Android Malware Familial Classification and Representative Sample Selection via Frequent Subgraph Analysis

Ming Fan, Jun Liu, Xiapu Luo, Kai Chen, Zhenzhou Tian, Qinghua Zheng, and Ting Liu

***Abstract*— The rapid increase in the number of Android malware poses great challenges to anti-malware systems, because the sheer number of malware samples overwhelms malware analysis systems. The classification of malware samples into families, such that the common features shared by malware samples in the same family can be exploited in malware detection and inspection, is a promising approach for accelerating malware analysis. Furthermore, the selection of representative malware samples in each family can drastically decrease the number of malware to be analyzed. However, the existing classification solutions are limited because of the following reasons. First, the legitimate part of the malware may misguide the classification algorithms because the majority of Android malware are constructed by inserting malicious components into popular apps. Second, the polymorphic variants of Android malware can evade detection by employing transformation attacks. In this paper, we propose a novel approach that constructs frequent subgraphs (*fregraphs*) to represent the common behaviors of malware samples that belong to the same family. Moreover, we propose and develop FalDroid, a novel system that automatically classifies Android malware and selects representative malware samples in accordance with *fregraphs*. We apply it to 8407 malware samples from 36 families. Experimental results show that FalDroid can correctly classify 94.2% of malware samples into their families using approximately 4.6 sec per app. FalDroid can also dramatically reduce the cost of malware investigation by selecting only 8.5% to 22% representative samples that exhibit the most common malicious behavior among all samples.**

***Index Terms*— Android malware, frequent subgraph, familial classification.**

Manuscript received March 14, 2017; revised October 7, 2017 and January 8, 2018; accepted February 8, 2018. Date of publication February 16, 2018; date of current version April 4, 2018. This work was supported in part by the National Key Research and Development Program of China under Grant 2016YFB1000903, in part by the National Science Foundation of China under Grant 61632015, Grant 61772408, Grant U1766215, Grant 61672419, Grant 61702414, Grant 61721002, Grant 61428206, Grant 61472318, Grant 61532004, Grant 61532015, and Grant 61602369, in part by the Project of China Knowledge Centre for Engineering Science and Technology, in part by the Fok Ying-Tong Education Foundation under Grant 151067, in part by the Ministry of Education Innovation Research Team under Grant IRT17R86, and in part by the Fundamental Research Funds for the Central Universities. The work of X. Luo was supported in part by Hong Kong GRF under Grant PolyU 5389/13E and Grant 152279/16E, in part by the Hong Kong RGC Project under Grant CityU C1008-16G, in part by the HKPolyU Research Grants under Grant G-YBJX, and in part by the Shenzhen City Science and Technology R&D Fund under Grant JCYJ20150630115257892. The work of K. Chen was supported in part by the National Key R&D Program of China under Grant 2016QY04W0805, Grant NSFC U1536106, and Grant 61728209, in part by the National Top-notch Youth Talents Program of China, Youth Innovation Promotion Association CAS, in part by the Beijing Nova Program, and in part by a Research Grant from Ant Financial. This paper was presented at IEEE 26th ISSRE. The associate editor coordinating the review of this manuscript and approving it for publication was Dr. Tomas Pevny. *(Corresponding author: Ting Liu.)*

M. Fan is with the MOEKLINNS Laboratory, Department of Computer Science and Technology, Xi'an Jiaotong University, Xi'an 710049, China, and also with the Department of Computing, The Hong Kong Polytechnic University (e-mail: fanming.911025@stu.xjtu.edu.cn).

J. Liu is with the National Engineering Laboratory of Big Data Analytics, Department of Computer Science and Technology, Xi'an Jiaotong University, Xi'an 710049, China (e-mail: liukeen@mail.xjtu.edu.cn).

X. Luo is with the Department of Computing, The Hong Kong Polytechnic University (e-mail: luoxiapu@gmail.com).

K. Chen is with the State Key Laboratory of Information Security, Institute of Information Engineering, Chinese Academy of Sciences, and also with the School of Cyber Security, University of Chinese Academy of Sciences, Beijing, 100195, China (e-mail: chenkai@iie.ac.cn).

Z. Tian is with the School of Computer Science and Technology, Xi'an University of Posts and Telecommunications, Xi'an 710049, China (e-mail: tianzhenzhou@xupt.edu.cn).

Q. Zheng and T. Liu are with the MOEKLINNS Laboratory, Department of Computer Science and Technology, Xi'an Jiaotong University, Xi'an 710049, China (e-mail: qhzheng@mail.xjtu.edu.cn; tingliu@mail.xjtu.edu.cn).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TIFS.2018.2806891

I. Introduction

In THE third quarter of 2016, Android, the most popular mobile operating system, accounted for 86.8% of the market share of smartphones [1]. Meanwhile, it has become the major target of 97% of mobile malware [2]. A recent security report shows that on average, 38,000 new mobile malware samples were captured per day during the third quarter of 2016 [3]. The analysis of each malware sample requires ample time [4]–[6]. Hence, the sheer number of malware samples overwhelms malware analysis systems.

The majority of new malware samples are polymorphic variants of known malware [7], [8]. Thus, to accelerate malware analysis, we can classify malware samples into various families and then select representative samples from each family. However, the familial classification of Android malware is challenging because of two reasons.

First, the accurate separation of malicious components and the legitimate part from the majority of Android malware, which are repackaged popular apps, is nontrivial [9]–[12]. Zhou and Jiang [7] found that 86% of Android malware samples are repackaged apps produced by injecting malicious components into legitimate apps. The injected malicious components are hidden within the functionalities of popular apps and usually constitute only a small portion of the repackaged apps. Differentiating between the legitimate part and malicious components of malware is difficult for existing features, such as system calls [13] and sensitive path [14].

Second, polymorphic variants of Android malware that belong to the same family perform the same malicious

```
.class public final Lcom/geinimi/c/f;
.method public constructor <init> (Lcom/geinimi/Adservice;)V
    ...
    invoke-virtual {p0},Landroid/telephony/TelephonyManager;->getDeviceId()Ljava/lang/String;
    invoke-virtual {p0},Landroid/telephony/TelephonyManager;->getLine1Number()Ljava/lang/String;
    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

.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;->getDeviceId()Ljava/lang/String;
    invoke-virtual {p0},Landroid/telephony/TelephonyManager;->getLine1Number()Ljava/lang/String;
    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;
```

Listing 1. Different implementations of the same functionality in two malware samples within *geinimi* family.

activities with different implementations. Therefore, such malware can easily evade existing classification solutions [15], [16] that seek an exact match of a given specification. For example, Listing 1 illustrates different implementations of the same functionality (i.e., obtain device id, phone number, and voice mail number) in two malware samples. The two malware samples belong to the same family, *geinimi*. These bot-like malware samples steal personal information and send it to a remote server. Three major differences (highlighted in red) are observed in the two implementations. First, the structures of class names are different. Second, the arguments of the two functions are different. One takes a service (*Lcom/geinimi/Adservice*), one of the four basic components of Android apps, as an argument. By contrast, the other uses an object of the class *rally/e* as an argument. Third, the former function contains two more statements (including one invocation) than the latter.

To address the above challenges, we propose a novel approach that exploits the following two observations:

Observation 1: Android malware usually invokes sensitive application program interface (API) calls that operate on sensitive data to perform malicious activities. For example, the malware samples presented in Listing 1 invoke *getLine1Number()* to obtain the phone number of users.

Observation 2: Malware and its variants within the same family invoke sensitive API calls by following similar patterns even if their codes may be obfuscated. As illustrated in Listing 1, three commonly invoked sensitive API calls (i.e., *getDeviceId()*, *getLine1Number()*, and *getVoiceMailNumber()*), which are highlighted in blue, exist in the two methods of different malware samples. The three sensitive API calls are sequentially invoked in the two methods, thus illustrating a similar pattern of sensitive API calls in different samples within the same family.

By exploiting the above two observations, we first distill program semantics into function call graph (FCG) representation and assign different weights to different sensitive API calls with a term frequency-inverse document frequency (TF-IDF)-like approach (see Section II-A for details). TF-IDF is a numerical statistic that evaluates the importance of a word to a document in a collection or corpus.

Then, we propose two key techniques to solve the challenges (see Section II-B for details), as follows: 1) We propose a clustering-based approach to extract common malicious behavior in each family and to address the inaccurate separation of malicious components and the legitimate part of repackaged apps. Thus, we can reduce the side-effects of the legitimate part in the malware. 2) For the different implementations of the same functionality, we propose a weighted-sensitive-API-call-based graph matching approach to calculate the similarity between graphs generated by community detection algorithms. Community detection algorithms are used to determine whether or not a graph has community structure if the nodes of the graph can be easily grouped into sets of nodes, such that each set of nodes is internally densely connected. Our approach can detect homogeneous malicious behavior while tolerating minor differences in implementation, such as function renaming and junk-code insertion. Sensitive API calls constitute only a small portion of the entire Android API calls, and they cannot be easily obfuscated by existing typical obfuscation techniques, whereas the names of user-defined functions are usually obfuscated as *a*, *b*, or *c*.

To represent common malicious behaviors shared by malware samples within the same family, we construct frequent subgraphs (*fregraphs*), which are novel graph-based features extracted from generated FCGs, on the basis of two key techniques. Moreover, we propose and develop FalDroid, an automatic system for classifying Android malware and selecting representative samples of each family in accordance with *fregraphs*, in 8,100 lines of Java code and 900 lines of Python code. We apply FalDroid to 8,407 malware in 36 different families and find that it exhibits impressive familial classification performance. Moreover, it can effectively reduce workload and accelerate malware analysis.

In summary, our major contributions include the following:

(i) We propose *fregraph*, a novel graph-based feature, to represent the common behavior of malware within the same family. We then employ *fregraph* to conduct familial classification and representative malware selection.

(ii) We propose a novel weighted-sensitive-API-call-based graph matching approach that can detect the homogeneous malicious behavior of malware within the same family while tolerating minor differences in implementation.

(iii) We design and implement FalDroid, a novel system that can handle the familial classification of large-scale Android malware with high accuracy and effectively decrease the number of malware to be analyzed.

(iv) We conduct extensive experiments to evaluate FalDroid. Our results show that FalDroid can achieve 94.2% accuracy and only requires approximately 4.6 sec to process an app. Moreover, it can also dramatically decrease the cost of malware investigation by selecting only 8.5% to 22% of representative samples that present the most malicious behavior among all samples.

The remainder of this paper is organized as follows. The methodology of FalDroid is detailed in Section II, and its two usages are presented in Section III. The experimental results are reported in Section IV. After providing a discussion of the limitations of FalDroid in Section V, we introduce related work in Section VI. We conclude the paper with a discussion of future work in Section VII.

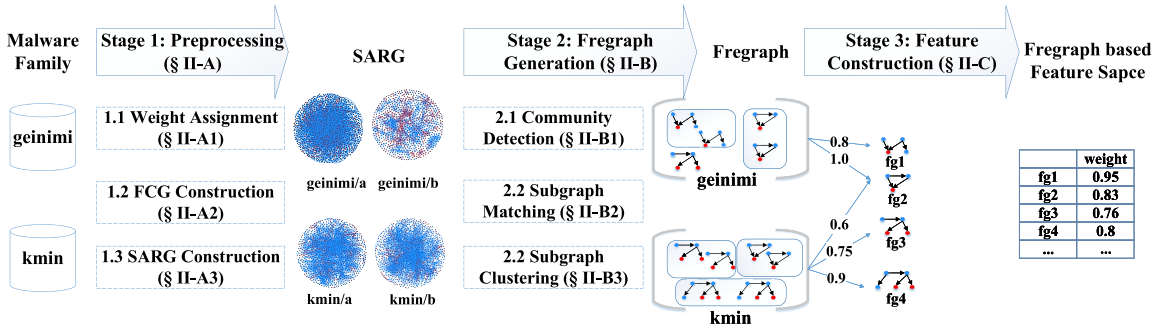

Fig. 1. The overall architecture of FalDroid: (1) *Preprocessing* stage contains three processes, weight assignment (Section II-A1), FCG construction (Section II-A2), and SARG construction (Section II-A3); (2) *Fregraph Generation* stage contains three processes, community detection (Section II-B1), subgraph matching (Section II-B2), and subgraph clustering (Section II-B3); (3) *Feature Construction* stage (Section II-C).

II. Methodology of FalDroid

Fig. 1 shows the overall architecture of FalDroid, which consists of three main stages.

The *Preprocessing* stage constructs the basic behavior model for each app, and it contains three processes. First, different weights are assigned to each sensitive API call by using a TF-IDF-like approach to differentiate their corresponding importance given that the importance of the sensitive API calls differs across different families. Second, to depict the program semantics of an app, an FCG is constructed to represent the app on the basis of its disassembled code. Third, given that the direct analysis of the FCG is time consuming because it usually contains thousands of nodes, the FCG is simplified into a **sensitive API call related graph (SARG)** (Definition 1) by retaining only sensitive API call nodes and their parent nodes. Therefore, the malicious behaviors of the apps are maintained, whereas the complexities of the graph models are reduced.

In the *Fregraph Generation* stage, fregraphs are produced to denote the common malicious behaviors shared by malware within the same family. To easily locate the common functionalities of different malware and reduce the complexity of graph similarity calculation, the SARG is initially divided into a set of subgraphs using community detection algorithms [17]–[20]. Specifically, subgraphs with sensitive API call nodes are designated as **sensitive subgraphs** (Definition 2). Using subgraph matching and clustering techniques, the sensitive subgraphs used by most samples in one family are defined as the **fregraphs** (Definition 3) of a specific family.

In the *Feature Construction* stage, a feature vector is constructed for each app. On this basis, known machine learning algorithms can be applied to perform the familial classification task. To this end, the fregraphs of all known families are embedded in a feature space, and each fregraph is assigned with a weighted score to indicate its significance for malware familial analysis.

A. Preprocessing

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. We can obtain the Dalvik code from the APK by using disassemble tools (e.g., *apktool* [21]).

Android malware usually invokes sensitive API calls that operate on sensitive data to perform malicious activities. We employ the result reported by [22] to obtain a set of sensitive API calls. A total of 26,322 sensitive API calls are available.

1) Weight Assignment of Sensitive API Calls: To differentiate the importance of sensitive API calls, we assign different weights to each sensitive API call in different families. In particular, we define three metrics for each sensitive API call s in family f to characterize its usages in different families.

- $num(s, f)$: number of samples that invoke the sensitive API call s in family f.
- $per(s, f)$: percentage of samples that invoke the sensitive API call s in family f, $per(s, f) = \frac{num(s,f)}{falNum(f)}$, where $falNum(f)$ denotes the number of samples in f.
- $w(s, f)$: weight of sensitive API call s in family f.

In addition, we use $allNum$ to denote the number of all collected samples and $totalNum(s)$ to denote the number of samples that invoke s in all families; $totalNum(s) = \sum_{f_j \in F} num(s, f_j)$, where $F = \{f_j | 1 \le j \le m\}$ denotes the set of all families, and m denotes the number of families.

We collect 8,407 malware samples in 36 families from Virusshare [23] for evaluation. TABLE I lists the $totalNum$ of six sensitive API calls and their num, per and w in three different families. We observe that the usages of different sensitive API calls in the same family are different. For example, *sendTextMessage()* is used by all 105 samples in the *geinimi* family, whereas *divideMessage()* is used by only six samples. Moreover, some sensitive API calls (e.g., *getDeviceId()*) are used by most malware samples.

The two observations indicate that the weight of a sensitive API call in one family should be positively related with its per and be negatively related with its $totalNum$. By borrowing the idea of TF-IDF [24], we propose a TF-IDF-like approach, which allows the TF to measure the frequency of sensitive API call s that appears in family f, and IDF to measure the inverse frequency of s that appears across all malware samples. Then, the weight of sensitive API call s in family f is defined as follows:

$$w(s, f) = per(s, f) * \log \frac{allNum}{totalNum(s)}. \quad (1)$$

TABLE I shows that the weight of *sendTextMessage()* is 0.567 in the *geinimi* family, whereas that of

TABLE I
SIX SENSITIVE API CALLS' $totalNum$ AND THEIR CORRESPONDING num, per AND w IN THREE FAMILIES ($allNum = 8,407$)

sensitive API call	$totalNum$	*geinimi*($falNum = 105$)			*plankton*($falNum = 896$)			*droidkungfu*($falNum = 736$)		
		num	per	w	num	per	w	num	per	w
getDeviceSoftwareVersion()	183	105	1.000	1.662	0	0.000	0.000	0	0.000	0.000
getDeviceId()	6,950	105	1.000	0.083	896	1.000	0.083	736	1.000	0.083
getLine1Number()	4,827	105	1.000	0.241	471	0.526	0.127	677	0.920	0.222
getConnectionInfo()	4,055	0	0.000	0.000	896	1.000	0.317	359	0.488	0.155
sendTextMessage()	2,277	105	1.000	0.567	37	0.041	0.023	25	0.034	0.193
divideMessage()	330	6	0.057	0.080	2	0.002	0.003	7	0.009	0.013

divideMessage() is only 0.080 because the per of *sendTextMessage()* is considerably higher than that of *divideMessage()*. Moreover, *getDeviceId()* is used by all samples in the three families. Thus, it is less important than the others for malware classification, and its weight is only 0.083, which is considerably less than the weights of the other sensitive API calls. Intuitively, the results show that the weight assignment of our approach can effectively measure the importance of a sensitive API call to one family.

2) Construction of FCG: After applying *apktool* [21] to APK files, we can obtain the corresponding Dalvik code. Then, we extract the callers and the 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. Thus, we can abstract the program semantics of an app into an FCG representation, which contains the necessary structural information to profile the behaviors of an app. The FCG is represented as a directed 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.

3) Construction of SARG: Thousands of nodes are found in the FCG. Analyzing the entire FCG is neither effective (e.g., the malicious part is hidden in the legitimate part) nor efficient (e.g., excessive number of nodes and edges to analyze). Thus, we exclude nodes with no paths to sensitive API call nodes to reduce the complexity of graph analysis, and FCG G is then simplified into the SARG G'. We designate the nodes that represent sensitive API calls as sensitive API call nodes.

Definition 1 (SARG): It is an induced subgraph of FCG and is maximal with respect to the number of nodes, where each node has at least one directed path to sensitive API call nodes, or the node itself is a sensitive API call node.

SARG $G' = (V', E')$ can be obtained using Eqs. (2) and (3), where $V_s \subseteq V$ is the set of sensitive API calls invoked by the app, and the function $dis(v_j, v_i)$ returns the length of the shortest path length from node v_j to node v_i.

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

$$V' = V_s \cup V_g, \quad E' = (V' \times V') \cap E \quad (3)$$

In general, the size of SARG is reduced by approximately 72% compared with that of the original FCG (see Section IV-E for details). Fig. 2 presents the original FCG (2,000 nodes) of a malware in the *geinimi* family and its SARG (450 nodes), where red nodes denote sensitive API call nodes and blue nodes denote general nodes. The red edges indicate that their callee functions are sensitive API call nodes.

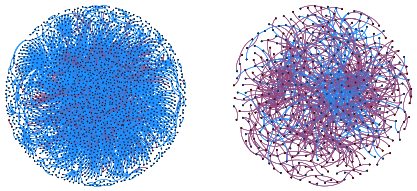

Fig. 2. The original FCG (left) of a *geinimi* sample and its generated SARG (right).

B. Fregraph Generation

This section describes the two key techniques presented in this work, namely, a clustering-based approach to extract the common malicious behaviors of each family (Sections II-B1 and II-B3) and a weighted-sensitive-API-call-based graph matching approach to calculate the similarity between subgraphs generated with community detection algorithms (Section II-B2).

1) Community Detection: After the *Preprocessing* stage, we obtain the following observations from the generated SARGs of the same family. *Apps in the same family have similar subgraphs, which constitute only a small portion of SARGs even if a large portion of their SARGs is different*. The small portion of the generated SARG represents the common malicious functionalities of malware samples within the same family, whereas the other large portion of SARGs represents different legitimate functionalities.

Fig. 3 presents the SARGs of two different samples in the *geinimi* family. The two SARGs contain 267 and 715 nodes. The subgraphs marked with red circles are nearly identical, indicating similar behaviors, whereas the other parts are entirely different. The direct identification of similar subgraphs from SARGs is inefficient because the graph isomorphism problem is NP complete [25]. Hence, we divide the SARGs into a set of smaller subgraphs to easily locate the common functionalities of different malware samples and reduce the complexity of graph similarity calculation.

As introduced in [18] and [26], a major network feature is the community structure, which refers to the gathering of vertices into groups such that a higher density of edges exists within groups than between groups. Previous studies [27], [28] have demonstrated that FCG is a typical network with community structures. Software functions in one community structure

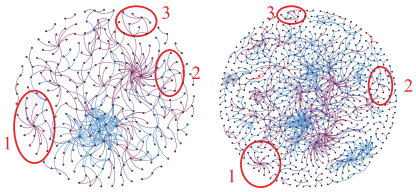

Fig. 3. SARGs of two malware samples in the *geinimi* family. Three similar subgraphs marked with red circles indicate the similar behaviors.

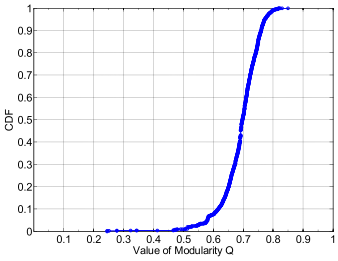

Fig. 4. CDF of modularity Q with *infomap* algorithm.

have strong connections and are frequently located in the same class or package to realize collective software functionalities.

To determine whether or not our generated SARGs are networks with community structures, we adopt four widely used community detection algorithms, including *infomap* [17], *fast greedy* [18], *fast partitioning* [19], and *multilevel* [20], to divide SARGs into a set of subgraphs. We implement the algorithms using *Networkx* [29], which is a package for computation of complex networks. We select *infomap* [17] as the main community detection algorithm in the experiments given that it generates more subgraphs with fewer nodes than the other three algorithms, thereby effectively reducing the complexity of graph matching.

Newman and Girvan [26] proposed the concept of *modularity Q* to quantify the quality of a detected community structure. No community structure is found when the value of Q approaches 0. On the contrary, an ideal community structure is obtained when Q is close to 1. We evaluate the generated SARGs in our dataset using the *infomap* algorithm. Fig. 4 shows the cumulative distribution function (CDF) of modularity Q. More than 90% of Q values range from 0.6 to 0.8. The range demonstrates that the generated SARGs have significant community structures.

Moreover, given that most subgraphs divided by community detection algorithms have no relation with sensitive data, they might provide little help for malware classification. Therefore, we define the sensitive subgraph.

Definition 2 (Sensitive Subgraph): It is a subgraph divided from SARG using the community detection algorithm and contains at least one sensitive API call node. No common node exists in any two sensitive subgraphs from the same SARG.

Sensitive subgraph sg in family f has a weighted value $w(sg, f)$, as defined in Eq. (4), to denote its importance to f. $V_s(sg)$ is the set of sensitive API call nodes in sg.

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

2) Subgraph Matching: To quantify the similarity of two sensitive subgraphs, we propose a novel weighted-sensitive-API-call-based approach that can detect the homogeneous app behavior of malware within the same family and can tolerate minor differences in implementation.

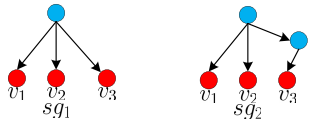

Fig. 5. Two subgraph examples sg_1 and sg_2 in family f.

Fig. 5 presents two subgraph examples sg_1 and sg_2 in family f. Both subgraphs contain three sensitive API call nodes, v_1, v_2 and v_3. We assume that the three nodes are assigned with weights 0.2, 0.5, and 0.8 on the basis of our TF-IDF-like approach. To calculate the similarity of sg_1 and sg_2 in family f, we focus on the similarities between their sensitive API call nodes because such nodes cannot be easily changed by typical obfuscation techniques. The similarity between the same sensitive API call nodes in two subgraphs is calculated on the basis of their structural equivalence. The structural equivalence hypothesis [30] states that nodes with similar structural roles in subgraphs should be collectively and closely embedded in the same feature space. Specifically, the similarity $sim_f(sg_1, sg_2)$ is calculated in three steps.

Step 1 (Construct Distance Matrices for Two Subgraphs): We initially construct a distance matrix for each subgraph, which is used to measure the relations among different sensitive API call nodes in the specific subgraph. The matrix of sg_k $(k = 1, 2)$ is obtained through Eq. (5), and its size is $t \times t, t = |V_s(sg_1) \cup V_s(sg_2)|$. In Eq. (5), the graph is regarded as an undirected graph when calculating the shortest path length $dis'(v_i, v_j)$ between two nodes v_i and v_j.

$$Matrix_k[i, j] = \begin{cases} dis'(v_i, v_j) & v_i, v_j \in V_s(sg_k) \\ \infty & otherwise \end{cases} \tag{5}$$

For the two subgraphs presented in Fig. 5, the sizes of the two constructed distance matrices are 3×3 as calculated in step 1. $Matrix_1[1, 3] = 2$ whereas $Matrix_2[1, 3] = 3$ given that an additional normal node exists in the path between v_1 and v_3 in sg_2 compared with that in sg_1.

Step 2 (Calculate the Similarity of Sensitive Nodes): To formalize the structural role of a sensitive API call node in a subgraph, we embed it into a vector with t dimensions through Eq. (6). The value for each dimension is calculated on the basis of the shortest path distance between the current sensitive API call node and other sensitive API call nodes. Then, the similarity of the same sensitive API node in sg_1 and sg_2 is denoted as $ns(v_i)$ and is measured through a standard cosine metric in Eq. (7).

$$\overrightarrow{vec(v_i, sg_k)} = \langle \frac{1}{Matrix_k(i, 1)}, \ldots, \frac{1}{Matrix_k(i, t)} \rangle \tag{6}$$

$$ns(v_i) = cos(\overrightarrow{vec(v_i, sg_1)}, \overrightarrow{vec(v_i, sg_2)}) \tag{7}$$

The vectors of v_1 in the two subgraphs presented in Fig. 5 are $\overrightarrow{vec(v_1, sg_1)} = \langle 0, \frac{1}{2}, \frac{1}{2} \rangle$ and $\overrightarrow{vec(v_1, sg_2)} = \langle 0, \frac{1}{2}, \frac{1}{3} \rangle$ with

step 2. Therefore, $ns(v_1) = 0.98$ on the basis of the standard cosine metric. Similarly, $ns(v_2) = 0.98$ and $ns(v_3) = 1.0$.

Step 3 (Calculate the Similarity of Subgraphs): We calculate $sim_f(sg_1, sg_2)$ with a normalized weighted sum of the cosine distances among nodes in the intersection of two subgraphs given that each sensitive API call node is assigned with a weight to indicate its importance to a specific family f. The computation is as follows:

$$sim_f(sg_1, sg_2) = \frac{\sum_{v_i \in V_s(sg_1) \cap V_s(sg_2)} (w(v_i, f) * ns(v_i))}{\sum_{v_i \in V_s(sg_1) \cup V_s(sg_2)} w(v_i, f)}. \tag{8}$$

Therefore, the similarity of the two subgraphs presented in Fig. 5 is $sim_f(sg_1, sg_2) = \frac{0.98*0.2+0.98*0.5+1.0*0.8}{0.2+0.5+0.8} = 0.99$. The similarity ranges from 0 to 1. The maximum value 1 indicates that the two subgraphs exhibit the exact same behavior, whereas the minimum value 0 indicates that the two subgraphs exhibit entirely different behaviors. The examples demonstrate that our subgraph similarity calculation approach can well tolerate minor differences of implementation.

3) Subgraph Clustering: With the effective and efficient graph matching approach, we generate fregraphs on the basis of subgraph clustering without prior knowledge.

Algorithm 1 Clustering of Sensitive Subgraphs

Input:

SG_f // SG_f denotes the set of sensitive subgraphs in family f.

$\epsilon = 0.8$ // ϵ denotes the similarity threshold value.

Output:

C // C denotes the set of output clusters and each cluster contains a set of similar sensitive subgraphs.

```
p = 1, c_1 = {sg_1}, C = {c_1}
for each sg_{i,i≠1} in SG_f do
  c' = argmax_{c_j ∈ C} sim_f(sg_i, c_j)
  if sim_f(sg_i, c') ≥ ε then
    c' = c' ∪ {sg_i}
  else
    p = p + 1, c_p = {sg_i}, C = C ∪ {c_p}
  end if
end for
return C
```

Algorithm 1 lists the steps of sensitive subgraphs clustering with the input of a set of sensitive subgraphs in family f and similarity threshold ϵ. The output of the algorithm is C, which denotes a set of output clusters. Each cluster contains a set of similar sensitive subgraphs. In the algorithm, sg_i denotes the i^{th} subgraph element in SG_f, and c_j denotes the j^{th} cluster element in C. At first, C is initialized with only one cluster $c_1 = \{sg_1\}$. Then, all the other subgraphs in SG_f are successively calculated to check whether a cluster exists in C, which the current subgraph can be added in. To this end, we first calculate the similarities of the current subgraph sg_i with each cluster in C. The similarity of subgraph sg_i with cluster c_j is denoted as $\overline{sim_f}(sg_i, c_j)$, which is obtained on the basis of the average similarity of sg_i with all the subgraphs in c_j. Then, we select the cluster c' that contains the highest similarity with sg_i. If the similarity is not less than ϵ, then sg_i is added in c', or a new cluster that contains only sg_i is created and added in C.

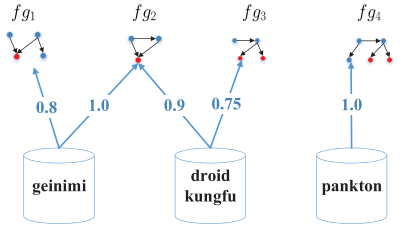

Fig. 6. A mapping between four fregraphs and three malware families.

ϵ is an important argument in Algorithm 1. To appropriately set the parameter ϵ, we first manually construct the ground truth called similar set, which consists of 50 similar subgraphs. Then, we calculate the similarity of any two subgraphs. To ensure that all subgraphs in our ground truth can be placed into the same cluster, we select $\epsilon = 0.8$ as the similarity threshold for clustering subgraphs (see Section IV-F for details).

Definition 3 (Fregraph): Given cluster $c_j \in C$ in family f and minimum support threshold θ, a sensitive subgraph $sg = argmax_{sg_i \in c_j} w(sg_i, f)$ is regarded as a fregraph when its support $sup_f(sg) = \frac{|c_j|}{falNum(f)}$ is not less than θ.

C. Feature Construction

To enable malware familial analysis, all fregraphs in known families are embedded into a feature space, and each fregraph fg is assigned with a weighted score fs to denote its significance to malware familial analysis.

Mapping exists between fregraphs and families given that some fregraphs belong to more than one family. Fig. 6 shows an example of such mapping between four fregraphs and three malware families. The number between a fregraph and a family is defined as the support of the fregraph to its corresponding family. The fregraphs that belong to several families (e.g., fg_2) should have lower significance to malware familial analysis than fregraphs that belong to only one family (e.g., fg_3) because the latter provide more useful information than the former.

We define the weighted score of fregraph fg as follows:

$$fs(fg) = cb'(fg) * \sum_{f_j \in F} w(fg, f_j) * p(f_j|fg), \tag{9}$$

where $p(f_j|fg)$ denotes the probability that the app belongs to family f_j when it contains fregraph fg. It is calculated using Eq. (10) as follows:

$$p(f_j|fg) = \frac{sup_{f_j}(fg)}{\sum_{f_i \in F} sup_{f_i}(fg)}. \tag{10}$$

$cb'(fg)$ indicates the normalized entropy value of fg. $cb'(fg)$ is obtained through Eqs. (11)-(12), where cb_{max} and cb_{min} denote the corresponding maximum and minimum values, respectively. Specifically, $cb'(fg)$ ranges from 0 to 1. A high $cb'(fg)$ indicates that fg belongs to few families.

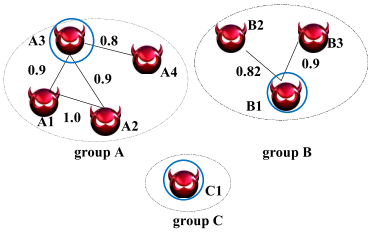

Fig. 7. An example of MSG.

If $cb'(fg) = 1$, then the fregraph belongs to only one family (e.g., fg_1, fg_3 and fg_4 in Fig. 6).

$$cb(fg) = \sum_{f_j \in F} p(f_j|fg) * \log p(f_j|fg) \tag{11}$$

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

III. Usages of FalDroid

To accelerate malware analysis, we leverage FalDroid to classify a new malware sample into its family (Section III-A) and identify representative malware samples from one family, thereby reducing the analytical workload (Section III-B).

A. Familial Classification of Android Malware

FalDroid constructs a fregraph-based feature vector to represent each sample. Within the vector, the default value of each fregraph-based feature is 0, and it will be set to the weighted score when the sample contains this feature. For known samples in the training dataset, their family labels are attached to the feature vector. Then, a classifier is trained using diverse machine learning algorithms. Subsequently, the feature vector of a new malware sample without family label will be placed into the classifier to obtain a family label.

B. Selection of Representative Malware Samples

The in-depth inspection of each sample in several families, such as the *fakeinst* family, that contains excessive samples (1,504 samples in our dataset) is inefficient. We prioritize the inspection of representative malware samples from each family to reduce the analytical workload and accelerate malware analysis. Therefore, we initially construct a malware similarity graph (MSG) to characterize the relationships among malware samples within the same family.

Definition 4 (MSG): It is an undirected graph $MSG_f = \{MV, ME\}$ for one malware family f.

- $MV = \{\alpha_i | 1 \leq i \leq falNum(f)\}$ denotes the set of malware samples in the family f, where each node $\alpha_i \in MV$ indicates a malware sample.
- ME denotes the set of edges, where an edge (α_i, α_j) indicates that the similarity between samples α_i and α_j is higher than the threshold η.

One MSG contains several $groups$, where each $group$ denotes a connected subgraph in MSG. Notably, each node in MSG only belongs to one $group$.

Fig. 7 shows an example of MSG with three groups (i.e., $groups$ A, B, and C) given that $\eta = 0.8$. The number next to an edge denotes the similarity between the two corresponding nodes. Each malware sample is represented as a fregraph-based feature vector. The similarity of two malware samples α_1 and α_2 is calculated on the basis of the cosine value of their vectors $\overrightarrow{u}$ and $\overrightarrow{w}$; $|\overrightarrow{u}| = |\overrightarrow{w}| = l$.

$$sim(\alpha_1, \alpha_2) = \frac{\overrightarrow{u} \cdot \overrightarrow{w}}{\|\overrightarrow{u}\| \|\overrightarrow{w}\|} = \frac{\sum_{i=1}^{l} \overrightarrow{u_i}\overrightarrow{w_i}}{\sqrt{\sum_{i=1}^{l} \overrightarrow{u_i}^2}\sqrt{\sum_{i=1}^{l} \overrightarrow{w_i}^2}} \tag{13}$$

For each group in a family, the node with the largest summation of similarities with connected nodes is selected as the representative node, which is formally defined as:

$$\alpha' = \underset{\alpha \in GV(group)}{argmax} \sum_{\beta \in SN(\alpha)} sim(\alpha, \beta), \tag{14}$$

where $GV(group)$ denotes the set of nodes in the $group$, and $SN(\alpha)$ denotes the set of the neighbor nodes of α. In Fig. 7, the representative malware samples include A3, B1, and C1, which are marked with blue circles. The group that contains only one sample, such as $group$ C, exists. The sample C1 is not similar to the other samples given that all the similarities of C1 with the other nodes are lower than η. However, the inspection of sample C1 could be more interesting. With our approach, C1 is also regarded as one representative sample in the family, such as A3 and B1.

Security analysts should focus on the representative malware samples selected from each family instead of all malware samples. Therefore, FalDroid can reduce the analytical workload and accelerate malware analysis.

IV. Evaluation

We initially introduce the construction of our datasets, use metrics to evaluate FalDroid, and then address the following research questions:

***RQ** 1: Can fregraphs effectively represent the common behaviors of malware samples within the same family?*

***RQ** 2: Can FalDroid classify the new malware sample into its family with high accuracy?*

***RQ** 3: Can FalDroid effectively decrease the number of malware samples to be analyzed?*

***RQ** 4: Can FalDroid work efficiently and be scalable for a large number of apps?*

***RQ** 5: Is FalDroid resilient to polymorphic variants and code obfuscation techniques?*

A. Datasets and Metrics

We evaluate FalDroid using four datasets, including two datasets that are constructed by ourselves (FalDroid-I and FalDroid-II datasets) and two widely used benchmark datasets that are provided from Drebin [31] and Android Malware Genome Project [7]. TABLE II lists the descriptions of the four datasets. More than 90% of malware samples are smaller than 5 MB, and approximately 3% of malware samples are larger than 10 MB. The largest sample size is 64 MB, and the smallest sample size is only 5 KB.

After removing the families that contain only one sample, the dataset from Drebin [31] contains 5,513 samples in 132 malware families, and the dataset provided from

TABLE II
DESCRIPTIONS OF FOUR DIFFERENT DATASETS

Dataset	#Samples	#Families	Average Size (MB)	Time
Genome Project dataset [7]	1,247	33	1.3	2011∼2012
Drebin dataset [31]	5,513	132	1.3	2011∼2014
FalDroid-I dataset	8,407	36	1.9	2013∼2014
FalDroid-II dataset	643	43	2.0	2015∼2016

TABLE III
PART OF THE FAMILY LABEL DICTIONARY

Family	Other Labels
basebridge	bridge
droiddreamlight	ddlight/lightdd/drdlightd/
droidkungfu	kungf/gongf/droidkungf/droidkungfu2
fakeinst	fakeinstall/fakeins
plankton	planktonc/plangton
geinimi	geinim/geinimia/geinimix

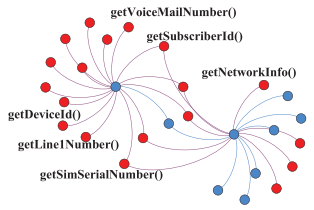

Fig. 8. The fregraph with the highest weight score.

Malware Genome Project [7] contains 1,247 samples in 36 families.

To construct the FalDroid-I dataset, approximately 15,000 malware samples are first downloaded from VirusShare [23] and uploaded to VirusTotal [32], which is a system with 53 anti-virus scanners (e.g., AVL, McAfee, and ESET-NOD32). The following two issues are found from the anti-virus scanners: 1) the family labels given by different anti-virus scanners are not always the same (e.g., *Plankton/Plangton/planktonc*); and 2) the results of the anti-virus scanners rarely reach a consensus. To address these issues, we initially construct a family label dictionary based on string-edit distance [33]. Part of the dictionary is listed in TABLE III. Then, we label the malware with the family name that is agreed by more than half of the anti-virus scanners. Finally, 8,407 malware samples in 36 families are labeled, and their information is listed in TABLE IV, where Num is the number of malware samples in each family.

The samples in the FalDroid-II dataset are provided by contagion [34] and MassVet [35] and labeled in the same manner as those in the FalDroid-I dataset. Finally, 643 malware samples in 43 families are labeled.

TABLE V lists the metrics used to evaluate FalDroid. We develop FalDroid in 8,100 lines of Java code and 900 lines of Python code. We conduct the experiments on a quad-core 3.20 GHz PC running Ubuntu 14.04(64 bit) with 16 GB RAM and 1 TB hard disk.

B. Effectiveness of Representing Common Behavior

We manually inspect the fregraph with the highest weighted score to evaluate whether the generated fregraphs can effectively represent the common malicious behaviors shared by malware samples within the same family. Fig. 8 shows the generated fregraph with the highest weighted score of 9.821, which is used by all 105 samples in the *geinimi* family. On the basis of the semantic meanings of sensitive API calls in the graph, we find that the fregraph is used to collect personal information (e.g., phone number and IMEI). Analysis of the packages that contain the fregraph reveals that the common malicious code is hidden in different packages, as listed in TABLE VI, in which the term Num denotes the number of malware samples in the *geinimi* family that contain the corresponding suspicious package. By representing their invocation patterns as fregraphs, FalDroid can effectively identify the common behaviors shared by different packages. The details of all the generated fregraph-based features in the 36 families are shared online at *https://github.com/xjtu1025/FalDroid.*

Answer to RQ 1: Fregraphs can effectively represent the common behaviors shared by malware samples within the same family..

TABLE IV
DESCRIPTIONS OF MALWARE FAMILIES

Id	Malware Family	Num	Id	Malware Family	Num
1	adwo	338	19	hongtoutou	46
2	airpush	76	20	iconosys	153
3	anserver	53	21	imlog	41
4	basebridge	303	22	jsmshider	22
5	boqx	49	23	kmin	248
6	boxer	95	24	kuguo	358
7	clicker	37	25	lovetrap	19
8	dowgin	851	26	mobiletx	81
9	droiddreamlight	101	27	pjapps	82
10	droidkungfu	736	28	plankton	896
11	droidsheep	14	29	smskey	111
12	fakeangry	16	30	smsreg	149
13	fakedoc	147	31	steek	20
14	fakeinst	1,504	32	utchi	285
15	fakeplay	43	33	waps	771
16	geinimi	105	34	youmi	113
17	gingermaster	385	35	yzhc	49
18	golddream	80	36	zitmo	30

TABLE V
DESCRIPTIONS OF THE USED METRICS

Term	Abbr	Definition
True Positive	TP	#malware in family f are correctly classified into family f.
True Negative	TN	#malware not in family f are correctly not classified into family f.
False Negative	FN	#malware in family f are incorrectly not classified into family f.
False Positive	FP	#malware 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
Classification Accuracy		percentage of malware which are correctly classified into their corresponding families.

C. Accuracy of Malware Familial Classification

1) Performance With Four Different Classifiers: We use FalDroid-I dataset to evaluate the familial classification performance of FalDroid equipped with four different classifiers, namely, support vector machine (SVM; linear kernel) [36], Decision Tree (C4.5) [37], k-nearest neighbor (k-NN; k = 1) [38] and Random Forest (tree num = 100) [39]. The experiment is conducted using 10-fold cross-validation.

TABLE VI

PACKAGES CONTAINING THE COMMON MALICIOUS BEHAVIORS OF THE GEINIMI FAMILY

Package Name	Num
com/geinimi/c/k	54
com/geinimi/c/f	18
com/dseffects/MonkeyJump2/jump2/e/k	6
com/xlabtech/MonsterTruckRally/rally/e/k	4
signcomsexgirl1/mm/model/e/k	3
com/myanimal/d/k	3
jp/co/kaku/spi/fs1006/Paid/activity/e/k	1
aC	1
com/rdwl/qwkj/malaup/android/action/welcome/d/k	1
com/xlabtech/HardcoreDirtBike/bike/e/k	1
redrabbit/CityDefense/application/e/k	1
com/ericliu/cg5/main/e/k	1
com/littlekillerz/legendsarcana/screen/d/k	1
sex/sexy/model13/e/k	1
com/gamevil/bs2010/launcher/e/k	1
chaire1/mm/model/e/k	1
cmp/netsentry/list/e/k	1
com/wuzla/game/ScooterHeroPaid/paid/e/k	1
com/masshabit/squibble/free/activity/e/k	1
com/apostek/SlotMachine/paid/game/e/k	1
cmp/LocalService/service/e/k	1
com/computertimeco/android/alienspresident/president/e/k	1
com/swampy/sexpos/pos/e/k	1
Total	**105**

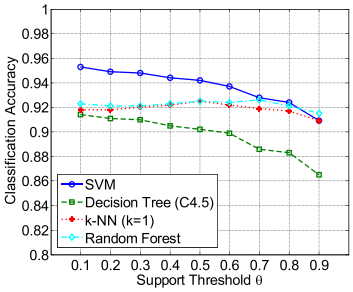

Fig. 9. Classification performance of FalDroid for four different classifiers under different support thresholds θ.

Fig. 9 shows the classification accuracies of the four classifiers with different θ ranging from 0.1 to 0.9. We can draw the following three conclusions from Fig. 9:

(i) All classifiers obtain an acceptable result (i.e., higher than 86%).

(ii) SVM outperforms other classifiers. Its accuracy is 0.953 when $\theta = 0.1$.

(iii) The performance of SVM decreases as θ increases, particularly when θ exceeds 0.5. As shown in Fig. 10, the number of fregraph-based features decreases with the increase in θ. Specifically, no fregraphs are found for some families (e.g., *airpush* and *boqx*) when $\theta > 0.5$, thereby resulting in low accuracy.

Moreover, Fig. 10 presents that the small number of fregraph-based features results in the small run-time overhead of feature construction for a new sample. The accuracy of SVM decreases by 1.1% when $\theta = 0.5$, whereas both the number of features and the run-time overhead of feature construction decrease by 82% when $\theta = 0.1$. Thus, we select SVM as our classifier and set $\theta = 0.5$ in latter experiments. Fig. 11 illustrates the increase in the number of fregraph-based features when each malware family is included. On average, 21 new fregraph-based features are added per family.

TABLE VII shows the classification results when $\theta = 0.5$. Most families have TPR higher than 0.9. Specifically,

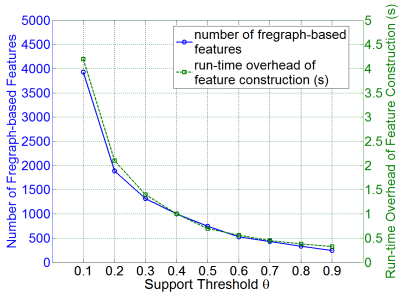

Fig. 10. Number of fregraph-based features and corresponding run-time overhead of feature construction under different support thresholds θ.

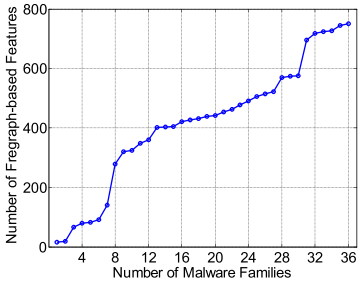

Fig. 11. Number of fregraph-based features by adding families when $\theta = 0.5$.

12 families achieve TPR equal to 1 with FPR equal to 0, indicating that all of their samples are accurately classified and no other samples are inaccurately classified into such families. However, FalDroid obtains poor results for some families, such as *boqx* and *anserver*. The *boqx* family contains only two unique fregraph-based features. All the samples in the *anserver* family are classified into the *basebridge* family because their samples evolved from samples in the *basebridge* family [7]. In summary, FalDroid performs effectively for most families.

2) Classification Performance on Different Datasets: We also evaluate FalDroid using four different datasets. TABLE VIII shows the classification performance of FalDroid for the four datasets when $\theta = 0.5$.

FalDroid can successfully classify 95.3% of the samples in the Drebin dataset [31] into their families. Its classification accuracy is 0.972 for the Genome Project dataset [7]. Misclassifications are attributed to two main reasons: First, few fregraphs are generated for some families, such as *boxer* in [31], thus causing performance to deteriorate. Second, some families, such as *DroidKungFu2* and *DroidKungFu3* in [7], exhibit similar malicious behavior. Therefore, malware samples in these families have similar fregraph-based feature vectors. The classification accuracies of FalDroid for our constructed databases are 0.942 and 0.919. In summary, FalDroid can achieve acceptable classification performance for all four datasets.

3) Comparison With State-of-the-Art Approaches: We compare FalDroid with seven state-of-the-art approaches, including, Dendroid [40], Apposcopy [8], DroidSIFT [41], MudFlow [42], TriFlow [43], DroidLegacy [10], and Astroid [44]. These approaches are briefly described below:

- Dendroid automatically classifies malware and analyzes families on the basis of code structures [40].
- Apposcopy extracts the data-flow and control-flow properties of an app to identify its family [8].

TABLE VII
CLASSIFICATION PERFORMANCE FOR 36 FAMILIES WITH SVM WHEN $\theta = 0.5$

Malware Family	TPR	FPR	p	r	F	AUC	Malware Family	TPR	FPR	p	r	F	AUC
adwo	0.896	0.003	0.921	0.896	0.909	0.947	hongtoutou	1	0	1	1	1	1
airpush	0.724	0.002	0.764	0.724	0.743	0.861	iconosys	1	0	0.975	1	0.987	1
anserver	0	0	0	0	0	0.5	imlog	1	0	1	1	1	1
basebridge	0.927	0.008	0.822	0.927	0.871	0.96	jsmshider	1	0	0.957	1	0.978	1
boqx	0.531	0.001	0.813	0.531	0.642	0.765	kmin	0.992	0	0.988	0.992	0.99	0.996
boxer	0.853	0.003	0.771	0.853	0.81	0.925	kuguo	0.936	0.004	0.905	0.936	0.92	0.966
clicker	1	0	0.974	1	0.987	1	lovetrap	1	0	0.864	1	0.927	1
dowgin	0.947	0.006	0.944	0.947	0.945	0.97	mobiletx	1	0	1	1	1	1
droiddreamlight	0.881	0.001	0.947	0.881	0.913	0.94	pjapps	0.915	0	0.962	0.915	0.938	0.957
droidkungfu	0.958	0.009	0.91	0.958	0.933	0.974	plankton	0.99	0.001	0.99	0.99	0.99	0.994
droidsheep	1	0	1	1	1	1	smskey	0.991	0	0.982	0.991	0.987	0.995
fakeangry	0.5	0	1	0.5	0.667	0.75	smsreg	0.832	0.002	0.905	0.832	0.867	0.915
fakedoc	0.993	0	0.986	0.993	0.99	0.996	steek	1	0	1	1	1	1
fakeinst	0.98	0.005	0.978	0.98	0.979	0.988	utchi	1	0	1	1	1	1
fakeplay	0.884	0	0.95	0.884	0.916	0.942	waps	0.929	0.007	0.931	0.929	0.93	0.961
geinimi	1	0	0.991	1	0.995	1	youmi	0.761	0.003	0.782	0.761	0.771	0.879
gingermaster	0.914	0.004	0.919	0.914	0.917	0.955	yzhc	1	0	0.961	1	0.98	1
golddream	0.938	0	0.974	0.938	0.955	0.969	zitmo	0.931	0.001	0.824	0.933	0.875	0.966
Avg.	**0.942**	**0.004**	**0.937**	**0.942**	**0.939**	**0.969**							

TABLE VIII
CLASSIFICATION PERFORMANCE ON FOUR DATASETS

Dataset	Classification Accuracy
Genome Project dataset [7]	0.972
Drebin dataset [31]	0.953
FalDroid-I dataset	0.942
FalDroid-II dataset	0.919

TABLE IX
CLASSIFICATION ACCURACIES OF FALDROID AND SEVEN STATE-OF-THE-ART APPROACHES ON GENOME PROJECT DATASET [7]

Baseline Approach	Classification Accuracy
Dendroid [40]	0.942
Apposcopy [8]	0.900
DroidSIFT [41]	0.930
MudFlow [42], TriFlow [43]	0.881
DroidLegacy [10]	0.929
Astroid [44]	0.938
FalDroid	**0.972**

- DroidSIFT is a semantic-based approach that classifies malware via API dependency graphs [41].
- MudFlow [42] and TriFlow [43] analyze malware samples on the basis of the source-and-sink method pairs extracted by FlowDroid [45].
- DroidLegacy partitions the app code into loosely coupled modules and identifies the malicious module of each piggybacked malware family [10].
- Astroid automatically synthesizes a maximally suspicious common subgraph of each malware family as a signature to perform familial classification [44].

Given that most of these systems are not publicly available and re-implementing the same systems with identical parameters is difficult, we apply FalDroid to the same Genome Project dataset [7], which has been used to evaluate these systems in their works. TABLE IX lists the results of comparison. FalDroid outperforms other seven approaches on the same dataset for malware familial classification.

Among these approaches, DroidSIFT is the most related to FalDroid. Two major differences are found between these two approaches. First, DroidSIFT requires a set of graphs extracted from benign apps to remove the common graphs extracted from malware, whereas FalDroid uses a clustering-based approach to mine fregraphs only from malware to identify their commonalities. Ensuring the completeness of the benign graph set is difficult for DroidSIFT. Moreover, DroidSIFT calculates similarities among graphs using an improved graph-edit distance (GED), whereas FalDroid employs a novel weighted-sensitive-API-call-based approach, which is more robust and effective than GED in detecting homogeneous app behaviors and tolerating minor differences in implementation (see Section IV-F for details).

4) Answer to RQ 2: FalDroid can classify malware samples into their families with high accuracy and can outperform state-of-the-art approaches on a widely used benchmark dataset [7].

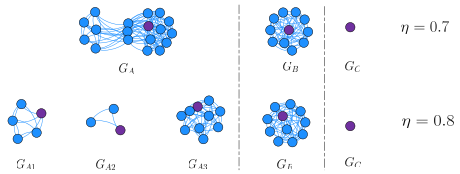

Fig. 12. MSGs of *zitmo* with $\eta = 0.7$ and $\eta = 0.8$.

D. Effectiveness of Representative Malware Sample Selection

To evaluate the capability of FalDroid in selecting representative malware samples, we first analyze the MSGs of the *zitmo* family as an example. We then apply our approach to the 36 malware families in our FalDroid-I dataset.

Fig. 12 illustrates the MSGs of *zitmo* with different similarity thresholds $\eta = 0.7$ and $\eta = 0.8$. In this figure, each node denotes a malware sample, and purple nodes denote selected representative samples. TABLE X lists the differences in representative samples after manual analysis. Moreover, these malware samples are in the same family, and their receivers and malicious behaviors exhibit minor differences. For example, samples in G_A contain three receivers, whereas samples in G_B and G_C contain only one receiver, thereby resulting in three groups when $\eta = 0.7$. Moreover, Group G_A is divided into three subgroups, namely, G_{A1}, G_{A2}, and G_{A3}, when $\eta = 0.8$. The malicious behaviors of the samples in the three subgroups also exhibit minor differences. For example, samples in G_{A1} can read the phone state compared with the samples in G_{A2} and G_{A3}. Therefore, our approach provides an

TABLE X

DIFFERENCES OF REPRESENTATIVE MALWARE SAMPLES IN *zitmo* FAMILY

Group	Activity	Receivers	Malicious Behaviors
G_{A1}	com.guard.smart.MainActivity	SmsReceiver, TimerReceiver, onBootReceiver	receive messages, boot, read phone state
G_{A2}	com.security.service.MainActivity	SmsReceiver, ActionReceiver, RebootReceiver	send/receive messages
G_{A3}	com.security.service.MainActivity	SmsReceiver, ActionReceiver, RebootReceiver	send/receive/edit messages
G_B	com.android.security.MainActivity	SecurityReceiver	send/receive/edit messages, modify/delete SD card contents, read phone state, intercept outgoing calls
G_C	com.systemsecurity6.gms.Activation	SmsReceiver	receive messages, full internet access, read phone state

TABLE XI

THE SELECTION OF REPRESENTATIVE MALWARE SAMPLES WITH DIFFERENT SIMILARITY THRESHOLD VALUES η

Malware Famlily (#samples)	#groups $\eta=0.5$	$\eta=0.6$	$\eta=0.7$	$\eta=0.8$	Malware Famlily (#samples)	#groups $\eta=0.5$	$\eta=0.6$	$\eta=0.7$	$\eta=0.8$
adwo (338)	3	29	74	120	hongtoutou (46)	3	6	9	17
airpush (76)	23	37	41	51	iconosys (153)	1	2	5	8
anserver (53)	1	1	1	1	imlog (41)	3	3	3	3
basebridge (303)	29	38	52	63	jsmshider (22)	3	3	4	4
boqx (49)	21	28	35	41	kmin (248)	3	4	4	5
boxer (95)	6	6	6	6	kuguo (358)	11	25	51	109
clicker (37)	1	1	2	4	lovetrap(19)	1	2	2	3
dowgin (851)	45	66	92	125	mobiletx (81)	2	2	2	2
droiddreamlight (101)	6	12	16	23	pjapps (82)	12	17	21	27
droidkungfu (736)	30	48	82	120	plankton (896)	7	18	40	83
droidsheep (14)	1	1	1	1	smskey (111)	3	19	32	41
fakeangry (16)	9	9	9	9	smsreg (149)	24	38	46	60
fakedoc (147)	2	2	3	6	steek (20)	1	1	1	1
fakeinst (1504)	10	14	17	23	utchi (285)	1	1	1	1
fakeplay (43)	3	4	4	5	waps (771)	27	76	146	243
geinimi (105)	1	1	1	1	youmi (113)	19	36	54	75
gingermaster (385)	27	81	149	189	yzhc (49)	1	2	4	5
golddream (80)	4	6	9	12	zitmo (30)	3	3	3	5

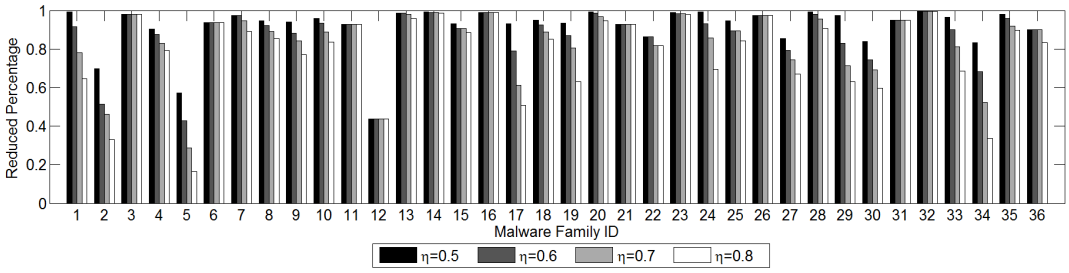

Fig. 13. Reduced percentages of samples to be analyzed in each family with different similarity threshold values η.

optional app similarity threshold for analysts when selecting the representative malware samples in each family. A high η indicates that high numbers of representative samples are selected for analysis.

TABLE XI shows the number of groups generated in all the 36 families with different similarity threshold η. The group number for each family increases or remains unchanged when η increases because it is more difficult for two nodes to have one edge. We can draw the following three conclusions.

(i) Group number is not related with family size. For example, *utchi* (285 samples) has only one group, whereas *boqx* (49 samples) has 41 groups when $\eta = 0.8$.

(ii) The group numbers of several malware families remain unchanged with the increase in η (e.g., the group numbers of *geinimi* and *utchi* are always one). In other words, the generated fregraph-based features indicate that malware samples in such families are highly similar.

(iii) The malware families with relatively small change in the group number exhibit better classification performance than families with a considerable increase in group numbers. For example, the TPR of *geinimi*, *utchi*, and *imlog* can achieve 1, whereas that of *airpush* and *boqx* is lower than 0.75. This phenomenon can be attributed to the relatively small change in group number, which indicates that samples in families, such as *geinimi*, exhibit higher similarities than those in *airpush*.

One representative malware sample is selected for each generated group. We use *reduced percentage* to denote the percentage of malware samples in which its inspection can be deferred because of the representative malware sample selected by FalDroid. $reduced\ percentage = 1 - \frac{groupNum(f)}{falNum(f)}$, where $groupNum(f)$ denotes the number of generated $groups$ in family f. For example, analysts should only inspect the most representative sample in this group rather than all 105 samples because only one group is found in the *geinimi* family. Consequently, we can effectively decrease 104 malware samples to be analyzed. Hence, the *reduced percentage* of *geinimi* is $1 - \frac{1}{105} = 0.99$.

Fig. 13 presents the *reduced percentage*s of all the 36 families with different η values. It shows that the *reduced*

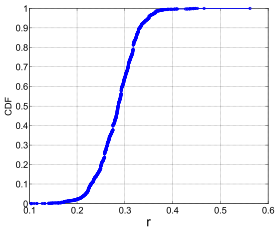

Fig. 14. CDF of the ratio of size of SARG to its FCG.

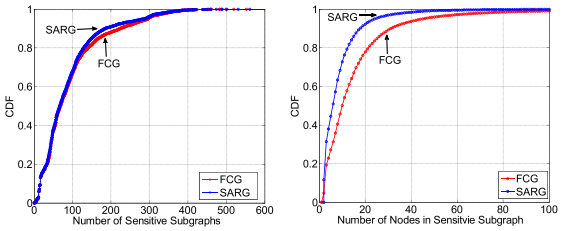

Fig. 15. CDFs of the number of sensitive subgraphs and the number of nodes in sensitive subgraphs.

percentage decreases with the increase in η. FalDroid can effectively decrease the number of malware samples to be analyzed by approximately 78% when $\eta = 0.8$ and by approximately 91.5% when $\eta = 0.5$ on average.

Answer to RQ 3: FalDroid can effectively decrease malware samples to be analyzed for each family.

E. Efficiency

1) Statistics of Generated Graphs: We use r to denote the size ratio of SARG with its corresponding FCG given that FalDroid initially generates SARG from FCG to exclude nodes without paths to sensitive nodes. Fig. 14 presents the CDF of r for all the samples in our datasets. More than 98% of r exists in the range from 0.2 to 0.4, and the average value is 0.28. Thus, the size of SARG is reduced by approximately 72% compared with the that of the original FCG.

Then, we divide the SARG into a set of sensitive subgraphs using community detection algorithms. Fig. 15 summarizes the statistics of the sensitive subgraphs generated from FCG and SARG. The left figure illustrates the CDFs of the number of sensitive subgraphs. The CDF of the generated sensitive subgraphs of SARG is close to that of FCG because the construction of SARG retains all sensitive nodes. On average, 90 sensitive subgraphs are generated for each malware, and more than 90% samples contain less than 200 sensitive subgraphs. The right figure in Fig. 15 shows the CDFs for the number of nodes in each sensitive subgraph. The sensitive subgraphs generated from SARG contain fewer nodes than those generated from FCG. On average, 10 nodes are found in the sensitive subgraph generated from SARG. Furthermore, approximately 750,000 sensitive subgraphs are found, and only 0.8% of these subgraphs have more than 50 nodes. However, 16 nodes are found in each sensitive subgraph directly generated from FCG. In addition, more than 4% subgraphs contain more than 50 nodes.

The results demonstrate that the construction of SARG can effectively reduce the complexity of graph analysis. This observation is important to the scalability of FalDroid because the run-time performance of graph matching depends on the number of sensitive subgraphs and their nodes.

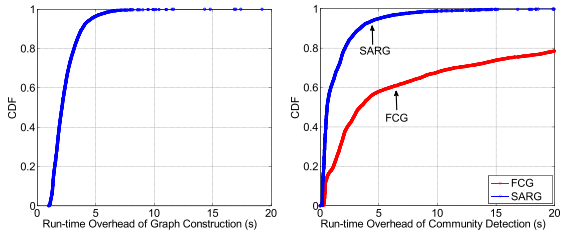

Fig. 16. CDFs of run-time overhead for graph construction and community detection.

2) Run-Time Overhead: FalDroid comprises the following main steps to analyze a new malware sample.

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

The run-time overheads of graph construction and community detection are shown in Fig. 16. An average of 2.4 sec is required to construct the graph model for a given APK file. SARG construction requires considerably less time than APK disassembly. In community detection, 1.5 sec is required to divide the graph into a set of subgraphs, whereas 16 sec is required when FCG is not simplified as an SARG. On average, 0.7 sec is required for the feature construction set to generate the feature vector of a new malware sample when $\theta = 0.5$.

The average run-time overhead of FalDroid is 4.6 sec, and 95% of the samples are processed within 10 sec. FalDroid requires considerably less time than DroidSIFT [41] and Apposcopy [8], which consume 175.8 and 275 sec, respectively, to analyze an app due to their heavyweight static code analysis. FalDroid consumes less time than DroidSIFT and Apposcopy because of the following two reasons. First, SARG is induced from the complex FCG by removing nodes without close relationships with sensitive API call nodes. Thus, graph size is reduced by 72%. The decrease in graph size effectively shortens graph analysis. Second, we use a weighted-sensitive-API-call-based graph matching approach, in which we focus on the local structure of the sensitive API call nodes rather than all the nodes in the subgraphs. Thus, our approach requires less time to complete one pair-wise graph matching compared with GED used in DroidSIFT.

3) Answer to RQ 4: The low run-time overhead allows FalDroid to work efficiently and be scalable to a large number of apps.

F. Resilience

1) Resilience to Polymorphic Variants: FalDroid performs graph matching with the proposed weighted-sensitive-API-call-based approach to compete against polymorphic variants. In this process, we evaluate the effectiveness of our graph matching approach and compare it with GED, which was widely used by existing studies [41], [46]. The GED metric

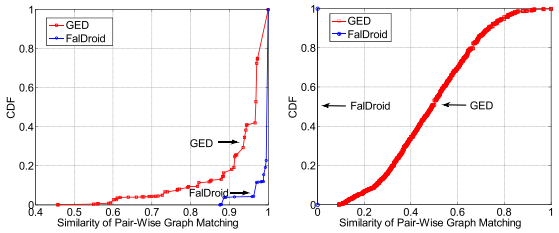

Fig. 17. Comparison between our weighted-sensitive-API-call-based graph matching approach with GED on the *similar set* and the *dissimilar set*.

depends on the selection of edit operations and the cost involved per operation (e.g., node insertion/deletion, edge insertion/deletion and node relabeling). Specifically, we ignore relabeling cost since the node label can be easily changed. We manually construct two subgraph sets.

- The *similar set*, which consists of 50 sensitive subgraphs generated from 50 different malware in the *geinimi* family. These 50 sensitive subgraphs exhibit similar malicious behaviors with minor differences.
- The *dissimilar set*, which consists of 50 sensitive subgraphs generated from one malware sample. Any two subgraphs do not contain the same sensitive nodes, indicating that they exhibit entirely different behaviors.

For the two subgraph sets, each subgraph is matched with the others. Thus, 49*49 pair-wise graph matching similarities are found. We compare the performance of our approach with that of GED for the *similar set* (illustrated in the lefthand side of Fig. 17) and the *dissimilar set* (illustrated in the righthand side of Fig. 17). For the *similar set*, all similarities computed by our approach are higher than 0.8, which is selected as the similarity threshold for clustering subgraphs. However, approximately 10% of similarities from GED are lower than 0.8. For the *dissimilar set*, all similarities computed by our approach are 0. GED similarities range from 0.1 to 1, and approximately 3% of similarities are higher than 0.8.

Our approach requires less than 1 ms to complete one pair-wise graph matching, whereas GED requires approximately 7 ms. The low run-time overhead enables our approach to be scalable for clustering thousands of subgraphs. In summary, FalDroid can better reveal homogeneous behaviors and tolerate minor differences than GED.

2) Resilience to Code Obfuscation Techniques: We initially evaluate the resilience of FalDroid to typical obfuscation techniques (e.g., renaming user-defined functions [47]–[50]) by using *Proguard* [51] to obfuscate ten apps from source codes. The results show that their similarities on graph matching are still 1. These typical obfuscation techniques do not affect the performance of FalDroid because they do not change the FCG structure.

Subsequently, we evaluate the resilience of FalDroid to control flow obfuscation techniques, which will change the FCG structure by inserting or deleting some useless methods. For this purpose, we apply FalDroid to ten apps obfuscated by the popular Android obfuscator, *DashO* [52], which can adopt control flow obfuscation techniques. Results show that the SARGs induced from FCGs will remain unchanged when the inserted or deleted method nodes have no relation with the sensitive API call nodes. By contrast, the subgraph matching results and the constructed feature vectors will be slightly affected when the inserted or deleted method nodes have invocation relations with the sensitive API call nodes.

The nested calls are a typical control flow obfuscation technique that inserts user-defined nodes when invoking sensitive API calls. Recall that subgraph sg_2 in Fig. 5 contains one additional nested call compared with sg_1. However, their similarity is still 0.99, which nearly does not affect our approach. However, when more nested call nodes are inserted, the effect on the similarity calculation would be magnified. To eliminate such cases, we can merge user-defined nodes with their parent nodes when they only have one parent node. Thus, we can reduce the effect of changing the shortest path distances among sensitive API call nodes.

Being a static analysis approach, the performance of FalDroid might be affected by advanced obfuscation techniques, such as reflection and encryption. However, we can address such limitations with the aid of existing open-source tools, such as *DroidRA* [53] and *PackerGrind* [54], [55]. Existing open-source tools address the above limitations through the following approaches:

- **Reflection:** Reflection techniques [56] can hide some edges in the call graph model by invoking functions with their corresponding names as arguments. To be resilient to reflection obfuscation techniques, we can use *DroidRA* [53], which is an open-source tool, to perform reflection analysis on our dataset through three steps. First, we conduct *DroidRA* on our dataset and obtain the analytical result. Second, we analyze the output result of *DroidRA* for each app to identify methods that use reflection techniques. Third, we add the missing edges into the corresponding FCG, where caller nodes are methods that use reflection techniques and callee nodes are reflected methods. On average, we add fifteen more edges into the FCG for each app, and only two edges contain a sensitive API call node, which barely affects the performance of our approach. Therefore, our approach can be resilient to reflection obfuscation techniques with the aid of *DroidRA*.
- **Encryption Packer:** Packers, such as *APKProtect* [57] and *Bangcle* [58], can protect apps by using encryption techniques to hide the actual Dex code. To address the limitations of packer usage, we use *PackerGrind* [54], [55], which is a novel adaptive unpacker system, to recover the actual Dex files. Then, our approach can be applied to the extracted Dex files.

3) Answer to RQ 5: FalDroid is resilient to polymorphic variants and code obfuscation techniques through the aid of existing tools.

V. Discussion

A. External Validity

Considering the difficulty of collecting Android malware samples with accurate labels, our dataset has only 8,407 malware samples from 36 families, whose labels are determined in accordance with VirusTotal results. We recognize that the results of VirusTotal may not be absolutely accurate. In future work, we plan to collaborate with anti-virus companies to

collect additional malware samples from more families for evaluation.

B. Unknown Malware Detection

It is worth noting that FalDroid aims at classifying malware into their families instead of detecting malware. To evaluate the capability of FalDroid to detect unknown malware, we apply it to address a binary classification problem. That is, given an app that is not contained in the training dataset, FalDroid will classify it as a malware or a benign one. We evaluate FalDroid for two types of unknown malware detection.

First, we regard the variants of known malware samples as type-I unknown malware samples. To evaluate the capability of FalDroid to detect type-I unknown malware samples, we construct a dataset that contains 643 malware samples and 643 randomly selected benign samples. Then, we apply FalDroid with 10-fold cross-validation to detect malware samples. The result shows that FalDroid can detect 97.8% type-I unknown malware samples with a 1.1% false positive rate (i.e., only seven benign samples are inaccurately detected as malicious).

Second, we regard recent malware samples, which are created several years after older known malware samples, as type-II unknown malware samples. To evaluate the capability of FalDroid to detect type-II unknown malware samples, we first construct a training dataset and a testing dataset. The training dataset consists of 8,407 old malware samples and 8,407 randomly selected benign samples. The testing dataset consists of 643 recent malware samples and 643 randomly selected benign samples. Even though the malware samples used for testing are known to be malicious, none of them belongs to the malware families in the training dataset. Therefore, the 643 malware samples are regarded as type-II unknown malware with respect to the malware samples used for training. The result shows that our approach can detect 75% type-II unknown malware samples with a 2% false positive rate. Our approach failed to report 25% type-II unknown malware samples because of the concept drift problem [59]. Recent malware samples adopt some new attack measures, such as encrypting important user files, that are different from the attack measures adopted by old malware samples, such as stealing personal information or sending premium messages by using sensitive API calls.

We also compare the capability of FalDroid to detect unknown malware with that of the most related approach DroidSIFT. For type-I unknown malware detection, DroidSIFT achieves a 97% true positive rate similar to that achieved by our approach. However, its false positive rate is approximately 5%, which is higher than that of our approach. The main reason is that it constructs a benign subgraph set to remove the common subgraphs extracted from malware. The incompleteness of the benign graph set would introduce false positives. For type-II unknown malware detection, DroidSIFT also achieves a result similar to that achieved by our approach. The concept drift problem remains a challenge for existing approaches with respect to unknown malware detection. We will enhance the capability of FalDroid to address the concept drift problem in future work.

C. Native Code

Malware can use native code [60] to access sensitive API calls, and thus the static analysis techniques for Dex/Java bytecode become unreliable. For the analysis of native code, we will use *Angr* [61], an open-source binary analysis framework, to construct the FCG of the native code. Then, we could apply our approach to the extracted FCGs to generate fregraph-based features and add them into the existing feature space. We will explore this approach in future work.

VI. Related Work

A. Familial Malware Classification

Previous studies used machine learning techniques to classify malware that targets PCs and mobile devices.

On PC platform, Kolter and Maloof [62] used n-grams of byte codes as features to generate a classifier for malware classification. Kinable and Kostakis [46] studied malware classification based on call graph clustering by representing malware samples as FCGs. Similarly, Hu *et al*. [63] developed a malware database management system that converts each malware sample into its FCG representation and then performs nearest neighbor search on the basis of this graph representation.

Compared with traditional malware that targets PCs, mobile malware are often produced by injecting malicious payloads into legitimate apps [7], [35], and they usually invoke sensitive API calls to perform malicious behaviors. Android is the major target of mobile malware. Suarez-Tangil *et al*. [40] proposed Dendroid, which automatically classifies malware and analyzes families on the basis of code structures. However, code structure could be easily obfuscated by bytecode-level transformation [56]. Yang *et al*. [14] proposed DroidMiner, which formalizes a two-level behavioral graph model and extracts sensitive paths to denote malicious behavioral patterns for malware classification. Sensitive paths may appear in the legitimate part and malicious components, thereby causing high false positive rates. The most related work to our approach is DroidSIFT [41], which classifies Android malware via dependency graphs. DroidSIFT relies on a set of benign subgraphs to remove common subgraphs in malware. However, ensuring the completeness of the benign subgraph set is difficult.

B. Graph-Based Program Analysis

Many recent studies on Android detection have leveraged graph analysis such as program dependence graph (PDG) [64], [65], control dependence graph (CDG) [66], [67], function call graph (FCG) [41], [68], and user interface (UI) graph [35], [69], [70]. They are structural representations that are less susceptible to instruction-level obfuscations commonly employed by malware authors to evade anti-virus scanners.

Crussell *et al*. [64] proposed DNADroid to detect cloned apps by comparing PDGs among functions in candidate apps.

Chen *et al*. [67] used the geometry characteristics (centroid) of CDGs to measure similarity among app functions. Gascon *et al*. [68] proposed a malware detection approach based on the embedding of FCGs with an explicit feature map. Chen *et al*. [35] proposed MassVet, which models the UIs of apps as a directed graph wherein each node is a view and each edge describes the navigation relations among them. With the similar view structures in different apps, MassVet can effectively identify repackaged apps. UI graphs are unsuitable for familial malware classification because malware samples in the same family usually have entirely different UIs. Although PDG and CDG provide more information than FCG, their extraction from apps and analysis are time consuming and require considerable computational resources.

C. Differences From Previous Version

This work is an extension of a previous work that we had presented as a conference paper [71]. We have added a considerable amount of new material to the present work. First, we implement and evaluate four widely used community detection algorithms to divide SARG into a set of subgraphs. Moreover, we analyze modularity Q values using *infomap* algorithm [17].

Second, we redesign the calculation of the weighted score of each fregraph-based feature. Specifically, we consider the distribution probability and the weight of a fregraph-based feature to measure its significance for malware familial analysis.

Third, we equip FalDroid with the new capability to select representative malware samples for each family by proposing MSG to characterize the relationships among malware samples in the same family. With this procedure, FalDroid allows security analysts to focus on representative malware samples, thereby decreasing the analytical workload and accelerating malware analysis.

Fourth, we conduct several evaluations on FalDroid. We initially extend our FalDroid-I dataset with 1,842 new malware samples and construct the FalDroid-II dataset with 643 recent malware samples. Then, we manually inspect the fregraph with the highest weighted score to determine whether the fregraph can represent the common malicious behavior shared by malware samples in the same family. We also evaluate FalDroid using a dataset from Drebin [31]. Furthermore, we apply FalDroid to the selection of representative samples from 36 families and scrutinize the generated groups of the *zitmo* family. We also evaluate the advantages of generating SARG from its FCG in terms of graph size and run-time overhead.

VII. Conclusion and Future Work

We propose the use of fregraphs to depict the common features shared by malware samples within the same family. Moreover, we design FalDroid, a novel system that can automatically classify Android malware samples with high accuracy and effectively accelerate malware analysis by recommending representative malware samples for scrutiny. FalDroid is more effective and efficient than state-of-the-art approaches. It provides considerable information for identifying and inspecting malware and raises the level for malware to evade analysis.

In future work, we will improve FalDroid by leveraging the dynamic analysis approach, in which we can design an enhanced graph model with data-flow information to involve considerable semantic information and manage advanced obfuscation techniques.

References

[1] IDC. (2016). *Smartphone OS Market Share, 2016 Q3*. [Online]. Available: https://goo.gl/0m73WP

[2] G. Kelly. (2014). *Report: 97% Of Mobile Malware is on Android. This is the Easy Way you Stay Safe*. [Online]. Available: http://goo.gl/MYDBKC

[3] Qihoo. (2016). *Report of Smartphone Security in China, 2016 Q3*. [Online]. Available: https://goo.gl/V9Vh1u

[4] Y. Zhang *et al.*, "Vetting undesirable behaviors in Android apps with permission use analysis," in *Proc. CCS*, 2013, pp. 611–622.

[5] K. Tam, S. J. Khan, A. Fattori, and L. Cavallaro, "CopperDroid: Automatic reconstruction of Android malware behaviors," in *Proc. NDSS*, 2015, pp. 1–15.

[6] J. Huang, X. Zhang, L. Tan, P. Wang, and B. Liang, "AsDroid: Detecting stealthy behaviors in Android applications by user interface and program behavior contradiction," in *Proc. ICSE*, 2014, pp. 1036–1046.

[7] Y. Zhou and X. Jiang, "Dissecting Android malware: Characterization and evolution," in *Proc. IEEE S&P*, May 2012, pp. 95–109.

[8] Y. Feng, S. Anand, I. Dillig, and A. Aiken, "Apposcopy: Semantics-based detection of Android malware through static analysis," in *Proc. FSE*, 2014, pp. 576–587.

[9] W. Zhou, Y. Zhou, M. Grace, X. Jiang, and S. Zou, "Fast, scalable detection of 'Piggybacked' mobile applications," in *Proc. CODASPY*, 2013, pp. 185–196.

[10] L. Deshotels, V. Notani, and A. Lakhotia, "Droidlegacy: Automated familial classification of Android malware," in *Proc. PPREW*, 2014, p. 3.

[11] 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, pp. 71–82.

[12] M. Fan, J. Liu, W. Wang, H. Li, Z. Tian, and T. Liu, "DAPASA: Detecting Android piggybacked apps through sensitive subgraph analysis," *IEEE Trans. Inf. Forensics Security*, vol. 12, no. 8, pp. 1772–1785, Aug. 2017.

[13] Y.-D. Lin, Y.-C. Lai, C.-H. Chen, and H.-C. Tsai, "Identifying Android malicious repackaged applications by thread-grained system call sequences," *Comput. Secur.*, vol. 39, pp. 340–350, Nov. 2013.

[14] 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 *Proc. ESORICS*, 2014, pp. 163–182.

[15] K. Chen *et al.*, "Contextual policy enforcement in Android applications with permission event graphs," in *Proc. NDSS*, 2013, p. 234.

[16] M. Fredrikson, S. Jha, M. Christodorescu, R. Sailer, and X. Yan, "Synthesizing near-optimal malware specifications from suspicious behaviors," in *Proc. IEEE S&P*, May 2010, pp. 45–60.

[17] M. Rosvall and C. T. Bergstrom, "Maps of random walks on complex networks reveal community structure," *Proc. Nat. Acad. Sci. USA*, vol. 105, no. 4, pp. 1118–1123, 2008.

[18] A. Clauset, M. E. Newman, and C. Moore, "Finding community structure in very large networks," *Phys. Rev. E, Stat. Phys. Plasmas Fluids Relat. Interdiscip. Top.*, vol. 70, no. 6, p. 066111, 2004.

[19] U. N. Raghavan, R. Albert, and S. Kumara, "Near linear time algorithm to detect community structures in large-scale networks," *Phys. Rev. E, Stat. Phys. Plasmas Fluids Relat. Interdiscip. Top.*, vol. 76, no. 3, p. 036106, 2007.

[20] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre, "Fast unfolding of communities in large networks," *J. Stat. Mech., Theory Experim.*, vol. 2008, no. 10, p. P10008, Oct. 2008.

[21] (2016). *Apktool: A Tool for Reverse Engineering Android Apk Files*. [Online]. Available: https://ibotpeaches.github.io/Apktool/

[22] S. Rasthofer, S. Arzt, and E. Bodden, "A machine-learning approach for classifying and categorizing Android sources and sinks," in *Proc. NDSS*, 2014, pp. 1–15.

[23] (2016). *Virusshare*. [Online]. Available: http://virusshare.com/

[24] H. Wu, R. W. P. Luk, K. Wong, and K. Kwok, "Interpreting TF-IDF term weights as making relevance decisions," *ACM Trans. Inf. Syst.*, vol. 26, no. 3, Jun. 2008, Art. no. 13.
[25] J. R. Ullmann, "An algorithm for subgraph isomorphism," *J. ACM*, vol. 23, no. 1, pp. 31–42, 1976.
[26] M. E. J. Newman and M. Girvan, "Finding and evaluating community structure in networks," *Phys. Rev. E, Stat. Phys. Plasmas Fluids Relat. Interdiscip. Top.*, vol. 69, no. 2, p. 026113, 2004.
[27] G. Concas, C. Monni, M. Orru, and R. Tonelli, "A study of the community structure of a complex software network," in *Proc. WETSoM*, May 2013, pp. 14–20.
[28] Y. Qu *et al.*, "Exploring community structure of software call graph and its applications in class cohesion measurement," *J. Syst. Softw.*, vol. 108, pp. 193–210, Oct. 2015.
[29] (2016). *High-Productivity Software for Complex Networks*. [Online]. Available: https://networkx.github.io/
[30] K. Henderson *et al.*, "RoLX: Structural role extraction & mining in large graphs," in *Proc. KDD*, 2012, pp. 1231–1239.
[31] 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, pp. 23–26.
[32] (2016). *VirusTotal*. [Online]. Available: https://www.virustotal.com
[33] E. S. Ristad and P. N. Yianilos, "Learning string-edit distance," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 20, no. 5, pp. 522–532, May 1998.
[34] (2017). *Mobile Malware Mini Dump*. [Online]. Available: http://contagiominidump.blogspot.hk/
[35] K. Chen *et al.*, "Finding unknown malice in 10 seconds: Mass vetting for new threats at the Google-play scale," in *Proc. USENIX SEC*, 2015, pp. 1–17.
[36] C.-C. Chang and C.-J. Lin, "LIBSVM: A library for support vector machines," *ACM Trans. Intell. Syst. Technol.*, vol. 2, no. 3, p. 27, 2011.
[37] S. L. Salzberg, "C4.5: Programs for machine learning by J. Ross Quinlan. Morgan Kaufmann Publishers, Inc., 1993," *Mach. Learn.*, vol. 16, no. 3, pp. 235–240, 1994.
[38] D. W. Aha, D. Kibler, and M. K. Albert, "Instance-based learning algorithms," *Mach. Learn.*, vol. 6, no. 1, pp. 37–66, 1991.
[39] L. Breiman, "Random forests," *Mach. Learn.*, vol. 45, no. 1, pp. 5–32, 2001.
[40] 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," *Exp. Syst. Appl.*, vol. 41, no. 4, pp. 1104–1117, 2014.
[41] M. Zhang, Y. Duan, H. Yin, and Z. Zhao, "Semantics-aware Android malware classification using weighted contextual api dependency graphs," in *Proc. CCS*, 2014, pp. 1105–1116.
[42] V. Avdiienko *et al.*, "Mining apps for abnormal usage of sensitive data," in *Proc. ICSE*, May 2015, pp. 426–436.
[43] O. Mirzaei, G. Suarez-Tangil, J. Tapiador, and J. M. de Fuentes, "Triflow: Triaging Android applications using speculative information flows," in *Proc. AsiaCCS*, 2017, pp. 640–651.
[44] Y. Feng, O. Bastani, R. Martins, I. Dillig, and S. Anand, "Automated synthesis of semantic malware signatures using maximum satisfiability," in *Proc. NDSS*, 2017, pp. 1–15.
[45] S. Arzt *et al.*, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps," *ACM SIGPLAN Notices*, vol. 49, no. 6, pp. 259–269, 2014.
[46] J. Kinable and O. Kostakis, "Malware classification based on call graph clustering," *J. Comput. Virol.*, vol. 7, no. 4, pp. 233–245, 2011.
[47] Z. Tian, T. Liu, Q. Zheng, M. Fan, E. Zhuang, and Z. Yang, "Exploiting thread-related system calls for plagiarism detection of multithreaded programs," *J. Syst. Softw.*, vol. 119, pp. 136–148, Sep. 2016.
[48] 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 Trans. Softw. Eng.*, vol. 41, no. 12, pp. 1217–1235, Dec. 2015.
[49] Z. Tian, T. Liu, Q. Zheng, E. Zhuang, M. Fan, and Z. Yang, "Reviving sequential program birthmarking for multithreaded software plagiarism detection," *IEEE Trans. Softw. Eng.*, to be published, doi: 10.1109/TSE.2017.2688383.
[50] H. Wang, T. Liu, X. Guan, C. Shen, Q. Zheng, and Z. Yang, "Dependence guided symbolic execution," *IEEE Trans. Softw. Eng.*, vol. 43, no. 3, pp. 252–271, Mar. 2017.
[51] (2016). *Proguard*. [Online]. Available: http://proguard.sourceforge.net/
[52] (2017). *Dasho*. [Online]. Available: https://www.preemptive.com/products/dasho/overview
[53] L. Li, T. Bissyandé, D. Octeau, and J. Klein, "Droidra: Taming reflection to support whole-program analysis of Android apps," in *Proc. ISSTA*, 2016, pp. 318–329.
[54] L. Xue, X. Luo, L. Yu, S. Wang, and D. Wu, "Adaptive unpacking of Android apps," in *Proc. ICSE*, May 2017, pp. 358–369.
[55] Y. Zhang, X. Luo, and H. Yin, "Dexhunter: Toward extracting hidden code from packed Android applications," in *Proc. ESORICS*, 2015, pp. 293–311.
[56] V. Rastogi, Y. Chen, and X. Jiang, "Catch me if you can: Evaluating Android anti-malware against transformation attacks," *IEEE Trans. Inf. Forensics Security*, vol. 9, no. 1, pp. 99–108, Jan. 2014.
[57] (2017). *Apk Protect*. [Online]. Available: https://sourceforge.net/projects/apkprotect
[58] (2017). *Bangcle*. [Online]. Available: http://www.bangcle.com/
[59] R. Jordaney *et al.*, "Transcend: Detecting concept drift in malware classification models," in *Proc. USENIX SEC*, 2017, pp. 1–20.
[60] C. Qian, X. Luo, Y. Shao, and A. T. S. Chan, "On tracking information flows through JNI in Android applications," in *Proc. DSN*, Jun. 2014, pp. 180–191.
[61] Y. Shoshitaishvili *et al.*, "SOK: (State of) the art of war: Offensive techniques in binary analysis," in *Proc. IEEE S&P*, May 2016, pp. 138–157.
[62] J. Z. Kolter and M. A. Maloof, "Learning to detect and classify malicious executables in the wild," *J. Mach. Learn. Res.*, vol. 7, pp. 2721–2744, Dec. 2006.
[63] X. Hu, T.-C. Chiueh, and K. G. Shin, "Large-scale malware indexing using function-call graphs," in *Proc. CCS*, 2009, pp. 611–620.
[64] J. Crussell, C. Gibler, and H. Chen, "Attack of the clones: Detecting cloned applications on Android markets," in *Proc. ESORICS*, 2012, pp. 37–54.
[65] J. Crussell, C. Gibler, and H. Chen, "AnDarwin: Scalable detection of semantically similar Android applications," in *Proc. ESORICS*, 2013, pp. 182–199.
[66] X. Sun, Y. Zhongyang, Z. Xin, B. Mao, and L. Xie, "Detecting code reuse in Android applications using component-based control flow graph," in *Proc. IFIP SEC*, 2014, pp. 142–155.
[67] K. Chen, P. Liu, and Y. Zhang, "Achieving accuracy and scalability simultaneously in detecting application clones on Android markets," in *Proc. ICSE*, 2014, pp. 175–186.
[68] H. Gascon, F. Yamaguchi, D. Arp, and K. Rieck, "Structural detection of Android malware using embedded call graphs," in *Proc. AiSec*, 2013, pp. 45–54.
[69] F. Zhang, H. Huang, S. Zhu, D. Wu, and P. Liu, "ViewDroid: Towards obfuscation-resilient mobile application repackaging detection," in *Proc. WiSec*, 2014, pp. 25–36.
[70] 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, pp. 56–65.
[71] M. Fan *et al.*, "Frequent subgraph based familial classification of Android malware," in *Proc. ISSRE*, Oct. 2016, pp. 24–35.

Authors' photographs and biographies not available at the time of publication.