Heterogeneous Graph Convolutional Networks for Android Malware Detection using Callback-Aware Caller-Callee Graphs

Vinayaka K V [1] and Jaidhar C D [2]

[1]National Institute of Technology Karnataka
[2]Affiliation not available

October 30, 2023

IEEE Access
Multidisciplinary : Rapid Review : Open Access Journal

Date of publication xxxx 00, 0000, date of current version xxxx 00, 0000.

Digital Object Identifier 10.1109/ACCESS.2017.DOI

Heterogeneous Graph Convolutional Networks for Android Malware Detection using Callback-Aware Caller-Callee Graphs

VINAYAKA K V [1], JAIDHAR C D [2]
[1] National Institute Technology Karnataka, Surathkal , Srinivasanagar , Mangaluru, Karnataka, India - 575025 (email: vinayakakv.193it001@nitk.edu.in)
[2] National Institute Technology Karnataka, Surathkal , Srinivasanagar , Mangaluru, Karnataka, India - 575025 (email: jaidharcd@nitk.edu.in)

Corresponding author:Vinayaka K V (email: vinayakakv.193it001@nitk.edu.in).

ABSTRACT The popularity of the Android Operating System in the smartphone market has given rise to lots of Android malware. To accurately detect these malware, many of the existing works use machine learning and deep learning-based methods, in which feature extraction methods were used to extract fixed-size feature vectors using the files present inside the Android Application Package (APK). Recently, Graph Convolutional Network (GCN) based methods applied on the Function Call Graph (FCG) extracted from the APK are gaining momentum in Android malware detection, as GCNs are effective at learning tasks on variable-sized graphs such as FCG, and FCG sufficiently captures the structure and behaviour of an APK. However, the FCG lacks information about callback methods as the Android Application Programming Interface (API) is event-driven. This paper proposes enhancing the FCG to eFCG (enhanced-FCG) using the callback information extracted using Android Framework Space Analysis to overcome this limitation. Further, we add permission - API method relationships to the eFCG. The eFCG is reduced using node contraction based on the classes to get R-eFCG (Reduced eFCG) to improve the generalisation ability of the Android malware detection model. The eFCG and R-eFCG are then given as the inputs to the Heterogeneous GCN models to determine whether the APK file from which they are extracted is malicious or not. To test the effectiveness of eFCG and R-eFCG, we conducted an ablation study by removing their various components. To determine the optimal neighbourhood size for GCN, we experimented with a varying number of GCN layers and found that the Android malware detection model using R-eFCG with all its components with four convolution layers achieved maximum accuracy of 96.28%.

INDEX TERMS Android, Computer security, Graph Convolutional Networks, Machine Learning, Program Analysis

I. INTRODUCTION

Android is a popular smartphone Operating System that powers around 70% of the smartphones and tablets worldwide [33]. Its popularity has long attracted a large amount of malware into its ecosystem [25] [31], threatening the privacy and security of its users. Three analysis techniques are prevalent to detect Android malware – static, dynamic and hybrid analysis [29]. In static analysis, features are extracted from the Android Application Package (APK) file without executing it. The dynamic analysis executes the APK inside a sandbox and extracts run-time features. The hybrid analysis is a combination of the above. Although obfuscation techniques can hinder static analysis [38], it is substantially faster than its counterparts.

The APK file provides several features to perform static analysis. The features such as permissions and intents can be extracted from the manifest file, which are the indicators of the behaviour of the Android application (app) [6] [18] [34] [1]. Apart from them, features such as sensitive Application Programming Interface (API) calls [1], API call graph [12] and Function Call Graph (FCG) [23] [39] [16] can be extracted from the `Dalvik Executable (dex) code`. Out of these features, FCG captures the structure of interactions between the methods of the app. The FCG is a directed graph with methods in the `dex code` as its nodes; its edges represent Caller-Callee relationships between the methods. If every node of the FCG is assigned features that represent its behaviour, it can capture the behaviour of an app as a whole [35].

The methods contained in the `dex code` can be *internal* or *external* depending on whether their implementation is contained in the `dex code` or not [35]. In general, the API methods (the *Framework Space*, $\mathcal{F}$) are external, while User-defined methods (the *Application Space*, $\mathcal{A}$) are internal. As FCGs are extracted entirely using the information present in the `dex code`, interactions from the Framework Space to the Application Space cannot be captured [10]. This information is crucial as the Android API is heavily event-driven. In Android event architecture, event handlers are implemented as Application Space *callback handlers*, which are the children of Framework Space *callback methods*. The Framework Space is made aware of callback handlers using *registration methods*, which are also a part of the Framework Space [10]. FCG is unable to capture the relationship between registration methods and callback handlers. The Framework Space has to be analysed to include such relationships, and its results have to be used while constructing the FCG [10] [13].

Graph Convolutional Networks (GCNs) [20] have become a natural choice to perform deep learning on graphs because of their flexibility [43]. GCNs process graphs by aggregating neighbourhood information, updating a node's features based on it and fine-tuning its learnable parameters for a particular task. An n-layer GCN aggregates features into a node from its n-hop neighbourhood. A global pooling operation on the graph is used to obtain the feature vector representing the graph. This vector can then be used for downstream tasks such as classification.

In this work, we analyse Framework Space code to extract Registration-Callback map motivated by the approach of [10]. We also consider the mapping of permissions required by an API method from [7]. This information is utilised while analysing APKs to convert FCGs extracted from them into *enhanced-FCGs* (eFCGs). The *reduced-eFCG* (R-eFCG) is then obtained by contracting nodes of eFCG in an approach similar to MaMaDroid's [26]. Separate heterogeneous GCN models are then trained on eFCG and R-eFCG to evaluate their effectiveness.

We answer the following research questions in this paper:

1. Which components of eFCG and R-eFCGs are essential in Android malware detection using heterogeneous GCNs?
2. Can R-eFCGs achieve better generalisation in terms of Android malware detection rate than eFCGs?
3. What is the optimal neighbourhood size n for GCNs to detect Android malware using eFCG and R-eFCGs?

To answer these research questions, we experiment with different components of eFCG and R-eFCGs to determine their contribution to the performance of the Android malware detection model. We also train separate models on eFCGs and R-eFCGs to access their generalisation ability. To determine the choice of optimal neighbourhood, we conducted a set of experiments by varying the number of GCN layers. As a result of these experiments, we obtained a maximum accuracy of 96.25% with R-eFCGs with all components and four GCN layers.

The key contributions of the present work are as follows:

1. We define eFCG and R-eFCG, containing the callback information and permission mappings along with the Caller-Callee information, and provide algorithms to obtain the same.
2. We conducted an ablation study to find essential components of eFCG and R-eFCG and found that all their components are essential.
3. We monitor the impact of the number of heterogeneous GCN layers on the performance of the Android malware detection model and found that its performance increases with the increasing number of layers.

The rest of this paper is organised as follows: Section II demonstrates a simple app and its FCG used throughout this paper. Several relevant related works are discussed in Section III. Section IV provides an overview of mathematical concepts used in this paper. The Algorithms to obtain eFCG and R-eFCG, along with the architecture of the Android malware detection approach, are described in Section V. The experimental framework to evaluate the current work and its results are discussed in Section VI. Finally, the paper is concluded in Section VII along with discussing future directions.

II. MOTIVATION

A simple app containing a button (class `Button`) and a text view (class `TextView`) has been used to demonstrate the FCG and its enhancements throughout this work. When the user clicks on the button, the app starts tracking their location in the background and logs it periodically to the text view. Its source code and the FCG are shown in Figure 1. where the registration methods `Button.setOnClickListener()` (line 45 in Figure 1a) and `LocationManager.requestLocationUpdates()` (line 26-31 in Figure 1a) are not connected to their callback handlers `onClick()` (line 20 in Figure 1a) and `onLocationChangeed()` (line 7 in Figure 1a), respectively, in the FCG.

To include relationships between registration methods and associated callback handlers, the Framework Space has to be analysed to obtain a mapping between all possible registration and callback methods. This list has to be used while analysing the APK file to identify the implementation of callback methods as callback handlers and associate them with their registration methods. This association has to be represented with a different *edge type* in FCG, as it is different from regular caller – callee edge type. The presence of multiple edge types makes FCG *heterogenous*. The heterogenous FCG can be further enhanced by adding relationships between the Framework Space and Permissions. FCGs can

Code snippet of the demo app.

The FCG of the code shown in 1a. Framework Space nodes are rectangle and Application Space nodes are oval in shape. Note that the registration methods Button; setOnClickListener and LocationManager; requestLocationUpdates are not connected to their callback handlers MainActivity$2; onClick and MainActivity$1; onLocationChanged, respectively. Also, their corresponding Framework Space callback methods are isolated from the rest of the nodes.

A simple app along with its FCG, showing FCG not capturing interations between callback methods and their registration methods.

TABLE 1: Summary of the notations

Notation	Meaning
Heterogeneous Graphs	
$\mathcal{V}$	Set of node types.
τ	A node type, $\tau \in \mathcal{V}$.
$\mathcal{E}$	Set of edge types.
t	An edge type, $t \in \mathcal{E}$.
V_τ	Set of nodes with type τ.
V	Set of all nodes.
E_t	Set of edges with type t.
E	Set of all edges.
G_M	Metagraph of the heterogeneous graph G.
A_τ	Attribute function for node type τ.
$\mathbb{A}\tau$	Attribute space of the node type τ.
Neighbourhood of node v in the graph G with node type τ.	
$\{\text{parents}_\tau\}_G(v)$	Set of parents of the node.
$\{\text{children}_\tau\}_G(v)$	Set of children of the node.
$\{\mathcal{N}_\tau\}_G(v)$	Set of 1-hop neighbours of the node.
$\text{pred}_G(v)$	Set of predecessors of node v in a DAG G.
$\text{succ}_G(v)$	Set of successors of node v in a DAG G.
The dex code	
$\mathcal{C}$	Set of classes defined and referenced in the `dex code`.
$\mathcal{M}$	Set of methods defined and referenced in the `dex code`.
$\text{is}F(\cdot)$	Is the flag F is present in the definition of its argument.
$\text{methods}(c)$	Set of methods of the class c.
$\text{class\_parents}(c)$	Set of parent classes of the class c.
$\text{constructors}(c)$	Set of constructors of the class c.
$\text{argumentTypes}(m)$	Set of arguments of the method m.
$\text{class}(m)$	The class to which the method m belongs to.
$\text{sig}(m)$	Signature of the method m.
Γ	The FCG.
$\mathcal{I}$	The Inheritance Graph.
$(\cdot)^{(\mathsf{M})}$	Method-level graphs and edges.
$(\cdot)^{(\mathsf{C})}$	Class-level graphs and edges.

B. GRAPHS

A *directed* graph $G(V, E)$ is a collection of nodes V and edges $(u, v) \in E$ where $u, v \in V$. A *multigraph* is a graph in which E is a multiset, allowing multiple edges between two nodes. A graph is *undirected* if $(u, v) \in E \implies (v, u) \in E$.

A path p is a sequence of edges $e_1 \to e_2 \to \cdots \to e_n$ where every edge $e_i = (u_i, v_i)$ is distinct and $u_i = v_{i-1} \forall i > 1$. Two nodes x and y are *connected* in G if there is a path between them. A graph is *acyclic* if there are no paths in G such that $u_1 = v_n$. A *Directed Acyclic Graph (DAG)* is a graph which is both directed and acyclic. As all these graphs consist of a single type of nodes and edges, they are *homogeneous*. Interested readers are referred to [21] for further information about graphs.

If a graph contains multiple types of nodes or edges (or both), it becomes *heterogeneous*. Heterogeneous graphs occur naturally in many fields such as Recommender Systems [36] [41] and Bioinformatics [22]. The concept of heterogeneous graphs is illustrated here in light of the FCG in Figure 1b, which contains nodes of Application Space $\mathcal{A}$ and Framework Space $\mathcal{F}$. Formally, a *directed heterogeneous graph* is $G(\mathcal{V}, \mathcal{E}, V, E)$ where,

- $\mathcal{V}$ is the set of *node types* (e.g, $\{\mathcal{A}, \mathcal{F}\}$),
- $\mathcal{E} \subseteq \mathcal{V}^2$ is a multiset of *edge types*, each associated with a name (e.g, $\{\text{calls} : (\mathcal{A}, \mathcal{A}),\ \text{calls} : (\mathcal{A}, \mathcal{F})\}$,
- $V = \cup_{\tau \in \mathcal{V}} V_\tau$ is the set of the nodes and,
- $E = \cup_{t \in \mathcal{E}} E_t$ is the set of edges.

An edge set can be denoted by the name of its type followed by the nodes it connects to (e.g., $E_{\text{calls}:\mathcal{A} \mapsto \mathcal{F}}$). The names of the nodes can be omitted if no other edge with the same name is present in the edge types.

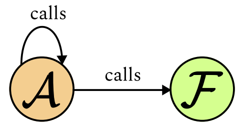

FIGURE 2: The metagraph of the FCG shown in Figure 1b.

A heterogeneous graph becomes undirected if $\forall t \in \mathcal{E}$, $\exists t' \in \mathcal{E}$ s.t $(u, v) \in E_t \implies (v, u) \in E_{t'}$. The structure of the heterogeneous graph is represented as a multigraph $G_M(\mathcal{V}, \mathcal{E})$ called as the *metragraph* of G. Figure 2 shows the metagraph of the FCG shown in Figure 1b. Interested readers are referred to [42] for further information about heterogeneous graphs.

If every node of the graph is associated with some attributes, the graph is called an *attributed graph.* The *attribute function* $A_\tau : V_\tau \to \mathbb{A}_\tau$ defines the attributes for each node $v \in V_\tau$ of type τ in *attribute space* $\mathbb{A}_\tau$. For homogeneous graphs, there is only one attribute space.

For a graph G with nodes V and edges E, we use following notations to denote the information about neighbourhood of $v \in V$ with node type $\tau \in \mathcal{V}$ in G: $\{\text{parents}_\tau\}_G(v) = \{u | (u, v) \in E \wedge u \in V_\tau\}$ is the set of v's parents with type τ, $\{\text{children}_\tau\}_G(v) = \{w | (v, w) \in E \wedge\ w \in V_\tau\}$ is the set of its children with type τ, $\{\mathcal{N}_\tau\}_G(v) = \{\text{children}_\tau\}_G(v)$ is the set of its 1-hop neighbours with type τ. If G is a Homogeneous DAG, $\text{pred}_G(v) = \{u | u \text{ and } v \text{ are connected in } G\}$ is the set of predecessors of v, $\text{succ}_G(v) = \{w | v \text{ and } w \text{ are connected in } G\}$ is the set of successors of v. In every notation, the subscript G is omitted when the graph in question can be inferred from the context and, type subscript τ is omitted if the graph is homogeneous or when we refer to nodes with all types.

C. THE DEX FILE

The `classes.dex` present inside the APK contains the application logic represented as the `dex code`, to be executed by Android Runtime [32]. Android API is also bundled in several `dex` files, residing in `/system/framework/framework.jar` in the case of Android 11.

By parsing the `dex code`, one can obtain the sets of classes $\mathcal{C}$ and methods $\mathcal{M}$ implemented and referenced within its scope. Note that the *interfaces* and *enums* are treated as classes, and the *constructors* are treated as methods in the `dex code`. The definition of every class $c \in \mathcal{C}$ and method $m \in \mathcal{M}$ associates them with several *flags*. These flags include *modifier* information (e.g., `public`, `static` and `abstract`) and the declaration type in the code (e.g., `interface`, `enum` and `constructor`). We define a Boolean function $isF(\cdot)$, which returns *true* whenever the flag F is present in the definition of its argument.

Apart from the flags, the definition of the class c includes a list of its methods ($\text{methods}(c)$), along with a list of its parents in the inheritance hierarchy (class_parents(c)). The constructors of c can be obtained by filtering its methods with the flag `constructor`, i.e., constructors(c) = $\{m \mid m \in \text{methods}(c) \ \wedge\ \text{isConstructor}(m)\}$. Similarly, the definition of method m includes the types of its arguments (argumentTypes(m)) and a reference to the class to which it belongs to (class(m)). Multiple methods in a class can have the same name due to *method overloading*; thus, the method name along with its argument type list (the *signature*, denoted by sig(m)) is unique for every method.

If a method m is internal, the `dex code` includes its bytecode in the `dex` format. The bytecode consists of a sequence of instructions, with each instruction containing an opcode and operand(s). Each opcode is 8-bit in length, making 256 opcodes possible, of which only 230 are used [15]. As many of the opcodes do a similar task (ex., opcode range `0x90-0xE2` consists of binary operations such as `add`, `sub` and `mul`), they can be grouped based on their functionality. While [23] constructed 15 opcode groups, [35] constructed 21 opcode groups. This work uses opcode groups of [35]. Interested readers are referred to the Dalvik Specification [32] to get more information about the `dex code`.

Using the relationships among the methods $\mathcal{M}$ and classes $\mathcal{C}$ contained in the `dex code`, several graphs can be constructed. Out of them, the **Class-level Inheritance Graph** $\mathcal{I}^{(\mathsf{C})}(\mathcal{C}, E^{(\mathsf{C})}_{\text{parentOf}})$, where $(c_i, c_j) \in E^{(\mathsf{C})}_{\text{parentOf}} \iff c_i \in \text{class\_parents}(c_j)$, represents the inheritance hierarchy among the classes. The **(Method-level) Inheritance Graph** $\mathcal{I}^{(\mathsf{M})}(\mathcal{M}, E^{(\mathsf{M})}_{\text{parentOf}})$ is obtained using $\mathcal{I}^{(\mathsf{C})}$ using (1).

$$E^{(\mathsf{M})}_{\text{parentOf}} = \{(m_i, m_j) \mid (\text{class}(m_i), \text{class}(m_j) \in E^{(\mathsf{C})}_{\text{parentOf}} \wedge \text{sig}(m_i) = \text{sig}(m_j)\} \tag{1}$$

Note that the Inheritance Graphs $\mathcal{I}^{(*)}$ are DAGs, as cyclic dependencies among classes (thus methods) in terms of inheritance are not allowed. The **Function Call Graph** $\Gamma^{(\mathsf{M})}(\mathcal{M}, E^{(\mathsf{M})}_{\text{calls}})$, where $(m_i, m_j) \in E^{(\mathsf{M})}_{\text{calls}}$ if m_i calls m_j in its code, captures the Caller-Callee relationships among the methods in the `dex code`. The superscripts (M) and (C) indicate that the edges are among methods and class nodes, respectively. If the superscript is not present in the graph name (e.g., Γ and $\mathcal{I}$), they are assumed to method-level.

V. PROPOSED APPROACH

The proposed Android malware detection approach consists of two analysis stages – Framework Space Analysis and Application Space Analysis, followed by a Heterogeneous GCN based Android malware detection model. The Framework Space Analysis is done once, and its outputs are re-used in the Application Space Analysis for every app. Separate Heterogeneous GCN models are trained for eFCG and R-eFCG obtained by the Application Space Analysis. The following sections describe every stage in detail.

A. FRAMEWORK SPACE ANALYSIS

The Framework Space Analysis analyses the Android Framework to extract a mapping between Registration and Callback methods. To do so, the `dex` file containing Framework Space code has to be parsed to get the set of Framework Classes $\mathcal{C}_\mathcal{F}$ and Framework Methods $\mathcal{M}_\mathcal{F}$. From $\mathcal{C}_\mathcal{F}$ and $\mathcal{M}_\mathcal{F}$, the Framework Space Inheritance Graph $\mathcal{I}_\mathcal{F}$ and Framework Space FCG $\Gamma_\mathcal{F}$ are obtained, respectively. The approach of [10] is adopted to extract potential callback methods from the Framework Space code, which are then filtered to obtain final callback methods along with corresponding registration methods. The architecture of Framework Space Analysis is shown in Figure 3.

1) Potential Callback Filter

A potential callback method is a Framework Space method which is visible to the Application Space and can be overridden by it. For a method m with $c = \text{class}(m)$, if all of the following criterion are satisfied, then it becomes a potential callback method [10]:

1. isPublic$(c) = 1$
2. isFinal$(c) = 0$
3. isInterface$(c) = 1 \ \vee \ \bigvee_{x \in \text{constructors}(c)} \text{isPublic}(x)$
4. isPublic$(m) = 1 \vee$ isProtected$(m) = 1$

Criterion 1, 2 and 3 ensure that the class c is visible to Application Space classes and can be extended; Criteria 4 ensures that the method m can be overridden in Application Space. As all interface methods are public by default, Criteria 4 is true for them. P denotes the set of all potential callbacks.

2) Registration-Callback Map Extraction

A method m being potential callback does not guarantee that its Application Space override m' can be *introduced* back to the Framework Space through an Application Space visible registration method r and, subsequently called back by the Framework Space. Note that to *introduce* m' to r, the method r must take an argument of type $c = \text{class}(m)$, thus, accepting any instance of class c' derived from c, overriding m in its method m'.

To filter out the methods m whose overrides cannot be introduced to Framework Space, we use *Argument Map*. The **Argument Map** is a multimap $\alpha_\mathcal{F} : \mathcal{C}_\mathcal{F} \rightarrow \mathcal{M}_\mathcal{F}$, where $(c, m) \in \alpha_\mathcal{F} \iff c \in \text{argumentTypes}(m)$. In other words, for a Framework Space class c, $\alpha_\mathcal{F}(c)$ is a set of methods $M \subset \mathcal{M}_\mathcal{F}$, in which c is an argument of. If $\alpha_\mathcal{F}(c) = \varnothing$ for $c = \text{class}(m)$, then the class c cannot be passed back to the Framework Space, therefore all of its methods are not callback methods.

A registration method r taking an argument of type c need not necessarily invoke the method m of c. To check for the invocation of m, a complete *reverse data-flow analysis* tracking c until the invocation of m is required as in [10]. However, we empirically observe that the invocation of m happens in a method μ either belonging to $u = \text{class}(r)$ or some nested class u' of u most of the times. Therefore, the

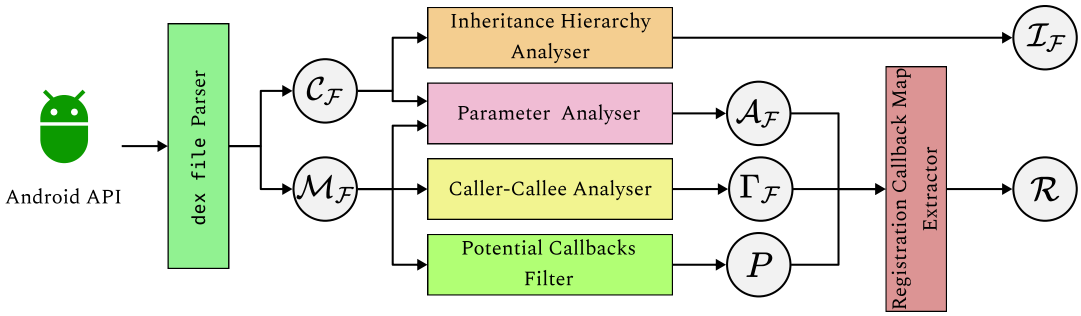

FIGURE 3: The workflow of Framework Space Analysis

criterion to consider the method m with $c = \text{class}(m)$ as a final callback method are defined as follows:

1. c is an argument of some Application Space visible method r. i.e., $\exists\ r \in \alpha_\mathcal{F}(c)$ s.t. $\text{isPublic}(r) \wedge \text{isPublic}(\text{class}(r))$, and,
2. Some method μ either belonging to $u = \text{class}(r)$ or some nested class u' of u invokes m in its code.

If a method m satisfies above criterion, then the method r is the *registration method* of m and, the pair (r, m) is added to the Registration-Callback map $\mathcal{R}$. The process of extracting the Registration-Callback map is summarized in Algorithm 1.

Algorithm 1 Framework Space Analysis

procedure ANALYSEFRAMEWORK($\mathcal{C}_\mathcal{F}, \mathcal{M}_\mathcal{F}$)
  ▷ Extract the Registration-Callback map $\mathcal{R}$ using $\mathcal{C}_\mathcal{F}$ – Set of Framework Space Classes and $\mathcal{M}_\mathcal{F}$ – Set of Framework Space Methods.
  $\mathcal{I}_\mathcal{F} \leftarrow$ Extract Inheritance Graph using $\mathcal{C}_\mathcal{F}$ ▷ See Section IV-C
  $\Gamma_\mathcal{F} \leftarrow$ Extract FCG using $\mathcal{M}_\mathcal{F}$ ▷ See Section IV-C
  $\alpha_\mathcal{F} \leftarrow$ Extract Argument Graph using $\mathcal{M}_\mathcal{F}$ and $\mathcal{C}_\mathcal{F}$ ▷ See Section V-A2
  $P \leftarrow$ Extract Set of Potential Callbacks from $\mathcal{M}_\mathcal{F}$ ▷ See Section V-A1
  $C_P \leftarrow \varnothing$ ▷ Multimap of methods in P keyed by their class
  for m in P **do**
    if $\exists u$ s.t. $(\text{class}(m), u) \in \alpha_\mathcal{F}$ **then**
      ▷ check if $\text{class}(m)$ is used anywhere
      $C_P \leftarrow C_P \cup (\text{class}(m), m)$
    end if
  end for
  $\mathcal{R} \leftarrow \varnothing$ ▷ Registration Callback Pairs
  for c in $\text{dom}(C_P)$ **do**
    $R \leftarrow \{(\text{class}(r), r) \mid r \in \alpha_\mathcal{F}(c) \wedge \text{isPublic}(r) \wedge \text{isPublic}(\text{class}(r))\}$ ▷ Multimap of possible registration methods for class c keyed by their classes
    for p in $C_P(c)$ **do** ▷ Loop through Potential Callback methods p of class c
      $U \leftarrow \{\text{class}(u) \mid u \in \text{parents}_{\Gamma_\mathcal{F}}(p)\}$
      ▷ Set of classes that have at least one method calling p
      $\mathcal{R} \leftarrow \mathcal{R} \cup \{(r, p) \mid c \in (U \cap \text{dom}(R)) \wedge r \in R(c)\}$
      ▷ Update Registration-Callback map considering the classes that call p and have registration method containing c in their argument. Note that the $\cap$ operation is *approximate* (see Section V-A2).
    end for
  end for
  return $\mathcal{R}, \mathcal{I}_\mathcal{F}$
end procedure

Note that whenever r is a registration method, any Framework Space child r' of r can be a registration method too, assuming that r' invokes r with its parameters. Therefore, $(r, p) \in \mathcal{R} \implies (r', p) \in \mathcal{R}$. As adding (r', p) to the Registration-Callback map $\mathcal{R}$ increases the size of $\mathcal{R}$ significantly, Framework Space Inheritance Graph $\mathcal{I}_\mathcal{F}$ is provided to Application Space Analysis to infer such relationships.

B. APPLICATION SPACE ANALYSIS

Application Space Analysis extracts the `dex` file from the APK and parses it to get the set of Application Space classes and methods $\mathcal{C}_\mathcal{A}$ and $\mathcal{M}_\mathcal{A}$, respectively. Note that the $\mathcal{C}_\mathcal{A}$ (and $\mathcal{M}_\mathcal{A}$) includes the classes (and methods) implemented in Application Space $C_\mathcal{A}$ ($M_\mathcal{A}$), along with the reference to classes (methods) from Framework Space $C_\mathcal{F} \subset \mathcal{C}_\mathcal{F}$ ($M_\mathcal{F} \subset \mathcal{M}_\mathcal{F}$). Therefore, $\mathcal{C}_\mathcal{A} = C_\mathcal{A} \cup C_\mathcal{F}$ and $\mathcal{M}_\mathcal{A} = M_\mathcal{A} \cup M_\mathcal{F}$. The $\mathcal{M}_\mathcal{A}$ and $\mathcal{C}_\mathcal{A}$ are used to derive Application Space FCG $\Gamma_\mathcal{A}\left(\mathcal{M}_\mathcal{A}, E^{(\mathsf{M})}_{\text{calls}}\right)$ and Application Space Method level Inheritance Graph $\mathcal{I}_\mathcal{A}\left(\mathcal{C}_\mathcal{A}, E^{(\mathsf{M})}_{\text{parentOf}}\right)$, respectively. As the methods in $M_\mathcal{F}$ are only references, their inheritance information is not contained in $\mathcal{I}_\mathcal{A}$. The edges $E^{(\mathsf{M})}_{\text{calls}}$ of the FCG $\Gamma_\mathcal{A}$ can be partitioned into $E^{(\mathsf{M})}_{\text{calls}:\mathcal{A}\mapsto\mathcal{A}}$ and $E^{(\mathsf{M})}_{\text{calls}:\mathcal{A}\mapsto\mathcal{F}}$ to represent Caller-Callee relationships between methods in different spaces.

The Application Space Analysis proceeds through several stages as outlined in Figure 4, each enriching the FCG, converting it to eFCG Γ_e at the end. The eFCG is a heterogeneous graph $\Gamma_e(\mathcal{V}, \mathcal{E}, V^{(\mathsf{M})}, E^{(\mathsf{M})})$ where,

- $\mathcal{V} = \{\mathcal{A}, \mathcal{F}, \mathcal{P}\}$ is the set of node types,
- $\mathcal{E} = \{\text{calls} : (\mathcal{A}, \mathcal{A}), \text{calls} : (\mathcal{A}, \mathcal{F}), \text{parentOf} : (\mathcal{A}, \mathcal{A}), \text{parentOf} : (\mathcal{F}, \mathcal{A}), \text{callsBack} : (\mathcal{F}, \mathcal{F}), \text{requires} : (\mathcal{F}, \mathcal{P})\}$ is the set of edge types,
- $V^{(\mathsf{M})}_\mathcal{F} = M_\mathcal{F}$, $V^{(\mathsf{M})}_\mathcal{A} = M_\mathcal{A}$, and $V^{(\mathsf{M})}_\mathcal{P} = P$ are the sets of nodes, and
- $E^{(\mathsf{M})}_{\text{calls}:\mathcal{A}\mapsto\mathcal{A}}$, $E^{(\mathsf{M})}_{\text{calls}:\mathcal{A}\mapsto\mathcal{F}}$, $E^{(\mathsf{M})}_{\text{parentOf}:\mathcal{A}\mapsto\mathcal{A}}$, $E^{(\mathsf{M})}_{\text{parentOf}:\mathcal{F}\mapsto\mathcal{A}}$, $E_{\text{callsBack}}$ and E_{requires} are the sets of edges.

The metagraph $\{\Gamma_e\}_M$ of the eFCG is shown in Figure 5. The Application Space Analysis further reduces eFCG into R-eFCG $\Gamma^{(\mathsf{C})}_e$ using eFCG reducer. These stages of the Application Space Analysis and the nodes and edges they add to the FCG are described in detail in the following paragraphs.

Stages in Application Space Analysis

Metagraph $\{\mathcal{G}^{(\cdot)}_e\}_M$ of eFCG and R-eFCG

1) Inheritance Edges Adder

The event handlers are implemented in the Application Space as an overridden method of a Framework Space callback method. The FCG cannot capture this information as the event handler does not call its parent callback method most of the time.

To add the relationship between event handler and its parent callback method to the FCG, the inheritance hierarchy has to be considered. By adding the edges in $E^{(M)}_{parentOf}$ contained in Method level Inheritance Graph I_A, the event handlers are connected to their parent callback methods, along with connecting Application Space methods to their parents. Thin dashed edges in Figure 6a represent the edges in $E^{(M)}_{parentOf}$.

As the inheritance may be among Application Space nodes A, or from the Framework Space nodes F to the Application Space nodes A, the inheritance edge set $E^{(M)}_{parentOf}$ can be partitioned into $E^{(M)}_{parentOf:\ A \mapsto A}$ and $E^{(M)}_{parentOf:\ F \mapsto A}$ to represent these cases, respectively.

2) Callback Edges Adder

The registration methods and the callback methods are not related in the FCG, as their Caller-Callee relationship cannot be inferred without the help of the results of Framework Space Analysis.

The Registration Callback map R can be used to add edges between the registration methods and the corresponding callback methods. As the Framework Space inheritance information is not contained in I_A (thus in $E^{(M)}_{parentOf}$), I_F has to be considered while adding callback edges.

For every Framework Space method m in M_F, with the help of R and I_F, it is determined whether m is a registration method. If so, the corresponding callback methods P are obtained. The edges between m and the callback method $p \in P$ is added if $p \in M_F$. The process of obtaining callback edges $E^{(M)}_{callsBack}$ is detailed in Algorithm 2. Bold dashed edges in Figure 6a represent the edges in $E^{(M)}_{callsBack}$. Note that the edges in $E^{(M)}_{callsBack}$ are always among Framework Space methods F.

Algorithm 2 Callback Edge Addition

```
procedure GETCALLBACKEDGES(G_A; I_F; R)
    ▷ Get a list of callback edges E^(M)_callsBack using Application Space
      FCG G_A, Framework Space Method level Inheritance Graph I_F, and
      Registration Callback map R.
    E^(M)_callsBack ← ∅
    for m in M_F do
        for p in {m} ∪ pred_{I_F}(m) do
            if p ∈ dom(R) then
                E^(M)_callsBack ← E^(M)_callsBack ∪ {(m; c) | c ∈ R(p) ∧ c ∈ M_F}
            end if
        end for
    end for
    return E^(M)_callsBack
end procedure
```

3) Permission Nodes Adder

The manifest file contains a list of permissions that are required by an app to run. As it is possible to request permission and not use it [11], permissions required by used Framework Space methods can be used to get a list of actual permissions needed. Axtool [7] provides a mapping $\Pi : M_F \to P$ between the Framework Space methods M_F and Permission Space P. For a Framework method $m \in M_F$, $\Pi(m)$ is the set of permissions that is required by m.

The permission nodes P and the edges $E^{(M)}_{requires}$ to be added to the FCG are calculated using (2) and (3), respectively.

$$P = \bigcup_{m \in M_F} \Pi(m) \tag{2}$$

TABLE 2: Summary of the experimental results

n	$\mathcal{V}$	Γ	Accuracy (%)	Precision	Recall	F1-Score
0	all	R-eFCG	86.69	0.8703	0.8606	0.8654
		eFCG	84.76	0.8468	0.8468	0.8468
	core	R-eFCG	87.11	0.8727	0.8675	0.8701
		eFCG	81.85	0.8324	0.7952	0.8134
	code	R-eFCG	71.70	0.7685	0.6170	0.6845
		eFCG	70.03	0.7463	0.6024	0.6667
1	all	R-eFCG	94.22	0.9319	0.9535	0.9426
		eFCG	92.89	0.9123	0.9484	0.9300
	core	R-eFCG	92.77	0.9218	0.9337	0.9277
		eFCG	90.03	0.8933	0.9079	0.9006
	code	R-eFCG	85.40	0.8756	0.8236	0.8488
		eFCG	82.62	0.8276	0.8219	0.8247
2	all	R-eFCG	95.55	0.9521	0.9587	0.9554
		eFCG	94.82	0.9385	0.9587	0.9485
	core	R-eFCG	95.21	0.9487	0.9552	0.9520
		eFCG	91.74	0.9061	0.9303	0.9180
	code	R-eFCG	89.38	0.9051	0.8787	0.8917
		eFCG	86.22	0.8730	0.8460	0.8593
3	all	R-eFCG	95.93	0.9556	0.9630	0.9593
		eFCG	94.43	0.9250	0.9664	0.9453
	core	R-eFCG	94.73	0.9377	0.9578	0.9476
		eFCG	92.08	0.9129	0.9294	0.9211
	code	R-eFCG	91.14	0.9252	0.8941	0.9094
		eFCG	87.50	0.8891	0.8554	0.8719
4	**all**	**R-eFCG**	**96.28**	**0.9662**	**0.9587**	**0.9624**
		eFCG	95.33	0.9466	0.9604	0.9534
	core	R-eFCG	96.15	0.9629	0.9596	0.9612
		eFCG	93.36	0.9242	0.9441	0.9340
	code	R-eFCG	91.57	0.9170	0.9131	0.9150
		eFCG	90.80	0.9216	0.8907	0.9059

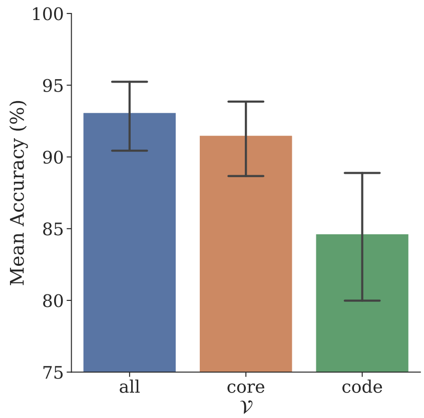

FIGURE 7: Mean and Standard Deviation of Accuracy of the model for different node type configurations.

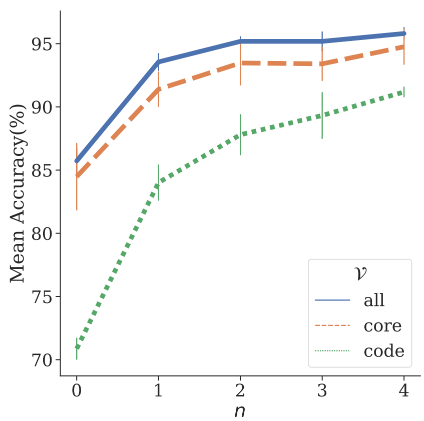

FIGURE 8: Mean Accuracy of the model containing n GCN layers. Shaded area represents the standard deviation of accuracy.

1) Effectiveness of Node types

With Application Space nodes $\mathcal{A}$ only, the model was able to achieve a mean accuracy of 84.63% with a standard deviation of 7.79%. With the addition of framework space nodes $\mathcal{F}$, the mean accuracy was increased by 6.86%, reaching 91.49% with a standard deviation of 4.29%. The addition of permission nodes slightly improved the mean accuracy by 1.58%, making the model achieve a mean accuracy of 93.07% with a standard deviation of 4.02%. The trend of increasing accuracy with the addition of node types is shown in Figure 7. These results emphasise that the Framework Space nodes are crucial to detect Android malware. Similarly, the contribution of permission nodes to the performance of the model is essential, although they are less in number.

2) Effect of neighbourhood size n

With $n = 0$, the baseline models performed better than a random-guess model obtaining a mean accuracy of 80.35% with a standard deviation of 7.60%, suggesting that the node attributes play an essential role to detect Android malware. Subsequent addition of GCN layers improved the mean accuracy by 9.29%, 2.49%, 0% and, 1.27%, respectively. No performance improvements were observed during the addition of the third GCN layer for "core" and "all" configurations. The addition of the fourth GCN layer did not improve the accuracy by a significant amount. The variation of accuracy with the addition of GCN layers shown in Figure 8 suggests that $n = 2$ is a sweet spot between accuracy and inference time, as the number of GCN layers directly affect the inference time.

3) Generalisation ability of R-eFCG

R-eFCGs performed better then eFCGs all node configurations as evident from Table 2. A statistical analysis of the accuracies obtained with eFCGs and R-eFCGs suggest that the R-eFCGs improve the mean accuracy by 2.35% with a standard deviation of 1.25%. Minimum improvements less than 1% were observed with $n = 4$ and node configuration "code" and "all" along with $n = 2$ with node configuration "all".

These results suggest that R-eFCGs can generalise better than eFCGs in most cases. In the *sweet spot* $n = 2$ with node configuration all, R-eFCGs can be used as a replacement to eFCGs, thus making inference faster, as they have fewer nodes than eFCGs. Note that R-eFCG $\Gamma_e^{(\mathsf{C})}$ has to be calculated after Γ_e (see Section V-B), thus adding additional computational step. However, the procedure of section V-B can be easily tuned to output R-eFCGs instead of eFCGs by considering classes instead of methods and using their attribute schemes.

Comparison with Related Works

The "core" configuration of this work using eFCGs is conceptually similar to FCGs used in [35]. While [35] reported accuracy of 92.29% with 3 GCN layers, the "core" configuration using eFCGs with $n = 3$ achieved a similar accuracy of 92.08%. The proposed method could not be compared with [9] [40] as they did not incorporate any node-count distribution balancing strategies and did not disclose their dataset.

VII. CONCLUSIONS AND FUTURE WORK

This paper proposed an Android malware detection approach based on the heterogenous Caller-Callee graphs extracted from the APK files. First, the heterogeneous graphs eFCG and R-eFCG were defined, and algorithm to obtain the same were discussed. These graphs incorporate the information about callback and permissions obtained by the Framework Space Analysis. Then, separate heterogeneous graph models were trained on them to evaluate their performance. Finally, the experiments to determine optimal neighbourhood and essential components of heterogeneous graphs were also conducted. As a result of these experiments, a maximum accuracy of 96.28% was obtained.

There is further scope to improve this work in multiple directions. During Framework Space Analysis, the algorithm to find Registration-Callback map can be made more exact, and the difference of their results with our approximate method can be compared and contrasted. In Application Space Analysis, the nodes can be assigned more informative features, such as package name-based embedding for Framework Space nodes and opcode sequence embedding for Application Space Nodes. Finally, explainability methods can be integrated with the GCN models to identify and understand critical nodes that contain malicious code.

REFERENCES

[1] Moutaz Alazab, Mamoun Alazab, Andrii Shalaginov, Abdelwadood Mesleh, and Albara Awajan. Intelligent mobile malware detection using permission requests and API calls. Future Generation Computer Systems, 107:509 – 521, 2020.

[2] Kevin Allix, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. AndroZoo: Collecting Millions of Android Apps for the Research Community. In Proceedings of the 13th International Conference on Mining Software Repositories, MSR '16, pages 468–471, New York, NY, USA, 2016. ACM.

[3] Androguard Development Team. Androguard. Online (URL: https://github.com/androguard/androguard), 2021. Accessed (18.05.2020).

[4] Android Developers. Package Index. Online (URL: https://developer.android.com/reference/packages), 2021. Accessed: 18.05.2021.

[5] Android Source. AndroidManifest.xml. Online (URL: https://github.com/aosp-mirror/platform_frameworks_base/blob/master/core/res/AndroidManifest.xml), 2021. Accessed 19.05.2021.

[6] Daniel Arp, Michael Spreitzenbarth, Malte Hübner, Hugo Gascon, and Konrad Rieck. Drebin: Effective and explainable detection of android malware in your pocket. In Proceedings 2014 Network and Distributed System Security Symposium. Internet Society, 2014.

[7] Michael Backes, Sven Bugiel, Erik Derr, Patrick McDaniel, Damien Octeau, and Sebastian Weisgerber. On Demystifying the Android Application Framework: Re-Visiting Android Permission Specification Analysis. In 25th USENIX Security Symposium (USENIX Security 16), pages 1101–1118, Austin, TX, August 2016. USENIX Association.

[8] Lukas Biewald. Experiment tracking with weights and biases, 2020. Software available from https://wandb.ai/site.

[9] Minghui Cai, Yuan Jiang, Cuiying Gao, Heng Li, and Wei Yuan. Learning features from enhanced function call graphs for Android malware detection. Neurocomputing, 423:301–307, 2021.

[10] Yinzhi Cao, Yanick Fratantonio, Antonio Bianchi, Manuel Egele, Christopher Kruegel, Giovanni Vigna, and Yan Chen. EdgeMiner: Automatically Detecting Implicit Control Flow Transitions through the Android Framework. In 22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015. The Internet Society, 2015.

[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, 15:987–1001, 2020.

[12] Khanh-Huu-The Dam and Tayssir Touili. Learning Android Malware. In Proceedings of the 12th International Conference on Availability, Reliability and Security, ARES '17, New York, NY, USA, 2017. Association for Computing Machinery.

[13] Danilo Dominguez Perez and Wei Le. Predicate Callback Summaries. In 2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C), pages 291–293, 2017.

[14] William Falcon et al. Pytorch lightning. GitHub. Note: https://github.com/PyTorchLightning/pytorch-lightning, 3, 2019.

[15] Gabor Paller. Dalvik opcodes, 2020. URL: http://pallergabor.uw.hu/androidblog/dalvik_opcodes.html (accessed: 07.12.2020).

[16] T. Gao, W. Peng, D. Sisodia, T. K. Saha, F. Li, and M. Al Hasan. Android malware detection via graphlet sampling. IEEE Transactions on Mobile Computing, 18(12):2754–2767, Dec 2019.

[17] Joblib Development Team. Joblib: running Python functions as pipeline jobs. Online (URL: https://joblib.readthedocs.io/en/latest/), 2021. Accessed (18.05.2021).

[18] H. M. Kim, H. M. Song, J. W. Seo, and H. K. Kim. Andro-Simnet: Android Malware Family Classification using Social Network Analysis. In 2018 16th Annual Conference on Privacy, Security and Trust (PST), pages 1–8, Aug 2018.

[19] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. In Yoshua Bengio and Yann LeCun, editors, 3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings, 2015.

[20] Thomas N. Kipf and Max Welling. Semi-Supervised Classification with Graph Convolutional Networks. In International Conference on Learning Representations (ICLR), 2017.

[21] Oscar Levin. Discrete mathematics : an open introduction. Oscar Levin, Greeley, Colorado, 2019.

[22] Jin Li, Sai Zhang, Tao Liu, Chenxi Ning, Zhuoxuan Zhang, and Wei Zhou. Neural inductive matrix completion with graph convolutional networks for miRNA-disease association prediction. Bioinformatics, 36(8):2538–2546, 01 2020.

[23] Yu Liu, Liqiang Zhang, and Xiangdong Huang. Using G Features to Improve the Efficiency of Function Call Graph Based Android Malware Detection. Wireless Personal Communications, 103(4):2947–2955, 2018.

[24] S. Mahdavifar, A. F. Abdul Kadir, R. Fatemi, D. Alhadidi, and A. A. Ghorbani. Dynamic Android Malware Category Classification using Semi-Supervised Deep Learning. In 2020 IEEE Intl Conf on Dependable, Autonomic and Secure Computing, Intl Conf on Pervasive Intelligence and Computing, Intl Conf on Cloud and Big Data Computing, Intl Conf on Cyber Science and Technology Congress (DASC/PiCom/CBDCom/CyberSciTech), pages 515–522, 2020.

[25] McAfee. McAfee Mobile Threat Report 2019. Online (URL: https://www.mcafee.com/enterprise/en-us/assets/reports/rp-mobile-threat-report-2019.pdf), 2020. Accessed: 29.07.2020.

[26] Lucky Onwuzurike, Enrico Mariconti, Panagiotis Andriotis, Emiliano De Cristofaro, Gordon Ross, and Gianluca Stringhini. MaMaDroid: Detecting Android Malware by Building Markov Chains of Behavioral Models (Extended Version). ACM Trans. Priv. Secur., 22(2), April 2019.

[27] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An Imperative Style,

High-Performance Deep Learning Library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, Advances in Neural Information Processing Systems 32, pages 8024–8035. Curran Associates, Inc., 2019.

[28] Abdurrahman Pektas and Tankut Acarman. Deep learning for effective android malware detection using api call graph embeddings. Soft Computing, 24(2):1027–1043, 2020.

[29] Junyang Qiu, Nepal, Surya, Wei Luo, Pan, Lei, Yonghang Tai, Jun Zhang, Yang Xiang, Xiaofeng Chen, Xinyi Huang, and Jun Zhang. Data-Driven Android Malware Intelligence: A Survey. In Machine Learning for Cyber Security, pages 183–202. Springer International Publishing, 2019.

[30] Zhongru Ren, Haomin Wu, Qian Ning, Iftikhar Hussain, and Bingcai Chen. End-to-end malware detection for android iot devices using deep learning. Ad Hoc Networks, 101:102098, 2020.

[31] Kaspersky Securelist. Mobile Malware Evolution 2019. Online (URL: https://securelist.com/mobile-malware-evolution-2019/96280/), 2020. Accessed: 29.07.2020.

[32] Android Source. Dalvik Executable format. Online (URL: https://source.android.com/devices/tech/dalvik/dex-format), 2021. Accessed 18.05.2021.

[33] StatCounter. Mobile, Tablet & Console Operating System Market Share Worldwide. Online (URL: https://gs.statcounter.com/os-market-share/mobile-tablet-console/worldwide/#monthly-201208-202104), May 2021. Accessed 18.05.2021.

[34] Rahim Taheri, Meysam Ghahramani, Reza Javidan, Mohammad Shojafar, Zahra Pooranian, and Mauro Conti. Similarity-based Android malware detection using Hamming distance of static binary features. Future Generation Computer Systems, 105:230 – 247, 2020.

[35] K. V. Vinayaka and C. D. Jaidhar. Android Malware Detection based on Function Call Graph using Graph Convolutional Networks. Accepted and Presented at the Second International Conference on Secure Cyber Computing and Communications (ICSCCC) held at NIT Jalandhar, India, 2021.

[36] Hongwei Wang, Miao Zhao, Xing Xie, Wenjie Li, and Minyi Guo. Knowledge graph convolutional networks for recommender systems. In The World Wide Web Conference, WWW '19, page 3307–3313, New York, NY, USA, 2019. Association for Computing Machinery.

[37] Minjie Wang, Da Zheng, Zihao Ye, Quan Gan, Mufei Li, Xiang Song, Jinjing Zhou, Chao Ma, Lingfan Yu, Yu Gai, Tianjun Xiao, Tong He, George Karypis, Jinyang Li, and Zheng Zhang. Deep Graph Library: A Graph-Centric, Highly-Performant Package for Graph Neural Networks. arXiv preprint arXiv:1909.01315, 2019.

[38] Michelle Y. Wong and David Lie. Tackling runtime-based obfuscation in Android with TIRO. In 27th USENIX Security Symposium (USENIX Security 18), pages 1247–1262, Baltimore, MD, August 2018. USENIX Association.

[39] Y. Wu, X. Li, D. Zou, W. Yang, X. Zhang, and H. Jin. Malscan: Fast market-wide mobile malware scanning by social-network centrality analysis. In 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), pages 139–150, Nov 2019.

[40] Yang Yang, Xuehui Du, Zhi Yang, and Xing Liu. Android Malware Detection Based on Structural Features of the Function Call Graph. Electronics, 10(2), 2021.

[41] Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L. Hamilton, and Jure Leskovec. Graph convolutional neural networks for web-scale recommender systems. In Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery &; Data Mining, KDD '18, page 974–983, New York, NY, USA, 2018. Association for Computing Machinery.

[42] Chuxu Zhang, Dongjin Song, Chao Huang, Ananthram Swami, and Nitesh V. Chawla. Heterogeneous Graph Neural Network. In Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery Data Mining, KDD '19, page 793–803, New York, NY, USA, 2019. Association for Computing Machinery.

[43] Si Zhang, Hanghang Tong, Jiejun Xu, and Ross Maciejewski. Graph Convolutional Networks: Algorithms, Applications and Open Challenges. In Xuemin Chen, Arunabha Sen, Wei Wayne Li, and My T. Thai, editors, Computational Data and Social Networks, pages 79–91, Cham, 2018. Springer International Publishing.