. Wan, J. Wang, Q. Xiong, and F. Shao, "αcyber: Enhancing robustness of android malware detection system against adversarial attacks on heterogeneous graph based model," in *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*, 2019, pp. 609–618.

[51] Z. Yuan, Y. Lu, and Y. Xue, "Droiddetector: android malware characterization and detection using deep learning," *Tsinghua Science and Technology*, vol. 21, no. 1, pp. 114–123, 2016.

[52] L. Xu, D. Zhang, N. Jayasena, and J. Cavazos, "Hadm: Hybrid analysis for detection of malware," in *Proceedings of SAI Intelligent Systems Conference (IntelliSys)*. Springer, 2016, pp. 702–724.

[53] S. Hou, A. Saas, L. Chen, and Y. Ye, "Deep4maldroid: A deep learning framework for android malware detection based on linux kernel system call graphs," in *2016 IEEE/WIC/ACM International Conference on Web Intelligence Workshops (WIW)*. IEEE, 2016, pp. 104–111.

[54] C. Yang, Z. Xu, G. Gu, V. Yegneswaran, and P. Porras, "Droidminer: Automated mining and characterization of fine-grained malicious behaviors in android applications," in *European symposium on research in computer security*. Springer, 2014, pp. 163–182.

[55] H. Gascon, F. Yamaguchi, D. Arp, and K. Rieck, "Structural detection of android malware using embedded call graphs," in *Proceedings of the 2013 ACM workshop on Artificial intelligence and security*, 2013, pp. 45–54.

[56] T. S. John, T. Thomas, and S. Emmanuel, "Graph convolutional networks for android malware detection with system call graphs," in *2020 Third ISEA Conference on Security and Privacy (ISEA-ISAP)*. IEEE, 2020, pp. 162–170.

[57] A. Sanchez-Gonzalez, N. Heess, J. T. Springenberg, J. Merel, M. Riedmiller, R. Hadsell, and P. Battaglia, "Graph networks as learnable physics engines for inference and control," *arXiv preprint arXiv:1806.01242*, 2018.

[58] A. Fout, J. Byrd, B. Shariat, and A. Ben-Hur, "Protein interface prediction using graph convolutional networks," in *Advances in neural information processing systems*, 2017, pp. 6530–6539.

[59] T. Hamaguchi, H. Oiwa, M. Shimbo, and Y. Matsumoto, "Knowledge transfer for out-of-knowledge-base entities: A graph neural network approach," *arXiv preprint arXiv:1706.05674*, 2017.

[60] M. Kampffmeyer, Y. Chen, X. Liang, H. Wang, Y. Zhang, and E. P. Xing, "Rethinking knowledge graph propagation for zero-shot learning," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2019, pp. 11 487–11 496.

[61] M. Allamanis, M. Brockschmidt, and M. Khademi, "Learning to represent programs with graphs," *arXiv preprint arXiv:1711.00740*, 2017.

[62] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," in *Advances in Neural Information Processing Systems*, 2019, pp. 10 197–10 207.

[63] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song, "Neural network-based graph embedding for cross-platform binary code similarity detection," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 363–376.

[64] L. Massarelli, G. A. Di Luna, F. Petroni, L. Querzoni, and R. Baldoni, "Investigating graph embedding neural networks with unsupervised features extraction for binary analysis," in

Proceedings of the 2nd Workshop on Binary Analysis Research (BAR), 2019.

[65] Z. Yu, W. Zheng, J. Wang, Q. Tang, S. Nie, and S. Wu, "Codecmr: Cross-modal retrieval for function-level binary source code matching," *Advances in Neural Information Processing Systems*, vol. 33, pp. 3872–3883, 2020.

[66] C. Yang, Z. Liu, D. Zhao, M. Sun, and E. Y. Chang, "Network representation learning with rich text information." in *International Joint Conferences on Artificial Intelligence (IJCAI)*, vol. 2015, 2015, pp. 2111–2117.

Yiling He received the BE degree in information security from Beijing University of Posts and Telecommunications in 2019. She is currently working toward the doctoral degree in cyberspace security at Zhejiang University. Her current research interests lie in the robustness and explainability of AI-assisted security applications, including malware detection, vulnerability discovery, and exploit generation.

Yiping Liu received the BE degree in information security from Beijing University of Posts and Telecommunications, in 2019. She is currently working toward master's degree in Cyberspace Security at Beijing University of Posts and Telecommunications. Her research interests include mobile security, binary security and automatic vulnerability discovery.

Lei Wu joined Zhejiang University in September 2019 as an assistant professor. His research interest lies mainly in security areas, including mobile security, program/binary analysis, vulnerability detection and exploitation, and blockchain security. He obtained his Ph.D. degree from North Carolina State University (NCSU) in June 2015. Prior to joining Zhejiang University as a faculty, he worked as a senior security researcher at Qihoo 360 from 2015 to 2017. After that, he became a co-founder of a startup company named PeckShield Inc. and worked as VP of Engineering from 2017 to 2019.

Ziqi Yang is an Assistant Professor (ZJU100 Young Professor), with both the Institute of Cyber Security Research and the College of Computer Science and Technology at Zhejiang University, China. Prior to Zhejiang University, he was a Research Fellow at National University of Singapore. He received a Ph.D degree in Computer Science from National University of Singapore. He is interested in cybersecurity and machine learning with a recent focus on the intersections between security, privacy, and machine learning.

Kui Ren is a Professor and Associate Dean of College of Computer Science and Technology at Zhejiang University, where he also directs the Institute of Cyber Science and Technology. Before that, he was SUNY Empire Innovation Professor at State University of New York at Buffalo. He received his PhD degree from Worcester Polytechnic Institute. Kui's current research interests include Data Security, IoT Security, AI Security, and Privacy. He received IEEE CISTC Technical Recognition Award in 2017, SUNY Chancellor's Research Excellence Award in 2017, Sigma Xi/IIT Research Excellence Award in 2012, and NSF CAREER Award in 2011.

Zhan Qin is the 100-Talents Young Professor in the Institute of CyberSpace Research in Zhejiang University since 2019. After graduating from the Department of Computer Science and Engineering at the State University of New York at Buffalo, he joined the University of Texas at San Antonio as Assistant Professor in 2017. His research enables AI security and Data Security. Specifically, he focuses on adversarial attack and defense to deep learning model, privacy-preserving data collection, computation and publication. He also explores and develops novel security sensitive algorithms and protocols for computation and communication on IoT devices.

APPENDIX A
COMPUTATION PERFORMANCE TRADE-OFF

We study the computation performance trade-off of embedding dimensionality in 2-hop situation here. The MACs for 2-hop behavior subgraphs in Table 2 are calculated with the average subgraph size of *Drebin*, i.e., 35 nodes and 122 edges (the 3-hop MACs are calculated with the average node number of 210 and edge number of 1383). Fig. 11 is a scatter-gram of the five models where the x-axis represents forward propagation MACs, and the y-axis is the testing F1-score. With the dashed trend line, we suggest that a 64-dimensional embedding also be a good choice if less computation is required.

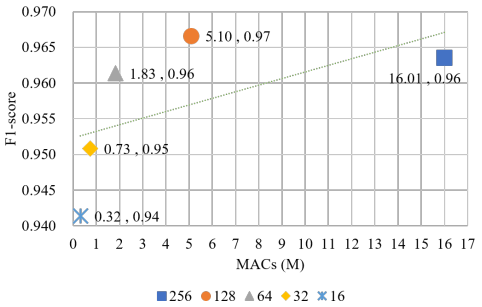


Fig. 11. MACs and F1-score trade-off of embedding dimensionality.

APPENDIX B
BEHAVIORAL CATEGORIES

The first level explanation maps the permission of detected suspicious API usages to behavioral semantics. The defined mapping is shown in Table 8.

Table 8. The behavioral categories of sensitive permissions.

Category	Discription	Typical permissions
Sensor	Permissions related with hardware which includes audio, camera, bluetooth and etc.	BLUETOOTH, RECORD-AUDIO
Change	Permissions that can change system statement, for example system settings and network change.	WRITE-SETTINGS, CHANGE-WIFI-STATE
System	Permissions related with operating system which includes clock, boot statement, package installation.	RESTART-PACKAGES, RECEIVE-BOOT-COMPLETED
Info	Permissions about personal information for example user account and history bookmarks.	GET-ACCOUNTS, READ-HISTORY-BOOKMARKS
Location	Permissions about GPS location.	ACCESS-COARSE-LOCATION, ACCESS-FINE-LOCATION
Phone	Permission related with sms message and phone state.	READ-PHONE-STATE, SEND-SMS
Net	Permissions about network state.	ACCESS-NETWORK-STATE, INTERNET

APPENDIX C
SIMILARITY SEARCH OF *Covid19*

For the second case study of the app *Covid19*, we analyze the third level explanation results, and the similarity scores with the most similar malware families (adware excluded) are shown in Table 9.

Table 9. Families of the malware in *AMD* who have the most similar behaviors around sensitives APIs with *Covid19*.

API	Similarity	Family
AudioManager.setStreamVolume(I,I,I)V	3.5746	BankBot
NotificationManager.notify(String,I,Notification)V	2.8091	BankBot
SmsManager.sendTextMessage(String,String,String,PendingIntent,PendingIntent)V	1.8395	BankBot
ConnectivityManager.isActiveNetworkMetered()Z	1.4767	Triada
AccountManager.getAccounts()Account[]	0.0165	Airpush
RingtoneManager.getRingtone(Context,Uri)Ringtone;	0.0122	Triada
LocationManager.getLastKnownLocation(String)Location;	0.0109	RuMMS
MediaRecorder.setAudioSource(I)V	0.0089	Andup
Ringtone.play()V	0.0073	Gumen
Ringtone.stop()V	0.0060	Dowgin

APPENDIX D
EXAMPLE CALLING HEAT GRAPH

The *calling heat graph* in Fig. 4 comes from an example malware [5] in dataset *Drebin*, The mappings of the node IDs and API names in the graph is given below.

```
0  Landroid/support/v4/app/FragmentTransaction;->
       add(I Landroid/support/v4/app/Fragment;)
       Landroid/support/v4/app/FragmentTransaction
       ;,
1  Lcom/zimperium/zanti/Anti;->a(Lcom/zimperium/
       zanti/Anti$TargetListItem; Z Ljava/lang/
       String;)V,
2  Landroid/support/v4/app/FragmentTransaction;->
       addToBackStack(Ljava/lang/String;)Landroid/
       support/v4/app/FragmentTransaction;,
3  Landroid/support/v4/app/FragmentTransaction;->
       commit()I,
4  Landroid/support/v4/app/FragmentTransaction;->
       setCustomAnimations(I I I I)Landroid/
       support/v4/app/FragmentTransaction;,
5  Ljava/util/ArrayList;-><init>()V,
6  Landroid/support/v4/app/FragmentManager;->
       beginTransaction()Landroid/support/v4/app/
       FragmentTransaction;,
7  Ljava/lang/Integer;->valueOf(I)Ljava/lang/
       Integer;,
8  Ljava/util/Iterator;->hasNext()Z,
9  Ljava/util/Iterator;->next()Ljava/lang/Object;,
10 Ljava/util/List;->add(Ljava/lang/Object;)Z,
11 Ljava/util/List;->size()I,
12 Ljava/lang/String;->contains(Ljava/lang/
       CharSequence;)Z,
```

5. *Hash*: c5798860ae9ae80c3cc5c68657e32ed208d8e956e26d7dd8f9d0c84f49bf9d34

```
13 Ljava/lang/String;->equalsIgnoreCase(Ljava/lang
       /String;)Z,
14 Lcom/viewpagerindicator/f;->b()Landroid/support
       /v4/app/Fragment;,
15 Ljava/util/List;->iterator()Ljava/util/Iterator
       ;,
16 Lcom/zimperium/zanti/MenuOptions/
       PluginMenuOption;-><init>(Lcom/zimperium/
       zanti/plugins/AntiPlugin; Ljava/lang/String
       ;)V,
17 Landroid/net/NetworkInfo;->isConnected()Z,
18 Lcom/zimperium/zanti/MenuOptions/
       ConnectMenuOption;-><init>(Ljava/lang/
       String;)V,
19 Lcom/zimperium/zanti/
       Helpers$MainMenuOptionThatShowsTargetSelect
       ;->f()Z,
20 Lcom/zimperium/zanti/Anti;->getSystemService(
       Ljava/lang/String;)Ljava/lang/Object;,
21 Lcom/zimperium/zanti/ao;-><init>()V,
22 Landroid/net/wifi/WifiManager;->isWifiEnabled()
       Z,
23 Lcom/zimperium/zanti/MenuOptions/
       EnableWifiMenuOption;-><init>(Z)V,
24 Ljava/util/List;->contains(Ljava/lang/Object;)Z
       ,
25 Lcom/zimperium/zanti/MenuOptions/
       RequiresRootMenuOption;-><init>(Lcom/
       zimperium/zanti/plugins/AntiPlugin; Ljava/
       lang/String;)V,
26 Landroid/net/ConnectivityManager;->
       getNetworkInfo(I)Landroid/net/NetworkInfo;,
27 Lcom/zimperium/zanti/Anti;->
       getSupportFragmentManager()Landroid/support
       /v4/app/FragmentManager;,
28 Lcom/zimperium/zanti/MenuOptions/ScanMenuOption
       ;-><init>(Ljava/lang/String;)V,
29 Lcom/zimperium/zanti/mainpage/r;-><init>(
       Landroid/content/Context; Ljava/util/List;
       Lcom/zimperium/zanti/Anti$TargetListItem;)V
       ,
30 Lcom/zimperium/zanti/MenuOptions/
       TraceLocationMenuOption;-><init>()V,
31 Lcom/zimperium/zanti/mainpage/q;->onClick(
       Landroid/view/View;)V,
32 Lcom/zimperium/zanti/WifiMonitor;->
       runOnUiThread(Ljava/lang/Runnable;)V,
33 Lcom/zimperium/zanti/dm;->run()V,
34 Landroid/net/wifi/WifiManager;->startScan()Z,
35 Lcom/zimperium/zanti/dn;-><init>(Lcom/zimperium
       /zanti/dm;)V,
36 Lnecom/zimperium/zanti/do;-><init>(Lcom/
       zimperium/zanti/dm;)V
```