Hindawi
Mobile Information Systems
Volume 2021, Article ID 5538841, 14 pages
https://doi.org/10.1155/2021/5538841

Research Article

Android Malware Detection via Graph Representation Learning

Pengbin Feng,[1] **Jianfeng Ma**,[1] **Teng Li**,[1] **Xindi Ma**,[1] **Ning Xi**,[1] **and Di Lu**[2]

[1]*School of Cyber Engineering, Xidian University, Xi'an, Shaanxi, China*
[2]*School of Computer Science & Technology, Xidian University, Xi'an, Shaanxi, China*

Correspondence should be addressed to Pengbin Feng; pbfeng@xidian.edu.cn

Received 1 February 2021; Revised 4 April 2021; Accepted 23 May 2021; Published 7 June 2021

Academic Editor: Raul Montoliu

With the widespread usage of Android smartphones in our daily lives, the Android platform has become an attractive target for malware authors. There is an urgent need for developing an automatic malware detection approach to prevent the spread of malware. The low code coverage and poor efficiency of the dynamic analysis limit the large-scale deployment of malware detection methods based on dynamic features. Therefore, researchers have proposed a plethora of detection approaches based on abundant static features to provide efficient malware detection. This paper explores the direction of Android malware detection based on graph representation learning. Without complex feature graph construction, we propose a new Android malware detection approach based on lightweight static analysis via the graph neural network (GNN). Instead of directly extracting Application Programming Interface (API) call information, we further analyze the source code of Android applications to extract high-level semantic information, which increases the barrier of evading detection. Particularly, we construct approximate call graphs from function invocation relationships within an Android application to represent this application and further extract intrafunction attributes, including required permission, security level, and Smali instructions' semantic information via Word2Vec, to form the node attributes within graph structures. Then, we use the graph neural network to generate a vector representation of the application, and then malware detection is performed on this representation space. We conduct experiments on real-world application samples. The experimental results demonstrate that our approach implements high effective malware detection and outperforms state-of-the-art detection approaches.

1. Introduction

Android smartphones have been widely used in our daily lives. They can be used to perform daily tasks such as instant messages, online shopping, entertainment, and even financial business. The popularity and openness of the Android platform have not only brought about opportunities for Android application developers but also attracted a large number of malware authors. Currently, a large number of Android apps have been published, updated, and distributed to users via application markets, such as Google Play and Amazon Appstore. Meanwhile, malware could also spread in a convenient way via these application markets. Even worse, most attackers prefer to bypass the security verification mechanism provided by the application market for publishing malware that is camouflaged as a "legitimate" application. Thus, there is an urgent need for an automatic market-scale malware detection approach to prevent the spread of malware.

Security companies have provided a number of anti-malware solutions to automatically detect malware. Their products mainly use the signature-based method, which first generates unique signatures for specific types of malware within a large dataset and then matches the generated signatures with suspicious applications to identify potential threats. However, this detection method can be easily evaded by a variety of code transformation techniques. In addition, the generated signatures cannot handle the explosive growth of malware.

Machine/deep learning-based Android malware detection methods have been an emerging trend. These methods reply on manually selected critical features or automatic feature engineering and are more automated and robust on classifying previously unseen malicious samples. The low

code coverage and poor efficiency of the dynamic analysis limit the large-scale deployment of the malware detection method based on dynamic features. Therefore, researchers have proposed a plethora of detection approaches using static analysis and machine/deep learning [1–4]. Existing approaches have explored abundant static features [5–9], consisting of basic features such as meta-information, opcodes, and API calls, hybrid features, and carefully designed features such as sensitive data flow, obfuscation-invariant features, and API relation graphs. They then build the robust detection model based on various machine/deep learning algorithms from these abundant features.

In this paper, we explore the direction of Android malware detection based on graph representation learning. Without complex feature graph construction, we propose a new approach that adopts the graph neural network directly on call graphs to automatically capture critical patterns. We extract the call graphs (CGs) to represent the whole execution of Android applications. CGs are widely used in software security analysis because they not only provide API calls' information but also reveal the function interaction relationship in the application. Because of the resource consumption and poor efficiency of the precise call graph construction, we instead propose an approximate call graph extraction algorithm based on Apktool [10] for implementing large-scale malware detection. Due to the different importance of different functions in malware detection, we further extract intrafunction features, including required permissions, security levels, and Smali instructions' semantic information. As stated in [11], many Natural Language Processing (NLP) techniques have been used in program language analysis to extract semantic information. Inspired by this direction, we treat intrafunction Smali instructions as words and leverage Word2Vec [12] to extract instruction embeddings that capture the semantic information. Next, we combine all intrafunction instruction embeddings as the function embedding to capture the intrafunction code characteristic information. Finally, the Android application is represented as a graph with node attributes. This approximate call graph could also represent the method invocation relationship within the whole application but save a lot of system resources. The graph neural network has been proven to be successful on a wide range of security detection domains [13]. Therefore, we adopt the graph neural network (GNN) to perform malware classification tasks on approximate CGs. Particularly, GNN is used to embed the extracted CGs into the vector representation for automatically capturing critical information from malware.

In conclusion, this paper makes the following contributions:

(i) We explore the direction of Android malware detection based on graph representation learning and propose a new Android malware detection system CGDroid using the graph neural network and Word2Vec technique.

(ii) CGDroid combines lightweight static analysis with graph representation learning to perform high-precision and -efficiency Android malware detection. We further leverage Word2Vec to generate embeddings for each function to capture intrafunction code characteristic information.

(iii) Compared with prior works based on static features, CGDroid is able to automatically capture semantic information from the call graphs of Android applications without expert knowledge.

(iv) We verify the effectiveness of CGDroid on a large number of real-world applications. The experimental results show that CGDroid is superior to existing detection methods. In addition, the graph neural network is promising in automatically mining the malicious features.

2. Background

In this section, we introduce basic components of Android applications and further describe the assembler of Dalvik bytecode—Smali.

2.1. Android Application. Android applications written in Java are interpreted and executed in the Android environment including the Dalvik Virtual Machine (DVM) and Android Runtime (ART). Each application is actually a zipped file including the source code file, resources, assets, and Android configuration file (AndroidManifest.xml) with the extension of *apk*. It consists of four types of essential components: activity, service, content provider, and broadcast receiver.

Activity is a common component that provides UI for interacting with users and switching context within and between apps. *Service* is a component that makes application run in the background and can start, bind, and stop. *Content provider* is an SQLite database, which provides a general and structured access interface for data sharing across applications. *Broadcast receiver* mainly receives system-wide broadcast events, which include system time changes, low battery power, and system restart. All components are required to register and declare in the Android configuration file of an application. The intent is a lightweight message delivery mechanism, which can perform intercomponent communications within an application and between different applications.

2.2. Smali Code. The source codes of an application are first compiled into Dalvik bytecodes, and then these bytecodes are executed within the Android environment. Smali code, an intermediate representation of Dalvik bytecode, contains abundant information. In the following, we introduce some typical instruction categories which contain critical semantic information within Smali. Constant string; the instruction *const-string* is used to define string constants, which may contain malicious URLs and risky system commands. Call instruction: the call instructions *invoke-* $*$ represent the call relationship between two functions. These instructions are vitally important in call graph construction. Some call sequences could lead to the leakage of sensitive information.

Compare instruction: the compare instructions (e.g., *cmp* ∗) are used to compare two values. Jump instruction: the jump instructions (e.g., *if*∗ and *goto*∗) denote the transfer of program control flow, which is critical in capturing program execution semantic. Move instruction: the move instructions (e.g., *move* ∗) represent the data assignment between variables which may carry the semantic information of critical data transmission. Data declaration instruction: the data declaration instructions (e.g., *new-instance* and *instance-of*) are mainly used to define new variables. The number of the above instructions can be used to summarize the code characterization of the Smali function.

3. Design

CGDroid combines lightweight program analysis with the graph neural network to perform high-precision malware detection. It allows security analysts to efficiently identify malware in application markets and prevents the propagation of Android malware. To reduce resource consumption, we apply lightweight program analysis to generate approximate call graphs for responding to the rapid growth of malware in the real world. To ensure effectiveness, we adopt the graph neural network to automatically extract critical features from the call graph for malicious behavior identification. In the following, we first give the architecture of CGDroid and then introduce each component in detail.

3.1. CGDroid: Architecture. Figure 1 shows the architecture of CGDroid. CGDroid first leverages lightweight program analysis to extract approximate call graphs from both malicious and benign applications. Then, it summarizes intrafunction code characteristics (e.g., jump instructions, call instructions, and constants) as numerical values, which convert the call graph into an attributed call graph. The attributed call graph carries the function invocation information of the whole application and intrafunction code characteristics. Next, the GNN aggregates all information within the attributed call graph into graph embeddings. Finally, in the training phase, graph embeddings of both malicious and benign applications are fed into a multilayer perceptron (MLP) to train a malware classifier. In the detection phase, the built classifier could identify malware according to the embedding.

3.2. Call Graph Extraction. The construction of a precise call graph for an Android application requires the algorithm to handle the following program cases:

(i) *Indirect Call.* When a calling statement is executed, the actual callee function may be resolved at compilation time or runtime. In object-oriented languages, object polymorphic (a child class overloads its parent class's function) using the virtual function is an example of determining the actual callee function at runtime.

(ii) *Multiple Entry Points.* Unlike the Java program, Android applications usually contain multiple entry methods instead of starting from a specific main function [1]. These entry points are usually event handlers (e.g., the life-cycle functions of Android components) which are implicitly handled by the Android framework.

(iii) *Callback.* Due to the event-driver properties of the Android system, callbacks are prevalent within Android applications. Existing works such as FlowDroid [14] and DroidSafe [15] propose manually crafted lists or ad hoc interactive heuristics to handle the callback mechanism. In addition, EdgeMiner [16] performs a systematic static analysis to extract all registration-callback pairs within the Android framework and proves the effectiveness of the extracted pairs in improving the precision of static analysis. Precise call graph construction needs to correctly identify all the call relationships caused by the callback mechanism.

Given that the precise call graph construction and complex semantic information extraction are time consuming, the detection approach based on precision call graphs is not suitable for real-world large-scale Android malware detection. Therefore, we develop a lightweight program analysis algorithm based on Apktool [10] to construct approximate call graphs. The workflow of the algorithm is shown in the following steps: (a) decompiling the application into Smali files; (b) scanning each Smali file to extract all defined functions and constructing the node for each function; (c) traversing each call statement (e.g., "*invoke-* ∗") within each Smali file to identify all call relationships and then building the edge between caller and callee nodes according to these call relationships.

The approximate call graph contains all functions defined in the Smali code, which represents the function invocation logic within the whole application.

3.3. Function Attribute Extraction. To convert call graphs into structures that are applicable to machine learning, we need to extract numerical attributes that summarize intrafunction characteristics for each function node.

We leverage state-of-the-art NLP technique Word2Vec to extract semantic information of each function. We treat Smali instructions as words to extract semantic information and further assemble function-level embeddings. Hence, these embeddings contain the semantic information for functions. We normalize the Smali instruction to remove redundant code differences and eliminate the OOV problems. The generation of the function-level embedding consists of three phases, including normalization, Smali2Vec, and function embedding generation.

3.3.1. Normalization. In this phase, Smali files are scanned line by line to extract the Smali instruction sequence. Before sending the instruction sequence to train the Smali2Vec model, we simplify the Smali instruction to reduce the code differences that carry less semantic information. Dalvik bytecode contains more than 200 Smali instructions, while a

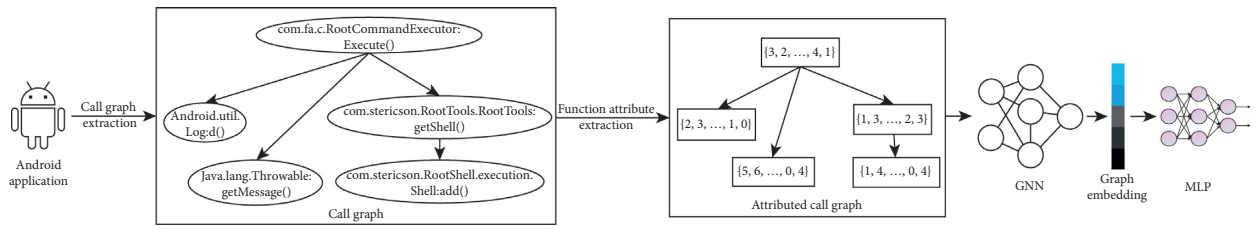

Figure 1: The system architecture of CGDroid.

majority of Smali instructions are similar with the same semantic information (e.g., invoke-direct and invoke-super), such as the operand type and length within operators. Although Smali instructions provide detailed information for the Android application, they significantly increase the burdens of bytecode analysis. Instruction normalization is widely used as a necessary step of bytecode analysis, such as dynamic information flow analysis [17], symbolic execution [18], and even malware detection [19]. As a result, CGDroid then performs normalization by using the following rules: (1) replacing the original instructions within one category as one representative instruction; (2) replacing all numeric constant values with the string "im"; (3) removing the definition of local variables.

Figure 2 shows an example of the Smali instruction normalization process. In Figure 2, the Smali instruction `invoke-super` in line 7 and `invoke-virtual` in line 10 belong to instruction category `invoke_`. The Smali instruction `sget-object` in line 8 belongs to instruction category `sget_`. According to the normalization rules, the original instructions are replaced with the instruction categories, and the local variables `v0`, `v1`, `p0`, and `p1` are removed. In particular, CGDroid only normalizes Smali instructions but retains any other metainformation, such as constant string, class name, superclass names, fields, and functions.

```
Bytecode
1.  class public Lcom/example/MainActivity;
2.  super Landroid/app/Activity;
3.  source "MainActivity.java"
4.  method public onCreate()
5.      registers 2
6.      parameter "saveInstanceState"
7       invoke-super {p0,p1}, Landroid/app/Activity;->onCreate(Landroid/os/Bundle;)V
8       sget-object v0, Ljava/lang/System;->out:Ljava/io/PrintStream;
9       const-string v1, "Test."
10      nvoke-virtual {v0, v1}, Ljava/io/PrintStream;->println(Ljava/lang/String;)V
11      return-void
12. end method
Normalized bytecode
1.  class public Lcom/example/MainActivity;
2.  super Landroid/app/Activity;
3.  source "MainActivity.java"
4.  method public onCreate()
5.      registers 2
6.      parameter "saveInstanceState"
7       invoke_ Landroid/app/Activity;->onCreate(Landroid/os/Bundle;)V
8       sget_Ljava/lang/System;->out:Ljava/io/PrintStream;
9       const-string_"Test."
10      invoke_Ljava/io/PrintStream;->println(Ljava/lang/String;)V
11      return-void
12. end method
```

Figure 2: Example of the Smali instruction normalization process.

3.3.2. Smali2Vec. In this phase, CGDroid represents each line of normalized Smali instruction with a dense and real-valued Smali vector, which captures instruction semantics via its local context. For a target line of the Smali instruction, its context represents the neighbor instructions within a preset window size in the same function. As shown in Figure 3, Smali2Vec consists of three steps, including instruction pairing, CBOW modeling, and instruction embeddings' generation.

Instruction Pairing. CGDroid firstly builds `instruction vocabulary` which contains Smali instructions from all known applications. Then, it generates instruction context pairs by matching the target instruction with its context. Let S_i be the i^{th} instruction in the instruction vocabulary and C represent the preset window size. For the target instruction S_i, its local context is the neighborhood instruction $\{S_{i,1}, S_{i,2}, \ldots, S_{i,C}\}$ including instructions ahead and instructions behind. Accordingly, CGDroid generates C instruction context pairs by matching S_i with each instruction in its context. Since the class names, superclass names, and file names contain important semantic information, CGDroid also merges these instructions into member functions and generates instruction context pairs. Note that since we generate embeddings for each function, CGDroid chooses functions as the granularity of generating instruction context pairs.

CBOW Modeling. With the generated instruction context pairs, CGDroid utilizes the Continuous Bag-of-Words (CBOW) model to train an instruction embedding model via taking each instruction pair as a

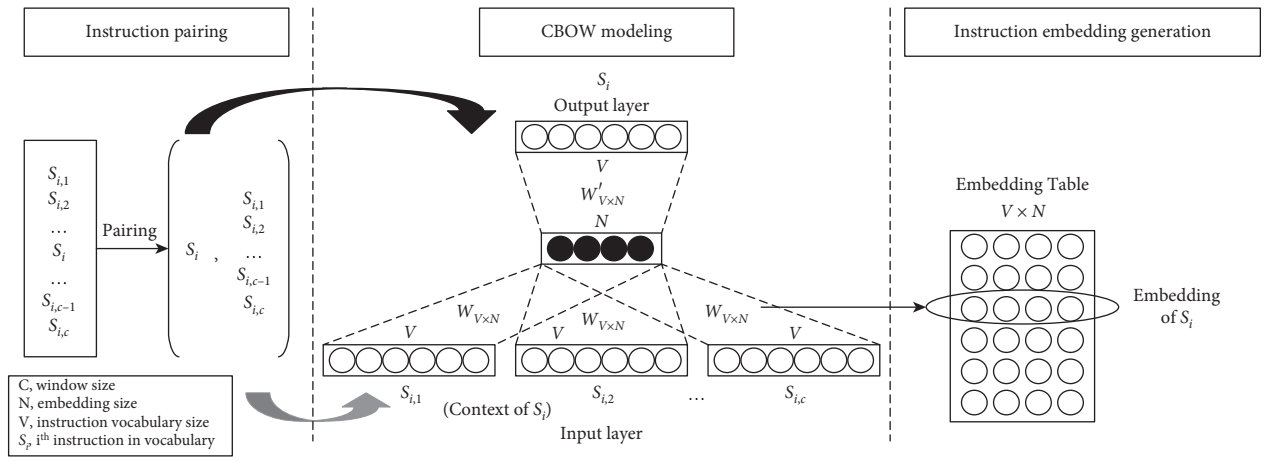

Figure 3: The workflow of Smali2Vec.

training instance. Note that the model only requires one-time training. In the training process, each instruction is in the one-hot encoding form and represented as a V-dimension vector with 1 at the index corresponding to the instruction and zeros in all other indexes. V denotes the size of instruction vocabulary. In particular, with S_i as the output and its context $\{S_{i,1}, S_{i,2}, \ldots, S_{i,C}\}$ as the input, the CBOW model projects the input and output into an embedding space, which learns the semantic representation for each instruction as a numerical vector. The objective of the CBOW model is to maximize the average log probability $J(S)$ as shown in the following:

$$J(S) = \frac{1}{V} \sum_{i=1}^{V} \sum_{j=0}^{C} \log\, p\left(S_{i,j} \mid S_i\right), \tag{1}$$

where C is the window size for the context and $p(S_{i,j} \mid S_i)$ is defined as

$$p\left(S_{i,j} \mid S_i\right) = \frac{\exp\left(\text{embed}_{S_{i,j}}^{T} \text{embed}_{S_i}\right)}{\sum_{j=1}^{C} \exp\left(\text{embed}_{S_{i,j}}^{T} \text{embed}_{S_i}\right)}, \tag{2}$$

where $\text{embed}_{S_{i,j}}$ and embed_{S_i} are the vector representations of $S_{i,j}$ and S_i.

After training the COBW model with instruction context pairs, CGDroid obtains two $V \times N$ weight matrices, W between the input layer and the hidden layer and W' between the hidden layer and the output layer, where V is the size of instruction vocabulary and N is the size of the embedding set by CGDroid. In particular, CGDroid chooses weight matrix W, which contains the embeddings of all instructions within instruction vocabulary. For example, the instruction embedding of S_i is the i^{th} row in W. Weight matrix W is then applied as the embedding table to search for the embedding given an instruction.

3.3.3. Function Embedding Generation. Function embeddings are then generated based on the Smali instruction embeddings. Since each function contains multiple instructions, we sum up all instructions within the function to formulate the function embeddings. In addition, instructions may not be of equal importance in terms of malware detection. For example, malware often uses `getDeviceId()` to collect sensitive information before leaking out, while function call `println()` is popular among Android applications to display debug information. In this case, instruction invoking `println()` is less important than the `getDeviceId()` instruction during detection. To tackle this problem, CGDroid adopts the TF-IDF model [20] to adjust the weights of instructions according to their importance. The calculated weight indicates the importance of the function that contains the instruction in all functions.

Particularly, for a function $f = \{S_1, S_2, \ldots, S_L\}$ containing L instructions, its function embeddings FE_f are the sum of its instructions, as depicted as

$$\text{FE}_f = \sum_{i=1}^{L} \left(\text{embed}_{S_i} \times \text{weight}_{S_i}\right), \tag{3}$$

where embed_{S_i} is the instruction embedding of S_i and weight_{S_i} is the TF-IDF weight importance of S_i.

More importantly, the generated function embeddings encode intrafunction code semantic information. If two different functions perform similar functionalities, they should generate similar function embeddings.

3.3.4. Sensitive Properties. We also extract sensitive properties of each function covering the following two types of attributes:

(i) *Security Level.* We categorize all functions into two types, critical API and normal API, which follow the criteria defined in [21]. Critical API contains sensitive sources and sinks [22] which operate on sensitive data, e.g., contact list or SMS messages, or requires key services, e.g., HTTPResponse; suspicious API calls [2], which are frequently used in malware samples, e.g., the execution of external commands (*Runtime.exec()*) and privilege elevation commands (*system/bin/su*); reflection functions and

dynamic code loading functions [23], which are popular functions adopted by malware. All other functions, including nonsensitive Android system API, API provided by Java libraries, and user-defined methods, are categorized as normal API.

(ii) *Required Permission*. Permission is an important security mechanism in the Android system. An Android application requires specific permissions to access system resources. Required permissions have always been a necessary feature in existing Android malware detection research [24, 25]. Hence, we build the maps between API calls and permissions according to Pscout [26] and consider the required permissions as function attributes.

After generating sensitive property attributes for each function, we concatenate the function embedding with the sensitive properties as the whole function attribute. For each function f, we thus obtain a security label SL_f, a permission set P_f, and a function embedding FE_f extracted via the Word2Vec technique. To keep as many raw attributes as possible, we apply a one-hot encoding scheme to convert the permission sets into vectors. Specifically, we denote $P = p_1, \ldots, p_{|P|}$ as the set of required permissions for all functions. Then, a transformation function $\phi(\cdot)$ is applied to convert the permission sets P_f into set vector x_f by

$$x_f^i = \phi(p_i) = \begin{cases} 1 & \text{if } p_i \in P_f, \\ 0, & \text{otherwise.} \end{cases} \tag{4}$$

The final permission set vector x_f is represented as $\{\ldots, x_f^i, \ldots\}$. Next, we concatenate function embedding FE_f, permission set vector x_f, and security label SL_f as function attribute FA_f:

$$\mathrm{FA}_f = \left(\mathrm{FE}_F \| x_f \| \mathrm{SL}_f\right). \tag{5}$$

The attributed call graph is constructed via replacing the function node in the call graph with the function attribute. Finally, these attributed call graphs are fed into the GNN to generate graph embeddings, which capture code characteristics and whole interaction structures of applications.

3.4. Graph Neural Network. In this paper, we exploit the graph representation of malware samples provided by the call graph and propose the malware detection approach based on call graphs in combination with the GNN. In the following, we describe how to apply the GNN for malware detection in the formal form.

After the processing of the *function attribute extraction* component, we obtain a number of call graphs with corresponding node attributes. The nodes within a call graph are actually functions. Each function has a numerical attribute including required permissions, certain security levels, and the semantic information of code characteristics. We define an attributed call graph as $G = (F, E)$, in which F represents the node set and E represents the edge set. All function attributes are combined as a new matrix X in which the f-th row is $x_f' = \mathrm{FA}_f$. We define A as the adjacent matrix of the attributed call graph G. Then, the GNN transforms the tuple (A, X) into graph embedding $z_G \in \mathbb{R}^d$, where d is the predefined embedding dimension of the graph. Finally, the MLP classifies the input z_G into category malware or benign.

Traditional approaches often carefully design feature engineering techniques and measure local information from A, which are inflexible and limited under the rapid evolving of Android malware samples. GNN attempts to learn the representation of the call graph by mapping entire graphs and node attributes into a vector space. Hence, we leverage the GNN to automatically learn critical features from call graphs.

Once we obtain the tuple (A, X), GNN would play the role of learning the embedding representation of the graph G. The intuition behind the GNN is to generate an embedding vector through interactively aggregating node vectors from its neighbor nodes. In this process, each node f is associated with a set of new hidden representations $\{\ldots, h_f^t, \ldots\}$, and the initial representation is $h_f = x_f'$. At layer $t+1$, h_f^t aggregates the t-layer hidden representations from its neighbors as follows:

$$h_f^{t+1} = \sigma\left(W^{t+1} \sum_{f' \in N_{(f)}} h_{f'}^t + h_f^t\right), \tag{6}$$

where N_f is the neighbor nodes of f, σ represents the activation function at layer t, and W^{t+1} denotes the weight matrix at layer $t+1$. After T rounds of iterative processing, we could obtain the T-th hidden representations for all nodes h_f^T. This iterative procedure implies the node features are propagating into deeper layers, and the final hidden representations depict local neighborhood information. We average all final hidden node representations as the embedding vector of graph G:

$$z_G = \frac{\sum_{f \in F} h_f^T}{|F|}. \tag{7}$$

After obtaining the graph-level representation z_G of the call graph, we use MLP to judge the category of this application. This process can be described as label = $\mathrm{MLP}(z_G)$, where label is whether malware or benign.

4. Experiments and Evaluation

In this section, we evaluate the performance of our proposed CGDroid approach on the Android malware detection task. In the following, we first introduce the dataset of malware and benign applications. After that, we compare CGDroid with a variety of existing machine learning-based malware detection approaches.

4.1. Experimental and Parameter Setup. In the following experiments, CGDroid is deployed on a computer with Intel(R) Xeon(R) E5-2620 CPU (2.4 GHz) and 16 GB of

RAM. We also conduct a set of experiments to validate the performance of CGDroid in this machine.

In this paper, we build a representative set of Android applications to capture critical information from approximate call graphs. For malicious applications, we first consider the Drebin dataset [2], which consists of 5560 malicious applications. We downloaded the same number of benign applications from the AndroZoo [27] dataset from May to September in 2016. Benign applications in AndroZoo have been detected by VirusTotal [28] to remove potential malware. Then, we obtain a dataset M consisting of 5560 benign and 5560 malicious applications. In the above dataset, all applications that failed to be analyzed by Apktool have been removed.

We now present the parameter settings within CGDroid. For Smali2Vec, the instruction embedding size N and window size C are set in such a way to take a balance between efficiency and effectiveness. Although the larger value of N and C allows the instruction embedding to contain more comprehensive semantic information of the Smali instruction, they also increase the complexities of the CBOW model and degrade the detection performance of the final malware detection model. Due to such consideration, we empirically set the instructions embeddings size N to 128 and the windows size C to 4 in our experiments. The weights of the CBOW model are initialized randomly and updated in the training process. For the GNN model, we set the iterative layer T to 3 and adopt ReLU as the activation function. Particularly, we set the graph embedding size as 32. For the MLP, the input layer is set to the same as the graph embedding size, the hidden layer is set to 8, and the output layer is set to 2.

4.2. Measurements and Metrics. We conduct tenfold cross-validation to measure the performance of CGDroid. The dataset M is randomly divided into ten equal-sized subsets. Each subset contains the same number of benign and malicious applications. Since hyperparameters within the GNN have significant influences on the detection performance, it is better to perform hyperparameter selection and evaluation on different datasets. To prevent overfitting and provide better generalization, CGDroid is trained and tested in ten rounds. In each round, eight subsets are treated as the training set for training the detection model. One subset is considered as the validation set for selecting hyperparameters. The remaining subset is then used for measuring the performance of the trained model. We repeat the above process ten times to ensure every application in M is trained and could obtain the detection result.

We evaluate the Android malware detection performance of different methods using the following measures: precision, recall, false positive rate (FPR), accuracy (ACC), and $F1$. These measures are derived from TN, TP, FP, and FN. TP denotes the count of malicious apps correctly detected. TN is the count of benign apps correctly identified. FP represents the count of benign apps being misclassified as malicious. FN denotes the count of malicious apps being classified as benign. The above measure metrics are calculated by the following:

$$\begin{aligned} \text{precision} &= \frac{\text{TP}}{\text{TP} + \text{FP}}, \\ \text{recall} &= \frac{\text{TP}}{\text{TP} + \text{FN}}, \\ \text{ACC} &= \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}}, \\ \text{FPR} &= \frac{\text{FP}}{\text{TN} + \text{FP}}, \\ F1 &= 2 \times \frac{\text{precision} \times \text{recall}}{\text{precision} + \text{recall}}. \end{aligned} \tag{8}$$

4.3. Feature Analysis. We analyze the approximate call graph and intrafunction code characteristics extracted by CGDroid. Table 1 shows the statistical information of the approximate call graphs in M. From Table 1, we observe that the node and edge distributions of Android applications earn a large span. Due to the diverse functions of Android applications, their size can range from KB to MB in M. In the dataset M, the largest file of the malicious application is 29 MB and of benign files is 78 MB. The diversity in application size will lead to the diversity in the size of the call graph, which will bring challenges for training the GNN and building graph embeddings. The approximate call graph still contains abundant information which can be explored by the graph neural network to perform malware detection.

In this paper, we empirically select 391 critical APIs to assign security levels. Some critical API calls and their function descriptions are listed in Table 2. From Table 2, we observe that these critical APIs correspond to common malicious behaviors, including accessing sensitive user information, dynamically loading malicious payload, and scanning the Android system to check whether antivirus is installed or running. In dataset M, the required permissions of functions almost cover all the Android system permissions. Table 3 shows the maximum number of intrafunction Smali instructions. There exists a big discrepancy in the implementation of different functions. Some initial functions (e.g., *init()*) only contain several data declaration instructions. Therefore, we list the maximum of instruction statistics to present the information within intrafunction code characteristics. Thus, we leverage Word2Vec to extract the semantic information including in Smali instructions and assemble all intrafunction Smali instructions as function attributes, which represent the semantic information and sensitive information of this function.

4.4. Call Graph Extraction Analysis. In this paper, we propose a lightweight call graph construction approach as described in Section 3.2, which sacrifices some precision for efficiency. We combine lightweight program analysis, NLP technique Word2Vec, and GNN to release resource

Table 1: The statistical information of the extracted approximate call graph on M.

Dataset	Drebin	Benign
# of samples	5560	5560
# of nodes (avg)	9352	31,875
# of nodes (max)	39,326	71,324
# of edges (avg)	17,541	42,243
# of edges (max)	130,601	27,651

Table 2: Critical APIs and their description.

Class name	API call name	Description
android.telephony.TelephonyManager	getDeviceId()	The access of unique device ID
android.telephony.TelephonyManager	getSubscriberId()	The access of unique subscriber device ID
android.net.wifi.WifiManager	setWifiEnabled()	The request of enable or disable Wi-Fi
java.lang.Runtime	exec()	The execution of a specific command
android.telephony.SmsManager	sendTextMessage	The sending of a text message
android.net.NetworkInfo	getExtraInfo()	The access of information about the network state
android.content.pm.PackageManager	getInstalledPackages()	The access of the installed package list
android.app.ActivityManager	getRunningServices()	The access of the running services' list
java.lang.ClassLoader	loadClass()	The dynamic load of the external class
android.app.ActivityManager	getMemoryInfo()	The access of system memory

Table 3: The statistics of intrafunction instructions on M.

Category	String constant	Call	Compare	Jump	Move	Data declaration
Count (max)	12	26	27	48	38	19

consumption caused by complex program analysis. FlowDroid proposes an interactive precision call graph construction algorithm to take into consideration the life-cycle interaction mechanism of Android components, callback mechanism, and indirect calls. In our experiment platform, the precision call graph construction of FlowDroid runs out of memory on most Android applications. Hence, we select 100 Android applications, in which precision call graphs can be constructed via FlowDroid. The time cost of lightweight call graph construction and precision call graph construction is shown in Table 4. From Table 4, our approach could significantly reduce the time cost of call graph construction. As described in FlowDroid, the precision approach builds a life-cycle model for each component, interactively includes all entry points and system callback in the call graph, and leverages point-to-analysis to parse indirect call. This interactive process is time consuming and resource consuming. Alternatively, CGDroid chooses to generate an approximate call graph via lightweight program analysis, which has more scalability and is suitable for real-world deployment. For the sacrificed precision, we leverage the NLP technique and GNN to perform effective malware detection via this approximate call graph.

4.5. Performance of Malware Detection. To evaluate our approach, we compare the performance against the detection method using static analysis without the graph structure. For the baseline approach, we take the approach in [2], which extracts features from the manifest and source codes via lightweight static analysis and proposes an open-source implementation. For CGDroid, we also replace the final MLP algorithm with the support vector machine (SVM), naive Bayes (NB), KNN, decision tree (DT), and random forest (RF) to illustrate the effectiveness of our graph representation approach. The experimental results are shown in Table 5. All machine learning algorithms are implemented in Python scripts via the *sklearn* [29] library. For SVM, we use the linear SVM in our experiment, and the penalty is set to 2. For KNN, the nearest neighbor is set to 5. For DT, we choose Gini coefficient as the criterion for feature selection. For RF, the maximum depth is set as 6 to take a tradeoff between time and performance.

From Table 5, we can clearly observe that our proposed GNN-based approach significantly outperforms the baseline approach by nearly 8.32% compared with the baseline method. After the generation of graph embedding, most of the learning approaches have a precision above 98%. However, without modeling the semantics of the intrafunction Smali instructions and using the GNN, it is difficult to obtain comparable performance to our GNN approach. For some complicated malware cases, the performance difference will be obvious.

The most significant improvement of the GNN is the FPR; Table 5 shows that our proposed method reduces the FPR from 6.7% to 0.9% and improves $F1$ from 91.37% to 99.33%, corresponding to nearly 50 fewer false alarms during the evaluation of 5560 samples. False alarms have always been a big concern in the security field and would cost considerable human efforts for a security analyst to get rid of them. Although NB provides a high detection rate of 99.91% as an MLP algorithm, it causes high FPR (5.12%), which diminishes the overall malware detection performance.

Table 4: Time cost comparison of call graph construction.

Approach	Time cost (avg.)	Time cost (max)
Lightweight approach	8.93	65.36
Precision call graph construction	15.73	112.84

Table 5: Malware detection performance of CGDroid.

Approach	Precision (%)	Recall (%)	FPR (%)	ACC (%)	*F*1 (%)
Baseline	91.10	91.64	6.7	91.20	91.37
CGDroid-KNN	97.86	98.56	1.16	98.20	98.21
CGDroid-SVM	98.32	98.90	1.69	98.60	98.61
CGDroid-NB	98.67	98.08	1.33	98.78	98.88
CGDroid-DT	98.85	99.08	1.12	98.20	98.97
CGDroid-RF	98.90	99.12	1.10	98.92	98.99
CGDroid	99.21	99.44	0.90	99.22	99.33

We plot the ROC curve to evaluate the detection performance of CGDroid, as shown in Figure 4. The ROC curve is another way to measure the overall performance of a classifier and is created by plotting true positive rate against false positive rate as the discrimination threshold of the classifier varies. From Figure 4, we find that CGDroid achieves good performance and outperforms the baseline methods.

We further present the malware detection performance of CGDroid with the different number of training subsets, covering {1, 2, 4, 8} subsets. The detection results are shown in Table 6. From Table 6, we observe that the number of training subsets slightly affects the malware detection performance of CGDroid. This variant rule reflects that the performance of malware detection increases with the amount of the training data.

The effectiveness of our proposed approach can be attributed to the two characteristics. First, Android applications are transformed into call graphs, which provide a detailed interaction view of the whole application such as data flow. However, traditional malware detection methods only consider static statistical features, which cannot capture enough information about the application. Second, the function attributes are more expressive in our model. The Smali2Vec model is able to capture the semantic information of the intrafunction Smali instructions. The intrafunction semantic information and interfunction interaction information can be exploited by our proposed method, while traditional methods lack this information.

Our graph representation approach can greatly improve the effectiveness of malware detection on static analysis. Meanwhile, in our approximate call graph scene, the MLP algorithm achieves the highest detection performance. This illustrates that our proposed approach is able to extract critical information from the approximate call graph to perform effective malware detection. Furthermore, the graph neural network is promising in malware detection and other security-related problems.

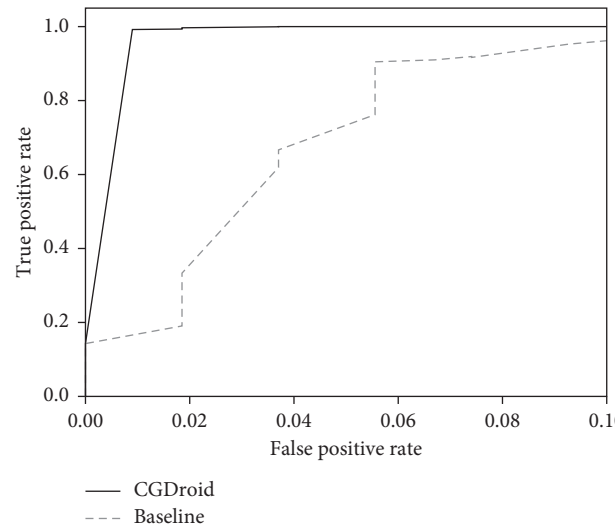

Figure 4: Detection performance as the ROC curve.

Table 6: Malware detection performance of CGDroid with different numbers of training subsets.

# of subsets	Precision (%)	Recall (%)	FPR (%)	ACC (%)	*F*1 (%)
1	99.01	98.83	1.16	98.92	98.92
2	91.10	98.92	1.06	99.01	99.01
4	99.12	99.01	0.98	99.05	98.06
8	99.21	99.44	0.90	99.22	99.33

CGDroid combines lightweight program analysis with the GNN to perform malware detection, which causes some time and memory consumption. We further figure out the average time and memory cost by each step of our approach, especially the call graph extraction and graph embedding, in order to estimate the resource consumption on detecting any unknown application. The average time and memory cost of each step are listed in Table 7. Clearly, the time costs of function attribute extraction and MLP classification are ignoble. The call graph extraction and application disassembly occupy most of the time consumption. The average time of call graph extraction, indicating the construction of an approximate call graph via lightweight program analysis, is 8.92 s. The graph embedding and call graph extraction occupy most of the memory cost. Generally, the time and the memory cost are positively correlated with the size of applications. In summary, the time and the memory cost of CGDroid are reasonable.

4.6. Detection of Malware Families. In this experiment, we also evaluate the malware family classification performance of CGDroid on dataset M. In M, malicious applications belong to well-known malware families, such as DroidKungFu and GoldDream [30, 31]. We present each of the top 20 largest amounts of malware family detection performance in Table 8.

From Table 8, we observe that the overall family detection accuracy of CGDroid is 97.1%. This illustrates that

Table 7: Time and memory costs of each step of CGDroid.

Step	Disassembly	Call graph extraction	Function attribute extraction	Graph embedding	MLP classification
Avg. time (s)	3.76s	8.93	3.58×10^{-3}	0.26	1.63×10^{-3}
Avg. memory (MB)	15.2	26.8	0.4	63.5	7.8

Table 8: Malware family detection performance.

Malware family	# of samples	# of detected samples	Accuracy (%)
FakeInstaller	925	913	98.7
DroidKungFu	662	655	98.9
Opfake	612	603	98.5
Plankton	553	527	95.3
GinerMaster	338	331	98.0
BaseBridge	315	308	97.8
Iconosys	135	131	97.0
FakeDoc	132	124	93.9
Kmin	96	96	100.0
Geinimi	84	81	96.4
Adrd	82	78	95.1
DroidDream	78	75	96.2
MobileTx	69	66	95.7
LinuxLotoor	68	65	95.6
GoldDream	68	67	98.5
FakeRun	61	57	98.5
SendPay	59	59	100
Gappusin	46	25	54.3
Imlog	43	40	93.0
SMSreg	40	37	92.5
Total	4466	4338	97.1

CGDroid could accurately capture the critical malicious behaviors among malware families and correctly assign a family label to 97 out of 100 malware samples. It is also observed that malware detected by CGDroid is uniformly distributed in malware families. This experiment further proves that intrafunction semantic information and interfunction structure information can be effective in Android malware family detection.

From Table 8, we also observed that two families (Kmin and SendPay) implement the perfect identification, and other families show a detection rate of more than 92%. When looking into the results on the malware family Gappusin, the result is quite poor. CGDroid only correctly identifies 25 out of 46 malware samples. After inspecting samples within this family, we find that these samples act as a client to receive a malicious payload from the C&C server and do not exhibit other malicious behaviors. The access of external malicious server behaviors within the Gappusin family contains too few features to identify these samples as malicious.

4.7. Comparison with the Existing Approach. In order to prove the effectiveness of our proposed approach, we also compare CGDroid with existing malware detection methods based on static analysis. The comparison results are shown in Figure 5. From Figure 5, we observe that our GNN-based detection model has the highest detection rate. Existing researchers have proposed detection approaches based on API call [4] and even call traces [1] from entry points to the API call to perform malware detection. These approaches all require security analysts to carefully design critical features. This process, even for experts, is a tedious, subjective, and sometimes error-prone task. In this paper, CGDroid explores the direction of Android malware based on graph representation learning and leverages the graph neural network to automatically capture critical information from the call graph. For Droidetec [32], it leverages deep learning to extract critical information from API call sequences for malware detection. Alternatively, CGDroid leverages Word2Vec to extract semantic information from intrafunction code characteristics and the GNN to extract structure information from the function interaction relationship, which could implement more accurate malware detection.

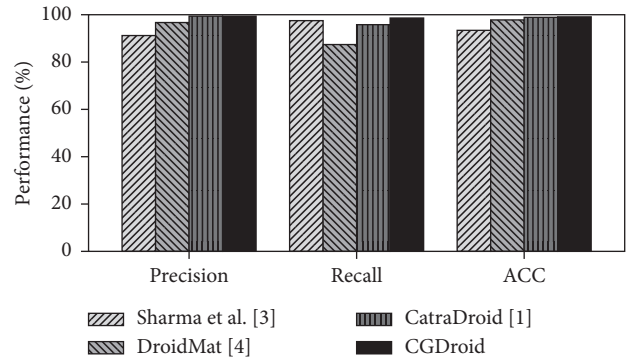

Figure 5: The detection performance comparison with existing approaches.

5. Discussion

In this paper, we leverage Apktool to disassemble Android applications. However, some applications leverage code obfuscation techniques such as packing, dynamic code loading, and bytecode encryption to evade static analysis. These applications cannot be handled by CGDroid for the inability to extract intrafunction Smali instructions and construct an approximate call graph. Another limitation is that CGDroid constructs an approximate call graph to cover more call relationships between functions which sacrifice some precision for large-scale malware detection. Some malware may hide malicious codes through reflection, implicit callback, and implicit control flow, which cannot be captured by our approximate call graph. Thus, CGDroid may not correctly identify these applications. Actually, CGDroid is trying to explore the direction of combining the NLP technique and deep learning algorithm to automatically understand the program language and remedy the

performance deficiency caused by the complex program analysis algorithm. In this paper, we automatically extract intrafunction semantic information and interfunction structure information via Word2Vec and GNN techniques to perform effective malware detection. In the future, we plan to integrate other program language embedding algorithms, code2vec [33], into our framework to handle advanced malware.

As malware continues to evolve, the performance degradation of learning-based malware detection becomes a great challenge. Researchers have proposed methods to address the performance of degradation and improve the sustainability of learning-based malware detection techniques. Jordaney et al. [34] proposed a conformal evaluator to identify the concept drift problems within aging classification models. Then, case studies were performed on detecting decay of binary and multiclass classification performance. To avoiding frequently retraining Android malware detectors when resisting newly emerged malware, Cai and Jenkins [35] investigated the evolution behaviors of benign and malicious apps over time and discovered the most consistently discriminating behavioral traits, distribution of sensitive access. These behaviors can be used to build a sustainable Android malware detector. Fu and Cai [36] proposed the hypothesis that the deterioration of malware detectors causes their inability in capturing new malware. Then, they verified this hypothesis on state-of-the-art Android malware detectors and developed a new detector based on the evolutionary characterization of applications. Xu et al. [37] proposed an Android malware detection system DroidEvolver, which could automatically and continually update itself through online learning techniques with evolving feature set. The experiment result shows that DroidEvolver is able to handle concept drifting during the change of Android frameworks and superior to existing solutions in terms of effectiveness and efficiency. Cai [38] studied the sustainability problem for learning-based app classifiers and proposed a novel classification system based on the lightweight behavioral profile from the sensitive access distribution. The experimental results show the proposed approach outperforms existing solutions in maintaining sustainability. Aiming at mitigating the degradation in detection performance caused by the evolution of malware, Zhang et al. [9] proposed to build API relation graphs from official Android documents to enhance state-of-the-art malware classifiers. The approach captures the semantically equivalent or similar API usages from evolved malware and thus slows down the classifier aging problems. As the training dataset changes, the sustainability problem is the intrinsic attribute within the learning-based approach. We plan to explore more invariant graph features to improve the scalability of the GNN-based malware detection approach in the future.

6. Related Work

In order to continuously fight against the rapid development and evolution of Android malware, there have been a large number of academic research studies on analyzing and detecting malware prior to installation via machine learning and static analysis [1, 2, 32]. Drebin [2] combines comprehensive static features and linear SVM to perform explainable malware detection. Mudflow [8] observes that malware can be detected through the abnormal usage of sensitive data. It represents the usage of sensitive data as statically extracted sensitive data flows and builds an outlier detection model based on these data flows. DroidSieve [5] explores the obfuscation-invariant features and artifacts inspired from common obfuscation mechanisms. The invariant features consist of used resource-derived features and code and metadata-derived features. The paper shows that the detection system based on invariant features remains resilient against state-of-the-art obfuscation techniques. Fan et al. [7] aimed at classifying the polymorphic variants of Android malware into families and proposed to construct frequency-sensitive API relation graphs to capture common behaviors of malware samples within the same family. Zhang et al. [6] proposed a hybrid-feature malware detection technique that integrates two types of features, opcodes and API calls. The proposed approach leverages conventional neural networks and a backpropagation neural network to encode opcode sequences and API call sequences, separately. Although this paper aimed at detecting Windows malware, the feature fusion framework can be used for Android malware detection and other security domains, such as instruction detection and antifraud. Kumar et al. [39] proposed a new explainable detection approach that builds an Android malware detector based on app permissions and static code features using machine learning.

Some researchers have proposed to combine dynamic analysis and machine learning to perform malware detection [40–42]. Afonso et al. [40] proposed effective dynamic malware detection based on features extracted from API calls and system call traces. They built a portable dynamic analysis platform via APK instrumentation and a built-in system call hooking mechanism. DroidScribe [41] proposes a multiclass malware family classification method based on dynamic analysis. It points out that the lower-level system call cannot capture the semantic patterns of malware. Then, it proves that the reconstructed binder calls and parameters, high-level file and network operations, and conformal prediction can be used to improve the classification performance via experiment results on a real-world dataset. Droidcat [42] proposes a novel dynamic malware classification approach based on a diverse set of dynamic features. By using the structure and security features from method calls and intercomponent communication features, the proposed approach is verified to be resilient towards analyzing challenges including complex reflection, resource obfuscation, system call obfuscation, and the use of runtime permission.

Researchers have adopted deep neural network-based detection methods to defend the complex Android malware. Kim et al. [43] proposed a novel Android malware detection framework. It leverages the multimodal deep learning method to encompassing various existence-based or similarity-based features. α-Cyber [44] points out that

heterogeneous graph- (HG-) based Android malware detection systems have demonstrated success. However, incentivize attackers could explore ways to defeat HG-based models to bypass the detection. Then, a novel practical adversarial attack model on HG data is proposed, and a resilient yet elegant defense model to enhance the robustness of the HG-based Android malware classifier is further proposed. Huang et al. [45] explored the API features based on Android malware detection methods. They selected critical API related to the malware class, discovered structure relationships between APIs, and built an effective CNN-based malware detector based on these relationships. Feng et al. [46] pointed out that server-side Android malware detection cannot provide sufficient protection for end-users. Apps from unofficial and third-party markets are still causing serious security threats. In addition, the uploading process of server-side detection also suffers from the security threats of attackers. Then, a customized deep neural network-based real-time detection system on mobile devices is built. Yuan et al. [47] pointed out that the in-cloud Android malware detection suffers from privacy leakage and communication overheads and then proposed a multilayer neural network-based Android malware detector that can be fully and incrementally trained directly on mobile devices. Han et al. [48] proposed a novel feature transformation-based Android malware detector. It leverages three new types of feature transformations to irreversibly transform well-known critical features into a new feature domain. Then, a robust detector is built using this new feature domain, which is resilient to the ML classifier evading attacks. Similarly, AMalNet [49] also proposes an Android malware detection system based on word embedding and graph convolutional networks. It builds the unique graph relation based on the relative position and leverages an independent recurrent neural network to improve the detection performance. Differently, CGDroid adopts the graph neural network directly on the application's call graph to capture critical patterns.

In this paper, we propose CGDroid, which leverages the graph neural network [13] to automatically extract critical structure information from the approximate call graph and adopts Word2Vec to extract semantic information from intrafunction Smali instructions. In this way, a malware detection approach can be constructed without feature engineering-related domain knowledge from security experts.

As the popularity of Android IoT devices, researchers have proposed approaches to prevent the spread of malware within IoT networks [50–52]. Kumar et al. [50] proposed a novel approach based on runtime risky permission to distinguish malware and benign applications. Kumar et al. [51] enhanced the clustering-based feature selection algorithm and multifeature naive Bayes algorithm to provide high-accuracy and robust IoT malware detection. Kumar et al. [52] combined the blockchain and deep learning model to provide real-time malware activity detection for Android IoT devices. They adopted a multilevel deep learning model to extract multiple types of malware features and customized smart contracts to provide malicious activity sharing among IoT networks.

7. Conclusion

In this paper, we present an Android malware detection framework, called CGDroid, which could automatically extract critical information from the call graph via the graph neural network. Instead of using API calls or API call sequences, we utilize lightweight static analysis to extract approximate call graphs and intrafunction semantic information to represent Android applications. This representation form not only provides a high-level call relationship of the whole application but also covers detailed Smali instruction information, making it difficult for attackers to evade the detection. Based on the extracted features, we present a new malware detection framework based on the graph neural network and incorporate intrafunction semantic information from Smali instructions. Experimental results on real-world application samples show that our approach achieves high effective malware detection and outperforms state-of-the-art approaches.

Data Availability

The dataset used in our paper is available from the corresponding author upon request for research purpose. The benign application can be accessed via AndroZoo [1]. The malicious application can be accessed via Drebin [2]. [1] K. Allix, T. F. Bissyande, 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. [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 Ndss, vol. 14, 2014, pp. 23–26.

Disclosure

This manuscript is an extension of a conference paper published in the proceedings of the 2020 International Conference on Networking and Network Applications (NANA 2020). The conference paper is the NANA 2020 Conference's recommendation paper and recommended to this SCI journal *Mobile Information Systems*. The authors substantially extended the conference version of the paper with new contributions.

Conflicts of Interest

The authors declare that there are no conflicts of interest regarding the publication of this article.

Acknowledgments

This research was funded by the Fundamental Research Funds for the Central Universities (nos. XJS201503, JB191507, and JB191508), National Natural Science Foundation of China (Grant nos. 61902290, 61902291, 61872283, 62072359, and 62072352), China Postdoctoral Science Foundation Funded Project (2019M653567), Key Research and Development Program of Shaanxi (Grant nos.

2020ZDLGY09-06 and 2019ZDLGY12-04), and Natural Science Foundation of Shaanxi Province (Grant no. 2019JM-109).

References

[1] C. Sun, J. Chen, P. Feng, and J. Ma, "CatraDroid: a call trace driven detection of malicious behaiviors in android applications," in *Proceedings of the International Conference on Machine Learning for Cyber Security*, pp. 63–77, Springer, Xi'an, China, September 2019.

[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 *Proceedings of the 2014 Network and Distributed System Security Symposium*, vol. 14, pp. 23–26, San Diego, CA, USA, February 2014.

[3] A. Sharma and S. K. Dash, "Mining api calls and permissions for android malware detection," in *Proceedings of the International Conference on Cryptology and Network Security*, pp. 191–205, Springer, Heraklion, Crete, Greece, October 2014.

[4] D.-J. Wu, C.-H. Mao, T.-E. Wei, H.-M. Lee, and K.-P. Wu, "Droidmat: android malware detection through manifest and api calls tracing," in *Proceedings of the 2012 Seventh Asia Joint Conference on Information Security*, pp. 62–69, IEEE, Tokyo, Japan, August 2012.

[5] G. Suarez-Tangil, S. K. Dash, M. Ahmadi, J. Kinder, G. Giacinto, and L. Cavallaro, "Droidsieve: fast and accurate classification of obfuscated android malware," in *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, pp. 309–320, Scottsdale, AZ, USA, March 2017.

[6] J. Zhang, Z. Qin, H. Yin, L. Ou, and K. Zhang, "A feature-hybrid malware variants detection using CNN based opcode embedding and BPNN based api embedding," *Computers & Security*, vol. 84, pp. 376–392, 2019.

[7] M. Fan, J. Liu, X. Luo et al., "Android malware familial classification and representative sample selection via frequent subgraph analysis," *IEEE Transactions on Information Forensics and Security*, vol. 13, no. 8, pp. 1890–1905, 2018.

[8] V. Avdiienko, K. Kuznetsov, A. Gorla et al., "Mining apps for abnormal usage of sensitive data,," in *Proceedings of the 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1, pp. 426–436, IEEE, Florence, Italy, May 2015.

[9] X. Zhang, Y. Zhang, M. Zhong et al., "Enhancing state-of-the-art classifiers with api semantics to detect evolved android malware," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pp. 757–770, Virtual Event, USA, November 2020.

[10] C. Tumbleson and R. Wiśniewski, "Apktool—a tool for reverse engineering android apk files," 2020, https://ibotpeaches.github.io/Apktool/.

[11] H. J. Kang, T. F. Bissyandé, and D. Lo, "Assessing the generalizability of Code2vec token embeddings," in *Proceedings of the 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 1–12, IEEE, San Diego, CA, USA, November 2019.

[12] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," 2013, https://arxiv.org/abs/1310.4546.

[13] Z. Zhang, P. Cui, and W. Zhu, "Deep learning on graphs: a survey," *IEEE Transactions on Knowledge and Data Engineering*, vol. 1, 2020.

[14] S. Arzt, S. Rasthofer, C. Fritz et al., "FlowDroid," *Acm Sigplan Notices*, vol. 49, no. 6, pp. 259–269, 2014.

[15] M. I. Gordon, D. Kim, J. H. Perkins, L. Gilham, N. Nguyen, and M. C. Rinard, "Information flow analysis of android applications in DroidSafe," in *Proceedings of the 2015 Network and Distributed System Security Symposium*, vol. 15, p. 110, San Diego, CA, USA, February 2015.

[16] Y. Cao, Y. Fratantonio, A. Bianchi et al., "Automatically detecting implicit control flow transitions through the android framework," in *Proceedings of the 2015 Network and Distributed System Security Symposium*, San Diego, CA, USA, February 2015.

[17] M. Sun, T. Wei, and J. C. Lui, "Taintart: a practical multi-level information-flow tracking system for android runtime," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pp. 331–342, Vienna, Austria, October 2016.

[18] J. Jeon, K. K. Micinski, and J. S. Foster, "Symdroid: symbolic execution for dalvik bytecode," Technical Report, University of Maryland, College Park, MD, USA, 2012.

[19] 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*, pp. 45–54, Berlin, Germany, November 2013.

[20] K. S. Jones, "A statistical interpretation of term specificity and its application in retrieval," *Journal of Documentation*, vol. 28, no. 1, pp. 11–21, 1972.

[21] W. Yang, X. Xiao, B. Andow, S. Li, T. Xie, and W. Enck, "Appcontext: differentiating malicious and benign mobile app behaviors using context,"vol. 1, pp. 303–313, in *Proceedings of the 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1, pp. 303–313, IEEE, Florence, Italy, May 2015.

[22] S. Rasthofer, S. Arzt, and E. Bodden, "A machine-learning approach for classifying and categorizing android sources and sinks," in *Proceedings of the 2014 Network and Distributed System Security Symposium*, vol. 14, p. 1125, Citeseer, San Diego, CA, USA, February 2014.

[23] Y. Xue, G. Meng, Y. Liu et al., "Auditing anti-malware tools by evolving android malware and dynamic loading technique," *IEEE Transactions on Information Forensics and Security*, vol. 12, no. 7, pp. 1529–1544, 2017.

[24] J. Li, L. Sun, Q. Yan, Z. Li, W. Srisa-An, and H. Ye, "Significant permission identification for machine-learning-based android malware detection," *IEEE Transactions on Industrial Informatics*, vol. 14, no. 7, pp. 3216–3225, 2018.

[25] L. Sun, Z. Li, Q. Yan, W. Srisa-an, and Y. Pan, "Sigpid: significant permission identification for android malware detection," in *Proceedings of the 2016 11th International Conference on Malicious and Unwanted Software (MALWARE)*, pp. 1–8, IEEE, Fajardo, PR, USA, October 2016.

[26] 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*, pp. 217–228, Raleigh, NC, USA, October 2012.

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

[28] VirusTotal, "Virustotal—a free virus, malware and url online scanning service," 2020, https://www.virustotal.com/.

[29] Scikit-learn machine learning in python, http://scikit-learn.org/stable/.
[30] X. Jiang, "Security alert: Golddream," 2011, https://www.csc2.ncsu.edu/faculty/xjiang4/GoldDream/.
[31] "Security alert: New droidkungfu variant," 2011, https://www.csc2.ncsu.edu/faculty/xjiang4/GingerMaster/.
[32] Z. Ma, H. Ge, Z. Wang, Y. Liu, and X. Liu, "Droidetec: android malware detection and malicious code localization through deep learning," 2020, https://arxiv.org/abs/2002.03594.
[33] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: learning distributed representations of code," *Proceedings of the ACM on Programming Languages*, vol. 3, pp. 1–29, 2019.
[34] R. Jordaney, K. Sharad, S. K. Dash et al., "Detecting concept drift in malware classification models," in *Proceedings of the 26th {USENIX} Security Symposium ({USENIX} Security 17)*, pp. 625–642, Vancouver, BC, Canada, August 2017.
[35] H. Cai and J. Jenkins, "Towards sustainable android malware detection," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, pp. 350-351, New York, NY, USA, May 2018.
[36] X. Fu and H. Cai, "On the deterioration of learning-based malware detectors for android," *In*, IEEE, in *Proceedings of the 2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pp. 272-273, IEEE, Montreal, Quebec, Canada, May 2019.
[37] K. Xu, Y. Li, R. Deng, K. Chen, and J. Xu, "Droidevolver: self-evolving android malware detection system," in *Proceedings of the 2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, pp. 47–62, IEEE, Stockholm, Sweden, June 2019.
[38] H. Cai, "Assessing and improving malware detection sustainability through app evolution studies," *ACM Transactions on Software Engineering and Methodology*, vol. 29, no. 2, pp. 1–28, 2020.
[39] R. Kumar, Z. Xiaosong, R. U. Khan, J. Kumar, and I. Ahad, "Effective and explainable detection of android malware based on machine learning algorithms," in *Proceedings of the 2018 International Conference on Computing and Artificial Intelligence*, pp. 35–40, Chengdu, China, March 2018.
[40] V. M. Afonso, M. F. de Amorim, A. R. A. Grégio, G. B. Junquera, and P. L. de Geus, "Identifying android malware using dynamically obtained features," *Journal of Computer Virology and Hacking Techniques*, vol. 11, no. 1, pp. 9–17, 2015.
[41] S. K. Dash, G. Suarez-Tangil, S. Khan et al., "Classifying android malware based on runtime behavior," in *Proceedings of the 2016 IEEE Security and Privacy Workshops (SPW)*, pp. 252–261, IEEE, San Jose, CA, USA, May 2016.
[42] H. Cai, N. Meng, B. Ryder, and D. Yao, "Droidcat: effective android malware detection and categorization via app-level profiling," *IEEE Transactions on Information Forensics and Security*, vol. 14, no. 6, pp. 1455–1470, 2018.
[43] 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.
[44] S. Hou, Y. Fan, Y. Zhang et al., "α cyber: 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*, pp. 609–618, Beijing, China, November 2019.
[45] N. Huang, M. Xu, N. Zheng, T. Qiao, and K.-K. R. Choo, "Deep android malware classification with api-based feature graph," in *Proceedings of the 2019 18th IEEE International Conference on Trust, Security and Privacy in Computing and Communications/13th IEEE International Conference on Big Data Science and Engineering (TrustCom/BigDataSE)*, pp. 296–303, IEEE, Rotorua, New Zealand, 2019.
[46] R. Feng, S. Chen, X. Xie, G. Meng, S.-W. Lin, and Y. Liu, "A performance-sensitive malware detection system using deep learning on mobile devices," *IEEE Transactions on Information Forensics and Security*, vol. 16, pp. 1563–1578, 2020.
[47] W. Yuan, Y. Jiang, H. Li, and M. Cai, "A lightweight on-device detection method for android malware," *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, vol. 1, pp. 1–12, 2019.
[48] Q. Han, V. S. Subrahmanian, and Y. Xiong, "Android malware detection via (somewhat) robust irreversible feature transformations," *IEEE Transactions on Information Forensics and Security*, vol. 15, pp. 3511–3525, 2020.
[49] X. Pei, L. Yu, and S. Tian, "Amalnet: a deep learning framework based on graph convolutional networks for malware detection," *Computers & Security*, vol. 93, Article ID 101792, 2020.
[50] R. Kumar, X. Zhang, R. Khan, and A. Sharif, "Research on data mining of permission-induced risk for android iot devices," *Applied Sciences*, vol. 9, no. 2, p. 277, 2019.
[51] R. Kumar, X. Zhang, W. Wang, R. U. Khan, J. Kumar, and A. Sharif, "A multimodal malware detection technique for android iot devices using various features," *IEEE Access*, vol. 7, pp. 64 411–64 430, 2019.
[52] R. Kumar, W. Wang, J. Kumar et al., "Iotmalware: android IoT malware detection based on deep neural network and blockchain technology," 2021, https://arxiv.org/abs/2102.13376.