2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)

Graph Embedding based Familial Analysis of Android Malware using Unsupervised Learning

Ming Fan*†, Xiapu Luo†, Jun Liu*, Meng Wang‡, Chunyin Nong*, Qinghua Zheng*, Ting Liu*
*School of Electronic and Information Engineering, Xi'an Jiaotong University, 710049, China
†Department of Computing, The Hong Kong Polytechnic University, China
‡School of Computer Science and Engineering, Southeast University, China
fanming.911025@stu.xjtu.edu.cn; luoxiapu@gmail.com; liukeen@xjtu.edu.cn;
meng.wang@seu.edu.cn; cynong@foxmail.com; qhzheng@xjtu.edu.cn; tingliu@xjtu.edu.cn

***Abstract*—The rapid growth of Android malware has posed severe security threats to smartphone users. On the basis of the familial trait of Android malware observed by previous work, the familial analysis is a promising way to help analysts better focus on the commonalities of malware samples within the same families, thus reducing the analytical workload and accelerating malware analysis. The majority of existing approaches rely on supervised learning and face three main challenges, i.e., low accuracy, low efficiency, and the lack of labeled dataset. To address these challenges, we first construct a fine-grained behavior model by abstracting the program semantics into a set of subgraphs. Then, we propose SRA, a novel feature that depicts the similarity relationships between the Structural Roles of sensitive API call nodes in subgraphs. An SRA is obtained based on graph embedding techniques and represented as a vector, thus we can effectively reduce the high complexity of graph matching. After that, instead of training a classifier with labeled samples, we construct malware link network based on SRAs and apply community detection algorithms on it to group the unlabeled samples into groups. We implement these ideas in a system called GefDroid that performs Graph embedding based familial analysis of AnDroid malware using unsupervised learning. Moreover, we conduct extensive experiments to evaluate GefDroid on three datasets with ground truth. The results show that GefDroid can achieve high agreements (0.707-0.883 in term of NMI) between the clustering results and the ground truth. Furthermore, GefDroid requires only linear run-time overhead and takes around 8.6s to analyze a sample on average, which is considerably faster than the previous work.**

***Keywords*-Android malware, graph embedding, familial analysis, unsupervised learning**

I. Introduction

With the rapid development of smartphones, mobile applications (apps) have become an inherent part of our everyday life since many convenient services are provided to us through mobile apps. Unfortunately, Android, the most popular mobile operating system, has become the major target of 97% mobile malware [1]. A new security report showed that about 7.57 million malware samples were captured in 2017 [2]. Such malware has posed severe security threats to smartphone users.

Many recent studies [3]–[5] reveal that the Android malware exhibits obvious familial trait because attackers usually create malware by injecting the similar malicious components into different popular apps. In other words, malware samples within the same family have similar malicious behaviors. Similar to the clone detection [6]–[9] that discovers similar fragments between code snippets, familial analysis is a promising way to identify the common malicious components among malware samples within the same families, thus reducing the analytical workload and accelerating malware analysis.

Most existing familial analysis approaches rely on supervised learning that first trains a classifier using labeled dataset and then use it to classify new malware samples. They differ in their features, which could be roughly divided into two categories, including (1) string-based features (e.g, permissions [10] and API calls [11]); (2) graph-based features (e.g., function call graph (FCG) [12] and control flow graph (CFG) [13]). However, these approaches have three main limitations as follows.

Low accuracy: String-based features are insufficient to distinguish the malicious components and the legitimate part of popular apps. For example, the API `getLastKnownLocation()` for obtaining the location information is widely used in both malware samples and benign apps [14].

Low efficiency: Although graph-based features could profile the behaviors of malware samples, the similarity calculation of the graph-based features is bounded by the efficiency of existing graph matching approaches [15], which are usually slow since the graph isomorphism problem is NP complete.

Lack of labeled dataset: It is time-consuming and labor-intensive to label a large scale of malware samples with family names. Moreover, since classifiers are trained using known malware samples, they cannot correctly classify new malware samples from unknown families. Note that retraining the classifier model for every new sample may be impractical [16].

To tackle these challenges, we propose GefDroid, a novel **G**raph **e**mbedding based **f**amilial analysis approach of An**Droid** malware with the following salient features:

High accuracy: To achieve a high accuracy of familial analysis, we use the graph-based features that contain the structural information to profile the app behaviors rather than the string-based features. Specifically, we abstract the program semantics of an app into an FCG representation, and further divide the FCG into a set of small subgraphs according to the app's file directory structure. By doing so, a fine-grained

Corresponding author: Ting Liu.

1558-1225/19/$31.00 ©2019 IEEE
DOI 10.1109/ICSE.2019.00085

behavior model is constructed in order to locate the malicious components when performing familial analysis.

High efficiency: To reduce the high complexity of directly using graph matching, inspired by the graph embedding techniques that can transform the high-dimensional graph structure data into low-dimensional space, we propose a novel feature called SRA to depict the similarity relationships of **S**tructural **R**oles of sensitive **A**PI call nodes in a graph. The structural roles refer to the graph position and the structure of local graph neighborhood. Specifically, we employ graph embedding techniques to learn low-dimensional vector representations for graph nodes, and then calculate the SRA based on the vector representations of sensitive API call nodes. Finally, the similarity computation of two graphs is simplified to the similarity comparison between two vectors based on the generated SRAs instead of the high-cost graph matching.

No need of labeled dataset: Instead of training a classifier, we leverage unsupervised learning to cluster unlabeled samples according to their similarity. In particular, we construct a malware link network (MLN) to represent the similarity relationships among samples based on their similar SRAs. Then, we apply community detection algorithms to group the samples into a set of clusters.

After applying GefDroid to three widely-used datasets, we find that it exhibits impressive performance of familial analysis. In summary, our major contributions include:

(i) We propose SRA, a novel feature to represent the similarity between the structural roles of sensitive API call nodes in a graph. Based on SRAs, we transform the high-cost graph matching into an easy-to-compute similarity calculation between vectors.

(ii) We propose and develop GefDroid, a novel system for familial analysis of Android malware by using unsupervised learning and constructing malware link network (MLN) based on SRAs.

(iii) We conduct extensive experiments to evaluate GefDroid. The results show that GefDroid achieves high agreements (0.707-0.883 in term of NMI) between clustering results and the ground truth datasets. Furthermore, GefDroid requires only linear run-time overhead and takes around 8.6s to analyze a sample on average, which is considerably faster than the previous work.

The rest of this paper is organized as follows. Section II introduces the problem. Section III details GefDroid and Section IV reports the experimental results. After discussing the threats to validity in Section V, we introduce the related work in Section VI and conclude the paper in Section VII.

II. Motivation and Problem Definition

A. Motivating Scenario

Let us consider a security analyst who faces thousands of unlabeled malware samples captured every day as illustrated in Fig. 1. These malware samples are generally produced by injecting different kinds of malicious components into popular apps. The analyst aims to find and analyze the new injected malicious components. However, it is time-consuming to conduct an in-depth analysis on each sample. Therefore, the analyst tries to group these malware samples into a set of clusters, where the samples belonging to the same cluster share similar malicious components. By inspecting the similar malicious components in each cluster, the analytical workload of the analyst can be effectively reduced. However, the analyst faces two challenges: First, how to effectively identify the malicious components that usually constitute only a small portion of the samples and may not be implemented in the same way? Second, how to efficiently accomplish the clustering of thousands of malware samples with low overhead? Note that directly applying pair-wise exact matching is neither effective nor efficient [17].

Fig. 1: Motivation Scenario of GefDroid

To tackle the challenges in the above scenario, we first propose a fine-grained feature called SRA that can not only retain the properties of malicious components but also can be resilient to their polymorphic variants. Furthermore, SRA is represented as vectors in a low-dimensional space so that a great deal of malware samples can be handled efficiently. We further develop a new system named GefDroid for automating the analysis process.

B. Problem Definition

Let $M = \{m_1, m_2, \ldots, m_K\}$ be a set of given Android malware samples without family labels, where K is the number of samples. The main task of our work is to construct an MLN that depicts the similarity relationships among different malware samples. Let $MLN = \{M, L\}$, where $L \subseteq M \times M$ denotes the edge set. Each $(m_i, m_j, w_{ij}) \in L$ denotes that there exists an edge with weight w_{ij} between m_i and m_j and they share similar malicious components. The key challenge for this task is how to determine the edges between malware samples. Thus, in our approach, we aim to propose an effective and efficient feature, based on which we can quickly determine the similarities between thousands of malware samples with high accuracy.

Then, the constructed MLN is similar to a social network. The malware families that we aim to find are regarded as the communities existed in the network. In general, community detection algorithms are widely used to detect community structures in social networks, thus they can be applied on our constructed MLN in a similar way. Formally, after constructing the MLN, we aim to find the families as:

$$\mathcal{C}(MLN) \Rightarrow Y = \{y_1, y_2, \ldots, y_R\} \quad (1)$$

where Y denotes the set of clusters generated by community detection algorithm $\mathcal{C}$; R denotes the number of generated clusters. Note that, each sample in M belongs to only one cluster in Y, thus $\sum_{r=1}^{R} |y_r| = K$.

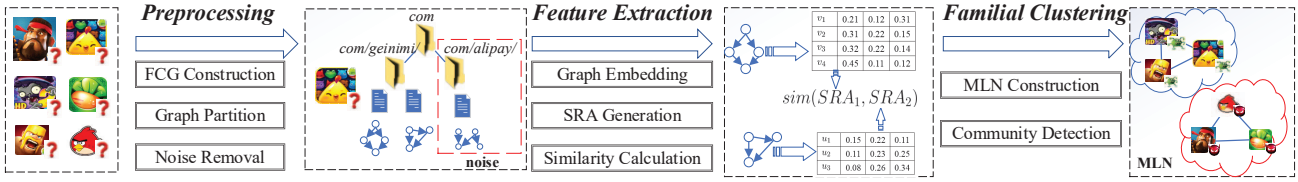

Fig. 2: The overview architecture of GefDroid: (1) Preprocessing stage contains three processes, FCG construction (Section III-A1), graph partition (Section III-A2), and noise removal (Section III-A3); (2) Feature extraction stage contains three processes, graph embedding (Section III-B1), SRA generation (Section III-B2), and similarity calculation (Section III-B3); (3) Familial clustering stage contains two processes, MLN construction (Section III-C1) and community detection (Section III-C2).

III. Methodology

Fig. 2 illustrates the overview architecture of GefDroid, which consists of three main stages.

The preprocessing stage constructs the basic behavior model for each malware sample, and it contains three processes. First, an FCG is constructed to depict the program semantics of a malware sample. Second, due to the inefficiency of analyzing the whole FCG, it is divided into a set of subgraphs according to the app's file directory structure. Third, the subgraphs that belong to the third-party or advertisement libraries are regarded as noises and are removed from the subgraph set.

In the feature extraction stage, an SRA is extracted from each subgraph to perform an easy-to-compute similarity calculation between vectors instead of the high-cost graph matching between subgraphs.

In the familial clustering stage, to perform a clustering task of the malware samples without family labels, an MLN is first constructed based on the similar SRAs between samples. Then, community detection algorithms are applied on the MLN to divide it into a set of clusters, which would be regarded as malware families.

A. Preprocessing

1) FCG Construction: Android apps are normally written in Java and compiled to dalvik code (DEX) stored in a `classes.dex` file. The compiled code and the required resources are packaged into an APK file. On the basis of existing disassemble tools (e.g., *apktool* [18]), we can obtain the dalvik code from the APK.

Given that the dalvik code can be easily changed by typical code obfuscation techniques (e.g., renaming of methods or classes), directly analyzing the dalvik code is not effective. Furthermore, the malware samples within the same family only share similar malicious components that constitute only a small portion of the apps, it is also not efficient to mine similar code snippets with information retrieval techniques.

To retain the program semantics and be resilient to typical code obfuscation techniques, different kinds of effective graph models, including FCG [12], [19], [20], CFG [6], [21], [22], and user interface graph (UIG) [23], [24], are proposed. In our approach, we use FCG rather than CFG and UIG as our graph model to depict app behaviors because of two reasons. First, UIG is not suitable for similarity detection between malware samples since they usually have entirely different UIs. Second, although CFG is a fine-grained graph model that contains detail information of the basic blocks in methods, the extraction and analysis of CFGs is a time-consuming job that requires considerable computational resources. In addition, the results of related approaches [12], [19], [20] have proved that FCG contains enough semantic information to perform malware analysis.

To construct the FCG of a given app, we extract the callers and callees from the dalvik code by identifying the invocation statements, such as "invoke-direct." Then we add the callers and callees as nodes in a graph and insert an edge between two nodes if a function call relation exists between them. The FCG is represented as a directed, unweighted graph $G = (V, E)$.

- $V = \{v_i | 1 \leq i \leq n\}$ denotes the set of functions invoked by an app, where each $v_i \in V$ indicates a function name.
- $E \subseteq V \times V$ denotes the set of function calls, where edge $(v_i, v_j) \in E$ indicates that a function call exists from the caller function v_i to the callee function v_j.

2) Graph Partition: Thousands of nodes are usually found in a constructed FCG. The analysis of entire FCGs are neither effective (i.e., the malicious components constitute only a small portion) nor efficient (i.e., excessive number of nodes and edges to analyze) [19]. However, the malicious components are generally inserted as a package of class files into the popular benign apps [25]. For example, the malware samples in the family called `adrd` are produced by injecting the malicious package called `com.xxx.yyy`, where the malware samples receive control instructions from control servers and send collected device information to a data server. Therefore, it is a promising way to leverage the app's file directory structure rather than its whole graph to identify the commonalities among malware samples. To this end, we divide the FCG into a set of subgraphs as:

$$\mathcal{F}_{div}(G) \Rightarrow SG = \{sg_t | 1 \leq t \leq T\}, \tag{2}$$

where $\mathcal{F}_{div}(*)$ denotes the graph partition function and T denotes the number of class files of the given app. Specifically, we first traverse the app's file structure and record all the class file names. Then, for each class file, a subgraph is constructed by extracting the corresponding part in FCG. For example, to construct the subgraph of a class file whose name is `com.geinimi.c`, we extract all the caller nodes whose class names are `com.geinimi.c` and add them into a subgraph. Then we add their callee nodes as well as the corresponding edges into the subgraph.

3) Noise Removal: It is worth noting that the widely-used third-party and advertisement libraries can introduce false positive links when constructing the MLN. For example, two

malware samples that belong to different families might use the same advertisement libraries such as `com.alipay.sdk`, which is a widely-used library for financial apps. Thus, the similarity between the samples that share any common third-party or advertisement libraries would be higher than their true similarity in terms of malicious activities.

To solve the problem, two filtering methods are applied in GefDroid. First, a list that contains widely-used library names provided by existing approaches [26], [27] is constructed. Second, a list of class names collected from 5,000 benign apps is also constructed. The class names on the two lists are regarded as noises and their corresponding subgraphs are removed from the subgraph set. Even if the two lists can work well for most apps, they are not sound for the class files whose names are obfuscated as `a`, `b` or `c`. To this end, we use the tool *Deguard* [28] to reverse the obfuscated names of given apps. Then we are able to remove the subgraphs of such obfuscated class files if they are obtained in the above two lists.

B. Feature Extraction

1) Graph Embedding: After the construction of graph models, it is straightforward to apply graph matching algorithms (e.g., bipartite graph matching [15]) to perform app similarity detection. However, the graph matching algorithms are slow since they require super-linear time running in the graph size. Furthermore, there are generally hundreds of thousands of graphs that are required to calculate similarities between each other. Thus, the approaches [12], [21], [22], [29] based on graph matching algorithms are inevitably inefficient.

In recent years, deep learning [30] has been applied to many application domains, including graph embedding [31]–[35], which aims at learning low-dimensional vector representations for nodes of a given graph. Graph embedding has been proven to be useful in many tasks of graph analysis, including link prediction [36], node classification [37], and visualization [38]. The learned low-dimensional vector representations for nodes can effectively transform the high-cost graph matching to an easy-to-compute distance calculation between vectors.

In our approach, the applied graph embedding technique should satisfy two requirements. First, given that new malware samples are constantly being discovered, the graph embedding algorithm should work with the input of only one graph per time rather than a graph set (see details in Section VI-B). In this way, the trained model does not need retraining process for the new coming samples. Second, the latent representation of nodes should not depend on the node or edge attribute, especially the node labels (i.e., method names) that can be easily changed by obfuscation techniques. Consider integrating the performance and scalability, we use *struc2vec* [34] as our default graph embedding technique.

Given a subgraph $sg_t = \{V_t, E_t\}$, after applying *struc2vec*, we use $\mathbf{U}_{sg_t} \in \mathbb{R}^{|V_t| \times d}$ to denote the embedding result. Note that sg_t is regarded as an undirected graph here. For each node v in V_t, it will learn a d dimensional feature vector $\mathbf{u}_v$. The learned feature vectors enable the nodes with similar structural roles to be embedded in the near points in Euclidean space.

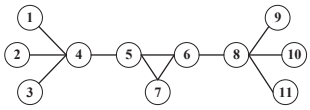

Fig. 3: An example of an undirected graph that contains 11 nodes and 11 edges, where node 4 and node 8 are structurally similar since both of their degrees are 4.

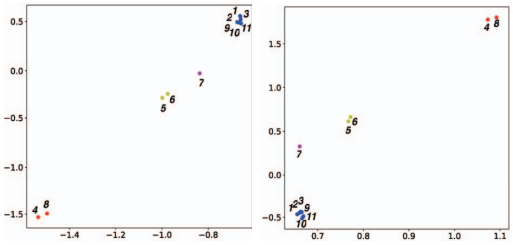

Fig. 4: Visualization of the embedding results of the same graph after twice applying of *struc2vec* with the same arguments.

Fig. 3 presents an example of an undirected graph that contains 11 nodes and 11 edges. The embedding results of the example graph are illustrated in Fig. 4, where the dimension argument is set as 2 for visualization here. As can be seen from the two figures, the learned feature vectors of the same nodes are quite different due to the random walk strategy used in *struc2vec*. Thus, it is not effective to directly apply the embedding technique in our work.

However, we observe that even the vectors of the same nodes are different, the distance relationships between the nodes are well remained. For example, node 4 and node 8 are structurally similar and the distances between them in the two figures are nearly the same while their locations are different.

2) SRA Generation: Inspired by the above observation found from the embedding result, we leverage the similarity relationships between the structural roles of identified node pairs (e.g., node 4 and node 8) to represent the structural feature of a subgraph. However, it is impossible to map the user-defined method nodes between two subgraphs since their names can be changed by the obfuscation techniques. Thus, we focus on the sensitive API call nodes that cannot be easily changed. Furthermore, the sensitive API calls are generally invoked by malware samples to perform malicious activities, which could provide useful information for the malware similarity detection [19].

To obtain the set of sensitive API calls, we rely on the work of *SuSi* [39], which provides a list of source API calls (e.g., `getLine1Number()` that returns the phone number of the user) and a list of sink API calls (e.g., `sendTextMessage()` that sends short messages). Finally, we use SA to denote the set of 9,730 sensitive API calls.

According to the constructed SA, we generate the SRA, representing the similarity relationships between the structural roles of sensitive API call nodes for each given subgraph. In detail, SRA_t of subgraph sg_t is calculated with two steps.

First, a subgraph sg_t contains a set of sensitive API call nodes, which is denoted as $SA_t \subseteq SA$. Thus, a set of sensitive API node pairs $\{(v,u)|v,u \in SA_t\}$ is obtained if the subgraph sg_t contains at least two sensitive API call nodes.

Second, on the basis of the learned low-dimensional vector

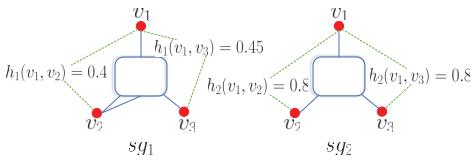

Fig. 5: An example of the similarity calculation of two SRAs generated from sg_1 and sg_2.

representations of sensitive API call nodes using *struc2vec*, let $h_t(v, u)$ be the similarity relationship between the structural roles of node v and node u, and it is calculated with the standard cosine similarity metric as:

$$h_t(v, u) = \cos(\mathbf{u}_v, \mathbf{u}_u) = \frac{\mathbf{u}_v \cdot \mathbf{u}_u}{\|\mathbf{u}_v\|\|\mathbf{u}_u\|} \tag{3}$$

where $\mathbf{u}_v$ and $\mathbf{u}_u$ denote the vector representations of node v and node u, respectively. Furthermore, $h_t(v, u) = h_t(u, v)$. In our work, we rank the sensitive API calls in a dictionary ordered method. Thus, for two sensitive API call nodes v and u, $h_t(v, u)$ is stored only if the dictionary order index of v is less than that of u, or $h_t(u, v)$ is stored.

Finally, SRA_t is obtained as:

$$SRA_t = \{h_t(v, u) | v, u \in SA_t \quad and \quad v \neq u\}. \tag{4}$$

Thus, $|SRA_t| = \frac{|SA_t| \cdot (|SA_t| - 1)}{2}$, where $|SA_t|$ is considerably less than the subgraph's node number.

3) SRA Similarity Calculation: After generating SRA for each subgraph, we are able to transform the high-cost graph matching between subgraphs into the similarity calculation between SRAs. There are two intuitions for the similarity calculation between SRAs and they are listed as below:

- If two SRAs share less common sensitive API call nodes, the functionalities of their corresponding classes would be less similar.
- If the common sensitive API call nodes of two SRAs present less similar structural roles between each other, their invocation patterns as well as the functionalities of their corresponding classes would be less similar.

On the basis of the above two intuitions, the similarity of two given SRAs generated from sg_1 and sg_2, denoted as $sim(SRA_1, SRA_2)$, is obtained with Eq. (5).

$$sim(SRA_1, SRA_2) = \frac{\sum_{v_i \in SA_1 \cap SA_2} sim(\mathbf{sr}_1(v_i), \mathbf{sr}_2(v_i))}{|SA_1 \cup SA_2|} \tag{5}$$

where $\mathbf{sr}_1(v_i)$ and $\mathbf{sr}_2(v_i)$ are represented as two vectors and they denote the similarity relationships between node v_i with other sensitive API call nodes in subgraphs sg_1 and sg_2, respectively. To obtain $\mathbf{sr}_1(v_i)$ and $\mathbf{sr}_2(v_i)$, for convenience, we first construct two distance matrices $D_t (t = 1, 2)$ for two subgraphs as Eq. (6). Then $\mathbf{sr}_t(v_i)$ is the i^{th} row vector of the constructed matrices as Eq. (7).

$$D_t[i, j] = \begin{cases} h_t(v_i, v_j) & v_i, v_j \in SA_1 \cap SA_2, i \neq j \\ 0 & i = j \end{cases} \tag{6}$$

$$\mathbf{sr}_t(v_i) = D_t[i, :] \tag{7}$$

$$sim(\mathbf{sr}_1(v_i), \mathbf{sr}_2(v_i))) = \frac{1}{1 + \|\mathbf{sr}_1(v_i) - \mathbf{sr}_2(v_i)\|_2} \tag{8}$$

Fig. 5 presents an example of the similarity calculation of two SRAs generated from sg_1 and sg_2. Note that only parts of the subgraphs are shown, the other parts located in rectangles are quite different. The two subgraphs have three common sensitive API call nodes (i.e., red nodes v_1, v_2, and v_3). For subgraph sg_1, $h_1(v_1, v_2) = 0.4$ and $h_1(v_1, v_3) = 0.45$. For subgraph sg_2, $h_2(v_1, v_2) = h_2(v_1, v_3) = 0.8$. Therefore, $sim(\mathbf{sr}_1(v_1), \mathbf{sr}_2(v_1))) = sim(< 0.4, 0.45 >, < 0.8, 0.8 >)$. Note that the cosine metric result of the two vectors is 0.998. However, the high similarity calculated with cosine metric cannot depict the different sensitive API invocation patterns here. Thus, we apply the Euclidean metric with Eq. (8) rather than the cosine metric. The Euclidean metric result is 0.653, considerably less than the result of cosine metric.

C. Familial Clustering

1) MLN Construction: After the feature extraction stage, given two malware samples, we are able to capture their similarity relationship based on their similar SRAs. To perform familial analysis using unsupervised learning, we aim to construct an MLN, where each node denotes a malware sample, and each edge between two samples denotes that there exist similar SRAs between them. Therefore, the MLN can depict the similarity relationships among all the malware samples to be analyzed.

Algorithm 1 lists the steps of constructing the MLN with the input of malware set M and two threshold values, θ and ϵ. θ denotes the similarity threshold value between SRAs. In other words, if the similarity of two SRAs calculated with Eqs. (5-8) is no less than θ, they are regarded as the same, indicating that their corresponding classes share similar functionalities. ϵ denotes the threshold value of adding edges between sample nodes. If the number of same SRAs shared by two samples is no less than ϵ, then an edge is added between the two samples.

In Algorithm 1, after the preprocessing of each sample in M (lines 2-3), a set of SRAs, denoted as $SRASet$, is constructed (lines 4-7). Then each sample is added to the MLN as a node (line 8). After that, for each sample-pair in M, the number of same SRAs between them is calculated and represented as k (lines 11-18). An edge with weight k is added for the sample-pair if there exist no less than ϵ same SRAs between them (lines 19-21).

2) Community Detection: To group the malware samples into clusters on the basis of the constructed MLN, community detection algorithms are effective to determine whether the MLN has community structures if the nodes can be easily grouped into sets of nodes, such that each set of nodes is internally densely connected. As a result, the malware samples grouped in the same cluster could be regarded as belonging to the same malware family. For the new samples that are constantly being discovered, they are placed into the clusters that have connections with them by calculating the similarity relationships with existing samples.

Fig. 6 presents an example of community detection result of MLN for fifteen malware samples in three families, i.e., `geinimi`, `droidkungfu`, and `adrd`. It is obvious that the

Algorithm 1 Construction of MLN

Input: M // M denotes the set of malware samples to be analyzed.
θ // θ denotes the similarity threshold value between $SRAs$.
ϵ // ϵ denotes the threshold value of adding edges between nodes.
Output: $MLN = \{M, L\}$

for each malware sample m_i in M **do**
  $SG_{m_i} = \mathcal{F}_{div}(G_{m_i})$
  $RemoveNoise(SG_{m_i})$
  for each sg_t in SG_{m_i} **do**
    $SRA_t = GenerateSRA(sg_t)$
  end for
  $SRASet_{m_i} = \{SRA_t | 1 \leq t \leq T\}$
  $MLN.addNode(m_i)$
end for
for each sample-pair (m_i, m_j) in M **do**
  $k = 0$
  for each SRA_t in $SRASet_{m_i}$ **do**
    for each $SRA_{t'}$ in $SRASet_{m_j}$ **do**
      if $sim(SRA_t, SRA_{t'}) \geq \theta$ **then**
        $k = k + 1$
      end if
    end for
  end for
  if $k \geq \epsilon$ **then**
    $MLN.addEdge(m_i, m_j, k)$ // k denotes the edge weight.
  end if
end for
return MLN

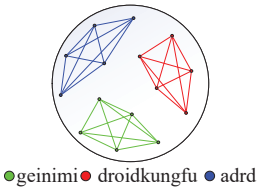

Fig. 6: An example of community detection result of MLN for fifteen malware samples in three families.

constructed MLN can be divided into three clusters. In each cluster, the samples are connected with each other, indicating that the samples within the same cluster share similar malicious components. On the basis of the clustering results, our approach can effectively help security analysts focus on the commonalities among malware samples within the same cluster, and potentially isolate the malicious behaviors of malware samples from different clusters.

IV. Evaluation

We use three datasets with real malware samples and six metrics to carefully evaluate GefDroid and answer five research questions:

***RQ** 1: Does GefDroid outperform the baseline approaches in term of accuracy?* (Section IV-B1)

***RQ** 2: Can GefDroid handle new malware samples without retraining the model?* (Section IV-B2)

***RQ** 3: Can GefDroid process a great deal of samples with low run-time overhead?* (Section IV-C1)

***RQ** 4: Does GefDroid outperform the baseline approaches in term of efficiency?* (Section IV-C2)

***RQ** 5: To what extent is GefDroid resilient to obfuscation techniques?* (Section IV-D)

RQ 1-2 examine the accuracy of GefDroid while RQ 3-4 investigate the efficiency. RQ 5 evaluates the resilience of GefDroid to code obfuscation.

TABLE I: Descriptions of three used datasets

Dataset	#Family (Q)	#Malware (K)	Max.	Min.	Avg.
dataset-I [3]	49	1,260	15.4MB	12KB	1.3MB
dataset-II [40]	179	5,560	24.8MB	5KB	1.3MB
dataset-III [41]	36	8,407	36.2MB	12KB	2MB

A. Study Setup

1) Datasets: We evaluate GefDroid on three ground truth datasets provided by Genome project [3], Drebin [40], and Fan *et al.* [41]. For convenience, they are named as dataset-I, dataset-II, and dataset-III. Their descriptions are listed in TABLE I, where columns 2-3 list the number of families (Q) and the number of malware samples (K). Different datasets have different distributions of malware samples. Columns 4-6 lists the maximum, minimum, and average size of malware samples. Note that each sample in these datasets has been attached to a family label given by experts.

2) Metrics: Six metrics are used to measure the clustering performance. They are normalized mutual information (NMI) [42], adjusted rand index (ARI) [43], Fowlkes-Mallows index (FMI) [44], Homogeneity [45], Completeness [45], and V-measure [45]. NMI, ARI, and FMI are three widely-used metrics that measure the agreement between the clustering result and the ground truth dataset. Homogeneity measures the extent of how each generated cluster contains only samples of a single family. Completeness measures the extent of how all samples of each family are assigned to the same cluster. V-measure is the harmonic mean of homogeneity and completeness. Except for the ARI, the values of the other five metrics range from 0 to 1, where a higher value indicates a better agreement between the predicted clusterings and the true clusterings. The value of ARI ranges from -1 to 1, where random labelings have an ARI value close to 0.0. For all the six metrics, 1.0 stands for a perfect match with the ground truth dataset. Recall the example of community detection result illustrated in Fig. 6, all the six metrics are 1.0.

3) Community detection algorithm: We apply four widely-used community detection algorithms to the MLNs constructed on the three datasets. These algorithms include:

- Infomap, which detects community structures of a network using the approach proposed by Rosvall *et al.* [46].
- Fast greedy, which is based on the greedy optimization of modularity [47], a metric to measure the quality or significance of a community structure.
- Label propagation, which is a fast partitioning algorithm proposed by Raghavan *et al.* [48].
- Multilevel, which is a layered and bottom-up community detection algorithm proposed by Blondel *et al.* [49].

Due to the page limitation, we cannot present all the clustering performance with the four algorithms on the three datasets. According to the performance, the infomap algorithm achieves the best clustering performance among these four algorithms. The average NMI value of infomap on the three datasets is 0.795, while those of the other three algorithms are 0.668, 0.762, and 0.728. In addition, infomap also performs best in terms of the other five metrics. Therefore, in our latter experiments, we select infomap as our default community

TABLE II: Clustering performance of GefDroid and baseline approaches with infomap algorithm on three datasets.

Dataset	Baseline Approaches	NMI	ARI	FMI	Homogeneity	Completeness	V-measure	#Cluster
dataset-I (Q=49, K=1,260)	Permission	0.676	0.447	0.585	0.559	0.818	0.664	34
	API	0.770	0.564	0.655	0.718	0.826	0.768	68
	FalDroid	0.812	0.686	0.720	0.860	0.767	0.811	74
	GroupDroid	0.798	0.583	0.630	0.834	0.765	0.798	70
	GefDroid (w/o NR)	0.703	0.433	0.576	0.604	0.820	0.695	53
	GefDroid	**0.883**	**0.870**	**0.886**	**0.892**	**0.876**	**0.883**	74
dataset-II (Q=179, K=5,560)	Permission	0.543	0.261	0.414	0.416	**0.707**	0.524	83
	API	0.718	0.456	0.500	0.773	0.666	0.716	392
	FalDroid	0.757	0.502	0.555	0.826	0.694	0.754	232
	GroupDroid	0.743	0.404	0.476	0.860	0.642	0.735	245
	GefDroid (w/o NR)	0.711	0.371	0.418	0.792	0.639	0.707	375
	GefDroid	**0.793**	**0.534**	**0.606**	**0.924**	0.681	**0.784**	471
dataset-III (Q=36, K=8,407)	Permission	0.507	0.176	0.388	0.351	**0.731**	0.474	42
	API	0.657	0.361	0.414	0.680	0.635	0.657	160
	FalDroid	0.672	0.488	0.530	0.711	0.634	0.671	75
	GroupDroid	0.693	0.365	0.426	0.674	0.712	0.692	59
	GefDroid (w/o NR)	0.642	0.256	0.361	0.628	0.656	0.642	204
	GefDroid	**0.707**	**0.509**	**0.559**	**0.832**	0.600	**0.697**	685

detection algorithm due to its superior performance.

4) Parameters: There are two parameters that play important roles in our approach, i.e., θ controls the similarity threshold value between SRAs; ϵ controls the threshold value of adding edges between sample nodes. To select the proper θ and ϵ, we vary the values of θ as $\{0.6, 0.65, 0.7, 0.75, 0.8, 0.85, 0.9, 0.95\}$ and vary the values of ϵ as $\{1, 2, 3, 4, 5\}$ when conducting the experiments on all the three used datasets. According to the clustering performance with different parameters, we select the default values of θ and ϵ as 0.75 and 1, respectively.

We conduct the experiments on a quad-core 3.20 GHz PC running Ubuntu 15.10(64 bit) with 32 GB RAM and 1 TB hard disk.

B. Accuracy of GefDroid

1) ***RQ1:*** *Does GefDroid outperform the baseline approaches in term of accuracy?:* We compare the accuracy of GefDroid with four baseline approaches that are briefly introduced as below:

- Wang *et al.* proposed an approach for malware detection based on the requested permissions, which are security-aware features that restrict the access of apps to the core facilities of devices [10].
- Aafer *et al.* proposed an approach for malware detection based on API calls, which are more fine-grained features than permissions since each permission governs several API calls [11].
- Fan *et al.* proposed FalDroid, which performs familial classification based on the generated fregraphs that denote the common behaviors of malware samples within the same families [12].
- Marastoni *et al.* proposed GroupDroid, which uses 3D-CFG centroids [6] as features to measure the similarities between malware samples and perform grouping [13].

Among these approaches, GroupDroid [13] performs a clustering task like GefDroid does, while the other three approaches [10]–[12] perform a classification task and they suffer two main limitations. First, they require a training dataset with family labels assigned by experts, which is not easy to obtain. Second, they can only identify the families that are only provided in the training dataset. Thus, for Android familial analysis, it is more practical to perform a clustering task as we do rather than performing a classification task.

To have a fair comparison of clustering performance with the approaches that perform a classification task [10]–[12], we construct different MLNs for such approaches based on their proposed features, e.g., fregraphs, permissions, and API calls. Then the infomap algorithm is applied on their MLNs to perform a clustering task. For GroupDroid [13], we re-implement it and perform a clustering task based on the extracted 3D-CFG centroids. In addition, we use GefDroid (w/o NR) to denote our approach without the preprocessing of noise removal, in order to evaluate whether the third-party or advertisement libraries affect the clustering performance.

The comparison results are listed in TABLE II, where the term #Cluster denotes the number of generated clusters. We can draw the following four conclusions from the results:

(i) Except for the completeness metric, GefDroid performs best among these approaches in terms of the other five metrics.
(ii) GefDroid generates the most clusters. The highest homogeneity values and the most clusters indicate that GefDroid can well isolate the malicious behaviors of malware samples from different families.
(iii) In general, the string-based features (i.e., permissions and API calls) perform worse than the graph-based features. The main reason is that they cannot well depict the program semantic meanings, thus insufficient to mine the common malicious components of malware samples within the same families.
(iv) The preprocessing of noise removal significantly improves the clustering performance, indicating that the widely used third-party or advertisement libraries would introduce noise edges into the MLNs.

Then, we investigate the clustering results of data-III that has only 36 families but 685 generated clusters. The clustering results are listed in TABLE III, where RP denotes the reduced percentage of malware samples in which its inspection can be deferred because of our clustering. RP

TABLE III: Clustering result of GefDroid on dataset-III, where RP denotes the reduced percentage of malware samples in which its inspection can be deferred because of our clustering.

Family	#Sample	#Cluster	RP	Family	#Sample	#Cluster	RP
adwo	338	105	0.689	hongtoutou	46	4	0.913
airpush	76	34	0.552	iconosys	153	7	0.954
anserver	53	1	0.981	imlog	41	4	0.902
basebridge	303	35	0.884	jsmshider	22	3	0.864
boqx	49	25	0.490	kmin	248	4	0.984
boxer	95	10	0.894	kuguo	358	93	0.740
clicker	37	19	0.486	lovetrap	19	1	0.947
dowgin	851	64	0.925	mobiletx	81	2	0.975
ddlight	101	14	0.861	pjapps	82	4	0.951
droidkungfu	736	39	0.947	plankton	896	3	0.997
droidsheep	14	1	0.929	smskey	111	38	0.658
fakeangry	16	6	0.625	smsreg	149	25	0.832
fakedoc	147	2	0.986	steek	20	1	0.950
fakeinst	1,504	61	0.959	utchi	285	1	0.996
fakeplay	43	13	0.698	waps	771	202	0.738
geinimi	105	1	0.990	youmi	113	62	0.451
gingermaster	385	12	0.969	yzhc	49	1	0.980
golddream	80	6	0.925	zitmo	30	4	0.867

is calculated as $RP = 1 - \frac{\#Cluster}{\#Sample}$. For example, the family *steek* contains 20 samples, and only one cluster is generated with our clustering approach. Therefore, its RP is $1 - \frac{1}{20} = 0.950$. In other words, the analyst only need to inspect one sample from the generated cluster because these samples share similar malicious components. According to the result, the average value of RP is 0.847, indicating that our approach can effectively reduce the analytical workload of the analyst. However, there are still some families whose RPs are lower than 0.5 (e.g., 0.451 for family *youmi*), indicating that it still requires a lot time for the analyst to manually review. We manually inspect the samples in family *youmi*, which are adware that just display annoying, and misleading advertisement. We find that 16 samples do not contain any SRA since their malicious advertisements are removing in our preprocessing stage. Therefore these samples have no connection with others. The detection of whether the contained advertisements are annoying or misleading is still a challenge for existing works, and we leave this as our future work.

Answer to RQ 1: *GefDroid can achieve high agreements between the clustering results and the ground truth datasets, which outperforms baseline approaches.*

*2) **RQ2**: Can GefDroid handle new malware samples without retraining the model?:* We randomly select 100 samples from each dataset and regard them as new samples. Then, we calculate their similarity relationships with existing samples in the MLN, and use three terms to evalaute the performance.

- True-Link Rate: the percent of new samples that have links with the samples that belong to the same families.
- False-Link Rate: the percent of new samples that have and only have links with the samples that belong to different families.
- No-Link Rate: the percent of new samples that have no links with existing samples.

We repeat this experiment 100 times on the three datasets. The average results listed in TABLE IV indicate that GefDroid can effectively link the new coming samples with their variants in the MLN. Moreover, we find that 0.52% of new samples actually belong to the families that contain only one sample in the datasets, thus causing their no link with existing samples.

TABLE IV: Performance of detecting new coming malware samples.

Dataset	True-Link Rate	False-Link Rate	No-Link Rate
dataset-I	94.91%	1.89%	3.20%
dataset-II	94.51%	1.21%	4.28%
dataset-III	92.80%	1.09%	6.11%

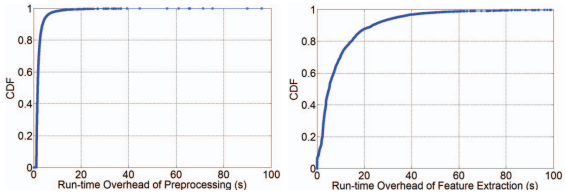

Fig. 7: CDFs for the run-time overhead of the preprocessing stage (left) and the feature extraction stage (right) on dataset-III.

TABLE V: Run-time overheads of MLN construction and community detection on three datasets.

Dataset	$\overline{T}$	#SRA pairs	MLN Construction	Community Detection
dataset-I	3.1	$7.46 * 10^6$	20s	2s
dataset-II	3.2	$1.58 * 10^8$	131s	11s
dataset-III	5.1	$9.09 * 10^8$	750s	45s

Answer to RQ 2: *GefDroid can effectively link the new samples with their variants in the MLN and identify the samples that belong to new families.*

C. Efficiency of GefDroid

*1) **RQ3**: How is the overhead of GefDroid for handling a great deal of samples?:* We evaluate the run-time overhead of GefDroid for its three main stages that are listed as below:

- *Preprocessing*: All the given samples are disassembled and a set of subgraphs for each sample are constructed.
- *Feature Extraction*: For each subgraph of a sample, its nodes are encoded into low-dimensional vectors with *struc2vec*. Then, an SRA is generated to represent the structural feature of the subgraph.
- *Familial Clustering*: An MLN is constructed based on the similarity calculation of SRAs. Then, the infomap algorithm is applied on the MLN for malware clustering.

Fig. 7 presents the cumulative distribution function (CDF) for the run-time overhead of the preprocessing procedure (left) and the feature extraction procedure (right) on dataset-III. For the preprocessing stage, only 2.1s is needed on average. Moreover, more than 98% of samples require less than 10s. For the feature extraction stage, 6.5s is needed on average. Furthermore, only about 6.1% of samples require more than 30s. The cost of feature extraction mainly depends on the size of the subgraphs that are embedded. The used embedding algorithm *struc2vec* [34] scales super-linearly but closer to linear. It is worth noting that the preprocessing and the feature extraction stages could be conducted on several PCs in parallel, thus further reducing the total overhead.

For the stage of familial clustering, an MLN is first constructed by calculating the similarities between SRAs. Thus, the calculation complexity of the SRA pairs is about $O(\frac{K*(K-1)}{2} * \overline{T} * \overline{T})$, where K and $\overline{T}$ denote the total number of samples and the average number of SRAs per sample

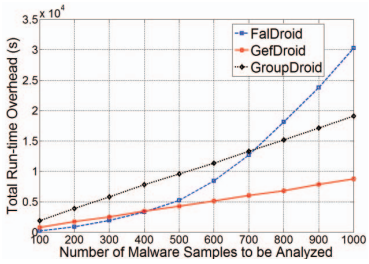

Fig. 8: Comparison result of the run-time overheads.

has, respectively. TABLE V lists the run-time overheads of MLN construction and community detection on three datasets. Even for the biggest dataset-III, the similarity calculation of $9.09 * 10^8$ pair of SRAs is accomplished in only 750s. The cost for the community detection is considerably less than that of the MLN construction. Furthermore, for a new coming sample, it only needs 0.2s to calculate the similarities with existing samples.

Answer to RQ 3: *GefDroid only takes around 8.6s to analyze a sample on average, and thus it can handle a large scale of samples efficiently.*

2) ***RQ 4****: Does GefDroid outperform the baseline approaches in term of efficiency?:* We compare the overhead of GefDroid and that of the baseline approaches. For the permissions- and API-calls-based approaches, their clustering performance is considerably worse than GefDroid. Moreover, compared with the graph-based features, the permissions- and API-calls-based features cannot provide enough explanations to the relationships between malware samples within the same clusters. Hence, we focus on the efficiency comparison between GefDroid and the two graph-based approaches, i.e., FalDroid and GroupDroid. More precisely, we first randomly select 1,000 malware samples. Then, a set of subgraphs are generated for each sample in the three approaches. After that, FalDroid adopts a weighted-sensitive-API-calls-based graph matching approach to calculate the graph similarities, which has been proved to be faster than the graph edit distance algorithm. For GroupDroid, a 3D-CFG centroid represented as a four-dimensional vector is calculated for each subgraph. In GefDroid, an SRA is generated to represent each subgraph. In summary, the similarity detection between samples of FalDroid is based on the graph matching algorithm while GroupDroid and GefDroid rely on similarity calculation between vectors.

Fig. 8 use the blue line, black line, and red line denote the increase of total run-time overhead of FalDroid, GroupDroid, and GefDroid, respectively. We can see that the blue line grows exponentially while the black line and the red line show linear growths. About 19s is required for GroupDroid to construct the 3D-CFGs and calculate the centroid, which is twice as the time GefDroid needs. For FalDroid, when the number of samples is lower than 400, it shows higher efficiency than GefDroid by directly applying the graph matching approach. However, with the increase in the number of samples, the cost of FalDroid is considerably higher than GefDroid.

Answer to RQ 4: *GefDroid only requires linear run-time overhead in terms of the number of samples, and thus is considerably faster than the previous work.*

D. Resilience of GefDroid

1) ***RQ 5****: To what extent is GefDroid resilient to obfuscation techniques?:* To answer RQ 5, we initially leverage different kinds of obfuscation techniques to produce the variants of our samples. Then we calculate the similarities between the SRAs generated by the original and obfuscated samples.

Given that SRA is proposed to depict the similarity relationships between structural roles of sensitive API call nodes in a graph, it is resilient to typical obfuscation techniques [50] such as function renaming, instruction substitution, and string encryption, which cannot change the structure of FCG. However, there are several advanced obfuscation techniques that can slightly change the FCG structure, including the control flow obfuscation and the reflection technique.

To evaluate the resilience of GefDroid to the control flow obfuscation, which changes the FCG structure by inserting or deleting useless method nodes, we apply GefDroid to ten samples obfuscated by the Android obfuscator called *DashO* [51]. After the similarity calculation between the SRAs generated by the original and obfuscated samples, the results show that even the FCG structures are slightly changed with several nodes, their similarities are still higher than our threshold value θ, which is set as 0.75 in our approach. For the reflection technique which might hide calling edges of FCG, we leverage *DroidRA* [52], an open-source tool, to detect reflection methods and add the missing edges. The experiment shows that on average only two edges containing a sensitive API call node are added into the FCG for each app, which barely affects the accuracy of GefDroid.

In addition to the above obfuscation techniques, encryption packers, such as *Bangcle* [53] and *Baidu* [54], are the most popular obfuscation tools now. They can hide the actual Dex code, thus making the disassembled tools unable to get the dalvik code. In our approach, we use the unpacker tool *PackerGrind* [55] to recover the actual Dex files.

Answer to RQ 5: *GefDroid is resilient to typical obfuscation techniques and can deal with advanced packing techniques by leveraging existing tools.*

V. Threats to Validity

A. Threats to Internal Validity

Native code. In our approach, we limit our analysis to the FCG model constructed based on the dalvik code. We do not analyze native code. Thus, our approach would miss the malicious behaviors implemented in native code. However, there are many binary analysis frameworks, such as *Angr* [56], that can help us address this limitation by constructing the FCG of the native code. Then, we could apply our approach to conducting similarity detection of such FCGs. We will explore this approach in future work.

Sensitive API calls. Our detection of sensitive API calls relies on the set provided by *SuSi* [39], which now, four years later, might be incomplete or outdated. Missing or incorrect

sensitive API calls contained in SA would make GefDroid miss or misidentify the common malicious behaviors between malware samples within the same families. Furthermore, since the sensitive API calls are extracted statically in GefDroid, the ones that are never executed by the malware samples would introduce noises when detecting the similarities between samples. In future work, combining the dynamic analysis [57] with the static analysis is a promising way to reduce the side-effects caused by the dead code that will never be executed.

B. Threats to External Validity

Third-party libraries. To remove the third-party and advisement libraries, we extend the widely-used library list by adding the class names of 5,000 benign apps. Even the list works well on our datasets, it is unclear how does the list performs when applying GefDroid on other datasets. In future work, we plan to construct bigger datasets that contain recent malware samples and evaluate the performance of GefDroid on them.
Multi-label malware. GefDroid can well handle the samples in our three used datasets from which each sample belongs to exactly one malware family. However, it might fail when dealing with the multi-label malware samples that contain code from multiple malware families. The multi-label malware samples belong to the overlapping region in the constructed MLN, which might be handled by the overlapping community detection algorithms [58]. We leave the detection of multi-label samples as our future work.

VI. Related Work

A. Malware Familial Analysis

There is a large body of research devoted to the familial analysis of Android malware. Deshotels *et al.* [5] proposed DroidLegacy, which partitions the app code into loosely coupled modules and identifies the malicious module of each piggybacked malware family. Suarez *et al.* [59] proposed Dendroid, which automatically classifies malware and analyzes families on the basis of code structures. Zhang *et al.* [29] proposed DroidSIFT, which constructs family features on the basis of the API dependency graphs. Feng *et al.* [60] proposed Astroid, which automatically synthesizes a maximally suspicious common subgraph of each malware family as a signature to perform the familial analysis.

Most of the above approaches work with supervised learning. They require a set of known malware samples labeled by experts to be used as training samples. Thus, they can only classify the malware samples that belong to known families in training dataset. Compared with these approaches, GefDroid works with unsupervised learning that does not need any labeled samples. Therefore, our approach is able to avoid the model retraining, and detect the new coming samples by calculating their connections with existing samples in MLN.

B. Graph Embedding

The embedding techniques based on representing graphs in vector spaces, while preserving their properties, have become widely popular. There are two types of representation learning.

The first is to encode nodes as low-dimensional vectors that summarize their structural roles in graphs. Perozzi *et al.* [32] proposed DeepWalk which first uses the random walks to generate node sequence as its context. Then Grover and Leskovec [33] improved the DeepWalk model by proposing node2vec that uses second-order random walks to generate the node sequence. Ribeiro *et al.* [34] proposed struc2vec, which uses a hierarchy to measure node similarity at different scales, and constructs a multilayer graph to encode structural similarities and generate the structural context for nodes. However, these approaches cannot be directly applied to our work since their embedding results of the same graph are not in a consensus due to the using of random walks. Thus we propose SRA to represent the graph feature for similarity calculation.

The second is to encode a graph as low-dimensional vectors instead of a node. Dai *et al.* [35] proposed structure2vec, which is based on the idea of embedding latent variable models into feature spaces and learning such feature spaces using discriminative information. Narayanan *et al.* [61] proposed graph2vec, which is also based on the skip-gram model for learning embedding similar to node2vec. The difference is that it views an entire graph as a document and the subgraphs around each node in the graph as words that compose the document. Even such approaches can learn representations for graphs, they require a graph set as input and need model retraining to deal with the new coming samples.

VII. Conclusion

We propose SRA, a novel feature to represent the similarity relationships between the structural roles of sensitive API call nodes in a graph. By doing so, we transform the high-cost graph matching into an easy-to-compute similarity calculation between vectors. Moreover, we design and develop GefDroid, a new system for familial analysis of Android malware by using unsupervised learning and constructing MLN based on SRAs. Our extensive evaluation results show that GefDroid outperforms the state-of-the-art approaches in terms of accuracy and efficiency.

Acknowledgment

This research was supported by National Key R&D Program of China (2016YFB1000903), National Natural Science Foundation of China (61532004, 61532015, 61632015, 61602369, U1766215, 61772408, 61702414, 61833015), Innovative Research Group of the National Natural Science Foundation of China (61721002), Ministry of Education Innovation Research Team (IRT_17R86), consulting research project of Chinese academy of engineering "The Online and Offline Mixed Educational Service System for 'The Belt and Road' Training in MOOC China" and Project of China Knowledge Centre for Engineering Science and Technology, and in part by RGC Project No. PolyU 152279/16E, 152223/17E, CityU C1008-16G.

REFERENCES

[1] F-Secure Inc., "Another reason 99https://safeandsavvy.f-secure.com/2017/02/15/another-reason-99-percent-of-mobile-malware-targets-androids/, 2017.

[2] Qihoo Inc., "Report of smartphone security in china, 2017," http://zt.360.cn/1101061855.php?dtid=1101061451&did=491056914, 2018.

[3] Y. Zhou and X. Jiang, "Dissecting android malware: Characterization and evolution," in *Proc. IEEE S&P*, 2012.

[4] Y. Feng, S. Anand, I. Dillig, and A. Aiken, "Apposcopy: Semantics-based detection of android malware through static analysis," in *Proc. FSE*, 2014.

[5] L. Deshotels, V. Notani, and A. Lakhotia, "Droidlegacy: Automated familial classification of android malware," in *Proc. PPREW*, 2014.

[6] K. Chen, P. Liu, and Y. Zhang, "Achieving accuracy and scalability simultaneously in detecting application clones on android markets," in *Proc. ICSE*, 2014.

[7] H. Wang, Y. Guo, Z. Ma, and X. Chen, "Wukong: A scalable and accurate two-phase approach to android app clone detection," in *Proc. ISSTA*, 2015.

[8] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes, "Sourcerercc: Scaling code clone detection to big-code," in *Proc. ICSE*, 2016.

[9] Y. Lin, Z. Xing, Y. Xue, Y. Liu, X. Peng, J. Sun, and W. Zhao, "Detecting differences across multiple instances of code clones," in *Proc. ICSE*, 2014.

[10] W. Wang, X. Wang, D. Feng, J. Liu, Z. Han, and X. Zhang, "Exploring permission-induced risk in android applications for malicious application detection," *IEEE TIFS*, vol. 9, no. 11, 2014.

[11] Y. Aafer, W. Du, and H. Yin, "Droidapiminer: Mining api-level features for robust malware detection in android," in *Proc. SecureComm*, 2013.

[12] M. Fan, J. Liu, X. Luo, K. Chen, T. Chen, Z. Tian, X. Zhang, Q. Zheng, and T. Liu, "Frequent subgraph based familial classification of android malware," in *Proc. ISSRE*, 2016.

[13] N. Marastoni, A. Continella, D. Quarta, S. Zanero, and M. D. Preda, "Groupdroid: Automatically grouping mobile malware by extracting code similarities," in *Proc. SSPREW*, 2017.

[14] A. Gorla, I. Tavecchia, F. Gross, and A. Zeller, "Checking app behavior against app descriptions," in *Proc. ICSE*, 2014.

[15] K. Riesen, S. Emmenegger, and H. Bunke, "A novel software toolkit for graph edit distance computation," in *Proc. GbRPR*, 2013.

[16] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song, "Neural network-based graph embedding for cross-platform binary code similarity detection," in *Proc. CCS*, 2017.

[17] J. Kinable and O. Kostakis, "Malware classification based on call graph clustering," *Journal in Computer Virology*, vol. 7, no. 4, 2011.

[18] "Apktool: A tool for reverse engineering android apk files," https://ibotpeaches.github.io/Apktool/, 2016.

[19] M. Fan, J. Liu, W. Wang, H. Li, Z. Tian, and T. Liu, "Dapasa: detecting android piggybacked apps through sensitive subgraph analysis," *IEEE TIFS*, vol. 12, no. 8, 2017.

[20] H. Gascon, F. Yamaguchi, D. Arp, and K. Rieck, "Structural detection of android malware using embedded call graphs," in *Proc. AiSec*, 2013.

[21] J. Crussell, C. Gibler, and H. Chen, "Andarwin: Scalable detection of semantically similar android applications," in *Proc. ESORICS*, 2013.

[22] ——, "Attack of the clones: Detecting cloned applications on android markets," in *Proc. ESORICS*, 2012.

[23] K. Chen, P. Wang, Y. Lee, X. Wang, N. Zhang, H. Huang, W. Zou, and P. Liu, "Finding unknown malice in 10 seconds: Mass vetting for new threats at the google-play scale," in *Proc. USENIX SEC*, 2015.

[24] Y. Shao, X. Luo, C. Qian, P. Zhu, and L. Zhang, "Towards a scalable resource-driven approach for detecting repackaged android applications," in *Proc. ACSAC*, 2014.

[25] L. Li, D. Li, T. F. Bissyandé, J. Klein, Y. Le Traon, D. Lo, and L. Cavallaro, "Understanding android app piggybacking: A systematic study of malicious code grafting," *IEEE TIFS*, vol. 12, no. 6, pp. 1269–1284, 2017.

[26] M. Li, W. Wang, P. Wang, S. Wang, D. Wu, J. Liu, R. Xue, and W. Huo, "Libd: Scalable and precise third-party library detection in android markets," in *Proc. ICSE*, 2017.

[27] L. Li, T. F. Bissyandé, J. Klein, and Y. Le Traon, "An investigation into the use of common libraries in android apps," in *Proc. SANER*, 2016.

[28] B. Bichsel, V. Raychev, P. Tsankov, and M. Vechev, "Statistical deobfuscation of android applications," in *Proc. CCS*, 2016.

[29] M. Zhang, Y. Duan, H. Yin, and Z. Zhao, "Semantics-aware android malware classification using weighted contextual api dependency graphs," in *Proc. CCS*, 2014.

[30] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, p. 436, 2015.

[31] M. Ou, P. Cui, J. Pei, Z. Zhang, and W. Zhu, "Asymmetric transitivity preserving graph embedding," in *Proc. KDD*, 2016.

[32] B. Perozzi, R. Al-Rfou, and S. Skiena, "Deepwalk: Online learning of social representations," in *Proc. KDD*, 2014.

[33] A. Grover and J. Leskovec, "node2vec: Scalable feature learning for networks," in *Proc. KDD*, 2016.

[34] L. F. Ribeiro, P. H. Saverese, and D. R. Figueiredo, "struc2vec: Learning node representations from structural identity," in *Proc. KDD*, 2017.

[35] H. Dai, B. Dai, and L. Song, "Discriminative embeddings of latent variable models for structured data," in *Proc. ICML*, 2016.

[36] D. Liben-Nowell and J. Kleinberg, "The link-prediction problem for social networks," *Journal of the Association for Information Science and Technology*, vol. 58, no. 7, pp. 1019–1031, 2007.

[37] S. Bhagat, G. Cormode, and S. Muthukrishnan, "Node classification in social networks," in *Proc. Social network data analytics*, 2011.

[38] L. v. d. Maaten and G. Hinton, "Visualizing data using t-sne," *Journal of machine learning research*, vol. 9, no. Nov, pp. 2579–2605, 2008.

[39] S. Rasthofer, S. Arzt, and E. Bodden, "A machine-learning approach for classifying and categorizing android sources and sinks," in *Proc. NDSS*, 2014.

[40] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, and K. Rieck, "Drebin: Effective and explainable detection of android malware in your pocket," in *Proc. NDSS*, 2014.

[41] M. Fan, J. Liu, X. Luo, K. Chen, Z. Tian, Q. Zheng, and T. Liu, "Android malware familial classification and representative sample selection via frequent subgraph analysis," *IEEE TIFS*, vol. 13, no. 8, pp. 1890–1905, 2018.

[42] P. A. Estévez, M. Tesmer, C. A. Perez, and J. M. Zurada, "Normalized mutual information feature selection," *IEEE TNN*, vol. 20, no. 2, pp. 189–201, 2009.

[43] D. Steinley, "Properties of the hubert-arable adjusted rand index," *Psychological methods*, vol. 9, no. 3, p. 386, 2004.

[44] E. B. Fowlkes and C. L. Mallows, "A method for comparing two hierarchical clusterings," *Journal of the American statistical association*, vol. 78, no. 383, pp. 553–569, 1983.

[45] A. Rosenberg and J. Hirschberg, "V-measure: A conditional entropy-based external cluster evaluation measure," in *Proc. EMNLP-CoNLL*, 2007.

[46] M. Rosvall and C. T. Bergstrom, "Maps of random walks on complex networks reveal community structure," *PNAS*, vol. 105, no. 4, 2008.

[47] A. Clauset, M. E. Newman, and C. Moore, "Finding community structure in very large networks," *Physical review E*, no. 6, 2004.

[48] U. Raghavan, R. Albert, and S. Kumara, "Near linear time algorithm to detect community structures in large-scale networks," *Physical review E*, vol. 76, no. 3, 2007.

[49] V. Blondel, J. Guillaume, R. Lambiotte, and E. Lefebvre, "Fast unfolding of communities in large networks," *Journal of Statistical Mechanics: Theory and Experiment*, vol. 2008, no. 10, 2008.

[50] Z. Tian, Q. Zheng, T. Liu, M. Fan, E. Zhuang, and Z. Yang, "Software plagiarism detection with birthmarks based on dynamic key instruction sequences," *IEEE TSE*, vol. 41, no. 12, 2015.

[51] "Dasho," https://www.preemptive.com/products/dasho/overview, 2017.

[52] L. Li, T. Bissyande, D. Octeau, and J. Klein, "Droidra: Taming reflection to support whole-program analysis of android apps," in *Proc. ISSTA*, 2016.

[53] Bangcle Inc., http://www.bangcle.com/, 2018.

[54] Ijiami Inc., http://www.ijiami.cn/, 2018.

[55] L. Xue, X. Luo, L. Yu, S. Wang, and D. Wu, "Adaptive unpacking of android apps," in *Proc. ICSE*, 2017.

[56] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "Sok: (state of) the art of war: Offensive techniques in binary analysis," in *Proc. IEEE S&P*, 2016.

[57] L. Xue, Y. Zhou, T. Chen, X. Luo, and G. Gu, "Malton: Towards on-device non-invasive mobile malware analysis for art," in *Proc. USENIX SEC*, 2017.

[58] J. Xie, S. Kelley, and B. K. Szymanski, "Overlapping community detection in networks: The state-of-the-art and comparative study," *Acm computing surveys*, vol. 45, no. 4, p. 43, 2013.
[59] G. Suarez-Tangil, J. E. Tapiador, P. Peris-Lopez, and J. Blasco, "Dendroid: A text mining approach to analyzing and classifying code structures in android malware families," *Expert Systems with Applications*, vol. 41, no. 4, 2014.
[60] Y. Feng, O. Bastani, R. Martins, I. Dillig, and S. Anand, "Automated synthesis of semantic malware signatures using maximum satisfiability," in *Proc. NDSS*, 2017.
[61] A. Narayanan, M. Chandramohan, R. Venkatesan, L. Chen, Y. Liu, and S. Jaiswal, "graph2vec: Learning distributed representations of graphs," *arXiv preprint arXiv:1707.05005*, 2017.