2016 IEEE 27th International Symposium on Software Reliability Engineering

Frequent Subgraph based Familial Classification of Android Malware

Ming Fan*, Jun Liu*, Xiapu Luo†, Kai Chen‡, Tianyi Chen*, Zhenzhou Tian*, Xiaodong Zhang*, Qinghua Zheng*, Ting Liu*
*MOEKLINNS Lab, Department of Computer Science and Technology, Xi'an Jiaotong University, 710049, China
†Department of Computer, The Hong Kong Polytechnic University, 999077, China
‡State Key Laboratory of Information Security, Chinese Academy of Sciences, 100093, China

***Abstract*—The rapid growth of Android malware poses great challenges to anti-malware systems because the sheer number of malware samples overwhelm malware analysis systems. A promising approach for speeding up malware analysis is to classify malware samples into families so that the common features in malwares belonging to the same family can be exploited for malware detection and inspection. However, the accuracy of existing classification solutions is limited because of two reasons. First, since the majority of Android malware is constructed by inserting malicious components into popular apps, the malware's legitimate part may misguide the classification algorithms. Second, the polymorphic variants of Android malware could evade the detection by employing transformation attacks. In this paper, we propose a novel approach that constructs frequent subgraph (*fregraph*) to represent the common behaviors of malwares in the same family for familial classification of Android malware. Moreover, we propose and develop FalDroid, an automatic system for classifying Android malware according to *fregraph*, and apply it to 6,565 malware samples from 30 families. The experimental results show that FalDroid can correctly classify 94.5% malwares into their families using around 4.4s per app.**

***Keywords*-Android malware; familial classification; frequent subgraph; sensitive API; clustering;**

I. Introduction

Being the most popular mobile operating system, Android has occupied 82.8% market share in the second quarter of 2015[1]. Meanwhile, Android has become the major target of 97% of mobile malware[2]. A recent security report shows that on average 51,342 new malware samples were captured per day in 2015[3]. Since it takes time to analyze each malware sample [1], [2], the sheer number of malware samples overwhelm the malware analysis systems.

Since the majority of new malware samples are the polymorphic variants of known malware [3], [4], we can classify them into various families and then extract the common features of each family to speed up the malware analysis. However, it is challenging to accomplish the familial classification of Android malware because of two reasons.

First, it is non-trivial to accurately separate the malicious components and the legitimate part in the majority of Android malware that are repackaged popular apps [5], [6]. Zhou et al. [3] found that 86% Android malware samples are repackaged ones produced by injecting malicious components into legitimate apps. Since the injected malicious components are hidden within the functionalities of popular apps and they usually constitute only a small portion of the repackaged app. It is difficult for existing features, such as system calls [7] and sensitive path [8], to differentiate the legitimate part and the malicious components.

Second, the polymorphic variants of Android malware in the same family conduct the same malicious activities with different implementations. Therefore, existing classification solutions [9], [10] that seek for an exact match for a given specification can be easily evaded by such malware. For example, Fig.1 illustrates the different implementations of the same functionality (i.e., get the number of voice mail) in two malware samples. They belong to the same family called *geinimi*, and such bot-like malware steals personal information and sends it to a remote server. There are three major differences (highlighted in red) in these two implementations. First, the structure of class names is different. Second, the two functions' arguments are different. One argument starts with a service, one of the four basic components of Android framework, whereas the other one starts with an object of the class *rally/e*. Third, the former method has two more statements (including one more invocation) than the latter.

To tackle the above two challenges, in this paper, we propose a novel approach based on frequent subgraph (*fregraph*) by exploiting two observations. First, Android malware usually invokes sensitive APIs that operate on sensitive data to perform malicious activities. Second, malware and its variants in the same family invoke sensitive APIs following similar patterns even if their codes may be obfuscated.

Exploiting the two observations, we first distill program semantics into a function call graph representation and assign different weights to different sensitive APIs with a TF-IDF-like approach.

Then, we propose two key techniques to solve the challenges (see Section II-B for details). 1) We propose a clustering-based approach to extract common malicious behaviors in the same family. By doing so, we can exclude the legitimate part in malware from familial classification. 2) We propose a weighted sensitive APIs-based approach to calculate the similarity between graphs which can detect

[1]http://www.idc.com/prodserv/smartphone-os-market-share.jsp
[2]http://goo.gl/MYDBKC
[3]http://zt.360.cn/1101061855.php?dtid=1101061451&did=1101593997

2332-6549/16 $31.00 © 2016 IEEE
DOI 10.1109/ISSRE.2016.14

```
.class public final Lcom/geinimi/c/f;
.method public constructor <init> (Lcom/geinimi/Adservice;)V
invoke-virtual {p0}, Landroid/telephony/
 TelephonyManager;>getVoiceMailNumber()Ljava/lang/String;
move-result-object v0
sput-object v0, Lcom/geinimi/c/f;->t:Ljava/lang/String;
new-instance v0, Landroid/os/Build;
invoke-direct {v0}, Landroid/os/Build;-><init>()V
sget-object v0, Landroid/os/Build;->MODEL:Ljava/lang/String;
……
```

```
.class public final Lcom/xlabtech/MonsterTruckRally/rally/e/k;
.method public constructor <init> (Lcom/xlabtech/MonsterTruckRally/rally/e;)V
invoke-virtual {p0}, Landroid/telephony/
TelephonyManager;>getVoiceMailNumber()Ljava/lang/String;
move-result-object v0
sput-object v0, Lcom/xlabtech/MonsterTruckRally/rally/e/k;->v:Ljava/lang/
String;
sget-object v0, Landroid/os/Build;->MODEL:Ljava/lang/String;
……
```

Figure 1: Different implementations of the same functionality in two malware samples in *geinimi* family.

homogeneous malicious behaviors while tolerating minor differences of implementation. It is worth noting that sensitive APIs constitute only a small portion of the whole Android APIs and they cannot be easily obfuscated.

Finally, based on these two key techniques we construct frequent subgraph (*fregraph*) to represent the common malicious behavior of malwares in the same family. Moreover, we propose and develop FalDroid, an automatic system for classifying Android malware according to *fregraph*, in 7,400 lines of Java code and 900 lines of Python code. By applying FalDroid to 6,565 malwares in 30 different families, we find that FalDroid can correctly classify 94.5% malwares into their families around 4.4s per app on average.

In summary, our major contributions include:

(i) We propose fregraph, a new feature for representing common behaviors of malwares in the same family, and employ it to conduct malware familial classification.
(ii) We propose a novel weighted sensitive APIs-based graph matching approach, which can detect homogeneous malicious behaviors of malwares in the same family while tolerating minor differences of implementation.
(iii) We design and implement FalDroid, an automatic system that can handle large scale of Android malwares for familial classification with high accuracy.
(iv) We conduct extensive experiments to evaluate FalDroid. The experimental results show that it can achieve 94.5% accuracy and just needs around 4.4s to process an app.

The remainder of this paper is organized as follows. Section II details the design of FalDroid and its algorithms. Section III reports the experimental results. After discussing the limitations and threats to validity in Section IV, we introduce the related work in Section V and conclude the paper with future work in Section VI.

II. Methodology

The overall architecture of FalDroid is shown in Fig.2, which is comprised of three main stages.

In the *Preprocessing* stage, in order to differentiate the importance of sensitive APIs to different families, each sensitive API in different families is assigned with different weights using a TF-IDF-like approach. Then, the program semantics is distilled into a function call graph representation. After that, the graph is simplified into a **sensitive API related graph** with the identified sensitive API nodes.

In the *Fregraph Generation* stage, the sensitive API related graph is first divided into a set of subgraphs with community detection algorithm proposed by Rosvall et al. [11]. The subgraph with sensitive APIs (**sensitive subgraph**) that is used by most samples in one family is defined as the **frequent subgraph (fregraph)** of the specific family.

In the *Feature Construction* stage, the fregraphs of all known families are embedded into a feature space and a classifier is generated with machine learning algorithm.

A. Preprocessing

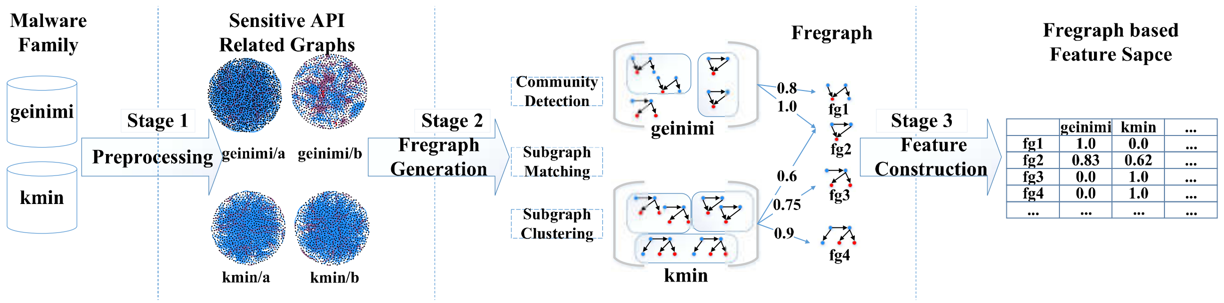
Android apps are normally written in Java and compiled to Dalvik code (DEX). All Java code is contained in the *classes.dex* file. The compiled code and resources are packaged as Android package (APK). With mature disassemble tools such as *apktool*[4], we are able to get the Dalvik code from the APK.

As stated in the first observation, Android malware usually invokes sensitive APIs that operate on sensitive data to perform malicious activities. Therefore, malwares in different families with different malicious behaviors utilize sensitive APIs in different ways.

To obtain the set of sensitive APIs, we rely on the work of Rasthofer et al. [12]. They proposed *SuSi*, a novel machine-learning guided approach for identifying *Sources* and *Sinks* directly from any Android API. *Sources* are APIs that return sensitive data such as *getDeviceId()* which returns the IMEI of a phone, and *Sinks* are APIs that use sensitive data as arguments such as *sendTextMessage()* which receives both the message text and the phone number. There are 18,044 *Sources* and 8,278 *Sinks* in total.

1) Weight Assignment of Sensitive APIs: In order to differentiate the importance of sensitive APIs, our approach assigns weights for each sensitive API in different families. To achieve this, 6,565 malware samples in 30 families are collected from VirusShare[5] (see Section III-A for details). Here we use three terms of a sensitive API s in family f to help us understand its usages in different families.

[4]https://code.google.com/p/android-apktool/
[5]http://virusshare.com/



	geinimi	kmin	...
fg1	1.0	0.0	...
fg2	0.83	0.62	...
fg3	0.0	1.0	...
fg4	0.0	1.0	...
...	...	...	...

Figure 2: The overall architecture of FalDroid

Table I: Six Sensitive APIs' tn And Their Corresponding sn, sp And w In Three Families

sensitive API	tn (6,565)	*geinimi*($fn = 105$)			*plankton*($fn = 626$)			*droidkungfu*($fn = 725$)		
		sn	sp	w	sn	sp	w	sn	sp	w
getVoiceMailNumber()	138	105	1.000	1.677	19	0.033	0.055	0	0.000	0.000
getDeviceSoftwareVersion()	171	105	1.000	1.584	0	0.000	0.000	0	0.000	0.000
getDeviceId()	5,927	105	1.000	0.044	626	1.000	0.044	725	1.000	0.044
getLine1Number()	3,903	105	1.000	0.226	334	0.536	0.121	667	0.920	0.208
sendTextMessage()	1,373	105	1.000	0.680	24	0.038	0.026	23	0.032	0.022
divideMessage()	302	6	0.057	0.076	2	0.003	0.004	4	0.006	0.008

- $sn(s, f)$: number of samples that invoke the sensitive API s in family f.
- $sp(s, f)$: percent of samples that invoke the sensitive API s in family f, $sp(s, f) = \frac{sn(s,f)}{fn(f)}$, where $fn(f)$ denotes the number of samples in f.
- $w(s, f)$: weight of sensitive API s in family f.

In addition, we use $tn(s)$ to denote the number of samples that invoke s in all families, it is obtained by $tn(s) = \sum_{f_j \in F} sn(s, f_j)$, where $F = \{f_j | 1 \leq j \leq m, m = 30\}$ denotes the set of all families.

TABLE I lists six sensitive APIs' tn and their corresponding sn, sp and w in three different families, respectively. There are two main observations we found from TABLE I.

- The usages of different sensitive APIs in the same family are quite different. For example, *sendTextMessage()* is used by all the 105 samples in *geinimi* family while *divideMessage()* is used by only 6 samples.
- There are some sensitive APIs that are used by most malware samples. For example, *getDeviceId()* is used by all the samples in the three families.

Based on the two observations, the weight of a sensitive API in one family should be positively related with its sp in the family, and be negatively related with its tn. Utilizing the idea of TF-IDF [13]–[15] for reference: the term frequency (TF) measures the number of sensitive API s appears in family f and the inverse document frequency (IDF) measures whether s is common or rare across all the malware samples. The weight of sensitive API s in family f is calculated as:

$$w(s, f) = sp(s, f) * \lg \frac{\sum_{1 \leq j \leq m} fn(f_j)}{tn(s)}. \qquad (1)$$

As listed in TABLE I, by using Eq.(1) the weight of *sendTextMessage()* is 0.680 in *geinimi* family while *divideMessage()* is only assigned with 0.076 since the sp of *sendTextMessage()* is much higher than that of *divideMessage()*. Moreover, *getDeviceId()* is used by all samples in the three families, it is assigned with only 0.044. Intuitively, the results show that the weight assignment of our approach can well measure how important a sensitive API to one family.

2) Construction of Sensitive API Related Graph: Based on the extracted callers and callees from the Dalvik code, we distill an app's program semantics into a function call graph representation, which contains all possible executed traces, as well as the structure information to depict app behaviors. It is represented as $G = (V, E)$.

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

In general, there are thousands of nodes in the whole graph of a given app. Analyzing the whole graph is neither effective (the malicious part is hidden behind the legitimate part) nor efficient (too many nodes and edges to analyze). Thus, excluding the nodes that have no path to sensitive n-odes can effectively reduce the complexity of graph analysis. We simplify the function call graph G into the sensitive API Related Graph G'.

Definition 1: **Sensitive API Related Graph (SARG)** : it is a subgraph of function call graph and in which all nodes have directed path to sensitive API nodes.

The SARG $G' = (V', E')$ can be obtained with Eq.(2) and Eq.(3), in which the function $dis(v_j, v_i)$ returns the shortest path length from node v_j to node v_i. In general, the size of SARG is reduced by about 80% compared to the original function call graph.

$$V_g = \{v_j | 0 < dis(v_j, v_i) < n, v_j \in V, v_i \in V_s\} \tag{2}$$

$$V' = V_s \cup V_g, E' = (V' \times V') \cap E \tag{3}$$

B. Fregraph Generation

In this section, we introduce our two key techniques: a clustering-based approach to extract common malicious behaviors in the same family (see Section II-B1 and Section II-B3 for details) and the sensitive APIs-based graph matching approach to calculate similarity between subgraphs (see Section II-B2).

1) Community Detection: After the stage of preprocessing, there is an observation that discovered from generated SARGs of a family: *even the big portion of the whole SARGs are different in various apps, they have similar subgraphs which constitute only a small portion of the whole SARGs*. Fig.3 presents two SARGs of two different samples in *geinimi* family, in which the red nodes denote the sensitive API nodes and the blue nodes denote the general nodes. The red edge denotes that its callee function is a sensitive API. The two SARGs contain 267 and 715 nodes, respectively. The subgraphs marked in red circles are nearly the same which implement similar behaviors and the other parts are totally different. It is not efficient to mine the similar subgraphs from the two whole SARGs since the graph isomorphism problem is a NP complete problem. Thus we divide the SARGs into a set of even smaller subgraphs to help locate the similar ones and reduce the complexity of graph similarity calculation.

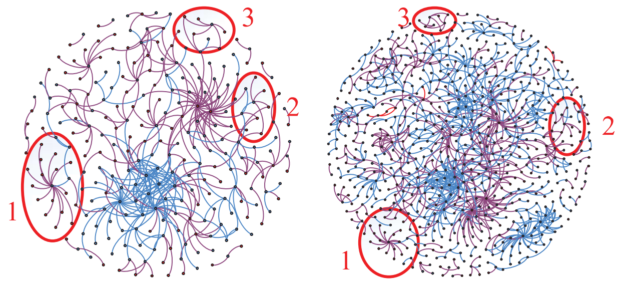


Figure 3: Two SARGs of two malware samples in family *geinimi* and three similar subgraphs marked in red circles

As introduced in [16], [17], one network feature that has been emphasized in recent work is community structure, the gathering of vertices into groups such that there is a higher density of edges within groups than between them. Prior works [18], [19] have demonstrated that the function call graph is also one typical network which can be used to detect its community structures. The functions of software in one community structure have strong connections and they are always located in same class or package to implement software functionalities together. Leveraging the characteristic of software network, we divide the SARG into a set of subgraphs with community detection algorithm [11] which is based on the probability flow of random walks on the network. With algorithm [11] more subgraphs with less nodes are generated than other algorithms [20], [21], which effectively reduces the complexity of graph matching.

Moreover, most subgraphs have no relation with sensitive data which might do little help for malware classification. Thus we introduce the definition of sensitive subgraph.

Definition 2: **Sensitive Subgraph**: it is a subgraph which contains at least one sensitive API node. Sensitive subgraph sg in family f also contains a weight value $w(sg, f)$ to denote its importance to the family. It is represented as:

$$w(sg, f) = \sum_{v_i \in V_s(sg)} w(v_i, f), \tag{4}$$

where $V_s(sg)$ denotes the set of sensitive APIs in sg.

2) Graph Matching: To quantify the similarity of two sensitive subgraphs, we propose a novel weighted sensitive APIs-based approach which can detect homogeneous app behaviors of malwares in the same family while tolerating minor differences of implementation.

For two sensitive subgraphs (sg_1 and sg_2) in family f, their similarity $sim_f(sg_1, sg_2)$ is calculated with three steps.

***Step** 1:* **construct distance matrixes for two subgraphs.** We first construct two distance matrixes for the two subgraphs, whose sizes are both $t \times t, t = |V_s(sg_1) \cup V_s(sg_2)|$. The distance matrix of sg_1 is obtained with Eq.(5) and the distance matrix of sg_2 is calculated similar as sg_1 where the distance is calculated in sg_2. Note that in Eq.(5), the graph is regarded as an undirected graph while calculating the shortest path length between two nodes ($dis'(v_i, v_j)$), which is different from that in Section II-A2.

$$Matrix_1[i,j] = \begin{cases} dis'(v_i, v_j) & i \neq j, v_i, v_j \in V_s(sg_1) \\ 0 & otherwise \end{cases} \tag{5}$$

***Step** 2:* **calculate the similarity of sensitive nodes.** In our work, we only focus on the sensitive nodes. The similarity of the same sensitive node v_i in two subgraphs is represented as $ns(v_i)$. It depends on the weight of the sensitive node ($w(v_i, f)$) and the distances from it to other sensitive nodes. $ns(v_i)$ is obtained with Eq.(6)-(8). $\overrightarrow{vec(v_i, sg_2)}$ is obtained similar as $\overrightarrow{vec(v_i, sg_1)}$ in which the corresponding elements are calculated in sg_2.

$$ns(v_i) = cos(\overrightarrow{vec(v_i, sg_1)}, \overrightarrow{vec(v_i, sg_2)}) \tag{6}$$

$$\overrightarrow{vec(v_i, sg_1)} = \langle e(v_i, v_1), \ldots, e(v_i, v_t) \rangle \tag{7}$$

$$e(v_i, v_j) = \begin{cases} \frac{w(v_j, f)}{dis'(v_i, v_j)} & a(v_i, v_j) \neq 0, 1 \leq j \leq t \\ 0 & otherwise \end{cases} \tag{8}$$

Step 3: **calculate the similarity of subgraphs.**

Based on the above two steps, $sim_f(sg_1, sg_2)$ is calculated with Eq.(9).

$$sim_f(sg_1, sg_2) = \frac{\sum_{v_i \in V_s(sg_1) \cap V_s(sg_2)} (w(v_i, f) * ns(v_i))}{\sum_{v_i \in V_s(sg_1) \cup V_s(sg_2)} w(v_i, f)} \tag{9}$$

The similarity ranges from 0 to 1 where the maximum value 1 means that the two subgraphs implement the exact same behaviors, while the minimum value 0 means that they are totally different. Note that the similarity between sg_1 and sg_2 is not higher than $\frac{min(w(sg_1,f),w(sg_2,f))}{max(w(sg_1,f),w(sg_2,f))}$, which can be used to reduce the number of pair-wise graph matching in latter work.

3) Subgraph Clustering: With the effective and efficient graph matching approach, we generate the fregraph based on the clustering of subgraphs without prior knowledge.

Algorithm 1 highlights the step of generating fregraph with the input of a set of sensitive subgraphs in family f and the similarity threshold value β. In our work, β is set to 0.8 which indicates that if the similarity of two sensitive subgraphs is higher than 0.8, they are considered as the same one and are put into the same cluster. In the algorithm, $\overline{sim_f}(sg_i, c_j)$ returns the average similarity of sg_i with all the sensitive subgraphs in cluster c_j. It is worth noting that the cluster c_j is a multiset of subgraphs which allows multiple instances of the multiset's elements.

Algorithm 1 Clustering of Sensitive Subgraphs

```
Input:
    $SG_f = \{sg_i | 1 \le i \le |SG_f|\}$   // the set of sensitive subgraphs in family $f$
    $\beta = 0.8$   // the similarity threshold value
    $C = \{c_j | 1 \le j \le p, p \in N\}$   // the set of output clusters
Output:
 1: $C \leftarrow \emptyset, p \leftarrow 0$
 2: for each $sg_i$ in $SG_f$ do
 3:   if $C \ne \emptyset$ then
 4:     $c_u = argmax_{c_j \in C} \overline{sim_f}(sg_i, c_j)$
 5:     if $\overline{sim_f}(sg_i, c_u) \ge \beta$ then
 6:       $c_u = c_u \cup \{sg_i\}$
 7:     else
 8:       $p = p + 1, c_p = \{sg_i\}$
 9:       $C = C \cup \{c_u\}$
10:     end if
11:   else
12:     $p = 1, c_1 = \{sg_i\}, C = \{c_1\}$
13:   end if
14: end for
15: return $C$
```

Definition 3: **Fregraph**: given a cluster set $C = \{c_1, c_2, \ldots, c_p\}$ in family f and a sensitive subgraph sg. The support of sg in family f is $sup_f(sg) = \frac{|c_j|}{fn(f)}, sg \in c_j$ and $fn(f)$ denotes the number of malwares in family f. A fregraph fg is a sensitive subgraph whose support $sup_f(sg)$ is no less than the minimum support threshold θ.

Fig.4 presents an example of generated fregraph, which is used by all the malware samples in *geinimi* family and its support is 1.0. According to the semantic meanings of the sensitive APIs in the graph, it collects various personal information such as phone number, IMEI.

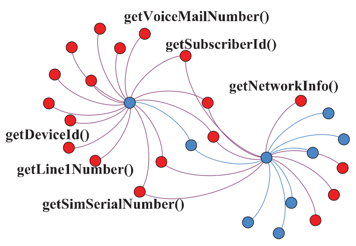


Figure 4: An example of generated fregraph

C. Feature Construction

To enable malware classification, all the fregraphs in known families are embedded into a feature space. However, there are some fregraphs which belong to more than one family. Therefore there is a map between fregraphs and families. Fig.5 is an example of a map between four fregraphs and three malware families. The number between a fregraph and a family denotes the support of fregraph to its corresponding family. Intuitively, the fregraphs which belong to several families such as fg_2 should have lower contributions to malware classification than the ones which belong to only one family such as fg_3.

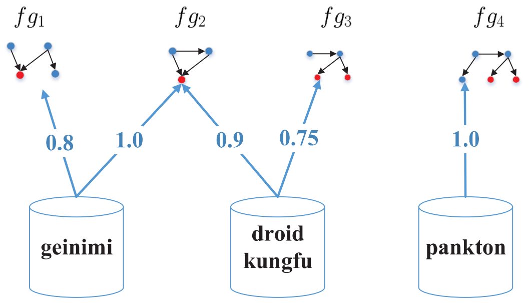


Figure 5: An example of map between four fregraphs and three malware families

The contribution of a fregraph to malware classification is represented as $cb(fg)$ and it is calculated based on the entropy value.

$$cb(fg) = \sum_{f_j \in F} p(f_j|fg) \log_2 p(f_j|fg), \tag{10}$$

where $p(f_j|fg)$ denotes the distribution probability the app belongs to the family f_j if it contains fregraph fg.

$$p(f_j|fg) = \frac{sup_{f_j}(fg)}{\sum_{f_i \in F} sup_{f_i}(fg)} \tag{11}$$

Then the contributions of all sensitive subgraphs are normalized as:

$$cb'(fg) = \frac{cb(fg) - cb_{min}}{cb_{max} - cb_{min}}. \tag{12}$$

Next, the feature vector of each malware is constructed for the purpose of classification. For each fregraph contained in the malware, its corresponding value in the vector is set to the contribution value of the fregraph, or it is set to 0. For example, the feature vector of a malware can be represented as $\langle cb'(fg_1), cb'(fg_2), 0, 0, \ldots \rangle$. Specifically, for the known malware samples in training dataset, their family labels are attached with the feature vector so that the classifiers can understand the discrepancy between different malware families. Once the feature vectors for the training malware samples are generated, a classifier can be trained with different machine learning algorithms such as SVM.

III. Evaluation

To evaluate the effectiveness of FalDroid, first we introduce the dataset and metrics. We then investigate the following three main research questions:

***RQ** 1: Can our approach classify the new malware into its family with a high accuracy?* For this purpose, we evaluate our approach on our dataset and compare it with three baseline approaches (see Section III-B for details).

***RQ** 2: Can our approach handle large scale of malwares?* For this purpose, we analyze the statistic of generated graphs and the run-time overhead of our approach (see Section III-C for details).

***RQ** 3: Can our approach be resilient to polymorphic variants and typical obfuscation techniques?* For this purpose, we compare the effectiveness of our sensitive APIs-based graph matching approach with GED and evaluate FalDroid on obfuscated apps and packed apps (see Section III-D for details).

A. Dataset and Metrics

All the malware samples in our dataset are downloaded from Virusshare and each of them has been uploaded to VirusTotal[6] which is a system that contains more than 50 anti-virus scanners. The anti-virus scanners such as AVL, McAfee and ESET-NOD32 are based on signature database detection and they are useful for known malwares but less effective for the unknown ones. There are two drawbacks about the results provided by the anti-virus scanners: 1) the family labels defined by each anti-virus scanner have minor differences such as *Plankton/Plangton/planktonc*; 2) the results of the anti-virus scanners seldom reach a consensus.

[6]http://www.virustotal.com/en/

To fight against the two drawbacks we first construct a family label dictionary based on string edit distance. We then label the malware with the family name which is the consistent result returned by more than half of the anti-virus scanners. Therefore, 6,565 malware samples in 30 families are labeled and they are listed in TABLE A1 in the Appendix. For each family, two-thirds of malwares are used as training samples and the remaining malwares are used as testing samples.

The metrics used to measure our classification results are shown in TABLE A2 in the Appendix. Our system is implemented in 7,400 lines of java code and 900 lines of python code. Our experiments are conducted on a quad-core 3.20 GHz PC operating on Ubuntu 14.04(64 bit) with 16GB RAM and 1TB hard disk.

B. Effectiveness of FalDroid

1) Performance with Four Different Classifiers: Our approach is evaluated with four different classifiers: SVM (linear kernel), Decision Tree (C4.5), K-NN (k=1) and Random Forest. For each family it has its own detection result, we use the term classification accuracy to denote the weighted percent of malwares that are correctly classified into their families. Fig.6 presents classification accuracies of the four classifiers for different support thresholds from 0.1 to 0.9. We can draw three conclusions From Fig.6:

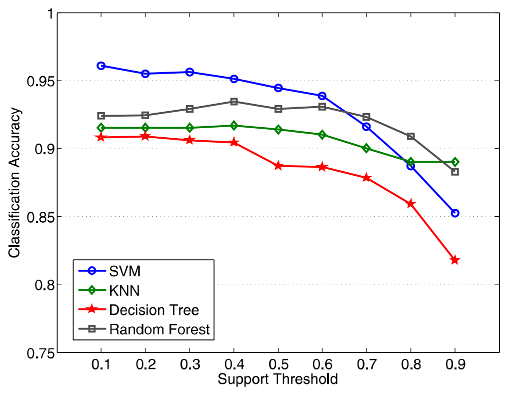


Figure 6: Classification accuracies of FalDroid with four different classifiers

(i) All of the four classifiers can get a good result which is higher than 80%.
(ii) SVM performs the best among the five classifiers. Its accuracy can achieve 0.961 when the threshold is 0.1.
(iii) The performance decreases with the increase of support threshold especially when it is higher than 0.6. As illustrated in Fig.7, with the increase of the support threshold from 0.1 to 0.9, the number of features decreases. Specifically, there are no fregraphs for some families when the support threshold is higher than 0.6 which causes the lower accuracy.

Table II: Classification Performance For 30 Families With SVM When The Support Threshold Is Set To 0.5

Malware Family	TPR	FPR	p	r	F	AUC	Malware Family	TPR	FPR	p	r	F	AUC
adwo	0.879	0.002	0.946	0.879	0.911	0.938	hongtoutou	1	0	1	1	1	1
airpush	0.600	0.001	0.833	0.600	0.698	0.799	iconosys	0.98	0	1	0.98	0.99	0.99
basebridge	0.950	0.002	0.960	0.950	0.955	0.974	imlog	1	0	1	1	1	1
boqx	0.375	0.001	0.667	0.375	0.480	0.687	kmin	0.976	0	1	0.976	0.988	0.988
boxer	1	0	1	1	1	1	kuguo	0.924	0.003	0.940	0.924	0.932	0.960
clicker	1	0	1	1	1	1	mobiletx	1	0	1	1	1	1
dowgin	0.908	0.006	0.933	0.908	0.921	0.951	pjapps	0.963	0	1	0.963	0.981	0.981
droiddreamlight	0.941	0.002	0.865	0.941	0.901	0.969	plankton	0.981	0.002	0.986	0.981	0.983	0.99
droidkungfu	0.975	0.011	0.918	0.975	0.988	0.988	smskey	0.972	0.001	0.946	0.972	0.959	0.986
fakedoc	1	0	1	1	1	1	smsreg	0.825	0.006	0.733	0.825	0.776	0.910
fakeinst	0.996	0.002	0.987	0.996	0.991	0.997	utchi	1	0	1	1	1	1
fakeplay	1	0.001	0.875	1	0.933	1	waps	0.959	0.005	0.950	0.959	0.955	0.977
geinimi	1	0	1	1	1	1	youmi	0.711	0.003	0.794	0.711	0.750	0.854
gingermaster	0.893	0.010	0.838	0.893	0.865	0.942	yzhc	1	0	1	1	1	1
golddream	0.963	0	1	0.963	0.981	0.981	zitmo	1	0	0.909	1	0.952	1
Avg.	**0.945**	**0.004**	**0.944**	**0.945**	**0.944**	**0.971**							

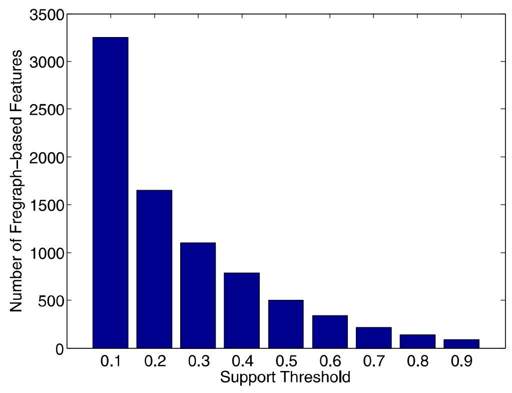


Figure 7: Number of fregraph-based features for different support thresholds

Moreover, when the support threshold is set to 0.5, the accuracy of SVM decreases by 1.6% while the number of features decreases by 85% compared to the result when the threshold is set to 0.1. Thus, we select SVM as our classifier and set the support threshold to 0.5 in latter experiments.

TABLE II shows the detail classification results for the 30 families when the support threshold is 0.5. Most families get a TPR value higher than 0.9. Furthermore, 11 families even get a TPR value as high as 1, and a FPR value as low as 0, which means that all their samples are correctly classified and no other malware samples are incorrectly classified into such families. However, there are still some families such as *boqx* that performs not as well as others, since it only contains 2 unique fregraph-based features. In summary, FalDroid performs well for most families.

FalDroid can correctly classify 94.5% malwares into their families on our dataset.

2) Comparison Result with Baseline Approaches: We compare our approach with three baseline approaches [4], [22], [23] on a widely used benchmark dataset,which is provided by Android Malware Genome Project [3] and has been tested in the three approaches. The descriptions of the three approaches are listed as below:

- Suarez et al. [23] proposed Dendroid, which automatically classifies malware and analyzes families based on the code structures.
- Feng et al. [4] proposed Apposcopy, which extracts data-flow and control-flow properties of a new app to identify the family it belongs to.
- Zhang et al. [22] proposed a semantic-based approach called DroidSIFT that classifies Android malware via API dependency graphs.

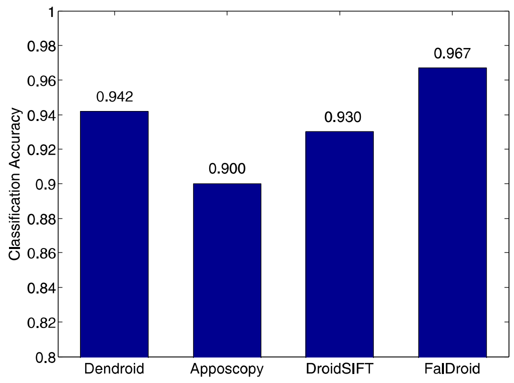


Figure 8: Classification accuracies of FalDroid and three baseline approaches on same dataset

The comparison result is illustrated as Fig.8. FalDroid performs better than the three baseline approaches on the same dataset. Most related to our work is DroidSIFT and there are two major differences between them. First, DroidSIFT needs a set of graphs extracted from benign apps to remove the common ones extracted from malware while FalDroid uses a clustering-based approach to mine fregraphs only from malwares in families to depict their commonalities. It is hard for DroidSIFT to ensure the completeness of the benign graph set. Second, DroidSIFT calculates the similarity between graphs based on an improved weighted GED while FalDroid uses a novel weighted sensitive APIs based approach, which is more robust and effective than GED for detecting homogeneous app behaviors while tolerating minor differences of implementation (see Section III-D1 for details).

FalDroid performs better than three existing baseline approaches on same dataset.

C. *Efficiency of FalDroid*

1) Summary of Generated Subgraphs: Fig.9 summarizes the statistic information of sensitive subgraphs. The left figure illustrates the cumulative distribution function (CDF) for the number of sensitive subgraphs generated by community detection algorithm. On average, 90 sensitive subgraphs are generated for each malware, and more than 90% of the malware samples contain less than 200 sensitive subgraphs. The right figure illustrates the CDF for the number of nodes in each sensitive subgraph. On average, there are 10 nodes in sensitive subgraph. Furthermore, there are about 750,000 sensitive subgraphs in total and only 0.8% of which contain more than 50 nodes. These facts serve as the basic requirements for the scalability of our approach, since the run-time performance of graph matching depends on the number of sensitive subgraphs and the nodes in them.

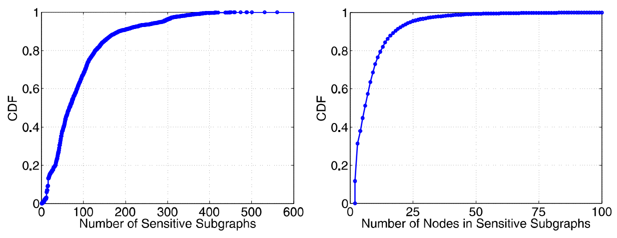


Figure 9: CDFs for number of sensitive subgraphs and number of nodes in sensitive subgraphss

2) Run-time overhead: Our approach consists of three main procedures when analyzing a new malware:

- **Graph Construction**: the APK file is disassembled to generate the Dalvik code and a SARG is constructed.
- **Community Detection**: the SARG is divided into a set of subgraphs with community detection algorithm.
- **Feature Construction**: the subgraphs of the new malware are matched with the fregraph-based features to generate a feature vector.

The runtime overheads of the three main procedures are illustrated in Fig.10 and Fig.11. It costs 2.4s on average to construct the graph model for a given APK file. In the procedure of community detection, 1.5s is needed on average to divide the graph into a set of subgraphs while 12.5s is needed if the graph is not simplified with Eq.(2) and Eq.(3). The run-time overhead of feature construction depends on the size of feature space. With the increase of support threshold, the corresponding cost of feature construction decreases. It costs only 0.5s on average to generate the feature vector of a new malware when the threshold is 0.5.

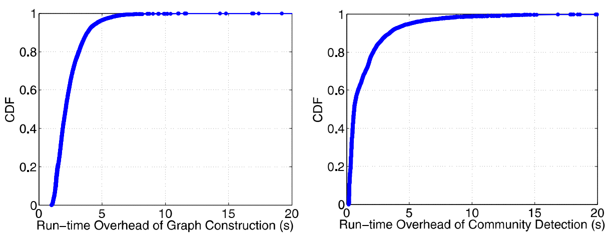


Figure 10: CDFs of run-time overhead for graph construction and community detection

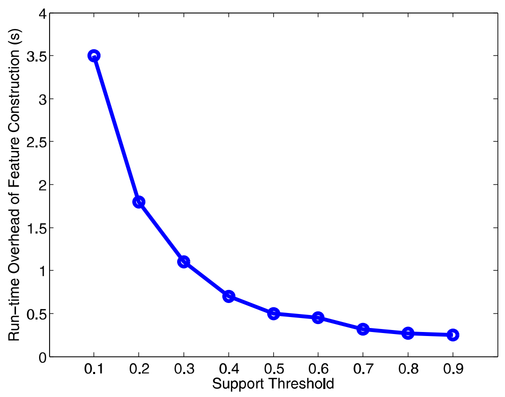


Figure 11: Run-time overhead of feature construction for different support threshold

The average run-time overhead of FalDroid is 4.4s and 95% of the samples are processed within 10s. FalDroid needs less time than DroidSIFT [22] and Apposcopy [4] which cost 175.8s and 275s to analyze an app on average, respectively.

The low run-time overhead makes FalDroid be able to handle large scale of malwares.

D. *Resilience of FalDroid*

1) Resilience to Polymorphic Variants: In our work, to fight against the polymorphic variants we perform graph matching with a novel sensitive APIs-based approach. Here, we evaluate the effectiveness of our graph matching approach and compare it with the GED which has been widely used by existing works [22], [24], [25].

The GED metric depends on the choice of edit operations and the cost involved with each operation (node insertion/deletion, edge insertion/deletion and node relabeling). Specifically, we do not consider the cost of relabeling since the label of node is easily changed by obfuscation techniques. However, there is another issue that if two subgraphs have similar structure but their labels are different (not obfuscated), they will be regarded as the similar ones with GED.

To this end, we manually construct two subgraph sets:

- *Similar set*, which consists of 50 sensitive subgraphs generated from 50 different malwares in geinimi family. These 50 sensitive subgraphs implement similar malicious behaviors and they are regarded as the same one even they have minor differences.
- *Dissimilar set*, which consists of 50 sensitive subgraphs generated from one malware. Any two of them do not contain same sensitive node, which means that they implement totally different behaviors.

In both the two subgraph sets, each subgraph is matched with others, and there are 49*49 pair-wise graph matching similarities.

Fig.12 presents the effectiveness of our work and GED in *similar set* (illustrated in left figure) and *dissimilar set* (illustrated in right figure). In the *similar set*, with our approach all the similarities are higher than 0.8 (selected as the similarity threshold when clustering subgraphs). For GED, about 10% of similarities are lower than 0.8. In the *dissimilar set*, all the similarities are 0 with our approach. However, the similarities range from 0.1 to 1 and there are about 3% of similarities higher than 0.8 with GED.

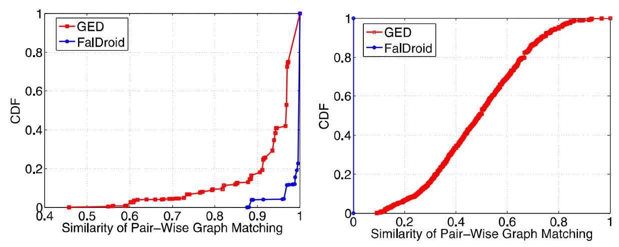


Figure 12: Effectiveness of sensitive APIs-based graph matching and GED in *similar set* and *dissimilar set*

Furthermore, our approach costs less than 1ms to finish one pair-wise graph matching on average while GED needs about 7ms. The low run-time overhead of our approach makes it scalable for the clustering of thousands of subgraphs.

In summary, our approach can better uncover homogeneous behaviors while tolerating minor differences compared to GED.

FalDroid is resilient to polymorphic variants based on the sensitive APIs-based graph matching.

2) Resilience to Code Obfuscation and Packer: We evaluate the resilience of FalDroid to typical local obfuscation techniques such as renaming of the user-defined functions, instruction reordering and branch inversion. To this end, we employ *Proguard*[7] to obfuscate apps from source codes. The results show that even though for the obfuscated ones there might be several functions less than the source apps, their similarities of graph matching are still 1.

Then we use existing packers such as *APKProtect*[8] and *Bangcle*[9] which pack apps with a shell to fight against the de-compilation tools. It fails to get the Dalvik code by *apktool* directly. However, we can successfully remove the shell with the tool *DexDumper* [26] and can also get the Dalvik code. Therefore, such packers have little influence on our approach.

FalDroid is resilient to typical code obfuscation techniques and packers.

IV. LIMITATIONS AND THREATS TO VALIDITY

Like any empirical study, our evaluation is subject to threats to validity, many of which are induced by limitations of our approach. The most important threats and limitations are listed as below.

External validity. Due to the current limitations in our implementation and testing environment, our dataset consists of 6,565 malware samples from 30 families labeled with the results of VirusTotal, which may not be absolutely accurate. Furthermore, only 30 families may not be representative of the entire malware families. We plan to collect more malware families in the future to address such limitations.

Native code. In this work only the Dalvik code is analyzed by our approach. We do not consider the native code. Hence, the malicious behavior implemented in the native code is not reflected in our graph model and we are not able to correctly classify it into its family.

Static analysis. Since FalDroid relies on static function call graph, it suffers from limitations that are typical for static analysis. FalDroid fails to overcome advanced obfuscation techniques such as encryption and reflection [27]. It is very hard for the de-compilation tools to get the source code once the app is encrypted, which makes it impossible to construct our graph model. Furthermore, the reflection techniques can hide away some edges in the call graph model by invoking functions with their corresponding names as arguments. Dynamic analysis paired with test generation may be a better option to catch the missed behavior [28], [29].

[7]http://proguard.sourceforge.net/
[8]http://www.eoeandroid.com/forum.php?mod=viewthread&tid=304353
[9]http://www.bangcle.com/appProtect/

V. Related Work

In this section, we discuss the previous work related to malware classification and graph-based program analysis.

A. *Familial Malware Classification*

Many prior efforts have been made to automatically classify malware via machine learning on PC platform and mobile platform.

On PC platform, Kolter and Maloof [30] proposed an approach which uses n-grams of byte codes as features to generate a classifier for malware classifying. Kinable and Kostakis [25] studied malware classification based on call graph clustering which represents the malware samples as function call graphs. Similarly, Hu et al. [31] implemented a malware database management system called SMIT which also converts each malware into its call graph representation, and performs nearest neighbor search based on this graph representation.

Compared to the malware in forms of Internet worms and computer viruses on PC platform, the mobile malware contains two main differences: 1) mobile malware invokes the sensitive APIs to perform malicious behaviors; 2) mobile malware is more easily produced by injecting malicious payloads into legitimate apps with mature de-compilation tools compared to the malicious executables on PC platform. Leveraging the two main differences, malware classification approaches on mobile platform have better performance than those on PC platform.

On mobile platform, Android is the major target of mobile malware. Suarez et al. [23] proposed Dendroid, which automatically classifies malware and analyzes families based on the code structures. However, the code structure is easily evaded by bytecode-level transformation. Yang et al. [8] proposed DroidMiner which formalizes a two level behavioral graph model and extracts sensitive path to denote the malicious behavioral patterns as features for malware classification. The sensitive path might appear in both legitimate part and malicious components, which would affect the classification performance. Most related to our work is the approach proposed by Zhang et al. [22], DroidSIFT, which is a semantic-based approach that classifies Android malware via dependency graphs. However, it relies on a set of benign subgraphs to remove common ones in malwares, and it is hard to ensure the completeness of the benign subgraph set.

B. *Graph-based Program Analysis*

More and more recent works on Android detections are based on graph analysis such as PDG (Program Dependence Graph) [32], [33], CDG (Control Dependence Graph) [34], [35], FCG (Function Call Graph) [22], [36], and UI (user interface) Graph [37]–[39]. They are structural representations known to be less susceptible to instruction-level obfuscations commonly employed by malware authors to evade anti-virus scanners.

Crussell et al. [32] proposed DNADroid to detect cloned apps by comparing the PDGs between functions in candidate apps. Chen et al. [35] used the geometry characteristics (centroid) of CDGs to measure the similarity between functions of apps. Gascon et al. [36] proposed an approach on malware detection based on efficient embedding of function call graphs with an explicit feature map. Chen et al. [39] proposed MassVet which models the app's UIs as a directed graph where each node is a view and each edge describes the navigation (triggered by the input events) relations among them. With the similar view structures in different apps, MassVet can effectively discover the repackaged apps.

UI graphs are not proper for familial malware classification since they are totally different for various malwares in same family. PDG and CDG are more fine-grained model than FCG, and they need expensive cost to analyze the app behaviors. With FCG, we can get an enough good result with low run-time overhead.

VI. Conclusions and Future Work

In this paper, we construct fregraph to depict the commonalities of malwares in the same family and propose FalDroid, an automatic system that can handle large scale of Android malwares for familial classification with high accuracy. At first, we propose two observations to reflect the characteristics of the malwares in the same family. Exploiting the two observations, the common malicious behaviors in the same family are extracted with an automatic clustering-based approach, in which the subgraphs are matching with a weighted sensitive APIs-based approach. Finally, each generated fregraph is embedded into a feature space to classify the unknown malware samples into its correct family. FalDroid is evaluated on 6,565 malware samples in 30 families. Experiments show that it can correctly classify 94.5% malwares into their families around 4.4s per app on average.

Our work presented in this paper can be improved and extended by building a more detailed graph model combined with data-flow information. Subgraphs with both control-flow and data-flow information can better depict far more precise semantic meanings of malware families. Furthermore, combining with the dynamic analysis can help us better fight against the advanced obfuscation techniques.

Acknowledgment

The research was supported in part by National Science Foundation of China under grants (91218301, 61428206, 91418205, 61472318, 61532004, 61532015), Ministry of Education Innovation Research Team (IRT13035), the Fundamental Research Funds for the Central Universities. Kai Chen was supported in part by NSFC U1536106, 61303248, 61100226, Youth Innovation Promotion Association CAS, and strategic priority research program of CAS (XDA06010701).

References

[1] Y. Zhang, M. Yang, B. Xu, Z. Yang, G. Gu, P. Ning, X. S. Wang, and B. Zang, "Vetting undesirable behaviors in android apps with permission use analysis," in *Proceedings of the 20th ACM conference on Computer and Communications Security (CCS)*, Berlin, Germany, 2013, pp. 611–622.

[2] K. Tam, S. J. Khan, A. Fattori, and L. Cavallaro, "Copperdroid: Automatic reconstruction of android malware behaviors," in *Proceedings of the 22nd ISOC Network and Distributed System Security Symposium (NDSS)*, San Diego, USA, 2015.

[3] Y. Zhou and X. Jiang, "Dissecting android malware: Characterization and evolution," in *Proceedings of the 33rd IEEE Symposium on Security and Privacy (SP)*, Oakland, USA, 2012, pp. 95–109.

[4] Y. Feng, S. Anand, I. Dillig, and A. Aiken, "Apposcopy: Semantics-based detection of android malware through static analysis," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, HongKong, China, 2014, pp. 576–587.

[5] W. Zhou, Y. Zhou, M. Grace, X. Jiang, and S. Zou, "Fast, scalable detection of "piggybacked" mobile applications," in *Proceedings of the Third ACM Conference on Data and Application Security and Privacy (CODASPY)*, San Antonio, USA, 2013, pp. 185–196.

[6] L. Deshotels, V. Notani, and A. Lakhotia, "Droidlegacy: Automated familial classification of android malware," in *Proceedings of ACM SIGPLAN on Program Protection and Reverse Engineering Workshop*, San Diego, USA, 2014, pp. 1–12.

[7] Y.-D. Lin, Y.-C. Lai, C.-H. Chen, and H.-C. Tsai, "Identifying android malicious repackaged applications by thread-grained system call sequences," *Computers and Security*, vol. 39, pp. 340–350, 2013.

[8] C. Yang, Z. Xu, G. Gu, V. Yegneswaran, and P. Porras, "Droidminer: Automated mining and characterization of fine-grained malicious behaviors in android applications," in *Computer Security-ESORICS 2014*. Springer, 2014, pp. 163–182.

[9] K. Z. Chen, N. M. Johnson, V. D'Silva, S. Dai, K. MacNamara, T. R. Magrino, E. X. Wu, M. Rinard, and D. X. Song, "Contextual policy enforcement in android applications with permission event graphs," in *Proceedings of the 20th ISOC Network and Distributed System Security Symposium (NDSS)*, San Diego, USA, 2013.

[10] M. Fredrikson, S. Jha, M. Christodorescu, R. Sailer, and X. Yan, "Synthesizing near-optimal malware specifications from suspicious behaviors," in *Proceedings of the 31st IEEE Symposium on Security and Privacy (SP)*, Oakland, USA, 2010, pp. 45–60.

[11] M. Rosvall and C. T. Bergstrom, "Maps of random walks on complex networks reveal community structure," *Proceedings of the National Academy of Sciences (PNAS)*, vol. 105, no. 4, pp. 1118–1123, 2008.

[12] S. Rasthofer, S. Arzt, and E. Bodden, "A machine-learning approach for classifying and categorizing android sources and sinks," in *Proceedings of the 21st ISOC Network and Distributed System Security Symposium (NDSS)*, San Diego, USA, 2014.

[13] H. C. Wu, R. W. P. Luk, K. F. Wong, and K. L. Kwok, "Interpreting tf-idf term weights as making relevance decisions," *ACM Transactions on Information Systems (TOIS)*, vol. 26, no. 3, p. 13, 2008.

[14] J. Ramos, "Using tf-idf to determine word relevance in document queries," in *Proceedings of the First Instructional Conference on Machine Learning*, 2003.

[15] A. Aizawa, "An information-theoretic perspective of tf-idf measures," *Information Processing and Management*, vol. 39, no. 1, pp. 45–65, 2003.

[16] M. E. Newman and M. Girvan, "Finding and evaluating community structure in networks," *Physical review E*, vol. 69, no. 2, p. 026113, 2004.

[17] A. Clauset, M. E. Newman, and C. Moore, "Finding community structure in very large networks," *Physical review E*, vol. 70, no. 6, p. 066111, 2004.

[18] G. Concas, C. Monni, M. Orru, and R. Tonelli, "A study of the community structure of a complex software network," in *Proceedings of the 4th International Workshop on Emerging Trends in Software Metrics (WETSoM)*, San Francisco, USA, 2013, pp. 14–20.

[19] Y. Qu, X. Guan, Q. Zheng, T. Liu, L. Wang, Y. Hou, and Z. Yang, "Exploring community structure of software call graph and its applications in class cohesion measurement," *Journal of Systems and Software*, vol. 108, pp. 193–210, 2015.

[20] U. N. 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, p. 036106, 2007.

[21] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre, "Fast unfolding of communities in large networks," *Journal of Statistical Mechanics: Theory and Experiment*, vol. 2008, no. 10, p. P10008, 2008.

[22] M. Zhang, Y. Duan, H. Yin, and Z. Zhao, "Semantics-aware android malware classification using weighted contextual api dependency graphs," in *Proceedings of the 21st ACM Conference on Computer and Communications Security (CCS)*, Scottsdale, USA, 2014, pp. 1105–1116.

[23] 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, pp. 1104–1117, 2014.

[24] A. Sanfeliu and K.-S. Fu, "A distance measure between attributed relational graphs for pattern recognition," *IEEE Transactions on Systems, Man and Cybernetics*, vol. SMC-13, no. 3, pp. 353–362, 1983.

[25] J. Kinable and O. Kostakis, "Malware classification based on call graph clustering," *Journal in Computer Virology*, vol. 7, no. 4, pp. 233–245, 2011.

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

[27] V. Rastogi, Y. Chen, and X. Jiang, "Catch me if you can: Evaluating android anti-malware against transformation attacks," *IEEE Transactions on Information Forensics and Security (TIFS)*, vol. 9, no. 1, pp. 99–108, 2014.

[28] Z. Tian, T. Liu, Q. Zheng, M. Fan, E. Zhuang, and Z. Yang, "Exploiting thread-related system calls for plagiarism detection of multithreaded programs," *Journal of Systems and Software*, vol. 119, pp. 136–148, 2016.

[29] 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 Transactions on Software Engineering*, vol. 41, no. 12, pp. 1217–1235, 2015.

[30] J. Z. Kolter and M. A. Maloof, "Learning to detect and classify malicious executables in the wild," *The Journal of Machine Learning Research*, vol. 7, pp. 2721–2744, 2006.

[31] X. Hu, T.-c. Chiueh, and K. G. Shin, "Large-scale malware indexing using function-call graphs," in *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS)*, Chicago, USA, 2009, pp. 611–620.

[32] J. Crussell, C. Gibler, and H. Chen, "Attack of the clones: Detecting cloned applications on android markets," in *Computer Security–ESORICS 2012*. Springer, 2012, pp. 37–54.

[33] ——, "Andarwin: Scalable detection of semantically similar android applications," in *Computer Security–ESORICS 2013*. Springer, 2013, pp. 182–199.

[34] X. Sun, Y. Zhongyang, Z. Xin, B. Mao, and L. Xie, "Detecting code reuse in android applications using component-based control flow graph," in *ICT Systems Security and Privacy Protection*. Springer, 2014, pp. 142–155.

[35] K. Chen, P. Liu, and Y. Zhang, "Achieving accuracy and scalability simultaneously in detecting application clones on android markets," in *Proceedings of the 36th International Conference on Software Engineering (ICSE)*, Hyderabad, India, 2014, pp. 175–186.

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

[37] F. Zhang, H. Huang, S. Zhu, D. Wu, and P. Liu, "Viewdroid: Towards obfuscation-resilient mobile application repackaging detection," in *Proceedings of the 2014 ACM Conference on Security and Privacy in Wireless and Mobile Networks*, Oxford, UK, 2014, pp. 25–36.

[38] Y. Shao, X. Luo, C. Qian, P. Zhu, and L. Zhang, "Towards a scalable resource-driven approach for detecting repackaged android applications," in *Proceedings of the 30th Annual Computer Security Applications Conference (ACSAC)*, New Orleans, USA, 2014, pp. 56–65.

[39] 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 *Proceedings of the 24th USENIX Security Symposium*, Washington, USA, 2015, pp. 659–674.

Appendix

Table A1: Descriptions Of Malware Families

Malware Family	#Num	Malware Family	#Num
adwo	292	hongtoutou	46
airpush	75	iconosys	153
basebridge	303	imlog	41
boqx	49	kmin	247
boxer	85	kuguo	358
clicker	37	mobiletx	81
dowgin	556	pjapps	82
droiddreamlight	101	plankton	626
droidkungfu	725	smskey	111
fakedoc	146	smsreg	123
fakeinst	668	utchi	285
fakeplay	43	waps	590
geinimi	105	youmi	113
gingermaster	366	yzhc	49
golddream	79	zitmo	30

Table A2: Descriptions Of The Used Metrics

Term	Abbr	Definition
True Positive	TP	#malwares in family f are correctly classified into family f.
True Negative	TN	#malwares not in family f are correctly not classified into family f.
False Negative	FN	#malwares in family f are incorrectly not classified into family f.
False Positive	FP	#malwares not in family f are incorrectly classified into family f.
True Positive Rate	TPR	TP/(TP+FN)
False Positive Rate	FPR	FP/(FP+TN)
Precision	p	TP/(TP+FP)
Recall	r	TP/(TP+FN)
F-measure	F_1	2rp/(r+p)
ROC Area	AUC	Area under ROC curve